Autor

Pablo Sierra Fernández

Dirigido por

Pedro García López

DATAWHISPER: ARQUITECTURA BASADA EN AGENTES PARA *TEXT-TO-SQL*

TRABAJO FIN DE GRADO

DOBLE GRADO EN BIOTECNOLOGÍA E INGENIERÍA INFORMÁTICA

UNIVERSITAT ROVIRA I VIRGILI

Tarragona, 2024

Resumen

Este proyecto tiene como objetivo desarrollar un sistema que traduce consultas en lenguaje natural a SQL utilizando una arquitectura basada en agentes. La motivación principal es mejorar la precisión y la eficiencia en la generación de consultas SQL a partir de consultas en lenguaje natural. La metodología empleada incluye el diseño de una arquitectura compuesta por varios agentes inteligentes y especializados, que colaboran para identificar y generar las consultas SQL adecuadas.
Los resultados mostraron que el sistema alcanza una precisión del 95% en la generación de consultas. La implementación final demostró ser efectiva, con una interfaz de usuario accesible a través de una aplicación web. El código fuente y los componentes del proyecto están disponibles en un repositorio público de GitHub para facilitar su replicación y mejora continua.

Resum

Aquest projecte té com a objectiu desenvolupar un sistema que tradueixi consultes en llenguatge natural a SQL utilitzant una arquitectura basada en agents. La motivació principal és millorar la precisió i l'eficiència en la generació de consultes SQL a partir de consultes en llenguatge natural. La metodologia emprada inclou el disseny d'una arquitectura composta per diversos agents intel·ligents i especialitzats, que col·laboren per identificar i generar les consultes SQL adequades. Els resultats van mostrar que el sistema aconsegueix una precisió del 95% en la generació de consultes. La implementació final es va mostrar eficaç, amb una interfície d'usuari accessible a través d'una aplicació web. El codi font i els components del projecte estan disponibles en un repositori públic de GitHub per facilitar la seva replicació i millora contínua.

Abstract

This project aims to develop a system that translates natural language queries into SQL using an agent-based architecture. The main motivation is to improve the accuracy and efficiency of SQL query generation from natural language queries. The methodology employed includes designing an architecture composed of several intelligent and specialized agents that collaborate to identify and generate the appropriate SQL queries. Results showed that the system achieves a 95% accuracy in query generation. The final implementation proved to be effective, with a user interface accessible through a web application. The source code and project components are available in a public GitHub repository to facilitate replication and continuous improvement.

Índice

Índice de tablas

Índice de figuras

1 Introducción

1.1 Contexto

La gestión de bases de datos ha sido un área central de la informática desde la aparición de los sistemas de gestión de bases de datos (DBMS) en las décadas de 1970 y 1980. Estos sistemas fueron desarrollados para organizar, almacenar, recuperar y manipular grandes volúmenes de datos, proporcionando una estructura eficiente para manejar información en multitud de sectores, como la banca y la medicina. Sin embargo, a medida que la cantidad y complejidad de los datos han crecido exponencialmente, también lo han hecho los desafíos asociados con su gestión **[1]**.

Históricamente, la gestión de bases de datos se ha basado en enfoques relacionales, donde el *Structured Query Language* (SQL) se ha consolidado como el estándar para la consulta y manipulación de datos. Aunque SQL ha demostrado ser muy versátil, su uso requiere un conocimiento tanto del lenguaje como de la estructura interna de la base de datos. Esto ha limitado su accesibilidad a expertos en bases de datos y desarrolladores, excluyendo a usuarios menos técnicos que también necesitan acceder a la información.

En los últimos años, la inteligencia artificial (IA) ha evolucionado considerablemente, extendiéndose a la gestión de bases de datos. Tradicionalmente, bases de datos relacionales como PostgreSQL han sido administradas por expertos en bases de datos que requieren un conocimiento tanto del sistema de gestión como de la estructura de datos específica. Sin embargo, con la llegada de la IA, especialmente de modelos de lenguaje natural como GPT-3.5 y GPT-4, la automatización y optimización de tareas complejas relacionadas con bases de datos más asequibles.

La integración de la IA en la gestión de bases de datos permite automatizar tareas que antes eran manuales y propensas a errores, como la generación de consultas SQL complejas, la optimización de índices y la predicción de patrones de acceso a los datos. Además, los modelos de IA pueden aprender de interacciones anteriores para mejorar continuamente la precisión y relevancia de sus acciones, lo que reduce la carga de trabajo de los administradores de bases de datos y el riesgo de errores humanos **[2]**.
Un aspecto importante en la gestión de bases de datos asistida por IA es la capacidad de interpretar y procesar consultas en lenguaje natural, traduciéndose en sentencias SQL, facilitando la interacción para usuarios no técnicos. Modelos como GPT-4 son capaces de evaluar la relevancia de una solicitud en lenguaje natural y determinar cómo convertirla en una consulta SQL, teniendo en cuenta factores como la estructura de las tablas **[3]**.

1.2 La Nueva Inteligencia Artificial

El aprendizaje automático, y en particular las redes neuronales artificiales, comenzaron a ganar fama durante los años 1980 y 1990. Aunque las primeras redes neuronales, como el perceptrón, tenían capacidades limitadas, investigaciones posteriores llevaron al desarrollo de

redes más profundas y complejas. Un avance clave fue el algoritmo de retropropagación, que permitió entrenar redes neuronales multicapa de manera más efectiva, mejorando su rendimiento en tareas de clasificación y reconocimiento.

A medida que la capacidad computacional y la disponibilidad de datos crecieron en las décadas siguientes, el aprendizaje automático se consolidó como el enfoque dominante en la IA. Los algoritmos de *Machine Learning* (ML) comenzaron a superar a los modelos anteriores en una variedad de tareas, desde el reconocimiento de voz hasta la visión por computadora. Sin embargo, a pesar de estos éxitos, muchos algoritmos de aprendizaje automático seguían siendo difíciles de interpretar y requerían grandes cantidades de datos etiquetados, lo que limitaba su aplicabilidad en contextos más amplios.

1.3 Deep Learning

El cambio más radical en la inteligencia artificial vino con la llegada del *deep learning*, un subcampo del aprendizaje automático que se enfoca en el uso de redes neuronales profundas para modelar datos complejos y de gran dimensión. A partir de la década de 2010, el *deep learning* comenzó a dominar la investigación en IA, impulsado por el aumento de la capacidad de procesamiento gráfico (GPUs) y la disponibilidad de grandes volúmenes de datos. Modelos como ImageNet, demostraron que las redes neuronales profundas podían superar a los humanos en tareas de clasificación de imágenes, lo que marcó el inicio de la revolución del *deep learning* **[4]**.

El *deep learning* mejoró el rendimiento en tareas tradicionales de IA y abrió nuevas posibilidades en áreas como la generación de lenguaje natural, la síntesis de imágenes y la predicción de secuencias. Modelos de redes neuronales recurrentes (RNNs) y redes de memoria a largo corto plazo (LSTM) permitieron avances en la traducción automática, el reconocimiento de voz y el análisis de series temporales. Al mismo tiempo, las redes neuronales convolucionales (CNNs) revolucionaron la visión por computadora, permitiendo a las máquinas identificar objetos y patrones con una precisión sin precedentes.

Un avance clave en este período fue el desarrollo de la arquitectura de transformadores, introducida por Vaswani et al. en 2017. Los transformadores permitieron un procesamiento paralelo más eficiente de secuencias de datos, lo que llevó a mejoras en la generación de lenguaje natural y la comprensión del contexto. Esta arquitectura es la base de los modelos de lenguaje modernos, como GPT-3 y GPT-4, que han demostrado altas capacidades en la generación de texto, la respuesta a preguntas y la traducción automática **[5]**.

1.4 De Modelos Generativos a Sistemas Autónomos

El impacto de estos modelos ha sido muy grande en áreas como el procesamiento de lenguaje natural (NLP), donde han transformado aplicaciones que van desde *chatbots* hasta asistentes virtuales y herramientas de escritura automatizada. Además, los modelos generativos han encontrado aplicaciones en la creación de contenido multimedia, la generación de imágenes y la síntesis de voz, ampliando el alcance de la IA a nuevas áreas creativas y artísticas.

Más allá de los modelos generativos, "la nueva IA" también se caracteriza por la integración de capacidades de aprendizaje profundo con otras disciplinas, como la robótica, la visión por computadora y el razonamiento automatizado. Esto ha llevado al desarrollo de sistemas autónomos que pueden operar en entornos complejos y dinámicos, como vehículos autónomos, robots industriales y sistemas de gestión de infraestructuras. Estos sistemas realizan tareas predefinidas, pueden adaptarse a nuevas situaciones, aprender de sus errores y colaborar con otros sistemas y humanos **[6]**.

Un área emergente en esta nueva era de la IA es el aprendizaje por refuerzo, una técnica que permite a los sistemas aprender a tomar decisiones mediante la interacción con su entorno y la maximización de recompensas a largo plazo. Este enfoque ha sido utilizado en aplicaciones tan diversas como el control de robots, la optimización de redes y el desarrollo de estrategias de juego en entornos complejos. El aprendizaje por refuerzo ha demostrado ser eficaz en escenarios donde las soluciones óptimas no son evidentes y requieren exploración y adaptación continua **[7]**.

1.5 Estado del Arte: Técnicas *Text-to-SQL*

En los últimos años, el campo del procesamiento NLP ha experimentado avances notables, impulsados principalmente por el desarrollo de modelos de lenguaje de gran tamaño (LLM) como GPT-3 y GPT-4, que han demostrado una gran capacidad para comprender, procesar y sintetizar texto en diversos contextos. Uno de los desafíos emergentes en este ámbito es la generación automática de consultas SQL a partir de descripciones en lenguaje natural, una tarea conocida como *text-to-SQL*. Esta técnica promete cambiar la interacción con bases de datos, facilitando que usuarios sin conocimientos avanzados en SQL puedan extraer y manipular información de manera eficiente **[8]**.

En este contexto, se han desarrollado múltiples técnicas y modelos para automatizar y optimizar este proceso, con el objetivo de democratizar el acceso a los datos y mejorar la productividad en su manejo. Este apartado se centra en explorar las técnicas más avanzadas en el ámbito del *text-to-SQL*, actualmente.

1.5.1 Modelos y Técnicas Actuales

El desarrollo de técnicas para la conversión de texto en consultas SQL ha estado marcado por una evolución continua, en la que se han empleado diversos enfoques para mejorar la precisión y la eficiencia de las consultas generadas. A continuación, se describen las técnicas y modelos más destacados en este campo.

- **Modelos Basados en Plantillas (*Template-Based Models*)**

  Los modelos basados en plantillas fueron una de las primeras aproximaciones en la tarea de *text-to-SQL*. Estos modelos funcionan mapeando oraciones en lenguaje natural a plantillas SQL predefinidas, que luego se rellenan con los valores correspondientes. Aunque este enfoque es efectivo para consultas simples, su aplicabilidad es limitada en consultas más complejas o en bases de datos con múltiples tablas y relaciones.

Además, la dependencia de la disponibilidad y calidad de las plantillas restringe la capacidad de estos modelos para generalizar a nuevos dominios o estructuras de datos.

- **Modelos de Traducción Neural (*Neural Translation Models*)**

  Con la aparición de los modelos neuronales de traducción, inspirados en los avances en traducción automática, surgió una nueva generación de técnicas para la generación de SQL. Estos modelos utilizan redes neuronales, particularmente arquitecturas de codificador-decodificador (*encoder-decoder*), para traducir una oración en lenguaje natural a una consulta SQL. El enfoque de *Sequence-to-Sequence* (*Seq2Seq*) fue pionero en este ámbito, utilizando un codificador para transformar la entrada de texto en un vector de contexto, y un decodificador para generar la consulta SQL correspondiente.
  El enfoque *Seq2Seq* fue rápidamente superado por la introducción de arquitecturas de atención (*attention mechanisms*), que permitieron a los modelos centrarse en partes específicas de la entrada durante la generación de la consulta. Este avance mejoró la capacidad de los modelos para manejar consultas más complejas y aumentar la precisión en la generación de SQL **[8]**.

- **Modelos Preentrenados (*Pretrained Language Models*)**

  Estos modelos, entrenados en grandes volúmenes de datos de texto, han demostrado ser muy efectivos para diversas tareas de NLP, incluida la generación de SQL. El preentrenamiento permite a los modelos captar una comprensión profunda del lenguaje, que luego puede ser ajustada (*fine-tuning*) para tareas específicas como *text-to-SQL*.
  Modelos como T5, que sigue un enfoque de transformación de texto a texto, han mostrado un buen rendimiento en *benchmarks* estándar como Spider y WikiSQL. Estos modelos se entrenan utilizando grandes conjuntos de datos que contienen pares de preguntas en lenguaje natural y sus correspondientes consultas SQL, lo que les permite aprender a mapear el lenguaje natural a SQL de manera más precisa **[9]**.

- **Modelos de Codificación Semántica y Recuperación (*Semantic Parsing and Retrieval Models*)**
  Estos modelos se centran en comprender el significado subyacente de una oración en lenguaje natural y mapearlo directamente a la estructura semántica de una consulta SQL. A diferencia de los enfoques de traducción directa, los modelos de codificación semántica descomponen el problema en subcomponentes más manejables, como la identificación de entidades, la clasificación de intenciones y la recuperación de fragmentos SQL relevantes.
  Un ejemplo destacado de este enfoque es la técnica de *retrieval-augmented generation* (RAG), que combina la generación de texto con la recuperación de información relevante de bases de datos o documentos preexistentes. Esta técnica ha demostrado ser útil en escenarios donde el conocimiento externo es necesario para generar consultas SQL precisas y contextualmente relevantes **[10]**.

- **Arquitecturas Híbridas (*Hybrid Architectures*)**
  Las arquitecturas híbridas combinan múltiples enfoques y técnicas para aprovechar las fortalezas de cada uno. Por ejemplo, un modelo híbrido podría utilizar un modelo preentrenado para comprender el lenguaje natural junto con un modelo de codificación semántica para mapear esa comprensión a una consulta SQL precisa permitiendo abordar mejor la complejidad de las consultas SQL.
  Un ejemplo de arquitectura híbrida es el *framework* SQL-Sidekick, que integra múltiples modelos de última generación, tanto de código abierto como propietarios, para generar consultas SQL. Este *framework* utiliza técnicas de *prompt engineering*, *in-context learning*, y *self-correction loops* para mejorar la precisión y adaptabilidad del modelo en entornos del mundo real **[11]**.

1.5.2 Modelos de Última Generación (State-of-the-Art)

Actualmente, varios modelos de última generación han logrado avances en la tarea de *text-to-SQL*. A continuación, se presentan algunos de los modelos más avanzados en este campo.

- **SQLCoder-7B-2**

  SQLCoder-7B-2 es un modelo de código abierto, derivado del modelo base StarCoder, que ha sido ajustado específicamente para la tarea de *text-to-SQL*. Los *benchmarks* publicados por los autores del modelo muestran una precisión superior al 85% en conjuntos de datos complejos como Spider **[12]**.

- **NSQL-Llama-2-7B**

  El modelo NSQL-Llama-2-7B, basado en la arquitectura Llama-2 de Meta, es otro modelo de última generación que ha sido entrenado y ajustado para la generación de SQL. El enfoque de *few-shot learning* empleado por NSQL-Llama-2-7B le permite generalizar mejor a nuevos dominios y datos no vistos previamente **[13]**.

- **GPT-3.5 y GPT-4**

  Los modelos GPT-3.5 y GPT-4 de OpenAI también han demostrado ser efectivos en la tarea de *text-to-SQL*. Además, la capacidad de GPT-4 para seguir instrucciones detalladas y adaptarse a diferentes contextos le ha permitido superar a muchos otros modelos en *benchmarks* como BIRD y Spider **[14]**.

1.5.3 Benchmarks Estándar para Evaluación

La evaluación de los modelos *text-to-SQL* se lleva a cabo utilizando una serie de benchmarks estándar que permiten medir la precisión, eficiencia y adaptabilidad de los modelos en diferentes escenarios. Los tres *benchmarks* más utilizados en la actualidad son WikiSQL,

Spider y BIRD, cada uno con características específicas que los hacen adecuados para evaluar diferentes aspectos de los modelos.

- **WikiSQL**

  WikiSQL fue uno de los primeros conjuntos de datos a gran escala diseñados específicamente para la tarea de *text-to-SQL*. Introducido por Salesforce en 2017, este *benchmark* se compone de preguntas en lenguaje natural emparejadas con consultas SQL simples, que involucran sólo cláusulas SELECT, FROM y WHERE. Aunque WikiSQL es útil para entrenar y evaluar modelos en consultas simples, su limitación radica en la falta de complejidad y en la ausencia de relaciones entre tablas, lo que restringe su aplicabilidad a escenarios del mundo real **[8]**.

- **Spider**

  Spider, desarrollado por la Universidad de Yale, aborda las limitaciones de WikiSQL al introducir consultas SQL más complejas y bases de datos con múltiples tablas y relaciones. Este *benchmark* se distingue por su enfoque en la generación de SQL a través de dominios, evaluando la capacidad de los modelos para generalizar a nuevas bases de datos que no han visto durante el entrenamiento. Spider es actualmente considerado uno de los *benchmarks* más representativos para la evaluación de modelos *text-to-SQL* **[15]**.

- **BIRD**

  BIRD es un *benchmark* más reciente que se enfoca en la evaluación de la generación de SQL en entornos empresariales. Este conjunto de datos incluye consultas SQL más complejas que requieren una comprensión profunda del contexto empresarial y del dominio de los datos **[16]**.

1.6 Sistemas Multi Agente

El campo de los sistemas multiagente (SMA) ha experimentado un crecimiento exponencial en los últimos años, en paralelo con los avances en IA y el desarrollo de técnicas de procesamiento de lenguaje natural (PLN) como las discutidas en el apartado anterior sobre las técnicas de *text-to-SQL*. Aunque los SMA y las técnicas de *text-to-SQL* pueden parecer disciplinas separadas, ambas comparten el objetivo común de mejorar la interacción y la comunicación entre sistemas autónomos y entre humanos y máquinas. Este vínculo resalta la importancia de los SMA en la evolución de la inteligencia artificial y su capacidad para realizar tareas complejas y distribuidas en entornos dinámicos.
Los SMA son sistemas compuestos por múltiples agentes autónomos que interactúan entre sí y con su entorno para lograr objetivos individuales o colectivos. Estos agentes son entidades que poseen un grado de autonomía y la capacidad de tomar decisiones basadas en la información que reciben y procesan. En muchos casos, los SMA se utilizan en aplicaciones que requieren

coordinación, colaboración, y toma de decisiones en tiempo real, como la robótica, la gestión de tráfico, la simulación de mercados financieros, y la optimización de redes de distribución **[17]**.

Un sistema multiagente se define por la presencia de múltiples agentes que pueden ser homogéneos o heterogéneos en su diseño y capacidades. Estos agentes pueden ser físicos, como robots en un entorno de fabricación, o virtuales, como programas de software que gestionan transacciones en un mercado financiero.
A diferencia de los sistemas centralizados, donde un solo agente controla todas las decisiones, los SMA permiten la distribución de la toma de decisiones entre múltiples agentes, lo que aumenta la escalabilidad y la robustez del sistema. Además, los SMA son intrínsecamente adaptativos, lo que les permite ajustar sus estrategias en respuesta a cambios en el entorno o en los objetivos de otros agentes .
Los agentes pueden intercambiar información, negociar y tomar decisiones conjuntas para lograr un objetivo común. Esta capacidad de comunicación se facilita mediante protocolos de comunicación predefinidos y lenguajes de agentes que permiten la interoperabilidad entre diferentes agentes dentro del sistema **[17]**.

1.6.1 Arquitecturas

Existen varias arquitecturas para diseñar y construir sistemas multiagente, cada una con sus propias ventajas y desventajas **[17]**. Las arquitecturas más comunes son:

1. Arquitectura de Control Centralizado: En este enfoque, un agente centralizado tiene la autoridad para coordinar las acciones de todos los demás agentes. Aunque este enfoque puede ser eficiente en ciertos escenarios, es menos robusto frente a fallos en el agente central y puede no escalar bien a sistemas con muchos agentes.

2. Arquitectura Distribuida: En una arquitectura distribuida, no existe un único agente de control, y la toma de decisiones se reparte entre todos los agentes. Este enfoque es más robusto y escalable, pero puede ser más complejo de implementar, especialmente en términos de coordinación y comunicación entre agentes.

3. Arquitectura Híbrida: Combina aspectos de las arquitecturas centralizadas y distribuidas. En una arquitectura híbrida, algunos agentes pueden tener roles de coordinación mientras que otros operan de manera más autónoma.

1.6.2 Técnicas de Coordinación

La coordinación es un aspecto importante en los sistemas multiagente. Existen diversas técnicas desarrolladas para facilitar la coordinación entre agentes **[17]**:

1. Algoritmos de Subasta: En este tipo de algoritmos, los agentes compiten por recursos limitados a través de un proceso de subasta. Cada agente presenta una oferta por un recurso basado en su valor estimado, y el recurso se asigna al agente con la mejor oferta. Este enfoque es común en aplicaciones de asignación de tareas y recursos .

2. Negociación Basada en Reglas: Los agentes negocian entre sí utilizando un conjunto de reglas predefinidas. Estas reglas determinan cómo se lleva a cabo la negociación, qué concesiones están dispuestos a hacer los agentes, y cómo se resuelven los conflictos.

3. Planificación Multiagente: En la planificación multiagente, los agentes colaboran para desarrollar un plan conjunto que permita alcanzar los objetivos del sistema. La planificación puede ser centralizada, con un único agente responsable de la planificación, o distribuida, con los agentes colaborando para generar un plan.

2 Problemática

La gestión y el análisis de datos se han convertido en elementos indispensables para la toma de decisiones en una amplia gama de sectores, desde empresas hasta instituciones académicas y gubernamentales. Uno de los retos a los que se enfrentan los usuarios sin conocimientos técnicos avanzados, sobre todo los que carecen de ellos o tienen conocimientos limitados de SQL, es la interfaz con las bases de datos relacionales. Aunque existen tecnologías que pretenden simplificar este proceso, como los sistemas de conversión de lenguaje natural a SQL o de traducción de lenguaje natural a SQL, a menudo estas soluciones se quedan cortas en cuanto a precisión, escalabilidad, eficacia o accesibilidad **[18]**.

Limitaciones en Técnicas de los Métodos Actuales

Los métodos tradicionales de traducción de lenguajes naturales a SQL presentan varias limitaciones que impiden su adopción. En particular, la precisión sigue siendo un problema a la hora de crear consultas utilizando lenguajes naturales; a menudo, estos sistemas no consiguen interpretar correctamente la intención del usuario, lo que se traduce en imprecisiones en los resultados de las consultas.
Además, estos sistemas carecen de un factor importante: la escalabilidad. Los métodos actuales no suelen abordar la gestión de grandes volúmenes de datos o bases de datos complejas con estructuras complejas, lo que restringe su aplicación en entornos empresariales en los que el volumen de datos y la complejidad de las consultas pueden ser un factor a considerar. La falta de escalabilidad también impide que estos sistemas se adapten a usuarios con necesidades y demandas diferentes, lo que limita su adaptabilidad en situaciones reales.

Ineficiencia en la Gestión de Recursos Computacionales

Otro problema importante es la gestión de los recursos. Los modelos lingüísticos utilizados para la conversión de lenguaje natural a SQL, como los modelos de redes neuronales profundas, requieren una potencia informática significativa para su procesamiento; esto puede dar lugar a elevados costes operativos cuando los sistemas se utilizan de forma intensiva o en entornos con recursos restringidos.

Accesibilidad Limitada para Usuarios no Técnicos

Uno de los principales obstáculos a la implantación de sistemas basados en datos para usuarios no técnicos es su inaccesibilidad. Muchos sistemas de traducción de lenguajes naturales a SQL siguen siendo difíciles de usar para quienes no tienen conocimientos de informática o bases de datos, lo que restringe su accesibilidad para que los utilicen personas no expertas. Según el informe de McKinsey & Company (2018), la falta de accesibilidad a las herramientas analíticas se identificó como uno de los principales obstáculos para que las pequeñas y medianas organizaciones adopten tecnologías basadas en datos para adoptarlas en sus organizaciones **[18]**.

Complejidades en la Interacción con Bases de Datos

El manejo de bases de datos relacionales -especialmente en entornos corporativos y científicos- suele presentar muchas dificultades como constar de múltiples tablas vinculadas con estructuras y restricciones individuales. Crear consultas precisas obliga a conocer a fondo estas relaciones y la lógica subyacente de la base de datos.

Los problemas relacionados con la traducción de lenguajes naturales a SQL presentan una serie de retos técnicos y operativos. Desde la precisión, la escalabilidad y la ineficacia en la gestión de recursos hasta la limitada accesibilidad para usuarios no técnicos, estos problemas motivan la urgencia de crear sistemas más avanzados, eficientes y accesibles. Este proyecto propone objetivos destinados a aportar soluciones que abarquen tanto los aspectos técnicos como los operativos.

3 Objetivos

La problemática descrita anteriormente muestra la necesidad de sistemas que aborden las limitaciones técnicas de los métodos tradicionales y actuales de conversión de lenguaje natural a SQL, así mismo que también mejoren la eficiencia, accesibilidad y flexibilidad en la gestión y análisis de datos. A continuación, se presentan los objetivos generales y específicos que buscan ofrecer una solución a estos desafíos.

Transformación de la Interacción con Bases de Datos

El objetivo general de este proyecto es cambiar la forma en que los usuarios interactúan con bases de datos relacionales, facilitando la generación, evaluación y ejecución de consultas SQL. Esto implica el desarrollo de un sistema que pueda interpretar lenguaje natural y convertirlo en consultas SQL precisas, optimizadas y ejecutables, de manera que se reduzcan las barreras técnicas y se incremente la accesibilidad para usuarios no técnicos.

Implementación de Agentes Inteligentes y Optimización de la Arquitectura

Un objetivo específico en la implantación de una arquitectura de agentes inteligentes es la implementación de un proceso de generación y ejecución de consultas SQL por pasos, en etapas manejables. Cada agente debe estar diseñado para tareas específicas como la identificación de tablas relevantes, la creación de consultas SQL y su ejecución; de forma que se garantice que cada paso se realiza con precisión.
La optimización de la arquitectura también es importante, ya que permitirá la integración de agentes o tecnologías emergentes, garantizando que el sistema pueda adaptarse a los cambios en las necesidades de los usuarios o en las características de la base de datos a lo largo del tiempo.

Evaluación de Costos y Eficiencia

Otro objetivo específico es la evaluación y optimización de los costos asociados a la operación del sistema, particularmente en relación con el uso de modelos de lenguaje avanzados y otros recursos computacionales. Dado que el procesamiento de consultas en lenguaje natural puede ser intensivo en recursos, es importante desarrollar un sistema que pueda monitorear y gestionar estos costos de manera efectiva.

Desarrollo de una Aplicación Interactiva

Un tercer objetivo específico es el desarrollo de una aplicación interactiva y amigable para el usuario, que permita a los usuarios interactuar con el sistema de manera intuitiva. Esta aplicación debe ofrecer una interfaz gráfica de usuario (GUI) que facilite la introducción de

prompts en lenguaje natural, la visualización de los resultados de las consultas SQL, y la generación de *insights* adicionales a partir de los datos consultados.
La creación de esta aplicación mejorará la experiencia del usuario permitiendo un acceso más sencillo a los datos.

Publicación en un Repositorio de Código Abierto

Finalmente, un objetivo específico adicional es la publicación del sistema en un repositorio de código abierto, donde pueda ser accedido y utilizado libremente por la comunidad. La liberación del código permitirá que otros desarrolladores y equipos de investigación utilicen el sistema y contribuyan a su mejora.

4 Metodología

Este trabajo se enfoca principalmente en establecer un entorno robusto y seguro para la implementación de un sistema que permita la generación, evaluación y ejecución automatizada de consultas SQL en una base de datos cualquiera de tipo relacional. Inspirado en la arquitectura creada por el usuario BenGWeeks **[19]**, aunque con nuevas implementaciones y mejorada, este sistema estará potenciado por agentes inteligentes que se encargan de diferentes tareas dentro del flujo de trabajo para finalmente obtener la respuesta de una consulta en lenguaje humano y finalmente servirlo a través de una API en el *back* y mostrarlo en un *front* reactivo. A continuación, se describe detalladamente cada uno de los procedimientos.

4.1 Configuración del Entorno

Es necesario configurar un entorno de trabajo que permita la correcta implementación y ejecución del código. La configuración incluye la conexión segura a la base de datos PostgreSQL en AWS RDS **[20]**, la autenticación con la API de OpenAI para el uso de los modelos GPT y la creación de un entorno virtual (venv) que garantice la independencia de las dependencias y la facilidad de gestión del entorno de desarrollo.

4.1.1 Entorno Virtual (venv)

El uso de un entorno virtual (venv) es una práctica recomendada en el desarrollo de proyectos en Python **[21]**, ya que permite aislar las dependencias del proyecto y evitar conflictos con otros paquetes instalados en el sistema. Un entorno virtual asegura que todas las bibliotecas y módulos necesarios para el proyecto estén gestionados y actualizados sin interferir con otros proyectos.

Para crear y activar un entorno virtual en Python, se pueden utilizar los siguientes comandos:

```
# Crear un entorno virtual
python -m venv venv

# Activar el entorno virtual
source venv/bin/activate   # En Linux/MacOS
venv\Scripts\activate       # En Windows
```

Código 1: Creación de entorno virtual.

Posteriormente se instalan algunas librerías necesarias, como *dotenv* **[22]** para la gestión de variables de entorno, *tiktoken* **[23]** para el manejo de *tokens* y *autogen* **[24]** para la orquestación de agentes, entre otras.

4.1.2 Conexión a PostgreSQL en AWS RDS[1]

Uno de los primeros pasos en la configuración del entorno es establecer una conexión segura a la base de datos PostgreSQL alojada en AWS RDS. PostgreSQL es un sistema de gestión de bases de datos relacional de código abierto, conocido por su estabilidad y escalabilidad. Primero deberemos ir a la página oficial de Amazon AWS **[25]** y registrarnos. Una vez registrados debemos configurar el servicio de AWS RDS haciendo uso del *Free Tier* para obtener el servicio gratuito. AWS RDS facilita la administración de la base de datos al dar servicios como copias de seguridad automáticas, actualizaciones de software, escalabilidad y nos permite tener alojada la base de datos en la nube.

El código utilizado para conectarse a PostgreSQL en AWS RDS se basa en la obtención de las credenciales y configuraciones necesarias desde un archivo `.env`, el cual debe contener las variables de entorno con la información de conexión, tales como el nombre del *host*, el nombre de la base de datos, el nombre de usuario, la contraseña y el puerto de conexión. Esto sirve para que las credenciales (sensibles) no se incluyan directamente en el código fuente, para mejorar la seguridad del sistema.

```
import dotenv
import os
dotenv.load_dotenv(override=True)
# Obtener las variables de entorno
hostname = os.getenv('hostname')
database = os.getenv('database')
username = os.getenv('username')
password = os.getenv('password')
port = os.getenv('port')
```

Código 2: Carga de variables de entorno desde un archivo .env y obtención de credenciales de configuración.

La conexión a la base de datos es necesaria ya que el sistema interactúa directamente con PostgreSQL para obtener tablas relevantes, generar consultas SQL y ejecutar dichas consultas. Esta conexión se gestiona a través de la clase `PostgresManager`, la cual encapsula toda la lógica necesaria para interactuar con la base de datos.

4.1.3 Autenticación con OpenAI

Para obtener las credenciales a la API de OpenAI **[26]** primero deberemos registrarnos y pagar una pequeña cuota para poder hacer uso. Los modelos de lenguaje como GPT-3.5 y GPT-4, utilizados en este proyecto, requieren de una clave API para poder ser accedidos y utilizados. Al igual que con las credenciales de la base de datos, esta clave API se gestiona a través de variables de entorno para mantener la seguridad.

[1] Aunque se ha utilizado AWS RDS con PostgreSQL para esta implementación, es posible conectar el sistema a cualquier base de datos relacional compatible.

```
OPENAI_API_KEY = os.environ.get("OPENAI_API_KEY")
```

Código 2: Obtención de la clave de la API de OpenAI desde las variables de entorno del sistema.

La variable `OPENAI_API_KEY` se extrae del entorno y se utiliza para autenticar las solicitudes a la API de OpenAI. Esta clave permite que el sistema acceda a los modelos de lenguaje y ejecute las tareas que se le asignan.

Es importante destacar que la autenticación con OpenAI no solo se trata de acceder a los modelos de lenguaje, sino también de gestionar los costos asociados con su uso (tal y como se muestra en la **Figura 1**) y optimizar el uso de los recursos, lo cual, también se explicará más adelante.

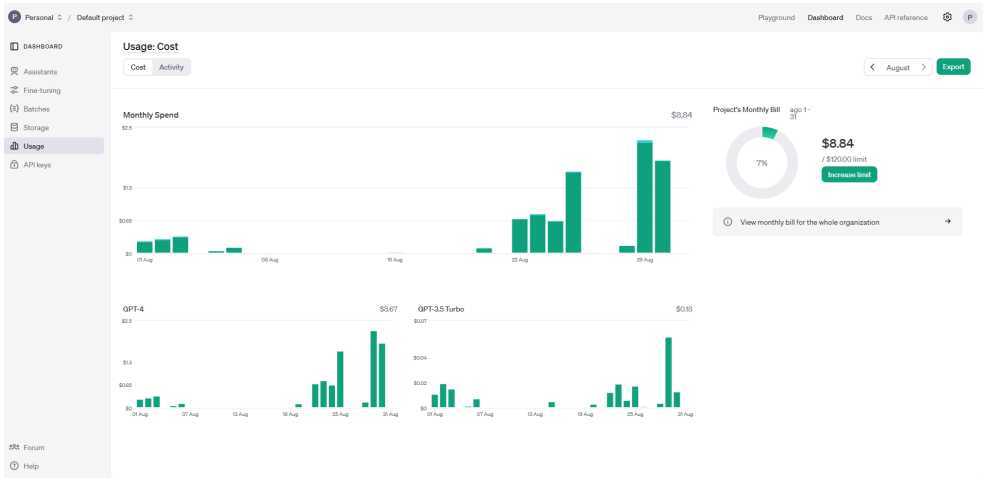

Figura 1. Panel de Control de costes de la API de OpenAI.

4.2 Agente Proxy (Evaluador de NLQ)

El Agente Proxy es un módulo autónomo dentro del sistema que recibe los *prompts* en lenguaje natural y evalúa su relevancia para la generación de consultas SQL. Su objetivo es evitar que se generen consultas SQL a partir de *prompts* irrelevantes o ambiguos, lo que podría llevar a un uso ineficiente de los recursos del sistema. Este agente clasifica los *prompts* en una escala del 1 al 5, donde 1 indica irrelevancia total y 5 máxima relevancia.
Podemos ver su contexto en la arquitectura en la **Figura 2**.

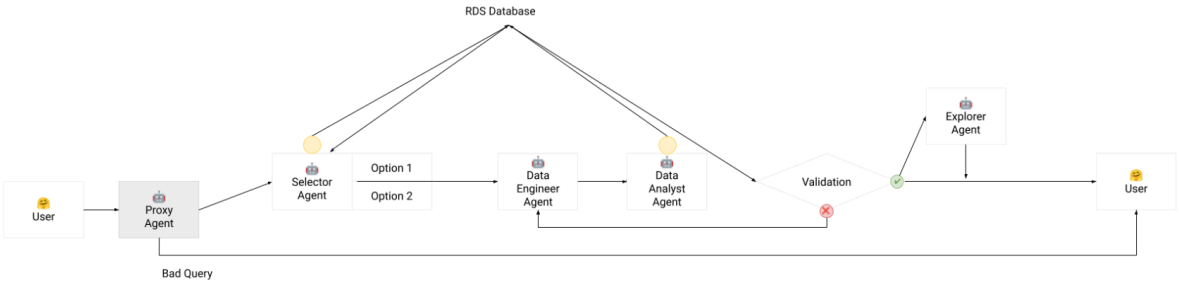

Figura 2. Agente Proxy en contexto a la arquitectura.

4.2.1 Proceso

El núcleo del Agente Proxy es su capacidad para interpretar y clasificar los *prompts* de manera precisa. Para lograr esto, se configura un agente utilizando el modelo `gpt-3.5-turbo` de OpenAI.

```
system_message = """You are an expert in classifying Natural Language
Queries (NLQ) for SQL.
Your task is to determine if a given input is relevant to an SQL query.
Rank the input from 1 to 5, where:
1: Definitely not NLQ
2: Likely not NLQ
3: Neutral / Unsure
4: Likely NLQ
5: Definitely NLQ
Respond with only the number representing your ranking."""
```

Código 4: Definición de un mensaje del sistema para clasificar consultas en lenguaje natural según su relevancia para una consulta SQL, utilizando una escala de 1 a 5.

Este mensaje de sistema guía al modelo para que clasifique los *prompts* basándose en su relevancia para las consultas SQL. El agente utiliza este mensaje como un marco de referencia para emitir un juicio estructurado sobre la relevancia de cada *prompt*.

La configuración del agente es la siguiente:

```
sql_relevance_agent = autogen.AssistantAgent(
    name="SQL_Relevance_Agent",
    llm_config={
        "temperature": 0,
        "config_list": [
            {
```

```
                "model": "gpt-3.5-turbo",
                "api_key": OPENAI_API_KEY
            }
        ]
    },
    system_message=system_message,
    code_execution_config=False,
    human_input_mode="NEVER"
)
```

Código 5: Creación de un agente de asistencia llamado *SQL_Relevance_Agent*, configurado para evaluar la relevancia de consultas en lenguaje natural para SQL. Utiliza el modelo *gpt-3.5-turbo* con un nivel de temperatura fijo en 0, y está programado para no ejecutar código ni requerir entrada humana.

Este agente está diseñado para ser determinístico, lo que significa que su clasificación no varía entre ejecuciones con el mismo *prompt*, y así obtener una mayor consistencia.
Una vez configurado, el Agente Proxy evalúa cada *prompt* a través de una función dedicada. Esta función envía el *prompt* al agente, recibe la clasificación numérica y la devuelve al sistema principal.

```
def check_relevance(prompt):
    sql_relevance_agent.send(prompt, sql_relevance_agent)
    reply = sql_relevance_agent.generate_reply(sender=sql_relevance_agent)
    print("REPLY: ",  reply)
    try:
        return int(reply.strip().replace("\n", "").replace("\t",
"").replace(" ", ""))
    except Exception as e:
        print(e)
        return -1
```

Código 6: Función *check_relevance* que envía un *prompt* al agente *sql_relevance_agent*, genera una respuesta de relevancia para SQL, y devuelve el resultado como un entero; retorna *-1* si ocurre un error.

Esta función asegura que solo los *prompts* que superen un umbral de relevancia sean considerados para la generación de consultas SQL. Si el sistema recibe una puntuación baja (por ejemplo, 1, 2 o 3), se evita el procesamiento adicional, lo que ahorra recursos y tiempo. El valor de la relevancia obtenido es entonces utilizado para tomar una decisión informada sobre si proceder o no con la generación de una consulta SQL. Esto se logra a través de un condicional que evalúa si el *prompt* es lo suficientemente relevante.

```
relevance_score = check_relevance(prompt)
if relevance_score == -1:
    print("Error: NLQ Classifier did not return a valid integer.")
    return 0
if relevance_score > 3:
    # Proceder con la generación de la consulta SQL
```

Código 7: Evaluación del *relevance_score* devuelto por la función *check_relevance*. Si el valor es *-1*, se imprime un mensaje de error y se retorna *0*. Si el valor es mayor a *3*, se procede con la generación de la consulta SQL.

4.3 Agente Selector (Selección de Tablas)

La precisión en la selección de tablas reduce el coste computacional y mejora la eficiencia, ya que minimiza la ventana de contexto enviada y acorta los tiempos de procesamiento. En este trabajo, se implementan dos enfoques distintos para la selección de tablas. Podemos ver su contexto en la arquitectura en la **Figura 3**.

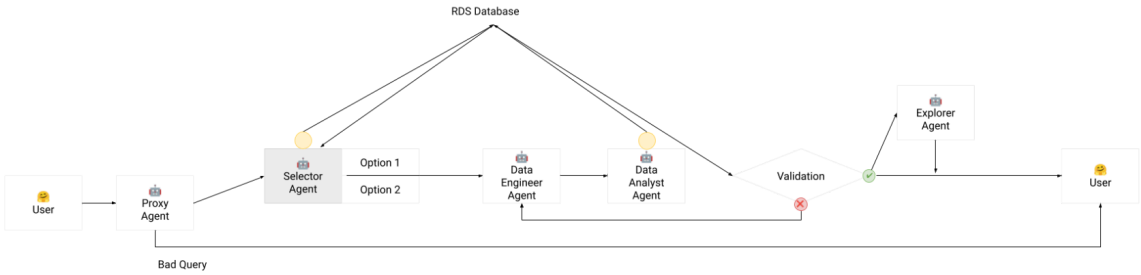

Figura 3. Agente Selector en contexto a la arquitectura.

4.3.1 Opción 1

El primer enfoque consiste en utilizar un agente selector basado en un modelo de lenguaje (en este caso, GPT-3.5). Este agente es responsable de identificar las tablas más relevantes para una consulta SQL, basándose en el contenido de la misma.

El sistema se configura con una serie de reglas claras para guiar al agente en la selección de tablas. Estas reglas son las siguientes:

```
 table_names = "\n".join(db.get_all_table_names())
   system = f"""You are a database expert. Your task is to identify the MOST
RELEVANT tables for a given SQL query.
   Available tables are:
   {table_names}

   Rules:
   1. Only select tables that are DIRECTLY relevant to answering the query.
   2. Do not include tables that are merely related but not necessary for the
specific query.
   3. If the query can be answered with a single table, only return that table.
   4. Return table names as a comma-separated list, with no additional text or
explanations.
```

```
    5. If no tables are relevant or you're unsure, return an empty list.
    """

    human = "User query: {input}\nRelevant tables:"

    prompt = ChatPromptTemplate.from_messages([
        ("system", system),
        ("human", human),
    ])
```

Código 8: Configuración de un sistema de prompt para identificar las tablas más relevantes para una consulta SQL. Se proporciona una lista de tablas disponibles y se establecen reglas para seleccionar únicamente las tablas relevantes para la consulta. Se utiliza *ChatPromptTemplate* para crear el prompt, que incluye el mensaje del sistema y el mensaje del usuario.

La lógica del agente selector se ejecuta mediante una serie de pasos:

1. Inicialización del modelo de lenguaje con la configuración adecuada.
2. Creación de la plantilla de mensajes que sigue las reglas establecidas anteriormente.
3. Procesamiento de la consulta del usuario para identificar las tablas relevantes.

```
 llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0,
api_key=OPENAI_API_KEY)
    output_parser = CommaSeparatedListOutputParser()

    chain = prompt | llm | output_parser

    result = chain.invoke({"input": user_query})
```

Código 9: Procesa la consulta del usuario para identificar tablas relevantes usando *gpt-3.5-turbo* y devuelve una lista de tablas separadas por comas.

El agente devuelve un listado de las tablas relevantes junto con el conteo de los *tokens* utilizados, lo cual permite una estimación del costo asociado al uso del agente.

4.3.2 Opción 2

El segundo enfoque prescinde del uso de un agente basado en un modelo de lenguaje, utilizando en su lugar una aproximación semántica y la técnica de cercanía por coseno para identificar las tablas relevantes. Este método es útil para reducir los costes asociados al empleo de modelos de lenguaje.

En un primer paso, se obtienen todos los nombres de las tablas junto con sus esquemas. Estos datos se procesan para crear descripciones comprensibles para un modelo de incrustación de oraciones.

```
table_names = db.get_all_table_names()
```

```
    table_schemas = {}
    for table in table_names:
        try:
            table_schemas[table] = get_table_schema(db, table)
        except Exception as e:
            print(f"Error getting schema for table {table}: {e}")
            table_schemas[table] = "Schema unavailable"

    table_descriptions = [f"Table: {table}, Schema: {schema}" for table,
schema in table_schemas.items()]
```

Código 10: Obtiene los nombres de las tablas de la base de datos y sus esquemas. Si ocurre un error al obtener el esquema de una tabla, se registra un mensaje de error y se marca el esquema como "Schema unavailable". Luego, genera descripciones de las tablas con su esquema correspondiente.

Se utiliza una librería de similaridad de alta eficiencia para crear un índice basado en las representaciones vectoriales de las descripciones de las tablas. A continuación se detalla el proceso.

```
model = SentenceTransformer('all-MiniLM-L6-v2')
    table_embeddings = model.encode(table_descriptions)
    dimension = table_embeddings.shape[1]
    index = faiss.IndexFlatL2(dimension)
    index.add(table_embeddings.astype('float32'))
```

Código 11: Carga un modelo de *SentenceTransformer* para generar embeddings de descripciones de tablas. Luego, crea un índice de búsqueda *faiss* usando estos *embeddings* para facilitar la búsqueda de similitudes entre tablas.

Se comienza cargando el modelo `all-MiniLM-L6-v2` **[27]** de `SentenceTransformer` **[28]**, un modelo de aprendizaje profundo basado en transformadores que convierte texto, como frases u oraciones, en vectores numéricos conocidos como *embedding*, para lo cual se generan para crear una lista de descripciones de tablas (`table_descriptions`) utilizando este modelo, lo que produce una matriz en la que cada fila es el vector que representa una descripción de tabla. Luego, se determina la dimensión de estos *embeddings* y se crea un índice de búsqueda usando FAISS **[29]**, una biblioteca de Facebook diseñada para la búsqueda eficiente de similitudes entre vectores. El índice se basa en la distancia L2 (distancia euclidiana), y los *embeddings* convertidos a `float32` se agregan a este índice, permitiendo búsquedas rápidas de descripciones similares en el futuro.

La consulta del usuario, por lo tanto, se convierte a una representación vectorial y se utiliza para buscar las tablas más relevantes en el índice.

```
query_embedding = model.encode([user_query])
    distances, indices = index.search(query_embedding.astype('float32'), k)
    relevant_tables = [table_names[i] for i in indices[0]]
```

Código 12: Codifica la consulta del usuario en un embedding, realiza una búsqueda en el índice *faiss* para encontrar las tablas más relevantes basadas en la similitud, y obtiene los nombres de las tablas correspondientes.

A diferencia de la primera opción, al usar un modelo *in-situ* y no usar un agente GPT el costo de la operación es de 0 *token*s.

Ambos enfoques tienen sus propias ventajas y desventajas. El enfoque basado en un agente selector proporciona resultados más exactos al estar respaldado por un modelo de lenguaje avanzado, aunque a un mayor costo. Por otro lado, la aproximación semántica es más económica y rápida, aunque puede no ser tan precisa en todos los casos.

4.4 Agentes Analistas (Generación y Ejecución)

Este apartado se centra en los métodos y procesos utilizados para generar y ejecutar estas consultas mediante la colaboración entre un equipo de agentes inteligentes.

4.4.1 Estructura del Equipo

El sistema se estructura a través de agentes especializados que colaboran para llevar a cabo la tarea de generación y ejecución de consultas SQL. Cada agente tiene un rol definido y se coordina a través de un orquestador que asegura un flujo de trabajo eficiente.

El equipo de agentes en el sistema incluyen:

- **User Proxy Agent:** Este agente actúa como intermediario humano virtual, asegurando que toda interacción y plan de acción sea aprobado antes de continuar.

- **Data Engineer Agent (SQL Coder):** Este agente se encarga de la generación inicial de la consulta SQL basada en los requisitos proporcionados. Su función es estructurar la consulta sin cometer errores básicos, como el uso incorrecto de palabras reservadas de SQL o la asignación de alias entre otros.

- **Data Analyst Agent:** Este agente se encarga de ejecutar la consulta SQL generada para lo cual se le asigna una función específica de ejecución directa de SQL.

Podemos ver su contexto en la arquitectura en la **Figura 4**.

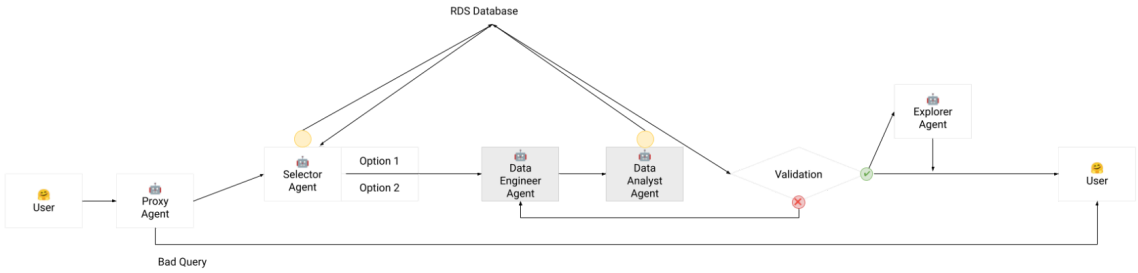

Figura 4. Equipo de analistas en contexto a la arquitectura.

```
user_proxy = autogen.UserProxyAgent(
    name="Admin",
    system_message="A human admin. Interact with the Product Manager to
discuss the plan. Plan execution needs to be approved by this admin.",
    code_execution_config=False,
    human_input_mode="NEVER"
)

data_engineer = autogen.AssistantAgent(
    name="Engineer",
    llm_config=base_config,
    system_message="A Data Engineer. You follow an approved plan. Generate
the initial SQL based on the requirements provided. Do not use SQL reserved
words as aliases. put the name of the tables and columns in double quotes.
If there's an error in the SQL execution, analyze the error message and
provide a corrected SQL query.",
    code_execution_config=False,
    human_input_mode="NEVER"
)

sr_data_analyst = autogen.AssistantAgent(
    name="Sr_Data_Analyst",
    llm_config=gpt4_config_with_functions,
    system_message="Sr Data Analyst. You follow an approved plan. You run
the SQL query. Only run SQL and not show the results",
    human_input_mode="NEVER",
    function_map=function_map,
)
```

Código 13: Define tres agentes automáticos para gestionar diferentes roles en el proceso de consulta SQL.

Estos agentes trabajan en conjunto bajo la supervisión del `Orchestrator`, el componente que coordina las interacciones entre los agentes y asegura que cada uno cumpla su rol de manera ordenada.

4.4.2 Orquestación y Secuencialidad

El orquestador es el componente que garantiza que el flujo de trabajo entre los agentes sea secuencial y ordenado. Este se encarga de pasar los mensajes y resultados de un agente a otro hasta que la tarea esté completa.

```
data_eng_orchestrator = Orchestrator(
    name="Postgres Data Analytics Multi-Agent ::: Data Engineering Team",
    agents=[user_proxy, data_engineer, sr_data_analyst],
    tables_def=table_definitions
)
```

Código 14: Crea un orquestador para coordinar un equipo de agentes.

```
messages = data_eng_orchestrator.sequential_conversation(prompt)
```

Código 15: Ejecuta una conversación secuencial con el orquestador de datos, enviando el `prompt` para coordinar las respuestas de los agentes involucrados.

Durante este proceso, se registra cada interacción y mensaje entre los agentes para llevar un control detallado de las operaciones, permitiendo revisiones posteriores.

```
def register_message(self, from_name, to_name, message):
    log_entry = {
        "from_name": from_name,
        "to_name": to_name,
        "message": str(message)
    }
    with open("conversation_log.json", "a") as log_file:
        json.dump(log_entry, log_file)
        log_file.write("\n")
```

Código 16: Registra un mensaje en un archivo de registro JSON, incluyendo el nombre del remitente, el nombre del destinatario y el mensaje. Cada entrada se agrega como una línea en el archivo *conversation_log.json*.

El `Orchestrator` supervisa el proceso de principio a fin, desde la generación de la consulta hasta la ejecución y la generación de *insights* adicionales.

4.4.3 Añadir Otros Modelos a los Agentes

Para dar una mayor flexibilidad al sistema, se incluye la capacidad de integrar otros modelos de lenguaje en el agente Data Engineer (o SQL Coder). Lo ideal y necesario es que este modelo sea diseñado específicamente para la generación de consultas SQL.

El modelo se integra en el sistema a través de la API de HuggingFace.

```
def generate_sql_with_sqlcoder_api(prompt, table_definitions):
    API_URL = f"https://api-inference.huggingface.co/models/{model_name}"
    headers = {"Authorization": f"Bearer
{os.environ.get('HUGGINGFACE_API_KEY')}"}
    payload = {
        "inputs": full_prompt,
        "parameters": {"max_new_tokens": 250, "temperature": 0.1,
"do_sample": True}
    }
    response = requests.post(API_URL, headers=headers, json=payload)
    result = response.json()
    if isinstance(result, list) and len(result) > 0:
        sql_query = result[0].get('generated_text', '').strip()
        return sql_query
    else:
        raise Exception("Unexpected API response format")
```

Código 17: Genera una consulta SQL utilizando la API de HuggingFace hacia el modelo seleccionado. Envía un *prompt* y *table_definitions* al endpoint de la API y recibe una consulta SQL generada en respuesta. Si la respuesta es válida, devuelve la consulta SQL; de lo contrario, lanza una excepción.

Para facilitar la integración del modelo se personaliza un agente específico denominado SQLEngineerAgent. Este agente interactúa con la API de HuggingFace para obtener las consultas SQL generadas y pasar esta información al orquestador.

```
class SQLEngineerAgent(autogen.AssistantAgent):
    def __init__(self, name, system_message, table_definitions):
        super().__init__(name=name, system_message=system_message,
llm_config=False)
        self.table_definitions = table_definitions

    def generate_reply(self, messages=None, sender=None, config=None):
        last_message = messages[-1]['content'] if messages else ""
        try:
            sql_query = generate_sql_with_sqlcoder_api(last_message,
self.table_definitions)
            return f"Here's the SQL query generated:\n\n

```sql\n{sql_query}\n

```"
        except Exception as e:
            return f"An error occurred while generating the SQL query:
{str(e)}"
```

Código 18: Define *SQLEngineerAgent*, una clase para generar consultas SQL a partir de un modelo alojado en HuggingFace. Usa el último mensaje del historial para crear una consulta SQL y devuelve el resultado o un mensaje de error.

Así pues se permite una mayor flexibilidad a la hora de elegir el agente SQL Coder, el cual será el encargado de generar la consulta. La elección de este modelo dependerá de las necesidades y limitaciones de cada caso.

4.5 Mecanismos de Iteración y Corrección de Errores

La implementación de mecanismos de iteración y corrección de errores es una de las partes más críticas de esta arquitectura para garantizar que el sistema pueda manejar errores y producir resultados precisos y útiles. En este apartado, se detallan los mecanismos utilizados en el proyecto para iterar y corregir errores en la generación de consultas SQL.

El proceso de generación de consultas SQL automáticas no es infalible. Debido a la naturaleza ambigua y variable del lenguaje natural, es común que el sistema genere consultas SQL que pueden contener errores de sintaxis, referencias incorrectas a tablas o columnas, o simplemente no proporcionar los resultados esperados. Este proceso se basa en un ciclo de retroalimentación continuo entre diferentes agentes que se encargan de diferentes tareas en la cadena de generación de consultas.

4.5.1 Proceso

El proceso de iteración se gestiona a través de un orquestador, el cual coordina las interacciones entre varios agentes. A continuación, se describen las etapas clave del proceso.

Identificación de Errores

El primer paso en el proceso es la identificación de errores. Cuando el "Sr Data Analyst" ejecuta la consulta SQL generada, se monitorea la respuesta para detectar posibles errores. Esto incluye:

1. Errores de Sintaxis SQL: Si la consulta SQL generada tiene un error de sintaxis, el motor de base de datos devolverá un mensaje de error. Este mensaje es capturado y analizado por el sistema.
Cuando se detecta un error, el sistema vuelve al agente "Data Engineer" con instrucciones específicas para corregir el problema. Por ejemplo, si se detecta un error de sintaxis, el mensaje de error es utilizado para guiar al agente en la modificación de la consulta.

```
   if re.search(r'\berror\b', self.latest_message['content'],
re.IGNORECASE):
       self.basic_chat(previous_agent, agent_a,
           "Generate the correct SQL. Fix the following bug or bugs to fix
the correct SQL: "
           + self.latest_message['content']
           + " \n Keep the following in mind when creates the SQL: 1. Put
the tables names between quotes.\n 2.Check the aliases that correspond
correctly with the tables. \n 3. Check the correct columns names and table
```

```
names. \nScheme:"
            + self.tables_def + "\n Only retrieve the SQL."
        )
```

Código 19: Verifica si el último mensaje contiene la palabra "error". Si es así, envía un mensaje a *agent_a* solicitando la corrección de errores en la consulta SQL, proporcionando detalles sobre las correcciones necesarias y recordando las reglas para la generación del SQL.

Este fragmento de código muestra cómo el sistema analiza el mensaje de error recibido y proporciona directrices específicas proporcionando instrucciones adicionales para modificar la consulta al agente "Data Engineer" para corregir la consulta SQL.

2. Resultados Vacíos: Otro tipo de error ocurre cuando la consulta devuelve un conjunto de resultados vacío que, bajo las expectativas de la consulta, no debería estar vacío. El sistema detecta este tipo de situaciones a través de una verificación del contenido de la respuesta.

```
    elif is_empty_result(self.latest_message['content']):
        additional_instructions = """
        When performing SQL queries, keep the following in mind:
        1. You can use `LOWER()` for case-insensitive comparisons.
        2. Consider using `LIKE` and wildcards like '%' to search for partial
matches, as a word you're looking for might appear partially.
        """
        self.basic_chat(previous_agent, agent_a,
            "Fix the following bug or bugs: "
            + self.latest_message['content']
            + "\nScheme:" + self.tables_def + "\n" + additional_instructions
        )
```

Código 20: Si el último mensaje indica un resultado vacío, envía un mensaje a *agent_a* para corregir la consulta SQL, añadiendo instrucciones adicionales sobre el uso de funciones como *LOWER()* y *LIKE* para mejorar las búsquedas en consultas SQL.

En este código, si la consulta devuelve resultados vacíos que no son esperados, el sistema proporciona instrucciones adicionales para ajustar la consulta, como el uso de comparaciones insensibles a mayúsculas o búsquedas parciales con `LIKE`.

Iteración de Corrección

Una vez que se ha generado una consulta corregida, esta es reenviada al "Sr Data Analyst" para su ejecución. El proceso de ejecución y verificación de errores se repite hasta que la consulta se ejecuta correctamente o se alcanzan los límites de iteración establecidos.

4.6 Agente Explorer (Generador de *Insights*)

El Agente Explorer es un componente diseñado para enriquecer la experiencia del usuario más allá de la consulta inicial. Su principal función es generar nuevas ideas relacionadas con la consulta original, proporcionando *insights* adicionales que podrían ser de interés para el usuario. Estos *insights* pretenden ampliar la información obtenida.

Podemos ver su contexto en la arquitectura en la **Figura 5**.

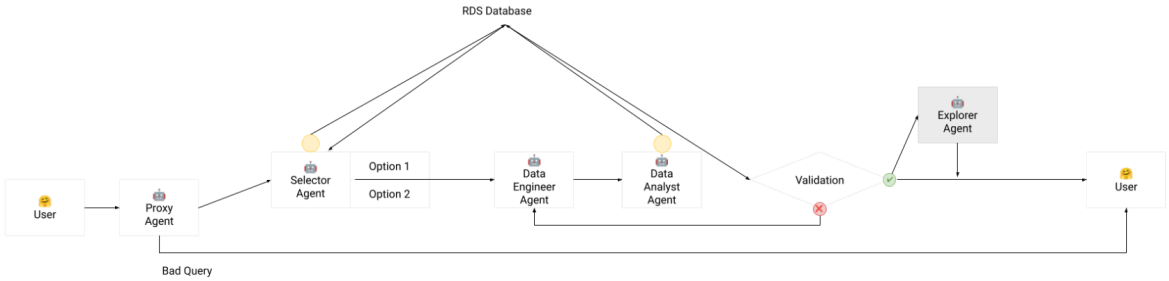

Figura 5. Agente Explorer en contexto a la arquitectura.

La implementación del Agente Explorer se realiza a través de la clase `Orchestrator`, específicamente en el método `generate_insights`. Este método utiliza un agente especializado llamado `Insights_Generator` para crear las nuevas consultas.

```
insights_agent = autogen.AssistantAgent(
    name="Insights_Generator",
    llm_config={
        "temperature": 0.7,
        "config_list": [
            {
                "model": "gpt-4",
                "api_key": OPENAI_API_KEY
            }
        ]
    },
    system_message="You are a data analyst expert in SQL. You generate
insightful SQL queries based on given information.",
)
```

Código 21: Configura un agente de análisis de datos llamado *Insights_Generator* que utiliza el modelo GPT-4 para generar consultas SQL basadas en la información proporcionada.

El Agente Explorer se configura con una temperatura más alta (0.7) para fomentar la creatividad en la generación de *insights* utilizando el modelo GPT-4, que ofrece capacidades más avanzadas.

4.6.1 Proceso

El proceso de generación de *insights* se inicia después de que la consulta original ha sido ejecutada y procesada. Los pasos principales son:

1. **Preparación del *Prompt*:** Se crea un *prompt* especializado que incluye la consulta original y las definiciones de las tablas relevantes.

```
prompt = f"""Based on the original query: '{original_query}' and the
following table definitions:

{table_definitions}

Generate {num_insights} new SQL queries that could provide additional
valuable insights related to the original query. Each insight should
include:
1. A brief description of what the query aims to uncover
2. The business value of this information
3. The SQL query itself

Format your response as a JSON array with the following structure:
[
{{
    "description": "Brief description of what the query aims to uncover",
    "business_value": "Explanation of how this information can drive
business value",
    "sql": "The SQL query"
}}
]
Ensure that each SQL query is valid and uses the provided table structures
correctly.
IMPORTANT: Your entire response must be a valid JSON array."""
```

Código 22: Genera consultas SQL adicionales basadas en una consulta original y definiciones de tablas. Incluye una descripción, valor empresarial y la consulta SQL, formateada como un array JSON.

2. **Generación de *Insights*:** El Agente Explorer procesa el *prompt* y genera los *insights* solicitados.

3. **Procesamiento de la Respuesta y Validación:** La respuesta del agente se procesa para extraer y validar los *insights* generados.

```
try:
    insights = json.loads(reply)
except json.JSONDecodeError:
    json_match = re.search(r'$$[\s\S]*$$', reply)
```

```
    if json_match:
       try:
            insights = json.loads(json_match.group())
        except json.JSONDecodeError:
            print("Error: No se pudo extraer un JSON válido de la
respuesta")
            return []
    else:
        print("Error: No se encontró una estructura JSON en la respuesta")
        return [], ""
...
insights = insights[:num_insights]
valid_insights = []
for insight in insights:
    if all(key in insight for key in ["description", "business_value",
"sql"]):
        valid_insights.append(insight)
    else:
       print(f"Advertencia: Se omitió un insight con estructura incorrecta:
{insight}")
```

Código 23: Procesa la respuesta para extraer un array JSON válido de consultas SQL adicionales. Si la respuesta no es JSON válido, intenta extraer el JSON usando expresiones regulares. Filtra los *insights* para asegurar que contengan los campos "description", "business_value" y "sql".

Una característica clave del Agente Explorer es su flexibilidad. El número de *insights* generados (`num_insights`) es una variable que puede ajustarse según las necesidades del usuario o del proyecto.

```
num_insights = 3
```

Código 24: Define el número de insights adicionales a generar.

Esta flexibilidad permite adaptar la cantidad de información adicional generada, equilibrando entre la profundidad del análisis, el tiempo de procesamiento y el coste asociado.

El Agente Explorer se integra *seamlessly* en el flujo principal de ejecución:

```
insights_result, prompt_insights =
data_eng_orchestrator.generate_insights(args.prompt, table_definitions,
insights_agent, num_insights)

print("\nGenerated Insights:")
print(json.dumps(insights_result, indent=2))
```

```
with open("generated_insights.json", "w") as f:
    json.dump(insights_result, f, indent=2)
```

Código 25: Generación y almacenamiento de *insights*. Se generan *insights* adicionales usando el agente *insights_agent* y se guardan en un archivo JSON llamado *generated_insights.json*.

Los *insights* generados se almacenan en un archivo JSON para su posterior referencia.

4.7 Registro y Cálculo de Costos

El sistema debe llevar a cabo varias tareas, como la clasificación de consultas, la generación de SQL, la ejecución de consultas y la extracción de información relevante, todo lo cual conlleva un costo en términos de uso de *tokens*. Este apartado detalla la metodología empleada para registrar y calcular dichos costos.
En el sistema propuesto, el cálculo de costos se basa en el número de *tokens* procesados por los modelos LLM. Un *token*, en términos generales, es una unidad básica de texto que puede representar una palabra, parte de una palabra o un carácter específico. En los modelos como GPT-3.5 y GPT-4, cada vez que se procesa una entrada de texto, esta se divide en *tokens*, y el modelo cobra una tarifa basada en el número de tokens procesados.

El código incluye una definición clara de los costos asociados con dos modelos específicos:

```
TOKEN_COSTS = {
    "gpt-3.5-turbo": 0.002,  # Cost per 1k tokens in dollars
    "gpt-4": 0.03,           # Cost per 1k tokens in dollars
}
```

Código 26: Definición de costos por token. Se establece el costo por 1,000 tokens para los modelos *gpt-3.5-turbo* y *gpt-4*.

Esta estructura permite que el sistema mantenga un registro del número total de *tokens* procesados y, a su vez, calcule el costo total basado en el modelo utilizado para cada tarea.

4.7.1 Proceso

Cada vez que se envía una consulta al modelo LLM, se cuenta el número de *tokens* utilizando una función dedicada:

```
def count_tokens(text, model='gpt-3.5-turbo'):
    enc = tiktoken.encoding_for_model(model)
    tokens = enc.encode(text)
    return len(tokens)
```

Código 27: Contador de tokens. Esta función calcula el número de tokens en un texto dado utilizando el modelo especificado *(gpt-3.5-turbo* por defecto).

Esta función recibe como parámetros el texto a procesar y el modelo específico para el cual se realizará el conteo de *tokens*. El uso de `tiktoken`, una biblioteca de OpenAI diseñada para el procesamiento de *tokens*, permite que el conteo sea preciso y adaptado a las características específicas de cada modelo. Esto es importante porque el tamaño del *token* puede variar entre diferentes modelos, lo que afecta directamente el costo final.

El procedimiento para calcular los costos es acumulativo, es decir, se va registrando el número de *tokens* generados en cada paso del proceso y se va sumando para obtener el total al final de la operación. En el código, este proceso está integrado de la siguiente manera:

1. **Inicio del Cálculo:** Al inicio de la función principal `main`, el sistema empieza a contar los *tokens* desde el primer momento en que se recibe el *prompt*.

2. **Relevancia de la Consulta y Selección de Tablas:** Una vez que se identifica la relevancia de la consulta y se generan las tablas relevantes, el sistema continúa el proceso de generación de SQL, contando los *tokens* en cada paso.

3. **Interacciones del Agente y *Tokens*:** Las interacciones entre agentes en el sistema (e.g., `Engineer`, `Sr_Data_Analyst`) también generan *tokens* adicionales que se deben registrar. Hay que tener en cuenta que si se opta por un modelo gratuito alojado en HuggingFace no se incrementará el coste de este agente.

4. **Generación de *Insights*:** Finalmente, si se generan *insights* adicionales, estos también se cuentan y se suman al total de *tokens* procesados.

5. **Cálculo Final del Costo:** Una vez que todos los pasos han sido ejecutados y los *tokens* han sido contados, se calcula el costo total basado en la cantidad de *tokens* procesados y el costo por 1000 *tokens* del modelo utilizado.

Este enfoque asegura que se tenga un control preciso sobre el costo de cada operación.

4.7.2 Estrategias de Ahorro

En este apartado, se exponen las estrategias implementadas para optimizar el uso de recursos en el proceso de generación de consultas SQL a partir de *inputs* textuales. El objetivo principal es maximizar la eficiencia y minimizar los costos asociados, particularmente en relación con el consumo de *tokens* cuando se utiliza un modelo como GPT.

- **Clasificación de la Relevancia SQL**

  El primer paso en el *pipeline* es la clasificación de la relevancia de la consulta en relación con SQL. Esta operación implica el uso del modelo para determinar si el *input*

es relevante para una consulta SQL. El costo aquí está directamente relacionado con el número de *tokens* en la consulta y la respuesta del modelo.

```
relevance_score = check_relevance(prompt)
```

Código 28: Evaluación de relevancia de la consulta. Este código llama a la función *check_relevance* con el *prompt* proporcionado para obtener una puntuación que indica la relevancia del *input* para una consulta SQL.

En el caso de que la consulta sea considerada relevante (`relevance_score > 2`), se procede con los pasos subsiguientes. De lo contrario, se evita el procesamiento innecesario, optimizando así el uso de *tokens*.

- **Identificación de Tablas Relevantes**

  El siguiente paso implica la identificación de tablas relevantes para la consulta SQL. Este proceso es crítico porque impacta directamente en la consulta generada y, por ende, en la cantidad de *tokens* procesados. Aquí es donde entra en juego una decisión importante en términos de costos y precisión.

  - ***Estrategia 1: Uso del Modelo GPT para Identificación de Tablas***

    Una opción para la identificación de tablas es utilizar el modelo GPT para evaluar la consulta y determinar todas las tablas relevantes dentro de la base de datos. Esta estrategia es más precisa, ya que GPT puede analizar la estructura y los posibles vínculos entre tablas. Sin embargo, este enfoque también es costoso debido al alto consumo de *tokens* asociado con el uso del modelo y puede ser contraproducente en un esquema muy grande en bases de datos masivas.

  - ***Estrategia 2: Uso de un Modelo In Situ para la Selección de Tablas***

    Como alternativa, se puede optar por una estrategia de reducción de costos utilizando un modelo *in situ* que selecciona las *k* tablas más relevantes basándose en reglas predefinidas o algoritmos de búsqueda de coincidencias simples. Este método no depende del modelo GPT para la identificación de tablas, lo que lo hace aparentemente más económico en términos de *tokens* procesados y una mejor alternativa si el esquema de la base de datos es muy grande. Sin embargo, el principal inconveniente de este enfoque es que puede ser menos preciso. Dado que se limita a devolver solo *k* tablas, existe el riesgo de que algunas tablas relevantes no sean incluidas, lo que podría afectar la calidad de la consulta SQL generada.

4.8 Desarrollo de la Interfaz Gráfica

Este apartado se enfoca en el desarrollo de una interfaz gráfica de usuario web robusta, utilizando el *framework* Django para el *backend* y Vue.js para el *frontend*. La arquitectura está diseñada para separar claramente las responsabilidades del servidor y el cliente, permitiendo una mayor flexibilidad, escalabilidad y mantenimiento. Este enfoque facilita la interacción con la base de datos y la visualización de los resultados de las consultas SQL generadas a partir de lenguaje natural, proporcionando una experiencia de usuario mejorada.

4.8.1 Backend con Django

El backend del proyecto se construyó utilizando Django **[30]**, un *framework* de alto nivel para el desarrollo de aplicaciones web en Python. Django REST Framework **[31]** se utilizó para crear una API que maneje la lógica de negocio y las interacciones con la base de datos.

Estructura de URLs y Vistas

La configuración de las URLs en Django es esencial para definir cómo las solicitudes HTTP se asignan a las vistas correspondientes. En este proyecto, se definieron varias rutas en `core.urls.py` y se conecta con `chat` que es la aplicación o módulo creado para manejar diferentes aspectos de la aplicación y tener un *backend* mejor modularizado, además de permitir la reutilización de estos módulos en otras implementaciones..

```
# core/urls.py
urlpatterns = [
    path('admin/', admin.site.urls),
    path('chat/', include('chat.urls')),
]

# chat/urls.py
urlpatterns = [
    path('messages/', MessageListCreateView.as_view(),
name='message-list-create'),
    path('configuration/', APIConfigurationView.as_view(),
name='configuration'),
    path('run-script/', RunScriptView.as_view(), name='run-script'),
    path('clear-messages/', ClearMessagesView.as_view(),
name='clear-messages'),
]
```

Código 29: Configuración de URLs para la aplicación Django, definiendo rutas para mensajes y configuración.

Estas rutas permiten al usuario interactuar con diferentes aspectos de la aplicación, como la gestión de mensajes tanto para listarlos (GET) como para crearlos (POST) y la configuración de la API con la conexión a la base de datos con la que se quiere interactuar. También está la vista y url para eliminar el historial de mensajes en la conversación.

Modelos de Datos

Django facilita la creación de modelos de datos a través de su ORM (*Object-Relational Mapping*). En este proyecto, se definieron varios modelos clave, como `Message`, `Insight`, y `APIConfiguration`.

En la **Figura 6** se puede ver la organización de los modelos.

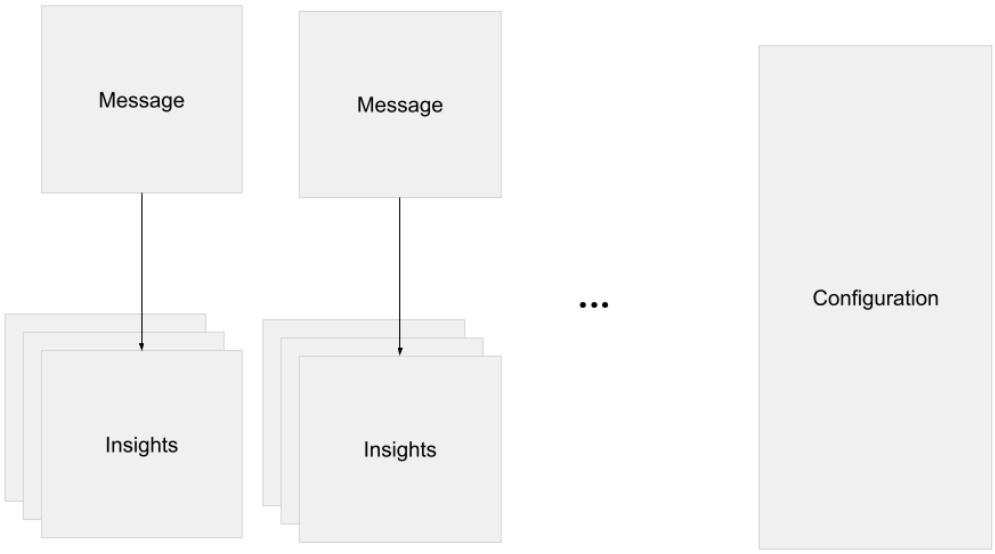

Figura 6. Modelos Message, Insight y Configuration.

```
from django.db import models

class Message(models.Model):
    sender = models.CharField(max_length=255)
    content = models.TextField()
    timestamp = models.DateTimeField(auto_now_add=True)
    prompt = models.TextField(null=True, blank=True)

class Insight(models.Model):
    message = models.ForeignKey(Message, related_name='insights',
on_delete=models.CASCADE)
    description = models.CharField(max_length=255)
    business_value = models.CharField(max_length=255, null=True,
blank=True)
    sql = models.TextField(null=True, blank=True)

class APIConfiguration(models.Model):
    hostname = models.CharField(max_length=255)
    database = models.CharField(max_length=255)
    username = models.CharField(max_length=255)
```

```
    password = models.CharField(max_length=255)
    port = models.IntegerField()
    openai_api_key = models.CharField(max_length=255)
    huggingface_api_key = models.CharField(max_length=255, default=None)
    opcion = models.BooleanField(default=False)
    model_name = models.CharField(max_length=200, blank=True, default='')
    num_insights = models.IntegerField(default=1)
```

Código 30: Definición de modelos Django para almacenar mensajes, *insights* y configuraciones API, incluyendo campos para texto, claves foráneas y parámetros de configuración.

Estos modelos representan las entidades principales de la aplicación.

- `Message` almacena los mensajes enviados por los usuarios, incluyendo el contenido que representa la respuesta del sistema de agentes y la marca de tiempo para tener un registro en el historial de mensajes del chat.
- `Insight` se asocia con un mensaje específico y almacena la descripción, el valor de negocio y la consulta SQL generada. Un mensaje puede tener varios *insights*.
- `APIConfiguration` permite configurar los parámetros necesarios para la conexión con bases de datos con la que se quiere chatear y el uso de APIs externas como OpenAI y Hugging Face para el acceso a los modelos. Además también se registra el número de *insights* que el usuario quiere que le ofrezca el sistema y si quiere o no activar la opción de ahorro de *tokens* a la hora de seleccionar las tablas, así como el modelo que se usará como SQL Coder (Data Engineer).

Este modelo está implementado de tal manera que asegura que solo exista una configuración global activa en cualquier momento, garantizando que los parámetros de conexión y API sean consistentes en toda la aplicación.

4.8.2 Frontend con Vue.js y Vite

Para el *frontend*, Vue.js **[32]** fue la tecnología elegida debido a su flexibilidad y facilidad de integración con otras herramientas modernas de desarrollo web. Vite **[33]**, un constructor rápido y ligero para Vue.js, se utilizó para gestionar y construir el proyecto.

Rutas y Componentes

El *frontend* se estructura en torno a un sistema de enrutamiento, permitiendo que los usuarios naveguen entre diferentes páginas, como la página de chat, la página de configuración y la página de información. Estas rutas se definieron en el archivo `router/index.js`.

```
import { createRouter, createWebHistory } from 'vue-router'
import HomePage from '../views/ChatPage.vue'
import AboutPage from '../views/AboutPage.vue'
import ConfigurationPage from '../views/ConfigurationPage.vue'

const routes = [
  { path: '/', name: 'Home', component: HomePage },
  { path: '/about', name: 'About', component: AboutPage },
  { path: '/configuration', name: 'Configuration', component:
ConfigurationPage },
]

const router = createRouter({
  history: createWebHistory(),
  routes,
})

export default router
```

Código 31: Configuración de rutas para la aplicación Vue.js, definiendo las rutas para la página principal del chat, la página de información y la página de configuración.

Cada ruta está asociada a un componente Vue específico que maneja la lógica y la interfaz de usuario para esa sección de la aplicación. Por ejemplo, `ChatPage.vue` es responsable de la interfaz principal donde los usuarios interactúan con el sistema de chat, mientras que `ConfigurationPage.vue` permite la modificación de los parámetros de configuración.

Estructura de la Aplicación

El componente principal `App.vue` actúa como el contenedor general de la aplicación, integrando otros componentes como la cabecera y el pie de página, y sirviendo como punto de entrada para las diferentes vistas definidas en las rutas.

```
<script setup>
import HeaderComponent from './components/HeaderComponent.vue'
import FooterComponent from './components/FooterComponent.vue'
</script>

<template>
 <HeaderComponent/>
  <router-view/>
  <FooterComponent/>
</template>

<style scoped>
</style>
```

Código 32: Componente principal de Vue.js que incluye el encabezado y el pie de página, con un área central para renderizar componentes de vista según la ruta activa.

Este enfoque modular permite un desarrollo más organizado y facilita el mantenimiento de la interfaz de usuario, además permite viajar entre la aplicación sin necesidad de recargar las páginas.

4.8.3 Separación del Backend y Frontend

Una de las decisiones arquitectónicas clave en este proyecto fue la separación del *backend* y el *frontend* en dos aplicaciones independientes. Esta separación no solo sigue las mejores prácticas modernas de desarrollo web, sino que también ofrece varios beneficios:

- **Despliegue Independiente:** Permite desplegar el *backend* y el *frontend* en servidores separados o en contenedores Docker individuales, lo que facilita la escalabilidad y la gestión de recursos.

- **Modularidad y Mantenimiento:** Al mantener el *backend* y el *frontend* separados, es más fácil realizar cambios en uno sin afectar al otro. Esto reduce el riesgo de introducir errores y facilita la adición de nuevas funcionalidades.

- **Mejora en el Rendimiento:** La separación permite optimizar cada parte de la aplicación según sus necesidades específicas. Por ejemplo, el *backend* puede enfocarse en la eficiencia en el manejo de datos, mientras que el *frontend* puede optimizarse para una experiencia de usuario fluida y receptiva.

4.8.4 Dockerización

Finalmente, para asegurar un entorno de despliegue consistente y reproducible, tanto el *backend* como el *frontend* fueron dockerizados. La dockerización es un proceso fundamental en el desarrollo de software moderno, ya que encapsula todas las dependencias y configuraciones necesarias para ejecutar una aplicación dentro de un contenedor permitiendo una mejor escalabilidad y mantenimiento.

Importancia de la Dockerización

- **Aislamiento de Entorno:** Docker **[34]** garantiza que la aplicación se ejecute en el mismo entorno en cualquier máquina, eliminando problemas de configuraciones locales.

- **Escalabilidad:** Los contenedores Docker pueden ser fácilmente escalados para manejar cargas de trabajo variables, lo que es útil en entornos de producción donde la demanda puede fluctuar.

- **Despliegue Simplificado:** Docker simplifica el proceso de despliegue, permitiendo que la misma imagen de contenedor sea desplegada en diferentes entornos.

4.9 Seguridad y Prompt Injection

En esta sección se discuten las estrategias implementadas para garantizar la seguridad de la base de datos ante posibles ataques de inyección de *prompts*. Es importante tenerlo en cuenta, ya que una consulta maliciosa podría comprometer la seguridad de la base de datos, exponiendo información sensible o causando daños irreversibles.

El uso de modelos de lenguaje natural para la generación automática de consultas SQL introduce un nuevo vector de ataque: el *prompt injection*. Este tipo de ataque consiste en manipular el *input* del usuario o del sistema para insertar comandos maliciosos que podrían ejecutarse en la base de datos. Dado que los modelos de lenguaje natural, como los utilizados en esta investigación, interpretan texto en lenguaje natural para convertirlo en consultas SQL, cualquier vulnerabilidad en este proceso podría ser explotada por atacantes para ejecutar comandos no deseados.

El objetivo principal al abordar el *prompt injection* es prevenir la ejecución de cualquier consulta SQL que no sea estrictamente una consulta de tipo `SELECT`. Esto se debe a que estas consultas son consultas de solo lectura y no pueden modificar la base de datos. Cualquier otra consulta, como `INSERT`, `UPDATE`, `DELETE`, o incluso comandos de administración como `DROP TABLE`, pueden ser un riesgo.

4.9.1 Estrategias

Para mitigar estos riesgos, se han implementado varias técnicas tanto a nivel de *prompting* como a nivel sintáctico en el código que maneja las interacciones con la base de datos.

1. **Control de Acceso Basado en Roles**

   Uno de los primeros niveles de seguridad implementados es la configuración de un rol de solo lectura para el usuario que se conecta a la base de datos. Esto se logra mediante el siguiente fragmento de código:

```
    def connect_ddbb(self, hostname, database, username, password, port):

            self.conn = psycopg2.connect(
                host=hostname,
                database=database,
                user=username,
                password=password,
                port=port
            )

            self.cur = self.conn.cursor()
```

```
        self.cur.execute("DO $$ BEGIN IF NOT EXISTS (SELECT FROM
pg_catalog.pg_roles WHERE rolname = 'readonly') THEN CREATE ROLE
readonly; END IF; END $$;")


        self.cur.execute(sql.SQL("""
            GRANT CONNECT ON DATABASE {database} TO readonly;
            GRANT USAGE ON SCHEMA public TO readonly;
            GRANT SELECT ON ALL TABLES IN SCHEMA public TO readonly;
            ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON
TABLES TO readonly;
        """).format(database=sql.Identifier(database)))


        self.cur.execute(sql.SQL("GRANT readonly TO
{user};").format(user=sql.Identifier(username)))


        self.conn.commit()
```

Código 33: Conexión a base de datos a través de un rol de solo lectura.

En este código, se define un rol llamado `readonly` que está limitado a realizar operaciones de lectura en la base de datos. Este rol se asigna dinámicamente al usuario que se conecta, lo que asegura que, independientemente del *input* recibido, el usuario solo pueda realizar consultas `SELECT`.

2. Validación de Sintaxis en la Ejecución de Consultas

El siguiente nivel de seguridad se implementa en la función que realmente ejecuta las consultas SQL. Antes de ejecutar cualquier consulta, se verifica que la consulta sea de tipo `SELECT`. Si no lo es, la función levanta una excepción y no ejecuta la consulta:

```
def run_sql(self, sql) -> str:
        self.cur.execute("ROLLBACK;")
        if not "SELECT" in sql:
            raise ValueError("Only SELECT queries")

        self.cur.execute(sql)
        columns = [desc[0] for desc in self.cur.description]
        res = self.cur.fetchall()

        list_of_dicts = [dict(zip(columns, row)) for row in res]

        json_result = json.dumps(list_of_dicts, indent=4,
```

```
        default=self.datetime_handler)

                return json_result
```

Código 34: Cortafuegos para la detección solo consultas *SELECT*.

Esta función primero realiza un `ROLLBACK` para asegurarse de que no haya transacciones activas pendientes que puedan interferir con la consulta. Luego, verifica que la consulta contiene la palabra `SELECT`. Si no la contiene, la función no permite que se ejecute, evitando así cualquier consulta que podría modificar los datos.

Además, al capturar y manejar las excepciones de manera adecuada, la función asegura que cualquier intento de ejecutar una consulta no permitida sea reportado y manejado sin comprometer la estabilidad del sistema.

5 Resultados y Discusión

En esta sección, se da un análisis detallado de los resultados obtenidos tras la implementación de los componentes del sistema propuesto. La discusión se enfocará en cómo cada componente del sistema contribuye a alcanzar los objetivos establecidos.
La arquitectura final del proyecto puede verse en la **Figura 7**.

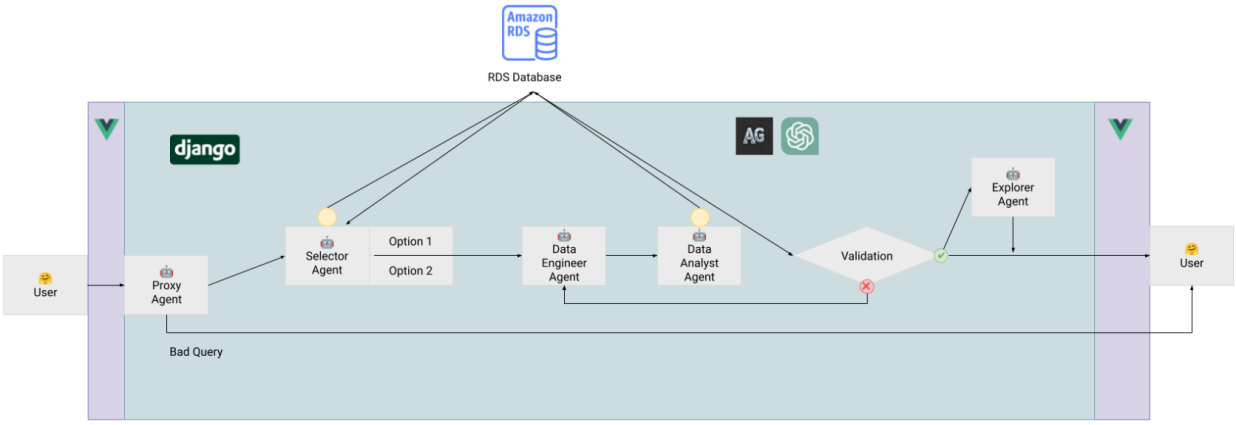

Figura 7. Arquitectura final.

5.1 Consideraciones

Las pruebas realizadas durante este estudio se hicieron con la base de datos Chinook, dado su amplio reconocimiento y utilización. Chinook **[35]** es una base de datos ampliamente conocida en el mundo del *data analytics* y es frecuentemente utilizada como referencia en experimentos de generación y análisis de consultas SQL. Su popularidad se debe a su estructura y contenido, que simula un entorno de tienda de medios digitales, incluyendo tablas que representan artistas, álbumes, pistas, clientes, empleados, e información sobre facturación y ventas. Esta estructura nos da un escenario ideal para probar una variedad de consultas SQL, desde operaciones simples hasta consultas más complejas que implican múltiples uniones y subconsultas.

Sin embargo, a pesar de las ventajas que ofrece Chinook, es importante reconocer que no es la única base de datos disponible para este tipo de análisis. Existen otras bases de datos reconocidas, como Spider, Bird, y WikiSQL (ya comentadas anteriormente), que son también ampliamente utilizadas en la investigación.
A pesar del potencial de estas bases de datos para proporcionar pruebas más complejas, no fueron consideradas en este estudio debido a las limitaciones de recursos disponibles, particularmente en términos de la inversión en la API de OpenAI. La complejidad y el volumen de datos que manejan estas bases de datos habrían requerido mayores costos asociados. El uso de la API de OpenAI implica un costo por cada solicitud realizada, y dado

que la generación de consultas SQL en estas bases de datos más complejas habría implicado un número mayor de iteraciones y pruebas, los costos asociados habrían superado los recursos disponibles para este estudio.

El enfoque en Chinook permitió, por lo tanto, una evaluación efectiva dentro de los límites presupuestarios establecidos. La selección de esta base de datos se alineó con el objetivo de obtener resultados representativos, manteniendo al mismo tiempo un control sobre los costos.
En cuanto a la selección de las preguntas utilizadas para las pruebas, se optó por utilizar un conjunto de preguntas proporcionadas por un usuario de GitHub, específicamente en el repositorio "Postgres-sql-Chinook" **[36]**. Este repositorio ofrece un conjunto de preguntas diseñadas para trabajar con la base de datos Chinook, con una base sólida para evaluar las capacidades de los modelos en la generación y ejecución de consultas SQL.
El uso de preguntas predefinidas y disponibles públicamente asegura un nivel de estandarización, reproducibilidad en los experimentos y se evita la introducción de sesgos al formular preguntas.
En relación con la elección del modelo Data Engineer (SQL Coder), solo se permiten modelos pequeños que permita la API de HuggingFace, en las últimas semanas han cambiado los requerimientos de la API y ya no permite modelos más grandes de por ejemplo 7-B de parámetros.
En cuanto a los costos asociados al uso de la API de OpenAI para la generación y ejecución de consultas SQL, es importante señalar que el crédito total gastado en estas pruebas fue de aproximadamente 15 dólares, además del uso del período de prueba gratuito (0$) de AWS para mantener el servicio RDS (Relational Database Service) donde se almacenó la base de datos Chinook.

5.2 Agente Proxy (Evaluador NLQ)

Para evaluar la efectividad del agente proxy, se diseñó un experimento en el que se formularon un total de 20 consultas en lenguaje natural. Estas consultas se dividieron en dos categorías: 10 de ellas fueron consultas con una clara relevancia para SQL, mientras que las otras 10 eran consultas irrelevantes o ambiguas, es decir, consultas que no deberían dar lugar a una consulta SQL ejecutable.

El proceso de evaluación consistió en presentar cada una de estas consultas al agente proxy para determinar si era capaz de identificar correctamente la relevancia de la consulta. Para cada consulta, se registró la relevancia esperada (ya sea SQL o no SQL) y la relevancia obtenida por el agente. La precisión del agente se evaluó en términos de su capacidad para filtrar las consultas irrelevantes y permitir el paso de aquellas que realmente requerían una consulta SQL.

De las 20 consultas evaluadas, el agente proxy clasificó correctamente 19 de ellas. Específicamente, identificó correctamente 9 de las 10 consultas irrelevantes como no SQL y 10 de las 10 consultas relevantes como SQL. Esto demuestra una alta precisión en la identificación de consultas relevantes, con solo un error en la clasificación de una consulta irrelevante.

La siguiente tabla presenta los resultados obtenidos durante el experimento:

Query	Expected Relevance	Obtained Relevance
How many active users are in the database?	SQL	✅
What is the average salary of employees in the sales department?	SQL	✅
What products were sold in the last month?	SQL	✅
How many orders were placed in the last week?	SQL	✅
What is the email address of the customer with ID 1234?	SQL	✅
How many employees are there in each department?	SQL	✅
What is the date of the last recorded transaction?	SQL	✅
Which products have an inventory of less than 10 units?	SQL	✅
Which customers have not made any purchases this year?	SQL	✅
How many times has the user table been updated this month?	SQL	✅
What day is today?	Not SQL	✅
What time is it in Tokyo now?	Not SQL	✅
How do I cook a Spanish omelette?	Not SQL	✅
How many liters of water should I drink per day?	Not SQL	✅
What is the formula for calculating speed?	Not SQL	✅
How do you write a formal letter?	Not SQL	✅
What does the word "otolaryngologist" mean?	Not SQL	✅
What color do you get by mixing red and blue?	Not SQL	✅
How do you say "thank you" in	Not SQL	✅

Query	Expected Relevance	Obtained Relevance
Japanese?		
What books do you recommend reading this year?	Not SQL	✅
What is the main ingredient of guacamole?	Not SQL	❌
How can I improve my memory?	Not SQL	✅

Tabla 1: Resultados de la Evaluación del Agente Proxy en la Clasificación de la Relevancia de Consultas SQL.

Como se puede observar, el agente proxy mostró un rendimiento sólido, con una tasa de precisión del 95%.

Al identificar correctamente las consultas irrelevantes, el sistema puede evitar el procesamiento innecesario de estas, lo que se traduce en un ahorro directo de recursos.

5.3 Agente Selector (Selección de Tablas)

En la arquitectura implementada se propuso abordar la fase de selección de tablas mediante la implementación de dos enfoques distintos que, aunque comparten el objetivo común de identificar las tablas más relevantes para una consulta SQL, se diferencian en sus métodos y recursos utilizados.
El primer enfoque que hemos implementado se basa en un agente selector alimentado el modelo GPT-3.5. Este agente tiene la tarea de analizar el contenido de la consulta SQL en lenguaje natural y determinar, con gran precisión, qué tablas dentro de la base de datos son las más relevantes para satisfacer la consulta.
El segundo enfoque implementado busca mitigar los altos costos asociados al uso de modelos de lenguaje, sin sacrificar en gran medida la precisión en la selección de tablas. En lugar de depender de un modelo de lenguaje avanzado, este método utiliza una aproximación semántica combinada con la técnica de cercanía por coseno para identificar las tablas relevantes.

Para evaluar la efectividad de ambos enfoques, realizamos una serie de pruebas utilizando un conjunto de 20 consultas en lenguaje natural. En la **Tabla 2**, se detallan los resultados obtenidos para cada una de las 20 consultas analizadas.

Prompt	Option 1(paid)	Option 2 (free)
Provide Customers (just their full names, customer ID and country) who are not in the US.	✅	✅
Customers from Brazil.	✅	✅

Prompt	Option 1(paid)	Option 2 (free)
Invoices of customers who are from Brazil.	✅	✅
list of billing countries from the Invoice table	✅	✅
invoices of customers who are from Brazil.	✅	✅
invoices associated with each sales agent. The resultant table should include the Sales Agent's full name.	✅	✅
shows the invoices associated with each sales agent. The resultant table should include the Sales Agent's full name.	✅	✅
shows the Invoice Total, Customer name, Country and Sale Agent name for all invoices and customers.	✅	✅
Looking at the InvoiceLine table, COUNTs the number of line items for Invoice ID 37.	✅	✅
Looking at the InvoiceLine table, COUNTs the number of line items for each Invoice.	✅	✅
Provide the track name with each invoice line item.	✅	✅
Provide the purchased track name AND artist name with each invoice line item.	✅	✅
shows the # of invoices per country.	✅	✅
shows the total number of tracks in each playlist. The Playlist name should be include on the resultant table.	✅	✅
shows all the Tracks, but displays no IDs. The resultant table should include the Album name, Media type and Genre.	✅	✅
Invoices but includes the # of invoice line items.	✅	✅
Which sales agent made the most in sales in 2009?	✅	✅
shows the # of customers assigned to each sales agent.	✅	✅
shows the total sales per country. Which country's customers spent the most?	✅	✅
shows the most purchased track of 2013.	✅	✅

Tabla 2: Comparación de Resultados entre Opción 1 y Opción 2 en la Selección de Tablas.

Los resultados obtenidos fueron sorprendentemente similares entre ambos enfoques. Tanto el agente basado en GPT-3.5 (Opción 1) como la aproximación semántica con cercanía por

coseno (Opción 2) lograron identificar correctamente las tablas relevantes en todas las consultas realizadas.
Este resultado es significativo por varias razones. Primero, demuestra que la aproximación semántica y la técnica de cercanía por coseno, a pesar de ser un método aparentemente más económico, puede rivalizar en precisión con un modelo de lenguaje avanzado como GPT-3.5. Esto sugiere que, dependiendo del contexto y los recursos disponibles, la opción 2 puede ser una alternativa viable aunque ya veremos en el apartado de costes realmente sus implicaciones y se compararán los costes.

Además, la capacidad de ambos enfoques para acertar muestra la efectividad general del *pipeline* de selección de tablas implementado en la arquitectura. Aunque los dos métodos presentan diferentes fortalezas y debilidades, el hecho de que ambos logren un rendimiento tan alto sugiere que el sistema en su conjunto es robusto.

5.4 Agentes Analistas, Iteración y Corrección de Errores

En esta sección, se analiza el rendimiento del sistema de agentes analistas en la generación y ejecución de consultas SQL, haciendo énfasis en cómo estos agentes colaboran de manera iterativa para mejorar la precisión de las consultas generadas. Este enfoque se centra en un proceso de retroalimentación iterativa, que permite a los agentes corregir errores y optimizar las consultas hasta que se obtienen los resultados deseados.

Como ya se ha mencionado anteriormente, el sistema está diseñado alrededor de un equipo de agentes especializados, cada uno con un rol específico en el proceso de generación y ejecución de consultas SQL. Estos agentes incluyen al "User Proxy Agent", "Data Engineer Agent" y "Sr Data Analyst Agent", todos coordinados por un orquestador central que asegura un flujo de trabajo fluido y eficiente.
Inicialmente, el equipo de agentes incluía a un "Product Manager", cuyo rol consistía en validar las respuestas generadas y presentarlas en un formato más humano y accesible. Sin embargo, se tomó la decisión de prescindir de este agente debido a varios factores. En primer lugar, los costos asociados a este proceso se incrementaban considerablemente al volcar los resultados de las consultas en el agente, ya que involucraba interacciones adicionales con los modelos de lenguaje avanzados. Además, esto resultaba contraproducente, pues ocupaba gran parte de la ventana de contexto, limitando la capacidad del sistema para procesar información relevante y además los modelos de lenguaje tienen un límite máximo de ventana de contexto. También se observó la posibilidad de que datos privados o sensibles fueran volcados en los servidores externos lo cual comprometía la confidencialidad de la información. Por estas razones, se optó por simplificar el proceso, eliminando al "Product Manager" virtual y ajustando el flujo de trabajo.

Dado que el lenguaje natural es ambiguo y puede llevar a la generación de consultas incorrectas, se ha implementado un ciclo de retroalimentación que permite a los agentes identificar y corregir errores de manera iterativa.
Para evaluar la efectividad del proceso iterativo, se realizaron pruebas utilizando 20 preguntas diferentes.

- Una Iteración: En la primera iteración, el sistema mostró un rendimiento moderado, con 6 de las 20 consultas fallando en generar los resultados correctos.

- Dos Iteraciones: Al permitir que el sistema realice una segunda iteración, el número de fallos se redujo, con solo 1 de las 20 consultas fallando. Esto demuestra la efectividad del proceso iterativo en la corrección de errores.

- Tres Iteraciones: Con tres iteraciones, el sistema mantuvo un nivel de precisión alto, con solo 1 de las 20 consultas fallando. Esto sugiere que, en la mayoría de los casos, dos iteraciones son suficientes para corregir errores, y una tercera iteración solo mejora en casos puntuales.

Como se puede observar en la **Tabla 3**, los resultados muestran claramente la importancia del proceso iterativo en la mejora de la precisión de las consultas SQL generadas.

Query	Iterations		
	1	2	3
Provide Customers (just their full names, customer ID and country) who are not in the US.	✅		
Customers from Brazil.	✅		
Invoices of customers who are from Brazil.	✅		
list of billing countries from the Invoice table	✅		
invoices of customers who are from Brazil.	✅		
invoices associated with each sales agent. The resultant table should include the Sales Agent's full name.	❌	✅	
shows the invoices associated with each sales agent. The resultant table should include the Sales Agent's full name.	❌	✅	
shows the Invoice Total, Customer name, Country and Sale Agent name for all invoices and customers.	✅		
Looking at the InvoiceLine table, COUNTs the number of line items for Invoice ID 37.	✅		
Looking at the InvoiceLine table, COUNTs the number of line items for each Invoice.	✅		
Provide the track name with each invoice line item.	✅		
Provide the purchased track name AND artist name with each invoice line item.	❌	❌	❌
shows the # of invoices per country.	✅		

Query	Iterations		
	1	2	3
shows the total number of tracks in each playlist. The Playlist name should be include on the resultant table.	❌	✅	
shows all the Tracks, but displays no IDs. The resultant table should include the Album name, Media type and Genre.	✅		
Invoices but includes the # of invoice line items.	✅		
Which sales agent made the most in sales in 2009?	❌	✅	
shows the # of customers assigned to each sales agent.	❌	✅	
shows the total sales per country. Which country's customers spent the most?	✅		
shows the most purchased track of 2013.	✅		

Tabla 3: Precisión de los Agentes Analistas en Diferentes Iteraciones.

Así pues, el sistema de agentes analistas ha demostrado ser efectivo en la generación y ejecución de consultas SQL. Los resultados obtenidos muestran la importancia de la colaboración entre agentes.

5.5 Análisis de Costes y Tiempo de Ejecución

En esta sección, analizamos cómo la generación de consultas SQL, tanto con la opción 1 (modelo pago) como con la opción 2 (modelo gratuito), afecta a los costos y tiempos de procesamiento.
Para entender esta dinámica, se realizaron experimentos comparativos, ejecutando 10 consultas con 0 insights y luego repitiendo el mismo proceso con 2 insights, midiendo tanto el costo como el tiempo promedio de cada opción tal y como se muestra en la **Tabla 4** y **5**:

	0 insights			
	Option 1(paid)		Option 2 (free)	
Query	Cost	Time	Cost	Time
1	0,001144	4,720	0,001222	7,900
2	0,001074	4,790	0,001154	6,140
3	0,001538	4,400	0,001288	6,120
4	0,000934	5,750	0,000894	6,480
5	0,001534	4,530	0,001284	5,830

6	0,001868	4,980	0,001826	6,310
7	0,001876	4,940	0,001334	6,520
8	0,001892	5,010	0,001378	6,150
9	0,000954	4,670	0,000928	5,940
10	0,001124	4,460	0,000928	6,150

Tabla 4: Comparativa de costes y tiempo de ejecución usando la opción 1 y 2 para 0 insights.

	2 insights			
	Option 1(paid)		**Option 2 (free)**	
Query	**Cost**	**Time**	**Cost**	**Time**
1	0,0022	14,07	0,00322	17,87
2	0,002066	11,39	0,003088	15,32
3	0,002926	15,58	0,002856	15,21
4	0,002528	13,85	0,002458	18,1
5	0,002918	10,46	0,002848	17,04
6	0,003382	15,31	0,00343	17,17
7	0,003398	14,18	0,002946	16,13
8	0,003886	14,15	0,002982	16,1
9	0,001714	14,82	0,002342	16,25
10	0,002244	12,7	0,00233	14,29

Tabla 5: Comparativa de costes y tiempo de ejecución usando la opción 1 y 2 para 2 insights.

Los resultados obtenidos ofrecen una perspectiva clara sobre la eficiencia y rentabilidad de ambas opciones en diferentes escenarios. Como se observa en la **Tabla 6** y **7**, para el conjunto de consultas sin generación de *insights*, la opción 2 parece ser ligeramente más económica en términos de costo directo. Sin embargo, esta diferencia es tan pequeña que, en la práctica, podría considerarse insignificante. La opción 1, aunque ligeramente más costosa, no presenta una diferencia significativa en esta primera fase, sugiriendo que ambos modelos son casi equivalentes en términos de costo cuando no se generan *insights* adicionales.

	0 insights	
	Option 1(paid)	**Option 2 (free)**
MEAN		

	Cost	Time	Cost	Time
	0,0013938	4,825	0,0012236	6,354

Tabla 6: Promedio de la comparativa de costes y tiempo de ejecución usando la opción 1 y 2 para 0 *insights*.

	2 insights			
	Option 1(paid)		Option 2 (free)	
	Cost	Time	Cost	Time
MEAN	0,0027262	13,651	0,00285	16,348

Tabla 7: Promedio de la comparativa de costes y tiempo de ejecución usando la opción 1 y 2 para 2 *insights*.

No obstante, el panorama cambia de manera notable al incorporar la generación de 2 *insights* en las consultas. En este caso, la opción 1 demuestra ser más económica que la opción 2, lo que a primera vista podría parecer contradictorio, dado que la opción 2 inicialmente parecía más rentable. Esta discrepancia se debe a la forma en que cada modelo maneja la selección y procesamiento de tablas relevantes. El modelo de la opción 2, que se basa en la selección "gratuita" de tablas, fija el número de tablas relevantes en 5, independientemente de si todas son necesarias para la consulta. Este enfoque resulta en un incremento en la cantidad de datos que se deben manejar en las siguientes fases del proceso. Específicamente, al pasar 5 tablas al equipo de agentes junto con sus respectivos esquemas, se genera un volumen de datos considerablemente mayor. En contraste, la opción 1, aunque incurre en un costo adicional para la selección de tablas, optimiza el proceso al elegir únicamente aquellas tablas que son realmente necesarias, generalmente entre 1 y 2 tablas. Este enfoque más selectivo reduce la cantidad de *tokens* que se deben procesar en el contexto y minimiza la complejidad y el tiempo requerido para manejar estos datos en las etapas posteriores.

La ventaja de la opción 1 se hace aún más evidente cuando se consideran las consultas con generación de *insights*. Como se observa en las **Tablas 6** y **7**, al incluir 2 *insights*, el modelo de la opción 1 resulta más económico. Esto se debe a que la opción 2, al manejar un mayor volumen de tablas innecesarias, incurre en costos adicionales relacionados con el procesamiento de datos y la generación de *insights*. En contraste, la opción 1, al reducir la cantidad de tablas a las estrictamente necesarias, minimiza estos costos.
Cabe destacar que, en la opción 1 y 2, el costo de la operación prácticamente se duplica al pasar de una consulta sin *insights* a una con *insights*, aunque hay que tener en cuenta que se usó el modelo GPT-4 el cual ya supone el doble de coste en comparación con el modelo GPT-3.5.

Además del costo, el tiempo de ejecución también es un factor importante. En este aspecto, la opción 1 muestra un mejor rendimiento en términos de tiempo, tanto para consultas sin *insights* como con *insights*. Como se puede ver en las **Tablas 6** y **7**, el tiempo promedio de ejecución es menor en la opción 1 en ambos escenarios. Esta ventaja en el tiempo puede atribuirse nuevamente a la eficiencia en la selección de tablas, que no solo reduce el volumen de datos a procesar, sino que también agiliza el flujo de trabajo del equipo de agentes.
Un aspecto adicional a considerar es el incremento en el tiempo de procesamiento asociado con la generación de *insights*. La adición de *insights* introduce un aumento significativo en el tiempo de ejecución, independientemente de la opción utilizada. Este incremento, que en promedio ronda los 10 segundos, muestra la complejidad añadida que implica la generación de *insights*.

5.6 Prompt Injection

Las estrategias implementadas para la prevención de *prompt injection* han demostrado ser efectivas en mantener la seguridad e integridad de la base de datos. Durante las pruebas realizadas, no se registraron incidentes donde se haya logrado ejecutar una consulta no autorizada, lo cual es un indicador claro de la robustez de las medidas de seguridad implementadas.

El uso de roles de solo lectura, combinado con una validación estricta de las consultas antes de su ejecución, ofrece una solución robusta contra el *prompt injection*. Estas técnicas aseguran que la base de datos permanezca protegida contra ataques directos y evitan que usuarios inadvertidamente ejecuten consultas que podrían tener consecuencias no deseadas. Las restricciones implementadas aseguran que estos intentos sean bloqueados antes de que puedan causar cualquier daño.

5.7 Interfaz y Despliegue

En esta sección, se aborda el proceso de implementación y despliegue de la aplicación web, así como las funcionalidades clave que ofrece a los usuarios finales.

5.7.1 Despliegue

Uno de los primeros pasos en la puesta en marcha de la aplicación fue la decisión de utilizar Docker para la contenerización del *front-end* y el *back-end* facilitando así el despliegue y su reproducibilidad. Se crearon dos imágenes distintas: una para el *back-end* desarrollado con Django y otra para el *front-end* desarrollado con Vue.js, tal y como ya se mencionó en apartados anteriores.
A través de un archivo `docker-compose.yml` se definen los servicios para ambas partes de la aplicación y para su montaje. El servicio de Django está configurado para escucharse en el puerto 8000, mientras que el servicio de Vue.js opera en el puerto 5173. Ambos servicios pueden ser levantados simultáneamente mediante el comando `docker-compose up`, lo que simplifica el proceso de despliegue.

5.7.2 Páginas y Componentes de la Aplicación Web

La aplicación web cuenta con tres páginas principales:

Chat: La página de Chat es el corazón de la aplicación, donde el usuario puede interactuar directamente con el equipo de agentes a través de un simple input de chat tal y como se muestra en la **Figura 8**.

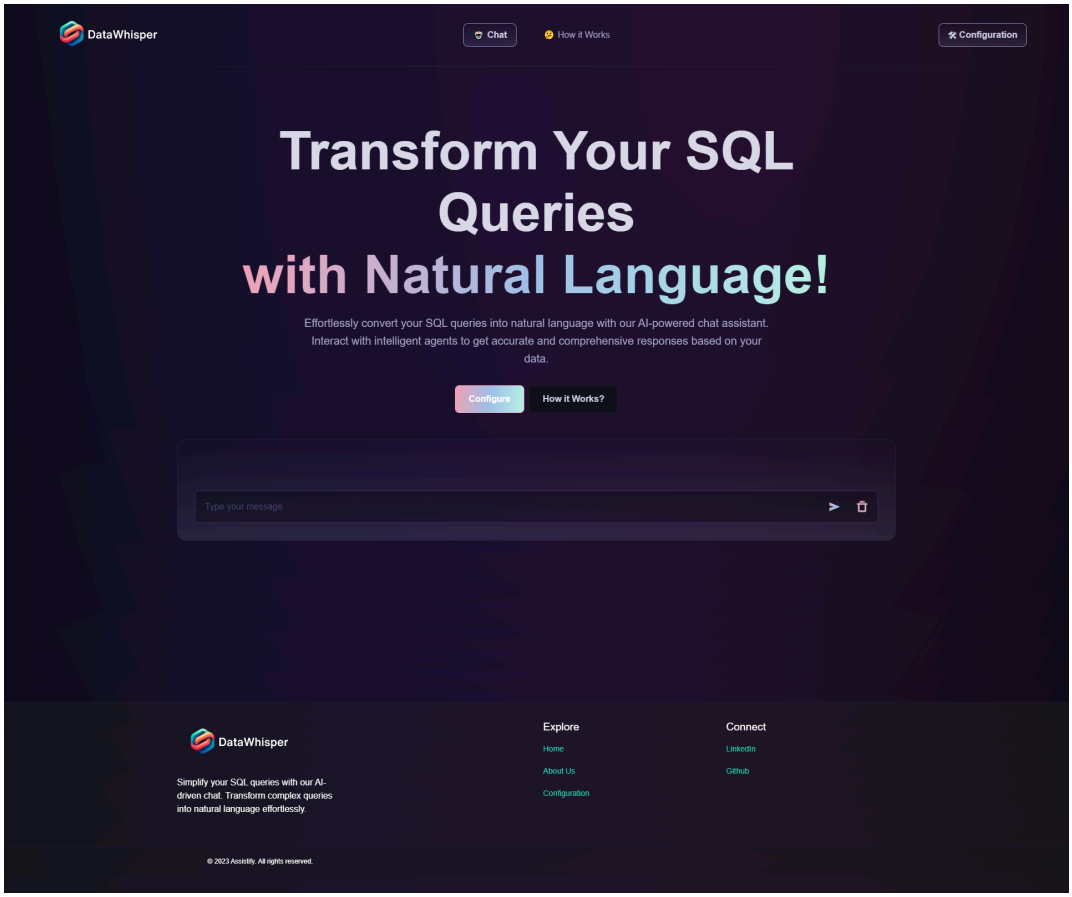

Figura 8. Captura de la página del Chat.

Cuando un usuario envía una consulta en lenguaje natural, la solicitud se procesa y se muestra un *skeleton* (esqueleto de carga) mientras se genera la respuesta, proporcionando así una experiencia de usuario más fluida y menos frustrante, evitando que el usuario perciba demoras significativas durante el procesamiento, tal y como se observa en la **Figura 9**.

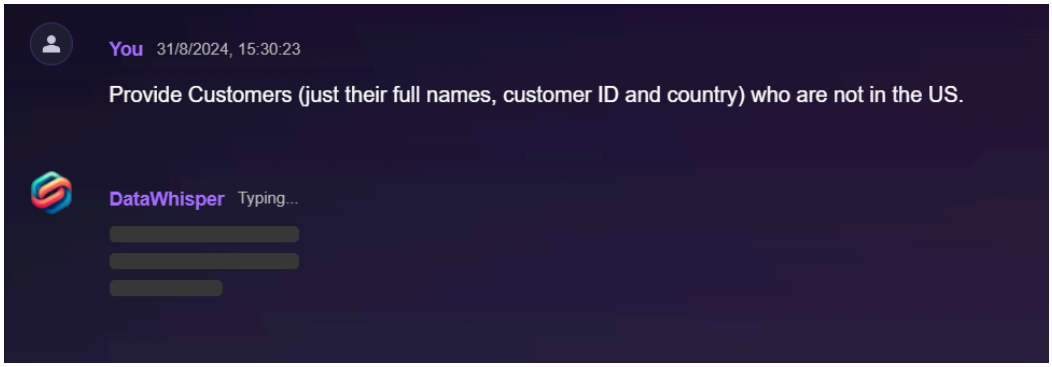

Figura 9. Captura que muestra la implementación del *skeleton*.

Una vez que se recibe la respuesta, se convierte automáticamente en una tabla legible, lo que facilita la interpretación de los resultados por parte del usuario.
Si los *insights* están activados, estos aparecerán como botones interactivos debajo de la respuesta. Estos botones permiten al usuario enviar automáticamente estos *insights* como nuevas consultas al chat simplificando la exploración de datos adicionales sin necesidad de escribir nuevas preguntas, tal y como se observa en la **Figura 10**.
Todos estos detalles pueden observarse en la **Figura 10**.

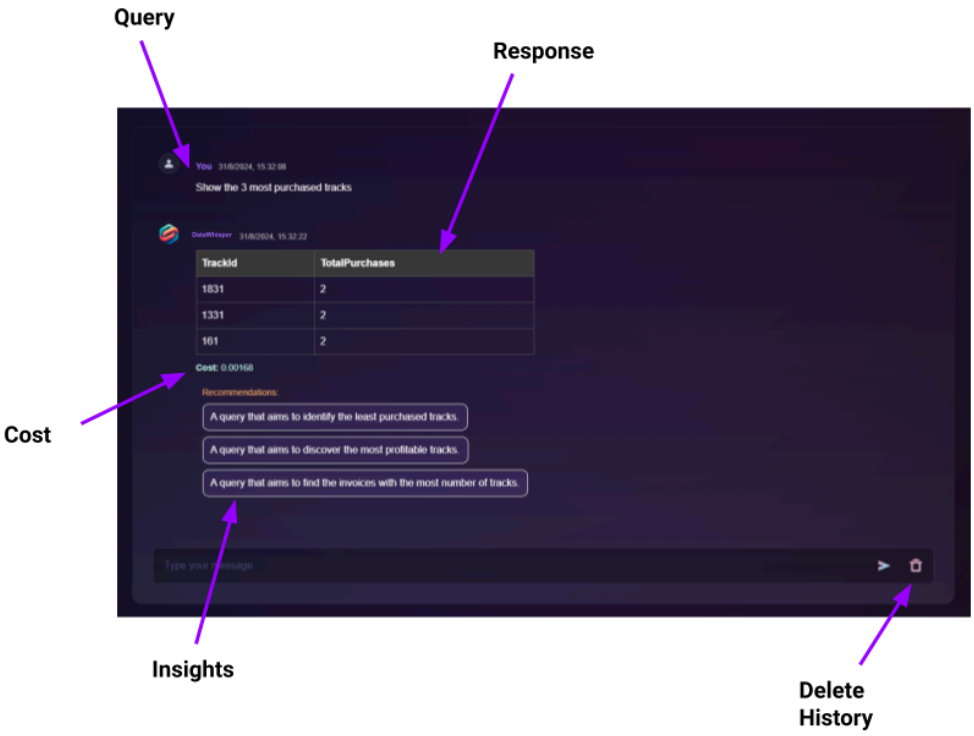

Figura 10. Captura con detalles del componente del Chat.

Existe, además, un botón de "Borrar Chat", el cual permite al usuario limpiar todo el historial de mensajes. Este botón elimina los mensajes del almacenamiento local en el navegador y realiza una solicitud a la API para eliminar los registros correspondientes de la base de datos en el *back-end*. Así pues, todos los mensajes intercambiados en la sesión de chat se guardan en la base de datos SQLite en el *back-end* y se almacenan localmente en el navegador del usuario. Esta doble capa de almacenamiento asegura que el historial de chat permanezca accesible incluso si el usuario navega entre las diferentes páginas de la aplicación, como la página de “Configuración” o la de “Cómo Funciona”, sin tener que recargar la página. Además, la aplicación es completamente reactiva, lo que significa que todas estas operaciones ocurren en tiempo real sin la necesidad de recargar la página.

Configuración: La segunda página de la aplicación es la página de “Configuración” (**Figura 11**), donde el usuario puede personalizar los parámetros y permite ajustar la configuración de las APIs de OpenAI y Hugging Face, así como la conexión a una base de datos relacional, que puede ser cualquier base de datos compatible con SQL.

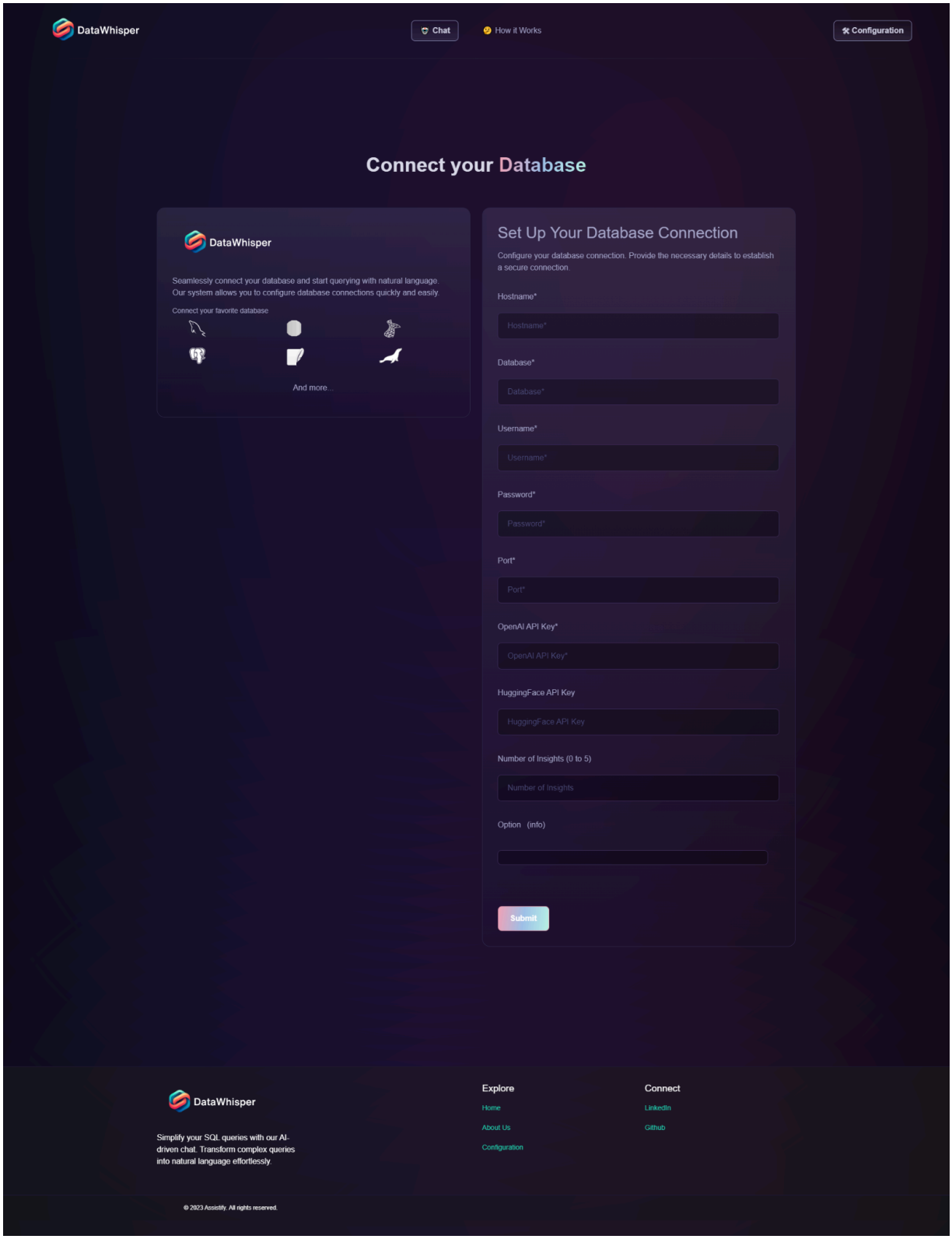

Figura 11. Captura de la página de Configuración.

Un aspecto a destacar de la página es la capacidad de elegir entre las dos opciones diferentes para la selección de tablas relevantes. El usuario también puede configurar el número de *insights* generados por consulta, con opciones que van de 0 a 5 *insights*.
Esta flexibilidad ofrece una adaptabilidad a diferentes escenarios y necesidades, permitiendo a los usuarios optimizar el rendimiento según sus requisitos específicos de coste, tiempo y profundidad de análisis.

Cómo Funciona: Finalmente, la página de "Cómo Funciona" (**Figura 12**) ofrece una explicación sencilla del funcionamiento y sus características principales. Esta sección está diseñada para proporcionar al usuario una comprensión clara.


DataWhisper
Chat
How it Works
Configuration
Simplify Your SQL Queries with AI-Powered Assistance
Human-Like Query Parsing
Translate natural language questions into accurate SQL queries, enabling quick access to the data you need without complex syntax.
Seamless Database Integration
Effortlessly connect to multiple databases, allowing the assistant to pull data from various sources for comprehensive analysis.
AI-Powered Insights
Leverage AI to automatically generate insights and reports, helping you make informed decisions with minimal effort.
Your AI-Driven SQL Query Automation Solution
Multi-Agent Collaboration
Harness the power of multiple intelligent agents working together to automate SQL query generation, execution, and analysis. Each agent specializes in different tasks, ensuring accuracy and efficiency.
Learn more
Insights Generation
Automatically generate additional SQL queries to uncover deeper insights from your data, maximizing the value of your database interactions.
Automatic Table Selection
The system identifies relevant tables for your query either through intelligent agents or using semantic approximation techniques.
Iterative Query Refinement
Agents collaborate to refine and correct queries, ensuring successful execution even in the face of errors or empty results.
Cost-Efficient Operations
Track and optimize the costs of AI operations by monitoring token usage, ensuring that your resources are used effectively.
Adaptive Query Relevance
Our system intelligently assesses the relevance of user prompts for SQL queries. Only the most pertinent prompts trigger database interactions, optimizing performance and reducing unnecessary costs.
You
¿Quien es el usuario que más ha gastado?
DataWhisper
FirstName
LastName
TotalSpent
Helena
Holy
49.62
Cost: 0.002798
Recommendations:
Una consulta que tiene como objetivo descubrir en qué ciudad viven la mayoría de los clientes que más gastan
Intuitive and User-Friendly
Our system offers a seamless and user-friendly interface, making data analysis more accessible than ever. No need for deep technical knowledge—simply input your query, and our intelligent agents handle the rest, as demonstrated in the example on the left.
Whether you're looking to extract insights, run complex queries, or just get quick answers, our intuitive design ensures that anyone on your team can easily navigate and use the platform to its full potential.
DataWhisper
Simplify your SQL queries with our AI-driven chat. Transform complex queries into natural language effortlessly.
Explore
Home
About Us
Configuration
Connect
LinkedIn
Github
© 2023 Acciotty. All rights reserved.

Figura 12. Captura de la página de "Cómo Funciona".

Cada una de las tres páginas principales de la aplicación ha sido diseñada con el usuario en mente, buscando siempre mejorar la experiencia de interacción con la plataforma. Desde la fluidez de la página de Chat, pasando por la flexibilidad de la página de Configuración, hasta la claridad de la página de Cómo Funciona.

6 Conclusiones

El desarrollo y la implementación del sistema propuesto (**Figura 7**), basado en una arquitectura innovadora utilizando agentes para la traducción de consultas en lenguaje natural a consultas SQL, ha alcanzado sus objetivos con éxito. La investigación y los resultados presentados en este trabajo demuestran la viabilidad y eficacia de esta metodología, proporcionando una solución avanzada y eficiente para la conversión automática de consultas en lenguaje natural a SQL, un desafío que sigue siendo relevante en el campo del procesamiento de lenguaje natural y la inteligencia artificial.

Los resultados de las pruebas realizadas con la base de datos Chinook han confirmado la robustez y la precisión del sistema. La evaluación del Agente Proxy, encargado de la identificación de la relevancia de las consultas, ha mostrado una tasa de precisión del 95% (19 de 20 consultas correctamente clasificadas). Este alto nivel de precisión muestra la efectividad del agente en la filtración de consultas irrelevantes y de que aquellas consultas que requieren la generación de SQL sean procesadas.

En cuanto a la selección de tablas, el sistema ha sido probado con dos enfoques distintos: uno basado en el modelo de lenguaje avanzado GPT-3.5 y otro utilizando una aproximación semántica combinada con la técnica de cercanía por coseno. Los resultados obtenidos han sido comparables en ambos métodos, demostrando que, a pesar de sus diferencias, ambos enfoques son igualmente efectivos para identificar las tablas relevantes.

La fase de iteración y corrección de errores, facilitada por los Agentes Analistas, ha mostrado que un proceso iterativo puede mejorar significativamente la precisión de las consultas SQL generadas. Con dos iteraciones, el sistema logró un nivel de precisión del 95% (19 de 20 consultas correctas), lo que indica que el proceso iterativo es altamente efectivo para ajustar y optimizar las consultas.

El análisis de costes y tiempos de ejecución ha proporcionado una visión detallada del sistema bajo diferentes configuraciones. La comparación entre las dos opciones de generación de consultas ha revelado que, aunque la opción basada en el modelo de lenguaje avanzado (Opción 1) puede resultar más costosa en términos de coste directo, ofrece una ventaja significativa en términos de tiempo de ejecución. Por otro lado, la generación de *insights* ha demostrado incrementar en más del doble el coste económico de la consulta, por lo que es algo a tener en cuenta a la hora de interactuar con el sistema, aunque debemos tener en cuenta que durante las pruebas se usó el modelo GPT-4 como Agente Explorer, lo cual usando el modelo GPT-3 bajaría sustancialmente el coste asociado.

La implementación de medidas de seguridad contra ataques de *prompt injection* ha sido efectiva y garantiza que la integridad de la base de datos sea protegida.

La interfaz de usuario desarrollada ha sido diseñada con un enfoque en la usabilidad. La aplicación web, desplegada utilizando Docker para asegurar un entorno de desarrollo coherente y reproducible, permite a los usuarios interactuar con el sistema de manera fluida.

Las funcionalidades ofrecidas, desde la generación de consultas SQL en la página de Chat hasta la configuración personalizada en la página de Configuración, proporcionan una experiencia completa y adaptable.

En conclusión, el sistema basado en agentes para la traducción de consultas en lenguaje natural a SQL ha demostrado ser una solución efectiva y avanzada para abordar el desafío de la generación automática de consultas. La implementación exitosa de esta arquitectura, junto con los resultados positivos en términos de precisión, eficiencia y coste, subraya la viabilidad de este enfoque en el contexto de procesamiento de lenguaje natural y análisis de datos. Los hallazgos obtenidos no solo ofrecen una base sólida para futuras investigaciones y desarrollos en esta área, sino que también proporcionan una herramienta útil para la comunidad de datos y la inteligencia artificial.

Disponibilidad y acceso

Para facilitar el acceso y la replicación del sistema, el código fuente y los componentes del proyecto están disponibles en GitHub.
El enlace al repositorio de GitHub es **https://github.com/pablosierrafernandez/DataWhisper-Arquitectura-basada-en-Agentes-para-text-to-sql**, donde se puede encontrar toda la documentación relevante, el código fuente y las instrucciones de despliegue.

7 Referencias

[1] Elmasri, R. A., & Navathe, S. B. (1989). *Fundamentals of Database Systems*. http://ci.nii.ac.jp/ncid/BA31580259

[2] Rahm, E., & Bernstein, P. A. (2001). A survey of approaches to automatic schema matching. *The VLDB Journal*, *10*(4), 334-350. https://doi.org/10.1007/s007780100057

[3] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., . . . Amodei, D. (2020). Language Models are Few-Shot Learners. *Neural Information Processing Systems*, *33*, 1877-1901. https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[4] Deng, J., Dong, W., Socher, R., Li, L., Li, N. K., & Fei-Fei, N. L. (2009). ImageNet: A large-scale hierarchical image database. *2009 IEEE Conference On Computer Vision And Pattern Recognition*. https://doi.org/10.1109/cvpr.2009.5206848

[5] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017b, junio 12). *Attention is all you need*. arXiv.org. https://arxiv.org/abs/1706.03762

[6] Baktash, J. A., & Dawodi, M. (2023, 4 mayo). *Gpt-4: A Review on Advancements and Opportunities in Natural Language Processing*. arXiv.org. https://arxiv.org/abs/2305.03195

[7] Sutton, R., & Barto, A. (2005). Reinforcement Learning: An Introduction. *IEEE Transactions On Neural Networks*, *16*(1), 285-286. https://doi.org/10.1109/tnn.2004.842673

[8] Zhong, V., Xiong, C., & Socher, R. (2017b, agosto 31). *Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning*. arXiv.org. https://arxiv.org/abs/1709.00103

[9] Wang, H., Li, J., Wu, H., Hovy, E., & Sun, Y. (2023). Pre-Trained Language Models and Their Applications. *Engineering*, *25*, 51-65. https://doi.org/10.1016/j.eng.2022.04.024

[10] Hu, Y., & Lu, Y. (2024, 30 abril). *RAG and RAU: A Survey on Retrieval-Augmented Language Model in Natural Language Processing*. arXiv.org. https://arxiv.org/abs/2404.19543

[11] H2oai. (s. f.). *GitHub - h2oai/sql-sidekick: Experiment on QnA tabular data using LLMs and SQL*. GitHub. https://github.com/h2oai/sql-sidekick

[12] *defog/sqlcoder-7b-2 · Hugging Face*. (s. f.). https://huggingface.co/defog/sqlcoder-7b-2

[13] *NumbersStation/nsql-llama-2-7B · Hugging face*. (s. f.). https://huggingface.co/NumbersStation/nsql-llama-2-7B

[14] Yeadon, W., Peach, A., & Testrow, C. P. (2024, 25 marzo). *A comparison of Human, GPT-3.5, and GPT-4 Performance in a University-Level Coding Course*. arXiv.org. https://arxiv.org/abs/2403.16977

[15] Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., & Radev, D. (2018b, septiembre 24). *Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task*. arXiv.org. https://arxiv.org/abs/1809.08887

[16] Bird-Bench. (s. f.). *GitHub - bird-bench/mini_dev*. GitHub. https://github.com/bird-bench/mini_dev

[17] Cruz, C. J. X. (2024, 12 marzo). *Transforming Competition into Collaboration: The Revolutionary Role of Multi-Agent Systems and Language Models in Modern Organizations*. arXiv.org. https://arxiv.org/abs/2403.07769

[18] Kim, H., So, B., Han, W., & Lee, H. (2020). Natural language to SQL. *Proceedings Of The VLDB Endowment*, *13*(10), 1737-1750. https://doi.org/10.14778/3401960.3401970

[19] BenGWeeks. (s. f.). *GitHub - BenGWeeks/multi-agent-sql-data-analytics: The way we interact with our data is changing.* GitHub. https://github.com/BenGWeeks/multi-agent-sql-data-analytics

[20] *AWS | Servicio de bases de datos relacionales (RDS)*. (s. f.). Amazon Web Services, Inc. https://aws.amazon.com/es/rds/

[21] *Python España*. (s. f.). https://es.python.org/

[22] *python-dotenv*. (2024, 23 enero). PyPI. https://pypi.org/project/python-dotenv/

[23] *tiktoken*. (2024, 13 mayo). PyPI. https://pypi.org/project/tiktoken/

[24] *Installation | AutoGen*. (s. f.). https://microsoft.github.io/autogen/docs/installation/

[25] *Cloud computing services - Amazon Web Services (AWS)*. (s. f.). Amazon Web Services, Inc. https://aws.amazon.com/

[26] *OpenAI API*. (s. f.). OpenAI. https://openai.com/api/

[27] *sentence-transformers/all-MiniLM-L6-v2 · Hugging Face*. (2001, 5 enero). https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2
[28] *sentence-transformers*. (2024, 7 junio). PyPI. https://pypi.org/project/sentence-transformers/
[29] *faiss-cpu*. (2024, 24 junio). PyPI. https://pypi.org/project/faiss-cpu/
[30] *Django*. (s. f.). Django Project. https://www.djangoproject.com/
[31] Christie, T. (s. f.). *Home - Django REST framework*. https://www.django-rest-framework.org/
[32] *Vue.js*. (s. f.). The Progressive JavaScript Framework | Vue.js. https://vuejs.org/
[33] *Vite*. (s. f.). Vitejs. https://vitejs.dev/
[34] *Docker: Accelerated Container Application Development*. (2024, 8 julio). Docker. https://www.docker.com/
[35] Lerocha. (s. f.). *GitHub - lerocha/chinook-database: Sample database for SQL Server, Oracle, MySQL, PostgreSQL, SQLite, DB2*. GitHub. https://github.com/lerocha/chinook-database
[36] Josemguerra. (s. f.). *GitHub - josemguerra/Postgres-sql-Chinook: PostgreSql - Code Institute coursework*. GitHub. https://github.com/josemguerra/Postgres-sql-Chinook/tree/main