uc3m | Universidad Carlos III de Madrid

Grado en Ingeniería Informática

2025-2026

Trabajo Fin de Grado

"Sistema basado en procesamiento de lenguaje natural para la generación de filtros en sistemas de búsqueda"

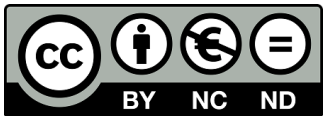
Rosa Reyes Concepción

Tutores

Eduardo Cibrián Sánchez

Fabio Galicia García de Yébenes

Leganés, Madrid, Junio 2026



RESUMEN

Hoy en día, los catálogos de productos digitales manejan un volumen de información abrumador, implementando decenas de atributos técnicos que se organizan en taxonomías cada vez más complejas. Para encontrar algo concreto, el usuario suele tener que buscar en esa estructura interna y configurar manualmente un buen número de filtros. En consecuencia, esto genera bastante fricción y alarga muchísimo la búsqueda. En este proyecto, aterrizamos ese problema general de la industria en un caso muy real: se analiza el catálogo de materiales de construcción de la plataforma Stylib.

Para resolver este factor limitante, este Trabajo de Fin de Grado propone un sistema intermediario capaz de traducir, de forma natural, las consultas de los usuarios a los filtros estructurados que necesita el motor de búsqueda. La solución cobra vida a través de una arquitectura multiagente de tres etapas. El primer agente extrae las entidades y condiciones lógicas apoyándose en la taxonomía del catálogo. Después, un segundo agente toma el relevo y traduce esas condiciones a una sintaxis de filtros de manera predefinida. Y por último, como la inteligencia artificial a veces se equivoca, un tercer agente hace de juez, y valida el resultado y corrige los fallos mediante reintentos. Al final, la salida se serializa en una URL codificada lista para el *frontend*, lo cual dota de una tremenda escalabilidad al proyecto, ya que el sistema queda desacoplado y se podría llegar a usar con diferentes bases de datos sin apenas esfuerzo.

Para comprobar el funcionamiento correcto, el sistema se ha evaluado sobre un conjunto de referencia de 52 consultas, utilizando un marco de ocho métricas centrado en el F1 a nivel de token de filtro. La experimentación comparó cinco modelos de lenguaje, midiendo la calidad, la latencia y el coste. Los resultados fueron muy positivos: la mejor configuración alcanza un F1 de 0,93, logrando una clasificación de categoría perfecta y manejando con total soltura la negación y los operadores relacionales. Además, se ha comprobado que asignar un modelo potente a la etapa de extracción y uno más ligero al resto del pipeline resulta ser una estrategia muy acertada, ya que abarata drásticamente el coste sin sacrificar la calidad.

Palabras clave

Procesamiento de lenguaje natural; modelos de lenguaje de gran escala; sistemas multiagente; generación de filtros; búsqueda semántica; NL2Query.

ABSTRACT

Nowadays, digital product catalogs handle an overwhelming amount of information, implementing dozens of technical attributes organized into increasingly complex taxonomies. To find something specific, users often have to dive into that internal structure and manually set up a fair number of filters. Honestly, this adds a lot of friction and significantly slows down the search experience. In this project, we ground this industry-wide problem in a very real scenario: analyzing the construction materials catalog of the Stylib platform.

To tackle this bottleneck, this Bachelor's Thesis proposes an intermediary system that naturally translates user queries into the structured filters required by the search engine. The solution comes to life through a three-stage multi-agent architecture. The first agent extracts entities and logical conditions by relying on the catalog's taxonomy. Then, a second agent steps in and translates those conditions into a predefined filter syntax. Since artificial intelligence can sometimes miss the mark, a third agent acts as a judge: it validates the result and fixes any mistakes through retries. Ultimately, the final output is serialized into an encoded URL ready for the *frontend*. This approach brings tremendous scalability to the project, as the system remains decoupled and could easily be used with different databases with minimal effort.

To see if this actually worked, the system was evaluated on a reference set of 52 hand-annotated queries, using a framework of eight metrics built around filter-token F1. The experimentation compared five language models and several hybrid combinations, measuring not only quality, but also latency and cost. The results have been very positive: the best configuration reaches an F1 of 0.93, achieving perfect category classification and handling negation and relational operators with ease. Furthermore, we observed that assigning a powerful model to the extraction stage and a lighter one to the rest of the pipeline proves to be a highly effective strategy, as it drastically lowers the cost without sacrificing quality.

Keywords

Natural language processing; large language models; multi-agent systems; filter generation; semantic search; NL2Query.

A mi padre, por sostenerme siempre.

A mi abuela, que nunca dejó de creer en mí.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

SIGLAS

API Application Programming Interface.

CoT Chain of Thought.

CTO Chief Technology Officer.

E2E End-to-End.

IA Inteligencia Artificial.

ICL In-Context Learning.

JSON JavaScript Object Notation.

LLM Large Language Model.

LOPDGDD Ley Orgánica de Protección de Datos Personales y Garantía de los Derechos Digitales.

NER Named Entity Recognition.

NLIDB Natural Language Interfaces for Databases.

PLN Procesamiento de Lenguaje Natural.

RAG Retrieval-Augmented Generation.

REST Representational State Transfer.

RGPD Reglamento General de Protección de Datos.

TFG Trabajo de Fin de Grado.

UI User Interface.

URL Uniform Resource Locator.

UX User Experience.

1. INTRODUCCIÓN

1.1. Motivación y Contexto

En el sector de los materiales de construcción y el diseño de interiores, las plataformas de comercio electrónico manejan catálogos extraordinariamente extensos, lo que dificulta significativamente que los usuarios encuentren productos específicos [1]. Cada material posee multitud de atributos técnicos que deben organizarse mediante taxonomías y facetas de búsqueda. Este paradigma facetado es intuitivo para la exploración, pero presenta desafíos críticos cuando la información es abundante y compleja [2]. En este contexto, la empresa Stylib dispone de una base de datos estructurada mediante un potente sistema de filtrado.

Sin embargo, el uso de modelos Booleanos estrictos en la búsqueda facetada tradicional impone barreras: si una consulta no coincide exactamente con los términos del catálogo, el usuario puede perder opciones relevantes, lo que dispara el tiempo de búsqueda y el riesgo de error [1]. Para realizar una búsqueda específica (por ejemplo: «paneles acústicos negros de menos de 20 mm»), el usuario se enfrenta a una alta carga cognitiva. La literatura técnica demuestra que la formulación de consultas es la etapa que más recursos mentales consume, ya que exige que el usuario recuerde términos técnicos y entienda la estructura interna del sistema, en lugar de simplemente reconocer opciones visuales [3]. Esta complejidad genera una alta fricción, agravada por el "problema del vocabulario", donde las palabras que el usuario tiene en mente no coinciden con las etiquetas asignadas por los indexadores del catálogo [2].

En los últimos años, el avance de la Inteligencia Artificial ha permitido proponer interfaces que mitigan esta carga mental. El propósito central de este Trabajo de Fin de Grado (TFG) es diseñar un sistema capaz de traducir consultas en lenguaje natural a los filtros estructurados que requiere el motor de búsqueda. Al permitir que el usuario se exprese en su propio lenguaje, se reduce el esfuerzo de navegación y se supera la rigidez de los filtros Booleanos tradicionales, mejorando la eficiencia en la recuperación de productos complejos [1], [3].

1.2. Objetivos del Proyecto

El objetivo general de este trabajo es desarrollar y evaluar una arquitectura multiagente basada en Inteligencia Artificial capaz de transformar consultas en lenguaje natural (NL) en filtros estructurados y semánticamente correctos para un sistema de búsqueda de materiales de construcción.

Para alcanzar esta meta principal, se establecen los siguientes objetivos específicos:

1. **Diseñar un pipeline multiagente especializado:** Crear un flujo de procesamiento dividido en agentes con responsabilidades únicas (Extracción de Entidades, Mapeo de Filtros y Validación Semántica) para reducir los errores inherentes a los modelos de lenguaje monolíticos.

2. **Integrar conocimiento estructurado en el modelo de lenguaje:** Desarrollar un sistema de inyección dinámica que alimente a la IA con una taxonomía y facetas globales del catálogo, asegurando que las predicciones se ciñan a las claves y valores reales de la base de datos de Stylib.

3. **Formalizar la salida del sistema:** Implementar algoritmos de serialización que conviertan las condiciones lógicas extraídas en cadenas de filtros formales (*Filter Groups*), generando finalmente una URL codificada diseñada de forma universal para ser consumida por cualquier cliente (por ejemplo, el *frontend* de la plataforma, una API externa o un microservicio).

4. **Establecer un marco de evaluación cuantitativa:** Crear *datasets* de prueba y definir métricas precisas (F1-score a nivel de token de filtro) para medir objetivamente la precisión del sistema.

5. **Realizar una experimentación comparativa (*Model Sweep*):** Evaluar el rendimiento, la latencia y el coste computacional de distintas combinaciones de modelos LLM de última generación para encontrar la configuración óptima para cada agente del sistema.

1.3. Estructura de la Memoria

El resto de este documento académico se organiza siguiendo la estructura detallada a continuación para dar respuesta al problema planteado:

- **Capítulo 2. Marco Teórico:** Revisión de la evolución del PLN hacia los LLMs, el uso de agentes para extracción en dominios específicos (NER), las técnicas de *prompting* (Schema Linking, CoT, Reflexion) y el estado del arte en sistemas NL2Query.

- **Capítulo 3. Análisis del Sistema:** Descripción del problema detectado en Stylib, definición de la solución propuesta, especificación de requisitos funcionales y no funcionales y casos de uso.

- **Capítulo 4. Diseño del Sistema:** Arquitectura general del *pipeline* multiagente, selección del *stack* tecnológico (Python, FastAPI, Pydantic, OpenRouter, React), contratos de datos del dominio y gestión de la taxonomía.

- **Capítulo 5. Implementación Técnica:** Ingeniería de *prompts* de cada agente, orquestación del sistema, mecanismo de autocuración (*Retry Loop*) y exposición del servicio mediante una API REST.

- **Capítulo 6. Experimentación y Evaluación:** Construcción del *Golden Dataset*, marco de ocho métricas en torno al Token F1 y análisis de los experimentos (modelos uniformes, combinaciones híbridas e impacto del validador), incluyendo coste y latencia.
- **Capítulo 7. Planificación y Presupuesto:** Planificación temporal del proyecto, diagrama de Gantt y estimación de costes de recursos humanos, hardware y software.
- **Capítulo 8. Marco Regulador y Entorno Socio-Económico:** Análisis del marco normativo aplicable (RGPD, LOPDGDD, AI Act), propiedad intelectual e impacto socio-económico del sistema.
- **Capítulo 9. Conclusiones y Trabajo Futuro:** Revisión del cumplimiento de los objetivos, limitaciones encontradas y líneas de continuación del proyecto.
- **Anexo A. Declaración de uso de IA generativa:** Declaración formal sobre el uso de herramientas de IA generativa durante el desarrollo del TFG.

2. MARCO TEÓRICO

En este capítulo se presenta la evolución de las bases tecnológicas que sustentan el proyecto, siguiendo un hilo que va desde la organización formal del conocimiento mediante ontologías hasta las arquitecturas actuales de agentes de inteligencia artificial capaces de extraer información estructurada a partir de lenguaje natural. El objetivo es que el lector pueda entender no sólo qué técnicas se utilizan, sino también por qué tienen sentido en el contexto concreto de este trabajo.

2.1. Búsqueda Semántica y Evolución del Filtrado

Los primeros sistemas de recuperación de información en la Web y en catálogos digitales se han construido, en esencia, sobre la coincidencia léxica entre las palabras de la consulta y los términos almacenados en un índice invertido. En este enfoque clásico, la relevancia se estima mediante funciones de ranking como TF–IDF o BM25, que ponderan la frecuencia de aparición de cada término en los documentos frente a su frecuencia en la colección global [4]. Durante años este paradigma ha demostrado ser eficaz y muy escalable, pero arrastra una suposición fuerte: que el usuario conoce (o adivina) el vocabulario exacto del sistema y que una coincidencia de cadenas es una aproximación razonable al significado de la consulta.

Búsqueda tradicional y el papel de las ontologías. Para reducir la brecha entre el lenguaje del usuario y el vocabulario interno de la aplicación, muchos dominios técnicos han incorporado ontologías y taxonomías jerárquicas que organizan los conceptos de forma explícita [5]. En el contexto de catálogos de productos, estas ontologías permiten definir relaciones de tipo “es un” (por ejemplo, panel acústico es un tipo de revestimiento) o relaciones de equivalencia y sinonimia entre términos cercanos. De este modo, el motor de búsqueda puede expandir la consulta con sinónimos, restringirla a subcategorías relevantes o aplicar reglas de negocio basadas en la estructura jerárquica del dominio [6]. Sin embargo, este enfoque sigue siendo fundamentalmente simbólico y rígido: cualquier cambio en el vocabulario exige actualizar manualmente la ontología, y cuesta capturar matices semánticos o asociaciones más sutiles entre conceptos que los expertos usan de forma informal en su día a día.

***Embeddings* y representación vectorial del significado.** El auge del aprendizaje profundo en Procesamiento de Lenguaje Natural ha traído consigo representaciones distribuidas, conocidas como *embeddings*, que codifican palabras, frases o documentos como vectores densos en un espacio de alta dimensión [7]. En estas representaciones, la proximidad geométrica entre vectores refleja similitud semántica, de modo que términos relacionados tienden a situarse cerca entre sí. Modelos más recientes basados en *Transformers*, como BERT y sus variantes, llevan esta idea un paso más allá al ofrecer *embeddings* contex-

tuales que dependen de la frase completa, capturando mejor la desambiguación según el contexto de uso [8]. Para los sistemas de búsqueda, esto abre la puerta al paradigma del recuperación densa (*dense retrieval*), que permite representar tanto las consultas como los ítems del catálogo en un mismo espacio vectorial y recuperar candidatos relevantes mediante funciones de similitud semántica (como el producto escalar), superando las limitaciones de las coincidencias léxicas exactas [9].

Similitud semántica frente a coincidencia de palabras clave. La transición desde la clásica búsqueda por palabras clave hacia la semántica representa un salto cualitativo enorme. La verdad es que el enfoque evoluciona para ir directo a la intención real del usuario, conectando conceptos afines aunque el vocabulario empleado sea completamente distinto [10]. Gracias a esto, si alguien teclea «panel acústico», la plataforma es capaz de devolver productos etiquetados como «panel fonoabsorbente». Y lo hace de forma fluida, asimilando sinónimos, plurales o formas de hablar cotidianas de manera automática.

Toda esta flexibilidad trae consigo su propia complejidad. Para que el sistema acierte de verdad, dependemos de *embeddings* muy bien afinados al vocabulario técnico del sector. Además, perdemos un poco de transparencia a la hora de justificar por qué la máquina recomienda un material concreto frente a otro. Por eso, el escenario ideal consiste en unir lo mejor de ambos mundos [11]. En este contexto, la búsqueda semántica debe entenderse como un complemento a los mecanismos tradicionales de filtrado, no como un reemplazo: combina el significado con la precisión de las facetas estructuradas.

2.2. Modelos de Lenguaje de Gran Escala (LLMs)

La base técnica de los modelos de lenguaje de gran escala actuales se apoya en la arquitectura *Transformer*, que supuso un punto de inflexión en el tratamiento de secuencias al eliminar la dependencia de redes recurrentes o convolucionales [12]. Su pieza central es el mecanismo de autoatención (*self-attention*), que permite al modelo establecer dependencias globales entre palabras sin importar su distancia en el texto, y hacerlo de forma muy paralelizable durante el entrenamiento [12]. A medida que los Transformers se han escalado en número de parámetros y volumen de datos, han aparecido las llamadas capacidades emergentes, entre las que el aprendizaje en contexto (*In-Context Learning*, ICL) se ha convertido en una de las más llamativas [13]. Mientras que generaciones anteriores de modelos se apoyaban en arquitecturas bidireccionales o esquemas codificador–decodificador, la mayoría de los LLMs modernos han convergido hacia diseños de decodificador causal (*decoder-only*), optimizados para predecir el siguiente token y resolver tareas complejas a través de la generación de texto [13].

El *ICL* introduce un modo de trabajo diferente: los modelos realizan predicciones basándose únicamente en el contexto proporcionado en el *prompt*, sin necesidad de actualizar explícitamente sus pesos mediante gradientes [14]. En la práctica, esto significa que el LLM puede reconocer y ejecutar una tarea completamente nueva a partir de unos pocos

ejemplos de demostración (*few-shot*) o incluso sólo mediante una descripción verbal bien formulada de la tarea (*zero-shot*) incluida junto con la consulta del usuario [13], [14]. Para exprimir al máximo estas capacidades, se han desarrollado procesos de *ajuste de instrucciones* (*instruction tuning*), en los que el modelo se entrena con conjuntos de datos formados por pares instrucción–respuesta en lenguaje natural. Este tipo de ajuste refuerza la habilidad del modelo para seguir instrucciones humanas complejas y generalizar hacia tareas que no han aparecido explícitamente durante el entrenamiento [13].

En el ámbito de las interfaces de lenguaje natural para bases de datos (*Natural Language Interfaces for Databases*, NLIDB), todo lo anterior se traduce en una mejora clara: estas capacidades permiten transformar consultas ambiguas en lenguajes ejecutables de forma mucho más flexible que los enfoques simbólicos clásicos [15]. El uso de LLMs en tareas de *traducción semántica* (por ejemplo, de lenguaje natural a SQL o a estructuras de filtros) ayuda a capturar mejor la variabilidad del lenguaje del usuario y a generar representaciones formales precisas incluso en dominios especializados [15]. Así, el sistema puede interpretar la intención subyacente de la consulta y mapearla a una sintaxis estructurada sin requerir un reentrenamiento masivo para cada nuevo dominio, apoyándose en estrategias de *few-shot prompting*, *instruction tuning* y aprendizaje en contexto para adaptar el comportamiento del modelo a las necesidades concretas del problema [13], [14], [15].

2.3. Arquitecturas de Agentes de IA

Hasta hace poco, la tentación natural al trabajar con modelos de lenguaje era enviarles un *prompt* gigantesco y cruzar los dedos esperando que resolvieran todo el problema. Sin embargo, la experiencia nos ha enseñado que las arquitecturas monolíticas se saturan rápido frente a la complejidad. Por eso, en los últimos años, el diseño de arquitecturas multiagente ha ganado muchísimo terreno. La idea es: en lugar de un único "cerebro" abrumado, creamos un equipo de especialistas donde cada uno asume una tarea muy concreta (buscar datos, desambiguar, revisar y generar). Un ejemplo muy bueno de esto es KDR-Agent, un marco para extraer entidades en dominios difíciles que pone a trabajar juntos a agentes de recuperación y de análisis reflexivo, logrando resultados mucho más robustos que si se lo pidiéramos todo a un solo modelo [16].

Y es que este enfoque de sistema multi-paso brilla especialmente cuando necesitamos que la máquina de algo más complejo que una simple charla, como un informe estructurado, una tabla o, en nuestro caso, una consulta de filtros. Arnoldsson y Karimi, por ejemplo, montaron una especie de "redacción virtual" con agentes que recopilan datos, escriben borradores y se editan entre sí [17]. De forma muy parecida, el proyecto TabAgent organiza la extracción de tablas poniendo a un agente a detectar, a otro a entender la semántica y a un tercero a consolidar la información [18]. Al final, lo que nos dicen estos trabajos es sentido común: para resolver un problema difícil, primero hay que entender la intención, luego estructurarla y, solo al final, preocuparse por el formato técnico de salida.

Un aspecto clave de estas arquitecturas es la consistencia y validación de las respuestas generadas. TabAgent incorpora mecanismos explícitos de validación estructural y corrección de errores entre agentes, reduciendo el impacto de alucinaciones y garantizando que las tablas resultantes cumplan restricciones de formato [18]. En el contexto de sistemas de recuperación aumentada, MAIN-RAG introduce un conjunto de agentes que colaboran para filtrar y reordenar documentos antes de la generación, utilizando agentes de predicción y jueces que supervisan la calidad de la evidencia [19]. De forma similar, KARMA aplica un enfoque multiagente al enriquecimiento de grafos de conocimiento, combinando agentes de descubrimiento de entidades, extracción de relaciones y resolución de conflictos para mantener la coherencia del grafo actualizado [20]. Estos ejemplos inspiran el diseño de este trabajo, en el que un conjunto de agentes (para extracción de entidades, mapeo de filtros y validación) trabaja de forma orquestada para traducir consultas en lenguaje natural a filtros estructurados y semánticamente consistentes sobre la taxonomía de materiales.

2.3.1. Técnicas Avanzadas de Interacción: Schema Linking, CoT y Reflexion

A medida que las tareas delegadas a los LLMs se vuelven más algorítmicas y menos conversacionales, la literatura académica ha formalizado una serie de patrones de *prompting* destinados a mitigar alucinaciones y mejorar el determinismo en la generación de estructuras de datos [21]. En el contexto de interfaces NLIDB y sistemas Text-to-SQL, destacan tres metodologías fundamentales:

Schema Linking y Task Decomposition. La verdad es que el mayor salto al vacío para un LLM es intentar adivinar qué palabras del usuario corresponden a qué columnas de nuestra base de datos. Para evitar que la máquina vaya a ciegas, la técnica de *Schema Linking* propone inyectar justo el trocito de esquema que necesita directamente en su contexto, acotando su mundo a lo que de verdad existe en el catálogo [15]. Si a esto se le suma la *Task Decomposition* (obligarle a resolver el puzzle en operaciones atómicas paso a paso), la precisión se dispara en comparación con pedirle el resultado final de golpe.

Chain-of-Thought (CoT) Prompting. Presentado por Wei et al. [22], el patrón de "cadena de pensamiento" (CoT) es un hito en la investigación de LLMs. Esta técnica fuerza al modelo a verbalizar explícitamente su razonamiento intermedio antes de emitir una respuesta final. En tareas de extracción de datos, exigir al modelo que justifique por qué ha seleccionado una determinada entidad o valor, reduce los errores lógicos y mejora la adherencia a las restricciones del formato solicitado.

Reflexion: Autocorrección mediante refuerzo verbal. Incluso con instrucciones robustas, los modelos probabilísticos pueden generar salidas sintácticamente inválidas. Para arreglar este problema, Shinn et al. [23] proponen *Reflexion*, una arquitectura donde el agente (o un agente validador externo) evalúa la respuesta generada frente a unas reglas estrictas. Si se detecta un error, el sistema no falla silenciosamente, sino que genera una retroalimentación verbal explícita (por ejemplo, "La clave proporcionada no existe en el

esquema"). Esta crítica se reinyecta en el contexto del agente generador en una nueva iteración, dotando al sistema de una capacidad de autocuración (*self-healing*) iterativa que resulta muy útil en entornos de producción.

2.4. Extracción de Información Específica de Dominio

2.4.1. Reconocimiento de Entidades Nombradas (NER) Técnico

El Reconocimiento de Entidades Nombradas (NER) aborda la tarea, en apariencia sencilla pero en realidad bastante delicada, de localizar segmentos de texto que hacen referencia a objetos del mundo real (entidades) y asignarles una categoría predefinida [24]. En dominios generales estas categorías suelen ser personas, organizaciones o lugares; sin embargo, cuando nos movemos a contextos técnicos empiezan a aparecer tipos de producto, parámetros físicos, unidades o normas, y de pronto la etiqueta deja de ser tan evidente [24]. Esto complica la definición de esquemas de anotación y dispara la variedad léxica. En entornos como el comercio electrónico, por ejemplo, las consultas suelen mezclar marcas, modelos, medidas y usos ("panel acústico 40mm negro rockwool"), lo que genera cadenas ruidosas y ambiguas que exigen modelos capaces de manejar vocabularios extensos con muy pocos datos etiquetados [25].

Frente a los enfoques clásicos supervisados, que dependen en gran medida de corpus anotados a mano, los marcos basados en LLMs permiten abordar NER en escenarios de bajo recurso apoyándose en conocimiento externo y estrategias de aprendizaje en contexto. Un ejemplo especialmente ilustrativo es KDR-Agent, que descompone la tarea de NER multidominio en varios agentes especializados (recuperación de conocimiento, desambiguación y análisis reflexivo) para mejorar la calidad de las etiquetas en dominios con poca anotación disponible [16]. En este trabajo, el *Entity Agent* se concibe precisamente como un NER técnico adaptado al dominio de materiales de construcción: su cometido es extraer, a partir de una consulta en lenguaje natural, atributos relevantes (grosor, color, uso previsto, certificaciones) y sus valores numéricos o categóricos, dejando la información lista para el siguiente paso del sistema, donde se convertirá en filtros estructurados.

2.4.2. Mapeo Semántico a Taxonomías y Facetas

Quedarse únicamente en la extracción de entidades se queda corto si queremos ejecutar búsquedas reales sobre un catálogo. El verdadero reto consiste en vincular esas palabras sueltas con claves y valores que el sistema entienda. En este punto, las ontologías y taxonomías brillan al actuar como mapas conceptuales que organizan jerarquías, sinónimos y restricciones de forma explícita [6]. Si bien la literatura reciente sobre *ontology embeddings* demuestra que es posible incrustar estos conceptos en espacios vectoriales para facilitar el alineamiento con el lenguaje natural [5], en nuestro proyecto hemos apostado por un enfoque mucho más directo y determinista.

Para lograrlo, la taxonomía de Stylib se ha modelado como una gran estructura de árbol. En su primer nivel residen las categorías generales de materiales sobre las que puede versar la consulta (como suelos de madera o cerámica). Esta decisión de diseño es clave, ya que nos permite acotar el terreno y forzar a la capa de extracción (el *Entity Agent*) a que devuelva únicamente los atributos que pertenecen a la categoría previamente seleccionada.

Una vez canalizada la información, el módulo de mapeo toma el relevo utilizando esta taxonomía como vocabulario controlado. Se encarga de recoger las entidades detectadas, normalizarlas para lidiar con unidades o variaciones léxicas, y traducirlas a los pares clave–valor exactos del catálogo. Al inyectar todo este árbol de conocimiento directamente en el contexto del LLM, conseguimos anclar el modelo a la realidad. En lugar de dejar espacio a las alucinaciones, garantizamos que las condiciones lógicas fluyan de manera segura hasta convertirse en filtros formales, conectando así el NER técnico con la lógica estricta de la base de datos [5], [6].

2.5. Herramientas y Tecnologías de Implementación

El sistema propuesto se apoya en un ecosistema tecnológico diseñado para facilitar la integración de modelos de IA, la validación estricta de datos y la exposición del servicio mediante APIs web. El núcleo de la implementación está desarrollado en *Python*, un lenguaje de propósito general ampliamente utilizado en ciencia de datos y aprendizaje automático, cuyo ecosistema de librerías y documentación oficial lo convierten en una opción natural para proyectos de PLN [26]. Para exponer la lógica del sistema como un servicio REST se emplea *FastAPI*, un *framework* web moderno orientado a alto rendimiento que se basa en las anotaciones de tipos de Python para generar automáticamente documentación interactiva y realizar validación de datos en las peticiones y respuestas [27]. Esta combinación permite construir una API tipada y fácilmente integrable con aplicaciones frontend o servicios externos.

La validación estructurada de la información intercambiada entre agentes y con el exterior se articula mediante *Pydantic*, una librería de validación y gestión de datos que utiliza modelos definidos con anotaciones de tipos para parsear y verificar la entrada, garantizando que los objetos manejados por la lógica de negocio cumplen un esquema bien definido [28]. Sobre esta base se emplea *Pydantic AI*, un *framework* de agentes GenAI desarrollado por el propio equipo de Pydantic, orientado a construir agentes de LLM con salidas tipadas, validación automática y mecanismos de reintento cuando la respuesta del modelo no se ajusta al esquema especificado [29]. Esto encaja directamente con la necesidad de que los agentes del sistema (Entity, Filter y Validator) produzcan estructuras de datos coherentes con la taxonomía de Stylib, reduciendo el impacto de alucinaciones y errores de formato.

Para mantener el sistema agnóstico al proveedor de modelos, se utiliza *OpenRouter*

como pasarela unificada hacia múltiples LLMs comerciales y de código abierto. OpenRouter expone una API compatible con el formato de *chat completions* y permite seleccionar modelos de distintos proveedores mediante un mismo punto de entrada, habilitando cambios de modelo o experimentos de *model sweep* sin modificar la lógica de los agentes [30]. Finalmente, la interfaz de usuario se implementa en *React*, una librería de JavaScript orientada a la construcción de interfaces declarativas basadas en componentes reutilizables, lo que facilita desarrollar una experiencia de búsqueda interactiva que consuma la API de FastAPI y visualice en tiempo real los filtros generados por el sistema [31]. En conjunto, estas herramientas proporcionan una base tecnológica coherente con los requisitos de rendimiento, mantenibilidad y extensibilidad del proyecto.

2.6. Trabajos Relacionados y Estado del Arte

La línea de trabajo más cercana al sistema propuesto es la de las interfaces de lenguaje natural a bases de datos (*Natural Language Interfaces for Databases*, NLIDB), cuyo objetivo es traducir consultas en lenguaje natural a lenguajes formales ejecutables como SQL. La verdad es que el problema consiste en "enseñar" a la máquina a interpretar una intención y convertirla en una consulta técnica precisa. La revisión sistemática de Liu et al. (2025) analiza el ciclo de vida completo de estas tareas, desde las arquitecturas clásicas basadas en reglas hasta los sistemas impulsados por LLMs, resaltando el uso de una representación intermedia (IR) como puente crítico para simplificar el razonamiento y guiar la generación final de la consulta [15]. En paralelo, otros trabajos sobre Text-to-SQL estudian cómo estos modelos han mejorado la generación de consultas complejas mediante técnicas de *prompting* y *fine-tuning* supervisado [15]. Modelos como RAT-SQL, que introduce una codificación relacional explícita del esquema, o PICARD, que impone restricciones sintácticas durante la decodificación, muestran cómo el control estructural permite producir resultados más válidos en benchmarks exigentes [32], [33].

Otra línea especialmente relevante es la de los sistemas de generación aumentada por recuperación (*Retrieval-Augmented Generation*, RAG), que combinan un módulo de recuperación de documentos con un modelo generativo para obtener respuestas mejor fundamentadas. Estos sistemas buscan mitigar limitaciones intrínsecas de los LLMs, como las alucinaciones y la obsolescencia del conocimiento interno, apoyando la generación en evidencias actualizadas de una base de conocimiento externa [34]. Los surveys recientes sobre RAG ofrecen una taxonomía detallada que clasifica estas arquitecturas según sus componentes de recuperación, generación e integración (a nivel de entrada, salida o capas intermedias), analizando además estrategias de entrenamiento y la robustez del sistema frente a información poco fiable [34]. En el contexto de motores de búsqueda aplicados a catálogos, los trabajos sobre *hybrid search* muestran cómo la combinación de técnicas de recuperación tradicionales (por ejemplo, BM25) con búsquedas semánticas basadas en embeddings puede mejorar de forma notable la pertinencia de los resultados, especialmente en dominios técnicos donde coexisten descripciones textuales y metadatos estruc-

turados [11]. Estas aproximaciones RAG e híbridas resultan especialmente interesantes como posibles extensiones futuras de este proyecto, ya que permitirían complementar los filtros estructurados con la recuperación de fichas de producto o documentación técnica en lenguaje natural.

2.6.1. Aportación Original del Proyecto

Frente a estos trabajos, el sistema desarrollado en este TFG se centra en un problema muy concreto de una empresa real: la plataforma Stylib, que mantiene un catálogo de materiales de construcción con una taxonomía compleja y un motor de filtrado muy expresivo. A diferencia de los enfoques clásicos de Text-to-SQL y de las NLIDB analizadas en la literatura actual, cuyo objetivo es generar consultas SQL genéricas sobre bases de datos relacionales [15], la meta aquí es transformar consultas en lenguaje natural en *filtros estructurados y URLs parametrizadas*. Estas deben respetar estrictamente la taxonomía, los operadores lógicos y las convenciones internas del sistema de Stylib, lo que requiere un mapeo preciso entre el lenguaje libre del usuario y la sintaxis formal del catálogo [15]. De este modo, el proyecto busca realizar una consulta y además "hablar el idioma exacto" de una base de conocimiento industrial específica, aplicando técnicas de diseño modular descritas en los surveys de referencia pero adaptadas a un dominio técnico restringido.

Además, en lugar de delegar toda la responsabilidad en un único modelo monolítico, se adopta una arquitectura multiagente que separa explícitamente la extracción de entidades de dominio, el mapeo a claves y valores de la taxonomía y la validación semántica, integrando los principios de NER técnico y de ontologías/taxonomías descritos en las secciones anteriores [5], [6], [16]. Esta descomposición permite controlar mejor cada paso del proceso y, sobre todo, detectar y corregir errores de forma localizada. En conjunto, el proyecto aporta una aplicación muy concreta de estas líneas de investigación al contexto industrial de Stylib, mostrando cómo combinar LLMs, conocimiento estructurado y diseño de agentes puede reducir de forma tangible la fricción a la hora de utilizar un catálogo real de materiales de construcción.

3. ANÁLISIS DEL SISTEMA

3.1. Descubrimiento del Problema

3.1.1. Complejidad de la interfaz actual

La plataforma de Stylib cuenta con un motor de filtrado muy capaz, que combina decenas de atributos técnicos mediante operadores lógicos, rangos numéricos y varias facetas a la vez. Para un usuario experto eso es una ventaja: puede expresar búsquedas muy precisas. Pero en el uso diario la cosa se complica. Muchos usuarios se topan con una interfaz densa, llena de menús desplegables, etiquetas especializadas y parámetros que, si no se conocen de antemano, generan confusión y cierta sensación de "estar tocando algo que no se debería".

Durante el análisis preliminar se identificaron varios puntos de fricción. Por ejemplo, para responder a una necesidad aparentemente simple como "paneles acústicos negros de menos de 20 mm de grosor", el usuario debe conocer en qué categoría exacta se encuentran esos paneles, qué faceta controla el grosor, qué unidades utiliza el sistema y cómo combinar correctamente varios filtros con operadores lógicos. Además, la interfaz obliga a trasladar mentalmente la intención (hablar de "paneles acústicos negros y finos") a la estructura interna del catálogo, lo que provoca errores, iteraciones repetitivas y, en muchos casos, abandono antes de llegar a un resultado satisfactorio. La experiencia se vuelve especialmente compleja para perfiles no técnicos o para usuarios que consultan la plataforma de forma esporádica.

3.1.2. Necesidad de una búsqueda más accesible

A partir de esta observación, surge una necesidad clara: ofrecer una forma de búsqueda más accesible, que se acerque al lenguaje natural con el que los usuarios ya piensan sus problemas. La idea no es sustituir el sistema de filtrado existente (que sigue siendo esencial para ejecutar las consultas sobre el catálogo), sino colocar delante una capa intermedia que "traduzca" la intención del usuario a filtros técnicos sin obligarle a conocer la taxonomía de Stylib al detalle.

En la práctica, esto significa permitir que un arquitecto, un interiorista o incluso un cliente final puedan escribir consultas como "paneles fonoabsorbentes negros para un techo de oficina, que no sean muy gruesos" y que el sistema se encargue de interpretar esa petición, extraer los atributos relevantes y configurar los filtros adecuados. De esta manera, la interfaz deja de ser un muro de opciones técnicas y pasa a comportarse como un asistente que entiende la intención del usuario y la adapta al lenguaje del catálogo. La motivación de este trabajo nace precisamente de allí: reducir la distancia entre cómo el

usuario formula su necesidad y cómo el sistema espera recibirla para poder trabajar con ella.

3.2. Definición de la Solución Propuesta

3.2.1. Objetivo funcional del sistema

Partiendo del problema expuesto, el objetivo funcional del sistema puede resumirse de la siguiente manera: desarrollar y evaluar un componente intermedio que transforme consultas en lenguaje natural en filtros estructurados y semánticamente correctos para el catálogo de Stylib. Este componente debe actuar como un "intérprete" entre el usuario y el sistema, de forma que el primero pueda expresarse de manera libre y el segundo siga recibiendo los filtros que necesita para operar.

Para lograrlo, se propone una arquitectura basada en modelos de lenguaje de gran escala orquestados en un *pipeline* multiagente. Cada agente asume una responsabilidad concreta: uno se centra en extraer entidades y atributos de la consulta, otro en mapearlos a claves y valores válidos de la taxonomía de Stylib y un tercero en validar la coherencia del resultado antes de devolverlo al frontend. El sistema debe integrarse con la infraestructura existente de Stylib, de modo que, desde el punto de vista del usuario, la experiencia se reduzca a introducir una frase en un cuadro de búsqueda y ver cómo los resultados del catálogo se muestran con los filtros ya configurados.

3.2.2. Restricciones y supuestos

La solución propuesta se diseña bajo una serie de restricciones y supuestos que acotan su alcance y condicionan las decisiones técnicas. En primer lugar, el sistema se orienta específicamente al dominio de materiales de construcción de Stylib y a la taxonomía disponible en el momento del desarrollo. Se asume que dicha taxonomía, junto con sus facetas y valores, puede exponerse al sistema en forma de base de conocimiento estructurada y actualizarse de manera controlada. No se pretende cubrir, al menos en esta primera versión, otros dominios ni catálogos externos.

En segundo lugar, se considera que las consultas de los usuarios estarán centradas en la búsqueda de productos dentro del catálogo, y no en tareas más abiertas como preguntas generales sobre normativas o recomendaciones de diseño. El sistema se limita, por tanto, a producir filtros y parámetros de búsqueda, sin generar explicaciones largas ni textos comerciales. Además, se presupone que el servicio funcionará como una API interna, por lo que deberá cumplir requisitos razonables de latencia y coste por llamada, compatibles con el uso en un entorno web interactivo.

Por último, se asume que la calidad de las salidas del sistema puede evaluarse tanto de forma cuantitativa (comparando los filtros generados con un *ground truth* predefinido)

como cualitativa, a través de juicios de expertos sobre la utilidad de los resultados. Bajo estas condiciones, el proyecto se plantea como un primer paso hacia una interfaz de búsqueda más natural en Stylib, pero también como un laboratorio controlado para explorar el uso de arquitecturas multiagente basadas en LLMs en un caso de uso real de empresa.

3.3. Propuesta de Solución

3.3.1. Enfoque multiagente

Ante la complejidad del problema descrito, la solución adoptada se apoya en una arquitectura multiagente en la que distintos módulos especializados colaboran de forma secuencial para transformar una consulta en lenguaje natural en un conjunto de filtros estructurados válidos para el catálogo de Stylib. La intuición detrás de este enfoque es sencilla: en lugar de pedirle a un único modelo que "lo haga todo a la vez", es decir, extraer entidades, mapearlas a la taxonomía y validar el resultado en un solo paso, se descompone el problema en tareas más acotadas, cada una con su propio contexto y su propia lógica de verificación [16].

El sistema está formado por tres agentes principales:

- Entity Agent: recibe la consulta original del usuario y la analiza para extraer los atributos relevantes del dominio (materiales, colores, dimensiones, usos, certificaciones, etc.) junto con sus valores. Su salida es una lista estructurada de pares atributo–valor en lenguaje natural.
- Filter Agent: toma esa lista de entidades y la mapea a las claves y valores exactos de la taxonomía de Stylib, produciendo una representación formal de los filtros que puede serializarse directamente como parámetros de la URL de búsqueda. Para ello, dispone de acceso a la taxonomía del catálogo, un contexto de apoyo con las claves y valores válidos.
- Validator Agent: verifica la coherencia semántica y estructural de los filtros generados, detectando posibles combinaciones imposibles, valores fuera de rango o claves no existentes, y devuelve al sistema una versión corregida o una señal de error con retroalimentación.

Este diseño ofrece varias ventajas prácticas. La descomposición en agentes facilita el diagnóstico y la corrección de errores de forma localizada: si el Filter Agent produce un valor incorrecto, no es necesario depurar toda la cadena, sino sólo ese componente.

3.3.2. Flujo general del sistema

El flujo de ejecución del sistema sigue una secuencia bien definida, ilustrada en la Figura 3.1. Cuando el usuario introduce una consulta en lenguaje natural desde el *frontend*

de Stylib, esta se envía al *backend* a través de una petición a la API. En ese momento, el sistema multiagente se activa. En primer lugar, el *Entity Agent* analiza la consulta apoyándose en la taxonomía, que actúa como contexto para identificar la categoría del producto y extraer las condiciones del dominio. Después, el *Filter Agent* traduce esas condiciones a las claves canónicas y operadores internos utilizados por Stylib. Finalmente, el *Validator Agent* revisa el resultado y comprueba si la representación generada mantiene coherencia semántica y estructural con la intención original del usuario.

Uno de los aspectos más importantes del sistema aparece cuando la validación falla. En lugar de devolver directamente un error o aceptar una salida defectuosa, se activa un bucle de reintento. El sistema analiza primero el origen probable del fallo y, a partir de ahí, decide a qué agente conviene retroceder. Si el problema se asocia al *Filter Agent* (por ejemplo, una clave canónica mal elegida, un operador incorrecto o una construcción inconsistente del filtro), solo se reinvoca dicho agente, proporcionándole retroalimentación específica sobre los errores detectados. En cambio, si el fallo parece originarse en el *Entity Agent* (como una categoría mal inferida o la omisión de una condición importante), se relanza la cadena completa desde esa etapa.

Por ende, esta estrategia reduce reejecuciones innecesarias y hace que el comportamiento global resulte más robusto y más eficiente. Una vez que el *Validator Agent* aprueba la salida, el *backend* construye la URL de búsqueda parametrizada y la devuelve al *frontend*. Desde allí, el catálogo de Stylib lanza la consulta final y muestra los resultados al usuario.

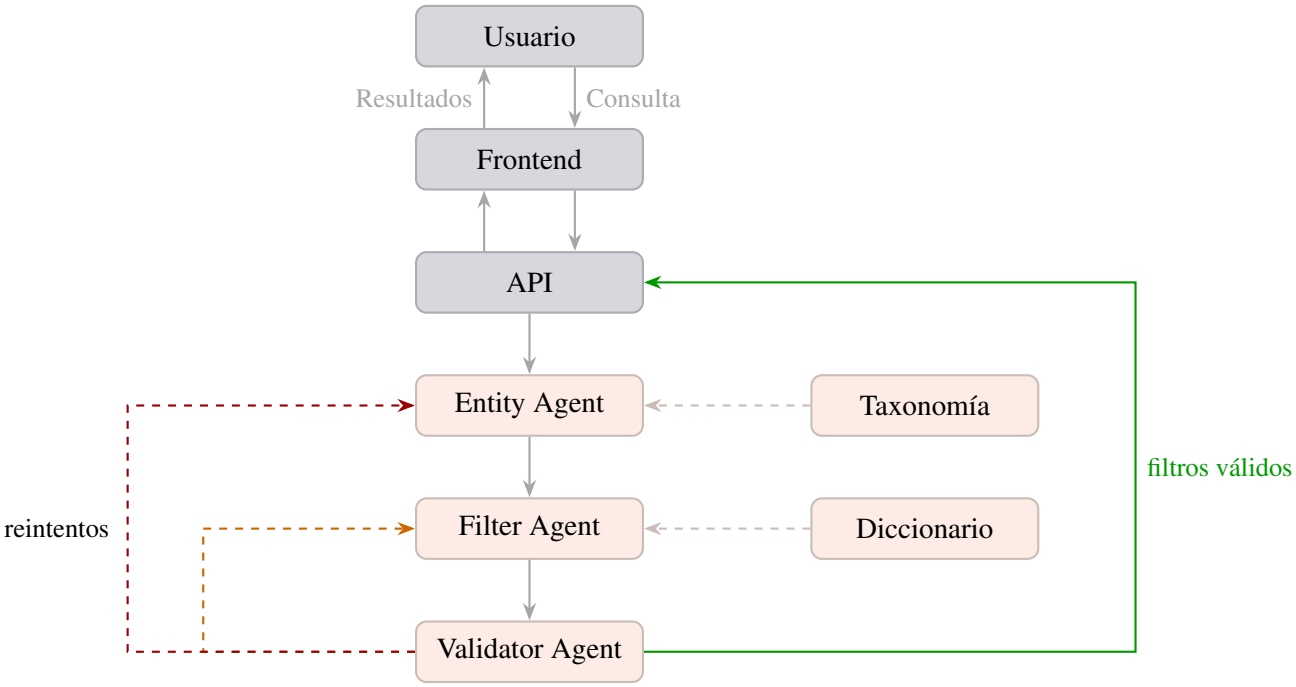


Fig. 3.1. Flujo general del sistema multiagente.

3.4. Especificación de Requisitos

En esta sección se recogen los compromisos que el sistema asume para resolver el problema de filtrado en Stylib. Para ello, se han dividido las necesidades en dos grandes bloques: los requisitos funcionales, es decir, las acciones concretas que el sistema debe poder ejecutar, y los requisitos no funcionales, que definen la calidad, el rendimiento y las restricciones tecnológicas de esas acciones.

A continuación, se detalla la especificación estructurada, utilizando un formato de tabla que permite catalogar cada requisito por su identificador (ID), nombre, descripción, nivel de prioridad (Alta, Media, Baja) y tipo (Funcional o No Funcional).

3.4.1. Requisitos funcionales

Los requisitos funcionales detallan el comportamiento del sistema desde el momento en que el usuario escribe su consulta hasta que recibe los filtros listos para usar en el catálogo.

Campo	**Descripción**
Identificador	RF-01
Nombre	Procesamiento de consultas en lenguaje natural
Descripción	El sistema debe proporcionar un punto de entrada (API) que reciba texto libre ingresado por el usuario, sin requerir una sintaxis o formato estricto, para iniciar el flujo de búsqueda.
Prioridad	Alta
Tipo	RF

TABLA 3.1. RF-01: PROCESAMIENTO DE CONSULTAS

Campo	**Descripción**
Identificador	RF-02
Nombre	Extracción de entidades de dominio
Descripción	A través del *Entity Agent*, el sistema debe ser capaz de identificar materiales, propiedades físicas (grosor, medidas), colores, usos previstos y certificaciones presentes en la consulta del usuario.
Prioridad	Alta
Tipo	RF

TABLA 3.2. RF-02: EXTRACCIÓN DE ENTIDADES DE DOMINIO

Campo	Descripción
Identificador	RF-03
Nombre	Mapeo a la taxonomía de Stylib
Descripción	El *Filter Agent* debe traducir las entidades extraídas a las claves y valores exactos del diccionario de datos (taxonomía) que utiliza internamente el catálogo de Stylib.
Prioridad	Alta
Tipo	RF

TABLA 3.3. RF-03: MAPEO A LA TAXONOMÍA DE STYLIB

Campo	Descripción
Identificador	RF-04
Nombre	Validación semántica y corrección
Descripción	El *Validator Agent* debe comprobar que los filtros generados tienen sentido (por ejemplo, que no haya valores numéricos en campos de texto) y solicitar una corrección automática en caso de error.
Prioridad	Media
Tipo	RF

TABLA 3.4. RF-04: VALIDACIÓN SEMÁNTICA Y CORRECCIÓN

Campo	Descripción
Identificador	RF-05
Nombre	Generación de URL de catálogo
Descripción	Una vez validados, el sistema debe empaquetar los filtros resultantes en una estructura JSON o en una cadena URL parametrizada que el frontend de Stylib pueda consumir directamente.
Prioridad	Alta
Tipo	RF

TABLA 3.5. RF-05: GENERACIÓN DE URL DE CATÁLOGO

3.4.2. Requisitos no funcionales

Más allá de lo que el sistema hace, importa cómo lo hace. Los requisitos no funcionales establecen las normas de juego en cuanto a arquitectura, tiempos de respuesta y mantenibilidad, garantizando que el paso de la teoría a la práctica industrial tenga sentido.

Campo	**Descripción**
Identificador	RNF-01
Nombre	Tiempo de respuesta interactivo
Descripción	El proceso completo (desde que el usuario envía la consulta hasta que se devuelven los filtros) debe ejecutarse en un tiempo razonable para una experiencia web fluida, idealmente por debajo de los 3-5 segundos.
Prioridad	Alta
Tipo	RNF

TABLA 3.6. RNF-01: TIEMPO DE RESPUESTA INTERACTIVO

Campo	**Descripción**
Identificador	RNF-02
Nombre	Agnosticismo de modelo (LLM)
Descripción	La arquitectura debe permitir el cambio de proveedor de inteligencia artificial (OpenAI, Anthropic) sin reescribir la lógica de negocio, apoyándose en pasarelas como OpenRouter.
Prioridad	Alta
Tipo	RNF

TABLA 3.7. RNF-02: AGNOSTICISMO DE MODELO (LLM)

Campo	Descripción
Identificador	RNF-03
Nombre	Tipado fuerte y validación de esquemas
Descripción	Las estructuras de datos internas y las respuestas de la API deben estar fuertemente tipadas utilizando herramientas como Pydantic, para evitar fallos de ejecución en tiempo real.
Prioridad	Alta
Tipo	RNF

TABLA 3.8. RNF-03: TIPADO FUERTE Y VALIDACIÓN DE ESQUEMAS

Campo	Descripción
Identificador	RNF-04
Nombre	Actualización de la taxonomía
Descripción	El sistema debe diseñarse de forma modular para que la taxonomía base (el vocabulario de materiales y atributos) pueda actualizarse sin necesidad de modificar el código fuente de los agentes.
Prioridad	Media
Tipo	RNF

TABLA 3.9. RNF-04: ACTUALIZACIÓN DE LA TAXONOMÍA

3.5. Casos de Uso

Para que un sistema pase de ser una buena idea técnica a convertirse en un producto real, primero hay que entender cómo interactúa con las personas y con los demás sistemas que lo rodean. Los casos de uso que se detallan a continuación recogen precisamente esa visión: describen qué quiere hacer el usuario, qué papel juega la infraestructura de Stylib y cómo el sistema multiagente se encarga de la lógica compleja por debajo.

En la Figura 3.2 se muestra el diagrama UML que resume las interacciones principales. En él aparecen tres actores externos: el Usuario, el *Frontend/Backend de Stylib* y el *Servicio LLM externo*, mientras que el *Sistema Multiagente* se modela como el rectángulo central que engloba los cuatro casos de uso y orquesta el flujo secuencial desde la introducción de la consulta hasta la ejecución de la búsqueda en el catálogo.

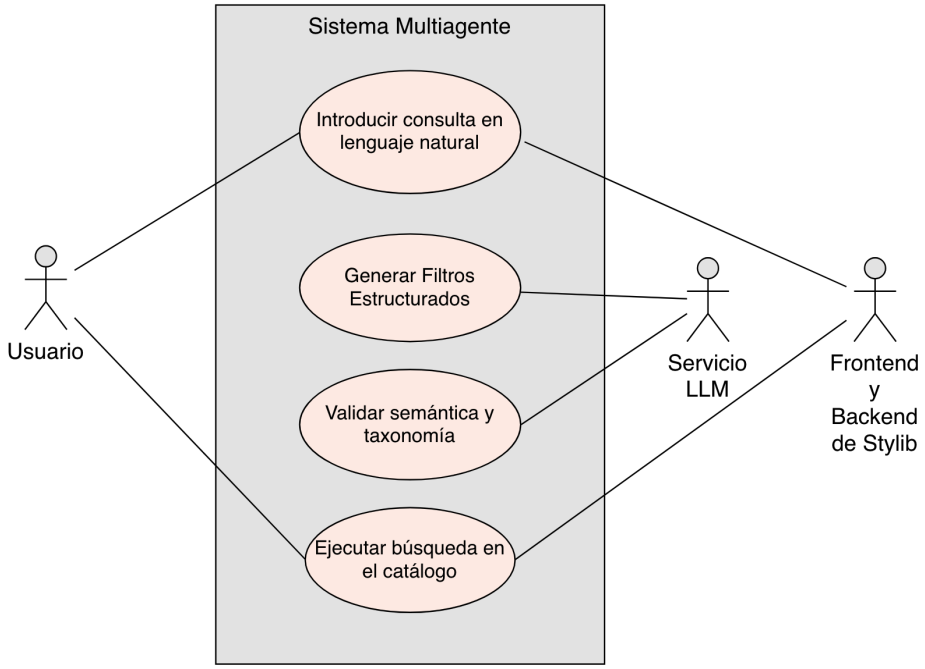


Fig. 3.2. Diagrama de casos de uso del sistema multiagente

Campo	Descripción
ID	CU-01
Nombre	Introducir consulta en lenguaje natural
Actores	Usuario
Descripción	El usuario utiliza un cuadro de texto en la interfaz para escribir, con sus propias palabras, las características de los materiales que está buscando (ej. “paneles acústicos de madera oscura para techo”).
Precondiciones	El sistema de backend y la interfaz web deben estar activos y accesibles.
Postcondiciones	La consulta es recibida por la API y se inicia el flujo de procesamiento.
Req. funcionales	RF-01

TABLA 3.10. CU-01: INTRODUCIR CONSULTA EN LENGUAJE NATURAL

Campo	Descripción
ID	CU-02
Nombre	Generar filtros estructurados
Actores	Servicio LLM externo
Descripción	De forma opaca para el usuario, los agentes de extracción y filtrado analizan el texto para identificar atributos y los traducen a las claves y valores exactos del diccionario interno de Stylib.
Precondiciones	La consulta en texto plano (CU-01) debe haber sido recibida correctamente por el backend.
Postcondiciones	Se genera un objeto JSON o diccionario temporal con los filtros propuestos.
Req. funcionales	RF-02, RF-03

TABLA 3.11. CU-02: GENERAR FILTROS ESTRUCTURADOS

Campo	Descripción
ID	CU-03
Nombre	Validar semántica y taxonomía
Actores	Servicio LLM externo
Descripción	El *Validator Agent* comprueba que los filtros generados en el paso anterior son coherentes y existen realmente en la base de datos de Stylib, corrigiendo posibles errores o alucinaciones del modelo.
Precondiciones	Debe existir una propuesta de filtros generada por el CU-02.
Postcondiciones	Los filtros se confirman como válidos y listos para su uso.
Req. funcionales	RF-04

TABLA 3.12. CU-03: VALIDAR SEMÁNTICA Y TAXONOMÍA

Campo	Descripción
ID	CU-04
Nombre	Ejecutar búsqueda en catálogo
Actores	Usuario, Frontend/Backend Stylib
Descripción	El sistema ensambla los filtros validados en una petición formal (por ejemplo, una URL con parámetros) y lanza la consulta contra el catálogo. La interfaz se actualiza mostrando al usuario los materiales encontrados.
Precondiciones	Los filtros deben haber pasado la validación (CU-03).
Postcondiciones	El usuario visualiza en pantalla los resultados filtrados según su petición inicial.
Req. funcionales	RF-05

TABLA 3.13. CU-04: EJECUTAR BÚSQUEDA EN CATÁLOGO

4. DISEÑO DEL SISTEMA

En este capítulo se desciende desde el plano conceptual hacia la arquitectura real del sistema. Si en apartados anteriores se definió qué debía hacer la aplicación para resolver el problema de filtrado en Stylib, aquí se detalla cómo están construidas sus piezas.

4.1. Arquitectura General

Diseñar un sistema que traduzca lenguaje natural a filtros estructurados exige algo más que enviar un *prompt* a un modelo de lenguaje y cruzar los dedos esperando que la respuesta tenga el formato correcto. Requiere una arquitectura defensiva, pensada para acotar la incertidumbre inherente a la Inteligencia Artificial. Por ello, la arquitectura general del sistema sigue el patrón Modelo–Vista–Controlador (MVC), separando la interfaz de usuario, la coordinación de peticiones y la lógica de dominio en capas independientes.

4.1.1. Diagrama de Componentes

En la Figura 4.1 se representa la arquitectura lógica del sistema mediante un diagrama de componentes UML 2.0 organizado según el patrón Modelo–Vista–Controlador (MVC). Esta vista pone el foco en los módulos desplegables y en los contratos que existen entre ellos, en lugar de en el detalle de clases o de mensajes individuales, siguiendo las buenas prácticas de modularidad y separación de responsabilidades descritas en la literatura de arquitectura de software.[35]

En la capa de Vista se encuentra la Interfaz Web, un componente React que captura la consulta en lenguaje natural y renderiza el catálogo filtrado. La capa de Controlador está formada por el componente Backend, implementado con FastAPI, que expone la API REST del sistema, valida las peticiones HTTP entrantes y coordina la ejecución del pipeline multiagente. Finalmente, la capa de Modelo agrupa la lógica de dominio y las fuentes de conocimiento: el Componente de Agentes, la Taxonomía, el archivo de Facetas y el Proveedor LLM externo.

Cada bloque de la figura se modela como un *component* UML, con sus dependencias representadas mediante el conector de interfaz (*ball-and-socket*). La Interfaz Web requiere una interfaz REST (`IBackendAPI`) que el Backend proporciona, mientras que el propio Backend requiere del Componente de Agentes una interfaz lógica única (`IAgentPipeline`), encargada de transformar consultas en lenguaje natural en filtros estructurados válidos. De este modo, las decisiones de implementación quedan encapsuladas tras contratos claros, lo que facilita, por ejemplo, sustituir el *framework* de *frontend* sin afectar al resto del sistema.[35]

En el Modelo, el Componente de Agentes encapsula tres subcomponentes: Agente de Entidad, Agente de Filtros y Agente Validador. Para resolver una consulta, este componente requiere a su vez tres servicios externos: la Taxonomía y las Facetas exponen sendas interfaces (`ITaxonomyProvider` e `IFacetsProvider`) con el conocimiento de dominio necesario para interpretar categorías y resolver claves canónicas, mientras que el Proveedor LLM ofrece la interfaz `ILLMProvider`, consumida por los tres agentes para delegar el razonamiento en el modelo de lenguaje subyacente. Aislar este último contrato permite cambiar de proveedor (por ejemplo, de OpenRouter a una API directa) sin tocar la lógica interna de los agentes. Esta organización refuerza la separación entre presentación, lógica de aplicación y modelo de dominio, y reduce el acoplamiento entre módulos.[35]

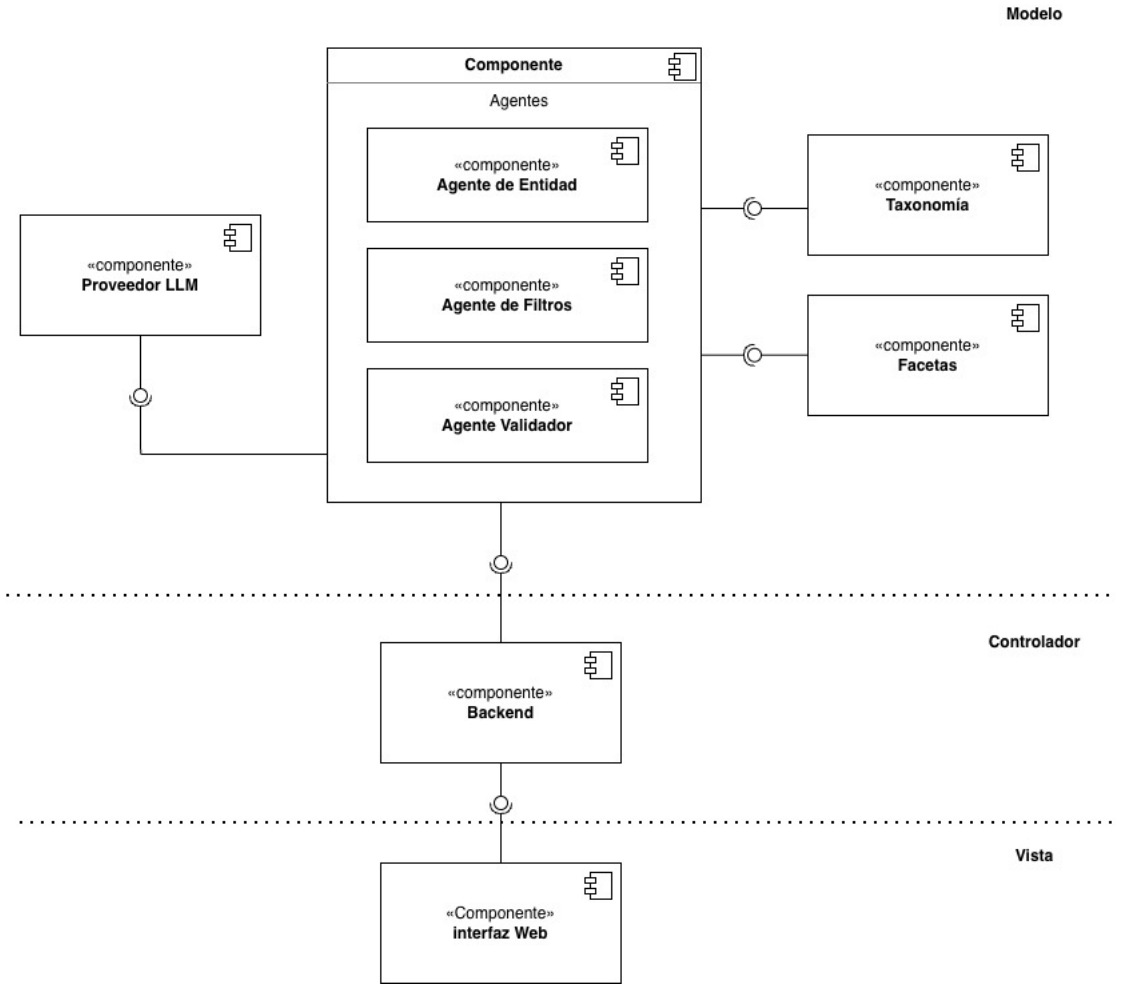


Fig. 4.1. Diagrama de componentes del sistema siguiendo el patrón MVC.

4.1.2. Enfoque de Pipeline Multiagente y Tolerancia a Fallos

En lugar de construir un "super-prompt" que le pida a un único modelo que lea la frase, estructure los operadores, aplique la lógica de negocio y devuelva un JSON perfecto en una sola pasada, el sistema divide y vencerá. Se ha diseñado un *pipeline* multiagente donde cada eslabón de la cadena asume una parte de la carga cognitiva.

El *Entity Agent* actúa como el primer analista. Al iniciar su ejecución, se le inyecta mediante contexto toda la taxonomía (categorías, facetas globales y especificaciones técnicas). Su trabajo es extraer de la consulta en lenguaje natural una serie de *LogicalConditions*. No solo identifica los atributos, sino que resuelve los operadores (ej. rojo o azul"→ `red OR blue`), normaliza los rangos numéricos (ej: entre 15 y 20mm → `15-20`) y los asocia a los conceptos técnicos reales del catálogo (ej. *Wood flooring*).

A continuación, pasa el relevo al *Filter Agent*, cuyo rol es actuar como traductor estricto. Recibe las condiciones lógicas del paso anterior junto con un diccionario de mapeo interno (*mapping dict*) y una lista cerrada de claves válidas. Su objetivo es convertir esas condiciones conceptuales en la estructura de datos final y generar la cadena de filtrado exacta que consumirá la base de datos subyacente. Finalmente, el *Validator Agent* hace de juez: revisa el resultado propuesto cruzándolo con las claves permitidas y la consulta original.

El punto fuerte de este enfoque es su lógica de reintentos y tolerancia a fallos. Cuando el *Validator Agent* intercepta una inconsistencia, no rompe la ejecución ni devuelve un error al usuario: clasifica de dónde viene el fallo y reencamina el reintento en consecuencia. Si el error está en el mapeo, por ejemplo una clave que no existe en el diccionario o que viola las reglas de Stylib, se reejecuta solo el *Filter Agent* con el *feedback* inyectado. Si el fallo es anterior, en la interpretación de la consulta (una categoría equivocada o una condición omitida), el control vuelve al *Entity Agent* para rehacer la extracción. Este bucle *self-healing* (autocurativo) itera hasta lograr un resultado válido o agotar los intentos, y es lo que permite que el *frontend* reciba siempre datos estructurados y seguros.

4.1.3. Ciclo de vida de una petición (End-to-End)

Para comprender cómo colaboran estas piezas, resulta útil seguir el rastro de una petición de principio a fin (*End-to-End*), desde el navegador del usuario hasta el catálogo filtrado.

Todo comienza en la interfaz de *React*, que captura una búsqueda libre como “parquet de roble en espiga con grosor menor a 15mm” y lanza una petición HTTP asíncrona hacia el *backend*. La puerta de entrada es la API de *FastAPI*, que orquesta todo el proceso.

En el primer paso, FastAPI carga en memoria el esquema de catálogo (la taxonomía) y despierta al *Entity Agent*. Este agente, conectándose a un modelo LLM a través de la pasarela *OpenRouter*, analiza la frase y la descompone en entidades lógicas (Categoría: *Wood flooring*; Especie: *oak*; Formato: *herringbone*; Grosor: *<15*).

Con esa base estructurada, la API invoca al *Filter Agent*, proporcionándole las entidades, el diccionario interno y las claves de filtro válidas. El modelo genera un *FilterGroup* y una cadena estructurada. Inmediatamente, la respuesta se somete al escrutinio del *Validator Agent*. Si detecta un problema, rechaza el resultado y, según su origen, relanza el *Filter Agent* (cuando es un fallo de mapeo, como una *invalid key*) o el *Entity Agent* (cuan-

do la consulta se interpretó mal), inyectando el error detectado en un bucle iterativo hasta conseguir luz verde.

Una vez que los filtros superan todas las validaciones semánticas y de esquema de *Pydantic*, la API de *FastAPI* no devuelve el catálogo de productos directamente, sino que empaqueta la respuesta técnica definitiva. El sistema toma la cadena de filtros estructurada (*filter string*) y, utilizando una utilidad interna (*FilterEncoder*), la comprime y codifica en formato Base64. Esta cadena encriptada, junto con metadatos de la categoría, se devuelve a la interfaz de *React*.

En el último tramo, el *frontend* intercepta este objeto JSON y extrae la cadena Base64 para inyectarla directamente en los parámetros de la URL de la aplicación. Al cambiar la URL, la vista del catálogo se recarga automáticamente y es entonces cuando lanza una petición HTTP convencional (fuera del ecosistema multiagente) contra la base de datos real de Stylib. De este modo, no solo se desacopla la inteligencia artificial de la extracción de datos, sino que se consigue que las búsquedas generadas por lenguaje natural mantengan su estado en la URL, permitiendo al usuario guardar, compartir o recargar los resultados de su consulta sin perder el contexto.

4.2. Especificación de Componentes

En esta sección se detallan las tecnologías, lenguajes y librerías que conforman la base de la arquitectura. La selección de este *stack* tecnológico ha buscado en todo momento un equilibrio entre alto rendimiento, facilidad de mantenimiento a largo plazo y, sobre todo, la capacidad de integrar modelos de Inteligencia Artificial de forma tipada y robusta.

4.2.1. Lenguaje y Framework Base

El núcleo del sistema se apoya en un ecosistema muy extendido hoy tanto en Inteligencia Artificial como en el desarrollo backend.

Python (3.13): Como se analizó en el estado del arte, Python es actualmente el lenguaje *de facto* en el ámbito de la Inteligencia Artificial y el Procesamiento de Lenguaje Natural (PLN). Su ecosistema maduro, su tipado dinámico (ahora fuertemente enriquecido con *Type Hints*) y su sintaxis expresiva permiten un desarrollo ágil de lógicas complejas. En este proyecto, Python actúa como el motor principal que orquesta la carga de la taxonomía local, el control de flujo de los agentes y la transformación matemática de los datos (la codificación en Base64).

FastAPI: Para exponer la funcionalidad del sistema multiagente hacia el *frontend*, se ha implementado una API REST utilizando FastAPI. Este *framework* destaca por su rendimiento asíncrono y, lo más importante para este proyecto, por su integración nativa con *Pydantic*. Esto significa que las validaciones de los datos de entrada

(las consultas de usuario) y de salida (los objetos JSON de respuesta) se realizan automáticamente a nivel de *middleware*. Además, genera documentación interactiva (Swagger UI), lo que facilita la integración y las pruebas de los *endpoints*.

4.2.2. Orquestación Agéntica y Tipado

El mayor reto al trabajar con Modelos de Lenguaje (LLMs) es su naturaleza probabilística: lo que devuelven es texto, no una estructura de datos predecible. Para controlar esa incertidumbre y forzar al modelo a devolver estructuras de datos predecibles, el sistema se apoya en dos librerías.

Pydantic: En un entorno donde los agentes de IA deben devolver *arrays*, claves de catálogo, operadores lógicos y valores numéricos estrictos, el formato no puede dejarse al azar. *Pydantic* es una librería de validación de datos para Python que utiliza las anotaciones de tipos nativas (*Type Hints*). En este trabajo se utiliza para definir los "Contratos de Datos" (esquemas como `Filter`, `FilterGroup` o `ValidationResult`): funciona como un embudo que intercepta al instante cualquier alucinación estructural del modelo y la convierte en un error de validación estándar.

PydanticAI: Es un *framework* de orquestación de agentes pensado para construir flujos de trabajo con LLMs sin perder el control sobre el formato de salida. A diferencia de otros ecosistemas más genéricos o basados en manipulación de cadenas de texto (como LangChain), *PydanticAI* usa los esquemas de *Pydantic* como base. Se ha elegido para programar el *Entity Agent*, el *Filter Agent* y el *Validator Agent*, ya que permite inyectar dependencias dinámicas (como la taxonomía de Stylib) directamente en el contexto de ejecución (*RunContext*) y forzar las salidas estructuradas.

4.2.3. Abstracción de Modelos LLM

Depender exclusivamente de la API directa de un único proveedor corporativo (como OpenAI o Anthropic) limitaría mucho la capacidad de experimentación y la viabilidad económica del proyecto a futuro.

OpenRouter: Para solucionar la dependencia del proveedor (*vendor lock-in*), se utiliza OpenRouter. Esta es una interfaz unificada (API) que proporciona acceso a multitud de LLMs del mercado a través de un único *endpoint* estandarizado. Al delegar las llamadas a OpenRouter, el código interno de los agentes se vuelve completamente agnóstico al modelo subyacente.

La justificación principal de esta decisión arquitectónica radica en los experimentos (benchmark) planificados para fases posteriores: con un simple cambio en una constante de configuración (por ejemplo, pasando de `google/gemini-3.1-flash`

a `anthropic/claude-3-haiku`), es posible evaluar la velocidad, coste y precisión de diferentes arquitecturas neuronales sin tener que reescribir ni una sola línea de la lógica de reintentos o conexión.

4.2.4. Interfaz de Usuario

Para la implementación del sistema cliente (la pieza de *software* con la que interactúa el usuario final y que renderiza el catálogo) se ha recurrido al ecosistema *JavaScript/TypeScript*.

React: React permite construir una experiencia de usuario (UX) asíncrona y reactiva, algo imprescindible en un buscador moderno. En lugar de recargar la página entera tras cada consulta, React mantiene el campo de búsqueda en lenguaje natural y modifica el estado de la aplicación inyectando la cadena Base64 (devuelta por la API) en el enrutador (*React Router*). Así solo se actualiza la cuadrícula de materiales del catálogo y la respuesta se siente casi instantánea.

4.3. Contratos de Datos y Modelado del Dominio

En un sistema tradicional de búsqueda, los filtros suelen tratarse como simples diccionarios o pares de clave-valor. Sin embargo, al introducir un LLM en la ecuación, la ambigüedad del lenguaje natural obliga a establecer fronteras matemáticas muy claras. En esta sección se describe cómo se han modelado los conceptos del negocio de Stylib para que los agentes puedan procesarlos sin margen de error.

4.3.1. Esquemas de comunicación inter-agente

Para garantizar que el *pipeline* sea tolerante a fallos (como se describió en la Sección 4.1.2), cada agente tiene prohibido devolver texto libre. En su lugar, firman un contrato de datos definido mediante clases de *Pydantic*, resumidos en las Tablas 4.1, 4.2 y 4.3.

El primer contrato crítico es el `LogicalAnalysisResult` (Tabla 4.1), la salida estructurada del *Entity Agent*. Este modelo obliga al LLM a clasificar la intención del usuario dentro de una única categoría principal y a desglosar la frase en una lista de `LogicalCondition`, donde cada condición separa el qué se busca (el concepto, como "Species" o "Thickness") del cómo se busca (la expresión, que puede ser una igualdad exacta como "oak", un operador lógico como "red OR blue", o un rango matemático como "<15").

Campo	Detalle
Esquema	`LogicalAnalysisResult`, salida estructurada del *Entity Agent*
Campos	`category: str`,categoría principal de la taxonomía detectada `conditions: list[LogicalCondition]`,condiciones extraídas, cada una con `concept: str` (campo conceptual, p. ej. *Species*) y `expression: str` (valor y relación lógica, p. ej. *oak*, *red OR blue*, *<15*) `reasoning: str`,razonamiento explícito (*Chain of Thought*) previo a la respuesta
Rol en el pipeline	Separa el qué se busca (`concept`) del cómo se busca (`expression`); sirve de entrada al *Filter Agent*

TABLA 4.1. DOCUMENTACIÓN DEL ESQUEMA DE SALIDA DEL *ENTITY AGENT*.

Posteriormente, el *Filter Agent* consume esta abstracción y devuelve un `FilterResult` (Tabla 4.2). Este esquema contiene el `FilterGroup` definitivo, la representación en árbol (con nodos AND/OR) que el backend de Stylib es capaz de ejecutar, junto con la cadena de filtrado ya serializada.

Campo	Detalle
Esquema	`FilterResult`, salida estructurada del *Filter Agent*
Campos	`category: str` y `conditions: list[LogicalCondition]`,heredados del *Entity Agent* `filter_group: FilterGroup`,árbol de nodos AND/OR de objetos `Filter` (`key`, `value`, `operator`, `negate`) `filter_string: str`,cadena de filtrado serializada `validation: ValidationResult | None`,resultado de la validación del *Validator Agent*
Operaciones	Un validador (`@model_validator`) calcula automáticamente `filter_string` a partir de `filter_group` mediante `to_filter_string()`

TABLA 4.2. DOCUMENTACIÓN DEL ESQUEMA DE SALIDA DEL *FILTER AGENT*.

Finalmente, la robustez del ciclo se sella con el `ValidationResult` (Tabla 4.3), el esquema devuelto por el *Validator Agent*. Si `is_valid` es falso, el orquestador sabe matemáticamente que debe abortar el envío al cliente e iniciar el bucle de reintento, enrutándolo según el campo `error_source`.

Campo	Detalle
Esquema	`ValidationResult`, salida estructurada del *Validator Agent*
Campos	`is_valid: bool`,indica si los filtros propuestos representan fielmente la consulta original `error_source: "entity" | "filter" | "both"`,etapa donde se originó el fallo detectado `issues: list[str]`,incidencias concretas encontradas `explanation: str`,valoración general de la validación
Rol en el pipeline	Si `is_valid` es `false`, el orquestador aborta el envío al cliente e inicia el bucle de reintento, enrutándolo según `error_source`

TABLA 4.3. DOCUMENTACIÓN DEL ESQUEMA DE SALIDA DEL *VALIDATOR AGENT*.

4.4. Gestión de la Base de Conocimiento (Taxonomía y Facetas)

Incluso utilizando el LLM más avanzado del mercado, el modelo desconoce por completo qué productos tiene Stylib o cómo se estructuran internamente sus bases de datos. Para evitar que el sistema invente materiales, es necesario inyectar el catálogo real como contexto en cada petición.

4.4.1. Estructura de la taxonomía base y facetas globales

El diccionario de datos original de la plataforma se divide en dos grandes bloques:

1. **Taxonomía (`taxonomy.json`):** Define la jerarquía pura de los materiales. Agrupa los productos en grandes categorías (ej. *Wood flooring*, *Carpets & rugs*, *Ceramic tiles*) y asocia a cada una los atributos técnicos que le son exclusivos. Por ejemplo, la taxonomía dicta que la categoría "Suelos de madera" tiene la propiedad "Especie" (*Species: oak, pine*), pero no tiene la propiedad "Construcción de pila" (*Pile construction*), que es exclusiva de las moquetas.
2. **Facetas (`facets.json`):** Representan las propiedades globales transversales. Son atributos como el Precio (*Price*), el Grosor (*Thickness*) o el Ancho (*Width*), que aplican por igual a una baldosa de cerámica que a una tabla de parqué. Al ser globales, el backend original de la empresa las almacena fuera de la taxonomía.

4.4.2. Consolidación de la taxonomía y las facetas

Pasar al contexto del LLM dos archivos JSON grandes y separados eleva el riesgo de alucinación, porque el modelo tiene que cruzar por su cuenta qué facetas globales aplican

a qué categorías.

Para evitarlo, antes de arrancar el *pipeline* el sistema consolida ambos archivos en una sola estructura. Una función (`_build_augmented_taxonomy`, documentada en la Tabla 5.1) recorre el archivo de facetas, se queda solo con las imprescindibles mediante una lista de permitidos y las incrusta dentro de los campos de cada categoría de la taxonomía. Esa lista de permitidos se reduce a cuatro facetas globales:

- Ancho (*Width*)
- Grosor (*Thickness*)
- Alto/longitud (*Height/length*)
- Precio (*Price*)

Gracias a esta inyección en tiempo de ejecución, el *Entity Agent* recibe a través de su entorno de ejecución (`RunContext`) un único "super diccionario" consolidado. Si el usuario pide "suelo de roble de menos de 40 euros", el modelo puede verificar en un solo lugar que la categoría "Suelos de madera" efectivamente soporta tanto la clave técnica "Especie" como la faceta numérica "Precio", asegurando un mapeo semántico preciso y sin inventar atributos inexistentes.

5. IMPLEMENTACIÓN TÉCNICA

El código fuente completo del sistema descrito en este capítulo se encuentra alojado en un repositorio privado, accesible bajo petición.[1]

5.1. Ingeniería de Prompts e Implementación de Agentes

Gran parte del trabajo de este proyecto consiste en estructurar el lenguaje natural para que una máquina pueda interpretarlo, más que en escribir código convencional. Un *prompt* conversacional genérico produce salidas ambiguas, poco aptas para alimentar un sistema automático. Por eso el problema se ha tratado como una traducción formal de lenguaje natural a estructuras de datos, en la línea de los sistemas NL2SQL.

Para que el modelo no invente atributos que no existen hacen falta reglas de inferencia estrictas. Partiendo de la literatura reciente sobre extracción de datos y diseño de instrucciones [21], cada agente sigue una misma metodología de cuatro fases:

1. *Role and Schema Context:* Se da al modelo el contexto operativo exacto. Al inyectar la taxonomía de la categoría se habilita el alineamiento relacional (*Schema Linking*) [32], que conecta el vocabulario del usuario con las claves reales de la base de datos.

2. *Task Decomposition:* Para que las consultas complejas no saturen al modelo, se le pide descomponer el problema paso a paso y resolver cada concepto por separado antes de montar la estructura completa [21].

3. *Chain of Thought (CoT) Examples:* Se introduce razonamiento explícito [22]. Con unos pocos ejemplos de aprendizaje en contexto (*few-shot learning*) [14], el modelo justifica su lógica antes de emitir el resultado final.

4. *Output Format:* Por último, Pydantic obliga a que la respuesta sea un objeto JSON válido y conforme al esquema.

A continuación se detalla cómo se ha implementado cada agente bajo este esquema.

5.1.1. Entity Agent: Extracción semántica y normalización

El *Entity Agent* es la primera etapa del *pipeline*. Se ocupa solo de analizar la consulta del usuario y extraer las entidades lógicas, sin conocer las claves internas del catálogo.

[1] Repositorio del proyecto: `https://github.com/stylib/llm-semantic-search`. Al tratarse de un desarrollo en el entorno de Stylib, el acceso se concede bajo petición razonada al autor o a los tutores.

Para un *Schema Linking* sólido [32], su contexto recibe en tiempo de ejecución la taxonomía completa de la categoría inferida. Siguiendo la *Task Decomposition*, el modelo genera objetos `LogicalCondition`, cada uno con un concepto (`concept`, p. ej. "Grosor") y su valor (`expression`, p. ej. "15-20").

La normalización numérica es una pieza delicada. El agente abstrae las unidades de medida y convierte descriptores ambiguos en rangos cerrados (*min-max*) o límites abiertos (p. ej. ">8"). El campo de razonamiento (`reasoning`) que rellena antes de generar datos, siguiendo CoT [22], le hace fijarse en los detalles y reduce la omisión de filtros secundarios.

5.1.2. Filter Agent: Mapeo de reglas y operadores canónicos

Con las condiciones lógicas ya extraídas, el *Filter Agent* actúa como mapeador estricto. Traduce los conceptos abstractos del *Entity Agent* a los objetos `Filter` que espera el *backend* de Stylib.

A diferencia del agente anterior, aquí el contexto se reduce a un diccionario de mapeo (`canonical_key_by_display`), que asocia los nombres de visualización con las claves canónicas del sistema, en lugar de la taxonomía completa. Por ejemplo, "Plank format" se corresponde con `meta.optionset.type::Plank format`.

Lo más difícil para este agente es inferir los operadores relacionales. A partir de la `expression`, y con instrucciones deterministas, asigna el operador que corresponde: `EQ` (igualdad), `IO` (inclusión múltiple tipo *OR*), `IA` (intersección tipo *AND*) o los de rango `GE/LE/GT/LT`. Su *prompt* le prohíbe terminantemente inventar claves que no figuren en la base de conocimiento que recibe [6].

5.1.3. Validator Agent: Autocorrección de alucinaciones

Por muy cuidados que estén los pasos anteriores, la naturaleza probabilística de estos modelos obliga a añadir controles. El *Validator Agent* hace de auditor independiente: revisa la coherencia semántica y sintáctica de lo que produce el *Filter Agent*. Es una aplicación directa del patrón *Reflexion* [23], que da al sistema capacidad de automejora a partir de retroalimentación en lenguaje natural.

El modelo recibe la consulta original, las condiciones lógicas del *Entity Agent* y la cadena de filtros propuesta, y las repasa siguiendo una lista de comprobaciones fija: que cada clave exista en el diccionario de atributos válidos, que los rangos numéricos estén codificados como dos filtros `GE/LE` y no como un único `EQ`, que los valores elegidos encajen con la consulta, que la categoría detectada corresponda al producto pedido, que no haya filtros contradictorios entre sí, que no se haya colado ningún valor inventado y que no falte ningún filtro que la consulta pedía de forma explícita.

Cuando detecta una desviación, por ejemplo una clave alucinada como `supplierHeader`,

no falla en silencio. Devuelve un objeto `ValidationResult` con `is_valid` en falso, describe el problema de forma accionable en el vector `issues` y, como último paso de la revisión, clasifica el origen del fallo en el campo `error_source`: si la responsabilidad es del *Entity Agent*, del *Filter Agent* o de ambos. Esa clasificación es justo lo que permite, como se verá en la siguiente sección, dirigir el bucle de reintentos al agente que corresponde en lugar de repetir todo el proceso desde cero.

5.2. Orquestación del Sistema Multiagente

Para que los tres agentes funcionen juntos hace falta un controlador central que gestione el flujo de datos, inyecte el contexto de cada fase y coordine los ciclos de evaluación. Esa orquestación vive en un *pipeline* asíncrono escrito en Python, que convierte la consulta inicial en un objeto `PipelineResult` con los filtros finales y las métricas de rendimiento del proceso.

Sus tres componentes principales se describen a continuación.

5.2.1. Construcción e inyección del contexto

Como se justificó antes, el sistema necesita conocer el esquema de la base de datos para un *Schema Linking* preciso. El problema es que, en los datos de Stylib, algunas facetas globales como el precio o las dimensiones físicas son transversales y no cuelgan de ninguna categoría concreta.

Para resolver esa fragmentación, una rutina consolida la taxonomía en tiempo de ejecución. La función `build_augmented_taxonomy` recorre las facetas globales, se queda con las imprescindibles y reescribe su estructura para insertarlas en cada categoría de la taxonomía.

Campo	Detalle
Función	`_build_augmented_taxonomy(taxonomy, facets)`
Entrada	`taxonomy: dict`,estructura base de la taxonomía `facets: list[dict]`,facetas globales a inyectar
Salida	`dict`,taxonomía consolidada con las facetas globales inyectadas en cada categoría
Operaciones	1. Copia profunda de la taxonomía mediante `deepcopy` 2. Filtrado de facetas según `_OPTIONSET_FACETS_TO_INJECT` 3. Transformación estructural de cada faceta con `_facet_to_taxonomy_field` 4. Inyección del campo transformado en cada categoría del diccionario

TABLA 5.1. DOCUMENTACIÓN DE LA FUNCIÓN DE INYECCIÓN DE FACETAS GLOBALES EN LA TAXONOMÍA.

El resultado es un único diccionario consolidado que se pasa al *Entity Agent* mediante su entorno de dependencias (`deps=EntityContext(taxonomy)`). En paralelo se arma otro diccionario, `canonical_key_by_display`, que resuelve los nombres de interfaz hacia sus claves canónicas. Este se encapsula en los contextos `FilterContext` y `ValidatorContext` y se inyecta en las fases siguientes.

5.2.2. Lógica de control del Pipeline Central

El núcleo de la ejecución es la función asíncrona `run_pipeline`. Sigue un patrón de inyección de dependencias que permite asignar un modelo distinto a cada agente (`model_entity`, `model_filter`, `model_validator`). Esta flexibilidad es la que hace posible la experimentación: se puede dar la extracción más exigente a un modelo potente y dejar la validación a uno más rápido.

El control es estrictamente secuencial. Primero se llama al *Entity Agent* con la consulta en lenguaje natural. Devuelve la categoría detectada y la lista de condiciones lógicas (`LogicalAnalysisResult`), que ya no se modifican y sirven de entrada a la fase siguiente.

La implementación con *PydanticAI* aporta algo útil en cómo se transmite el contexto. En vez de concatenar el diccionario de mapeo en el mensaje del usuario en cada interacción, se inyecta como dependencia en las instrucciones de sistema del agente:

Campo	Detalle
Función	`_inject_context(ctx: RunContext[ValidatorContext])`
Entrada	`ctx: RunContext[ValidatorContext]`,contexto de ejecución con las dependencias inyectadas
Salida	`str`,cadena JSON formateada lista para su inyección en las instrucciones de sistema del agente
Operaciones	1. Extracción de `canonical_key_by_display` desde `ctx.deps` 2. Serialización del diccionario con `json.dumps(..., indent=2)` 3. Prefijo identificativo de la clave para el *prompt* de sistema

TABLA 5.2. DOCUMENTACIÓN DE LA FUNCIÓN DE INYECCIÓN DE DEPENDENCIAS EN EL CONTEXTO DE EJECUCIÓN.

Así el *prompt* no crece en cada iteración y se ahorra consumo de *tokens*.

5.2.3. Mecanismo de autocuración (Retry Loop)

El patrón *Reflexion* [23] toma forma en esta fase como un bucle de reintentos explícito (*Retry Loop*). En lugar de quedarse con la primera salida, la ejecución entra en un bloque iterativo asíncrono, gestionado por la función `_retry_loop` y limitado por la constante `MAX_RETRIES`.

En cada vuelta se ejecutan los agentes en orden: el *Entity Agent* (solo cuando hay que rehacer la extracción), el *Filter Agent* y, por último, el *Validator Agent*, que examina el resultado. Si da el visto bueno (`is_valid == True`), el bucle se rompe (`break`) y se consolidan los resultados.

El reintento en sí es lo de menos: lo que importa es a quién se devuelve el control cuando algo falla. El validador no se limita a marcar la salida como inválida: también clasifica el origen del error en el campo `error_source`, que distingue entre un fallo del *Entity Agent*, del *Filter Agent* o de ambos. Con esa señal, el bucle enruta el reintento por una de dos vías, tal como refleja el diagrama de componentes (Figura 4.1):

- *Retry* local (fallo de mapeo). El problema está en el mapeo: una clave inexistente o un operador mal asignado. Se reejecuta solo el *Filter Agent*, al que `build_retry_input` le inyecta los errores concretos señalados por el validador. La extracción no se repite.
- *Retry* global (fallo semántico). El error nace antes, en la interpretación de la consulta (categoría equivocada o condiciones omitidas). Aquí corregir solo el mapeo no sirve de nada, así que se vuelve al *Entity Agent* con el *feedback* del validador añadido a la consulta original, para que rehaga la extracción.

Campo	Detalle
Función	`_retry_loop` (asíncrona, acotada por `MAX_RETRIES`)
Entrada	`query: str`,consulta original del usuario `entity_out`,resultado del *Entity Agent* `validator_deps`,dependencias del *Validator Agent* `canonical_key_by_display: dict`,diccionario de mapeo de claves
Salida	Resultado validado (`PipelineResult`) o excepción tras agotar el número máximo de reintentos
Operaciones	1. Ejecución del *Validator Agent* sobre la salida del *Filter Agent* 2. Si `is_valid == True`: salida del bucle y consolidación del resultado 3. Si `error_source` es **"entity"** o **"both"**: reintento global,se reenvía la consulta al *Entity Agent* con el *feedback* del validador concatenado 4. En caso contrario: reintento local,se reejecuta solo el *Filter Agent* con la entrada corregida por `build_retry_input`

TABLA 5.3. DOCUMENTACIÓN DE LA LÓGICA DE ENRUTAMIENTO DEL MECANISMO DE AUTOCURACIÓN.

Esta distinción evita reintentos inútiles, ya que no tiene sentido pedir al mapeador que arregle un fallo que viene de una mala interpretación inicial. Al atacar cada error en la etapa que de verdad lo origina, el sistema aísla a la aplicación cliente de buena parte de las alucinaciones de los modelos generativos.

5.3. Exposición del Servicio e Integración

La última fase consiste en empaquetar el *pipeline* multiagente y exponerlo para usarlo en producción. Se ha planteado como un servicio independiente, listo para integrarse más adelante con el *backend* principal de Stylib.

5.3.1. Desarrollo de la API REST con FastAPI

El servicio web se expone con una API REST construida sobre FastAPI [27]. Su carácter asíncrono es imprescindible, porque las llamadas a los modelos de lenguaje arrastran latencias de red altas.

El contrato de comunicación se apoya en la validación de Pydantic. Las peticiones y respuestas se ciñen a esquemas estrictos (`FilterRequest` y `FilterResponse`), de modo

que la API comprueba la integridad del paquete antes de invocar al orquestador.

Campo	Detalle
Función	`get_filters(payload: FilterRequest)`,ruta `POST /filters`
Entrada	`payload: FilterRequest`,esquema Pydantic con el campo `query: str`
Salida	`FilterResponse`,esquema con los campos `query`, `category`, `filter_string`, `encoded_filter_string`, `validation_ok` e `issues`
Operaciones	1. Invocación de `run_pipeline(payload.query)` para obtener los filtros validados 2. Serialización del grupo de filtros con `FilterEncoder.from_filter_group` 3. Codificación del resultado en Base64 y *percent-encoding* de los caracteres especiales para que la cadena viaje de forma segura como parámetro de la URL (`b64=True`) 4. Gestión de excepciones con respuesta HTTP 500 ante fallos internos

TABLA 5.4. DOCUMENTACIÓN DEL *ENDPOINT* PRINCIPAL DE LA API REST.

Como se ve en la Tabla 5.4, el *endpoint* gestiona las excepciones y responde con códigos HTTP estándar (500) ante un fallo interno, lo que facilita la depuración.

Cuando el sistema multiagente devuelve los filtros validados, el *backend* no consulta directamente la base de datos de productos. Sigue un enfoque desacoplado: empaqueta el grupo de filtros con la clase `FilterEncoder`, lo codifica en Base64 y aplica *percent-encoding* para que el resultado sea seguro como parámetro de la URL.

La API devuelve ese identificador (`encoded_filter_string`) al *frontend*, que lo añade a la URL del navegador como parámetro de búsqueda. Al detectar el cambio, la vista del catálogo se recarga y lanza una petición HTTP normal contra la base de datos real de Stylib.

Este diseño aporta tres ventajas para la integración:

1. Desacoplamiento: La infraestructura de Inteligencia Artificial queda separada de la base de datos transaccional del catálogo.

2. Persistencia de estado: Las búsquedas generadas por lenguaje natural se convierten en URLs compartibles.

3. Robustez sintáctica: Al ir en Base64, el filtro no muestra su sintaxis en la barra de direcciones y se evitan errores de formato (*parsing*) durante la navegación.

6. EXPERIMENTACIÓN Y EVALUACIÓN

Una vez definida la arquitectura y completada la implementación técnica del sistema multiagente, debemos pasar a validar la solución de manera empírica. Este capítulo detalla el marco de evaluación diseñado para medir tanto la precisión semántica en la extracción de filtros como la viabilidad del sistema en un entorno de producción: la latencia, el consumo de *tokens* y el coste económico de los distintos modelos fundacionales.

A través de una serie de experimentos controlados, se evaluará el rendimiento de modelos individuales, el impacto del agente validador y la eficiencia de configuraciones híbridas, culminando en un análisis cualitativo de los límites actuales de la Inteligencia Artificial en el mapeo de taxonomías complejas.

6.1. Metodología de Evaluación y Entorno de Pruebas

Evaluar sistemas generativos en tareas de estructuración de datos requiere herramientas que trasciendan la simple coincidencia exacta de cadenas de texto (*Exact Match*). El sistema puede generar un filtro lógicamente correcto pero con los atributos desordenados, por lo que es necesario un enfoque de evaluación granular.

6.1.1. Construcción del Golden Dataset

Para establecer la base de la verdad (*ground truth*), se ha construido de forma manual un conjunto de datos de referencia, denominado *Golden Dataset*, compuesto por 52 consultas en lenguaje natural. Estas consultas fueron diseñadas para representar fielmente la variabilidad e intención de búsqueda de un usuario real en la plataforma Stylib.

El conjunto de datos abarca 6 categorías principales de productos (como *Wood flooring*, *Carpets & rugs* o *Cork*) e incluye un amplio espectro de complejidad lingüística: descripciones ambiguas, rangos numéricos explícitos, condiciones de exclusión (negaciones) y relaciones lógicas de tipo *OR*. Cada consulta ha sido rigurosamente anotada con:

- La categoría de producto esperada.
- Las entidades lógicas subyacentes (`LogicalConditions`).
- La cadena de filtros definitiva en formato serializado (`key[op]:value`), utilizando más de cinco operadores relacionales distintos (`eq`, `io`, `ia`, `ge`, `le`) y el prefijo `!` para negaciones.

6.1.2. Marco de métricas multidimensional

Para capturar el rendimiento del sistema desde distintos ángulos se diseñó un marco de evaluación con ocho métricas complementarias:

- *Token F1* (Métrica principal): Trata cada filtro como una tupla compuesta por la clave, el operador, los valores ordenados y el indicador de negación. Mide el solapamiento entre el conjunto de filtros predicho y el esperado. Para evitar penalizaciones sintácticas injustas, los operadores semánticamente equivalentes para un solo valor (`eq`, `ia`, `io`) se normalizan internamente.
- *Key F1* y *Value-set F1*: Separan la evaluación en dos ejes. *Key F1* penaliza si el modelo falla al identificar las facetas correctas (ej. confunde grosor con longitud). *Value-set F1* otorga crédito parcial si la clave es correcta pero solo se extraen algunos de los valores solicitados.
- *Exact Match Rate*: Porcentaje de consultas donde la predicción coincide al 100 % con el *ground truth* tras la normalización.
- Precisión de Operador y Negación: Evalúa, exclusivamente sobre las claves acertadas, con qué frecuencia el modelo deduce el operador relacional y la polaridad (negación) correctos.
- *Filter Count Delta*: Calcula la diferencia neta entre los filtros predichos y los esperados ($|Pred| - |Exp|$). Un valor negativo indica que el modelo omite condiciones, mientras que un valor positivo revela alucinaciones (filtros inventados).
- Métricas de Producción (Latencia y Coste): Se registra el tiempo de ejecución (*Avg* y percentil 95) por agente y del sistema total. Asimismo, se cuantifica el consumo de *tokens*: la etapa del *Entity Agent* concentra entre el 93 % y el 96 % del coste, debido a la inyección de la taxonomía.

La Figura 6.1 desglosa el consumo de *tokens* por etapa del *pipeline* y el coste medio por consulta. En la mitad izquierda se aprecia que la franja del Entity Agent ocupa casi toda la barra en los modelos Gemini, mientras que las etapas de Filter y Validator quedan reducidas a una porción mínima. En Mercury la franja de Filter crece algo más, reflejo de los reintentos que necesita. La mitad derecha ordena el coste medio por consulta y deja claro el rango del estudio: desde los 0,021 $ de las configuraciones más baratas hasta los 0,066 $ de Kimi. Este desglose justifica por qué la optimización de coste del Experimento 2 se centra en abaratar las etapas de mapeo: son las únicas sobre las que se puede actuar sin tocar la parte cara, que es la extracción.

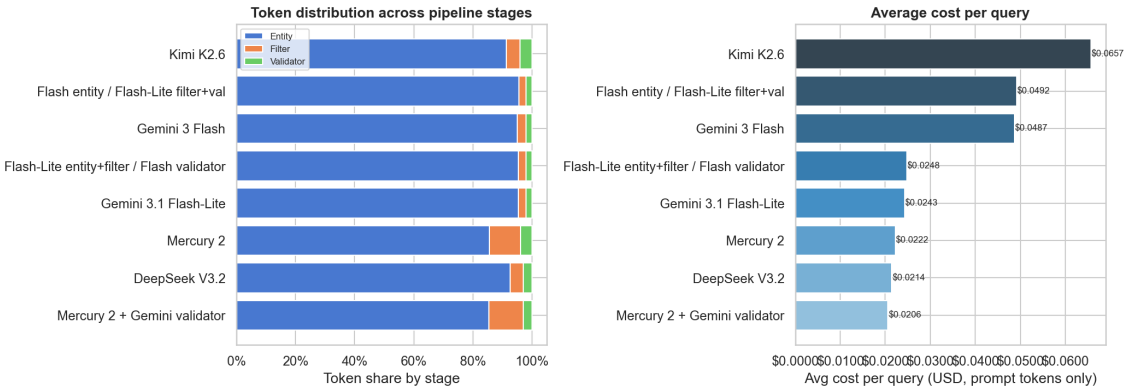


Fig. 6.1. Reparto de *tokens* por etapa del pipeline (izquierda) y coste medio por consulta (derecha).

6.1.3. Modelos y configuraciones evaluadas

Para los experimentos se han seleccionado cinco modelos fundacionales con características arquitectónicas y de coste diferenciadas, accesibles todos a través de la pasarela unificada OpenRouter:

1. *Gemini 3 Flash* (`google/gemini-3-flash-preview`): Modelo de gran capacidad y razonamiento profundo, representante del estado del arte comercial, con un coste computacional mayor.

2. *Gemini 3.1 Flash-Lite* (`google/gemini-3.1-flash-lite-preview`): Versión destilada del modelo anterior, optimizada para tareas de baja latencia y alta eficiencia económica.

3. *Mercury 2* (`inception/mercury-2`): LLM basado en arquitecturas de difusión. Se ha incluido como línea base no autorregresiva. Debido a inestabilidades detectadas en su API (errores 502/529), su configuración requirió ajustar parámetros de tolerancia (`max_tokens=4096` y reintentos en pasarela).

4. *DeepSeek V3.2* (`deepseek/deepseek-v3.2`): Modelo abierto de coste muy reducido por *token*. Se incorporó para comprobar si una alternativa económica del ecosistema abierto podía competir con la familia Gemini, asumiendo una latencia notablemente más alta.

5. *Kimi K2.6* (`moonshotai/kimi-k2.6`): Modelo de gran tamaño con calidad cercana a la de Gemini. Su inclusión permite contrastar un sistema de altas prestaciones, aunque su tiempo de respuesta resultó ser el más alto de todo el conjunto.

6.1.4. Herramientas de instrumentación

Todo el entorno de pruebas se ha orquestado programáticamente utilizando la librería propia del ecosistema *Pydantic* para evaluaciones (`pydantic-evals`). La trazabilidad de los experimentos, la captura de la latencia por agente y la monitorización en tiempo real del uso de la red se llevaron a cabo mediante *Logfire*, lo que permitió una recolección de datos auditable y reproducible.

6.2. Experimento 1: Evaluación Base por Modelo Uniforme (Single-Model)

6.2.1. Diseño del experimento

El primer experimento busca una referencia limpia: cómo se comporta cada modelo cuando no se mezcla con ningún otro. Para ello se ejecutó el pipeline completo (`entity_filter_validated`), con sus tres etapas de Entity, Filter y Validator, pero forzando a que los tres agentes usaran exactamente el mismo modelo. Así, cualquier diferencia en los resultados se atribuye al modelo y no a una combinación afortunada de varios.

Cada uno de los cinco modelos resolvió las 52 consultas del *Golden Dataset*. La métrica principal fue el Token F1, acompañada del *Exact Match*, la latencia media por consulta, el número medio de reintentos del bucle de validación y el coste medio en dólares calculado a partir de los *tokens* de *prompt* consumidos y las tarifas de OpenRouter. El umbral de aprobado se fijó en un Token F1 de 0,5.

6.2.2. Análisis cuantitativo: calidad, latencia y coste

La Tabla 6.1 recoge el rendimiento de los cinco modelos en configuración uniforme. El primer dato que llama la atención es lo cerca que quedan los tres mejores. Gemini 3 Flash encabeza la calidad con un Token F1 de 0,925 y un *Exact Match* de 0,750, pero Kimi K2.6 (0,916) y Gemini 3.1 Flash-Lite (0,915) se sitúan a apenas una centésima. La diferencia real entre ellos está en lo que cuesta obtener el F1.

Modelo	Token F1	Exact Match	Lat. media (s)	Reintentos	Coste/consulta
Gemini 3 Flash	0,925	0,750	10,11	0,038	0,0487 $
Kimi K2.6	0,916	0,635	214,34	0,058	0,0657 $
Gemini 3.1 Flash-Lite	0,915	0,673	11,50	0,019	0,0243 $
DeepSeek V3.2	0,849	0,481	48,48	0,442	0,0214 $
Mercury 2	0,792	0,346	13,80	0,423	0,0222 $

TABLA 6.1. RENDIMIENTO DE LOS CINCO MODELOS

Kimi resulta revelador. Iguala a la familia Gemini en calidad, pero tarda de media 214 segundos por consulta frente a los 10 u 11 de los modelos Gemini. Veinte veces más lento

por la misma precisión, y la causa no es la longitud de la respuesta: Kimi consume de media 87.600 *tokens* por consulta, una cifra similar (incluso algo menor) a los 97.400 de Gemini 3 Flash. La diferencia está en la velocidad de inferencia, entre 16 y 33 veces más lenta en las tres etapas del *pipeline* por igual, lo que apunta a la infraestructura que sirve el modelo en OpenRouter más que a la tarea en sí. En un buscador donde el usuario espera resultados en tiempo real, ese coste de latencia lo descarta de inmediato, por buena que sea su salida. Algo parecido ocurre con DeepSeek: su coste por *token* es el más bajo de la tabla, pero su latencia de 48 segundos y su *Exact Match* de 0,481 lo dejan lejos de ser competitivo.

Mercury 2 cierra la tabla. Su Token F1 de 0,792 supera el umbral, así que el sistema sigue siendo funcional con él, pero su *Exact Match* de 0,346 indica que solo un tercio de sus respuestas coincide al cien por cien con la solución esperada. El número de reintentos también lo delata: 0,423 frente a los 0,038 de Gemini 3 Flash. Mercury necesita que el validador lo corrija con mucha más frecuencia que el resto.

La Figura 6.2 muestra estos mismos valores de forma visual, con y sin la etapa de validación. Casi todos los modelos cruzan el umbral de aprobado con holgura, y la mitad derecha de la gráfica (con validador) tiende a estar algo más comprimida hacia arriba que la izquierda, lo que adelanta el efecto que estudiaremos en el Experimento 3.

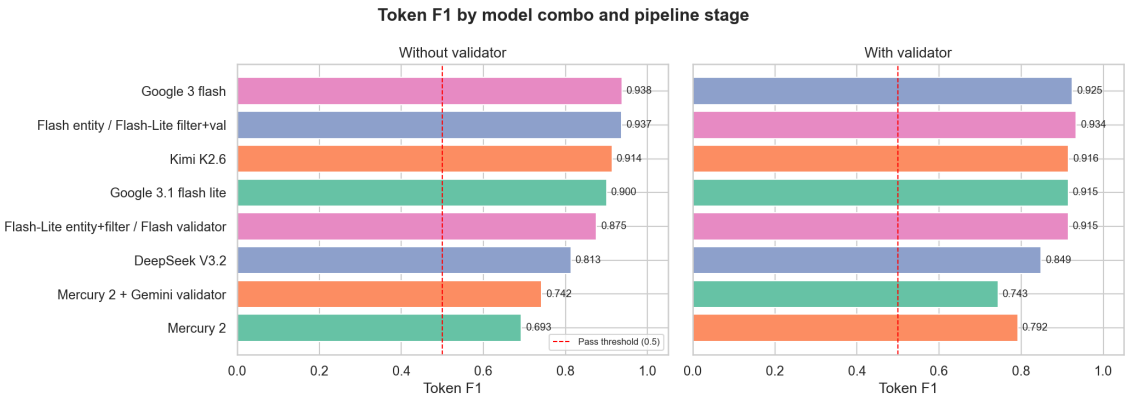


Fig. 6.2. Token F1 por modelo y etapa del pipeline, con y sin validador.

En cuanto a la clasificación de categoría, el resultado fue impecable para los modelos Gemini y para Kimi: acertaron la categoría de producto en las 52 consultas. DeepSeek falló una (51 de 52) y Mercury falló tres (49 de 52). Es un indicio temprano de que la parte difícil del problema es la traducción al filtro exacto.

6.2.3. Análisis de los modelos rezagados

Conviene detenerse en por qué Mercury 2 y DeepSeek V3.2 quedan por debajo de la familia Gemini, porque sus fallos no son del mismo tipo. En ambos casos el problema está en dejarse condiciones por el camino, no en inventarse valores que no existen. La métrica

de *filter count delta* lo refleja con claridad: ambos arrojan valores negativos, es decir, generan menos filtros de los que pide la consulta. Sin la red de seguridad del validador, DeepSeek llega a un delta de -0,67 y Mercury de -0,62, lo que significa que pierden más de medio filtro por consulta de media. Cuando una búsqueda combina tres o cuatro condiciones, omitir una cambia por completo el conjunto de resultados.

DeepSeek añade un problema propio de tiempo. Su latencia de 48 segundos responde a la naturaleza del modelo, no a la dificultad de la tarea, y eso lo deja fuera de un escenario interactivo aunque su coste por *token* sea atractivo. Mercury, por su parte, arrastra la inestabilidad de su API. Durante las pruebas devolvió errores 502 y 529 con cierta frecuencia, lo que obligó a subir el límite de *tokens* y a apoyarse en los reintentos de la pasarela para completar el barrido.

De este primer experimento se extrae una conclusión que orientó el resto del trabajo: la familia Gemini domina el problema, y la decisión importante es ver cuál de los modelos resuelve las tareas con un coste y latencia adecuados. Kimi demuestra que más potencia no significa mejores filtros aquí, y los modelos rezagados confirman que el cuello de botella está en la precisión fina del mapeo, no en la comprensión general de la consulta. Eso es justo lo que motiva el siguiente experimento.

6.3. Experimento 2: Evaluación de Combinaciones Híbridas (Model Sweep)

6.3.1. Diseño del barrido: asignación diferenciada por agente

El primer experimento trataba cada modelo como un bloque único. Pero la arquitectura del sistema nos permite dividirlo, ya que las tres etapas son independientes y nada impide asignar un modelo distinto a cada una. De ahí surge la hipótesis de este segundo experimento. El *Entity Agent* es la etapa que más razonamiento requiere, porque debe entender la consulta y descomponerla en condiciones lógicas. El Filter Agent y el Validator, en cambio, realizan tareas más mecánicas de mapeo y comprobación contra un conjunto de claves conocido.

Si esto es cierto, debería bastar con un modelo potente y caro en la primera etapa y uno más ligero y barato en las dos siguientes, sin perder calidad apenas. Para comprobarlo se diseñaron tres combinaciones híbridas: Gemini 3 Flash en Entity con Flash-Lite en Filter y Validator, la asignación inversa con Flash-Lite en Entity y Flash en el validador, y una tercera con Mercury 2 en las dos primeras etapas apoyado por un validador Gemini, para ver si un validador fuerte podía rescatar a un extractor débil.

6.3.2. Comparativa de configuraciones (Entity, Filter y Validator)

La Tabla 6.2 reúne las tres combinaciones híbridas junto a los dos modelos Gemini uniformes, que sirven de referencia. El resultado confirma la hipótesis. La configuración

con Gemini 3 Flash en la extracción y Flash-Lite en las dos etapas restantes alcanza un Token F1 de 0,934, el más alto de todo el estudio, por encima incluso del Gemini 3 Flash uniforme (0,925). Su *Exact Match* de 0,769 también es el mejor.

Configuración	**Token F1**	**Exact Match**	**Lat. media (s)**
Flash entity / Flash-Lite filtro+val	0,934	0,769	10,42
Gemini 3 Flash (uniforme)	0,925	0,750	10,11
Flash-Lite entity+filtro / Flash validador	0,915	0,673	10,19
Gemini 3.1 Flash-Lite (uniforme)	0,915	0,673	11,50
Mercury 2 + validador Gemini	0,743	0,269	13,95

TABLA 6.2. CONFIGURACIONES HÍBRIDAS FRENTE A LOS MODELOS GEMINI

El orden de los factores importa. Poner el modelo potente en la extracción y el ligero en el mapeo funciona; hacerlo al revés, no. La configuración inversa, con Flash-Lite extrayendo entidades y Flash validando, se queda en 0,915, exactamente igual que el Flash-Lite uniforme. Es decir, gastar un modelo caro en el validador no aporta nada cuando la extracción ya ha fijado el techo de calidad. Esto valida la intuición de partida: la etapa que decide el resultado es el *Entity Agent*.

La tercera combinación lo demuestra por la vía negativa. Mercury extrayendo entidades con un validador Gemini se queda en 0,743 y un *Exact Match* de apenas 0,269. Un buen validador no puede reconstruir un filtro que el extractor nunca llegó a plantear. Si la condición se perdió en la primera etapa, ya no hay nada que corregir después.

6.3.3. Análisis coste-calidad y conclusiones sobre la configuración óptima

La calidad por sí sola no decide la configuración de producción. Hay que ponerla frente al coste, y eso es lo que recoge la Figura 6.3, donde cada modelo aparece situado según su precio medio por consulta y su Token F1. La gráfica reúne en un mismo plano las configuraciones de la Tabla 6.2 y los cinco modelos individuales del Experimento 1 (Tabla 6.1); por eso aparecen también Kimi K2.6, DeepSeek V3.2 y Mercury 2 uniforme, que no formaron parte del barrido de combinaciones híbridas.

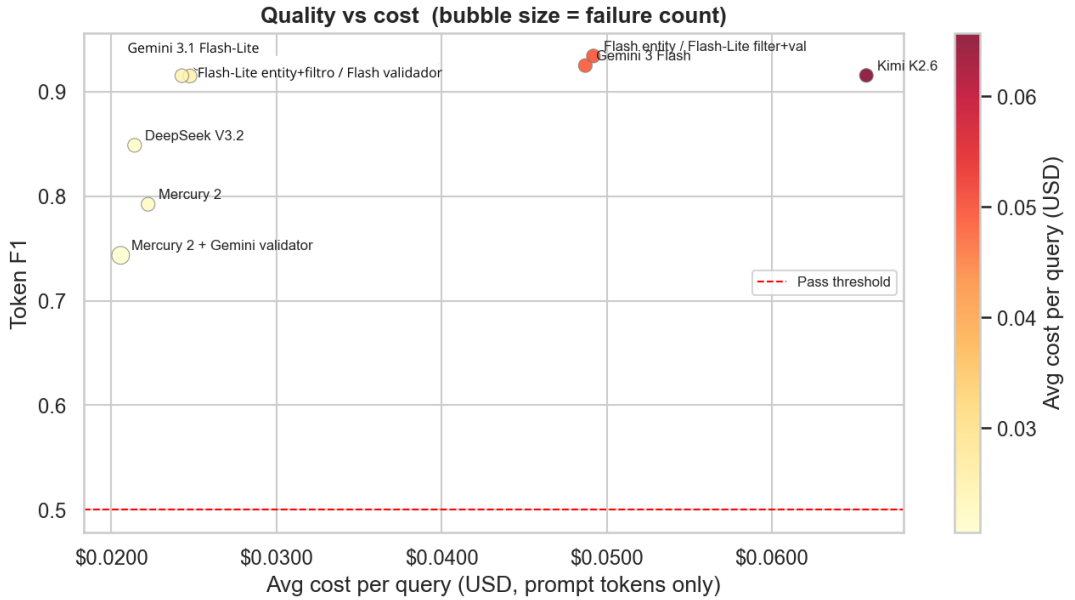


Fig. 6.3. Frontera coste-calidad. Token F1 frente a coste medio por consulta en dólares.

La gráfica deja ver dos grupos. A la izquierda, con costes en torno a 0,02 $ por consulta, se agolpan las configuraciones basadas en Flash-Lite, junto con DeepSeek V3.2 y Mercury 2 uniforme. Estos dos últimos comparten el rango de coste pero se quedan muy por debajo en Token F1 (0,849 y 0,792): de ahí que no se probaran en ninguna combinación de este experimento. A la derecha, sobre los 0,05 $, están Gemini 3 Flash y el híbrido que lo usa para extraer. Kimi queda aún más a la derecha, en 0,066 $, como la opción más cara sin ser la más precisa. Lo interesante es que la mejora de calidad al pasar del grupo barato (Flash-Lite) al caro es pequeña, de unas dos centésimas de F1, mientras que el coste se duplica.

Esto abre dos lecturas según la prioridad. Si lo que se busca es la máxima calidad posible, la configuración *Flash entity / Flash-Lite filtro+val* es la ganadora: el mejor F1 del estudio a un coste de 0,049 $, casi idéntico al del Gemini 3 Flash uniforme pero con una salida algo mejor, porque mover el mapeo y la validación a Flash-Lite no penaliza la calidad y sí abarata esas dos etapas. Si en cambio el objetivo es exprimir el presupuesto, el Gemini 3.1 Flash-Lite uniforme entrega un 0,915 de F1 a la mitad de precio (0,024 $), una rebaja del cincuenta por ciento a cambio de poco menos de dos puntos de calidad.

La conclusión práctica de este experimento es que la asignación híbrida es una palanca real de optimización, no un truco marginal. Permite recuperar el techo de calidad del modelo más caro mientras se traslada el grueso del trabajo barato a un modelo ligero. Para el despliegue en Stylib se adoptó por defecto el Gemini 3.1 Flash-Lite.

6.4. Experimento 3: Impacto del Agente Validador

6.4.1. Comparativa: pipeline de dos etapas vs. pipeline completo

El tercer agente, el Validador, es el más caro de justificar. Añade una llamada extra al modelo, suma latencia y puede disparar reintentos. La pregunta es si todo eso compensa. Para responderla se ejecutó cada configuración dos veces: una con el pipeline de dos etapas (`entity_filter`, solo extracción y mapeo) y otra con el pipeline completo (`entity_filter_validated`), que añade la validación y el bucle de reintentos de hasta tres intentos.

Más allá del F1, el validador tuvo un efecto que las medias de calidad no capturan del todo: redujo las ejecuciones fallidas. En la variante de dos etapas, Mercury 2 dejó cinco consultas sin completar y DeepSeek cuatro, normalmente por salidas malformadas o cortadas. Con el validador y sus reintentos, esos fallos casi desaparecen, porque una primera respuesta inválida tiene una segunda oportunidad en lugar de quedar perdida. Esa robustez es difícil de ver en una tabla de F1 medio, pero pesa mucho en un servicio real.

6.4.2. Autocorrección observada: reintentos y mejora por modelo

El efecto del validador sobre la calidad no es uniforme. La Tabla 6.3 y la Figura 6.4 muestran el delta de Token F1, es decir, la diferencia entre ejecutar con validador y sin él. Un valor positivo significa que la validación mejoró el resultado.

Configuración	**Δ Token F1**
Mercury 2	+0,100
Flash-Lite entity+filtro / Flash validador	+0,040
DeepSeek V3.2	+0,035
Gemini 3.1 Flash-Lite	+0,015
Kimi K2.6	+0,001
Mercury 2 + validador Gemini	+0,001
Flash entity / Flash-Lite filtro+val	-0,003
Gemini 3 Flash	-0,013

TABLA 6.3. IMPACTO DEL VALIDADOR SOBRE EL TOKEN F1

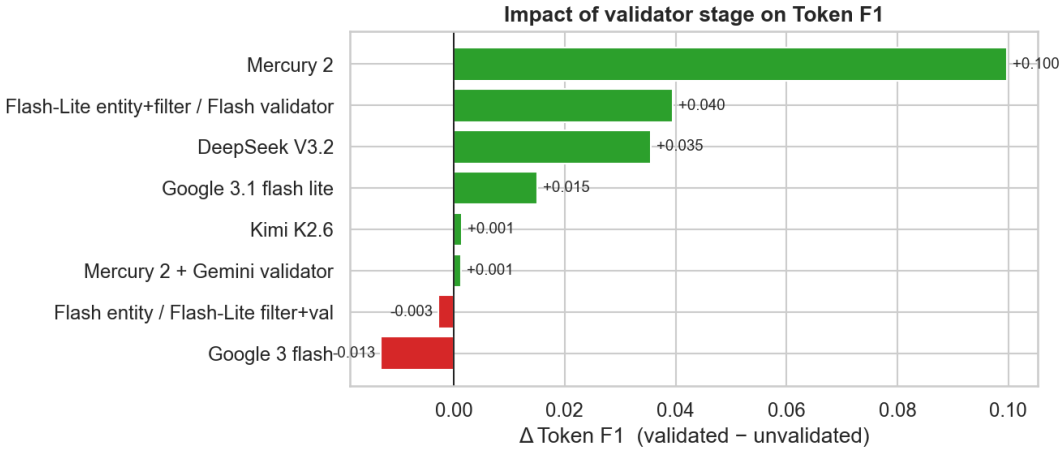


Fig. 6.4. Efecto de la etapa de validación sobre el Token F1 por configuración.

Como se puede ver, el validador rescata a los modelos débiles y apenas toca a los fuertes. Mercury 2 es el caso extremo: gana +0,100 de F1, lo que lo lleva de un 0,692 que rondaba el aprobado a un 0,792 cómodo. DeepSeek sube +0,035 y la configuración Flash-Lite con validador Flash gana +0,040. En todos ellos, la segunda mirada del validador detecta omisiones o errores de mapeo que el extractor barato había dejado pasar, y el bucle de reintentos los corrige.

En el otro extremo aparece un matiz: Gemini 3 Flash empeora ligeramente con el validador, un -0,013, y el híbrido con Flash en la extracción pierde -0,003. Cuando el extractor ya es muy bueno, el validador a veces corrige filtros que en realidad estaban bien, y eso introduce un ruido marginal. No es un fallo grave, pero sí una señal de que la validación no es gratis ni siempre positiva.

Por lo tanto, el validador es un seguro. En los modelos potentes su coste no se recupera en calidad y solo aporta robustez frente a fallos puntuales; en los modelos débiles o baratos, en cambio, es justo lo que los hace utilizables. Como la configuración elegida para producción se apoya en Flash-Lite, mantener el validador resulta plenamente justificado: es ahí donde su mejora de +0,015 y su capacidad de recuperar respuestas fallidas tienen más valor.

6.5. Evaluación Cualitativa y Límites del Sistema

6.5.1. Análisis por tipo de consulta y precisión categórica

Las métricas agregadas dan una visión de conjunto, pero conviene bajar al detalle de qué hace bien el sistema y dónde tropieza. Hay dos aspectos en los que el comportamiento fue prácticamente perfecto en la familia Gemini.

El primero es la clasificación de categoría. Tanto Gemini 3 Flash como Gemini 3.1 Flash-Lite y Kimi acertaron la categoría de producto en las 52 consultas del *Golden Da-*

taset, sin un solo fallo. Esto incluye consultas ambiguas donde la categoría no se nombra de forma explícita y hay que deducirla del contexto. El segundo es la negación. La precisión de polaridad fue de 1,0: cuando la consulta pedía excluir algo, por ejemplo "suelos de madera que no sean de roble", el sistema colocó el prefijo de negación en el filtro correcto sin confundirse. La precisión de operador relacional también fue de 1,0 en estos modelos, así que la elección entre igualdad, inclusión o los operadores de rango numérico se resolvió sin errores sobre las claves acertadas.

Estos dos resultados, sumados a la consistencia de los modelos Gemini frente a la dispersión de Kimi en latencia que ya vimos en el Experimento 1, confirman que la parte de comprensión de alto nivel está prácticamente resuelta. El sistema entiende qué se le pide; la dificultad, como veremos a continuación, está en el último tramo de precisión.

6.5.2. Desalineaciones semánticas y alucinaciones estructurales

Vale la pena terminar mirando los fallos que quedan, porque son más informativos que los aciertos. Lo primero que destaca al revisar los errores consulta a consulta es lo que *no* ocurre. El sistema casi nunca se inventa valores. No aparecen colores que la consulta no mencionaba ni proveedores fantasma. El comportamiento sobre los operadores lo respalda: la precisión de operador fue de 1,0 sobre las claves acertadas, y las pocas variaciones que se observan se dan entre operadores semánticamente equivalentes para un único valor (`eq`, `io`), que el marco de evaluación normaliza por considerarlos intercambiables. Cuando el modelo acierta la clave, acierta también el operador. La alucinación de rellenar huecos con datos inventados apenas se da.

Por consiguiente, el error real tiene dos formas. La primera es la confusión de clave dentro de una misma categoría. El modelo entiende la intención del usuario, pero la asigna a la faceta vecina equivocada. El caso típico es confundir el grosor (*Thickness*) con la altura de la pila (*Pile height*) en alfombras, o mezclar dos atributos dimensionales que comparten unidades. El usuario pedía algo concreto y el sistema devuelve algo casi correcto, sobre la clave de al lado. Al revisar el rendimiento consulta a consulta se ve que estos fallos no se reparten al azar: se concentran en unas pocas consultas difíciles que comparten esa ambigüedad de facetas, mientras la inmensa mayoría se resuelve con un Token F1 cercano a 1.

La segunda forma de error es la omisión, y es casi exclusiva de los modelos débiles. Mercury, sobre todo, tiende a dejarse condiciones enteras cuando la consulta encadena varias. No se equivoca de clave, simplemente no llega a generar el filtro. Es un fallo distinto y más grave que la confusión de faceta, porque reduce de forma silenciosa el conjunto de condiciones que el usuario pidió.

En conjunto, estos límites dibujan bien dónde está hoy la frontera. El sistema domina la parte semántica de alto nivel, entender de qué va la consulta, qué categoría es y qué hay que negar, y donde aún falla es en el último tramo de precisión, en distinguir entre facetas

muy próximas o en no perder condiciones bajo presión. Es un problema de granularidad fina, y eso sugiere que las mejoras futuras pasan más por afinar el contexto de la taxonomía que se inyecta al Entity Agent que por recurrir a modelos más grandes.

7. PLANIFICACIÓN Y PRESUPUESTO

7.1. Planificación Temporal

7.1.1. Metodología de Trabajo

Para el desarrollo de este proyecto se ha seguido una metodología iterativa e incremental, organizada en distintas fases que se han ido solapando cuando ha sido necesario. Este enfoque ha permitido avanzar de forma ordenada, validar decisiones técnicas sobre la marcha y corregir con rapidez aquellos aspectos que requerían ajuste.

El trabajo se ha desarrollado entre el 1 de febrero y el 10 de junio, combinando análisis, diseño, implementación, pruebas y redacción de la memoria. Esta última no se ha reservado únicamente para el final, sino que ha ido avanzando de forma progresiva durante todo el proyecto, de manera paralela al resto de tareas.

Además, se ha mantenido un seguimiento semanal con el CTO de Stylib para revisar avances, resolver dudas y validar las decisiones de desarrollo más relevantes.

7.1.2. Diagrama de Gantt

El proyecto ha comenzado el 1 de febrero y ha finalizado el 10 de junio, con una duración total de 130 días naturales. Su planificación se ha distribuido en cinco fases principales: análisis, diseño, implementación, pruebas y documentación.

Durante febrero se abordó el análisis del problema y la definición inicial de la arquitectura. Entre finales de febrero y marzo se trabajó en el diseño detallado y en las primeras implementaciones. A partir de abril se centró el esfuerzo en la implementación completa del sistema y su integración. Durante mayo se realizaron las pruebas, la validación y el refinamiento de resultados. Paralelamente, la redacción de la memoria comenzó en abril y se mantuvo hasta la entrega final.

De forma resumida, las fases principales del proyecto han sido las siguientes:

- **Análisis y definición del problema:** del 1 de febrero al 18 de febrero.
- **Diseño de la arquitectura del sistema:** del 19 de febrero al 10 de marzo.
- **Implementación de agentes y lógica principal:** del 11 de marzo al 28 de abril.
- **Pruebas, validación y refinamiento:** del 29 de abril al 26 de mayo.
- **Documentación y redacción final:** del 1 de abril al 10 de junio.

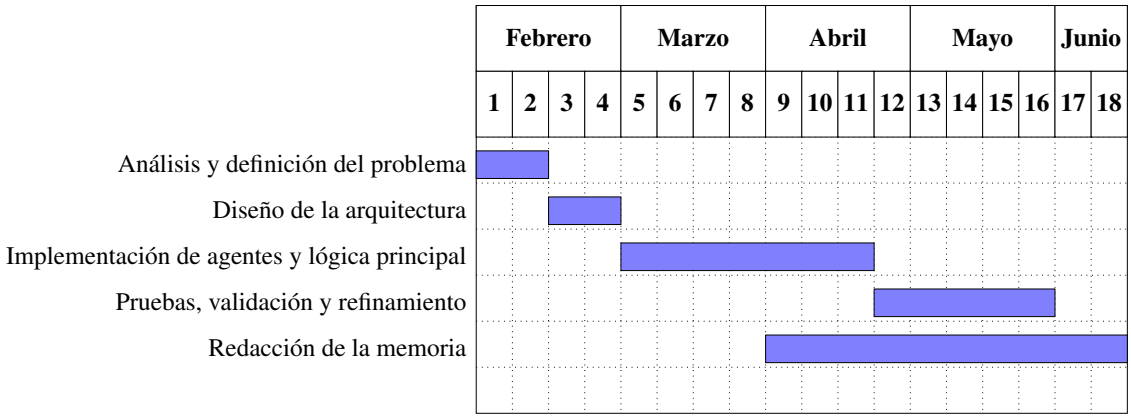


Fig. 7.1. Diagrama de Gantt del proyecto

7.2. Estimación de Costes

Para la estimación económica del proyecto se ha adoptado un enfoque orientado a reflejar el valor real del trabajo realizado y de los recursos empleados. Además, en este caso el desarrollo se ha llevado a cabo dentro de un entorno profesional, al formar parte de la actividad desarrollada en Stylib, una plataforma digital centrada en la búsqueda, gestión y descubrimiento de productos y materiales arquitectónicos. Por ello, el presupuesto no representa únicamente una estimación académica, sino una aproximación al coste real que tendría el desarrollo del sistema en un contexto profesional.

7.2.1. Costes de Recursos Humanos

En cuanto a los recursos humanos, el proyecto ha contado con dos perfiles principales: por un lado, la autora del trabajo, responsable del análisis, diseño, implementación, pruebas y redacción de la memoria; y, por otro, el CTO de Stylib, que ha ejercido como supervisor técnico del proyecto y con el que se han mantenido reuniones semanales de seguimiento para revisar avances, resolver dudas y validar decisiones de desarrollo.

Para estimar el coste del desarrollo, se ha considerado la dedicación real de la autora a lo largo de todo el proyecto. Tomando como referencia el periodo comprendido entre el 1 de febrero y el 10 de junio, se ha supuesto una carga media de 9 horas semanales de trabajo técnico, lo que refleja la combinación de tareas de análisis, implementación, pruebas y redacción llevadas a cabo de forma simultánea. Esto equivale aproximadamente a 18 semanas de dedicación, es decir, un total de 162 horas. Aplicando una tarifa de 25 euros por hora, el coste imputable al trabajo desarrollado asciende a 4.050 euros.

Por otra parte, se ha estimado una dedicación de 1 hora semanal del CTO de Stylib para reuniones de revisión, tutoría y validación de avances. Considerando esas mismas 18 semanas, se obtiene un total de 18 horas. Si se aplica una tarifa de 35 euros por hora, el coste imputable a esta labor asciende a 630 euros.

La estimación total de costes de personal se resume en la Tabla 7.1.

Perfil	**Tarifa**	**Horas**	**Coste total**
Desarrolladora	25,00 €/h	162 h	4.050,00 €
CTO de Stylib	35,00 €/h	18 h	630,00 €
Total personal		**180 h**	**4.680,00 €**

TABLA 7.1. COSTES DE RECURSOS HUMANOS

7.2.2. Costes de Hardware y Software

Desde el punto de vista material, el proyecto no ha requerido la adquisición de hardware específico adicional. El desarrollo se ha realizado utilizando un equipo informático ya disponible, por lo que en este apartado se considera únicamente la amortización parcial del ordenador empleado durante los meses de desarrollo.

Suponiendo un equipo con un valor aproximado de 2200 euros y una vida útil de 4 años, la amortización mensual sería de 45,83 euros, calculada mediante amortización lineal. Considerando algo más de cuatro meses de uso imputables al proyecto, el coste de hardware ascendería aproximadamente a 220 euros.

En cuanto al software, sí ha existido un coste directo asociado al uso de OpenRouter, ya que esta plataforma permite acceder a distintos modelos de lenguaje mediante un sistema de pago por uso basado en tokens. Durante el desarrollo del proyecto se han empleado los modelos *google/gemini-3-flash-preview*, *google/gemini-3.1-flash-lite-preview*, *inception/mercury-2*, *deepseek/deepseek-v3.2* y *moonshotai/kimi-k2.6*. A partir de los tokens consumidos en el conjunto completo de pruebas y de las tarifas oficiales publicadas por OpenRouter, el coste estimado total de este servicio asciende a 100 euros.

La estimación de estos costes se resume en la Tabla 7.2.

Concepto	**Coste imputable**
Amortización de hardware	220,00 €
Uso de OpenRouter	100,00 €
Total hardware y software	**320,00 €**

TABLA 7.2. COSTES DE HARDWARE Y SOFTWARE

7.3. Resumen Total de Costes

A partir de las partidas anteriores, el coste total estimado del proyecto asciende a la suma de los costes de personal, la amortización del hardware y el uso de OpenRouter. Esta cantidad recoge el valor imputable al trabajo técnico realizado, las labores de seguimiento

y supervisión y el uso de los recursos materiales y de software durante el periodo de desarrollo.

La Tabla 7.3 muestra el resumen final del presupuesto.

Partida	**Importe**
Costes de recursos humanos	4.680,00 €
Costes de hardware y software	320,00 €
Coste total estimado	**5.000,00 €**

TABLA 7.3. RESUMEN TOTAL DE COSTES

8. MARCO REGULADOR Y ENTORNO SOCIO-ECONÓMICO

8.1. Marco Regulador

El desarrollo de este proyecto se enmarca en un contexto normativo relacionado principalmente con la protección de datos, el uso responsable de sistemas de inteligencia artificial y la propiedad intelectual del software generado. Aunque la solución propuesta no está orientada al tratamiento masivo de datos personales, sí puede llegar a procesar consultas, trazas de uso o resultados de interacción, por lo que conviene tener presentes las obligaciones derivadas del RGPD (Reglamento General de Protección de Datos) y de la LOPDGDD (Ley Orgánica 3/2018 de Protección de Datos Personales y Garantía de los Derechos Digitales). En este tipo de proyectos, la precaución principal consiste en diseñar el sistema de manera que minimice desde el principio el tratamiento de información sensible, más allá del mero cumplimiento formal de la normativa. [36], [37]

En el ámbito de la inteligencia artificial, el proyecto también debe entenderse dentro del marco regulador europeo, especialmente el AI Act, que establece un enfoque progresivo en función del nivel de riesgo del sistema. En este caso, al tratarse de una herramienta de apoyo a la búsqueda y generación de filtros, el sistema no persigue decisiones automatizadas de alto impacto, pero sí debe desarrollarse con criterios de transparencia, supervisión humana y uso prudente de los datos. Esto resulta especialmente relevante al emplear servicios externos de inferencia. [38], [39]

Durante el desarrollo se ha utilizado OpenRouter como intermediario para acceder a distintos modelos de lenguaje. La documentación de la plataforma indica que no almacena *prompts* ni respuestas salvo que el usuario active explícitamente esa opción, aunque sí conserva metadatos técnicos de cada solicitud, como el número de *tokens* o la latencia. Además, cada proveedor integrado puede aplicar su propia política de retención y tratamiento de datos, por lo que se ha procurado no introducir información personal innecesaria en las pruebas y trabajar siempre con datos controlados o anonimizados. [40], [41], [42], [43]

Desde el punto de vista de la propiedad intelectual, la aportación principal de este TFG reside en el diseño de la arquitectura multiagente, la lógica de extracción y refinamiento y la integración de los distintos componentes del sistema. Las librerías, modelos y servicios externos se han empleado como herramientas de apoyo, respetando sus licencias y condiciones de uso. De este modo, el trabajo combina una base tecnológica existente con una contribución propia claramente identificable.

8.2. Entorno Socio-Económico

El proyecto se ha desarrollado en un entorno profesional y su presupuesto refleja el valor económico estimado del tiempo de trabajo, los recursos materiales empleados y el uso de servicios externos de inteligencia artificial. El coste principal corresponde al esfuerzo humano dedicado al análisis, diseño, implementación, pruebas y documentación del sistema, que se ha prolongado durante varios meses de dedicación parcial siguiendo la planificación descrita en el capítulo de gestión del proyecto. A ello se suma la amortización del equipo informático utilizado y el coste asociado al uso de OpenRouter durante la fase de experimentación y validación, donde se han ejecutado múltiples combinaciones de modelos para analizar su impacto en calidad, latencia y coste.

La utilidad del sistema está en automatizar la generación de filtros a partir de lenguaje natural. En catálogos amplios, esto reduce el tiempo de búsqueda y la carga manual de definir filtros a mano, sobre todo para quienes no conocen bien la estructura del catálogo. Además, al reutilizar el motor de filtrado existente, la plataforma obtiene esta mejora sin rediseñar la interfaz.

La Agenda 2030 plantea la digitalización como uno de los motores del progreso económico y social sostenible [44], [45]. El ODS 9 (industria, innovación e infraestructura), en concreto, promueve el desarrollo de infraestructuras fiables y el impulso de capacidades tecnológicas en los sectores productivos [46], [47]. La herramienta propuesta contribuye a este objetivo, ya que mejora la infraestructura digital de Stylib al aprovechar mejor un catálogo técnico ya existente y reducir la fricción en el acceso a la información.

Un sistema de búsqueda más accesible también facilita el acceso a la información técnica, lo que responde al ODS 16 sobre instituciones eficaces, responsables y transparentes [48]. Aunque el proyecto no se dirige a la administración pública, comparte principios similares: trazabilidad de las decisiones mediante métricas y registros, posibilidad de supervisión humana en todo momento, y menos diferencia entre lo que puede encontrar un usuario experto y uno que no lo es. Que un interiorista ocasional pueda llegar a los mismos resultados de catálogo que un perfil técnico especializado pone el sistema al alcance de quienes no dominan su lenguaje interno.

La mejora en la calidad y rapidez de las búsquedas también puede influir en las decisiones de selección de materiales, un punto relacionado con el ODS 12 (producción y consumo responsables) [49]. Si el catálogo permite encontrar con más facilidad productos con ciertas propiedades o certificaciones técnicas, resulta más fácil priorizar alternativas más eficientes o sostenibles cuando existen. Este TFG no evalúa el impacto ambiental de los materiales, pero un catálogo mejor explotado hace más viable que plataformas como Stylib incorporen en el futuro filtros de sostenibilidad que sean realmente utilizables en la práctica.

Un servicio de generación de filtros es, en sí mismo, un cambio local en la plataforma, pero sus efectos van más allá: reduce costes operativos, mejora la competitividad de Stylib

y acerca la información técnica del catálogo a un público más amplio, en línea con la digitalización sostenible que promueve la Agenda 2030 [50], [51].

9. CONCLUSIONES

9.1. Cumplimiento de los Objetivos

El proyecto partía de un objetivo general y cinco específicos. Conviene revisarlos uno a uno, porque no todos se cumplieron con la misma holgura ni dejaron las mismas lecciones.

El objetivo general era transformar consultas en lenguaje natural en filtros estructurados y semánticamente correctos. Se ha cumplido. El sistema toma una frase como «paneles acústicos negros de menos de 20 mm» y devuelve una cadena de filtros válida para el catálogo de Stylib. Los experimentos lo respaldan con un Token F1 de 0,93 en la mejor configuración, lo que significa que la inmensa mayoría de las consultas se traduce bien.

El primer objetivo específico pedía diseñar un pipeline multiagente con responsabilidades separadas. Aquí no solo se construyó la arquitectura de tres agentes, extracción, mapeo y validación, sino que se demostró que esa separación tiene sentido más allá de lo estético. El Experimento 2 dejó claro que la etapa de extracción es la que fija el techo de calidad, y el Experimento 3 mostró que el validador funciona como una red de seguridad que rescata a los modelos más flojos. Dividir el problema acabó siendo una decisión que los propios datos justificaron.

El segundo objetivo era integrar conocimiento estructurado mediante la taxonomía, y quizá sea donde el sistema mejor responde. La clasificación de categoría fue perfecta en los modelos Gemini, 52 de 52 consultas, y el sistema casi nunca se inventa valores que no estén en el catálogo. Eso es justo lo que se buscaba al inyectar las facetas reales de la base de datos: que la IA se ciña a lo que existe y no proponga claves inventadas.

El tercer objetivo consistía en formalizar la salida. El sistema serializa las condiciones lógicas en grupos de filtros y genera la URL codificada en base64 que el *frontend* consume directamente. Es la parte menos vistosa del trabajo, pero sin ella nada de lo anterior llegaría al usuario. Funciona de principio a fin.

El cuarto objetivo reclamaba un marco de evaluación cuantitativa. Se construyó a mano el *Golden Dataset* de 52 consultas anotadas y se definieron ocho métricas alrededor del Token F1. Este marco fue la herramienta que permitió tomar todas las decisiones posteriores con datos delante en lugar de por intuición, y sin él los experimentos no habrían pasado de ser impresiones.

El quinto objetivo, la experimentación comparativa, es probablemente el que más recorrido dio. Se evaluaron cinco modelos y varias combinaciones híbridas midiendo calidad, latencia y coste. De ahí salió la configuración que finalmente se adoptó: Gemini 3.1 Flash-Lite ofrece casi la misma calidad que los modelos más caros por la mitad de precio,

y cuando interesa apurar la calidad, poner Flash en la extracción y Flash-Lite en el resto da el mejor resultado de todo el estudio. El barrido enseñó además algo que no estaba en el guion: un modelo más grande y caro como Kimi no compra mejores filtros, solo más latencia.

En conjunto, los cinco objetivos específicos se han cumplido, y con ellos el general. Queda margen de mejora, sobre todo en la confusión entre facetas muy parecidas, pero el sistema hace lo que se propuso: entender al usuario y devolverle el filtro correcto.

9.2. Lecciones Aprendidas

A lo largo del proyecto se ha podido profundizar en los siguientes ámbitos:

En primer lugar, separar extracción, mapeo y validación en agentes distintos resultó útil, porque convirtió el sistema en un instrumento de diagnóstico. Cuando algo fallaba, era posible señalar exactamente dónde. Sin esa separación, todos los errores habrían aparecido mezclados en la salida final y habría sido mucho más difícil saber dónde intervenir.

A continuación, los experimentos mostraron que un modelo más grande no siempre produce mejores resultados. Por ejemplo, Kimi, el más potente del estudio, fue también el más lento y el más caro, pero sus filtros no superaban los de Gemini Flash-Lite. Para tareas con esquema fijo, seguir instrucciones con precisión importa más que la potencia bruta.

Finalmente, trabajar sobre un sistema pensado para producción cambia la forma de interpretar los números. Un Token F1 de 0,93 es un buen resultado en el papel, pero detrás del 7 % restante hay usuarios reales que no encuentran lo que buscan. Esa parte de la historia no la cuenta una media agregada sobre un *dataset* fijo.

9.3. Limitaciones

Una posible limitación sería el número de consultas realizadas, que podría afectar a la representatividad del proyecto: es suficiente para discriminar entre configuraciones y tomar decisiones con fundamento, pero pequeño para garantizar que los resultados se mantengan ante consultas que el conjunto de evaluación no contempla. A saber: tecnicismos poco frecuentes, términos en otros idiomas o formulaciones muy atípicas.

En segundo lugar, la confusión entre atributos próximos es el problema más difícil de atacar. Cuando una propiedad encaja en más de una clave de la taxonomía, el agente de mapeo tiende hacia la que aparece con más frecuencia en el contexto del *prompt*, que no siempre es la más adecuada. A medida que la taxonomía crece, el solapamiento entre categorías empeora.

Además, el sistema depende de OpenRouter para acceder a los modelos. Un cambio de tarifa, una interrupción del servicio o la retirada de un modelo afecta directamente al

comportamiento en producción. No hay mucho margen de actuación inmediata.

Asimismo, las métricas presentadas son medias sobre el *Golden Dataset*, no señales de producción. El sistema actual no registra qué consultas reales fallan ni su efecto en el usuario (búsquedas sin resultados, abandono, reformulaciones), así que ese 7 % de error medido en el laboratorio es, por ahora, la única referencia disponible.

Por último, el sistema no aprende. Cada consulta se procesa de forma independiente y las correcciones del usuario no se incorporan a ejecuciones futuras. El rendimiento se queda fijo en el momento del despliegue hasta que alguien revisa los prompts o actualiza la taxonomía a mano.

9.4. Trabajo Futuro

Uno de los caminos más interesantes para continuar este proyecto sería incorporar un cuarto agente encargado de extraer la categoría de la consulta. La idea es, si el sistema consigue identificar antes ese tipo de información, el agente de entidades no necesitaría recibir toda la taxonomía completa, sino únicamente la parte relacionada con esa categoría. Eso permitiría ahorrar una cantidad importante de tokens, algo especialmente valioso cuando se trabaja con modelos de lenguaje, donde cada palabra cuenta más de lo que parece.

Además, el hecho de que el sistema esté construido como un conjunto de piezas independientes abre la puerta a futuras mejoras bastante flexibles. Por ejemplo, sería posible sustituir el agente más costoso por otro más especializado, o incluso por un modelo entrenado específicamente para esta tarea. De esta forma, el sistema podría hacerse más eficiente sin necesidad de rehacer toda la arquitectura desde cero. La modularidad, al final, no solo facilita el mantenimiento, sino que también deja bastante margen para seguir afinando el rendimiento con el tiempo.

El trabajo futuro no pasa solo por añadir cosas nuevas, sino por pulir lo que ya hay. En un sistema de este tipo, pequeñas mejoras en cómo se reparte el trabajo entre agentes se traducen en menos coste, respuestas más rápidas y una arquitectura más limpia.

BIBLIOGRAFÍA

[1] D. Vandic, L. J. Nederstigt, F. Frasincar, U. Kaymak y E. Ido, "A framework for approximate product search using faceted navigation and user preference ranking," *Data & Knowledge Engineering*, vol. 149, pp. 102-241, 2024. DOI: https://doi.org/10.1016/j.datak.2023.102241. [En línea]. Disponible en: https://www.sciencedirect.com/science/article/pii/S0169023X23001015.

[2] D. Tunkelang, *Faceted search*. San Rafael, CA: Morgan & Claypool Publishers, 2009, vol. 5.

[3] J. Gwizdka, "Distribution of cognitive load in Web search," *Journal of the American Society for Information Science and Technology*, vol. 61, n.º 11, pp. 2167-2187, 2010. DOI: https://doi.org/10.1002/asi.21385. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/asi.21385. [En línea]. Disponible en: https://onlinelibrary.wiley.com/doi/abs/10.1002/asi.21385.

[4] C. D. Manning, P. Raghavan y H. Schütze, *Introduction to Information Retrieval*. Cambridge, UK: Cambridge University Press, 2008.

[5] J. Chen, O. Mashkova, F. Zhapa-Camacho, R. Hoehndorf, Y. He e I. Horrocks, "Ontology Embedding: A Survey of Methods, Applications and Resources," *IEEE Transactions on Knowledge and Data Engineering*, vol. 37, n.º 7, pp. 4193-4212, 2025. DOI: 10.1109/TKDE.2025.3559023.

[6] N. Guarino, D. Oberle y S. Staab, "What Is an Ontology?" En *Handbook on Ontologies*, S. Staab y R. Studer, eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1-17. DOI: 10.1007/978-3-540-92673-3_0. [En línea]. Disponible en: https://doi.org/10.1007/978-3-540-92673-3_0.

[7] Y. Bengio, R. Ducharme, P. Vincent y C. Jauvin, "A Neural Probabilistic Language Model," *Journal of Machine Learning Research*, vol. 3, pp. 1137-1155, 2003.

[8] J. Devlin, M.-W. Chang, K. Lee y K. Toutanova, "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, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran y T. Solorio, eds., Minneapolis, Minnesota: Association for Computational Linguistics, jun. de 2019, pp. 4171-4186. DOI: 10.18653/v1/N19-1423. [En línea]. Disponible en: https://aclanthology.org/N19-1423/.

[9] W. X. Zhao, J. Liu, R. Ren y J.-R. Wen, "Dense Text Retrieval Based on Pretrained Language Models: A Survey," *ACM Trans. Inf. Syst.*, vol. 42, n.º 4, feb. de 2024. DOI: 10.1145/3637870. [En línea]. Disponible en: https://doi.org/10.1145/3637870.

[10] Typedef.ai, *Embeddings for Semantic Search*, https://www.typedef.ai/resources/embeddings-semantic-search-statistics, Accedido: 02 de febrero de 2026, 2025.

[11] Meilisearch, *Understanding Hybrid Search RAG for Better AI Answers*, https://www.meilisearch.com/blog/hybrid-search-rag, Accedido: 02 de febrero de 2026, 2026.

[12] A. Vaswani et al., "Attention is All you Need," en *Advances in Neural Information Processing Systems*, I. Guyon et al., eds., vol. 30, Curran Associates, Inc., 2017, pp. 5998-6008. [En línea]. Disponible en: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[13] W. X. Zhao et al., "A Survey of Large Language Models," eng, *Frontiers of Computer Science*, vol. 20, n.º 12, pp. 2 012 627-, 2026.

[14] Q. Dong et al., "A Survey on In-context Learning," en *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Y. Al-Onaizan, M. Bansal e Y.-N. Chen, eds., Miami, Florida, USA: Association for Computational Linguistics, nov. de 2024, pp. 1107-1128. DOI: 10.18653/v1/2024.emnlp-main.64. [En línea]. Disponible en: https://aclanthology.org/2024.emnlp-main.64/.

[15] X. Liu et al., "A Survey of Text-to-SQL in the Era of LLMs: Where Are We, and Where Are We Going?" *IEEE Transactions on Knowledge and Data Engineering*, vol. 37, n.º 10, pp. 5735-5754, 2025. DOI: 10.1109/TKDE.2025.3592032.

[16] W. Mu, J. Ning, D. Zhao e Y. Zhang, "A Multi-Agent LLM Framework for Multi-Domain Low-Resource In-Context NER via Knowledge Retrieval, Disambiguation and Reflective Analysis," en *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 40, 2026, pp. 32 528-32 536.

[17] P. Arnoldsson y A. Karimi, *A Multi-Agent LLM Pipeline for Complex Report Generation from Multiple Sources with Human-in-the-Loop Feedback*, Västerås, Sweden, 2025. [En línea]. Disponible en: https://www.diva-portal.org/smash/get/diva2:1973948/FULLTEXT01.pdf.

[18] J. Wu, J. Han e Y. Gao, "TabAgent: A Multi-Agent Table Extraction Framework for Unstructured Documents," en *2025 5th International Symposium on Artificial Intelligence and Big Data (AIBDF)*, 2025, pp. 600-607. DOI: 10.1109/AIBDF67964.2025.11440749.

[19] C.-Y. Chang et al., "MAIN-RAG: Multi-Agent Filtering Retrieval-Augmented Generation," en *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, W. Che, J. Nabende, E. Shutova y M. T. Pilehvar, eds., Vienna, Austria: Association for Computational Linguistics, jul. de 2025, pp. 2607-2622. DOI: 10.18653/v1/2025.acl-long.131. [En línea]. Disponible en: https://aclanthology.org/2025.acl-long.131/.

[20] Y. Lu, W. Wu, X. Zhao, R. Peng y J. Wang, "KARMA: Leveraging Multi-Agent LLMs for Automated Knowledge Graph Enrichment," en *Advances in Neural Information Processing Systems*, D. Belgrave et al., eds., vol. 38, Curran Associates, Inc., 2025, pp. 56 421-56 453. [En línea]. Disponible en: https://proceedings.neurips.cc/paper_files/paper/2025/file/517f9b9c227b9dd51dba4560f37165ed-Paper-Conference.pdf.

[21] J. White et al., "A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT," en *Proceedings of the 28th European Conference on Pattern Languages of Programs (EuroPLoP)*, Hillside Europe, 2023.

[22] J. Wei et al., "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," en *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho y A. Oh, eds., vol. 35, Curran Associates, Inc., 2022, pp. 24 824-24 837. [En línea]. Disponible en: https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b
Paper-Conference.pdf.

[23] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan y S. Yao, "Reflexion: language agents with verbal reinforcement learning," en *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt y S. Levine, eds., vol. 36, Curran Associates, Inc., 2023, pp. 8634-8652. [En línea]. Disponible en: https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

[24] J. Li, A. Sun, J. Han y C. Li, "A Survey on Deep Learning for Named Entity Recognition," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, n.º 1, pp. 50-70, 2022. DOI: 10.1109/TKDE.2020.2981314.

[25] X. Cheng, M. Bowden, B. Bhange, P. Goyal, T. Packer y F. Javed, "An End-to-End Solution for Named Entity Recognition in eCommerce Search," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, pp. 15 098-15 106, mayo de 2021. DOI: 10.1609/aaai.v35i17.17773.

[26] Python Software Foundation, *Python 3.13 Documentation*, https://docs.python.org/3/, Accedido: 05 de marzo de 2026, 2025.

[27] S. Ramírez, *FastAPI*, https://fastapi.tiangolo.com, Accedido: 05 de marzo de 2026, 2026.

[28] Pydantic Team, *Models — Pydantic Documentation*, https://pydantic.dev/docs/validation/latest/concepts/models/, Accedido: 05 de marzo de 2026, 2024.

[29] Pydantic Team, *Pydantic AI: GenAI Agent Framework, the Pydantic Way*, https://pydantic.ai, Accedido: 05 de marzo de 2026, 2024.

[30] OpenRouter, *OpenRouter API Reference*, https://openrouter.ai/docs/api/reference/overview, Accedido: 07 de marzo de 2026, 2026.

[31] React Team, *Describing the UI — React Documentation*, https://react.dev/learn/describing-the-ui, Accedido: 07 de marzo de 2026, 2024.

[32] B. Wang, R. Shin, X. Liu, O. Polozov y M. Richardson, "RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers," en *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, D. Jurafsky, J. Chai, N. Schluter y J. Tetreault, eds., Online: Association for Computational Linguistics, jul. de 2020, pp. 7567-7578. DOI: 10.18653/v1/2020.acl-main.677. [En línea]. Disponible en: https://aclanthology.org/2020.acl-main.677/.

[33] T. Scholak, N. Schucher y D. Bahdanau, "PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models," en *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, M.-F. Moens, X. Huang, L. Specia y S. W.-t. Yih, eds., Online y Punta Cana, Dominican Republic: Association for Computational Linguistics, nov. de 2021, pp. 9895-9901. DOI: 10.18653/v1/2021.emnlp-main.779. [En línea]. Disponible en: https://aclanthology.org/2021.emnlp-main.779/.

[34] W. Fan et al., "A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models," en *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, ép. KDD '24, New York, NY, USA: Association for Computing Machinery, 2024, pp. 6491-6501. DOI: 10.1145/3637528.3671470.

[35] M. J. Ingeno, *Software Architect's Handbook: Become a Successful Software Architect by Implementing Effective Architecture Concepts*. Birmingham: Packt Publishing, 2018.

[36] European Union, *Regulation (EU) 2016/679 of the European Parliament and of the Council*, https://eur-lex.europa.eu/eli/reg/2016/679/oj, Accedido: 12 de mayo de 2026, abr. de 2016.

[37] España, *Ley Orgánica 3/2018, de Protección de Datos Personales y garantía de los derechos digitales*, https://www.boe.es/eli/es/lo/2018/12/05/3, Accedido: 12 de mayo de 2026, dic. de 2018.

[38] European Union, *Article 50: Transparency Obligations for Providers and Deployers of Certain AI Systems*, https://ai-act-service-desk.ec.europa.eu/en/ai-act/article-50, Accedido: 04 de abril de 2026, 2026.

[39] European Commission, *Draft of the guidelines on the implementation of the transparency obligations for certain AI systems under Article 50 of the AI Act*, https://digital-strategy.ec.europa.eu/en/library/draft-guidelines-implementation-transparency-obligations-certain-ai-systems-under-article-50-ai-act, Accedido: 12 de mayo de 2026, mayo de 2026.

[40] OpenRouter, *Data Collection | OpenRouter Privacy*, https://openrouter.ai/docs/guides/privacy/data-collection, Accedido: 04 de abril de 2026, mayo de 2026.

[41] OpenRouter, *Privacy Policy - OpenRouter*, https://openrouter.ai/privacy, Accedido: 04 de abril de 2026, abr. de 2025.

[42] OpenRouter, *Provider Logging - Provider Data Retention Policies - OpenRouter*, https://openrouter.ai/docs/guides/privacy/provider-logging, Accedido: 04 de abril de 2026, mayo de 2026.

[43] OpenRouter, *Zero Data Retention | How OpenRouter gives you control over your data*, https://openrouter.ai/docs/guides/features/zdr, Accedido: 04 de abril de 2026, mayo de 2026.

[44] Naciones Unidas, *Transformar nuestro mundo: la Agenda 2030 para el Desarrollo Sostenible*, https://unctad.org/system/files/official-document/ares70d1_es.pdf, Accedido: 01 de junio de 2026, 2015.

[45] Programa de las Naciones Unidas para el Desarrollo y Unión Internacional de Telecomunicaciones, *La tecnología digital contribuye directamente a la consecución de los Objetivos de Desarrollo Sostenible*, https://www.undp.org/es/comunicados-de-prensa/la-tecnologia-digital-contribuye-al-70-de-las-metas-de-los-ods-segun-la-uit-el-pnud, Accedido: 01 de junio de 2026, 2023.

[46] Naciones Unidas, *Objetivo 9: Industria, innovación e infraestructura*, https://www.undp.org/es/sustainable-development-goals/industria-innovacion-infraestructura, Accedido: 01 de junio de 2026, 2015.

[47] Ministerio para la Transición Ecológica y el Reto Demográfico, *ODS 9. Industria, innovación e infraestructura. Indicadores nacionales de la Agenda 2030*, https://www.miteco.gob.es/es/ministerio/servicios/estadisticas/miteco---indicadores-de-la-agenda-2030-para-desarrollo-sostenibl/ods-9-industria-innovacion-e-infraestructura.html, Accedido: 01 de junio de 2026, 2025.

[48] Naciones Unidas, *Objetivo 16: Paz, justicia e instituciones sólidas*, https://www.un.org/sustainabledevelopment/es/peace-justice/, Accedido: 01 de junio de 2026, 2015.

[49] Naciones Unidas, *Objetivo 12: Producción y consumo responsables*, https://www.un.org/sustainabledevelopment/es/sustainable-consumption-production/, Accedido: 01 de junio de 2026, 2015.

[50] Asociación para el Autocuidado de la Salud (Anefp), *Digitalización y sostenibilidad, un binomio imprescindible para las empresas*, https://anefp.org/es/notas-de-prensa/digitalizacion-y-sostenibilidad-un-binomio-imprescindible-para-las-empresas, Accedido: 01 de junio de 2026, 2024.

[51] Pacto Mundial Red España, *El poder de la digitalización sostenible*, https://www.pactomundial.org/noticia/digitalizacion-una-poderosa-herramienta-para-la-sostenibilidad/, Accedido: 01 de junio de 2026, 2025.

ANEXO A: DECLARACIÓN DE USO DE IA GENERATIVA

He usado IA Generativa en este trabajo

Marca lo que corresponda: ■ Sí □ No

Parte 1: declaración sobre comportamiento legal, ético y responsable

1. **En mi interacción con herramientas de IA Generativa he remitido datos de carácter sensible con la debida autorización de los interesados:**

   □ Sí, he usado estos datos con autorización

   □ No, he usado estos datos sin autorización

   ■ No, no he usado datos de carácter sensible

2. **En mi interacción con herramientas de IA Generativa he remitido materiales protegidos por derechos de autor con la debida autorización de los interesados:**

   □ Sí, he usado estos materiales con autorización

   □ No, he usado estos materiales sin autorización

   ■ No, no he usado materiales protegidos

3. **En mi interacción con herramientas de IA Generativa he remitido datos de carácter personal con la debida autorización de los interesados:**

   □ Sí, he usado estos datos con autorización

   □ No, he usado estos datos sin autorización

   ■ No, no he usado datos de carácter personal

Uso respetuoso y ético de la herramienta:

■ Sí □ No

Parte 2: Declaración de uso técnico

Declaro haber hecho uso del sistema de IA Generativa (ChatGPT, Claude, Perplexity) para:

Documentación y redacción
Se han empleado asistentes conversacionales para elevar el nivel formal de la memoria académica. Tras redactar los borradores iniciales con la justificación técnica de la arquitectura multiagente, la IA se utilizó para homogeneizar el vocabulario técnico, asegurar el uso correcto de la tercera persona pasiva y estructurar las explicaciones sobre la inyección de dependencias y el modelado del dominio con Pydantic.

Revisión y reescritura de párrafos
Las herramientas de IA actuaron como revisores de estilo, ayudando a corregir saltos bruscos de persona gramatical, mejorar la fluidez entre secciones (transiciones de párrafos) y pulir la redacción de conceptos abstractos (como la explicación del *Schema Linking* o el funcionamiento del *Retry Loop*), garantizando un tono unificado, objetivo y profesional en todo el documento.

Búsqueda de información
Se utilizaron motores impulsados por IA (como Perplexity) para la recuperación eficiente de literatura científica reciente, facilitando la localización de artículos fundamentales sobre metodologías de *prompt engineering* (CoT, Reflexion, In-Context Learning) e interfaces NLIDB. Esta asistencia permitió cruzar las decisiones de diseño propias con el estado del arte validado por la academia.

Desarrollo de contenido específico
Durante la fase de implementación, la IA sirvió como asistente de apoyo para formalizar la sintaxis de los *system prompts* de los agentes. Partiendo de las reglas lógicas y restricciones estrictas definidas por el autor, la herramienta ayudó a refactorizar los *prompts* siguiendo estándares académicos documentados (como el uso de cabeceras de rol, descomposición de tareas y ejemplos *few-shot*).

Generación de esquemas, imágenes, audios o vídeos
Se empleó asistencia generativa para la generación del código fuente en lenguaje LaTeX necesario para estructurar el documento (índices, bibliografía en formato BibTeX y formatos de tablas algorítmicas).

Procesos de optimización
En el ámbito del desarrollo de *software*, la IA fue consultada como herramienta de *pair programming* para discutir enfoques de refactorización. Ayudó a validar la eficiencia de

trasladar el paso de diccionarios pesados al *RunContext* de PydanticAI, optimizando el consumo de *tokens* en las llamadas asíncronas del *pipeline* central.

Tratamiento de datos
No se ha delegado en la IA ningún procesamiento crítico o confidencial de datos corporativos. Únicamente se utilizó para generar pequeños *mocks* de datos en formato JSON (datos sintéticos) que sirvieron para probar los esquemas de validación de Pydantic y el comportamiento de las APIs en entornos locales.

Inspiración de ideas
La IA funcionó como un tablero de debate (*brainstorming*) para estructurar el índice de la memoria. Se discutió con el asistente la jerarquía más lógica para presentar la información, decidiendo separar claramente el diseño conceptual (Capítulo 4) de la implementación técnica pura del código en Python (Capítulo 5).

Parte 3: Reflexión sobre utilidad

El uso ético y transparente de la Inteligencia Artificial Generativa ha supuesto, sin duda, un valor añadido muy importante para el desarrollo de este Trabajo de Fin de Grado. A nivel técnico, actuar como una especie de interlocutor me ha permitido poner a prueba decisiones arquitectónicas como la separación del sistema en tres agentes o la incorporación del bucle de autocuración, y gracias a ese proceso he podido llegar a un código asíncrono más sólido, más limpio y bastante más modular. A nivel documental, la IA ha reducido bastante la fricción que suele aparecer al maquetar en LaTeX y al gestionar las citas bibliográficas, lo que me ha dejado más margen para centrarme en lo verdaderamente importante, que es resolver el problema de ingeniería y sostener con base teórica el modelo de extracción semántica. En definitiva, la herramienta ha funcionado como un apoyo que multiplica la productividad, pero sin quitar en ningún momento protagonismo a la autoría, al diseño algorítmico ni al criterio, que han seguido estando bajo mi responsabilidad absoluta.