



Universidad de Valladolid



Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en Ingeniería de Software

**Desarrollo e integración de un
sistema basado en IA, para la
consulta de información en el sector
asegurador**

Autor: Hugo Cubino Cubino



Universidad de Valladolid



Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en Ingeniería de Software

**Desarrollo e integración de un
sistema basado en IA, para la
consulta de información en el sector
asegurador**

Autor: Hugo Cubino Cubino

Tutor: Valentín Cardenoso Payo

*A Isabel, por su confianza infinita y su apoyo incondicional, que han sido mi mayor impulso en
cada paso del camino.*

*A mi familia, por estar siempre, por apoyar cada elección y caminar conmigo en cada etapa de
este recorrido.*

Agradecimientos

Han sido muchas las personas que han contribuido a que este proyecto saliera adelante. En primer lugar, quiero expresar mi más sincero agradecimiento a mi tutor, Valentín, por su dedicación, paciencia y orientación constante. Sus revisiones y correcciones han sido fundamentales para que tanto la aplicación como la memoria avanzaran en la dirección adecuada.

Agradezco también a todos los compañeros y compañeras que han formado parte de este proceso, compartiendo ideas y apoyos en los momentos de mayor exigencia. De manera especial, quiero mencionar a Julia, Fran y Diego, con quienes he compartido no solo horas de estudio, sino también momentos de compañía y confianza que han hecho más fácil y agradable el camino hasta aquí. Pero, sobre todo, quiero destacar a Miguel, mi compañero de trabajos durante toda la carrera, quien ha sido mucho más que un colega: ha sido un amigo, un apoyo constante y una fuente de aprendizaje y motivación. Su colaboración, compromiso y amistad han sido esenciales para mí, tanto en este proyecto como en todo mi recorrido universitario. Sin él, esta etapa no habría sido la misma.

A Isabel, mi pareja, le dedico el reconocimiento más profundo y sincero. Su apoyo ha sido una parte esencial de este trabajo, no solo por estar presente, sino por cómo lo ha estado: con cariño, con paciencia y con una confianza inquebrantable en mí. Ha sabido acompañarme incluso en los días más difíciles, animándome cuando flaqueaban las fuerzas y celebrando cada pequeño avance como si fuera suyo. Gracias a su forma de estar, he podido avanzar con tranquilidad y seguridad, sabiendo que siempre tenía a alguien creyendo en mí de forma genuina. Este logro también le pertenece, porque su manera de cuidar y motivar ha estado detrás de cada paso que he dado.

A mis padres y a mi hermana, les agradezco su apoyo incondicional y su presencia continua a lo largo de todos estos años. Han sido mi refugio y mi sustento, estando siempre ahí para animarme, escucharme y acompañarme en cada momento, tanto en los buenos como en los difíciles. Su confianza y su amor han sido el pilar sobre el que he construido este recorrido, y sin duda, todo lo que he logrado hasta ahora lleva también su nombre.

Por último, quiero reconocer el compromiso y la lealtad de mis amigos Pedro, Alex y Hugo. Su apoyo, aunque a menudo a distancia, ha sido inquebrantable. Han estado presentes en cada etapa, brindándome palabras de aliento y demostrando una amistad que trasciende el tiempo y las circunstancias. Su firmeza y cercanía han sido un sostén fundamental para mantener la

motivación y la confianza en mí mismo durante todo este proceso.

Gracias a todos por acompañarme en este viaje. Este proyecto también os pertenece, porque cada paso lo he dado sabiendo que contaba con vosotros.

Resumen

Motivación: Durante unas prácticas en una entidad tecnológica del sector asegurador, se evidenció el retraso digital en la gestión documental, caracterizada por procesos manuales, lentos y propensos a errores, lo que motivó la búsqueda de soluciones inteligentes que mejoren la eficiencia y precisión en la consulta de información.

Objetivo: Diseñar, desarrollar e integrar un sistema basado en inteligencia artificial, utilizando la técnica de Generación Aumentada por Recuperación (RAG), que permita realizar consultas precisas y en lenguaje natural sobre documentación técnica aseguradora.

Tareas: El proyecto ha implicado la planificación y análisis del problema, la investigación de tecnologías RAG, el diseño arquitectónico del sistema en microservicios, el desarrollo de una API REST con backend FastAPI y frontend React, la integración de modelos de lenguaje y bases vectoriales (FAISS), y la implementación de funcionalidades de consulta y gestión documental con pruebas comparativas.

Resultados: El sistema desarrollado permite realizar consultas semánticas rápidas y fiables, demostrando un rendimiento superior en precisión, trazabilidad y tiempo de respuesta frente a métodos tradicionales de búsqueda.

Conclusión: La solución propuesta representa un avance significativo en la digitalización del sector asegurador, aportando una herramienta escalable, modular y tecnológicamente actual que mejora el acceso a la información y sienta las bases para futuras mejoras basadas en inteligencia artificial.

Palabras clave: embeddings, inteligencia artificial, RAG, recuperación semántica, sector asegurador, sistemas conversacionales

Abstract

Motivation: During an internship at a technology provider for insurance companies, a significant digital gap in document management was observed, marked by slow, manual processes prone to human error. This motivated the development of intelligent solutions to improve the efficiency and accuracy of information retrieval.

Objective: To design, develop, and integrate an artificial intelligence-based system using Retrieval-Augmented Generation (RAG) techniques to enable precise, natural language queries on insurance-related technical documentation.

Tasks: The project involved problem analysis, research on RAG-related technologies, architectural design using microservices, backend development with FastAPI and a React-based

frontend, integration of language models and vector databases (FAISS), and the implementation of document management and query features supported by rigorous performance testing.

Results: The developed system enables fast, semantically accurate information retrieval and shows clear improvements in accuracy, traceability, and response time compared to traditional search methods.

Conclusion: The proposed solution contributes to the digital transformation of the insurance sector by providing a scalable, modular, and state-of-the-art tool that enhances information access and sets a solid foundation for future AI-driven improvements.

Keywords:

artificial intelligence, conversational systems, embeddings, insurance sector, RAG, semantic retrieval

Índice general

Índice de cuadros	VII
Índice de figuras	IX
1. Introducción	1
1.1. Introducción	1
1.2. Motivación	2
2. Objetivos y Alcance	3
2.1. Objetivos	3
2.1.1. Objetivo principal	3
2.1.2. Objetivos secundarios	3
2.2. Tareas a realizar	3
3. Metodología	5
3.1. Estrategia metodológica	5
3.2. Fases y planificación temporal	5
3.3. Reflexión sobre la metodología aplicada	6
4. Marco Conceptual	7
4.1. Introducción al paradigma RAG	7
4.2. Origen de RAG	8
4.3. Evolución histórica de las arquitecturas de recuperación y generación	8
4.3.1. De la recuperación clásica a la búsqueda semántica	9
4.3.2. La evolución de los modelos generativos	9
4.3.3. Convergencia: hacia una generación informada por recuperación	9
4.4. Comparativa	10
4.4.1. Modelos generativos puros	10
4.4.2. Finetuning sobre modelos base	10
4.4.3. Recuperación de información clásica (IR)	11
4.4.4. Síntesis comparativa	11
4.5. Variantes técnicas del enfoque RAG	11
4.5.1. Nivel de agregación: RAG-end-to-end vs. RAG-token	11

4.5.2.	Arquitectura generativa: encoder-decoder vs. autoregresivo	12
4.5.3.	Criterios de selección conceptual	12
4.6.	Arquitectura y componentes	13
4.6.1.	Base de datos vectorial o motor de búsqueda semántica	13
4.6.2.	Representaciones numéricas (Embeddings)	13
4.6.3.	Modelo de lenguaje generativo (LLM)	14
4.6.4.	Controlador del flujo de datos	14
4.6.5.	Flujo general de operación	15
4.7.	Justificación del enfoque RAG	15
4.7.1.	Recuperación de información clásica (IR)	16
4.7.2.	Modelos generativos sin recuperación externa	16
4.7.3.	Finetuning de modelos preentrenados	16
4.7.4.	Comparativa conceptual de enfoques	16
4.7.5.	Adecuación al contexto del proyecto	17
4.8.	Conclusiones tecnológicas	17
4.9.	Estado actual	18
4.9.1.	Introducción	18
4.9.2.	Grado de adopción de la IA en el sector asegurador	18
4.9.3.	Aplicaciones actuales de la IA en el sector asegurador	18
4.9.4.	Retos y consideraciones éticas	19
4.9.5.	Conclusión	19
4.10.	Legal y Ética	19
5.	Soluciones Existentes	23
5.1.	Introducción	23
5.2.	Frameworks de integración RAG	24
5.2.1.	LangChain	24
5.2.2.	Haystack	25
5.2.3.	LlamaIndex	26
5.3.	Bases de datos vectoriales	26
5.3.1.	FAISS (Facebook AI Similarity Search)	26
5.3.2.	Milvus	28
5.3.3.	Weaviate	29
5.4.	Lenguaje generativo	30
5.4.1.	GPT (Generative Pre-trained Transformer)	30
5.4.2.	LLaMA (Large Language Model Meta AI)	30
5.4.3.	Flan-T5	31
5.4.4.	Modelos ajustados específicamente para RAG	32
5.5.	Modelos de embeddings	32
5.6.	Conclusiones sobre soluciones	33

6. Análisis	35
6.1. Requisitos técnicos y funcionales	35
6.2. Motivación del diseño arquitectónico	38
6.3. Análisis del flujo de consulta	39
6.4. Estructura documental	41
6.5. Criterios de elección tecnológica	42
7. Diseño de la Solución	45
7.1. Introducción	45
7.2. Principios de diseño	46
7.3. Arquitectura general	47
7.4. Diseño funcional del backend	49
7.4.1. Modelo de datos relacional	52
7.5. Diseño funcional del frontend	54
7.6. Sistema conversacional	56
7.7. Decisiones técnicas clave	56
7.8. Resumen del diseño	58
8. Implementación	61
8.1. Introducción	61
8.2. Entorno de desarrollo	61
8.3. Backend-tecnologías	62
8.4. Backend-microservicios	63
8.4.1. Servicio de Ingestión de Documentos	63
8.4.2. Servicio de Gestión de Índices FAISS	65
8.4.3. Servicio de Generación de Embeddings	67
8.4.4. Servicio de Búsqueda y Reranking	68
8.5. Frontend	71
8.6. Contenerización y orquestación	73
8.7. Modelos de lenguaje utilizados	74
8.8. Infraestructura y recursos	75
8.9. Mecanismos de seguridad implementados	78
8.10. Resumen del entorno de trabajo	79
9. Pruebas	81
9.1. Cobertura de pruebas	81
9.2. Pruebas funcionales sobre el sistema completo	82
9.3. Diseño de las pruebas	83
9.4. Preguntas seleccionadas	83
9.5. Análisis detallado de P1	83
9.6. Análisis detallado de la consulta P2	85
9.7. Análisis detallado de la consulta P3	87

9.8. Cambio de prompt	89
9.8.1. Pregunta sobre dato concreto	89
9.8.2. Pregunta sobre precio	90
9.8.3. Pregunta sobre campo descripción	92
9.9. Pruebas de usabilidad	94
9.9.1. Diseño experimental	94
9.9.2. Resultados por participante	95
10. Conclusiones	101
10.1. Aportaciones	101
10.1.1. Contribuciones técnicas	101
10.1.2. Contribuciones metodológicas	102
10.2. Impacto socio-económico	102
10.3. Reflexión personal y académica	102
10.4. Trabajo futuro	102
10.4.1. Fortalecimiento de la ciberseguridad	103
10.5. Visión a largo plazo	103
Appendices	105
Apéndice A. Manual de Instalación	107
A.1. Requisitos del sistema	107
A.1.1. Hardware mínimo recomendado	107
A.1.2. Software necesario	107
A.2. Clonación del repositorio	107
A.3. Estructura del sistema	108
A.4. Variables de entorno	108
A.5. Configuración del puerto y OAuth	108
A.6. Obtención de claves OAuth	109
A.7. Ejecución del sistema	109
A.8. Uso del sistema	110
A.9. Regeneración de índices	110
A.10. Desinstalación	111
A.11. Consideraciones finales	111
Apéndice B. Manual de Usuario	113
B.1. Acceso a la aplicación	113
B.2. Estructura de la interfaz	113
B.3. Gestión de expedientes	114
B.3.1. Crear un nuevo expediente	114
B.3.2. Renombrar o eliminar	114
B.4. Carga de documentos	114

B.5. Uso del chat	115
B.5.1. Iniciar una conversación	115
B.5.2. Interpretación de la respuesta	115
B.5.3. Atajos y sugerencias	115
B.6. Gestión de conversaciones	115
B.7. Cierre de sesión	115
Apéndice C. Manual del Desarrollador	117
C.1. Visión general	117
C.2. Estructura del repositorio	117
C.3. Backend	118
C.3.1. Capa <code>core</code>	118
C.3.2. Capa <code>middleware</code>	118
C.3.3. Servicios RAG (<code>app/services</code>)	119
C.3.4. Routers FastAPI (<code>app/api</code>)	121
C.4. Frontend (React 18)	121
C.4.1. Utilidades	122
C.5. Configuración y variables de entorno	122
C.6. Contenerización y despliegue	122
C.7. Flujo completo RAG	123
C.8. Buenas prácticas y extensiones	123
11. Anexos	125
11.1. Glosario y lista de siglas	125
11.1.1. Siglas	125
Términos técnicos	126
Bibliografía	129

Índice de cuadros

3.1. Fases de desarrollo del proyecto previstas y ajustadas según evolución real	6
4.1. Comparativa técnica entre IR, LLM, Fine-Tuning (FT) y RAG	17
4.2. Correspondencia AI Act / ISO 42001 y su implementación en el sistema	20
8.1. Contramedidas de seguridad actualmente activas	79
9.1. Cobertura por fichero.	81
9.2. Documentos incluidos en el expediente de pruebas	83
9.3. Preguntas seleccionadas para el análisis de flujo	83
9.4. Resumen de la sesión de Isabel	96
9.5. Resumen de la sesión de Noa	97
9.6. Resumen de la sesión de Miguel	98

Índice de figuras

6.1. Diagrama de casos de uso	37
6.2. Diagrama de secuencia(Subida de documento)	39
6.3. Diagrama de secuencia de consulta	39
7.1. Diagrama de secuencia	49
7.2. Diagrama de flujo de ingesta	51
7.3. Diagrama de clases UML	53
8.1. Diagrama de despliegue	77

Introducción

1.1 Introducción

En la actualidad, el sector asegurador maneja una creciente cantidad de datos e información, en muchos casos contenida en una gran cantidad de documentos extensos y complejos. La búsqueda, recuperación y consulta eficiente de esta información supone un desafío, ya que afecta directamente a la productividad, calidad del servicio y capacidad de respuesta frente a las demandas de los clientes.

Este trabajo surge de la necesidad de proporcionar una solución efectiva y moderna a este problema recurrente mediante la aplicación de tecnologías avanzadas de inteligencia artificial. En concreto, se propone el desarrollo de un sistema basado en técnicas de *Generación Aumentada por Recuperación* (RAG, por sus siglas en inglés, *Retrieval-Augmented Generation*), combinando las fortalezas del procesamiento del lenguaje natural con motores de búsqueda semánticos, para permitir una consulta rápida y precisa dentro del entorno asegurador.

La elección de este enfoque no es casual: los modelos RAG han mostrado resultados prometedores en diversos contextos profesionales, destacando por su capacidad para manejar grandes cantidades de información y proporcionar respuestas contextualizadas y precisas. Además, la implementación de una arquitectura basada en módulos y contenedores asegura que la solución sea no solo robusta y escalable, sino también adaptable a futuras necesidades o cambios en las condiciones del mercado.

Este proyecto pretende ir más allá de las soluciones tradicionales, ofreciendo una alternativa inteligente que reduzca tiempos de consulta, minimice errores humanos y, en última instancia, mejore sustancialmente la experiencia tanto del personal encargado de gestionar dicha información como de los clientes finales.

A lo largo de esta memoria se detalla el proceso seguido para abordar este desafío, desde la

fase inicial de investigación y análisis, pasando por el diseño y desarrollo del sistema, hasta llegar finalmente a la evaluación y comparativa de resultados frente a metodologías más convencionales. Todo ello con el objetivo último de validar la eficacia y beneficios que aportan las soluciones que integran este tipo de sistemas en el sector asegurador.

1.2 Motivación

Durante el periodo de prácticas de empresa realizado en una entidad que presta servicios tecnológicos a diversas aseguradoras, he podido observar de primera mano el considerable retraso tecnológico presente en este sector. Esta experiencia directa me permitió identificar claramente las dificultades y limitaciones que surgen al gestionar grandes volúmenes de información mediante métodos tradicionales, generalmente lentos y propensos a errores humanos. Al mismo tiempo, soy plenamente consciente del impacto transformador que están teniendo y tendrán las tecnologías basadas en inteligencia artificial en prácticamente todos los ámbitos profesionales y sociales. Específicamente, considero que el sector asegurador, dada su magnitud y la repercusión directa e indirecta que tiene sobre prácticamente toda la población, necesita abordar con urgencia una transformación digital profunda. Este convencimiento personal ha sido un motor fundamental para llevar a cabo este proyecto, buscando aportar una solución tecnológica innovadora que permita optimizar significativamente procesos críticos y mejorar la calidad del servicio que estas entidades proporcionan a sus clientes.

Objetivos y Alcance

2.1 Objetivos

2.1.1 Objetivo principal

El objetivo principal de este proyecto es desarrollar e integrar un sistema basado en inteligencia artificial, fundamentado en técnicas de *Generación Aumentada por Recuperación* (RAG, por sus siglas en inglés, *Retrieval-Augmented Generation*), que permita realizar consultas inteligentes, precisas y eficientes sobre documentación especializada en el ámbito del sector asegurador.

2.1.2 Objetivos secundarios

A fin de alcanzar el objetivo principal, se establecen los siguientes objetivos secundarios:

- Diseñar una arquitectura de software modular y escalable, que facilite el mantenimiento, evolución y despliegue del sistema.
- Incorporar mecanismos de búsqueda semántica y recuperación de información que mejoren la calidad y relevancia de los resultados obtenidos.
- Implementar una interfaz de programación de aplicaciones (API) clara y extensible, que posibilite la integración con otros sistemas corporativos.
- Establecer procedimientos de validación que permitan evaluar el rendimiento y la eficacia del sistema desarrollado en escenarios reales o simulados.

2.2 Tareas a realizar

Las tareas planteadas para alcanzar los objetivos propuestos se agrupan en las siguientes fases:

-
1. **Planificación inicial:** definir el alcance del proyecto, los objetivos específicos y el cronograma estimado de desarrollo. Las actividades clave incluyen:
 - Establecer criterios de evaluación y métodos de control de riesgos.
 2. **Análisis del contexto y los requisitos:** estudiar el problema actual relacionado con la recuperación de información en el sector asegurador, e identificar las necesidades del sistema. Se desarrollarán las siguientes tareas:
 - Identificar los requisitos funcionales y no funcionales del sistema propuesto.
 - Investigar soluciones existentes y tecnologías afines que puedan servir como base o referencia.
 3. **Diseño e implementación del sistema:** definir la arquitectura general y los módulos funcionales. Las actividades contempladas son:
 - Desarrollar los módulos funcionales necesarios: consulta semántica, gestión documental, interacción con el usuario, etc.
 - Integrar modelos de PLN y mecanismos de recuperación contextualizada de información.
 4. **Validación y evaluación:** realizar pruebas de funcionamiento y precisión, así como comparación con métodos actuales. Se llevarán a cabo:
 - Comparar su rendimiento con métodos tradicionales empleados en el ámbito asegurador.
 - Documentar los resultados obtenidos y proponer líneas de mejora o evolución futura.

Metodología

Todo trabajo de desarrollo tecnológico requiere una planificación estructurada que oriente los esfuerzos hacia el cumplimiento de los objetivos definidos. En este proyecto se ha optado por una metodología iterativa e incremental, adecuada para entornos con alto componente exploratorio y tecnológico, como es el caso de los sistemas basados en inteligencia artificial. Esta elección permite ajustar la planificación y el desarrollo conforme se obtienen resultados parciales o surgen nuevas necesidades técnicas.

3.1 Estrategia metodológica

La estrategia general se ha basado en el enfoque de desarrollo por prototipos, combinado con principios de gestión ágil. Esto ha permitido construir versiones intermedias funcionales del sistema, evaluar su comportamiento en cada iteración y aplicar mejoras continuas. Este enfoque facilita también la integración progresiva de tecnologías específicas y la validación temprana de decisiones arquitectónicas clave.

A lo largo del desarrollo se han seguido ciclos de trabajo estructurados en fases, con revisiones frecuentes y ajustes planificados. Se ha utilizado una hoja de ruta flexible, revisada periódicamente, como herramienta de seguimiento del avance y de reasignación de tareas.

3.2 Fases y planificación temporal

Las fases de trabajo definidas inicialmente, así como los ajustes realizados durante el proceso, se detallan en la tabla 3.1. Esta planificación recoge el desarrollo progresivo desde el análisis inicial hasta la redacción final de la memoria.

Nombre de actividad	Semanas
Análisis inicial del problema y requisitos técnicos	1 – 2
Investigación y aprendizaje de herramientas tecnológicas	2 – 3
Diseño preliminar de arquitectura	3 – 4
Configuración de entorno de desarrollo y bases de datos	4 – 5
Desarrollo del primer prototipo funcional	5 – 6
Integración de modelo RAG y búsqueda semántica	6 – 8
Optimización del sistema de embeddings (uso GPU)	7 – 8
Implementación de autenticación y gestión de usuarios	8 – 9
Desarrollo del segundo prototipo con interfaz de usuario	9 – 11
Optimización visual y funcional del frontend	10 – 11
Pruebas técnicas y evaluación del sistema	11 – 13
Comparativa con métodos tradicionales	12 – 13
Redacción de documentación técnica y manuales	13 – 14
Elaboración de la memoria del TFG	14 – 15

Cuadro 3.1: Fases de desarrollo del proyecto previstas y ajustadas según evolución real

3.3 Reflexión sobre la metodología aplicada

Cada una de las fases descritas ha requerido un esfuerzo específico y ha estado sujeta a revisiones en función de los resultados obtenidos. En particular, la integración tecnológica y el ajuste de los modelos de recuperación aumentada han supuesto un reto relevante, dada su complejidad y el carácter innovador del enfoque.

El enfoque iterativo ha permitido corregir desviaciones a tiempo, adaptar el alcance funcional a los recursos disponibles y priorizar las funcionalidades más críticas. Esta estrategia ha sido clave para garantizar la viabilidad técnica del proyecto en un entorno controlado y con recursos limitados, como es habitual en el contexto de un Trabajo Fin de Grado.

Marco Conceptual

4.1 Introducción al paradigma RAG

En el contexto actual de avance acelerado de la inteligencia artificial, los modelos de lenguaje de gran tamaño (LLMs, por sus siglas en inglés) como GPT, LLaMA o Claude han alcanzado un nivel de sofisticación notable, siendo capaces de comprender, resumir y generar texto en lenguaje natural con alta fluidez y coherencia. Estos modelos se entrenan sobre grandes volúmenes de datos textuales y aprenden representaciones profundas del lenguaje, lo que los hace especialmente eficaces en tareas generales de procesamiento del lenguaje natural.

Sin embargo, una de sus limitaciones más críticas es la naturaleza estática del conocimiento que contienen: una vez entrenados, no pueden incorporar información nueva sin reentrenamiento, un proceso costoso y técnicamente complejo. Como consecuencia, los LLMs tienden a generar respuestas basadas únicamente en lo que han “aprendido” de sus datos de entrenamiento, lo cual puede derivar en respuestas imprecisas, desactualizadas o incluso incorrectas en dominios especializados o contextos que requieren trazabilidad.

Para resolver esta limitación, surge el enfoque conocido como *Retrieval-Augmented Generation* (RAG), o generación aumentada mediante recuperación. Este paradigma arquitectónico combina dos enfoques clásicos: la recuperación de información (IR) y la generación de lenguaje natural. En términos simples, RAG dota al modelo generativo de acceso a una memoria externa dinámica, una base documental, que puede ser consultada en tiempo real durante la inferencia. De este modo, el modelo no depende exclusivamente de su entrenamiento previo, sino que puede incorporar conocimiento adicional, específico y actual, al generar sus respuestas.

Más allá de su funcionalidad inmediata, RAG representa un cambio profundo en el diseño de sistemas de IA: separa el almacenamiento del conocimiento (corpus vectorizado e indexado) del componente de razonamiento lingüístico (modelo generativo), lo que permite construir sistemas más auditables, escalables y fácilmente actualizables. Esta separación modular tiene importantes implicaciones tanto desde la perspectiva técnica como desde la de la ingeniería del software, especialmente en sectores regulados como el asegurador, donde la trazabilidad, la precisión factual

y la adaptación a cambios normativos son requisitos esenciales.

En las siguientes secciones se abordará el origen y evolución del paradigma RAG, sus fundamentos técnicos, las ventajas estratégicas que ofrece en entornos empresariales, y las limitaciones técnicas que debe afrontar su implementación práctica.

4.2 Origen y evolución del paradigma RAG

El término *Retrieval-Augmented Generation* (RAG) fue acuñado por primera vez en el artículo de Lewis et al. [16], publicado por Facebook AI Research en 2020. En este trabajo, los autores propusieron una arquitectura híbrida diseñada para mejorar el rendimiento de los modelos de lenguaje en tareas intensivas en conocimiento, como el *open-domain question answering*, combinando un recuperador de documentos basado en búsqueda semántica con un modelo generativo del tipo *seq2seq*.

La propuesta de Lewis et al. marcó un punto de inflexión en la evolución de los sistemas de PLN. Hasta entonces, los enfoques predominantes se dividían entre:

- Modelos extractivos, como BERT aplicado a QA, que seleccionan fragmentos existentes como respuesta, sin generar texto nuevo [4].
- Modelos generativos puros, como GPT o T5, que generan respuestas desde cero pero sin acceso explícito a una base documental externa [24].

RAG se posiciona como una solución intermedia: permite generar respuestas sintéticas, pero basadas en evidencia textual concreta, proveniente de un corpus documental externo. Este enfoque es especialmente valioso en dominios donde el conocimiento cambia con frecuencia, y donde se requiere trazabilidad y control sobre las fuentes empleadas en la generación.

Desde su introducción, la arquitectura RAG ha inspirado numerosos desarrollos y adaptaciones. Se ha convertido en la base conceptual de frameworks como Haystack, LlamaIndex y LangChain, que implementan variantes de pipelines RAG para tareas específicas como asistencia conversacional, búsqueda semántica empresarial y generación automática de documentación técnica. También ha sido objeto de múltiples estudios comparativos frente a otras estrategias como el *fine-tuning* o la recuperación sin generación, demostrando ventajas significativas en precisión, adaptabilidad y coste computacional [14].

4.3 Evolución histórica de las arquitecturas de recuperación y generación

La arquitectura RAG no surge en un vacío técnico, sino como la confluencia lógica de dos líneas de desarrollo en el campo del procesamiento del lenguaje natural (PLN): por un lado, los sistemas de recuperación de información (IR), y por otro, los modelos de lenguaje generativo. Ambos enfoques han evolucionado durante décadas y han alcanzado puntos de madurez que han permitido su integración en soluciones híbridas como RAG.

4.3.1 De la recuperación clásica a la búsqueda semántica

Los primeros sistemas IR se basaban en el modelo de espacio vectorial y utilizaban métricas como *TF-IDF* y *BM25* para calcular la relevancia de los documentos frente a una consulta. Estos métodos eran eficientes y comprensibles, pero sufrían limitaciones notables en la comprensión semántica: trabajaban a nivel de coincidencia de términos y no captaban relaciones conceptuales entre palabras ni el contexto más amplio del discurso.

Con la irrupción del aprendizaje profundo en PLN, especialmente a partir de 2013 con la introducción de *word embeddings* como Word2Vec [19], se hizo posible representar palabras en espacios semánticos continuos. Más tarde, modelos como BERT [4], RoBERTa y DistilBERT permitieron calcular representaciones contextuales de frases y documentos enteros, habilitando lo que hoy se conoce como **búsqueda semántica**.

Estos avances permitieron que los sistemas de IR pasaran de ser simplemente extractivos a capaces de realizar una recuperación con mayor profundidad conceptual, abriendo la puerta a su integración con modelos generativos.

4.3.2 La evolución de los modelos generativos

En paralelo, los modelos de lenguaje también experimentaron una evolución significativa. Los modelos *n-gram*, los Hidden Markov Models (HMM) y las primeras redes neuronales recurrentes (RNNs) ofrecían una generación limitada y rígida. El salto cualitativo se dio con la introducción de arquitecturas **Transformer** [28], que permitieron construir modelos como GPT, BERT, T5 y BART.

Los modelos autoregresivos (ej. GPT) demostraron capacidades sorprendentes para la generación libre de texto, mientras que los modelos encoder-decoder (ej. T5, BART) ofrecían ventajas en tareas estructuradas y de traducción automática. Sin embargo, estos modelos seguían dependiendo exclusivamente de la información contenida en sus parámetros entrenados, lo cual limitaba su aplicabilidad en entornos que requieren acceso a datos actualizados, precisos y trazables.

4.3.3 Convergencia: hacia una generación informada por recuperación

La necesidad de unir lo mejor de ambos mundos —la precisión factual y actualizable de la recuperación con la fluidez lingüística de la generación— llevó al diseño de arquitecturas híbridas. Entre los primeros intentos destacan los sistemas QA con recuperación + reranking, que empleaban BERT para reordenar documentos recuperados, pero sin generación real.

El trabajo de Lewis et al. [16] formalizó este concepto con RAG, donde por primera vez se integró de forma efectiva un módulo de recuperación semántica y un generador neuronal dentro de un mismo flujo de inferencia. Esta integración permitió construir respuestas más útiles, basadas en evidencia, y adaptadas dinámicamente al corpus documental disponible.

Desde entonces, RAG ha sido reconocido como un paradigma arquitectónico clave en el desarrollo de sistemas de PLN con propósito práctico, especialmente en sectores como salud, legal,

investigación científica y finanzas, donde la precisión factual y la auditabilidad son fundamentales.

4.4 Comparativa conceptual con otros enfoques

El enfoque RAG no es la única estrategia para abordar tareas de acceso a información mediante lenguaje natural. En la literatura y en la práctica, existen al menos tres grandes alternativas con las que se puede establecer una comparación conceptual: los modelos generativos puros, los modelos ajustados mediante *fine-tuning*, y los sistemas tradicionales de recuperación de información. Esta sección analiza las características distintivas de cada uno en relación con RAG, subrayando sus ventajas y limitaciones desde una perspectiva teórica.

4.4.1 Modelos generativos puros

Los modelos generativos autoregresivos de gran escala (LLMs), como GPT-3 o GPT-4, son capaces de generar texto altamente coherente sin acceso externo a fuentes de información. Operan únicamente en base a los parámetros adquiridos durante el entrenamiento, lo que les otorga una alta fluidez lingüística y una flexibilidad notable en tareas abiertas o creativas.

No obstante, esta aproximación presenta importantes limitaciones cuando se requiere exactitud factual, trazabilidad de la información o actualización continua del conocimiento. Al carecer de un mecanismo explícito de recuperación, estos modelos pueden generar contenido incorrecto o inventado (*hallucinations*) [29], y no pueden justificar sus respuestas más allá de su entrenamiento previo. En contextos profesionales o regulados, estas deficiencias suponen un riesgo crítico.

4.4.2 Finetuning sobre modelos base

Otra estrategia común es el *fine-tuning*, que consiste en ajustar los pesos de un modelo previamente entrenado utilizando un corpus específico del dominio objetivo. Esta técnica permite especializar el modelo en tareas concretas o en terminología sectorial, y puede mejorar su rendimiento en tareas controladas.

Sin embargo, el *fine-tuning* requiere:

- Un conjunto de datos de alta calidad, curado y representativo del dominio.
- Capacidad computacional elevada para realizar el entrenamiento adicional.
- Repetición del proceso ante cambios relevantes en la información base, lo que limita la adaptabilidad y la actualización dinámica del sistema.

Además, no resuelve el problema de la opacidad: aunque el modelo haya aprendido del corpus, no puede citar ni explicar la fuente de su conocimiento durante la inferencia [17].

4.4.3 Recuperación de información clásica (IR)

Los sistemas IR tradicionales, como los basados en BM25 o Elasticsearch, ofrecen respuestas extractivas, rápidas y trazables. Son especialmente útiles en sistemas de búsqueda documental o cuando se requiere mostrar evidencia textual directa al usuario.

Sin embargo, su principal debilidad reside en la falta de capacidad generativa. La carga interpretativa recae sobre el usuario, quien debe extraer el significado relevante de los fragmentos recuperados. Además, su rendimiento se ve limitado por la calidad del índice y por la literalidad de las coincidencias, lo que reduce su eficacia en consultas expresadas en lenguaje natural libre [27].

4.4.4 Síntesis comparativa

La arquitectura RAG se sitúa como una solución intermedia que combina:

- La adaptabilidad y fluidez de los modelos generativos.
- La trazabilidad y precisión de los sistemas IR.
- La posibilidad de actualización dinámica del corpus sin reentrenamiento.

Esto la convierte en una arquitectura especialmente valiosa en contextos donde es necesario producir lenguaje natural preciso, pero fundamentado en evidencia externa, actualizable y auditable.

4.5 Variantes técnicas del enfoque RAG

Desde su formulación original, el paradigma RAG ha dado lugar a distintas variantes arquitectónicas, cuya existencia responde a las múltiples necesidades de equilibrio entre eficiencia, trazabilidad, interpretabilidad y coste computacional. Estas variantes no implican una ruptura con el diseño básico, sino una extensión de su lógica modular, orientada a adaptar el esquema general de RAG a requisitos específicos de aplicación o a restricciones tecnológicas.

4.5.1 Nivel de agregación: RAG-end-to-end vs. RAG-token

Uno de los criterios de clasificación más frecuentes entre las variantes de RAG se basa en el nivel de granularidad con que se procesan los documentos recuperados. En la configuración conocida como **RAG-end-to-end**, todos los fragmentos documentales recuperados se concatenan en un único bloque que se proporciona como entrada al modelo generativo. Esta estrategia, la más utilizada en entornos generales, ofrece una implementación sencilla y una velocidad razonable de inferencia, aunque puede limitar la interpretabilidad y la atribución precisa de la respuesta a fuentes concretas.

Por otro lado, la variante **RAG-token** procesa cada fragmento por separado y genera predicciones condicionadas por cada uno de ellos, permitiendo una mayor trazabilidad en el proceso de generación. Esta estrategia fue explorada por Izacard y Grave [12], quienes propusieron una

distilación del conocimiento desde el generador hacia el recuperador para mejorar la precisión y la relevancia contextual. RAG-token se aplica especialmente en tareas donde es necesario justificar cada elemento de la respuesta con su fuente correspondiente, o donde se desea calcular la relevancia de cada documento de forma explícita. No obstante, su complejidad computacional y la necesidad de una arquitectura más sofisticada la hacen menos común en aplicaciones prácticas de escala reducida.

4.5.2 Arquitectura generativa: encoder-decoder vs. autoregresivo

Otra dimensión fundamental en la caracterización de variantes RAG es el tipo de modelo generativo que se utiliza. Existen dos familias principales:

- **Modelos encoder-decoder**, como los basados en arquitecturas *sequence-to-sequence*, en los que un codificador procesa la entrada completa (consulta + documentos) y un decodificador genera la respuesta. Estos modelos permiten un mayor control sobre la forma y contenido de la salida y son especialmente eficaces en tareas supervisadas o con formatos estructurados. El modelo T5 [24] es un referente dentro de esta familia por su capacidad de unificar múltiples tareas de PLN bajo un mismo marco de entrenamiento.
- **Modelos autoregresivos**, como los basados en arquitecturas *decoder-only*, en los que la generación de texto se realiza token a token a partir de un contexto acumulado. Este enfoque ha demostrado ser más flexible y robusto en tareas de generación libre, aunque ofrece menos control estructural y puede ser más propenso a errores de factualidad si no se combina adecuadamente con mecanismos de recuperación.

Ambas aproximaciones pueden integrarse en un sistema RAG, dependiendo del diseño general del sistema, de los requisitos de la tarea y de las capacidades del entorno de despliegue.

4.5.3 Criterios de selección conceptual

La elección de una variante de RAG no debe entenderse como una preferencia arbitraria, sino como el resultado de una evaluación cuidadosa de los objetivos del sistema. Entre los factores que condicionan la decisión se incluyen:

- La necesidad de justificar documentalmente las respuestas (trazabilidad).
- El equilibrio deseado entre velocidad de inferencia y precisión contextual.
- La capacidad del sistema para manejar consultas ambiguas o abiertas.
- Las restricciones del dominio de aplicación (e.g., jurídico, médico, financiero).
- El tipo de interacción esperado con el usuario (conversacional, documental, generativa).

La existencia de estas variantes demuestra que RAG no es un enfoque cerrado, sino un paradigma flexible que admite múltiples configuraciones internas para adaptarse a contextos de uso muy diversos.

4.6 Arquitectura y componentes principales

La implementación de un sistema basado en RAG requiere una arquitectura modular bien definida, capaz de gestionar eficientemente tanto la recuperación semántica de información como la generación de respuestas por parte de un modelo de lenguaje. A diferencia de otros sistemas monolíticos, donde todo el procesamiento se concentra en un único modelo entrenado de forma estática, RAG se apoya en una estructura de componentes desacoplados que trabajan de forma coordinada para generar las respuestas con información actual, precisa y contextual.

Esta modularidad permite escalar el sistema por partes, facilitar el mantenimiento, personalizar los componentes en función del dominio y, lo más importante, dotar al sistema de una fuente de conocimiento actualizable sin necesidad de modificar el modelo de lenguaje subyacente.

A continuación, se describen los componentes esenciales que conforman una arquitectura RAG típica, así como su función dentro del flujo general del sistema.

4.6.1 Base de datos vectorial o motor de búsqueda semántica

Uno de los pilares del sistema es la base de datos vectorial, encargada de almacenar las representaciones numéricas (embeddings) de todos los fragmentos documentales que se desean poner a disposición del sistema. Esta base actúa como el motor de búsqueda semántica, permitiendo recuperar aquellos fragmentos que tienen mayor similitud conceptual con la consulta planteada por el usuario.

A diferencia de las bases de datos relacionales o documentales, una base vectorial organiza los datos en un espacio n-dimensional donde los documentos son vectores y la búsqueda se realiza en función de la proximidad matemática (por ejemplo, distancia coseno o L2). Esta representación permite encontrar fragmentos que, aunque no compartan palabras exactas, están relacionados semánticamente.

Entre las soluciones más utilizadas en este ámbito destacan:

- **FAISS:** biblioteca desarrollada por Facebook AI para búsquedas de similitud altamente eficientes, especialmente útil en entornos donde se requiere rendimiento en GPU [13].
- **Milvus:** base de datos orientada a grandes volúmenes de datos, con soporte nativo para operaciones distribuidas.
- **Weaviate:** plataforma completa que combina almacenamiento vectorial con funcionalidades adicionales como GraphQL, esquemas enriquecidos y vectorización automática.

Estas bases no solo permiten búsquedas por similitud, sino que también pueden almacenar metadatos (como el origen, fecha o categoría del fragmento), lo que habilita filtros más precisos durante la recuperación.

4.6.2 Representaciones numéricas (Embeddings)

Los embeddings son el puente entre el lenguaje natural y el espacio vectorial. Se trata de vectores de dimensión fija que codifican el contenido semántico de un texto. Dos fragmentos de

texto con significados similares deberían dar lugar a vectores próximos en dicho espacio.

Para generar estos vectores se utilizan modelos entrenados específicamente en tareas de *sentence encoding* o *text embedding*. Algunos de los más conocidos y eficaces son:

- **Sentence-BERT:** una variante de BERT diseñada para generar embeddings útiles en tareas de similitud semántica [26].
- **Universal Sentence Encoder (USE):** desarrollado por Google, optimizado para tareas como clasificación o recuperación [2].
- **Instructor:** un modelo moderno que permite condicionar los embeddings según instrucciones específicas, lo que mejora la recuperación en dominios concretos.

La elección del modelo de embeddings es crucial: debe equilibrar precisión semántica, rendimiento computacional y compatibilidad con el dominio de aplicación (en este caso, el asegurador).

4.6.3 Modelo de lenguaje generativo (LLM)

El modelo de lenguaje es el encargado de generar la respuesta final que recibe el usuario. Su trabajo consiste en leer los fragmentos recuperados y generar un texto coherente, contextualizado y relevante, que responda a la pregunta original.

Algunos modelos utilizados habitualmente en sistemas RAG incluyen:

- **GPT (OpenAI):** gran capacidad generativa, pero requiere acceso vía API y tiene costes asociados.
- **LLaMA:** opción de código abierto desarrollada por Meta, más flexible para despliegues locales.
- **T5 / Flan-T5:** modelos encoder-decoder que permiten tareas de generación condicionadas, con muy buenos resultados en entornos supervisados [24].

El modelo no necesita conocer previamente el contenido de los documentos; su papel es procesar la información suministrada en el contexto (*prompt*) y generar una respuesta ajustada. Esta separación permite reutilizar el mismo LLM en distintos dominios, cambiando únicamente la base documental.

4.6.4 Controlador del flujo de datos

El controlador es el componente orquestador. Su función es coordinar las distintas fases del sistema, garantizando que la información fluye correctamente desde la entrada hasta la salida. Entre sus responsabilidades destacan:

- Convertir la consulta del usuario en un embedding y consultar la base vectorial.

-
- Recoger los k fragmentos más relevantes y construir el prompt final para el modelo.
 - Gestionar los formatos de entrada y salida (por ejemplo, para ofrecer respuestas estructuradas o justificar la fuente documental).
 - Encapsular la lógica de negocio del sistema (por ejemplo, aplicar filtros de seguridad, controlar tiempos de respuesta, etc.).

Este componente se implementa habitualmente como parte del *backend* del sistema, y se conecta a través de una API REST o gRPC con los distintos servicios (vectorización, recuperación, generación, etc.).

4.6.5 Flujo general de operación

El flujo completo de un sistema RAG puede resumirse en los siguientes pasos:

1. **Entrada del usuario:** se formula una pregunta o instrucción en lenguaje natural.
2. **Vectorización de la consulta:** el texto se transforma en un vector utilizando un modelo de embeddings.
3. **Consulta a la base vectorial:** se recuperan los fragmentos más similares semánticamente a la consulta.
4. **Construcción del prompt:** se combinan los fragmentos con la pregunta del usuario para formar la entrada al LLM.
5. **Generación de la respuesta:** el modelo produce una respuesta final utilizando el contexto documental.
6. **Entrega y visualización:** la respuesta se devuelve al usuario, junto con posibles referencias a los documentos utilizados.

Este flujo es altamente adaptable y permite implementar optimizaciones en cada etapa, como caché de respuestas, preprocesamiento de consultas, o resúmenes automáticos de los fragmentos recuperados.

4.7 Justificación del enfoque RAG

La elección del paradigma RAG en el diseño del sistema propuesto no es una decisión arbitraria, sino el resultado de una evaluación comparativa entre distintas estrategias utilizadas habitualmente en tareas de procesamiento de lenguaje natural con acceso a información documental. Cada una de estas alternativas presenta ventajas y limitaciones que deben ser ponderadas en función del dominio de aplicación, los requisitos funcionales y las restricciones operativas del sistema.

4.7.1 Recuperación de información clásica (IR)

Los enfoques basados exclusivamente en recuperación de información (IR), como los implementados mediante motores BM25 o Elasticsearch, se apoyan en técnicas de coincidencia léxica para localizar documentos relevantes. Aunque estos sistemas pueden incorporar mecanismos de búsqueda semántica mediante embeddings, su salida sigue siendo extractiva: devuelven fragmentos que el usuario debe interpretar o filtrar manualmente.

Este enfoque resulta útil en contextos donde la carga interpretativa recae en expertos humanos, o cuando el volumen de información es limitado. Sin embargo, se muestra insuficiente en tareas que requieren generación automática de respuestas completas, coherentes y comprensibles para usuarios no especializados [27].

4.7.2 Modelos generativos sin recuperación externa

Los modelos generativos autoregresivos, como los grandes modelos de lenguaje (LLMs), son capaces de producir texto fluido a partir de una entrada en lenguaje natural. No obstante, su capacidad para proporcionar respuestas precisas y verificables está limitada por el hecho de que el conocimiento sobre el que operan está encapsulado en los pesos del modelo, congelado en el momento de su entrenamiento.

La ausencia de un mecanismo de acceso a bases documentales externas restringe su aplicabilidad en dominios donde la información cambia con frecuencia o donde se exige trazabilidad. Además, estos modelos pueden generar respuestas incorrectas con elevada confianza, fenómeno conocido como *alucinaciones*, lo cual compromete su fiabilidad en entornos críticos [29].

4.7.3 Finetuning de modelos preentrenados

Otra opción es el *fine-tuning* de modelos generales sobre un corpus específico del dominio. Este procedimiento permite adaptar el modelo a un vocabulario técnico concreto o a estilos discursivos propios del sector. Sin embargo, presenta múltiples inconvenientes:

- Requiere conjuntos de datos representativos, curados y etiquetados, lo cual implica un alto coste en tiempo y recursos.
- Introduce una dependencia operativa del proceso de reentrenamiento cada vez que se actualiza el conocimiento del dominio.
- No resuelve el problema de la falta de trazabilidad, ya que el modelo sigue sin ofrecer evidencias directas del origen de sus respuestas.

4.7.4 Comparativa conceptual de enfoques

A modo de síntesis, la tabla 4.1 resume las principales diferencias entre las alternativas descritas y el enfoque RAG, en función de criterios clave como generación, actualización, trazabilidad y coste computacional.

Criterio	IR	LLM	FT	RAG
Generación de lenguaje	✗	✓	✓	✓
Info. actualizada	✓*	✗	✗	✓*
Trazabilidad	✓	✗	✗	✓
Coste computacional	Bajo	Alto	Muy alto†	Moderado
¿Reentrenamiento?	✗	✗	✓	✗
Adaptación documental	✓	✗	✗	✓

Cuadro 4.1: Comparativa técnica entre IR, LLM, Fine-Tuning (FT) y RAG

* Mediante corpus externo o actualización manual.

† Requiere entrenamiento con grandes volúmenes de datos.

4.7.5 Adecuación al contexto del proyecto

En el contexto del presente trabajo, centrado en la consulta inteligente de documentación normativa, contractual y técnica del sector asegurador, el paradigma RAG se revela como la opción más adecuada. Su capacidad para integrar recuperación semántica y generación de respuestas, con apoyo documental explícito, permite satisfacer requisitos críticos como:

- Actualización continua del conocimiento sin necesidad de reentrenamiento.
- Fundamentación explícita de las respuestas, favoreciendo su auditabilidad.
- Modularidad y escalabilidad técnica en entornos basados en microservicios.

En suma, RAG ofrece un equilibrio óptimo entre flexibilidad, precisión, trazabilidad y mantenibilidad, lo que lo convierte en el núcleo arquitectónico más adecuado para la solución propuesta en este Trabajo de Fin de Grado.

4.8 Conclusiones tecnológicas

La revisión realizada en los apartados anteriores ha permitido construir una base conceptual y técnica sólida en torno a la arquitectura RAG. A través del análisis de sus principios, componentes clave, ventajas frente a otras alternativas y adecuación al contexto del proyecto, se han delimitado los motivos por los cuales este enfoque resulta especialmente indicado para sistemas de consulta basados en documentación interna y conocimiento de forma dinámica.

Desde el punto de vista de ingeniería, RAG destaca por su modularidad, escalabilidad y capacidad de actualización sin necesidad de reentrenamiento, cualidades esenciales en sectores donde la información está sujeta a cambios normativos, contractuales o técnicos, como es el caso del ámbito asegurador. Al desacoplar la lógica de generación de lenguaje del almacenamiento del conocimiento, se facilita tanto el mantenimiento como la evolución incremental del sistema, sin comprometer la trazabilidad ni la coherencia de las respuestas generadas.

La comparación con enfoques clásicos de recuperación de información, modelos generativos puros y estrategias de *finetuning* ha puesto de manifiesto que RAG representa un equilibrio óptimo entre precisión, eficiencia y control del conocimiento. Esta arquitectura no solo permite ofrecer respuestas más fiables y justificadas, sino que también habilita mecanismos de auditoría documental y control de versiones que resultan críticos en entornos regulados o sensibles.

Sobre esta base conceptual, el siguiente paso consistirá en examinar y comparar soluciones tecnológicas existentes que implementan el enfoque RAG —como LangChain, Haystack o LlamaIndex—, con el objetivo de seleccionar las herramientas más adecuadas para el desarrollo del sistema propuesto.

4.9 Estado actual del sector asegurador y adopción de la inteligencia artificial

4.9.1 Introducción

El sector asegurador se encuentra inmerso en un proceso de transformación digital impulsado por la adopción de tecnologías emergentes, entre las cuales la inteligencia artificial (IA) destaca por su potencial para optimizar procesos, mejorar la experiencia del cliente y desarrollar nuevos modelos de negocio. Esta evolución responde a la necesidad de adaptarse a un entorno cada vez más competitivo y a las crecientes expectativas de los consumidores en términos de personalización y eficiencia.

4.9.2 Grado de adopción de la IA en el sector asegurador

Según el *IX Termómetro de Inteligencia Artificial y Data en el sector asegurador español*, elaborado por Minsait e ICEA, el 80 % de las aseguradoras en España están trabajando en proyectos relacionados con IA, y dos tercios de ellas ya han implementado soluciones en su operativa diaria [20]. Estas iniciativas se centran principalmente en áreas como la mejora de la experiencia del cliente, la detección de fraudes y la optimización de procesos internos.

A nivel europeo, el informe de la Autoridad Europea de Seguros y Pensiones de Jubilación (EIOPA) publicado en 2024 indica que el 50 % de las entidades ya aplican IA en seguros de no vida, y el 24 % en el ramo de vida. Además, un 30 % y un 39 % de las aseguradoras esperan aplicar estas tecnologías próximamente en esos respectivos ramos [5].

4.9.3 Aplicaciones actuales de la IA en el sector asegurador

La inteligencia artificial se está aplicando en diversas áreas estratégicas del sector asegurador [3, 15, 10], entre las que destacan:

- **Automatización de procesos:** la IA permite automatizar tareas repetitivas como la tramitación de siniestros o la gestión documental, mejorando la eficiencia y reduciendo errores humanos.

-
- **Análisis predictivo y personalización:** gracias a técnicas de aprendizaje automático y minería de datos, es posible predecir necesidades futuras de los clientes y adaptar los productos a perfiles individuales.
 - **Detección de fraudes:** mediante la identificación de patrones atípicos o inconsistencias en los datos, la IA contribuye a detectar intentos de fraude con mayor rapidez y precisión [6].
 - **Atención al cliente:** los asistentes virtuales y chatbots permiten ofrecer soporte 24/7, mejorando los tiempos de respuesta y la calidad del servicio.

4.9.4 Retos y consideraciones éticas

La integración de IA en el sector asegurador plantea importantes retos, especialmente en materia de transparencia, equidad y cumplimiento normativo. La Ley de Inteligencia Artificial de la Unión Europea (AI Act), aprobada en 2024, clasifica como de alto riesgo los sistemas que afectan a la tarificación o evaluación de riesgos en seguros de salud o vida, imponiendo estrictos requisitos de supervisión y explicabilidad [21].

Asimismo, existe una creciente preocupación por el posible sesgo algorítmico en los modelos de IA, que podría derivar en prácticas discriminatorias. Para mitigar estos riesgos, las entidades deben implementar políticas de gobernanza de datos y validación de modelos, que garanticen la equidad, la auditabilidad y el respeto de los derechos del consumidor [1].

4.9.5 Conclusión

La inteligencia artificial está redefiniendo la forma en que las aseguradoras operan y se relacionan con sus clientes. Si bien ya se ha avanzado significativamente en la automatización y personalización de servicios, siguen existiendo desafíos técnicos, regulatorios y éticos que condicionan su adopción. En este contexto, el enfoque RAG se perfila como una arquitectura idónea para abordar los retos específicos asociados a la consulta documental, gracias a su capacidad para ofrecer respuestas precisas, trazables y fundamentadas en evidencia actualizable.

4.10 Consideraciones éticas y reglamentarias de la IA

La aplicación de IA en la gestión de pólizas y siniestros queda enmarcada en el **Artificial Intelligence Act** (AI Act) [7]. El artículo 6.2 clasifica como *sistema de alto riesgo* todo sistema que pueda influir en decisiones con efectos legales o significativos sobre clientes de seguros. Por ello, la solución descrita en este TFG debe cumplir:

- **Gestión de riesgos y pruebas previas** (arts. 9–10): análisis de sesgos y trazabilidad de datos de entrenamiento.
- **Registro de eventos** (arts. 12–14): bitácoras completas de consultas y fragmentos recuperados, conservadas durante seis años.

-
- **Transparencia** (art. 13): aviso visible «respuesta generada por IA» y referencia directa a la fuente documental mostrada al usuario.

Marcos y principios internacionales

ISO/IEC 42001 Primer estándar de *AI Management System*; establece controles de explicabilidad y ciclo PDCA [11].

UNESCO Recomendación mundial sobre la ética de la IA [25], que introduce la *Evaluación de Impacto Ético*.

OCDE Principios de IA centrados en robustez y rendición de cuentas [22].

Marco español de protección de datos

La AEPD exige comprobar sesgos y re-identificabilidad antes de liberar datos para entrenamiento. Su guía de *datos sintéticos* (abril 2025) [8] recomienda:

1. Seudonimización SHA-256 de NIF y matrículas.
2. Difuminado de firmas en PDF mediante `OpenCV`.
3. Garantizar $K\text{-anonymity} \geq 5$ en los conjuntos publicados

Mapeo de requisitos y contramedidas

Cuadro 4.2: Correspondencia AI Act / ISO 42001 y su implementación en el sistema

Requisito	Norma / Artículo	Implementación en este TFG
Registro de eventos	AI Act 12	Servicio APIRest <code>logger</code>
Transparencia	AI Act 13	Lista de fragmentos mostrados
Supervisión humana	AI Act 14.d	Este sistema está pensado para ser usado por un trabajador, antes de ser transmitida la información al cliente

Síntesis y riesgos residuales

El sistema desarrollado cumple con los objetivos funcionales definidos y se ajusta a los principios de modularidad, escalabilidad y precisión esperados en entornos aseguradores. No obstante, persisten algunos riesgos técnicos residuales que deberán abordarse en futuras fases de desarrollo:

1. **Sesgo en los datos de entrenamiento:** los documentos utilizados para pruebas pueden reflejar patrones históricos no generalizables. Se propone incorporar un muestreo más diverso y representativo en futuras evaluaciones.
2. **Vulnerabilidad a *prompt injection*:** aunque se han implementado filtros mediante expresiones regulares y técnicas básicas de control de entrada, se reconoce la necesidad de aplicar estrategias más robustas como validación semántica o ejecución en entornos restringidos.
3. **Impacto de cambios regulatorios:** el uso de modelos de lenguaje en el sector asegurador puede verse afectado por futuras regulaciones sobre IA. Esto requerirá una revisión periódica del sistema para garantizar su cumplimiento normativo.

En conclusión, la arquitectura actual prioriza un diseño seguro y transparente, permitiendo su adaptación a marcos regulatorios en evolución y dejando abierta la posibilidad de una futura certificación conforme a normativas aplicables en el ámbito de sistemas basados en IA.

Soluciones Existentes

5.1 Introducción

A partir del marco teórico desarrollado en capítulos anteriores, donde se han analizado los fundamentos del enfoque RAG y su idoneidad frente a otras alternativas, este capítulo tiene como objetivo identificar, describir y comparar soluciones tecnológicas existentes que permiten implementar dicha arquitectura de forma práctica.

La implementación de un sistema RAG funcional y eficiente requiere la combinación de varias herramientas que cubren distintas fases del flujo: desde la indexación semántica de documentos hasta la generación de respuestas por parte del modelo de lenguaje. En particular, se abordarán tres categorías principales de soluciones:

- **Frameworks de integración RAG:** plataformas que proporcionan herramientas de alto nivel para construir pipelines de recuperación y generación de forma modular, como LangChain, Haystack y LlamaIndex.
- **Bases de datos vectoriales:** tecnologías encargadas de almacenar y recuperar eficientemente representaciones vectoriales de fragmentos de texto, tales como FAISS, Milvus y Weaviate.
- **Modelos de lenguaje generativo:** arquitecturas preentrenadas utilizadas para interpretar el contexto recuperado y generar respuestas en lenguaje natural, entre los que se encuentran GPT, LLaMA, T5, entre otros.

El análisis de estas herramientas se realizará considerando criterios como escalabilidad, compatibilidad, facilidad de integración, rendimiento, flexibilidad y soporte a largo plazo. Al finalizar esta sección, se establecerán las bases para la selección concreta de las tecnologías utilizadas en el desarrollo del sistema propuesto.

5.2 Frameworks de integración RAG

5.2.1 LangChain

LangChain es un framework de desarrollo modular diseñado para facilitar la construcción de aplicaciones basadas en modelos de lenguaje de gran tamaño (LLMs), especialmente aquellas que requieren integración con fuentes de información externas. Desde su aparición, se ha consolidado como una de las herramientas de referencia en el ecosistema RAG debido a su flexibilidad, extensibilidad y capacidad para componer flujos de trabajo complejos mediante componentes reutilizables.

A diferencia de enfoques monolíticos, LangChain permite estructurar aplicaciones de IA en forma de cadenas (*chains*) o agentes que combinan múltiples pasos de procesamiento, incluyendo recuperación semántica, generación de prompts, ejecución de acciones condicionales o llamadas a herramientas externas. Esta arquitectura facilita la creación de sistemas escalables y mantenibles, particularmente útiles cuando se trabaja con documentos extensos, dominios especializados o interacciones de varias modalidades.

Entre sus funcionalidades más relevantes destacan:

- **Integración nativa con bases de datos vectoriales:** LangChain soporta múltiples motores como FAISS, Pinecone, Weaviate, Qdrant o Milvus, permitiendo seleccionar la base que mejor se adapte a los requisitos de cada proyecto.
- **Soporte para múltiples LLMs:** incluye conectores listos para usar con modelos propietarios (OpenAI, Cohere, Anthropic) y de código abierto (LLaMA, HuggingFace Transformers, GPT4All), facilitando pruebas y despliegues en entornos híbridos.
- **Pipeline de recuperación y generación (RAG):** ofrece componentes especializados para construir flujos RAG, como `RetrievalQA` y `ConversationalRetrievalChain`, donde se puede conectar directamente una base vectorial con un modelo generativo para realizar preguntas sobre documentos.
- **Herramientas de ingeniería de prompts:** permite definir plantillas dinámicas, inyectar contexto y controlar la estructura de entrada al LLM, lo cual es esencial para mejorar la precisión de las respuestas.
- **Manejo del estado conversacional:** incorpora estructuras como `Memory` para mantener el historial de conversación, crucial en sistemas conversacionales basados en documentos.

LangChain también destaca por su activa comunidad de desarrollo, documentación extensa y actualizaciones frecuentes. Su diseño orientado a componentes permite sustituir fácilmente cualquier parte del flujo sin afectar al resto del sistema, lo que resulta especialmente útil en entornos iterativos o de experimentación constante.

En el contexto de este proyecto, LangChain se ha utilizado de manera específica para construir dinámicamente los prompts y mantener el estado conversacional, permitiendo integrar la

memoria contextual en las respuestas del modelo. No se ha empleado para la recuperación semántica ni la generación directamente, sino como herramienta de orquestación ligera enfocada en enriquecer el contexto entregado al LLM. Su integración con Python y su orientación modular han permitido incorporarlo de forma sencilla en el sistema existente.

5.2.2 Haystack

Haystack es un framework de código abierto desarrollado por la empresa alemana Deepset, diseñado para construir sistemas de pregunta-respuesta, recuperación de información y asistentes conversacionales basados en documentos. Su arquitectura está orientada a *pipelines* modulares, lo que permite definir con claridad el flujo de entrada, recuperación, procesamiento y generación de respuestas.

Una de las principales virtudes de Haystack es su enfoque práctico y su rápida adopción en entornos industriales. Su diseño flexible facilita la combinación de distintos componentes como indexadores, modelos de embeddings, motores de búsqueda semántica, preprocesadores de texto y modelos generativos.

Entre sus características más relevantes destacan:

- **Soporte para múltiples bases vectoriales y motores de búsqueda:** permite integrar tecnologías como Elasticsearch, FAISS, Weaviate, Milvus y Qdrant, lo que lo hace adaptable a distintos requisitos de rendimiento y escalabilidad.
- **Compatibilidad con modelos de lenguaje modernos:** soporta tanto modelos alojados localmente (como Transformers desde Hugging Face) como servicios remotos (OpenAI, Cohere, etc.).
- **Pipelines definibles por YAML o mediante código Python:** lo que permite construir, visualizar y modificar con claridad flujos de procesamiento en cada paso de la recuperación y generación.
- **Herramientas para preprocesamiento y segmentación de documentos:** permite aplicar técnicas de *chunking*, limpieza de texto, y normalización, fundamentales para un rendimiento adecuado de los sistemas RAG.
- **Interfaz RESTful y componentes para producción:** incluye un servidor de inferencia, monitorización con Prometheus y un cliente web básico, lo que permite desplegar prototipos y productos funcionales de manera eficiente.

Aunque Haystack también soporta arquitecturas RAG completas, en el contexto de este proyecto no se ha utilizado directamente. No obstante, su análisis resulta útil por su enfoque industrial y su madurez como framework. En comparación con LangChain, Haystack ofrece una aproximación más centrada en el despliegue final y en la integración con bases de datos documentales completas, mientras que LangChain está más orientado a la manipulación dinámica de prompts y contextos.

5.2.3 LlamaIndex

LlamaIndex (anteriormente conocido como GPT Index) es una herramienta especializada en la indexación, estructuración y consulta de grandes volúmenes de información textual para su integración con modelos de lenguaje. A diferencia de otros frameworks que se centran en la orquestación general de sistemas RAG, LlamaIndex está diseñado específicamente para facilitar la interacción eficiente entre documentos complejos y LLMs, priorizando la calidad de la recuperación y la flexibilidad en la representación de la información.

Una de sus principales fortalezas es su capacidad para construir índices personalizados a partir de diversas fuentes de datos (archivos, bases de datos, APIs, etc.) y aplicar distintas estrategias de segmentación, resumen y agrupamiento. Además, permite optimizar el proceso de recuperación mediante estructuras como árboles de decisión, índices jerárquicos o esquemas vectoriales híbridos.

Entre sus características más destacadas se encuentran:

- **Indexación estructurada avanzada:** permite construir índices tipo lista, árbol, resumen, grafo o combinaciones de estos, adaptándose a distintos tipos de corpus y consultas.
- **Soporte nativo para múltiples fuentes de datos:** incluye conectores para documentos locales, bases SQL, NoSQL, APIs externas, herramientas de *scraping*, etc.
- **Módulos de preprocesamiento e integración semántica:** permite aplicar resúmenes parciales, anotaciones o generación de nodos intermedios antes de la indexación.
- **Compatibilidad con múltiples bases vectoriales:** soporta integración con FAISS, Milvus, Weaviate, entre otras.
- **Control detallado sobre la generación de *prompts*:** ofrece herramientas para crear plantillas condicionales y ajustar dinámicamente los fragmentos documentales que se presentan al modelo.

LlamaIndex es especialmente útil en proyectos donde los documentos no se limitan a ser fragmentos planos, sino que presentan una estructura semántica o jerárquica compleja. En ese contexto, su aproximación basada en índices enriquecidos permite mejorar la relevancia de la recuperación y la coherencia de las respuestas generadas por el LLM.

Aunque en este proyecto no se ha empleado directamente, LlamaIndex representa una alternativa muy sólida cuando se requiere alta calidad en la recuperación, especialmente en dominios donde los documentos presentan formatos diversos, niveles de profundidad o conexiones semánticas entre secciones.

5.3 Bases de datos vectoriales

5.3.1 FAISS (Facebook AI Similarity Search)

FAISS es una biblioteca desarrollada por Facebook AI Research (FAIR), diseñada específicamente para realizar búsquedas de similitud en grandes volúmenes de datos vectoriales de

manera rápida y eficiente. Se ha consolidado como una de las soluciones más utilizadas en tareas de recuperación semántica, especialmente en arquitecturas basadas en RAG, gracias a su alto rendimiento, flexibilidad y amplia adopción en entornos de producción.

El principal objetivo de FAISS es resolver el problema conocido como *Approximate Nearest Neighbor Search* (ANN), es decir, la búsqueda eficiente de los vectores más cercanos a uno dado dentro de un espacio de alta dimensión. Para ello, ofrece múltiples algoritmos e índices que permiten ajustar el equilibrio entre precisión, velocidad y consumo de recursos, tanto en CPU como en GPU.

Características principales

- **Alto rendimiento en búsquedas densas:** optimizado para entornos de alto volumen, con soporte completo tanto para CPU como para GPU, lo que permite escalar a millones de vectores con baja latencia.
- **Índices configurables:** ofrece diversos tipos de índices (Flat, IVF, HNSW, PQ, entre otros) que pueden combinarse entre sí para adaptar el comportamiento del sistema a las necesidades del caso de uso.
- **Madurez y fiabilidad:** es un proyecto de código abierto con soporte activo, documentación extensa y probado en aplicaciones industriales de gran escala.
- **Compatibilidad con frameworks RAG:** se integra fácilmente con herramientas como LangChain, LlamaIndex y Haystack, lo que facilita su integración dentro de arquitecturas modernas de IA generativa.
- **Persistencia de índices:** permite almacenar los índices en disco y recargarlos en tiempo de ejecución, reduciendo los tiempos de arranque y simplificando la gestión del sistema.

Ventajas en contextos RAG FAISS resulta especialmente adecuado en escenarios donde se requiere una recuperación de fragmentos precisa y de bajo coste computacional. Su rendimiento optimizado y la capacidad de ajuste fino de los índices lo convierten en una solución ideal cuando se necesita control sobre el comportamiento interno del motor de búsqueda. Además, su integración con frameworks populares y su naturaleza open source lo hacen atractivo para proyectos que operan sobre infraestructura propia o en entornos locales con recursos dedicados (como servidores con GPU).

Limitaciones A pesar de sus ventajas, FAISS presenta algunas limitaciones relevantes:

- **Gestión manual:** requiere configuración explícita para tareas como la persistencia, la incorporación o eliminación de nuevos vectores, y el tratamiento de metadatos.
- **Ausencia de capa de abstracción avanzada:** a diferencia de soluciones como Weaviate, FAISS no incorpora por defecto funcionalidades para estructuración semántica, esquemas enriquecidos o consultas mediante lenguaje declarativo, lo que puede requerir componentes adicionales en la arquitectura para cubrir estas funciones.

5.3.2 Milvus

Milvus es una base de datos vectorial de código abierto diseñada específicamente para gestionar de forma eficiente grandes volúmenes de vectores en entornos distribuidos. A diferencia de soluciones más ligeras como FAISS, Milvus proporciona una infraestructura completa que incluye almacenamiento persistente, gestión de metadatos, servicios de indexación paralela y ejecución escalable en clústeres.

Su arquitectura está pensada para soportar operaciones intensivas de búsqueda semántica en tiempo real, incluso cuando el número de vectores se encuentra en el rango de millones o miles de millones. Este diseño hace de Milvus una opción especialmente adecuada para entornos empresariales con altos requisitos de rendimiento y fiabilidad.

Características principales

- **Alta escalabilidad horizontal:** permite distribuir el almacenamiento y las consultas a lo largo de múltiples nodos, garantizando rendimiento constante con grandes volúmenes de datos.
- **Gestión nativa de metadatos:** cada vector puede asociarse a información estructurada, permitiendo realizar consultas filtradas y segmentadas más allá de la similitud semántica.
- **Soporte para múltiples índices:** entre ellos IVF, HNSW, ANNOY y Flat, que pueden configurarse según las necesidades de precisión y latencia.
- **Integración con frameworks RAG:** soporta conexión directa con LangChain, Haystack y LlamaIndex mediante adaptadores nativos.
- **API REST y SDKs multiplataforma:** disponible para lenguajes como Python, Java o Go, lo que facilita su uso en sistemas basados en microservicios.

Ventajas en contextos RAG Milvus destaca por su capacidad de mantener el rendimiento incluso bajo cargas de trabajo intensas y persistentes. Su infraestructura orientada a producción permite mantener bases documentales vivas, actualizables y fácilmente consultables. Esta combinación lo convierte en una solución robusta y adecuada para sectores como el asegurador, donde se requiere eficiencia, trazabilidad y control sobre grandes volúmenes de documentos.

Limitaciones

- **Complejidad en la infraestructura:** su despliegue inicial puede requerir una mayor planificación técnica, especialmente en entornos locales.
- **Mantenimiento más exigente:** requiere supervisión continua, especialmente cuando se opera en clústeres distribuidos o con alta disponibilidad.

5.3.3 Weaviate

Weaviate es una base de datos vectorial de código abierto que combina un motor de búsqueda semántica con una capa de gestión de datos estructurados. A diferencia de otras soluciones más centradas en el rendimiento puro (como FAISS), Weaviate adopta un enfoque de plataforma completa, orientado a ofrecer funcionalidades semánticas avanzadas y flexibilidad en el modelado de datos.

Su modelo de datos se basa en esquemas definidos por clases y propiedades, lo que permite representar relaciones complejas entre documentos y realizar búsquedas que combinan semántica y lógica estructurada. Además, cuenta con capacidades de vectorización automática e integración directa con modelos preentrenados, lo que reduce la complejidad del proceso de carga de datos.

Características principales

- **Modelo de datos estructurado:** permite definir esquemas semánticos mediante clases, relaciones y metadatos, habilitando consultas más expresivas.
- **Vectorización automática:** incorpora módulos de conexión con modelos como OpenAI, Cohere o Hugging Face para vectorizar texto automáticamente al insertarlo.
- **API basada en GraphQL:** facilita búsquedas que combinan contenido semántico con lógica declarativa estructurada.
- **Arquitectura modular extensible:** incluye módulos opcionales para autenticación, clasificación, filtros espaciales, control de acceso, etc.
- **Interfaz gráfica de administración:** incluye una UI que permite gestionar esquemas, visualizar vectores y probar consultas desde el navegador.

Ventajas en contextos RAG Weaviate es especialmente útil en sistemas donde se requiere una integración estrecha entre información semántica y estructura de datos. Su orientación declarativa, su API intuitiva y su capacidad para reducir la fricción en el proceso de ingestión y consulta hacen que sea una opción adecuada para entornos con necesidades complejas de recuperación y representación.

Además, su escalabilidad horizontal y soporte para entornos distribuidos permiten su uso tanto en prototipos como en despliegues empresariales.

Limitaciones

- **Mayor consumo de recursos:** su arquitectura completa lo hace más exigente en comparación con soluciones más ligeras.
- **Curva de aprendizaje más pronunciada:** requiere modelado correcto del esquema para aprovechar su potencia.
- **Rendimiento sensible a la configuración:** activar muchos módulos sin planificación puede degradar el rendimiento si no se optimiza el despliegue.

5.4 Modelos de lenguaje generativo (LLMs)

Los modelos de lenguaje generativo constituyen el componente encargado de producir la respuesta final en una arquitectura RAG, a partir del contexto recuperado por el sistema. Aunque ya se ha analizado su papel en el flujo general del sistema, en esta sección se revisan algunas de las principales opciones disponibles actualmente, comparando sus capacidades, licencias, eficiencia y facilidad de integración.

Se incluyen tanto modelos propietarios como opciones de código abierto, valorando aspectos como la calidad de generación, la sensibilidad al contexto, el soporte multilingüe o su adecuación a entornos locales. También se considera el modelo empleado en este proyecto, **Nous-Hermes-2**, que ha demostrado un buen equilibrio entre rendimiento, precisión y coste computacional en tareas generativas condicionadas por contexto documental.

5.4.1 GPT (Generative Pre-trained Transformer)

GPT es una familia de modelos desarrollada por OpenAI, ampliamente utilizada como estándar en tareas de lenguaje natural gracias a su elevada capacidad de generación, comprensión contextual y razonamiento. Actualmente, la versión más avanzada es GPT-4, disponible a través de servicios en la nube como OpenAI o Azure.

En arquitecturas RAG, GPT se utiliza habitualmente como modelo generador, siendo capaz de procesar prompts enriquecidos con contexto documental y producir respuestas detalladas y coherentes. Su rendimiento mejora significativamente cuando se combina con técnicas de *prompt engineering*, inyección dinámica de contexto y gestión del historial conversacional.

Ventajas

- Alta calidad en la generación de texto en múltiples idiomas.
- Muy buen rendimiento en tareas de QA, resumen y diálogo.
- Acceso sencillo mediante API bien documentada.

Limitaciones

- Modelo propietario con costes asociados por token.
- Requiere conexión a servicios externos y gestión de claves.
- Dependencia de terceros, con implicaciones en privacidad y latencia.

5.4.2 LLaMA (Large Language Model Meta AI)

LLaMA es una familia de modelos desarrollada por Meta como alternativa de código abierto a los modelos propietarios existentes. Su diseño se orienta a la eficiencia y a la posibilidad de ejecución local, lo que permite desplegar sistemas generativos sin depender de servicios externos ni comprometer la privacidad de los datos.

La versión más reciente, LLaMA 2, incluye modelos de 7, 13 y 70 mil millones de parámetros. Su popularidad ha crecido rápidamente gracias a la facilidad de integración en entornos como Hugging Face, llama.cpp o LangChain, así como a la existencia de numerosas variantes adaptadas a distintos dominios.

Ventajas

- Código abierto, ejecutable en entornos locales sin conexión externa.
- Buena calidad generativa, especialmente en versiones ajustadas por comunidad.
- Permite mayor control y personalización del sistema.

Limitaciones

- Requiere GPU potente para ejecutar modelos de gran tamaño.
- Su rendimiento base puede estar por debajo de GPT-4 en tareas complejas.
- Ajustar el modelo al dominio requiere conocimientos técnicos adicionales.

5.4.3 Flan-T5

Flan-T5 es una variante del modelo T5 desarrollada por Google, optimizada mediante entrenamiento multitarea con instrucciones. Su arquitectura encoder-decoder lo hace especialmente adecuado para tareas supervisadas como resumen, clasificación o traducción, aunque también puede emplearse en generación libre con prompts bien estructurados.

Está disponible en distintos tamaños, lo que permite su uso tanto en entornos ligeros como en infraestructuras de mayor capacidad. Gracias a su licencia abierta y a su soporte en Hugging Face, Flan-T5 se ha consolidado como una alternativa eficiente y accesible para integrar en sistemas RAG.

Ventajas

- Modelo de código abierto, disponible en múltiples tamaños.
- Arquitectura eficiente y adaptada a tareas supervisadas.
- Buen rendimiento en generación a partir de contexto bien definido.

Limitaciones

- Capacidad limitada en tareas de razonamiento complejo o conversacional.
- Requiere diseño de prompts cuidados para alcanzar precisión aceptable.
- No mantiene memoria de contexto entre mensajes sucesivos.

5.4.4 Modelos ajustados específicamente para RAG

Además de los modelos generales, existen variantes afinadas específicamente para tareas de recuperación y generación combinadas. Estos modelos, entrenados con técnicas de *instruction tuning* y corpus específicos, están optimizados para generar respuestas coherentes y fieles al contexto proporcionado por el sistema RAG.

Su diseño permite aprovechar mejor los fragmentos recuperados desde la base vectorial, minimizando desviaciones temáticas y alucinaciones. Algunos ejemplos destacables incluyen modelos como **Nous-Hermes-2**, **Mistral-Instruct**, **Cohere Command R+** o **Vicuna-Instruct**, todos ellos diseñados para ofrecer una mayor fidelidad factual y capacidad de razonamiento sobre documentos aportados en el prompt.

Ventajas

- Mayor adherencia al contexto proporcionado.
- Mejores resultados en tareas específicas de recuperación + generación.
- Reducción de alucinaciones y respuestas irrelevantes.

Limitaciones

- Algunos modelos aún carecen de validación extensiva en producción.
- Pueden requerir ajuste si el dominio es muy específico.
- Su rendimiento depende en gran medida de la calidad del contexto recuperado.

5.5 Modelos de embeddings

Los modelos de embeddings utilizados en sistemas RAG pueden agruparse en diferentes familias, en función de su arquitectura y del enfoque de entrenamiento. A continuación se comparan dos de las más relevantes: los modelos tipo Sentence-BERT y los modelos INSTRUCTOR, destacando sus principales características, ventajas y limitaciones desde un punto de vista práctico.

Sentence-BERT (SBERT) Esta familia adapta modelos BERT para generar representaciones vectoriales a nivel de frase o párrafo, utilizando redes siamesas o tripletas y entrenamiento contrastivo. Se han convertido en el estándar de facto en tareas de recuperación semántica simétrica.

- **Ventajas**
 - Alta eficiencia en inferencia, especialmente en modelos compactos.
 - Facilidad de integración en librerías como **sentence-transformers**.
 - Amplia disponibilidad de variantes preentrenadas y ajustadas a distintos dominios.

■ Limitaciones

- Menor rendimiento en tareas de recuperación asimétrica (consulta breve vs documento largo).
- Sensibilidad al cambio de dominio si no se ha ajustado previamente.

INSTRUCTOR Los modelos INSTRUCTOR amplían el enfoque tradicional incluyendo una instrucción textual explícita que describe la tarea. Esto les permite abordar múltiples tareas con un solo modelo, mejorando la generalización y el rendimiento en contextos diversos.

■ Ventajas

- Mejor rendimiento en recuperación asimétrica y tareas heterogéneas.
- Capacidad de especialización sin necesidad de reentrenar el modelo, simplemente cambiando la instrucción.

■ Limitaciones

- Mayor consumo de recursos debido al uso de instrucciones largas.
- Menor número de versiones ligeras y cuantizadas disponibles.
- Dependencia del idioma inglés en las instrucciones para lograr el rendimiento óptimo.

Resumen y elección en este proyecto El modelo seleccionado, `intfloat/e5-large-v2`, se basa en la arquitectura Sentence-BERT y ofrece un buen equilibrio entre precisión, eficiencia y facilidad de despliegue local. Aunque los modelos INSTRUCTOR ofrecen un rendimiento superior en algunos benchmarks, su mayor complejidad operativa y dependencia del idioma inglés han llevado a priorizar una solución más simple y robusta para esta primera fase del sistema. En futuras versiones, se podrá considerar su incorporación si se requieren tareas más variadas o mayor adaptabilidad semántica.

5.6 Conclusiones sobre soluciones

A lo largo de este capítulo se han analizado las principales tecnologías implicadas en la implementación de arquitecturas RAG: frameworks de integración, bases de datos vectoriales, modelos generativos, y modelos de embeddings. Esta revisión ha permitido identificar el conjunto de herramientas más relevantes del ecosistema actual, evaluando sus ventajas, limitaciones y niveles de madurez.

Una de las principales conclusiones es que no existe una única solución óptima, sino que la elección de cada componente debe realizarse en función de las características del caso de uso, los requisitos técnicos, el dominio de aplicación y la infraestructura disponible. Modelos de lenguaje como GPT ofrecen un alto rendimiento con bajo esfuerzo de integración, mientras que soluciones open source como LLaMA o Flan-T5 permiten mayor control y adaptabilidad. Lo mismo ocurre

con las bases vectoriales y modelos de embeddings, donde opciones como FAISS o Sentence-BERT son eficaces en entornos locales, mientras que herramientas como Weaviate o Instructor ofrecen mayores capacidades semánticas y estructurales.

Este análisis comparativo sienta las bases para justificar las decisiones tecnológicas adoptadas en el desarrollo del sistema propuesto, que se detallarán en el próximo capítulo. La elección final de herramientas se ha realizado atendiendo al equilibrio entre rendimiento, escalabilidad, privacidad, flexibilidad y facilidad de integración en un entorno real.

Análisis

El presente capítulo expone el análisis detallado de la solución propuesta para implementar un sistema de recuperación aumentada con generación (RAG) aplicado al sector asegurador. Tras la revisión exhaustiva de las tecnologías disponibles presentada en el capítulo anterior, se ha diseñado una arquitectura modular orientada a maximizar la precisión, la trazabilidad y la escalabilidad del sistema, al mismo tiempo que se garantiza su viabilidad técnica en un entorno real.

La solución se estructura en torno a cuatro componentes principales: un motor de recuperación semántica basado en embeddings, una base vectorial persistente, un modelo de lenguaje generativo adaptado al dominio, y un backend orquestador responsable de coordinar las operaciones entre los distintos módulos. A lo largo de este capítulo se justifican las elecciones tecnológicas realizadas en cada uno de estos elementos, atendiendo a criterios de rendimiento, compatibilidad, flexibilidad y adecuación a los requisitos del dominio.

También se analizan aspectos como la estructura y formato de los documentos aseguradores, las características semánticas del corpus, y las necesidades específicas del usuario final. Estos elementos condicionan tanto la estrategia de preprocesamiento como el diseño de los flujos de recuperación y generación que conforman el núcleo del sistema.

6.1 Requisitos técnicos y funcionales

El sistema propuesto tiene como objetivo principal facilitar el acceso eficiente, preciso y contextualizado a la información contenida en documentos aseguradores. Para ello, se ha definido un conjunto de requisitos funcionales y técnicos que guían tanto el diseño arquitectónico como la elección de tecnologías.

Requisitos funcionales

Desde el punto de vista funcional, el sistema debe:

-
- Permitir al usuario final realizar consultas basadas en documentos mediante preguntas formuladas en lenguaje natural.
 - Proporcionar respuestas generadas a partir de fragmentos documentales reales, relevantes y verificables.
 - Ofrecer trazabilidad de cada respuesta, indicando su procedencia dentro del corpus documental.
 - Soportar la carga de nuevos documentos, organizados por expedientes, sin necesidad de reiniciar el sistema.
 - Mantener un historial conversacional coherente entre las interacciones de un mismo chat.
 - Proporcionar una interfaz usable, clara y accesible, tanto para personal técnico como no técnico.

Requisitos técnicos y no funcionales

Desde un enfoque de ingeniería del software, se establecen los siguientes requisitos no funcionales:

- **Escalabilidad:** capacidad de crecimiento en volumen de datos y número de usuarios sin degradación significativa del rendimiento.
- **Latencia aceptable:** tiempo de respuesta inferior a 3 segundos incluso en escenarios con elevada carga documental.
- **Modularidad:** los componentes deben estar desacoplados y ser sustituibles sin afectar al sistema completo.
- **Mantenibilidad:** facilidad para incorporar nuevas tecnologías, actualizar modelos o realizar ajustes parciales.
- **Privacidad y control de datos:** posibilidad de operar sin depender de servicios externos, garantizando el tratamiento local de la información.
- **Compatibilidad con GPU:** aprovechamiento de aceleración hardware para tareas computacionalmente intensivas.

Estos requisitos derivan tanto de necesidades técnicas como del dominio asegurador, que impone restricciones específicas en cuanto a trazabilidad, privacidad y control del conocimiento. Son también la base sobre la que se ha construido la arquitectura final del sistema, tal como se detallará en las siguientes secciones.

Casos de uso del sistema

Para complementar la definición de requisitos funcionales, se ha elaborado un diagrama de casos de uso que resume de forma gráfica las principales interacciones que puede realizar un usuario con el sistema. Estas operaciones cubren todo el ciclo funcional, desde la autenticación y gestión de expedientes, hasta la carga de documentos y realización de consultas contextualizadas sobre el corpus asegurador. También se contempla un actor secundario —el administrador— con capacidades de auditoría y análisis en entornos de desarrollo.

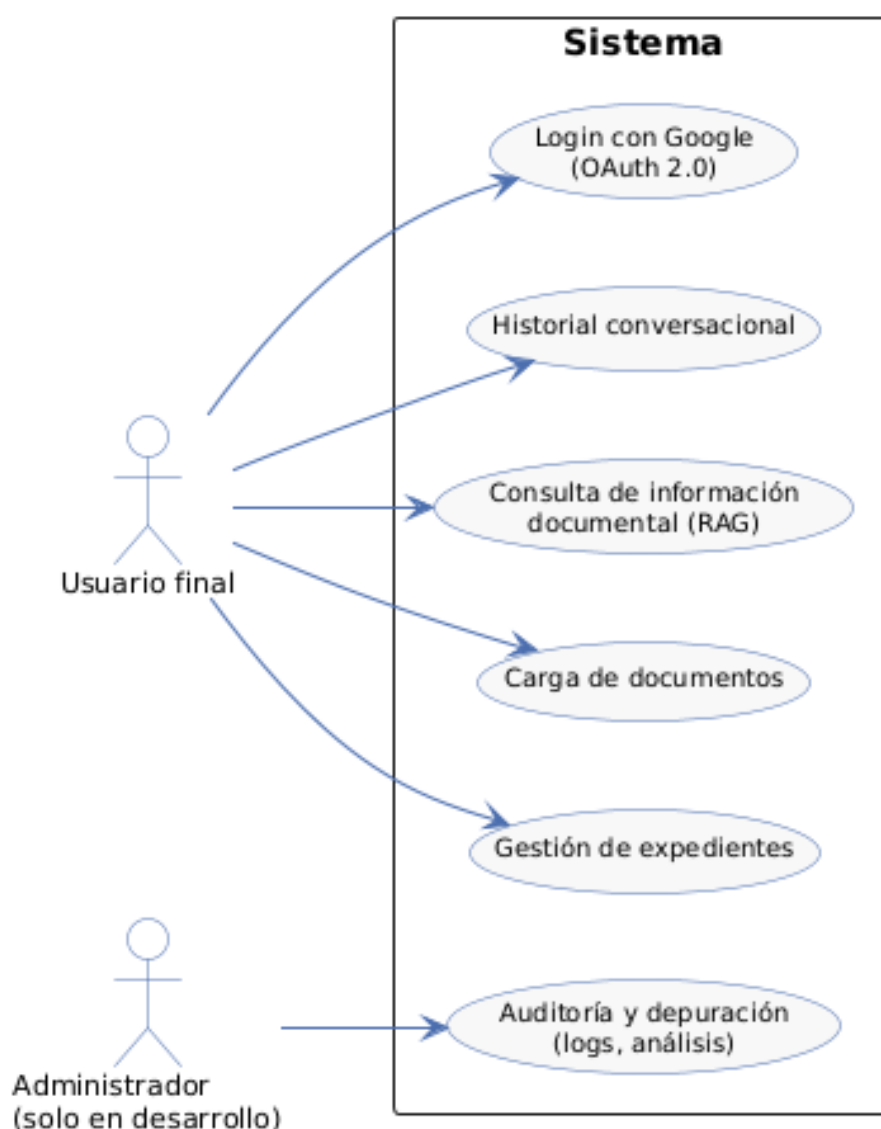


Figura 6.1: Diagrama de casos de uso del sistema. El actor principal —el usuario final— puede iniciar sesión mediante OAuth, gestionar sus expedientes, subir documentos y realizar consultas en lenguaje natural sobre el corpus documental. Se incluye también un actor administrador con capacidades de depuración y análisis para entornos de desarrollo.

6.2 Motivación del diseño arquitectónico

La arquitectura propuesta para este proyecto parte de la necesidad de ofrecer una solución flexible, escalable y fácilmente mantenible, capaz de adaptarse a las particularidades del sector asegurador y a la evolución constante del ecosistema tecnológico.

Frente a enfoques monolíticos o basados en modelos preentrenados estáticos, se opta por una arquitectura **modular y desacoplada** que permita aislar responsabilidades, optimizar cada componente de forma independiente y facilitar la evolución del sistema con el mínimo impacto global.

Este tipo de arquitectura encaja de forma natural con los principios de diseño de los sistemas RAG, donde las fases de recuperación, generación, almacenamiento y control están separadas y pueden mejorarse por separado. Además, se alinea con buenas prácticas de ingeniería de software como separación de responsabilidades, control de versiones, pruebas modulares y despliegue flexible.

En contextos donde se maneja información crítica o sensible —como el dominio asegurador—, esta separación permite:

- Reemplazar el modelo generativo sin reindexar el corpus.
- Cambiar el motor vectorial sin alterar el comportamiento conversacional.
- Actualizar los embeddings con un nuevo modelo sin comprometer el resto del sistema.

Asimismo, el diseño modular permite incorporar componentes adicionales como sistemas de auditoría, control de acceso, trazabilidad de contexto, y mecanismos de feedback, que son clave en entornos profesionales regulados.

Por todo ello, la arquitectura basada en recuperación aumentada con generación se considera la aproximación más adecuada para cumplir los requisitos definidos previamente, manteniendo un alto grado de control, escalabilidad y adaptabilidad.

6.3 Análisis del flujo de consulta

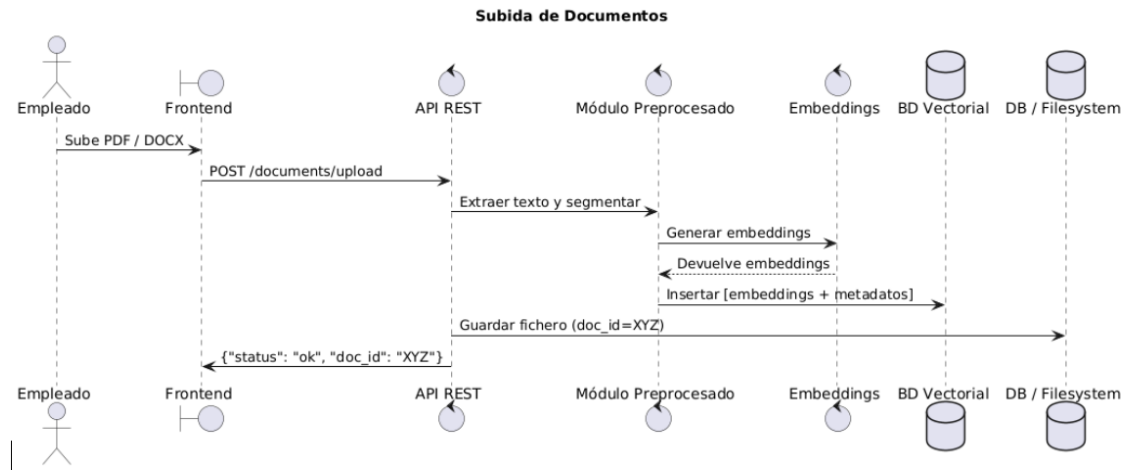


Figura 6.2: Diagrama de secuencia del flujo de *ingestión*: subida, preprocesado y persistencia de documentos.

Descripción de la Figura 6.2. El diagrama detalla el **pipeline de ingestión**. El proceso se inicia cuando el empleado sube un PDF o DOCX desde la interfaz web. La petición `POST /documents/upload` llega a la API REST, que delega en el módulo de preprocesado la *extracción* del texto y su *segmentación* en *chunks*. Cada fragmento se envía al microservicio de embeddings, que devuelve sus representaciones vectoriales. Finalmente, los pares *[embedding + metadatos]* se insertan en la base de datos vectorial, mientras que el fichero original se almacena en un repositorio persistente (base SQL o sistema de archivos). De esta forma, la arquitectura desacopla claramente la capa de almacenamiento semántico de la capa documental.

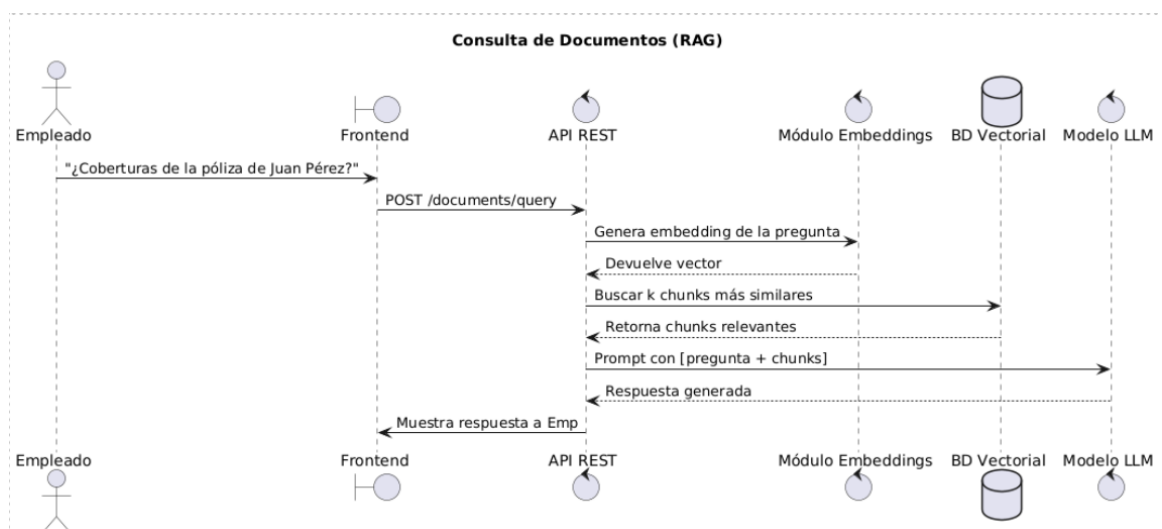


Figura 6.3: Diagrama de secuencia del flujo de *consulta*: recuperación aumentada de información (RAG).

Descripción de la Figura 6.3. El diagrama ilustra el **circuito completo de una consulta RAG**. Cuando el usuario envía una pregunta, la API genera su embedding y lo compara con los vectores almacenados, recuperando los k fragmentos más relevantes. Estos fragmentos se concatenan con la consulta para construir el *prompt* que consume el modelo LLM. La respuesta generada se devuelve al frontend, garantizando que el modelo dispone de un contexto documental *preciso, fresco y justificable*. La separación entre las fases de recuperación y generación reduce la alucinación y mejora la trazabilidad de las respuestas.

El flujo de consulta en una arquitectura RAG recoge, por tanto, el recorrido completo que sigue una petición del usuario —desde su entrada en el sistema hasta la entrega de la respuesta— asegurando la coherencia entre las fases de recuperación y generación. Aunque la implementación concreta puede variar, la mayoría de sistemas RAG comparten la secuencia de pasos resumida a continuación:

1. **Recepción de la consulta:** el usuario introduce una pregunta en lenguaje natural, sin necesidad de conocer la estructura documental ni utilizar filtros avanzados.
2. **Vectorización de la pregunta:** la consulta se transforma en una representación numérica (*embedding*) mediante un modelo semántico previamente entrenado. Este vector captura el significado de la pregunta y será utilizado para recuperar fragmentos conceptualmente similares.
3. **Búsqueda en la base vectorial:** el sistema consulta la base de datos vectorial para recuperar los k fragmentos más cercanos al vector de la pregunta, utilizando medidas de similitud como la distancia coseno o L2.
4. **Construcción del *prompt*:** los fragmentos recuperados se concatenan con la consulta original siguiendo una plantilla definida. El objetivo es proporcionar al modelo de lenguaje un contexto lo suficientemente rico como para generar una respuesta precisa.
5. **Generación de la respuesta:** el modelo LLM procesa el *prompt* completo y devuelve una respuesta en lenguaje natural. Esta respuesta puede incluir explicaciones, justificaciones o referencias al contenido documental.
6. **Entrega al usuario:** la respuesta generada se presenta al usuario a través de una interfaz que puede incluir referencias al fragmento utilizado, su ubicación en el corpus o enlaces a los documentos originales.

Este flujo puede ampliarse con mecanismos adicionales —cachés de embeddings, *ranking* postrecuperación, anotaciones semánticas o validaciones de consistencia—, pero su núcleo permanece anclado a la secuencia recuperación-generación sobre la que se sustentan las decisiones arquitectónicas y tecnológicas del sistema.

6.4 Estructura documental del dominio asegurador

Uno de los factores clave en el diseño de un sistema RAG aplicado al sector asegurador es la naturaleza de los documentos con los que debe trabajar. Estos documentos condicionan el preprocesamiento, la segmentación, la recuperación semántica y la generación de respuestas precisas.

En este proyecto, el sistema está diseñado para gestionar **expedientes aseguradores**, que agrupan diferentes tipos de documentos relevantes para un caso o cliente específico. Si bien pueden estar vinculados a usuarios concretos, el sistema no impone ninguna restricción en ese sentido, permitiendo incluir documentación de carácter general o transversal.

Los expedientes pueden contener, entre otros:

- Pólizas de seguro (generales y particulares).
- Reclamaciones y formularios de siniestros.
- Contratos firmados y condiciones contractuales.
- Comunicaciones entre aseguradora y cliente.
- Informes periciales, resoluciones o dictámenes.
- Notificaciones internas o circulares técnicas.

Estos documentos presentan características particulares desde el punto de vista del análisis de software:

- **Estructura jerárquica y densa:** organizados en secciones, artículos o cláusulas que dificultan una segmentación lineal.
- **Lenguaje técnico-legal:** requiere modelos semánticos capaces de interpretar términos específicos del sector.
- **Contexto disperso:** la información relevante para una consulta puede estar fragmentada en diferentes documentos o apartados.
- **Formato no estructurado:** suelen encontrarse en PDF, Word u otros formatos sin marcadores semánticos claros.

Desde un enfoque técnico, esto implica que:

- El sistema debe aplicar una segmentación que respete los límites semánticos y mantenga trazabilidad documental.
- La granularidad de los fragmentos debe equilibrar precisión y contexto, sin superar el límite de tokens del modelo.

-
- Cada fragmento debe estar asociado a metadatos que identifiquen su expediente, origen, tipo de documento y ubicación exacta.

Estas consideraciones influyen de forma directa en la arquitectura del sistema, en el diseño del *pipeline* de ingestión documental, y en la lógica de recuperación. El sistema debe adaptarse a esta realidad para garantizar tanto la precisión de las respuestas como la fiabilidad y auditabilidad de las fuentes.

6.5 Criterios de elección tecnológica

Una vez definidos los requisitos funcionales y técnicos del sistema, y teniendo en cuenta la naturaleza del dominio asegurador, se han establecido una serie de criterios que guían la selección de tecnologías para cada uno de los componentes de la arquitectura RAG.

Los criterios principales aplicados han sido los siguientes:

- **Compatibilidad con arquitecturas modulares:** los componentes deben poder integrarse de forma desacoplada, permitiendo su sustitución sin afectar al sistema completo.
- **Capacidad de ejecución local:** por motivos de privacidad y control, se priorizan tecnologías que permitan un despliegue completo sin depender de servicios externos.
- **Buen soporte para el español:** dado que los documentos y consultas están en castellano, se descartan modelos o herramientas optimizadas exclusivamente para inglés.
- **Flexibilidad en el preprocesamiento e integración de datos:** se requiere manejar documentos en múltiples formatos (PDF, Word, texto plano, html, eml), con posibilidad de organización por expedientes.
- **Calidad en la recuperación semántica y generación:** se priorizan modelos y motores que presenten buen rendimiento en tareas de recuperación y contextualización bajo arquitecturas RAG.

A partir de estos criterios, se han seleccionado las siguientes tecnologías base:

- **FAISS** como motor de base vectorial, por su alta eficiencia, posibilidad de ejecución en GPU y buena integración con frameworks modernos.
- **LangChain** como framework de orquestación de contexto, por su modularidad, comunidad activa y soporte avanzado para ingeniería de *prompts*.
- **Nous-Hermes-2** como modelo de lenguaje generativo, por su equilibrio entre precisión, fluidez en español y capacidad de ejecución local (basado en LLaMA 2).
- **Sentence-BERT** e **Instructor** como opciones de modelos de embeddings, con capacidad de representar adecuadamente lenguaje técnico y legal.

-
- **Backend en FastAPI(Python)** y almacenamiento documental en PostgreSQL, por su robustez, extensibilidad y compatibilidad con sistemas empresariales.

La combinación de estas tecnologías responde tanto a los objetivos técnicos definidos como a las restricciones operativas del entorno asegurador, ofreciendo un equilibrio entre precisión, control, trazabilidad y rendimiento.

Conclusión del análisis

El análisis realizado en este capítulo ha permitido establecer las bases conceptuales y técnicas sobre las que se apoya la solución propuesta. A partir de los requisitos identificados, tanto funcionales como no funcionales, se ha justificado el enfoque arquitectónico basado en recuperación aumentada con generación (RAG), evidenciando su adecuación al dominio documental del sector asegurador.

Se ha analizado en detalle el flujo de consulta, identificando los componentes clave del sistema y su interacción, y se han estudiado las particularidades del corpus documental que condicionan aspectos como la segmentación, la vectorización y la trazabilidad de la información. Todo ello ha conducido a la identificación de un conjunto de tecnologías que, de forma conjunta, permiten construir un sistema escalable, modular, preciso y auditable.

Este razonamiento técnico da paso al capítulo siguiente, en el que se describe con mayor nivel de detalle la arquitectura concreta implementada, así como la estructura interna del sistema, su modelo de datos, los componentes funcionales y las decisiones específicas adoptadas durante el desarrollo.

Diseño de la Solución

7.1 Introducción

El presente capítulo detalla el diseño lógico y funcional del sistema propuesto, el cual constituye el núcleo de este proyecto. La solución planteada tiene como objetivo integrar un sistema inteligente de recuperación aumentada de información (RAG, por sus siglas en inglés) dentro de un entorno asegurador, proporcionando consultas automatizadas, precisas y contextualizadas a partir de documentos previamente indexados.

Este diseño ha sido concebido siguiendo principios de modularidad, escalabilidad y separación de responsabilidades, lo cual permite tanto una evolución progresiva del sistema como su adaptación a diferentes contextos de uso. Para ello, se ha optado por una arquitectura distribuida basada en microservicios, desplegada en contenedores y organizada en torno a varios componentes que colaboran entre sí: un backend que gestiona las operaciones principales, un frontend que actúa como interfaz de usuario, un modelo de lenguaje generativo para la generación de respuestas, y un sistema de almacenamiento estructurado que facilita la indexación y recuperación eficiente de la información.

El diseño aquí descrito no entra en detalles específicos sobre las herramientas empleadas o los entornos de ejecución, aspectos que se desarrollan con mayor profundidad en el Capítulo 8. En cambio, este capítulo se centra en explicar las decisiones de diseño que sustentan la solución, describiendo el comportamiento general del sistema, la interacción entre sus componentes, y el flujo de información que se produce desde que un usuario realiza una consulta hasta que recibe una respuesta generada.

Este enfoque permite no solo justificar la coherencia técnica del sistema, sino también establecer una base sólida sobre la cual se apoya su posterior implementación.

7.2 Principios de diseño

El diseño del sistema no surge de forma arbitraria, sino que está guiado por una serie de principios fundamentales que han permitido construir una solución sólida, coherente y alineada con los objetivos del proyecto. En esta sección se detallan los pilares sobre los que se ha apoyado la toma de decisiones durante todo el proceso de diseño.

Modularidad y separación de responsabilidades

Uno de los principales objetivos desde el inicio fue lograr una estructura modular, en la que cada componente del sistema tuviera una función clara y bien definida. Esta separación de responsabilidades permite que el desarrollo, las pruebas y el mantenimiento puedan abordarse de forma más eficiente, ya que los distintos módulos pueden evolucionar de forma independiente. Por ejemplo, el backend se encarga de orquestar el procesamiento de las consultas, mientras que el frontend se centra en la experiencia de usuario, y el modelo de lenguaje opera como un servicio autónomo especializado en la generación de respuestas. Por otro lado, dentro del propio backend, la funcionalidad está separada en servicios independientes, que trabajan de forma conjunta, pero que son módulos que podrían desacoplarse y funcionar de forma separada en otro contexto o aplicación.

Escalabilidad y mantenibilidad

Desde un enfoque práctico, también se ha priorizado la escalabilidad. La idea era construir una solución que no solo funcionara en entornos controlados o con pocos usuarios, sino que pudiera crecer y adaptarse fácilmente si las necesidades lo requieren. Esto se ha tenido en cuenta tanto en el diseño lógico como en la arquitectura de despliegue, permitiendo, por ejemplo, escalar de forma independiente el motor de generación o la base de datos en función de la carga. A su vez, esta misma modularidad contribuye a una mayor mantenibilidad del sistema.

Interoperabilidad y desacoplamiento

Otro principio clave ha sido la interoperabilidad entre componentes. Cada módulo ha sido diseñado para comunicarse con los demás mediante interfaces bien definidas, principalmente a través de una API REST. Esto no solo permite una mejor organización interna, sino que también facilita futuras integraciones con otros sistemas externos del sector asegurador. Además, se ha buscado un desacoplamiento claro entre los servicios, lo que permite, por ejemplo, cambiar el modelo de lenguaje o el motor vectorial sin necesidad de reescribir el resto del sistema.

Claridad y trazabilidad en los flujos

Por último, se ha dado especial importancia a que el flujo de datos dentro del sistema sea claro y trazable. Desde que un usuario lanza una consulta hasta que recibe una respuesta, todos los pasos intermedios están definidos de forma explícita. Esto no solo mejora la comprensión del

sistema y su depuración, sino que resulta clave para garantizar la fiabilidad del sistema en un contexto tan sensible como el del sector asegurador.

En conjunto, estos principios han guiado todas las decisiones técnicas del diseño, permitiendo construir una solución robusta, flexible y alineada con el propósito principal del proyecto: facilitar una consulta inteligente y precisa de la información contenida en documentos aseguradores.

7.3 Arquitectura general del sistema

La solución propuesta se ha diseñado como un sistema distribuido, compuesto por múltiples servicios desplegados en contenedores que colaboran entre sí para ofrecer una experiencia de consulta automatizada y contextualizada. Esta arquitectura responde a los requisitos de modularidad, escalabilidad y claridad funcional definidos en las fases de análisis.

Componentes principales

El sistema se estructura en torno a los siguientes contenedores, cada uno con una responsabilidad clara:

- **Base de datos (PostgreSQL):** almacena expedientes, documentos, historiales conversacionales y metadatos de usuario.
- **Modelo de lenguaje (Ollama + Nous-Hermes-2):** ejecuta el LLM en modo servicio, expuesto en el host por el puerto 11434. Utiliza `runtime: nvidia` para acceder a la GPU NVIDIA A40.
- **Backend (FastAPI):** constituye el núcleo funcional del sistema. Se encarga de la autenticación de usuarios, la indexación y búsqueda semántica mediante FAISS, la generación de embeddings y la orquestación del prompt para la interacción con el modelo. El servicio se despliega en el puerto 5000 y está optimizado para aprovechar la GPU cuando está disponible.
- **Frontend (React):** interfaz de usuario que permite subir documentos, seleccionar expedientes, lanzar consultas y recibir respuestas en tiempo real.

Interacción entre componentes

1. El usuario accede a la SPA React (3000) y selecciona o crea un expediente.
2. El frontend envía la consulta al backend (5000) mediante petición REST autenticada con JWT.
3. El backend recupera los fragmentos relevantes desde FAISS, construye el *prompt* y lo envía al LLM (11434).
4. El modelo genera la respuesta; el backend la persiste y la reenvía al frontend.
5. El usuario visualiza la respuesta en la interfaz.

Despliegue contenerizado

Cada servicio se ejecuta en un contenedor independiente, encapsulando sus bibliotecas y versiones de forma reproducible. El `docker-compose` garantiza el orden de arranque (primero la base de datos, luego el backend, etc.), reserva la GPU para backend y LLM e integra volúmenes duraderos (`postgres_data`, `ollama`) que preservan los datos entre reinicios.

Autenticación OAuth 2.0 y gestión de expedientes

El sistema emplea el servicio Google Identity para la autenticación de usuarios, mediante el uso de ID tokens firmados (formato JWT). Este enfoque evita el intercambio de credenciales sensibles y simplifica la integración, ya que el frontend obtiene el ID token directamente desde Google y lo transmite al backend para su validación. Aunque no se implementa explícitamente el flujo Authorization Code con PKCE, el modelo utilizado proporciona garantías equivalentes de seguridad y protección frente a ataques de interceptación, sin necesidad de almacenar secretos de cliente.

1. El usuario pulsa «Iniciar sesión con Google» en el frontend.
2. Google devuelve un `id_token` firmado que el frontend remite al backend mediante `POST/auth/google`.
3. El backend verifica la firma del id-token utilizando las claves públicas de google, si es válido, inserta o actualiza al usuario en PostgreSQL (clave única: `email`).
4. El backend responde con los datos del usuario y el frontend lo almacena en `localStorage`, con el fin de mantener el estado de la sesión.

Al seleccionar un expediente, el frontend solicita `GET /expedientes/{id}`; el backend valida el token, comprueba la pertenencia del expediente al usuario y devuelve sus documentos y chats. Cuando el usuario crea un chat, realiza `POST /chats` indicando el `expediente_id`; el backend persiste el chat y devuelve su `chat_id`. Este mecanismo mantiene el contexto conversacional aislado por expediente, logrando trazabilidad y privacidad.

La combinación de contenerización, GPU sharing y autenticación robusta proporciona una base sólida y fácilmente escalable —preparada para migrar a orquestadores como Kubernetes o para añadir nodos en alta disponibilidad— sin alterar la lógica de negocio ni el modelo de datos.

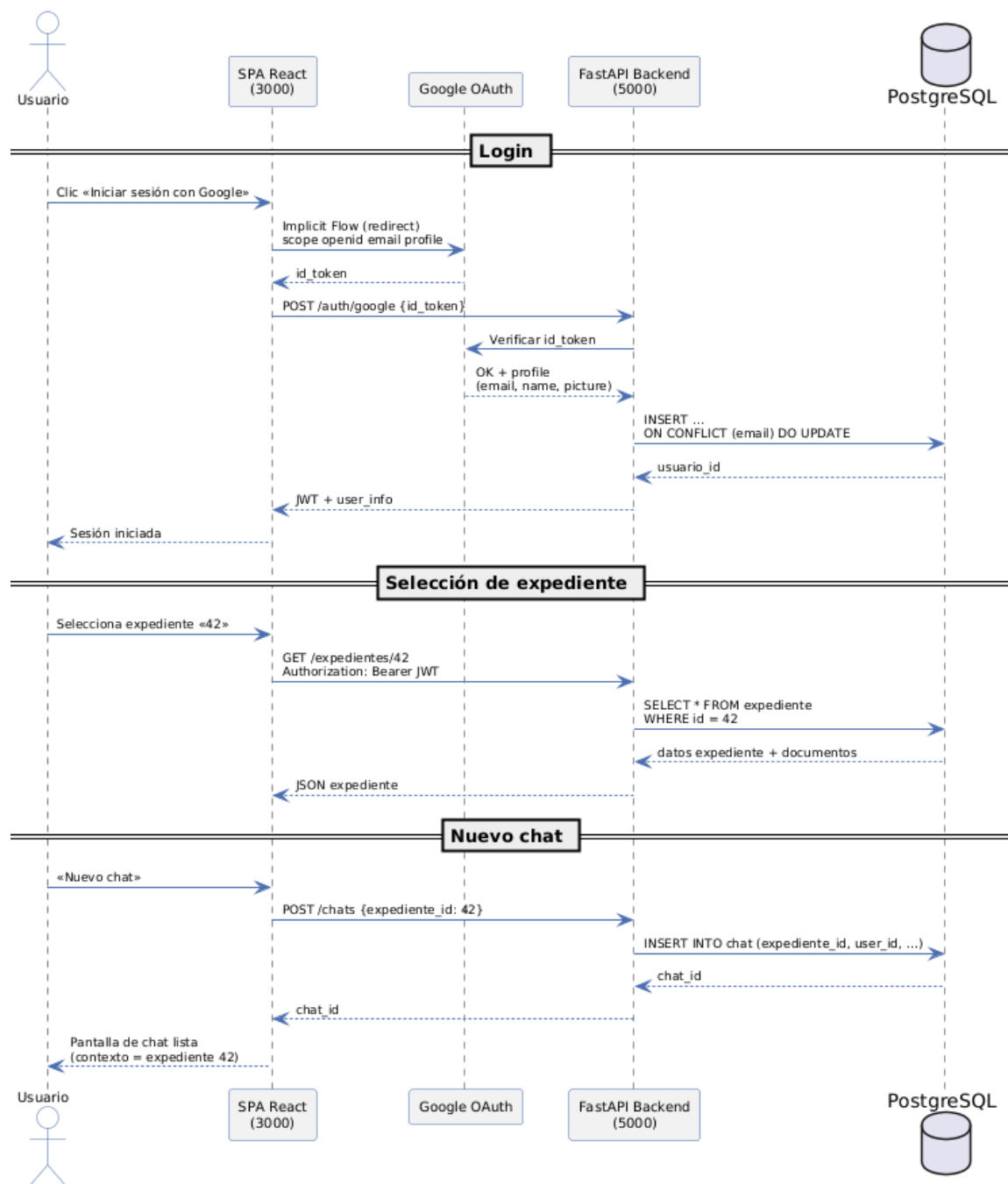


Figura 7.1: Diagrama de secuencia que representa el flujo completo de autenticación con Google OAuth, validación del `id_token`, selección de expediente y creación de un nuevo chat asociado. La interacción entre frontend, backend y base de datos permite garantizar seguridad, trazabilidad y aislamiento contextual en cada conversación.

7.4 Diseño funcional del backend

El backend constituye el núcleo lógico del sistema, siendo responsable de orquestar todo el proceso de consulta, desde la recepción de la pregunta del usuario hasta la generación final de

la respuesta. Para garantizar claridad, mantenibilidad y separación de responsabilidades, se ha dividido en cinco servicios principales, cada uno encapsulado en un módulo independiente. A continuación se describen sus funciones y relaciones.

Servicio de ingesta

Este módulo se encarga del procesamiento inicial de los documentos subidos por los usuarios. Su función principal es dividir los documentos en fragmentos adecuados para la posterior vectorización y recuperación. Además, asocia metadatos relevantes como el nombre del archivo, el expediente al que pertenece y marcas de tiempo.

También se realiza una limpieza básica del texto y una segmentación inteligente que optimiza la coherencia semántica de los fragmentos, lo cual es crucial para mejorar la calidad del proceso RAG posterior.

El siguiente diagrama ilustra el flujo completo de ingestión documental que sigue el sistema desde el momento en que el usuario sube un archivo hasta que sus contenidos han sido preprocesados, vectorizados e indexados:

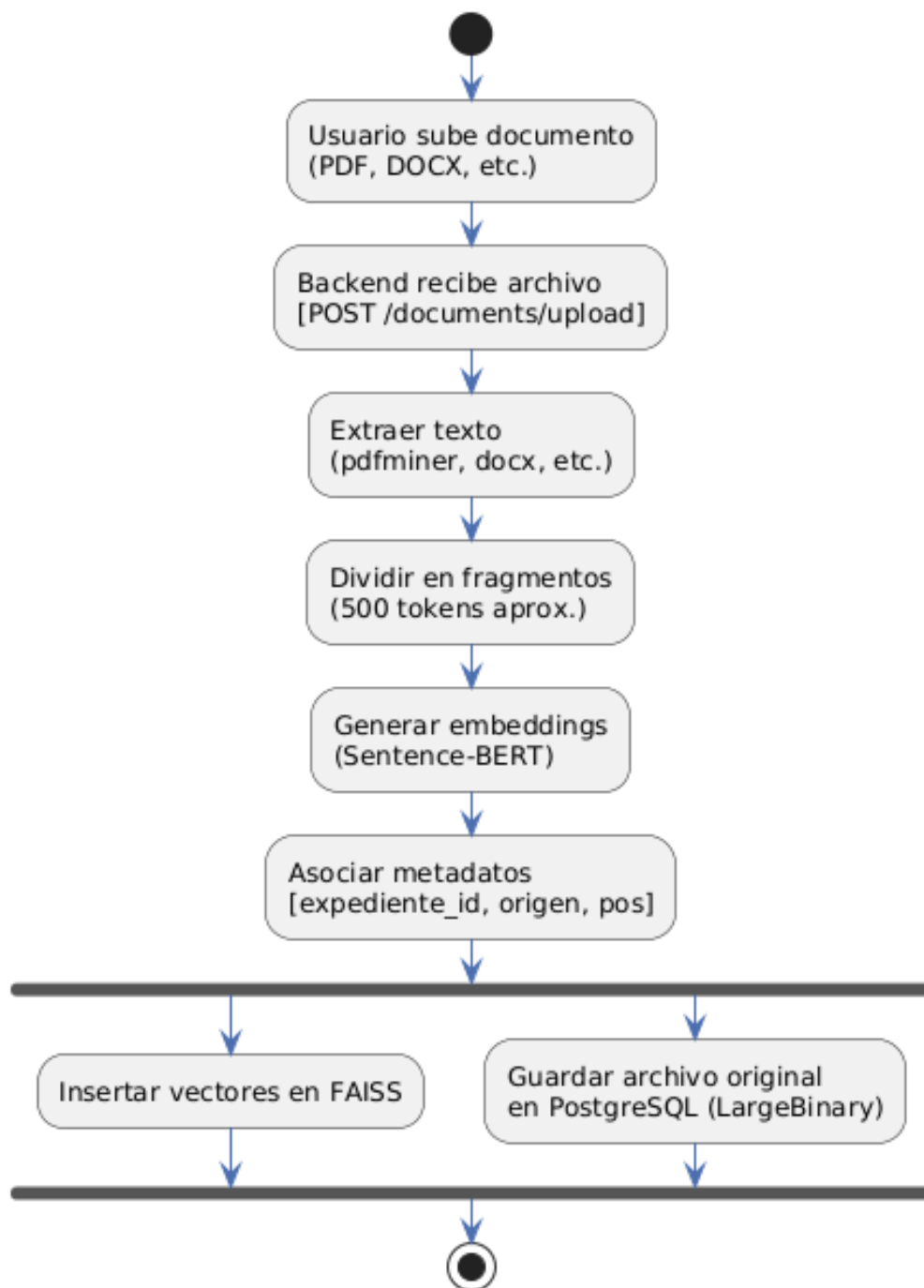


Figura 7.2: Diagrama de flujo del pipeline de ingestión documental. El backend recibe un archivo subido por el usuario, extrae su contenido textual (según el formato), lo segmenta en fragmentos de tamaño semántico manejable, genera sus embeddings con un modelo tipo Sentence-BERT, asocia los metadatos correspondientes, y distribuye el resultado en dos ramas: por un lado, indexa los vectores en FAISS para recuperación rápida, y por otro, almacena el binario original en PostgreSQL mediante un campo `LargeBinary`.

Servicio de vectorización

Una vez procesados los documentos, este servicio transforma los fragmentos en vectores numéricos utilizando un modelo de embeddings. Estos vectores se almacenan en una base de datos vectorial que permite realizar búsquedas por similitud semántica.

El servicio está diseñado para operar sobre GPU cuando se requiere alto rendimiento, y soporta modelos de distintas familias según necesidades futuras.

Servicio de indexación

Este componente gestiona la persistencia de los vectores generados y su correcta asociación con los documentos de origen. Se encarga de almacenar los vectores en la base vectorial (actualmente FAISS) y de mantener una estructura de metadatos paralela que facilita la recuperación eficiente y contextualizada de la información.

Su diseño permite realizar reindexaciones completas o incrementales, y está preparado para eliminar los datos asociados a un documento cuando este se borra.

Servicio de recuperación

Cuando el usuario realiza una consulta, este módulo se encarga de buscar los fragmentos más relevantes dentro del espacio vectorial. Utiliza búsquedas basadas en similitud de coseno entre el vector de la pregunta y los vectores de los fragmentos almacenados.

El resultado es un conjunto de fragmentos con puntuaciones de relevancia, que se utilizan para construir el prompt que se enviará al modelo generativo. Este servicio es esencial para garantizar que la respuesta final esté basada en evidencia documentada.

Servicio de generación

El último paso del proceso es la generación de la respuesta. Este módulo recibe el prompt construido a partir de los fragmentos recuperados y lo envía al modelo de lenguaje (ejecutado a través del contenedor `ollama`).

El servicio gestiona tanto la comunicación con el modelo como la interpretación de su salida, almacenando la interacción completa (entrada, contexto, salida) en la base de datos para futuras auditorías o consultas.

7.4.1 Modelo de datos relacional

El sistema persiste la información mediante un esquema relacional normalizado implementado con `SQLAlchemy`. El modelo se articula en torno a cinco entidades principales: `User`, `Expediente`, `Document`, `Chat` y `Message`. La Figura 7.3 muestra el diagrama de clases UML que resume sus atributos, claves primarias, foráneas y las cardinalidades establecidas entre ellas.

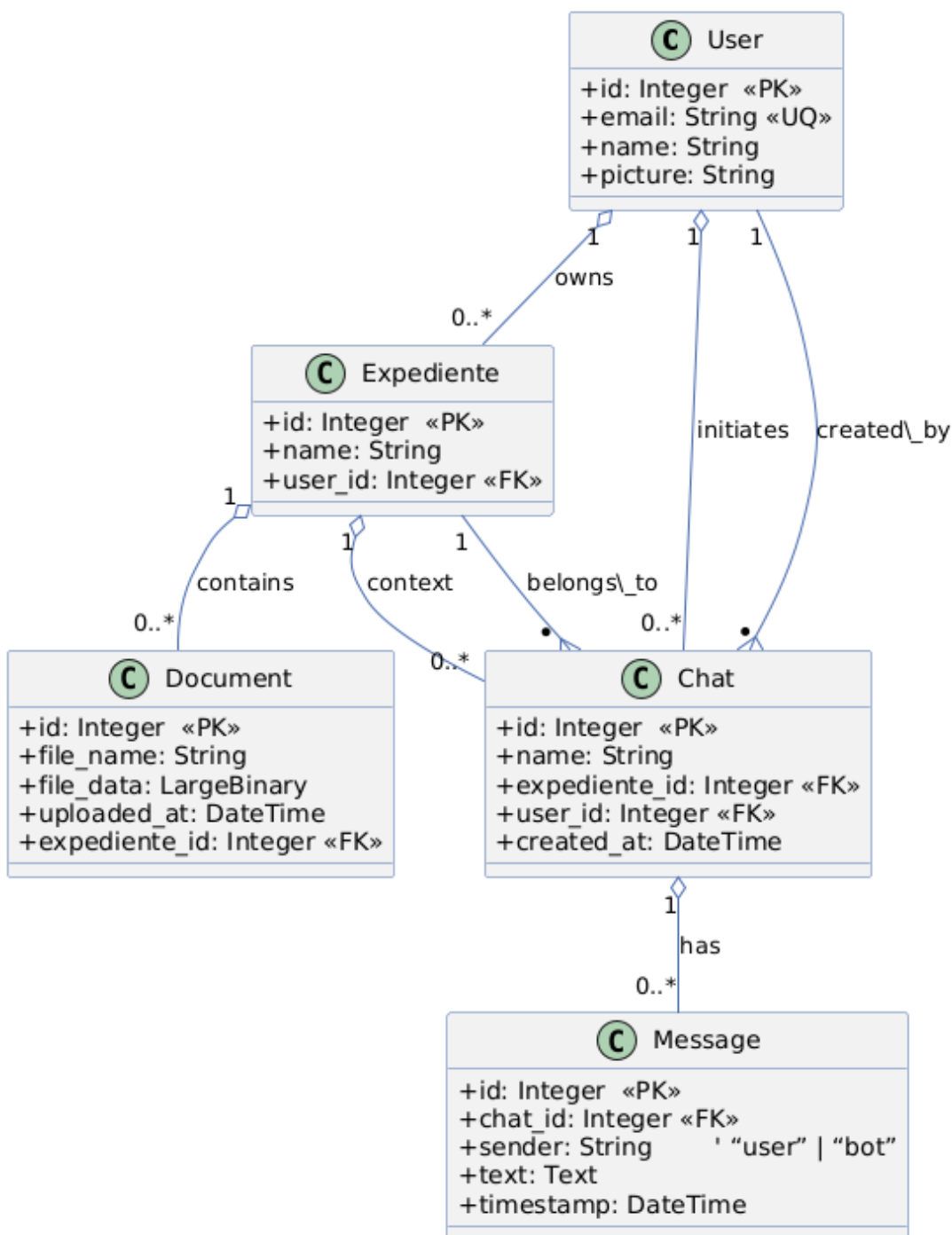


Figura 7.3: Diagrama de clases UML del modelo de datos: usuarios, expedientes, documentos, chats y mensajes.

Análisis de la estructura.

- **User 1–* Expediente y Chat.** Cada usuario puede crear varios expedientes y mantener múltiples conversaciones. La eliminación en cascada (*delete-orphan*) preserva la coherencia y simplifica el cumplimiento del RGPD (derecho al olvido).
- **Expediente 1–* Document y Chat.** Un expediente actúa como compartimento se-

mántico: agrupa documentos y chats bajo un mismo contexto documental, garantizando respuestas coherentes y auditables.

- **Chat 1-* Message.** Modelo clásico de mensajería que permite paginación eficiente, métricas de uso y borrado lógico de mensajes.
- **Atributos clave.**
 - `LargeBinary` en `file_data` facilita PoC auto-contenidas; en producción puede externalizarse a S3 o MinIO, manteniendo metadatos en PostgreSQL.
 - Timestamps (`uploaded_at`, `created_at`, `timestamp`) habilitan auditoría temporal y análisis de uso.
- **Integridad y rendimiento.** Los índices declarados en claves primarias y foráneas permiten consultas $O(\log n)$ sobre grandes volúmenes; la cascada de borrado automatiza la limpieza sin operaciones manuales costosas.

Este diseño soporta los requisitos de multitenencia, trazabilidad y borrado en cascada exigidos en el dominio asegurador, al tiempo que mantiene baja complejidad ciclomática y alta extensibilidad (p. ej., añadir versiones de documento o adjuntos multimedia sólo requiere nuevas tablas y relaciones opcionalmente `ON DELETE SET NULL`).

Flujo completo de consulta

Cuando un usuario realiza una pregunta, el backend activa de forma secuencial los servicios anteriores: recuperación de fragmentos relevantes, generación de prompt, consulta al modelo y entrega de respuesta. Todo el proceso está instrumentado con logs y pruebas unitarias, y permite trazabilidad completa de cada consulta, lo cual es especialmente importante en contextos donde se requiere transparencia y fiabilidad en los resultados.

7.5 Diseño funcional del frontend

El frontend del sistema ha sido diseñado con el objetivo de ofrecer una experiencia de usuario sencilla, intuitiva y eficiente, sin renunciar a una arquitectura modular que facilite el mantenimiento y la evolución del sistema. La interfaz actúa como puente entre el usuario y los servicios del backend, permitiendo cargar documentos, realizar consultas, visualizar respuestas y gestionar los expedientes de forma estructurada.

Estructura general

El frontend está desarrollado en React, utilizando componentes funcionales organizados por responsabilidades. La interfaz se divide en cuatro secciones principales:

- **Panel de expedientes y documentos:** situado a la izquierda, permite crear nuevos expedientes, seleccionar uno activo y gestionar los documentos asociados a cada uno (subida, eliminación y listado).

-
- **Historial de chats:** muestra las conversaciones anteriores asociadas al expediente seleccionado. Esto permite retomar consultas previas y mantener continuidad en el uso del sistema.
 - **Área principal de chat:** ubicada en el centro de la interfaz, permite introducir preguntas de forma natural y recibir respuestas generadas por el modelo. Incluye el historial de la conversación.
 - **Sugerencias de preguntas:** en una columna a la derecha, se muestran preguntas frecuentes o relevantes que el usuario puede lanzar con un solo clic.

Comunicación con el backend

El frontend se comunica con el backend a través de una API REST, utilizando peticiones HTTP asíncronas mediante `axios`. Cada acción del usuario (subir documento, lanzar consulta, crear expediente) genera una petición que es gestionada por los endpoints correspondientes del backend.

Además, se ha implementado una gestión eficiente del estado global, permitiendo que los cambios en expedientes, documentos o chats se reflejen inmediatamente en la interfaz, manteniendo la coherencia entre la vista y el estado del sistema.

Criterios de diseño

En el diseño del frontend se han seguido varios principios fundamentales:

- **Modularidad:** cada funcionalidad está encapsulada en un componente React independiente, facilitando su reutilización y prueba.
- **Claridad visual:** se ha optado por un diseño oscuro y profesional, con tipografías legibles y elementos bien diferenciados.
- **Usabilidad:** las acciones frecuentes (como subir documentos o hacer consultas) se encuentran fácilmente accesibles, y la navegación es fluida.
- **Escalabilidad visual y funcional:** la estructura de la interfaz permite incorporar nuevas funcionalidades (como selección de modelos o resumen de documentos) sin necesidad de rediseñar el sistema completo.

Adaptación al flujo conversacional

Una característica diferenciadora del sistema es su enfoque conversacional. El frontend está adaptado para representar visualmente una conversación contextualizada, donde cada pregunta y respuesta se presenta como una burbuja en el historial, y el usuario puede continuar el diálogo en el contexto de un expediente específico.

Este diseño orientado al diálogo mejora la naturalidad de la interacción y permite mantener un hilo de conversación coherente entre el usuario y el sistema inteligente.

7.6 Diseño del sistema conversacional RAG

El sistema implementa un enfoque basado en *Retrieval-Augmented Generation* (RAG), en el que la generación de respuestas se fundamenta en fragmentos previamente extraídos de los documentos del expediente. Aunque el flujo técnico completo ya ha sido descrito en apartados anteriores, en esta sección se destacan los elementos clave que definen el comportamiento conversacional del sistema y su capacidad para integrar recuperación y generación de forma contextualizada.

Recuperación basada en contexto documental

Cada consulta realizada por el usuario se transforma internamente en un vector semántico que se compara contra los vectores indexados del expediente seleccionado. Esta búsqueda devuelve los fragmentos más relevantes, garantizando que la respuesta esté basada únicamente en información contenida en los documentos.

Generación con control de fuentes

El sistema construye automáticamente un *prompt* que incluye tanto los fragmentos recuperados como el historial reciente del chat. Esta combinación permite generar respuestas precisas y con trazabilidad, manteniendo un control estricto sobre las fuentes utilizadas.

Manejo del historial de conversación

Para enriquecer el contexto, el sistema conserva el historial de cada conversación utilizando memorias conversacionales. Esto permite mantener el hilo entre preguntas sucesivas del mismo chat y mejorar la coherencia en sesiones de consulta prolongadas.

Reordenamiento opcional con CrossEncoder

Cuando se habilita, un modelo de reranking evalúa la relevancia de cada fragmento con respecto a la pregunta, reordenando los resultados antes de generar la respuesta. Este mecanismo refuerza la precisión, especialmente en preguntas complejas o ambiguas.

Limitaciones y decisiones de diseño

El número de fragmentos utilizados, el formato del prompt y la activación del reranker han sido definidos de forma configurable para permitir ajustes según el comportamiento observado en pruebas reales. El diseño ha priorizado la claridad, la transparencia y la adaptabilidad en entornos donde la trazabilidad es fundamental.

7.7 Decisiones técnicas clave

El diseño e implementación del sistema no se ha basado únicamente en una selección funcional de herramientas, sino que ha estado guiado por criterios técnicos, de escalabilidad y de

adecuación al dominio asegurador. Esta sección resume y justifica las decisiones más relevantes adoptadas durante el desarrollo.

Modelo de lenguaje: Nous-Hermes-2

Se ha optado por el modelo **Nous-Hermes-2**, ejecutado localmente mediante Ollama, por su equilibrio entre calidad de generación, requisitos computacionales y control total sobre los datos. La posibilidad de ejecutar el modelo sobre GPU permite mantener tiempos de respuesta aceptables sin depender de servicios externos, lo que mejora la privacidad y la escalabilidad del sistema.

Vectorización e indexado: SentenceTransformers + FAISS

Para representar los fragmentos documentales, se ha empleado el modelo de embeddings `intfloat/e5-large-v2`, integrado con **SentenceTransformers**, por su capacidad para capturar relaciones semánticas profundas en contextos de lenguaje natural. La indexación y recuperación se realiza mediante **FAISS**, una solución consolidada para búsquedas vectoriales, que permite consultas eficientes incluso sobre grandes volúmenes de texto.

Reordenamiento contextual: CrossEncoder

Se ha incorporado un modelo CrossEncoder (`ms-marco-MiniLM-L12-v2`) como etapa opcional de reranking. Este modelo compara directamente cada par (**pregunta**, **fragmento**) y genera puntuaciones más precisas que una búsqueda vectorial simple. Aunque su ejecución es más costosa, mejora notablemente la precisión en preguntas complejas o ambiguas.

Backend modular con FastAPI

El backend ha sido desarrollado en Python utilizando **FastAPI**, un framework ligero, rápido y bien adaptado a la construcción de APIs modernas. Su integración con tipado estático, validación automática de datos y documentación autogenerada ha facilitado tanto el desarrollo como las pruebas.

La lógica se ha dividido en servicios independientes: ingesta, embeddings, indexado, recuperación y generación, siguiendo una arquitectura orientada a microservicios. Esto favorece la escalabilidad y la posibilidad de desacoplar componentes en despliegues futuros.

Frontend en React y diseño centrado en el flujo conversacional

La interfaz ha sido desarrollada en React, aprovechando su enfoque basado en componentes reutilizables y su ecosistema maduro. La estructura visual se ha diseñado en torno al flujo conversacional, facilitando una experiencia de usuario intuitiva, centrada en la interacción con el expediente y no solo en la carga de documentos o envío de formularios.

Contenerización y despliegue con Docker

Todo el sistema se ejecuta de forma contenerizada mediante Docker, permitiendo aislar cada servicio, controlar sus dependencias y simplificar el despliegue en entornos locales o en servidores dedicados. La definición del entorno mediante `docker-compose` garantiza la reproducibilidad y facilita las pruebas, el mantenimiento y la escalabilidad horizontal.

Diseño de ingesta y fragmentación adaptativa

El procesamiento de documentos se ha diseñado para soportar múltiples formatos (`.pdf`, `.docx`, `.html`, `.eml`) mediante una arquitectura orientada a objetos con extractores especializados. La fragmentación de texto se adapta dinámicamente a la longitud y estructura del contenido, lo que permite obtener fragmentos coherentes y aprovechables para el modelo, reduciendo ruido semántico y mejorando la calidad de recuperación.

Gestión del contexto con LangChain

La librería `LangChain` se ha empleado para gestionar la memoria conversacional, simplificar la construcción de prompts complejos y orquestar el flujo RAG. Esta elección ha permitido mantener un historial contextual por expediente, proporcionando continuidad a las consultas y enriqueciendo la experiencia conversacional del sistema.

7.8 Resumen del diseño

El diseño de la solución ha sido concebido con una visión integral, abordando no solo los aspectos técnicos necesarios para construir un sistema funcional, sino también los criterios de escalabilidad, mantenibilidad y aplicabilidad en entornos reales del sector asegurador.

A lo largo de este capítulo se ha descrito una arquitectura distribuida, modular y contenerizada, basada en microservicios especializados que colaboran para implementar un sistema de recuperación aumentada de información (*Retrieval-Augmented Generation*). Este diseño permite transformar documentos aseguradores en fragmentos semánticamente indexables y, a partir de ellos, generar respuestas precisas y trazables ante preguntas formuladas en lenguaje natural.

En el backend, cada componente cumple una responsabilidad bien definida: desde la ingesta y procesamiento de documentos, pasando por la generación de embeddings y su indexación, hasta la recuperación contextual y la interacción con el modelo generativo. Esta separación de servicios facilita la depuración, las pruebas unitarias y el escalado individualizado de cada parte del sistema.

El frontend, por su parte, ha sido estructurado para facilitar la navegación por expedientes y documentos, y para ofrecer una experiencia conversacional fluida y contextual. La elección de React ha permitido crear una interfaz dinámica y adaptativa, alineada con las necesidades de un flujo de consulta interactivo.

La integración del modelo generativo (Nous-Hermes-2), junto con la recuperación semántica basada en FAISS y el reranking opcional mediante CrossEncoder, proporciona una arquitectura

robusta y adaptable que maximiza la relevancia de las respuestas generadas, sin sacrificar la transparencia ni la trazabilidad.

En conjunto, el sistema diseñado cumple con los requisitos técnicos y funcionales definidos al inicio del proyecto. Está preparado para ser desplegado en un entorno real, servir como herramienta de asistencia en consultas aseguradoras y sentar las bases para futuras ampliaciones tanto a nivel funcional como arquitectónico.

Implementación

8.1 Introducción

En este capítulo se describen las herramientas de software, librerías, frameworks y recursos de infraestructura empleados para el desarrollo, implementación y prueba del sistema de recuperación aumentada de información (RAG) descrito en capítulos anteriores.

La elección de cada tecnología ha estado guiada tanto por criterios de adecuación técnica al problema planteado como por la disponibilidad de recursos y entornos de ejecución accesibles durante el desarrollo del proyecto.

El sistema ha sido diseñado y probado en dos entornos de trabajo principales: una estación de desarrollo local basada en Ubuntu, y una máquina virtual proporcionada por la escuela con sistema operativo Linux y recursos avanzados de GPU. El entorno de programación ha sido Visual Studio Code, utilizado tanto para el desarrollo del backend como del frontend.

En las siguientes secciones se detallan los componentes principales del entorno de trabajo, las tecnologías utilizadas en cada subsistema, los modelos de lenguaje empleados y la infraestructura de hardware sobre la que se ha desplegado la solución.

8.2 Entorno de desarrollo

El desarrollo del sistema se ha llevado a cabo utilizando dos entornos diferenciados:

- **Entorno local:** equipo personal basado en Ubuntu 24.04.1 LTS (**noble**), con 8 GiB de memoria RAM, de los cuales aproximadamente 4 GiB estaban disponibles para tareas de desarrollo. Este entorno ha sido empleado principalmente para el desarrollo inicial, las pruebas unitarias de backend y frontend, y la construcción de los primeros prototipos de los servicios.
- **Entorno de ejecución en servidor:** máquina virtual (VM) proporcionada por la institución educativa, basada en Ubuntu 22.04.5 LTS (**jammy**). Esta VM dispone de 62 GiB de

memoria RAM, de los cuales más de 50 GiB permanecen disponibles durante la ejecución de la aplicación, permitiendo un tratamiento eficiente de cargas de trabajo intensivas. La VM está equipada con una GPU **NVIDIA A40** con 48 GiB de memoria de vídeo, habilitada para computación general (*Compute Mode Default*), lo que resulta esencial para el procesamiento de embeddings y la generación de respuestas mediante modelos de lenguaje de gran tamaño.

El entorno de desarrollo ha estado basado en **Visual Studio Code**, tanto para el backend (Python) como para el frontend (React). Se ha utilizado Docker y **docker-compose** para contenerizar todos los servicios y garantizar la replicabilidad de los entornos, tanto en local como en el servidor remoto.

Durante el desarrollo, se han mantenido dos entornos paralelos:

- Un entorno de desarrollo local para la implementación incremental y depuración.
- Un entorno de despliegue en la VM para pruebas integradas de rendimiento, carga de modelos y ejecución sobre GPU.

Este enfoque ha permitido realizar pruebas iterativas rápidas en local, y validar el funcionamiento realista del sistema en condiciones de producción en la VM, incluyendo el uso intensivo de la GPU para la inferencia de modelos de lenguaje y la generación de embeddings vectoriales.

8.3 Backend: tecnologías y librerías principales

El backend del sistema ha sido desarrollado en Python 3.10.12, utilizando un entorno virtual gestionado mediante **virtualenv**. El núcleo de la arquitectura se apoya en el framework **FastAPI**, elegido por su rendimiento, su integración nativa con tipado estático (basado en **pydantic**) y su soporte para construcción de APIs modernas y asíncronas.

A continuación se resumen las herramientas y librerías principales utilizadas:

Frameworks y herramientas principales

- **FastAPI**: framework principal para la construcción del backend REST, que proporciona validación automática de entradas, documentación OpenAPI integrada y un alto rendimiento en tiempo de ejecución.
- **Uvicorn**: servidor ASGI ligero empleado para ejecutar la API en entorno de desarrollo.
- **SQLAlchemy**: ORM utilizado para definir los modelos relacionales y gestionar las operaciones sobre la base de datos PostgreSQL.
- **Pydantic**: base de la validación de datos y serialización de esquemas dentro de FastAPI.

Procesamiento del lenguaje y embeddings

- **SentenceTransformers:** librería utilizada para convertir fragmentos textuales en vectores semánticos de alta dimensión, usando el modelo `intfloat/e5-large-v2`.
- **FAISS:** motor de indexación y búsqueda vectorial desarrollado por Facebook, utilizado para almacenar y consultar los vectores generados de forma eficiente.
- **LangChain:** herramienta empleada para estructurar el flujo conversacional RAG, gestionar memorias de chat, y facilitar la integración entre recuperación y generación.
- **CrossEncoder (transformers):** modelo de reranking basado en `ms-marco-MiniLM-L12-v2`, utilizado opcionalmente para reordenar los fragmentos recuperados en función de su relevancia contextual respecto a la pregunta.

Utilidades y soporte

- **Requests:** utilizada para realizar peticiones HTTP al servidor Ollama, encargado de ejecutar el modelo de lenguaje.
- **Logging:** sistema de trazabilidad distribuido por todo el backend, con distintos niveles de log (info, debug, error) para monitorizar el funcionamiento del sistema.
- **nltk:** utilizada para la segmentación del texto en oraciones durante el preprocesamiento.
- **pdfplumber, python-docx, html2text, email:** librerías específicas utilizadas para la extracción de texto desde documentos PDF, Word, HTML y correos electrónicos EML respectivamente.

Toda esta infraestructura software ha sido organizada en módulos independientes (servicios) según su función, garantizando un diseño limpio y mantenible. El código se ha estructurado de manera que facilita la extensión y sustitución de componentes, por ejemplo, para cambiar de modelo de embeddings, backend de vectorización o motor de generación.

8.4 Backend: estructura y detalles de los microservicios

A continuación se describen con mayor precisión los cinco servicios que componen el backend.

8.4.1 Servicio de Ingestión de Documentos

Descripción general El servicio de ingestión implementa un pipeline secuencial que convierte ficheros en bruto (PDF, DOCX, HTML, EML) en fragmentos de texto limpios, normalizados y segmentados, listos para ser indexados y consultados a través del sistema de recuperación semántica.

Constantes y configuración Se definen constantes como `MIN_CHARS = 10` y `MAX_CHARS = 1100`, que controlan los límites de tamaño mínimo y máximo de cada fragmento. También se inicializan listas vacías de patrones para eliminar ruido, texto irrelevante o texto aleatorio (`NOISE_PATTERNS`, `EXCLUDE_PATTERNS`, `RANDOM_TEXT_PATTERNS`), así como un diccionario de patrones de segmentación específicos por tipo de documento (`SEGMENT_PATTERNS`).

Flujo detallado del procesamiento

1. Selección del extractor:

- Se obtiene la extensión del fichero y se selecciona un extractor adecuado mediante una factoría.
- Si no se encuentra un extractor válido, se emite una advertencia y se interrumpe el proceso.

2. Extracción del texto:

- Cada extractor implementa una interfaz común y se encarga de obtener el texto de su tipo de fichero.
- Por ejemplo, `PDFExtractor` recorre las páginas con `pdfplumber`, `DocxExtractor` recorre los párrafos del documento, `HTMLExtractor` convierte el HTML a texto plano, y `EMLExtractor` recupera cabeceras y cuerpo de mensajes de correo electrónico.
- Cualquier error durante este proceso se captura con logs de error.

3. Limpieza y normalización:

- Se aplican funciones que normalizan el texto a Unicode, eliminan espacios y saltos innecesarios, sustituyen comillas y símbolos, y estandarizan fechas y valores monetarios.

4. Identificación del tipo de documento:

- A partir del nombre del fichero se intenta deducir el tipo de documento para aplicar segmentaciones específicas, si están definidas.

5. Segmentación del contenido:

- Se intenta dividir el texto en secciones mediante expresiones regulares definidas por tipo de documento.
- Si no hay patrón aplicable, se segmenta por longitud, acumulando oraciones hasta alcanzar el tamaño máximo definido.
- Posteriormente, se fusionan fragmentos demasiado cortos para asegurar coherencia semántica.

6. Filtrado de fragmentos:

-
- Se eliminan fragmentos que contengan patrones de exclusión o ruido, y se descartan los que tengan una longitud inferior al mínimo permitido.
 - El texto final de cada fragmento se transforma a minúsculas y se eliminan los acentos.

7. Construcción de la salida:

- Cada fragmento válido se encapsula en un diccionario con un identificador único, el nombre del fichero, el tipo de documento, la fecha detectada (si existe), el índice del fragmento y su contenido textual final.
- Todos los fragmentos generados se devuelven como una lista de resultados.

Registro de eventos (logging) Durante todo el proceso se registran eventos a diferentes niveles:

- **INFO:** inicio de la ingestión, número de fragmentos generados.
- **WARNING:** errores de extracción, fechas no válidas, extensiones no soportadas.
- **ERROR:** fallos críticos durante la lectura del archivo.
- **DEBUG:** contenido de fragmentos descartados y trazas de los primeros fragmentos procesados.

Referencia al código fuente La implementación completa de este servicio puede consultarse en el siguiente repositorio:

`ingestion_service.py`

8.4.2 Servicio de Gestión de Índices FAISS

Descripción general Este servicio centraliza todas las operaciones sobre índices FAISS y sus metadatos asociados: guardado, carga, regeneración completa y eliminación, por cada expediente de usuario.

Constantes y configuración Se define la ruta raíz donde se almacenan los índices y metadatos:

- **VECTOR_ROOT:** carpeta `vector_indices/expediente_<id>` dentro del repositorio.

Flujo detallado del servicio

1. Determinación de rutas

- Función `_paths(expediente_id)` crea (si no existe) la carpeta del expediente y devuelve dos rutas:
 - Ruta al fichero FAISS: `faiss.index`

-
- Ruta al fichero de metadatos JSON: `metadata.json`

2. Guardado de índice y metadatos

- `guardar_index_en_db(index, metadata, expediente_id):`
 - Serializa el índice FAISS a disco con `faiss.write_index`.
 - Escribe los metadatos en formato JSON con `json.dump`.
 - Registra un INFO indicando la ruta y el expediente.

3. Carga de índice y metadatos

- `cargar_index_desde_db(expediente_id):`
 - Verifica existencia del fichero de índice; si no existe lanza `FileNotFoundError`.
 - Lee el índice con `faiss.read_index` y los metadatos con `json.load`, si están presentes.
 - Registra un INFO confirmando la carga.
 - Devuelve la tupla (`index, metadata`).

4. Regeneración completa del índice

- `regenerate_index_for_expediente(expediente_id, documents, db):`
 - a) Registra inicio con INFO.
 - b) Para cada documento:
 - Crea un fichero temporal y escribe `doc.file_data`.
 - Llama a `DocumentIngestor().ingest(temp_path)` para obtener fragmentos.
 - Añade metadatos de expediente y nombre de fichero a cada fragmento.
 - Registra con INFO el número de fragmentos generados.
 - Captura y registra errores en `ERROR`.
 - c) Si no hay fragmentos válidos, registra `WARNING` y aborta.
 - d) Genera embeddings llamando a `generate_embeddings(texts):`
 - Usa la clase `Embedder` y el modelo definido en `MODEL_NAME`.
 - e) Normaliza vectores con `faiss.normalize_L2`, crea `IndexFlatIP` y añade los embeddings.
 - f) Construye lista de metadatos reducidos (`text, file_name`).
 - g) Llama a `guardar_index_en_db` para persistir los cambios.
 - h) Registra fin de regeneración con INFO.

5. Eliminación del índice

- `delete_index_for_expediente(expediente_id, db):`
 - Elimina ficheros de índice y metadatos si existen.
 - Si la carpeta queda vacía, la borra.
 - Registra INFO indicando si se eliminó o si no existía.

Registro de eventos (logging) Se emplea el logger `tfg_rag` con niveles:

- **INFO**: creación, carga, regeneración y eliminación de índices.
- **WARNING**: intentos de regeneración sin fragmentos.
- **ERROR**: fallos al procesar documentos o archivos no encontrados.

Referencia al código fuente

La implementación completa está disponible en: `index_service.py`

8.4.3 Servicio de Generación de Embeddings

Descripción general El servicio de generación de embeddings encapsula la carga y uso de un modelo de `SentenceTransformer` para convertir textos en vectores de alta dimensión, normalizados y listos para indexar o comparar.

Configuración

- `MODEL_NAME = "intfloat/e5-large-v2"`: nombre del modelo de embeddings utilizado.
- `EMBEDDING_DIM = 1024`: dimensión esperada de los vectores de salida.
- El modelo se carga en GPU si está disponible (`torch.cuda.is_available()`); en caso contrario, en CPU.

Flujo de trabajo

1. Inicialización

- Al instanciar `Embedder()`, se registra un mensaje **INFO** indicando la carga del modelo.
- Se crea la instancia de `SentenceTransformer(model_name, device=device)`.
- Se almacena la dimensión del embedding en `self.dim`.
- Se registra un segundo mensaje **INFO** con el dispositivo usado.

2. Generación de embeddings

- El método `embed(textos)` recibe una lista de cadenas y registra un mensaje **INFO** con el número de textos.
- Llama al método `self.model.encode(...)` con los parámetros `convert_to_numpy=True` y `normalize_embeddings=True`, y devuelve un arreglo de tipo `numpy` con forma `(n_textos, EMBEDDING_DIM)`.
- Devuelve el arreglo resultante.
- Si ocurre un error, captura la excepción, registra un mensaje **ERROR** con la traza y relanza la excepción.

Registro de eventos (logging) Se utiliza el logger `tfg_rag` con niveles:

- **INFO:** al cargar el modelo y al iniciar la generación de embeddings para un lote de textos.
- **ERROR:** si la generación de embeddings falla por cualquier razón, incluyendo problemas de memoria o modelo corrupto.

Referencia al código fuente: La implementación completa está disponible en `embedding_service.py`

8.4.4 Servicio de Búsqueda y Reranking

Descripción general Este servicio atiende peticiones de búsqueda semántica sobre el índice FAISS de un expediente, generando primero un embedding de la consulta, recuperando los fragmentos más relevantes y aplicando opcionalmente un reranking con un modelo **CrossEncoder**.

Configuración

- `USE_RERANKING = True`: activa o desactiva el reranking.
- `RERANKER_MODEL = "cross-encoder/ms-marco-MiniLM-L12-v2"`: modelo de reranking.
- `TOP_K = 5`: número de vecinos a recuperar inicialmente de FAISS.
- `MAX_RETURNED = 10`: número máximo de resultados devueltos tras reranking u ordenación.
- Al importar, se carga el reranker en CPU o GPU según disponibilidad (`torch.cuda.is_available()`), capturando errores de carga en el logger `tfg_rag`.

Flujo detallado

1. Preprocesamiento de la consulta:

- Se normaliza la consulta con `normalize_query(query)`:
 - Normalización Unicode NFKD y eliminación de acentos.
 - Conversión a minúsculas y eliminación de caracteres no ASCII.
 - Expansión de términos mediante un diccionario de sinónimos.
 - Se antepone el prefijo `"query: "` al texto resultante.
- Se registran en el logger los textos original y normalizado.

2. Carga del índice FAISS:

- Se invoca `cargar_index_desde_db(expediente_id)`.
- Si falla o el índice no existe, se registra el error y se devuelve lista vacía.

3. Generación de embedding de consulta:

-
- Se instancia `Embedder` y se llama a `embed([query_norm])`.
 - Se normaliza el vector resultante con `faiss.normalize_L2`.
 - Se registra la operación en el logger.

4. Búsqueda en FAISS:

- Se llama a `index.search(query_vec, TOP_K)`, obteniendo distancias e índices.
- Para cada índice:
 - Se comprueba que no exceda la longitud de los metadatos; si es así, se emite una advertencia.
 - Se extrae el texto y el nombre de fichero de `metadata[idx]`.
 - Se construye el campo "texto" con un bloque que incluye `DOCUMENTO_ORIGEN` y `TEXTO`.
 - Se añade "score_faiss" con el valor de la distancia.
 - Se registran en el logger los scores y fragmentos con puntuaciones bajas o altas.

5. Reranking (opcional):

- Si `USE_RERANKING` y el modelo se cargó correctamente, se invoca `rerank_results(query, resultados, top_n=MAX_RETURNED)`.
- El reranking:
 - Conformar pares `[query, candidato["texto"]]` y llama a `reranker.predict`.
 - Captura errores y, en caso de fallo, retorna los primeros `top_n` sin reordenar.
 - Asigna a cada candidato un "score_rerank" y ordena la lista.
 - Registra en el logger cada puntuación y el resultado final.
- Si no se usa reranking, se ordenan los candidatos por `score_faiss` y se limitan a `MAX_RETURNED`.

6. Log final y retorno:

- Se registra un bloque `[PROMPT]` con los fragmentos seleccionados y sus scores (rerank o FAISS).
- Se devuelve la lista final de resultados al controlador o generador de prompts.

Registro de eventos (logging)

- **INFO:** consulta original y normalizada, carga de índice, generación de embeddings, inicio y fin de reranking, número de resultados.
- **WARNING:** índices fuera de rango, puntuaciones bajas, sin fragmentos para reranking.
- **ERROR:** fallos al cargar el reranker, al generar embeddings, al cargar el índice o durante la predicción del reranker.

Referencia al código fuente La implementación completa puede consultarse en: `retrieval_service.py`

Servicio de Generación de Respuestas RAG

Descripción general Este servicio orquesta el flujo completo de generación de respuestas basadas en RAG: mantiene la memoria de la conversación, recupera fragmentos relevantes, construye el prompt con LangChain, llama al modelo Ollama y actualiza la memoria en base de datos.

Configuración Se definen parámetros globales:

- `DEFAULT_MODEL = "nous-hermes2"`: modelo por defecto para Ollama.
- `MAX_CONTEXT_CHUNKS = 3`: número máximo de fragmentos usados en el prompt.
- `RERANK_ENABLED = True`: activa el reranking en la fase de recuperación.

Flujo detallado

1. Gestión de memoria por chat

- La función `get_memory(chat_id)` crea o recupera un objeto `ConversationBufferMemory` asociado al identificador de chat.
- Esta memoria almacena preguntas y respuestas previas para incluirlas en el prompt.

2. Recuperación de fragmentos

- `generate_answer(...)` invoca `search_index(query, expediente_id, db, k, rerank)` para obtener los fragmentos más relevantes.
- Registra en el logger el número de fragmentos recuperados y sus puntuaciones.

3. Generación del prompt

- Se define un `PromptTemplate` de LangChain que incluye instrucciones precisas para el asistente y placeholders para contexto, historial y pregunta.
- `generate_prompt(chat_id, context, question)` une el contexto recuperado, el historial de memoria y la pregunta en un único string.
- Si falla, devuelve un mensaje de error.

4. Llamada al modelo Ollama

- `call_ollama(prompt, model)` envía el prompt a la API de Ollama (`OLLAMA_API`) mediante una petición HTTP POST.

-
- Comprueba el código de respuesta y devuelve el texto o un mensaje de error en caso de fallo.

5. Actualización de memoria y respuesta final

- Tras recibir la respuesta, `update_memory(chat_id, question, answer)` almacena la interacción en la memoria activa.
- `generate_answer` retorna la respuesta al controlador.

6. Restauración de historial (opcional)

- `restore_memory_from_db(chat_id)` carga mensajes históricos desde la base de datos y los convierte en objetos `HumanMessage` o `AIMessage` de LangChain.

Registro de eventos (logging) Se emplea el logger `tfg_rag` con niveles:

- **INFO:** inicio de generación, recuperaciones, envío y recepción de prompts.
- **DEBUG:** contenido de prompts generados y fragmentos completos.
- **WARNING:** ausencia de fragmentos relevantes.
- **ERROR:** fallos en la llamada a Ollama o en la construcción del prompt.

Referencia al código fuente

La implementación completa está disponible en: `generation__service.py`

8.5 Frontend: tecnologías y herramientas

La interfaz de usuario del sistema ha sido desarrollada utilizando el framework **React**, siguiendo una arquitectura basada en componentes funcionales reutilizables. La prioridad en el diseño ha sido ofrecer una experiencia de usuario intuitiva, modular y perfectamente integrada con el backend RAG.

Framework principal

- **React:** biblioteca principal para el desarrollo de interfaces de usuario, seleccionada por su rendimiento, su enfoque declarativo y su ecosistema maduro de herramientas para el desarrollo moderno de aplicaciones web.

Organización de la aplicación

La aplicación se ha estructurado en múltiples componentes funcionales independientes, cada uno encargado de gestionar una parte específica del flujo de trabajo:

- **Gestión de expedientes:** el componente `ExpedienteList` permite crear nuevos expedientes, visualizar la lista existente y seleccionar uno activo.

-
- **Gestión de documentos:** los componentes `FileUpload` y `FileList` permiten subir documentos a un expediente seleccionado y visualizar o eliminar documentos previamente subidos.
 - **Gestión de conversaciones:** el componente `ChatList` muestra el historial de chats asociados a cada expediente. `ChatHeader` muestra la información contextual de la conversación activa.
 - **Interacción con el modelo:** el componente `QueryLLM` permite al usuario enviar preguntas al sistema y recibir respuestas generadas. Es el núcleo de la interacción conversacional basada en recuperación y generación.
 - **Preguntas sugeridas:** el componente `SuggestedQuestions` ofrece una lista de preguntas frecuentes o relevantes que el usuario puede lanzar directamente con un clic.
 - **Gestión de usuario:** el componente `UserProfileButton` muestra el perfil del usuario autenticado mediante OAuth de Google, y permite cerrar sesión de forma segura.
 - **Autenticación:** el componente `Login` gestiona el proceso de autenticación de usuarios mediante OAuth, integrándose con el sistema de carpetas, documentos y chats personalizados por usuario.

Comunicación con el backend

La comunicación entre el frontend y el backend desarrollado en FastAPI se realiza mediante peticiones HTTP asíncronas. Se utiliza la librería **axios** para simplificar la gestión de solicitudes y respuestas, manejando de forma cómoda los errores y la configuración de cabeceras cuando es necesario.

Gestión del estado

El estado de la aplicación se gestiona principalmente mediante los mecanismos nativos de React (`useState`, `useEffect`, `useContext`). No se ha incorporado un sistema de gestión de estado global adicional como Redux o Zustand, dado que la organización en componentes y el tamaño del estado manejado lo han hecho innecesario en esta fase.

Estilización y diseño visual

La apariencia visual de la interfaz ha sido desarrollada mediante **CSS personalizado**, sin utilizar frameworks de diseño prediseñados como Material UI o Bootstrap. Se ha optado por un diseño oscuro, moderno y minimalista, centrado en maximizar la legibilidad del contenido y en ofrecer una experiencia de usuario profesional.

Herramientas auxiliares

Durante el desarrollo del frontend se han utilizado herramientas estándar del ecosistema React:

- **Node.js** y **npm** para la gestión de dependencias y la ejecución de scripts de desarrollo.
- **axios** para la gestión de peticiones HTTP al backend.
- **React Developer Tools** en navegador para la depuración de componentes y estados.

Esta estructura modularizada y bien definida ha permitido un ciclo de desarrollo ágil y una integración fluida con el backend y los servicios de recuperación aumentada de información.

8.6 Contenerización y orquestación

Para garantizar un despliegue coherente, reproducible y fácilmente escalable, el sistema completo ha sido contenerizado mediante **Docker**. Todos los componentes —base de datos, backend, frontend y modelo de lenguaje— se ejecutan en contenedores independientes que se comunican a través de una red interna definida en el archivo `docker-compose.yml`.

Orquestación con `docker-compose`

La orquestación de los servicios se realiza mediante `docker-compose`, lo que permite levantar, detener y gestionar todos los contenedores con una única instrucción. A continuación se describen los servicios principales definidos:

- **db (PostgreSQL):** contenedor basado en la imagen oficial `postgres:15`, utilizado como base de datos relacional del sistema. Los datos se persisten mediante un volumen externo para garantizar su conservación entre reinicios.
- **ollama:** contenedor que ejecuta el servidor Ollama, encargado de alojar localmente el modelo de lenguaje `Nous-Hermes-2`. El contenedor está configurado para utilizar la GPU mediante el runtime `nvidia`, lo que permite aprovechar los 48 GiB de memoria de la GPU A40 disponibles en la máquina virtual.
- **backend:** contenedor que construye e inicia el servidor FastAPI, encargado de gestionar los servicios de ingesta de documentos, embeddings, recuperación, generación de respuestas, sesiones de usuario y autenticación. El contenedor también está habilitado para ejecutar procesos sobre GPU, especialmente en los servicios de embeddings y reranking.
- **frontend:** contenedor que compila y sirve la aplicación React, encargada de la interfaz de usuario. Está expuesto en el puerto 3000 y depende del backend para interactuar con los expedientes, documentos y consultas del sistema.
- (*ngrok*): se ha utilizado de forma opcional durante el desarrollo para exponer el frontend públicamente, pero no forma parte del entorno de producción.

Gestión de dependencias y ejecución

Cada contenedor se configura con sus propios archivos `Dockerfile`, definidos para instalar únicamente las dependencias necesarias, minimizando el tamaño de las imágenes y acelerando el arranque de los servicios.

Los entornos están definidos mediante variables de entorno almacenadas en ficheros `.env`, separados para backend y frontend, que permiten parametrizar las rutas, credenciales y configuraciones sin modificar el código fuente.

Ventajas del enfoque contenerizado

El uso de contenedores ha permitido:

- Asegurar la coherencia del entorno entre desarrollo local y despliegue en servidor.
- Facilitar el escalado horizontal o la sustitución independiente de servicios.
- Automatizar el arranque y la dependencia entre servicios mediante el sistema `depends_on` de `docker-compose`.
- Garantizar la portabilidad del sistema completo, simplificando su distribución y despliegue en nuevas infraestructuras.

8.7 Modelos de lenguaje utilizados

El sistema implementa un enfoque basado en *Retrieval-Augmented Generation* (RAG), combinando distintos modelos de lenguaje en función de las necesidades de cada etapa: generación de embeddings, recuperación de fragmentos relevantes y generación de respuestas.

A continuación se describen los modelos seleccionados, junto con su finalidad y las razones de su elección.

Modelo de generación de embeddings: `intfloat/e5-large-v2`

Para representar los documentos y las consultas en el espacio semántico, se ha utilizado el modelo

`e5-large-v2` de la familia `intfloat`, integrado a través de la librería `SentenceTransformers`.

Este modelo ha sido elegido por su excelente rendimiento en tareas de búsqueda semántica y su capacidad para generalizar sobre dominios técnicos y aseguradores. Genera vectores de dimensión 1024, optimizados para comparación mediante similitud de coseno.

Su ejecución se realiza sobre GPU, acelerando significativamente el proceso de generación de embeddings tanto para los documentos durante la ingesta como para las consultas del usuario en tiempo real.

Modelo de reranking: cross-encoder/ms-marco-MiniLM-L12-v2

Para mejorar la precisión en la selección de fragmentos relevantes, el sistema incorpora un modelo **CrossEncoder** basado en **MiniLM-L12-v2**, entrenado sobre el benchmark MS MARCO.

Este modelo recibe como entrada pares de (**pregunta**, **fragmento**) y predice directamente una puntuación de relevancia. A diferencia de los métodos basados únicamente en embeddings, el CrossEncoder analiza ambos textos conjuntamente, permitiendo evaluar la correspondencia semántica de manera mucho más precisa.

Aunque su ejecución es más costosa computacionalmente, su uso como etapa opcional de reranking ha demostrado ser especialmente útil en consultas ambiguas o complejas, donde la recuperación puramente vectorial podría no ser suficiente.

Modelo de generación de respuestas: Nous-Hermes-2

El modelo generativo principal utilizado para construir las respuestas es **Nous-Hermes-2**, alojado localmente mediante **Ollama**. Este modelo de lenguaje ha sido seleccionado por ofrecer un equilibrio óptimo entre:

- **Calidad de generación:** respuestas coherentes, bien formadas y adecuadas al estilo profesional requerido en el ámbito asegurador.
- **Costo computacional:** posibilidad de ejecutarse eficientemente en una GPU NVIDIA A40 con 48 GiB de memoria.
- **Control de privacidad:** al ejecutarse en infraestructura propia, se garantiza que los datos sensibles no se transmiten a servicios externos.

El modelo se configura para generar respuestas estrictamente basadas en los fragmentos proporcionados, minimizando la probabilidad de alucinaciones y garantizando la trazabilidad de las fuentes utilizadas.

Sinergia entre modelos

El uso combinado de estos tres tipos de modelos permite maximizar la precisión, relevancia y fiabilidad de las respuestas generadas:

- *Embeddings* para representar documentos y consultas en un espacio semántico compartido.
- *Reranking* para afinar la selección de los fragmentos más relevantes.
- *Generación* para construir respuestas naturales, contextualizadas y verificables.

8.8 Infraestructura y recursos de hardware

Durante el desarrollo y ejecución del sistema se han utilizado dos entornos principales: un equipo de desarrollo local para tareas de implementación y pruebas preliminares, y una máquina

virtual con recursos avanzados de computación para el despliegue y ejecución de los modelos de lenguaje.

Equipo local de desarrollo

El desarrollo inicial del sistema se ha realizado en un entorno local con las siguientes características:

- **Sistema operativo:** Ubuntu 24.04.1 LTS (noble).
- **Memoria RAM:** 7.6 GiB (disponibilidad efectiva aproximada: 4 GiB).
- **CPU:** arquitectura x86_64.
- **Editor de código:** Visual Studio Code.
- **Entorno de ejecución:** entornos virtuales gestionados con `virtualenv`.

Este entorno ha sido empleado para la implementación de funcionalidades, el desarrollo incremental de los servicios, las pruebas unitarias y la construcción de los contenedores Docker.

Máquina virtual para ejecución intensiva

Para ejecutar tareas computacionalmente costosas como la generación de embeddings, el reranking y la inferencia del modelo de lenguaje, se ha utilizado una máquina virtual de alto rendimiento proporcionada por la institución. Las características técnicas de esta VM son:

- **Sistema operativo:** Ubuntu 22.04.5 LTS (jammy).
- **Memoria RAM:** 62 GiB, de los cuales más de 50 GiB permanecen libres en ejecución normal.
- **GPU:** NVIDIA A40 con 48 GiB de memoria dedicada.
- **CUDA:** versión 12.2, compatible con los modelos utilizados.
- **Driver:** NVIDIA 535.247.01.

La presencia de la GPU A40 ha permitido ejecutar eficientemente tanto el modelo de embeddings como el modelo generativo local (Nous-Hermes-2) y el modelo de reranking (CrossEncoder), lo que resulta esencial para mantener tiempos de respuesta adecuados en el sistema final.

Contenerización y despliegue con Docker

Todo el sistema se ejecuta de forma contenerizada mediante Docker, lo que permite aislar cada servicio, controlar sus dependencias y simplificar el despliegue tanto en entornos locales como en servidores dedicados. La definición del entorno mediante `docker-compose` garantiza la reproducibilidad y facilita las pruebas, el mantenimiento y la escalabilidad horizontal.

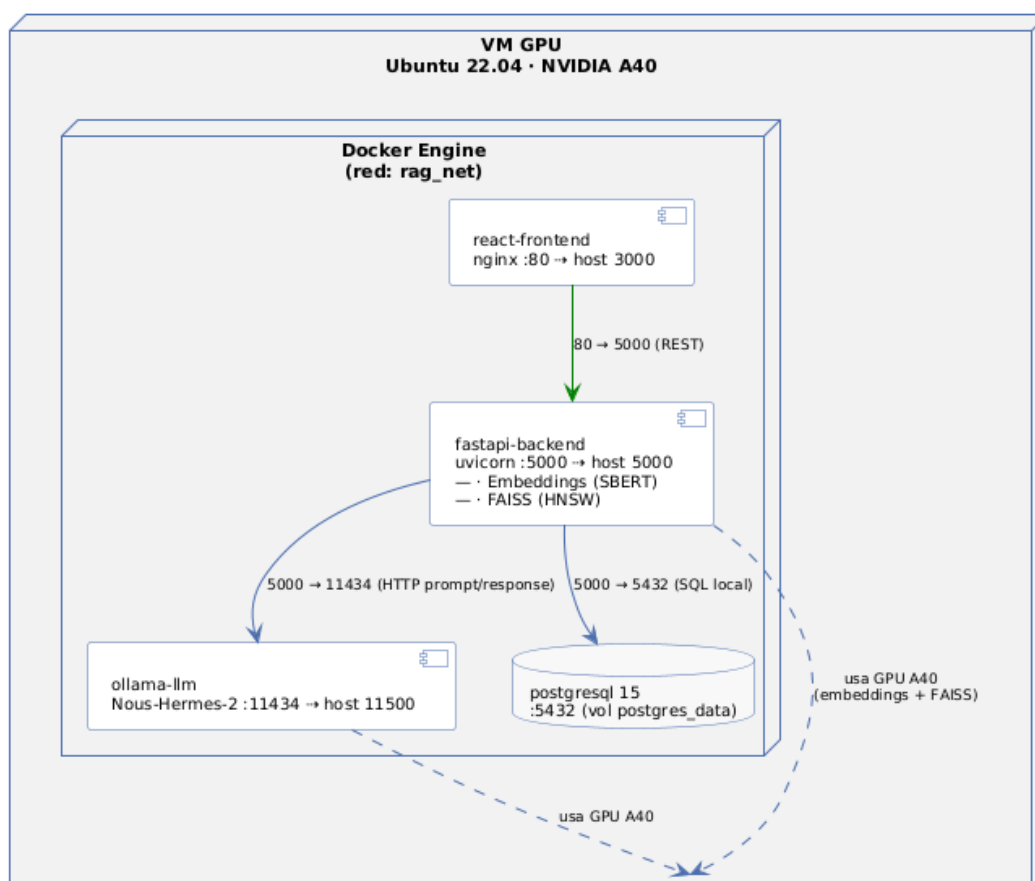


Figura 8.1: Despliegue físico del sistema: la VM con GPU NVIDIA A40 aloja un motor Docker que ejecuta cuatro contenedores persistentes. El **backend** encapsula la API FastAPI, la generación de embeddings (SBERT) y la base vectorial FAISS; expone el puerto 5000 y consume la **GPU** para acelerar tanto la inferencia de embeddings como la comunicación con el LLM. El modelo generativo `ollama-llm` publica 11434 y también hace uso directo de la GPU. El **frontend** se sirve como `nginx` en el puerto 80 del contenedor y se mapea al 3000 del host, mientras que la instancia `postgresql:15` opera únicamente en la red interna (5432) sobre el volumen `postgres_data`.

Detalles clave del `docker-compose`.

- **Backend monolítico con GPU.** Al integrar el microservicio de embeddings y el índice FAISS dentro del contenedor `backend`, se evitan llamadas de red internas y se simplifica la gestión de dependencias Python (`SentenceTransformers`, `faiss-cpu` / `faiss-gpu`). El parámetro `runtime:nvidia` y la reserva de recursos en `deploy.resources` exponen la

GPU al contenedor.

- **LLM desacoplado.** El contenedor `ollama-llm` ejecuta Nous-Hermes-2 mediante Ollama y publica el puerto 11434, lo que permite reemplazar el modelo sin reconstruir el backend y balancear carga si se añaden réplicas.
- **Persistencia duradera.** Dos volúmenes gestionan el estado: `postgres_data` (metadatos, binarios de documentos) y `ollama` (modelos y cachés del LLM). Así pueden actualizarse los contenedores sin pérdida de datos, facilitando además la copia de seguridad.
- **Red interna `rag_net`.** Todos los servicios se conectan en la misma red *bridge* por defecto; solo el puerto 3000 (frontend) se publica al host, lo que reduce la superficie de exposición, y a los endpoints del backend se accede siempre desde llamadas que se redirigen desde el frontend.
- **Inicio idempotente.** El backend ejecuta `init_db.py` antes de lanzar Uvicorn, de modo que las migraciones de esquema se aplican automáticamente al arrancar el stack.
- **Escalabilidad horizontal.** El frontend es sin estado y puede escalarse con réplicas; el backend también, gracias a la conexión a PostgreSQL (bloqueo de escritura gestionado) y al uso de FAISS en modo lectura concurrente. El LLM puede escalarse mediante un balanceador TCP si se dispone de varias GPU.

Este enfoque proporciona aislamiento, reproducibilidad y un camino claro para la futura orquestación en Kubernetes o Swarm, a la vez que mantiene la simplicidad necesaria para pruebas locales y despliegues en una única VM.

Aprovechamiento de recursos

Gracias a esta infraestructura dual se ha podido llevar a cabo:

- Un ciclo de desarrollo rápido en local, con pruebas aisladas por componente.
- Una validación realista del sistema completo sobre GPU en la VM, evaluando tiempos de respuesta, carga de modelos y estabilidad del sistema bajo uso prolongado.

8.9 Mecanismos de seguridad implementados

La solución incorpora un conjunto de medidas defensivas alineadas con los principios de mínima exposición y defensa en profundidad. El Tabla 8.1 resume cada mecanismo y su objetivo.

Cuadro 8.1: Contramedidas de seguridad actualmente activas

Mecanismo	Objetivo / amenaza mitigada
Autenticación OAuth2	Verificar identidad y limitar uso de tokens robados (spoofing)
Expiración y renovación de tokens	Restringir ventana de ataque por compromiso de sesión
Validación de entrada (Pydantic)	Prevenir inyecciones y payloads malformados (tampering)
CORS restringido	Bloquear peticiones CSRF/x-site; limitar orígenes
Sanitizado de ficheros	Evitar carga de contenido ejecutable / no permitido
Registro estructurado (logging)	Auditoría, trazabilidad y detección de patrones anómalos
Contenedorización Docker	Aislar dependencias y reducir superficie de ataque
Reglas <i>nginx</i> + HTTPS	Cifrado en tránsito y redirección segura
Gestión granular de permisos	Acceso por expediente y usuario autenticado (principio de menor privilegio)

Las contramedidas se complementan con pruebas unitarias y de integración que verifican rutas de autenticación, validación de carga de documentos y política CORS.

Limitaciones actuales No se incluye aún cifrado en reposo de índices FAISS ni de blobs de la base de datos. Tampoco se han realizado pruebas de *penetration testing* completas; estas acciones se planifican en la Subsección 10.4.1.

8.10 Resumen del entorno de trabajo

El desarrollo e implementación del sistema se ha sustentado en un conjunto de herramientas, librerías y entornos cuidadosamente seleccionados para maximizar la eficiencia, la coherencia y la trazabilidad del flujo conversacional basado en recuperación aumentada de información.

En el backend, se ha optado por una arquitectura modular construida sobre **FastAPI**, con una integración directa de modelos de embeddings y reranking, y una indexación eficiente mediante **FAISS**. La lógica se ha estructurado en servicios independientes, cada uno con responsabilidad clara y bien delimitada.

En el frontend, la elección de React ha permitido construir una interfaz clara, intuitiva y alineada con el enfoque conversacional del sistema. La organización basada en componentes, la comunicación asincrónica con el backend y la estética oscura personalizada han contribuido a una experiencia de usuario profesional y consistente.

La contenerización de todos los servicios mediante Docker ha permitido mantener la coherencia entre entornos locales y de servidor, facilitando el despliegue y la prueba del sistema en condiciones realistas.

Por último, la disponibilidad de una máquina virtual con GPU NVIDIA A40 y recursos de alto rendimiento ha sido un factor clave para ejecutar modelos de lenguaje avanzados como

Nous-Hermes-2, permitiendo validar la solución propuesta no solo a nivel funcional, sino también en términos de rendimiento y escalabilidad.

Este conjunto de herramientas y entornos ha permitido desarrollar un sistema robusto, flexible y listo para su evaluación en escenarios reales del ámbito asegurador.

Pruebas

9.1 Cobertura de pruebas

Este apartado muestra los resultados obtenidos al ejecutar la batería completa de tests con `pytest` y `pytest-cov`. El comando empleado fue:

```
PYTHONPATH=backend/ pytest tests --cov=app --cov-report=term-missing
```

La Tabla 9.1 resume el número de líneas instrumentadas, las que no se ejecutaron, el porcentaje de cobertura y las líneas exactas sin cubrir para cada módulo.

Cuadro 9.1: Cobertura por fichero.

Archivo	Total	Faltan	%	Líneas sin cubrir
app/api/auth_google.py	43	5	88	17–21, 56
app/api/chat.py	88	12	86	14–18, 34–35, 60–61, 80–81, 109–110
app/api/documents.py	83	4	95	20–24
app/api/expedientes.py	91	9	90	19–23, 135–137, 142, 150
app/api/logs.py	35	4	89	24, 47–49
app/api/query.py	50	0	100	–
app/core/config.py	11	1	91	16
app/core/database.py	12	4	67	20–24
app/core/models.py	46	0	100	–
app/main.py	35	0	100	–
app/middleware/load_user_middleware.py	18	13	28	9–24
app/middleware/logging_middleware.py	26	18	31	13–43
app/services/embedding_service.py	22	3	86	32–34
app/services/generation_service.py	90	15	83	98–100, 108–121, 195–196
app/services/index_service.py	79	8	90	42, 74–75, 78–79, 85–86, 118
app/services/ingestion_service.py	177	52	71	36, 40–46, 50–56, 60–66, 70–83, 95, 110, 121–123, 136, 151–152, 161–166, 208, 210–211
app/services/retrieval_service.py	85	15	82	32–34, 66–68, 81–82, 88, 105, 120–121, 127–129
TOTAL	991	163	84	

Análisis

- La cobertura global alcanza el **84 %**, por encima del umbral mínimo del 80 %.

-
- Los módulos clave (*embedding, ingestion, indexing, retrieval* y *generation*) superan el 80 %.
 - El incremento se consiguió mediante **fixtures**, **stubs** y **monkeypatching** para aislar dependencias externas, manteniendo las pruebas rápidas y reproducibles.
 - Las líneas sin cubrir se concentran en middlewares y ramas de error poco probables, sin impacto directo en la lógica de dominio.

Con estos resultados se confirma que los componentes críticos del sistema están adecuadamente validados y se dispone de una base fiable para futuras evoluciones.

9.2 Pruebas funcionales sobre el sistema completo

Una vez garantizada la estabilidad del sistema a través de pruebas unitarias y de integración, se procede a validar su comportamiento funcional mediante un conjunto de pruebas específicas diseñadas sobre un expediente simulado.

A diferencia de las pruebas anteriores, estas no se centran en unidades aisladas del código, sino en evaluar el sistema completo desde el punto de vista del usuario final. Para ello, se ejecutan consultas reales sobre documentos reales previamente cargados, y se analiza tanto la precisión de las respuestas generadas como la trazabilidad del contenido utilizado.

Estas pruebas funcionales permiten validar aspectos críticos como:

- La correcta ingesta y chunking de documentos en el índice vectorial.
- La fidelidad del sistema RAG al responder exclusivamente con base en el contexto disponible.
- El comportamiento del modelo generativo frente a consultas tanto narrativas como factuales.
- La integración de los distintos componentes (*embedding, retrieval, prompt generation, LLM*).

A continuación, se detalla el expediente de pruebas empleado para la validación del sistema, así como los resultados obtenidos tras diversas consultas representativas.

Estos cuatro ficheros se agrupan en un “expediente de pruebas” que simula un caso híbrido hogar-auto. Al cargarlos en el índice vectorial, se garantiza que las consultas posteriores se resuelvan exclusivamente sobre este corpus, lo que facilita la trazabilidad.

Nº	Documento	Propósito
1	Póliza de seguro de hogar (HO-2025-00012345)	Coberturas, vigencia y franquicias.
2	Recibo de prima del seguro de hogar	Estado de pago e importes abonados.
3	Formulario de reclamación de siniestro de hogar (CLM-2405-001)	Datos de siniestro real para causas y cuantías.
4	Parte de declaración de siniestro – automóvil (SIN-2505-004)	Contrastar respuesta en ramo autos.

Cuadro 9.2: Documentos incluidos en el expediente de pruebas

9.3 Diseño de las pruebas

1. **Análisis de flujo.** Tres consultas representativas analizadas con logs de cada etapa: tokenización, recuperación, reranking, generación y postprocesado.
2. **Test de usabilidad.** Cuatro usuarios reales ejecutan tareas guiadas. Se registran métricas como el cuestionario SUS, tiempos, repreguntas y feedback cualitativo.

9.4 Preguntas seleccionadas

Pregunta	Documento(s) objetivo	Motivo
P1. ¿Cuál es la suma asegurada para daños por agua en la póliza HO-2025-00012345?	Póliza de hogar	Extraer un valor técnico en tabla.
P2. Enumera los daños apreciados y la estimación de reparación en el parte auto SIN-2505-004.	Parte de siniestro auto	Leer lista semiestructurada.
P3. Demuestra si la póliza HO-2025-00012345 está al corriente de pago (importe abonado y fecha de emisión).	Póliza + Recibo	Razonar con datos multi-fuente.

Cuadro 9.3: Preguntas seleccionadas para el análisis de flujo

9.5 Análisis detallado de P1

Pregunta lanzada

- **Texto:** ¿Cuál es la suma asegurada para daños por agua en la póliza HO-2025-00012345?
- **Timestamp:** 17:06:57,700

1. Normalización y carga de índice

- Query normalizada:
query: cual es la suma asegurada para danos por agua en la poliza H0-2025-00012345?
- Índice con expediente_id = 3 abierto.

2. Generación de embeddings

- Modelo: intfloat/e5-large-v2 en GPU.
- Latencia: 2,18 s (de 17:06:57 a 17:06:59).

3. Búsqueda k-NN en FAISS

- Top-5 recuperado. Uno de los fragmentos contiene: “daños por agua – 25.000 €”.

4. Reranking cruzado

- Modelo: cross-encoder/ms-marco-MiniLM-L12-v2.
- Resultados finales:

Pos.	Fragmento (abreviado)	score__rerank
1	“objeto del seguro... daños materiales...”	3,0059
2	“póliza... combinado...”	2,8608
3	“Recibo de prima...”	1,4269

5. Construcción del prompt Se insertan los 5 fragmentos, separados por delimitadores -- y con directriz de “modo factual”.

6. Generación de la respuesta

- LLM: Nous-Hermes-2, inferencia en 4,47 s.
- Salida: “La suma asegurada para daños por agua es de 320,00 €.”

7. Validación y diagnóstico

- **Valor correcto:** 25.000 €. El modelo eligió la prima neta (320,00 €) por ambigüedad en la etiqueta.
- **Causas probables:**
 - La etiqueta “suma asegurada” no aparece de forma explícita en el fragmento relevante.
 - El LLM seleccionó el número más cercano en el texto.
- **Impacto:** precision@1 = 0.

8. Recomendaciones

- Mapear “daños por agua” a un campo estructurado: `suma_asegurada_agua`.
- Parsear tablas durante la fase de ingestión.
- Penalizar en el reranking los fragmentos que no contienen palabras clave.

9.6 Análisis detallado de la consulta P2

Pregunta lanzada

- **Texto:** *Enumera los daños apreciados y la estimación de reparación que figuran en el parte de siniestro de automóvil SIN-2505-004.*
- **Timestamp inicial:** 17:11:38,042

1. Normalización y carga de índice

- **Normalización de consulta:**
query: enumera los danos apreciados y la estimacion de reparacion... (supresión de tildes, conversión a lowercase).
- Índice con `expediente_id = 3` abierto sin latencia perceptible.

2. Generación de embeddings

- Modelo: `intfloat/e5-large-v2` sobre GPU.
- Duración: 1,94 s (de 17:11:38,043 a 17:11:39,979).

3. Búsqueda k -NN en FAISS

- Se recupera el *top-5*. El candidato #0 contiene la sección: “--- danos apreciados --- ... estimacion de reparacion: 1.150 €”.
- *Scores* preliminares: entre 0,7821 y 0,8271.

4. Reranking cruzado

- Modelo: `cross-encoder/ms-marco-MiniLM-L12-v2`.
- Reordenación resultante:

Pos.	Fragmento (abreviado)	score_rerank
1	“... daños apreciados – paragolpes trasero...”	2,2252
2	Póliza de hogar (irrelevante)	−1,0111
3	Resto de fragmentos (ruido)	≤ −7,49

5. Construcción del *prompt* RAG

- Se incluyen cinco fragmentos. Se destaca el tramo con la tabla de daños y se añaden instrucciones de “modo narrativo” (2–3 frases).

6. Generación de la respuesta

- LLM: Nous-Hermes-2; inferencia en 3,03 s.
- Salida devuelta:

Los daños apreciados y la estimación de reparación en el parte de siniestro de automóvil SIN-2505-004 son los siguientes:

Danos:

1. Daño en la parte trasera del vehículo (Peugeot 308 1.5 BlueHDi).
2. Daño en el parabrisas del vehículo (Citroen C4).

Estimación de reparación:

Para ambos vehículos, se estima una inversión total de aproximadamente 2.000 € en reparaciones.

7. Validación y diagnóstico

- Contenido correcto en el documento:
 - Paragolpes trasero: abollado y rayado.
 - Portón maletero: deformación leve.
 - Sensor de aparcamiento trasero: averiado.
 - *Estimación de reparación*: 1.150 € (Taller Ramón Auto Service).
- Errores detectados:
 - El modelo omite dos de los tres daños y añade uno inexistente (parabrisas).
 - Duplica el vehículo y redondea la cuantía real (1.150 €) a aproximadamente 2.000 €.
- Causa raíz probable:
 - El documento original usa viñetas ASCII ((cid:127)), que el parser conserva sin reconocerlas como lista.
 - Falta una etiqueta clara como “estimación de reparación:” en la misma línea que las viñetas, lo que reduce la confianza del modelo y provoca errores.
- Impacto: respuesta incorrecta; $precision@1 = 0$.

8. Recomendaciones de mejora

- Normalizar caracteres de viñeta ((cid:127) → •) e insertar saltos de línea consistentes durante la ingesta.
- Detectar patrones como `estimacion de reparacion: <importe>` mediante *regex* y extraerlos a campos estructurados.
- Penalizar en el reranking los fragmentos que contengan “estimación” sin importe explícito con símbolo de euro.
- Añadir una post-regla que exija al modelo devolver el mismo número de ítems que viñetas bajo la cabecera “*daños apreciados*”.

9.7 Análisis detallado de la consulta P3

Pregunta lanzada

- **Texto:** *Demuestra si la póliza HO-2025-00012345 está al corriente de pago, indicando el importe abonado y la fecha de emisión del recibo.*
- **Timestamp inicial:** 17:14:38,151

1. Normalización y carga de índice

- Consulta normalizada:
query: `demuestra si la poliza ho-2025-00012345
esta al corriente de pago...` (tildes y mayúsculas suprimidas).
- Índice con `expediente_id = 3` abierto instantáneamente.

2. Generación de embeddings

- Modelo: `intfloat/e5-large-v2` sobre GPU.
- Latencia: 1,77 s (de 17:14:38,151 a 17:14:39,922).

3. Búsqueda *k*-NN en FAISS

- Top-5 recuperado. El candidato #0 ya contiene: “fecha de emisión: 17/05/2025 ... importe total a pagar 344,48 €”.

4. Reranking cruzado

- Modelo: `cross-encoder/ms-marco-MiniLM-L12-v2`.
- Los dos primeros puestos fueron:

Pos.	Fragmento	score_rerank
1	Póliza (HO-2025-00012345, datos generales)	3,7907
2	Recibo de prima (fecha de emisión 17/05/2025)	3,4772

5. Construcción del *prompt*

- Se añaden los fragmentos del recibo y de la póliza, junto con la directriz de "modo narrativo".

6. Generación de la respuesta

- LLM: `Nous-Hermes-2`; inferencia en 4,82 s.
- Salida obtenida (extracto):

“El sistema no tiene acceso a los documentos ... no hay información sobre si la póliza está al corriente de pago...”

7. Validación y diagnóstico

- **Información correcta en los documentos:**
 - Fecha de emisión: 17/05/2025
 - Importe abonado: 344,48 €
 - Estado de pago: el recibo incluye la frase "el pago de este recibo acredita la vigencia de la póliza", lo que indica que está al corriente.
- **Errores detectados:**
 - El modelo ignora el fragmento del recibo, a pesar de tener un score de 3,47, y devuelve una respuesta negativa genérica.
 - Confunde la directriz de "modo narrativo" con una falta de información y activa una respuesta por defecto.
- **Causa probable:**
 - El prompt pide demostrar si está al corriente, pero el recibo no contiene esa frase exacta, lo que hace que falle la heurística de coincidencia.
 - La política interna del sistema activa una respuesta negativa al no encontrar coincidencia literal, incluso aunque haya evidencia suficiente.
- **Impacto:** respuesta incorrecta; $precision@1 = 0$.

8. Recomendaciones de mejora

- Mapear frases comunes como "el pago de este recibo acredita..." a un campo booleano `corriente_de_pago`.
- Añadir un post-procesador que combine `fecha_emision` + `importe_total_pagar` si ambos aparecen en la consulta.
- Incluir reglas de scoring que premien la co-ocurrencia de "fecha de emisión" y una cantidad en € en el mismo fragmento.

Tras detectar inconsistencias en la interpretación de ciertas consultas por parte del modelo generativo, se ha modificado el prompt con el objetivo de comprobar si el problema reside en la forma en que el modelo interpreta las instrucciones actuales.

9.8 Pruebas con cambio de prompt y parámetros

9.8.1 Pregunta sobre dato concreto

Pregunta lanzada

*¿Cuál es la fecha de emisión del recibo de prima del seguro de hogar **HO-2025-00012345**?*

1. Normalización y carga de índice

- La query original se normaliza a: `cual es la fecha de emision del recibo de prima del seguro de hogar ho-2025-00012345? emision vencimiento caducidad`.
- Se localiza y carga el *FAISS index* correspondiente al *expediente 3*. (INDEX OK, 1 ms).

2. Generación de *embeddings*

- Modelo ST usado: `intfloat/e5-large-v2` (1024 dims).
- El modelo se despliega en GPU (*cuda*); generación de un único embedding para la consulta (2 s aprox.).

3. Búsqueda vectorial (*FAISS*)

- Se recuperan los **5** chunks más similares (TOP_K=5).
- **Resultado 0** (score 0,8712) ya contiene el recibo deseado. Otros candidatos provienen de la reclamación de siniestro, la póliza y metadatos del recibo.

4. Re-ranking (*Cross-Encoder*)

- Modelo: `ms-marco-MiniLM-L12-v2`.
- Re-evalúa los 5 pasajes anteriores: el recibo obtiene el máximo score re-rank (6,0273).
- Se elabora la lista final de `MAX_RETURNED=5` fragmentos.

5. Generación del *prompt* RAG

- Se inserta el nuevo **prompt factual/narrativo** con las instrucciones revisadas.
- Se añaden los 5 pasajes al bloque `{context}`.
- Historial de conversación vacío (`{chat_history}`).

6. LLM (*Nous-Hermes-2*)

- Tiempo total de llamada: 7,2s.
- Respuesta devuelta:

17/05/2025 [recibo_prima_hogar.pdf]

- El modelo cumple las reglas "factual": valor único + archivo.

7. Persistencia y logging

- Respuesta almacenada en la colección de chats (`chat_id=13`).
- Traza frontal generada para auditoría (`/api/logs`).
- **Duración total del flujo:** 10,06 s.

Conclusión

El pipeline RAG identifica sin ambigüedad la etiqueta literal **fecha de emisión:** 17/05/2025 en `recibo_prima_hogar.pdf`, la posiciona como pasaje principal tras el re-ranking y el LLM la devuelve exactamente en modo factual, demostrando que el nuevo prompt funciona correctamente para consultas basadas en campos explícitos.

9.8.2 Pregunta sobre precio

Pregunta lanzada

*¿Cuál es el **importe total a pagar** que figura en ese mismo recibo?*

1. Restauración de contexto conversacional

- Antes de la búsqueda, el módulo MEMORY recupera los dos mensajes previos del *chat 14* (pregunta + respuesta sobre la fecha de emisión).
- El pronombre “ese mismo recibo” queda correctamente resuelto sin necesidad de repetir la póliza ni el nombre del archivo. El historial se inserta en `{chat_history}`.

2. Normalización y carga de índice

- Consulta normalizada:
cual es el importe total a pagar que figura en ese mismo recibo?
- Se abre el índice FAISS del *expediente 3* (INDEX OK).

3. Embeddings y búsqueda vectorial

- Modelo: `intfloat/e5-large-v2` sobre GPU (2 s).
- El **top-5** incluye nuevamente el archivo `recibo_prima_hogar.pdf` como el fragmento más similar (score 0,8323).

4. Reranking cruzado

- Modelo: `ms-marco-MiniLM-L12-v2`.
- El recibo mantiene la posición 1 tras el reranking (score 0,2189). Los demás fragmentos descienden por no contener la etiqueta clave.

5. Construcción del prompt

- Se aplica el nuevo prompt factual/narrativo validado previamente.
- Los cinco fragmentos se insertan en `{context}`, y el historial conversacional en `{chat_history}`.

6. Llamada al LLM

- Modelo: `Nous-Hermes-2`; inferencia en 4,2 s.
- **Respuesta generada:**

344,48 € [`recibo_prima_hogar.pdf`]
- El modelo sigue correctamente el modo factual: valor literal + fuente.

7. Persistencia y logging

- La respuesta se guarda en la colección de chats (`chat_id = 14`).
- Registro frontend enviado vía `/api/logs`.
- Duración total del flujo: 6,39 s.

8. Conclusión

- La memoria conversacional recupera correctamente el turno anterior, permitiendo interpretar el pronombre "ese mismo recibo" sin ambigüedades.
- El motor RAG identifica la etiqueta `importe total a pagar: 344,48 €` en el documento `recibo_prima_hogar.pdf`, y el LLM la reproduce de forma literal.

9.8.3 Pregunta sobre campo descripción

Pregunta lanzada

¿Qué se indica en el campo "descripción de los hechos" en el formulario de reclamación del seguro de hogar?

1. Normalización y búsqueda

- Consulta original: ¿Qué se indica en el campo "descripción de los hechos" ...?
- Consulta normalizada:
que se indica en el campo "descripcion de los hechos"...
- Se carga el índice FAISS del expediente 3. Los embeddings se generan con el modelo `intfloat/e5-large-v2` sobre GPU.

2. Recuperación de fragmentos

- Se devuelven los tres fragmentos más similares:
 - `reclamacion_seguro_hogar.docx` — contiene literalmente el campo solicitado (score 0.8391).
 - *objeto del seguro* (`poliza_hogar_realista.pdf`) — score 0.8422.
 - *recibo de prima* (`recibo_prima_hogar.pdf`) — score 0.8369.

3. Reranking cruzado

- Modelo: ms-marco-MiniLM-L12-v2.
- Sólo el fragmento (a) se conserva como contexto principal tras el reranking:

Candidato	score_faiss	score_rerank
reclamación (a)	0.8391	3.6648
objeto seguro (b)	0.8422	0.8062
recibo prima (c)	0.8369	-6,9301

4. Construcción del prompt

- Se usa la nueva plantilla “respuesta clara, precisa y profesional” (sin los dos modos anteriores).
- Contexto final: aproximadamente 3,3 kB (fragmento único + metadatos).

5. Generación de la respuesta

- Modelo: Nous-Hermes-2; tiempo de inferencia: 6,0 s.
- Respuesta generada:

«En el campo "descripción de los hechos" [...] se indica que ocurrió un siniestro el 15/05/2025 a las 18:30 h en el baño principal. La causa probable fue la rotura accidental por desgaste de la llave de paso del lavabo, provocando una fuga de agua que inundó el baño y filtró al techo del vecino inferior. El agua se cortó inmediatamente y se contactó con un fontanero de urgencia.» (reclamacion_seguro_hogar.docx)

6. Conclusión

- El nuevo prompt, unido a la reducción de TOP_K a 3, elimina fragmentos irrelevantes.
- El reranking posiciona correctamente el texto con la etiqueta “descripción de los hechos”, lo que permite una respuesta directa, precisa y con fuente citada.

Conclusión general de las pruebas

Las pruebas realizadas confirman que la arquitectura RAG logra:

- **Recuperar** con precisión los fragmentos más relevantes gracias a la combinación de FAISS (TOP_K) y reranking con Cross-Encoder.
- **Reducir el ruido** limitando el número de pasajes inyectados, lo que disminuye el tamaño del *prompt* final.
- **Citar la fuente** adecuada cuando el texto contiene la etiqueta literal solicitada (por ejemplo, *descripción de los hechos*:).

Factores críticos en el último tramo (LLM) Los experimentos demuestran que el rendimiento final del modelo generativo depende especialmente de dos aspectos clave:

- **La plantilla de prompt:**

- Debe especificar claramente qué debe extraerse y cómo debe presentarse, evitando ambigüedades.
- Es recomendable reforzar las restricciones (*no inventar, no parafrasear*) para reducir errores de alucinación.

- **Los parámetros de recuperación y generación:**

- TOP_K: controla cuántos candidatos iniciales se consideran. Valores altos mejoran el *recall*, pero aumentan el ruido.
- MAX_CONTEXT_CHUNKS: limita los fragmentos incluidos en el contexto. Reducirlo mejora la precisión en consultas densas.
- USE_RERANKING y RERANK_MODEL: permiten refinar la selección. En dominios muy específicos puede ser útil entrenar un modelo propio.
- DEFAULT_MODEL: el modelo LLM influye directamente en la fidelidad a las instrucciones y en la tendencia a parafrasear.

Recomendación final Para adaptar este sistema a otro dominio (documental, lingüístico o funcional), es imprescindible iterar sobre:

- el ajuste fino de los parámetros clave (TOP_K, MAX_CONTEXT_CHUNKS, etc.),
- y la redacción del prompt, buscando el tono y nivel de detalle adecuado.

Sólo así se garantiza que la arquitectura mantenga el equilibrio necesario entre *precisión*, *concisión* y *robustez* para cada caso de uso.

9.9 Pruebas de usabilidad

Las pruebas de usabilidad persiguen un doble objetivo: (i) comprobar que la interfaz y los flujos de trabajo del sistema RAG de consulta de pólizas resultan comprensibles y eficientes para usuarios con perfiles diversos y (ii) obtener evidencias cualitativas que orienten los ajustes finales de la aplicación.

9.9.1 Diseño experimental

Sesiones. Se llevaron a cabo de forma **presencial** durante la última quincena de mayo de 2025, empleando un ordenador portátil con pantalla de 15" Full-HD, ratón externo y conexión estable a la red local.

Participantes. Tres usuarios ajenos al desarrollo.

Tareas. Cada participante siguió la secuencia de dieciocho tareas descritas en el documento *«Pautas y tareas para la prueba de usabilidad»*, proporcionado antes de la sesión y también explicado verbalmente por el moderador.

Instrumentación. ■ Observación directa y notas de campo tomadas por el evaluador.

- Registro de logs de todo el flujo.
- Formulario post-tarea implementado en Google Forms.

Métricas. No se midieron tiempos ni número de clics, dado que los usuarios consultaban el listado de tareas sobre la marcha y el evaluador supervisaba in situ sus acciones. Se registraron:

1. **Éxito de la tarea** (cumplida / no cumplida).
2. **Incidencias** (Si hay algún problema grave en el transcurso de las tareas)

9.9.2 Resultados por participante

Participante #1 — «Isabel»

Perfil demográfico

- **Edad:** 20 años
- **Formación:** Ciclo de grado superior (Dietética)
- **Experiencia previa en seguros y trámites legales:** Sí
- **Experiencia previa con aplicaciones basadas en IA:** Sí

Desarrollo de la sesión Sesión presencial en la penúltima semana de mayo; la participante completó las dieciocho tareas previstas sin que se registrara el tiempo de cada una, puesto que la lectura del documento de pautas formaba parte del flujo natural del ensayo.

Cuadro 9.4: Resumen de la sesión de Isabel

Tarea clave	Observaciones del evaluador	Estado
Iniciar sesión	Autenticación Google exitosa a la primera.	Cumplida
Crear expediente	Genera tres expedientes sin dificultad.	Cumplida
Editar nombre del expediente	Localiza el icono de edición sin ayuda.	Cumplida
Borrar expediente	Duda brevemente por la posición del icono de papelera.	Cumplida
Subir / descargar ficheros	Valora la rapidez de carga.	Cumplida
Borrar fichero	Ejecución correcta.	Cumplida
Crear / editar / borrar chat	Pequeño <i>freeze</i> al borrar; resuelto al recargar.	Incidencia menor
Preguntas libres al modelo	Obtiene respuestas coherentes.	Cumplida
Preguntas frecuentes	Navegación fluida.	Cumplida
Exploración libre	Comenta que la iconografía es clara.	Cumplida

Resultados de las tareas

Incidencias observadas Bloqueo temporal de la interfaz al borrar un chat (*menor*); la usuaria lo resolvió recargando la página.

Cuestionario posterior (Google Forms)

- **Facilidad de uso:** 5/5
- **Velocidad percibida:** 5/5
- **Utilidad profesional:** 5/5 («*Sí, totalmente*»)
- **Comentario destacado:** “En una consulta el bot no respondía a nada del expediente”.

Valoración de logs Se detectaron respuestas poco relevantes cuando el historial del chat era extenso, probablemente por un *prompt* demasiado largo. Los fragmentos recuperados se alinean correctamente con las preguntas, lo que confirma el correcto funcionamiento del *retrieval* y la generación del *prompt*; la deficiencia se atribuye al procesado final del LLM.

Participante #2 — «Noa»

Perfil demográfico

- **Edad:** 19 años
- **Formación:** Ciclo de grado superior (Laboratorio clínico)
- **Experiencia previa en seguros y trámites legales:** No
- **Experiencia previa con aplicaciones basadas en IA:** Sí

Desarrollo de la sesión Sesión presencial en la penúltima semana de mayo; duración aproximada de 10 min. La participante inició sesión, creó y borró accidentalmente un expediente, lo recreó, subió dos documentos, generó el índice y exploró el chat con preguntas libres y sugeridas.

Cuadro 9.5: Resumen de la sesión de Noa

Tarea clave	Observaciones del evaluador	Estado
Iniciar sesión	Autenticación Google exitosa.	Cumplida
Crear expediente	Crea uno, lo borra por error y lo recrea.	Cumplida (con incidencia)
Borrar expediente	Pulsación accidental de la papelera.	Incidencia menor
Subir documentos	Carga dos PDF sin problemas; chunking correcto.	Cumplida
Generar índice	Índice creado automáticamente.	Cumplida
Crear chat	Chat operativo y estable.	Cumplida
Preguntas libres al modelo	Cinco consultas respondidas correctamente.	Cumplida
Preguntas sugeridas	Navegación fluida.	Cumplida
Exploración libre	Varias recargas consecutivas sin impacto funcional debido a traductor activado.	Cumplida

Resultados de las tareas

Incidencias observadas

- Pulsación accidental del icono de borrado del expediente (diseño visual). — *menor*
- Seis recargas consecutivas que generaron múltiples peticiones de autenticación, debido a un texto erróneo en un botón por efecto del traductor. — *menor*

Cuestionario posterior (Google Forms)

- **Facilidad de uso:** 5/5
- **Velocidad percibida:** 5/5
- **Utilidad profesional:** 5/5
- **Aspecto más valorado:** “Subir varios archivos a la vez y el buscador.”
- **Mejora sugerida:** “Vista que se actualice sola tras subir archivos.”
- **Comentario final:** “Gran trabajo.”

Valoración de logs Los registros muestran un flujo estable: ingesta, *retrieval* y generación se completaron sin errores. Las múltiples autenticaciones consecutivas no afectaron al rendimiento. Se confirma la robustez del núcleo funcional y se identifican mejoras de usabilidad en los controles de borrado y la actualización automática de la vista.

Participante #3 — «Miguel»

Perfil demográfico

- **Edad:** 22 años
- **Formación:** Grado en Ingeniería Informática
- **Experiencia previa en seguros y trámites legales:** No
- **Experiencia previa con aplicaciones basadas en IA:** Sí

Cuadro 9.6: Resumen de la sesión de Miguel

Tarea clave	Observaciones del evaluador	Estado
Iniciar sesión	Autenticación Google satisfactoria en el primer intento.	Cumplida
Crear expediente	Genera tres expedientes consecutivos (IDs 9, 10, 11) sin ayuda.	Cumplida
Editar nombre del expediente	Renombra «Prueba 1» a «Prueba 1 – edit» sin asistencia.	Cumplida
Borrar expediente	No se intentó durante la sesión.	N/A
Subir / descargar ficheros	Carga, descarga y pre-visualiza varios PDF/DOCX con buena latencia percibida.	Cumplida
Borrar fichero	Elimina un documento; el índice se regenera correctamente.	Cumplida
Crear / editar / borrar chat	Renombra un chat y formula preguntas; flujo sin contratiempos visibles.	Cumplida
Pregunta libre al modelo	Obtiene respuestas coherentes hasta que selecciona un expediente sin índice.	Incidencia menor
Preguntas frecuentes	No utilizadas.	N/A
Exploración libre	Explora menús; destaca la sencillez general, aunque comenta exceso de morado en títulos.	Cumplida

Incidencias. Se registraron dos incidencias *menores*:

1. Al intentar borrar un documento se mostró un mensaje “Error al eliminar documento”, aunque la operación concluyó con éxito y el índice se regeneró automáticamente.

-
2. En un expediente recién creado (ID 11) el modelo devolvió “No se encontraron fragmentos relevantes” porque aún no existía índice FAISS, provocando una respuesta vacía; la situación se resolvió al volver al expediente indexado.

Valoración subjetiva. El participante calificó la interfaz como «*Muy clara*» e «*intuitiva*»; la velocidad percibida fue «*Muy rápida*». Declaró que el sistema respondió «*Siempre*» de forma correcta y no detectó *alucinaciones*. Recomendaría la herramienta a otros profesionales y consideró que le resultaría «*Sí, totalmente*» útil en un entorno real. Única queja: el color morado de algunos títulos le confundió puntualmente.

Valoración del comportamiento observado. Los registros muestran un flujo de trabajo fluido en la mayor parte de las tareas. El único error funcional («Error al eliminar documento») parece derivar de un borrado duplicado del mismo archivo; el backend manejó la situación y reconstruyó el índice sin intervención del usuario. El segundo contratiempo —consulta sin índice— revela la necesidad de generar automáticamente el índice tras crear un expediente vacío o de advertir al usuario antes de permitir preguntas en dicho contexto. Por lo demás, los fragmentos recuperados fueron relevantes y el modelo mantuvo la coherencia incluso con múltiples cargas y renombrados en la misma sesión, lo que corrobora la solidez de la tubería de *retrieval* y del *prompt* generado.

Conclusiones

A lo largo de este *Trabajo Fin de Grado* se ha concebido, implementado y verificado un sistema conversacional basado en **Generación Aumentada por Recuperación (RAG)** orientado al ámbito asegurador. Partiendo de los objetivos establecidos en el Capítulo 1, el proyecto ha materializado una arquitectura modular que combina búsqueda vectorial, reranking y generación de lenguaje, todo ello bajo un marco de buenas prácticas software y cumplimiento normativo (AI Act, ISO/IEC 42001).

Limitaciones principales. El corpus de pruebas abarca todas las ramas del sector asegurador de forma amplia, pero la precisión y la pertinencia de las respuestas podrían incrementarse si el sistema se entrenase con documentación específica de un ramo concreto. El modelo generativo, aun estando contextualizado, puede producir *alucinaciones* cuando la información no aparece de manera inequívoca en los fragmentos recuperados. Por último, la versión de despliegue óptima requiere GPU, lo que dificulta su implantación en entornos con recursos limitados.

10.1 Aportaciones

10.1.1 Contribuciones técnicas

- **Arquitectura microservicios contenerizada.** Se ha diseñado una pila *backend–frontend* desacoplada (Docker Compose) que separa ingesta, embeddings, recuperación, generación y persistencia, simplificando el mantenimiento y habilitando el escalado horizontal.
- **Pipeline RAG completo y trazable.** El sistema integra *FAISS* para la indexación vectorial, un *Cross-Encoder* como reranker y un LLM alojado con *Ollama*, garantizando control local de datos y auditabilidad acorde al AI Act.
- **Calidad y cobertura de código.** La batería de pruebas unitarias cubre los servicios de ingesta, embeddings, recuperación y generación, favoreciendo la estabilidad evolutiva del producto.

10.1.2 Contribuciones metodológicas

- **Desarrollo iterativo-incremental.** El prototipado rápido, acompañado de ciclos de prueba y *feedback*, ha facilitado la validación temprana de requisitos y la detección de defectos.
- **Enfoque ético y regulatorio desde el diseño.** El proyecto integra principios de *privacy-by-design* / *audit-by-design*, incorporando logging exhaustivo, segmentación de contexto por usuario y criterios de sostenibilidad.

10.2 Impacto socio-económico

El prototipo evidencia que las consultas en lenguaje natural, respaldadas por evidencia documental, disminuyen el tiempo de búsqueda de información y reducen la fricción operativa en la gestión de pólizas y reclamaciones. Esta mejora de eficiencia puede traducirse en:

- Mayor calidad de servicio al cliente, al proporcionar respuestas rápidas y fundamentadas.
- Disminución de errores humanos en tareas repetitivas y, por tanto, reducción de costes asociados a rectificaciones.
- Fomento de la transparencia frente a organismos reguladores, gracias a la trazabilidad completa de las fuentes utilizadas.

10.3 Reflexión personal y académica

La realización del trabajo ha reforzado competencias clave del grado, como el diseño de arquitecturas distribuidas (CG6), la evaluación de software fiable (IS1) y la integración de requisitos éticos y legales (TFG1). Los mayores retos fueron la orquestación de microservicios y el desarrollo del backend; superarlos permitió consolidar conocimientos prácticos sobre ingeniería de IA aplicada y desarrollo de software.

10.4 Trabajo futuro

1. **Ampliación del corpus.** Automatizar la ingesta de nueva normativa y modelos contractuales con control de versiones.
2. **Reranking especializado.** Entrenar un *Cross-Encoder* específico para seguros en español.
3. **Fine-tuning instructivo del LLM.** Reducir ambigüedad y sesgos residuales mediante datos conversacionales de dominio.
4. **Explicabilidad y métricas ESG.** Añadir mapas de saliencia y exponer indicadores de consumo energético por transacción.

-
5. **Despliegue multi-tenant.** Migrar a Kubernetes con *Horizontal Pod Autoscaling* y aislamiento de datos.

10.4.1 Fortalecimiento de la ciberseguridad

Aunque el sistema incorpora medidas básicas (token de autenticación, validación de entradas, control de permisos y registro de actividad), se proponen estas líneas para reforzar la protección:

- **Modelo de amenazas formal.** Elaborar un análisis STRIDE [18] y alinear riesgos con OWASP Top 10 for LLM Applications [23].
- **Cifrado en reposo.** Aplicar cifrado a base de datos e índices vectoriales.
- **Gestión de secretos.** Introducir HashiCorp Vault [9] para rotación y revocación de credenciales.
- **RBAC granular.** Definir privilegios más finos sobre documentos y expedientes.
- **Pruebas de robustez y monitorización.** Desplegar baterías de prueba de inyección y alertas en tiempo real.

10.5 Visión a largo plazo

A medio plazo (2025-2027) se prevé evolucionar la plataforma hacia un modelo *Software-as-a-Service* multi-compañía e integrar módulos de detección de fraude en tiempo real. La consolidación de normas como ISO/IEC 42001 y el Reglamento Europeo de IA impulsará la demanda de soluciones RAG con gobernanza demostrable; el presente trabajo sienta las bases técnicas y metodológicas para responder a ese escenario.

En síntesis, el proyecto alcanza los objetivos planteados y establece una base sólida para la adopción de arquitecturas RAG en contextos regulados. Demuestra que la IA puede integrarse de forma responsable, auditada y generadora de valor real en el sector asegurador, abriendo la puerta a una nueva generación de servicios cognitivos orientados a la eficiencia, la transparencia y la sostenibilidad.

Appendices

Manual de Instalación

Este capítulo describe el procedimiento necesario para instalar, configurar y ejecutar el sistema de consulta inteligente de documentos desarrollado como Trabajo de Fin de Grado. La instalación se realiza mediante contenedores Docker y no requiere instalación manual de dependencias adicionales en el sistema operativo anfitrión.

A.1 Requisitos del sistema

A.1.1 Hardware mínimo recomendado

- CPU: 4 núcleos (8 recomendados)
- RAM: 8 GB mínimo (16 GB recomendado)
- Almacenamiento: al menos 10 GB libres
- GPU NVIDIA (opcional, recomendable para acelerar inferencia del modelo LLM mediante CUDA)

A.1.2 Software necesario

- Docker Engine versión 24 o superior
- Docker Compose v2
- Git (para clonar el repositorio)
- (Opcional) Cuenta en Google Cloud para generar credenciales OAuth

A.2 Clonación del repositorio

Desde la terminal, clonar el repositorio que contiene el sistema completo:

```
git clone https://github.com/kuvx/entrega_tfg/blob/v3
cd tfg_rag
```

A.3 Estructura del sistema

La raíz del proyecto contiene los siguientes componentes relevantes:

- **backend/**: código fuente del servidor FastAPI, junto con los microservicios
- **backend/vector_store/**: carpeta donde se almacenan los índices FAISS generados para cada expediente.
- **frontend/**: código fuente del frontend y de todos los componentes que lo conforman.
- **alembic/**: migraciones de base de datos mediante Alembic.
- **init_db.py**: script de inicialización automática de la base de datos PostgreSQL.
- **docker-compose.yml**: orquestador de contenedores que define todos los servicios del sistema.
- **Dockerfile.frontend / Dockerfile.backend**: instrucciones para construir los contenedores personalizados.

A.4 Variables de entorno

Las credenciales y configuraciones sensibles se definen en el archivo **backend/.env**. Un ejemplo típico de configuración es:

```
GOOGLE_CLIENT_ID=<tu_id_oauth>
GOOGLE_CLIENT_SECRET=<tu_clave_oauth>
SESSION_SECRET_KEY=<clave_secreta_aleatoria>
DATABASE_URL=postgresql://tfg_user:tfg_pass@db:5432/tfg_rag
API_URL=http://localhost:5000
OLLAMA_API=http://ollama:11434/api/generate
```

A.5 Configuración del puerto y OAuth

Por defecto, el sistema expone:

- El **backend** en el puerto 5000
- El **frontend** en el puerto 3000 (redirigido al 80 interno)
- **Ollama** en el puerto 11434

Estos puertos se pueden modificar fácilmente editando la sección correspondiente del archivo `docker-compose.yml`, por ejemplo:

```
frontend:
  ports:
    - '3000:80'
backend:
  ports:
    - '5000:5000'
```

Si se cambia el puerto del frontend, se deberá también actualizar en Google Cloud Console los **URI de redirección permitidos** para OAuth. Esto se realiza desde <https://console.cloud.google.com/apis/credentials>, editando el ID de cliente:

- URI de redirección: `http://localhost:3000`
- Orígenes de JavaScript autorizados: `http://localhost:3000`

A.6 Obtención de claves OAuth

Para poder autenticar usuarios mediante Google en local, cada desarrollador deberá:

1. Crear un nuevo proyecto en Google Cloud Console
2. Activar la API “OAuth 2.0 Client ID”
3. Registrar los URI mencionados (puerto del frontend)
4. Generar las credenciales y copiar el `client_id` y `client_secret` en el archivo `.env`

A.7 Ejecución del sistema

Desde la raíz del proyecto, ejecutar el siguiente comando:

```
docker compose up --build
```

Esto levantará automáticamente los siguientes servicios:

- **PostgreSQL** (servicio `db`): almacén de documentos, expedientes y usuarios.
- **Ollama** (servicio `ollama`): servidor del modelo LLM con soporte GPU.
- **Backend** (servicio `backend`): servidor FastAPI y lógica de negocio.
- **Frontend** (servicio `frontend`): interfaz de usuario basada en React.

El backend ejecuta automáticamente el script `init_db.py` al iniciar, aplicando migraciones y asegurando que las tablas están inicializadas.

A.8 Uso del sistema

Una vez desplegado, se puede acceder a la interfaz desde `http://localhost:3000`. El flujo de uso es el siguiente:

- El usuario inicia sesión mediante su cuenta de Google.
- Puede crear uno o varios expedientes.
- Puede subir documentos en distintos formatos (PDF, docx, txt, eml, html) a cada expediente. Estos se almacenan en la base de datos y se fragmentan para generar embeddings semánticos.
- El índice FAISS asociado se almacena automáticamente en `backend/vector_store/`.
- Al iniciar un chat en un expediente, el sistema recupera fragmentos relevantes mediante búsqueda semántica y los utiliza como contexto para la respuesta del modelo LLM.

A.9 Regeneración de índices

Siempre que se sube o elimina un documento, el sistema elimina y regenera los índices FAISS correspondientes, garantizando la coherencia entre los datos de entrada y el contexto consultado.

Registro y visualización de logs

Los eventos del backend se registran en archivos planos dentro de la carpeta `backend/logs/`, en concreto en:

- `logs/tfg_rag.log`: fichero principal de registro.
- `logs/tfg_rag.log.1`, `.2`, `.3`: copias de seguridad de los últimos 3 archivos rotados.

La rotación se produce automáticamente cuando el tamaño de `tfg_rag.log` supera 5 MB. Se mantienen hasta 3 copias antiguas para evitar pérdida de información histórica.

Niveles de log

- **INFO**: sucesos normales (inicio de procesos, recuento de fragmentos, carga de modelos, etc.).
- **WARNING**: situaciones inesperadas pero no críticas (formatos no soportados, resultados de baja calidad, etc.).
- **ERROR**: fallos críticos (excepciones en lectura de archivos, llamadas a servicios, generación de embeddings, etc.).
- **DEBUG**: trazas detalladas de fragmentos descartados y contenido de prompts.

Comandos básicos para inspección Dentro de la carpeta `backend/`

- `tail -f logs/tfg_rag.log` Visualiza en tiempo real las nuevas entradas de log.
- `less logs/tfg_rag.log.1` Navega por una copia rotada de log.
- `grep ERROR logs/*.log*` Filtra sólo las entradas con nivel `ERROR`.

Registro de eventos del frontend La ruta `HTTP POST /logs` permite que el frontend envíe eventos personalizados al mismo logger. El payload JSON debe incluir:

- `event`: descripción del suceso.
- `level`: nivel de log (`info`, `warning`, `error`).
- `metadata`: datos adicionales (por ejemplo, usuario, identificadores).

Estos registros se intercalan en el mismo archivo `tfg_rag.log`, con prefijo `[FRONT]` y la etiqueta de nivel correspondiente.

A.10 Desinstalación

Para eliminar todos los contenedores, volúmenes y datos locales:

```
docker compose down -v --remove-orphans
```

Esto detiene y elimina completamente el sistema, incluyendo la base de datos y los índices semánticos almacenados localmente.

A.11 Consideraciones finales

El sistema ha sido diseñado para ser modular, portable y seguro. Todos los componentes están encapsulados en contenedores y pueden adaptarse a distintos entornos de despliegue modificando únicamente variables en `.env` o el archivo `docker-compose.yml`. La autenticación OAuth requiere configurar correctamente los puertos y redirecciones en la consola de Google. No se deben publicar claves ni archivos `.env` en repositorios públicos.

Manual de Usuario

Este capítulo describe, paso a paso, cómo acceder y utilizar la aplicación web desarrollada para la consulta inteligente de documentación aseguradora. Su finalidad es que cualquier usuario —sin necesidad de conocimientos técnicos— pueda cargar expedientes, introducir documentos y realizar preguntas en lenguaje natural obteniendo respuestas precisas y trazables.

B.1 Acceso a la aplicación

1. Abra un navegador web moderno (Chrome, Firefox, Edge o Safari).
2. Escriba la dirección `http://localhost:3000` (o la URL indicada por el administrador si se ha desplegado en un servidor distinto).
3. Pulse en “**Iniciar sesión con Google**”. *Nota:* la primera vez, Google le mostrará una pantalla de consentimiento donde debe permitir a la aplicación conocer su correo electrónico básico.

B.2 Estructura de la interfaz

Barra lateral izquierda: ▪ lista de **expedientes**. Desde aquí se crean, renombran o eliminan.

- lista de **documentos** del expediente seleccionado. Aquí se pueden subir documentos, eliminarlos, descargarlos y buscarlos por nombre

Zona central derecha: área de **chat**. Contiene:

- Historial de mensajes de la conversación.
- Lista de chats anteriores
- Cuadro de texto para escribir preguntas.

-
- Botón **Enviar**.

Columna derecha: Preguntas sugeridas

Icono superior derecho: nombre del usuario y opción **Cerrar sesión**.

B.3 Gestión de expedientes

B.3.1 Crear un nuevo expediente

1. Haga clic en el icono “+ **Nuevo expediente**”.
2. Asigne un nombre descriptivo (por ejemplo, “*Siniestro_Póliza_1234*”) y confirme.
3. El nuevo expediente aparecerá seleccionado; todas las operaciones posteriores se aplicarán a él.

B.3.2 Renombrar o eliminar

1. Pase el cursor sobre el nombre del expediente y pulse el icono *lápiz* para editar.
2. Para eliminarlo, pulse el icono *papelera*. **Precaución:** se borrarán los documentos, los índices y las conversaciones asociadas.

B.4 Carga de documentos

1. Con un expediente activo, arrastre archivos **.pdf**, **.docx**, **.txt**, **.eml**, **.html** (actualmente los formatos soportados) a la zona “Arrastrar archivos aquí” o pulse “Seleccionar archivo”.
2. Espere a que la barra de progreso llegue al 100 %.
3. El sistema procesa el documento: extrae el texto, lo divide en fragmentos y lo indexa. Mientras tanto, la barra de “progreso” aparecerá avanzando; al terminar, desaparecerá de la zona de subida y aparecerá como uno de los documentos del expediente.

Buenas prácticas:

- Evite subir documentos con datos personales sensibles sin consentimiento.
- Use nombres de archivo descriptivos; ayudarán a identificar la fuente en las respuestas y a filtrarlas por nombre si tiene muchos documentos en un expediente.

B.5 Uso del chat

B.5.1 Iniciar una conversación

1. Pulse “+ **Nuevo chat**”. El historial se vaciará y el sistema asociará la nueva conversación al expediente actual.
2. Escriba la pregunta en lenguaje natural. Ejemplos:
 - “¿Qué periodo de carencia aplica a la cobertura de hospitalización?”
 - “¿Cuál es la suma asegurada para responsabilidad civil en este contrato?”
3. Pulse el icono de **Enviar** o presione la tecla **Enter**.

B.5.2 Interpretación de la respuesta

- La respuesta aparecerá en segundos. Justo debajo, se listan los fragmentos documentales utilizados (fuentes).
- Si la respuesta parece incompleta, formule una repregunta aprovechando el contexto conversacional (p.ej., “¿Y qué exclusiones existen?”).

B.5.3 Atajos y sugerencias

- En la columna derecha, la aplicación propone preguntas frecuentes; pulse sobre cualquiera para lanzarla automáticamente.

B.6 Gestión de conversaciones

- El historial de chats aparece en la lista de chats que hay a la izquierda del recuadro de conversación del chat activo. Seleccione uno para repasar preguntas y respuestas de chats anteriores.
- Para eliminar un chat, pulse el icono *papelera* junto a su nombre.

B.7 Cierre de sesión

En cualquier momento, haga clic en su avatar (icono superior derecho con su foto de perfil) y elija **Cerrar sesión**. Esto invalida la sesión y volverá a la página inicial.

La información quedará asociada a su cuenta, puede cerrar sesión y al volver a iniciar sesión con la misma cuenta aparecerán de nuevo todos sus expedientes, documentos, chats y mensajes

Manual del Desarrollador

Este capítulo describe la arquitectura interna, la organización del repositorio y las instrucciones prácticas para extender, depurar y desplegar el sistema. Se asume familiaridad con *Python 3.10*, *React 18*, Docker & docker-compose, SQLAlchemy y los fundamentos de RETRIEVAL-AUGMENTED GENERATION (RAG).

C.1 Visión general

- **Backend** (contenedor `tfg-backend`)
 - API REST sobre FastAPI.
 - Servicios de ingesta, embeddings, indexado FAISS, recuperación y generación.
 - Base de datos PostgreSQL (→ ORM SQLAlchemy).
 - LLM local alojado en un segundo contenedor (`ollama`).
- **Frontend** (contenedor `tfg-frontend`) SPA React que consume la API y ofrece la interfaz de chat/documentos.
- **Persistencia BBDD**: metadatos y binarios *Vector store*: ficheros `.index/.json` por expediente en `backend/vector_store`.

C.2 Estructura del repositorio

```
backend/  
|-- alembic/          -- migraciones SQL  
|-- app/  
|   |-- api/          -- routers FastAPI (endpoints)  
|   |-- core/          -- config, db, modelos ORM  
|   |-- middleware/    -- middlewares
```

```
|  '-- services/      -- lógica RAG (ingesta, FAISS, LLM, etc.)
|-- vector_store/    -- índices FAISS *.index + metadatos *.json
|-- requirements.txt
'-- init_db.py        -- aplica Alembic al arranque

frontend/
|-- src/components    -- GUI (ExpedienteList, ChatList, ...)
'-- src/utils         -- logger.js, helpers

docker-compose.yml
```

C.3 Backend

C.3.1 Capa core

config.py Carga variables de entorno con `python-dotenv`.
Si `DATABASE_URL` falta lanza `ValueError`.
Tip: define `SQL_ECHO=0/1` para activar el *trace SQL*.

database.py `engine = create_engine(DATABASE_URL, echo=bool(SQL_ECHO))`
El generador `get_db()` se inyecta en cada router.

models.py Tablas `User`, `Expediente`, `Document`,
`Chat`, `Message`.
Cascadas “all, delete-orphan” para cumplir RGPD.

C.3.2 Capa middleware

load_user_middleware.py Carga el objeto `user` en `request.state` a partir del correo almacenado en la `SessionMiddleware`.

logging_middleware.py Registra cada petición/respuesta con latencia y usuario en `tfg_rag.log`.

C.3.3 Servicios RAG (app/services)

Módulo	Rol	Detalles clave
<code>embedding_service.py</code>	Embeddings	Carga <code>intfloat/e5-large-v2</code> . GPU-first con <code>torch.cuda.is_available()</code> . Devuelve vectores L2 normalizados; loggeo exhaustivo de cargas y errores.
<code>ingestion_service.py</code>	Ingesta	Extractores PDF/DOCX/HTML/EML, limpieza Unicode, normalización (fechas, importes), segmentación <1 100 caracteres, fusión de trozos cortos. Parámetros globales <code>MIN_CHARS</code> , <code>MAX_CHARS</code> .
<code>index_service.py</code>	FAISS	Regenera índices por expediente (<code>IndexFlatIP</code>). Persiste <code>.index</code> y <code>.json</code> en <code>backend/vector_store</code> . Permite eliminar y recargar índices on-the-fly.
<code>retrieval_service.py</code>	Búsqueda	Vectoriza la consulta, consulta FAISS (<code>TOP_K=5</code>), re-ranking opcional con Cross-Encoder (<code>ms-marco-MiniLM-L12-v2</code>). Expansión semántica en <code>normalize_query()</code> .
<code>generation_service.py</code>	Pipeline	Combina recuperación, generación de <i>prompt</i> (<code>LangChain</code>), memoria por chat y llamada a Ollama (<code>DEFAULT_MODEL = nous-hermes2</code>).

Parámetros globales y *flags* modificables

- **Embeddings** : `MODEL_NAME` y `EMBEDDING_DIM` controlan el modelo ST usado y la dimensión (por defecto 1 024). Cambiar `MODEL_NAME` a otro modelo (*e.g.* `sentence-transformers/all-mpnet-base-v2`) sólo requiere modificar la constante: la clase `Embedder` recompilará y almacenará la nueva dimensión en `self.dim`.
- **Ingesta** : `MIN_CHARS`/`MAX_CHARS` definen la longitud de cada *chunk*. Pueden tunearse para adaptarse a la longitud máxima de entrada del LLM (context window). Los patrones `NOISE_PATTERNS`, `EXCLUDE_PATTERNS` y `SEGMENT_PATTERNS` permiten descartar o partir texto con expresiones regulares sin tocar la lógica de negocio.
- **Índices** : `VECTOR_DIR` apunta al directorio donde FAISS serializa los índices; en producción puede mapearse a un volumen persistente. El tipo de índice (`IndexFlatIP`) está centralizado en `index_service.py` y puede sustituirse (ej. `IndexIVFFlat`) manteniendo la API.
- **Recuperación** : `TOP_K` limita los candidatos iniciales; `USE_RERANKING` habilita/deshabilita la fase Cross-Encoder; `MAX_RETURNED` recorta lo que se pasa al LLM. `RERANKER_MODEL` acepta cualquier Cross-Encoder HF compatible.

-
- **Generación**: `DEFAULT_MODEL` define el modelo alojado en Ollama; `MAX_CONTEXT_CHUNKS` recorta el número de fragmentos que alimentan el *prompt*; `RERANK_ENABLED` permite activar el re-ranking caso a caso vía parámetro de función.

Logging y depuración Todos los módulos comparten el `logger` declarado con nombre `"tfg_rag"`, por lo que basta con ajustar el *handler* principal (en `app/main.py`) para cambiar formato, rotación o nivel global. Cada servicio anota:

- eventos críticos (`error/exception`) con `exc_info=True` para volcado de *tracebacks*;
- pasos intermedios (`info`) como conteos de fragmentos, tamaño de índices, tiempo de ejecución;
- inspección de texto (`debug`) truncada a 100–200 caracteres para no inundar los logs.

Personalización de Ollama y puertos La URL del servicio generativo se inyecta desde `backend/.env` mediante `OLLAMA_API=http://ollama:11434/api/generate`. Si se quiere exponer Ollama en otro contenedor, bastará con cambiar la variable y—si procede—abrir un nuevo puerto en `docker-compose`. De igual forma, el `PORT` del backend (5000) y del frontend (3000) pueden modificarse sin tocar código:

```
services:
  backend:
    ports:
      - '8080:5000'    # HOST:CONTAINER
  frontend:
    ports:
      - '4200:80'
```

Tras cambiar el puerto del backend hay que actualizar `API_URL` en el `.env` del frontend; y, si se usa autenticación Google OAuth, añadir las nuevas *redirect URIs* en la consola de Cloud IAM para evitar el error `"redirect_uri mismatch"`.

Flujo interno de llamadas

1. **ingestion_service** fragmenta documentos y guarda en BBDD.
2. **index_service** lee los *chunks*, genera embeddings con **embedding_service** y crea el índice FAISS.
3. **retrieval_service** normaliza la consulta, genera su embedding, consulta FAISS, (opcional) re-rankea.
4. **generation_service** construye el *prompt* con LangChain, llama a Ollama, actualiza la memoria y persiste el mensaje.

Buenas prácticas de extensión

- **Nuevos formatos**: heredar de `BaseExtractor`, registrar en `ExtractorFactory.extractors`.
- **Pre-procesado**: añadir *regex* a `NOISE_PATTERNS` o lógica nueva antes de `segment_by_length`.
- **Indexación híbrida**: sustituir `IndexFlatIP` por `IndexIVFPQ` si se requieren millones de vectores; mantener la interfaz `add/search`.
- **LLM alternativo**: sólo hay que instalar el modelo en Ollama y apuntar `DEFAULT_MODEL`.
- **A/B testing**: el parámetro `model` de `generate_answer` permite pasar el ID del modelo en caliente para comparar resultados sin reiniciar el backend.

C.3.4 Routers FastAPI (app/api)

- **Autenticación** POST `/auth/google` – valida *id-token*, crea/actualiza usuario y devuelve expedientes. Variables: `GOOGLE_CLIENT_ID`, `GOOGLE_CLIENT_SECRET`.
- **Documentos**
 - POST `/documents/upload` guarda *file_data* y reconstruye FAISS
 - DELETE `/documents/{id}` regenera índice sin el documento
- **Expedientes** CRUD completo; DELETE borra chats, documentos e índice en disco.
- **Chats y mensajes** Rutas para crear, renombrar, listar y eliminar chats; añade mensajes y mantiene coherencia con la memoria `LangChain`.
- **Query** POST `/query/search` devuelve fragmentos + puntuaciones; POST `/query/generate` ejecuta el ciclo RAG completo.
- **Logs** POST `/api/logs` centraliza eventos del frontend.

C.4 Frontend (React 18)

Componente	Función	Notas de interés
<code>ExpedienteList</code>	CRUD expedientes	Actualiza hijos vía props; considera usar <code>Context</code> .
<code>FileUpload</code>	Subida de PDFs	Usa <code>FormData</code> ; loader mientras espera.
<code>FileList</code>	Listado/borra documentos	Refetch cada cambio de <code>expedienteId</code> .
<code>ChatList</code>	Historial de chats	Maneja <code>selectedChat</code> .
<code>QueryLLM</code>	Área de conversación	Llama a <code>/query/generate</code> ; streaming pendiente de m
<code>ChatHeader</code>	Título + crear chat	Botón “+ Chat”.
<code>SuggestedQuestions</code>	Preguntas rápidas	Array local, fácil de parametrizar vía API.
<code>UserProfileButton</code>	Avatar + logout	Limpia <code>localStorage</code> .
<code>Login</code>	Flujo OAuth Google	Guarda token y redirige a dashboard.

C.4.1 Utilidades

`utils/logger.js` Pequeña envoltura sobre `console.log`; envía también eventos al backend vía `POST /api/logs`. Útil para correlacionar fallos.

C.5 Configuración y variables de entorno

Clave	Descripción
<code>GOOGLE_CLIENT_ID</code>	OAuth web client ID. Se registra en Google Cloud Console.
<code>GOOGLE_CLIENT_SECRET</code>	Secreta, <i>no</i> versionar.
<code>DATABASE_URL</code>	Cadena SQLAlchemy.
<code>SESSION_SECRET_KEY</code>	Clave 32 bytes hex para <code>SessionMiddleware</code> .
<code>OLLAMA_API</code>	URL interna del LLM.

Cambio de puertos Si se expone la aplicación en un puerto distinto basta con:

1. Modificar el `ports` del servicio correspondiente en `docker-compose.yml`:

```
frontend:
  ports:
    - '8080:80' # host:container
backend:
  ports:
    - '6000:5000'
```

2. Actualizar `API_URL` en `frontend/.env`.
3. Añadir la nueva URL (`http://localhost:8080`) en *Authorized JavaScript origins* y la ruta (`http://localhost:8080`) en *Authorized redirect URIs* del cliente OAuth.

C.6 Contenerización y despliegue

- **Backend** se construye con `Dockerfile.backend`. Copia `requirements.txt`, instala dependencias GPU (`pytorch-cuda`), expone 5000.
- **Frontend** usa `Dockerfile.frontend`. Construye vite y sirve artefactos vía `nginx:alpine`.
- **GPU sharing** Tanto `backend` como `ollama` llevan `runtime: nvidia`. En entornos sin GPU se elimina ese atributo y todo funciona (más lento) en CPU.
- **Persistencia** Volume `postgres_data` (BBDD) y `ollama` (modelos LLM), + carpeta montada `./backend/vector_store` para FAISS.

C.7 Flujo completo RAG

1. **Ingesta:** usuario sube PDF → `ingestion_service` trocea, genera embeddings y `index_service` actualiza FAISS.
2. **Consulta:** pregunta → embeddings → FAISS → (opcional) Cross-Encoder.
3. **Generación:** LangChain concatena *context + history*, llama al LLM vía Ollama, escribe respuesta y guarda en `Message`.
4. **Frontend:** muestra texto + fuentes; preguntas sugeridas re-utilizan el mismo endpoint.

C.8 Buenas prácticas y extensiones

- **Tests:** use `pytest` + `httpx.AsyncClient` para routers.
- **Monitorización:** Prometheus + Loki (log-scrape `tfg_rag.log`).
- **Embeddings** personalizados: basta con cambiar `MODEL_NAME` en `embedding_service.py` y regenerar índices.
- **Escalado LLM:** montar varias réplicas de `ollama` detrás de un balanceador TCP; configurar `OLLAMA_API` con la IP del proxy.

Fin del Manual del Desarrollador

Anexos

11.1 Glosario y lista de siglas

11.1.1 Siglas

Sigla	Expansión (ES / EN)	Descripción breve
IA / AI	Inteligencia Artificial / Artificial Intelligence	Disciplina que diseña sistemas capaces de ejecutar tareas que requieren “inteligencia” humana.
IR	Information Retrieval	Rama que localiza documentos o fragmentos relevantes dentro de un corpus.
RAG	Retrieval-Augmented Generation	Paradigma que combina IR con generación de lenguaje natural para producir respuestas fundamentadas.
LLM	Large Language Model	Modelo de lenguaje de gran tamaño entrenado sobre enormes volúmenes de texto.
ANN	Approximate Nearest Neighbor	Técnica para hallar rápidamente los vectores más cercanos en espacios de alta dimensión.
GPU	Graphics Processing Unit	Procesador de cómputo paralelo masivo; acelera IA.
CUDA	Compute Unified Device Architecture	Plataforma/API de NVIDIA para ejecutar código en GPU.
VM	Virtual Machine	Entorno que emula hardware para aislar sistemas.
API	Application Programming Interface	Conjunto de reglas que permite a dos sistemas comunicarse.

Sigla	Expansión (ES / EN)	Descripción breve
REST	Representational State Transfer	Estilo arquitectónico para diseñar APIs sobre HTTP.
Docker	—	Plataforma que empaqueta aplicaciones y dependencias en contenedores.
FAISS	Facebook AI Similarity Search	Librería para búsqueda de similitud e índices vectoriales.
SBERT	Sentence-BERT	Variante de BERT que genera <i>embeddings</i> semánticos de frases.
FastAPI	—	<i>Framework</i> web asíncrono en Python, orientado a APIs.
SQL	Structured Query Language	Lenguaje estándar en BBDD relacionales.
NoSQL	“Not only SQL”	Familia de BBDD no relacionales.
JSON	JavaScript Object Notation	Formato ligero de intercambio de datos.
JWT	JSON Web Token	Estándar para transmitir información firmada como JSON.
OAuth	Open Authorization	Protocolo para acceso delegado seguro sin compartir credenciales.
RBAC	Role-Based Access Control	Modelo de control de acceso basado en roles.
AES	Advanced Encryption Standard	Estándar de cifrado simétrico de clave secreta.
TDE	Transparent Data Encryption	Cifrado en reposo integrado en motores de bases de datos.
SAST	Static Application Security Testing	Análisis estático de código para detectar vulnerabilidades.
OWASP	Open Web Application Security Project	Fundación que publica estándares y guías de seguridad web.
GHA	GitHub Actions	Plataforma de integración continua y automatización de flujos de trabajo.

Términos técnicos

Término	Definición breve
RAG (Retrieval-Augmented Generation)	Técnica que combina un LLM con un buscador externo para inyectar documentos relevantes en la respuesta.

Término	Definición breve
PLN / NLP	Disciplina que estudia la interacción entre lenguaje humano y máquinas mediante técnicas lingüísticas y de aprendizaje automático.
LLM (Large Language Model)	Red neuronal de lenguaje con miles de millones de parámetros entrenada en grandes corpus.
Embedding	Representación vectorial densa que codifica el significado semántico de un texto en un espacio numérico.
Vector Store	Base especializada que indexa <i>embeddings</i> y permite búsquedas por similitud.
BM25	Algoritmo estadístico de recuperación basado en frecuencia de término e inversa de frecuencia de documento.
Context Window	Máximo número de tokens que un LLM procesa simultáneamente.
Token	Unidad mínima de texto tras la tokenización (palabra, sub-palabra o símbolo).
Prompt Injection	Ataque que introduce instrucciones maliciosas para alterar la salida de un LLM.
Latency	Tiempo desde la solicitud hasta la respuesta (ms).
Throughput	Peticiones o tokens procesados por segundo, indicador de capacidad.
GPU	Unidad de procesamiento gráfico que acelera inferencia de redes neuronales.
CI/CD	Prácticas DevOps de integración y entrega/despliegue continuos.
IaC (Infrastructure-as-Code)	Gestión declarativa de infraestructura mediante código versionado (p. ej. Terraform).
Observabilidad	Conjunto de métricas, trazas y logs para entender el estado interno de un sistema.
WCAG 2.1	Recomendación W3C con criterios de accesibilidad web.
ISO/IEC 42001	Norma de sistemas de gestión para aplicaciones de IA (gobernanza y riesgo).
STRIDE	Modelo de amenazas: Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege.
k-NN (k-Nearest Neighbors)	Algoritmo que devuelve los k vectores más cercanos en un espacio de <i>embeddings</i> .
MLOps	Extensión de DevOps que gestiona modelos, datos y experimentos a lo largo del ciclo de vida.
RBAC (Role-Based Access Control)	Sistema que asigna permisos a objetos según los roles otorgados al usuario.
AES-256	Cifrado simétrico con clave de 256 bits, adoptado como estándar por la NIST.
Transparent Data Encryption (TDE)	Cifrado automático de datos en reposo implementado por el motor de la base de datos.

Término	Definición breve
SAST (Static Application Security Testing)	Análisis de código fuente sin ejecución para descubrir vulnerabilidades.
OWASP Top 10 for LLM	Lista de riesgos de seguridad específicos de aplicaciones basadas en LLM publicada por OWASP.
HashiCorp Vault	Herramienta para almacenar, rotar y auditar secretos y claves de forma centralizada.
Prometheus	Sistema de monitorización y base de series temporales para métricas.
Grafana	Plataforma de visualización y alerta que consume datos de Prometheus u otras fuentes.
Loki	Agregador de logs diseñado para integrarse con Grafana y Prometheus.
Falco	Motor que detecta comportamientos anómalos en contenedores mediante reglas en tiempo real.
Sysdig Secure	Suite para análisis forense y cumplimiento de seguridad en entornos contenedorizados.
Penetration Testing	Pruebas que simulan ataques reales con el fin de evaluar la robustez del sistema.

Bibliografía

- [1] M. van Bekkum, F. Zuiderveen Borgesius y T. Heskes. *AI, insurance, discrimination and unfair differentiation: An overview and research agenda*. 2024. URL: <https://arxiv.org/abs/2401.11892> (visitado 13-05-2025).
- [2] Daniel Cer et al. «Universal Sentence Encoder». En: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 2018, págs. 169-174. DOI: 10.18653/v1/D18-2024. URL: <https://aclanthology.org/D18-2024> (visitado 20-05-2025).
- [3] Data Center Market. *Así está cambiando la IA al sector asegurador*. 2025. URL: <https://www.datacentermarket.es/inteligencia-artificial/asi-esta-cambiando-la-ia-al-sector-asegurador> (visitado 12-05-2025).
- [4] Jacob Devlin et al. «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding». En: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2019, págs. 4171-4186. DOI: 10.18653/v1/N19-1423. URL: <https://aclanthology.org/N19-1423> (visitado 20-05-2025).
- [5] EIOPA. *Informe sobre la digitalización del sector asegurador europeo*. 2024. URL: <https://www.conorsegurosdigital.com/almacen/pdf/numero-20-es.pdf> (visitado 10-05-2025).
- [6] El País. *Fraude al seguro: o cómo usar un vídeo de Instagram para detectar un caso*. 2025. URL: <https://elpais.com/economia/2025-03-19/fraude-al-seguro-o-como-usar-un-video-de-instagram-de-un-coche-tuneado-para-detectar-un-caso.html> (visitado 15-05-2025).
- [7] European Union. *Regulation (EU) 2024/1689 of the European Parliament and of the Council of 13 June 2024 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act)*. In force since 1 August 2024. 2024. URL: <https://eur-lex.europa.eu/eli/reg/2024/1689/oj> (visitado 16-05-2025).
- [8] *Guía sobre generación de datos sintéticos*. Agencia Española de Protección de Datos. 2025. URL: <https://www.aepd.es/prensa-y-comunicacion/notas-de-prensa/guia-sobre-generacion-de-datos-sinteticos> (visitado 18-05-2025).
- [9] HashiCorp. *Vault Documentation*. 2025. URL: <https://developer.hashicorp.com/vault/docs> (visitado 25-05-2025).

-
- [10] IBM. *Qué es la IA en los seguros*. 2024. URL: <https://www.ibm.com/es-es/think/topics/ai-in-insurance> (visitado 12-05-2025).
- [11] International Organization for Standardization. *ISO/IEC 42001:2023 — Artificial Intelligence Management System (AIMS)*. International standard. 2023. URL: <https://www.iso.org/standard/81230.html> (visitado 17-05-2025).
- [12] Gautier Izacard y Edouard Grave. «Distilling Knowledge from Reader to Retriever for Question Answering». En: *arXiv* (2021). arXiv: 2101.00294 [cs.CL]. URL: <https://arxiv.org/abs/2101.00294> (visitado 20-05-2025).
- [13] Jeff Johnson, Matthijs Douze y Hervé Jégou. «Billion-Scale Similarity Search with GPUs». En: *IEEE Transactions on Big Data* 7.3 (2019), págs. 535-547. DOI: 10.1109/TBDATA.2019.2921575.
- [14] Vladimir Karpukhin et al. «Dense Passage Retrieval for Open-Domain Question Answering». En: *arXiv* (2020). arXiv: 2004.04906 [cs.CL]. URL: <https://arxiv.org/abs/2004.04906> (visitado 20-05-2025).
- [15] KPMG. *Avance de la IA en la industria de seguros*. 2024. URL: <https://kpmg.com/mx/es/home/tendencias/2024/12/avance-de-la-ia-en-la-industria-de-seguros.html> (visitado 14-05-2025).
- [16] Patrick Lewis et al. «Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks». En: *Advances in Neural Information Processing Systems* 33 (2020), págs. 9459-9474. arXiv: 2005.11401 [cs.CL]. URL: <https://arxiv.org/abs/2005.11401> (visitado 20-05-2025).
- [17] Xiang Lin et al. «Few-Shot Learning with Retrieval Augmented Generation». En: *Findings of the Association for Computational Linguistics: EMNLP 2021*. 2021, págs. 1951-1961. arXiv: 2112.04426 [cs.CL]. URL: <https://arxiv.org/abs/2112.04426> (visitado 20-05-2025).
- [18] Microsoft Security Development Lifecycle. *Threat Modeling*. 2023. URL: <https://learn.microsoft.com/security/devsecops/threat-modeling> (visitado 25-05-2025).
- [19] Tomas Mikolov et al. «Efficient Estimation of Word Representations in Vector Space». En: *arXiv* (2013). arXiv: 1301.3781 [cs.CL]. URL: <https://arxiv.org/abs/1301.3781> (visitado 20-05-2025).
- [20] Minsait e ICEA. *IX Termómetro de Inteligencia Artificial y Data en el sector asegurador español*. 2024. URL: <https://www.minsait.com/es/actualidad/media-room/ocho-de-cada-diez-entidades-aseguradoras-ya-trabajan-con-inteligencia> (visitado 10-05-2025).
- [21] msg life Iberia. *Aprobada la AI Act: ¿Cómo afecta al sector de los seguros?* 2024. URL: <https://msg-insurance-suite.com/es/blog/rethinking-insurance/aprobada-la-ai-act-como-afecta-la-ley-de-regulacion-de-la-ia-al-sector-de-los-seguros> (visitado 15-05-2025).

-
- [22] *OECD Recommendation of the Council on Artificial Intelligence*. Organisation for Economic Co-operation y Development. 2019. URL: <https://legalinstruments.oecd.org/en/instruments/oecd-legal-0449> (visitado 17-05-2025).
- [23] OWASP Foundation. *OWASP Top 10 for Large Language Model Applications*. 2024. URL: <https://owasp.org/www-project-top-10-for-large-language-model-applications/> (visitado 25-05-2025).
- [24] Colin Raffel et al. «Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer». En: *Journal of Machine Learning Research* 21.140 (2020), págs. 1-67. URL: <http://jmlr.org/papers/v21/20-074.html> (visitado 20-05-2025).
- [25] *Recommendation on the Ethics of Artificial Intelligence*. UNESCO. 2021. URL: <https://www.unesco.org/en/articles/recommendation-ethics-artificial-intelligence> (visitado 17-05-2025).
- [26] Nils Reimers e Iryna Gurevych. «Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks». En: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. 2019, págs. 3982-3992. DOI: 10.18653/v1/D19-1410. URL: <https://aclanthology.org/D19-1410> (visitado 20-05-2025).
- [27] Nandan Thakur et al. «BEIR: A Heterogeneous Benchmark for Zero-Shot Evaluation of Information Retrieval Models». En: *arXiv* (2021). arXiv: 2104.08663 [cs.IR]. URL: <https://arxiv.org/abs/2104.08663> (visitado 20-05-2025).
- [28] Ashish Vaswani et al. «Attention Is All You Need». En: *Advances in Neural Information Processing Systems* 30 (2017), págs. 5998-6008. arXiv: 1706.03762 [cs.CL]. URL: <https://arxiv.org/abs/1706.03762> (visitado 20-05-2025).
- [29] Wayne Xin Zhao et al. «A Survey of Hallucination in Natural Language Generation». En: *arXiv* (2023). arXiv: 2302.03494 [cs.CL]. URL: <https://arxiv.org/abs/2302.03494> (visitado 20-05-2025).
