.UBAfiuba
FACULTAD DE INGENIERÍA

Informe Técnico

Sistema Multiagente para consultas en Lenguaje Natural

Jonathan Cagua

Juan Pablo Alianak

Github: https://github.com/jpalianak/PLNIII-TpFinal

Tabla de contenido

1. Introducción 4
2. Objetivos 4
3. Arquitectura General 5
   3.1 Componentes principales 5
   3.2 Capa de soporte 6
   3.3 Diagrama de arquitectura 7
4. Flujo de Ejecución 8
   4.1 RouterAgent 8
   4.2 RouteRequestAgent 8
   4.3 Agentes de Dominio: CustomerAgent, OrdersAgent, ProductAgent 8
   4.4 FilterCheckAgent 9
   4.5 FilterConditionAgent 9
   4.6 FuzzFilterAgent 10
   4.7 QueryGenerationAgent 10
   4.8 QueryValidationAgent 10
   4.9 ExecuteSQLAgent 10
   4.10 FinalResponseAgent 11
5. Ejemplo de ejecución 11
6. Detalles de implementación del sistema multiagente 13
   6.1 Flujo General de Datos 13
   6.2 Integración entre Grafo Principal y Subgrafos 14
   6.3 Comunicación entre Agentes 14
   6.4 Uso de LLMs y Plantillas 14
   6.5 Mecanismos de Validación y Seguridad 15
   6.6 Registro y Monitoreo 15
   6.7 Configuración del Sistema 15
   6.8 Interfaz de usuario con Streamlit 16
7. Configuración y personalización 16
   7.1 Configuración de Agentes 17
   7.2 Configuración del Sistema 17

7.3 Configuración de Tablas 18
7.4 Personalización de Plantillas y Prompts 18
7.4 Ajustes Avanzados 18
8. Conclusiones 19

1. Introducción

El presente trabajo desarrolla un sistema multiagente basado en modelos de lenguaje (LLMs) para el procesamiento de consultas expresadas en lenguaje natural sobre bases de datos relacionales. Este proyecto se enmarca en la materia Procesamiento del Lenguaje Natural III, y tiene como finalidad demostrar la aplicación práctica de arquitecturas inteligentes que integran técnicas de procesamiento del lenguaje natural (PLN), razonamiento automatizado y generación dinámica de consultas SQL.

En este contexto, el sistema propuesto está diseñado para interpretar y responder consultas formuladas en lenguaje natural, transformándolas en consultas SQL válidas que se ejecutan sobre una base de datos relacional. Los resultados obtenidos se devuelven posteriormente al usuario en lenguaje natural, garantizando un ciclo completo de comprensión, ejecución y comunicación automatizada.

El enfoque multiagente adoptado permite dividir el procesamiento en unidades especializadas, donde cada agente cumple un rol definido dentro del flujo: enrutamiento de la consulta, extracción de información, validación, generación de SQL y elaboración de la respuesta final. Esta arquitectura modular favorece la escalabilidad, la trazabilidad y la reutilización de componentes, características esenciales en sistemas basados en LLMs.

Finalmente, el desarrollo busca evidenciar cómo la integración de modelos de lenguaje con una arquitectura de agentes cooperativos puede ofrecer soluciones adaptables y seguras para el acceso inteligente a bases de datos, aportando una base sólida para futuras aplicaciones en entornos académicos y empresariales.

2. Objetivos

El propósito de este proyecto es desarrollar un sistema multiagente impulsado por modelos de lenguaje (LLMs) capaz de procesar consultas en lenguaje natural sobre bases de datos relacionales, garantizando precisión, seguridad y flexibilidad en su funcionamiento.

A continuación, se detallan los **objetivos específicos** que orientan el diseño e implementación del sistema:

- Procesamiento de lenguaje natural: Interpretar consultas formuladas por el usuario en lenguaje natural, identificando la intención y los elementos clave mediante el uso de LLMs.

- Arquitectura modular basada en agentes: Dividir el flujo de trabajo en agentes especializados, cada uno responsable de una etapa concreta (enrutamiento, extracción de información, filtrado, generación y validación de consultas SQL, y generación de la respuesta final).
- Validación y seguridad de datos: Implementar mecanismos de guardrails para validar entradas y salidas, evitando inyecciones SQL, respuestas inconsistentes o manipulaciones indebidas de datos.
- Configurabilidad y adaptabilidad: Permitir la personalización del sistema a través de archivos YAML (agents.yaml, system.yaml), que definen dominios, plantillas, tablas y canales de notificación, eliminando la necesidad de modificar el código fuente.
- Notificación multi-canal: Integrar distintos canales de comunicación (correo electrónico, Slack y Telegram) para el envío automatizado de resultados, ampliando la accesibilidad y la interacción con el usuario.

En conjunto, estos objetivos apuntan a construir un sistema inteligente, modular y seguro, capaz de comprender, ejecutar y comunicar consultas complejas sobre bases de datos de manera transparente y eficiente.

3. Arquitectura General

El sistema se construye sobre una arquitectura modular basada en grafos de agentes, implementada con LangGraph. Cada nodo del grafo representa un agente especializado que cumple un rol definido dentro del flujo de procesamiento, desde la comprensión de la consulta del usuario hasta la generación de la respuesta final.

Este enfoque facilita la escalabilidad, la trazabilidad y la integración de nuevos dominios o funciones sin modificar la estructura central.

3.1 Componentes principales

La arquitectura está compuesta por distintos grupos de agentes, organizados según su función dentro del flujo:

- **Router y enrutamiento de consultas**
    - RouterAgent: analiza la consulta del usuario y determina que agentes deben intervenir para responderla.
    - RouteRequestAgent: decide la ruta específica dentro del dominio seleccionado, enviando la consulta al agente de dominio correspondiente.
- **Agentes de dominio**

    - CustomerAgent, OrdersAgent, ProductAgent: dividen la pregunta del usuario en subpreguntas y sobre ellas extraen columnas relevantes de las tablas correspondientes que puedan responderla.
    - FilterCheckAgent: verifica si es necesario aplicar algún filtro para responder la pregunta. Devuelve la columna y el filtro a aplicar.
    - FilterConditionAgent: en función de la salida de FilterCheckAgent decide si ejecutar FuzzFilterAgent o no.
    - FuzzFilterAgent: aplica coincidencias difusas para manejar posibles inconsistencias en los valores de las columnas o filtros.
- **Generación y validación de SQL**
    - QueryGenerationAgent: genera la consulta SQL basada en la intención del usuario, las columnas seleccionadas y los filtros aplicables, siguiendo estrictamente las reglas de la plantilla de prompt.
    - QueryValidationAgent: asegura que la consulta generada sea sintácticamente correcta y compatible con el motor de base de datos (SQLite, SQL Server, etc.).
- **Ejecución y respuesta final**
    - ExecuteSQLAgent: ejecuta la consulta sobre la base de datos y obtiene los resultados.
    - FinalResponseAgent: transforma los resultados de la consulta en lenguaje natural, aplica validación de salida (OutputGuard) y envía notificaciones a los canales configurados.

3.2 Capa de soporte

Además de los agentes principales, el sistema incluye módulos transversales que brindan servicios de soporte y seguridad, garantizando un funcionamiento estable y controlado:

- Guardrails (InputGuard, OutputGuard): validan entradas y salidas para prevenir errores, inyecciones o datos inconsistentes.
- Loaders (engine_loader.py, llm_loader.py): inicializan la conexión a la base de datos y el modelo de lenguaje.
- Logging (logger_config.py): registra eventos de ejecución, errores y trazabilidad de cada agente.

- Configuración vía YAML (agents.yaml, system.yaml): define dominios, tablas, campos y canales de notificación sin necesidad de modificar código.

3.3 Diagrama de arquitectura

La Figura 1 ilustra la arquitectura completa del sistema y las relaciones entre agentes:

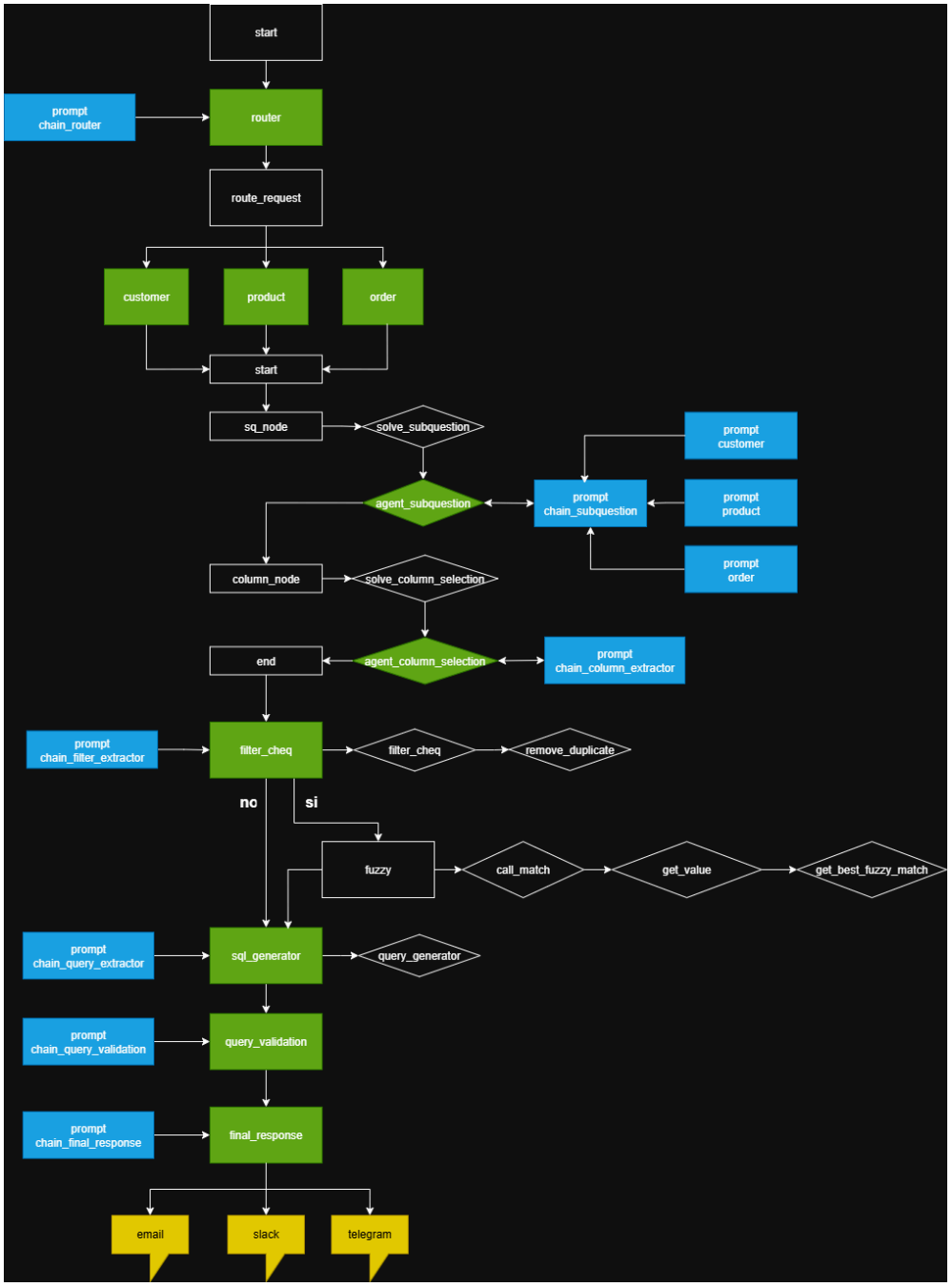

Fig 1: arquitectura del sistema multiagente

La imagen muestra cómo los agentes se conectan en el grafo:

- El flujo inicia en el RouterAgent, que dirige la consulta según el dominio.
- Los agentes de dominio extraen información de tablas específicas.
- Los agentes de filtrado y generación (FilterCheckAgent, FuzzFilterAgent, QueryGenerationAgent, QueryValidationAgent) preparan la query final.
- ExecuteSQLAgent ejecuta la query y FinalResponseAgent genera la respuesta final en lenguaje natural y la envía a los canales configurados.

4. Flujo de Ejecución

El sistema procesa las consultas del usuario mediante un grafo de agentes (StateGraph), donde cada nodo representa un agente especializado.
Cada agente recibe entradas definidas y produce salidas que alimentan los nodos siguientes, asegurando un flujo estructurado, trazable y modular.

A continuación se describen los agentes y su función dentro del flujo principal.

4.1 RouterAgent

- Rol: Determina que agentes deben intervenir para responder la consulta (clients, orders, products).
- Inputs:
  - user_query (consulta del usuario en lenguaje natural)
- Outputs:
  - router_out → lista de agentes que deben intervenir
- Dependencias:
  - Plantilla router.yaml
  - LLM para procesar la intención del usuario
  - RouteRequestAgent para la siguiente etapa del flujo

4.2 RouteRequestAgent

- Rol: Selecciona la ruta exacta dentro del dominio asignado por RouterAgent.
- Inputs:
  - router_out
- Outputs:
  - route_request_out → nombre del agente de dominio asignado: customer, orders o products

4.3 Agentes de Dominio: CustomerAgent, OrdersAgent, ProductAgent

- Rol: Dividen la pregunta del usuario en subpreguntas que pueden responder con las tablas asociadas. Con la información que disponen de cada una de las tablas, extraen las columnas necesarias para poder responder las subpreguntas.
- Inputs:
  - user_query
  - table_list → lista de tablas correspondientes a cada agente, definida en agents.yaml
  - agent_domain y agent_template → parámetros para la ejecución de la cadena LLM
- Outputs:
  - {"customer_out" / "orders_out" / "product_out": response } → contiene columnas y datos relevantes procesados por el subgrafo graph_final
- Dependencias:
  - graph_final (subgrafo de extracción y selección de columnas)
  - EnhancedTokenLogger para trazabilidad de tokens
  - LLM
- Flujo interno del subgrafo:
  - sq_node: genera subpreguntas para cada tabla basada en su descripción (table_description) usando LLM.
    - Output: table_extract (lista de tablas con descripciones relevantes)
  - column_node: selecciona columnas relevantes según la subpregunta y la consulta principal.
    - Output: column_extract (columnas por tabla listas para la generación de SQL)
  - El subgrafo asegura que los agentes entreguen datos consistentes y estructurados al flujo principal.

4.4 FilterCheckAgent

- Rol: Valida en función de la pregunta si es necesario aplicar algún tipo de filtro.
- Inputs:
  - customer_out, orders_out, product_out
- Outputs:
  - filter_check_out → columnas validadas
- Plantilla: filter_check.yaml

4.5 FilterConditionAgent

- Rol: Determina el camino a seguir si es o no necesario aplicar filtros adicionales sobre los datos.
- Inputs:
  - filter_check_out
- Outputs:
  - filter_condition_out → "yes" (aplicar FuzzFilterAgent) o "no" (pasar directamente a QueryGenerationAgent)

4.6 FuzzFilterAgent

- Rol: Aplica coincidencias difusas y ajustes sobre los filtros para mejorar la robustez y consistencia de la consulta.
- Inputs:
  - filter_condition_out
- Outputs:
  - fuzz_filter_out → filtros ajustados

4.7 QueryGenerationAgent

- Rol: Genera la consulta SQL basada en: intención del usuario, columnas seleccionadas y filtros aplicables.
- Inputs:
  - user_query
  - filter_check_out
  - fuzz_filter_out
- Outputs:
  - query_generation_out → SQL preliminar
- Dependencias:
  - Motor de base de datos (engine_type) para ajustar sintaxis SQL
  - Plantilla query_generation.yaml

4.8 QueryValidationAgent

- Rol: Valida la sintaxis de la query y asegura compatibilidad con el motor de base de datos.
- Inputs:
  - query_generation_out
- Outputs:
  - query_validation_out → SQL validada
- Dependencias:
  - Conexión a la base de datos (engine)
  - Plantilla query_validation.yaml

4.9 ExecuteSQLAgent

- Rol: Ejecuta la query SQL sobre la base de datos y obtiene los resultados.
- Inputs:
  - query_validation_out
- Outputs:
  - execute_sql_out → resultados en forma de tabla o DataFrame
- Dependencias:
  - Conexión a la base de datos (engine)

4.10 FinalResponseAgent

- Rol: Convierte los resultados de la consulta en lenguaje natural, valida la salida y envía notificaciones según la configuración del sistema.
- Inputs:
  - user_query
  - execute_sql_out
  - fuzz_filter_out
- Outputs:
  - final_response_out → respuesta final en lenguaje natural
- Dependencias:
  - Guardrails (OutputGuard) para validar la consistencia y formato de la salida
  - Canales de notificación (email, Slack, Telegram)
  - Plantilla final_response.yaml

5. Ejemplo de ejecución

Este ejemplo ilustra cómo el sistema multiagente procesa una consulta en lenguaje natural desde la entrada del usuario hasta la respuesta final. Se muestra el flujo entre agentes, subgrafos, validaciones y generación de resultados.

Consulta de usuario

> User: ¿Me das el id del mayor pedido?

Flujo de procesamiento

1. InputGuard valida la consulta y la limpia de posibles caracteres o secuencias inseguras.
   - Output: user_query seguro para el procesamiento.

2. RouterAgent analiza la intención de la consulta y determina que corresponde al dominio Orders.
    - Output: router_out = ["OrdersAgent"]
3. RouteRequestAgent selecciona la ruta exacta y asigna el agente de dominio.
    - Output: route_request_out = "OrdersAgent"
4. OrdersAgent ejecuta su subgrafo graph_final:
    - sq_node: genera subpreguntas basadas en la descripción de la tabla encabezado_pedidos.
        - Output: table_extract = ["encabezado_pedidos"]
    - column_node: identifica las columnas relevantes: id_pedido, cliente, fecha_pedido, monto_total.
        - Output: column_extract = ["id_pedido", "cliente", "fecha_pedido", "monto_total"]
5. FilterCheckAgent revisa si se deben aplicar filtros adicionales.
    - En este caso, no es necesario.
    - Output: filter_check_out = []
6. FilterConditionAgent evalúa si ejecutar FuzzFilterAgent.
    - Como no hay filtros, el flujo continúa hacia la generación de SQL.
    - Output: filter_condition_out = "no"
7. QueryGenerationAgent construye la SQL preliminar basada en las columnas seleccionadas:

        SELECT id_pedido, cliente, fecha_pedido

        FROM encabezado_pedidos

        ORDER BY monto_total DESC

        LIMIT 1;

    - Output: query_generation_out
8. QueryValidationAgent valida la sintaxis y compatibilidad con el motor SQL (por ejemplo, SQLite o SQL Server).
    - Output: query_validation_out → SQL lista para ejecutar.

9. ExecuteSQLAgent ejecuta la consulta en la base de datos y obtiene los resultados:
    - Output: execute_sql_out = [{"id_pedido": 42, "cliente": "Juan Pérez", "fecha_pedido": "01/10/2025"}]
    - FinalResponseAgent transforma los resultados en lenguaje natural, valida la salida con OutputGuard y envía notificaciones a los canales configurados:
    - Output: final_response_out = "El id del mayor pedido es 42, realizado por el cliente Juan Pérez el 01/10/2025."

6. Detalles de implementación del sistema multiagente

El sistema se basa en una arquitectura modular de agentes conectados mediante grafos de estados (StateGraph), implementada con LangGraph. Cada nodo representa un agente especializado que procesa una parte específica del flujo de trabajo, mientras que la comunicación entre ellos se gestiona a través de un diccionario de estado compartido (state), garantizando trazabilidad, modularidad y escalabilidad.

Un aspecto clave es la base de conocimiento generada automáticamente a partir de la base de datos, representada en un archivo JSON (knowledge.json). Este archivo contiene información detallada sobre las tablas y columnas disponibles, sus descripciones y ejemplos de valores, y es consultado por los agentes para tomar decisiones precisas durante la generación de consultas SQL.

6.1 Flujo General de Datos

El flujo de procesamiento inicia cuando el usuario envía una consulta en lenguaje natural (user_query). A partir de allí:

1. RouterAgent analiza la intención y determina el dominio de la consulta: clientes, pedidos o productos.
2. El agente de dominio seleccionado ejecuta su subgrafo (graph_final) para identificar tablas y columnas relevantes, consultando la base de conocimiento (knowledge.json).
    a. Cada tabla contiene table_description y una lista de columnas (columns) con su name, datatype, description y sample_values.
    b. Los agentes usan esta información para:
        i. Determinar qué columnas son relevantes para la consulta.

    ii. Construir prompts más precisos para el LLM.
    iii. Validar que las columnas y tablas consultadas existen en la base de conocimiento.
3. Los resultados se consolidan en el diccionario state y se envían a los agentes de filtrado: FilterCheckAgent y FuzzFilterAgent.
4. QueryGenerationAgent genera la consulta SQL según columnas seleccionadas y filtros aplicables.
5. QueryValidationAgent valida la sintaxis y compatibilidad con el motor SQL configurado.
6. ExecuteSQLAgent ejecuta la consulta y obtiene los resultados.
7. FinalResponseAgent interpreta los resultados, genera la respuesta en lenguaje natural y envía notificaciones a los canales externos configurados (Slack, Telegram, email).

Este enfoque modular permite incorporar nuevos agentes o dominios sin modificar la estructura central del grafo principal.

6.2 Integración entre Grafo Principal y Subgrafos

El grafo principal (build_main_graph) orquesta la ejecución general del sistema:

- Los agentes de dominio (CustomerAgent, OrdersAgent, ProductAgent) ejecutan un subgrafo independiente (graph_final) para manejar la extracción de tablas y selección de columnas.
- El subgrafo consta de dos etapas principales:
    - sq_node: genera subpreguntas basadas en las descripciones de las tablas obtenidas del knowledge.json.
    - column_node: selecciona columnas relevantes para la consulta principal.
- La base de conocimiento asegura que las columnas y tablas usadas por los agentes existan realmente, que los tipos de datos sean consistentes y que los filtros aplicados sean válidos según los valores de ejemplo (sample_values).

6.3 Comunicación entre Agentes

Cada agente escribe su salida en el diccionario state bajo una clave {nombre_del_agente}_out y consulta simultáneamente el knowledge.json para:

- Verificar existencia de columnas y tablas.
- Construir prompts para LLMs usando descripciones y tipos de datos.
- Validar filtros y condiciones aplicadas según los valores posibles de cada columna (sample_values).

Ejemplos de claves en state:

- router_out → salida de RouterAgent.
- customer_out → salida de CustomerAgent.
- query_generation_out → SQL generada.

6.4 Uso de LLMs y Plantillas

Cada agente ejecuta un LLM mediante una cadena que incluye:

- Carga de plantilla YAML especializada.
- Inclusión de información del knowledge.json sobre tablas y columnas relevantes.
- Parser de salida específico (por ejemplo, StrOutputParser).

Ventajas:

- Consistencia y formato uniforme de las respuestas.
- Reducción de errores de columnas inexistentes.
- Prompts más precisos y validados por la base de conocimiento.
- Fácil mantenimiento y personalización por dominio.

6.5 Mecanismos de Validación y Seguridad

El sistema incorpora guardrails para verificar entradas y salidas:

- InputGuard: asegura que las consultas del usuario no contengan contenido inseguro o inyectable.
- OutputGuard: valida que las salidas del agente final respeten el formato y la consistencia esperados.

Además, la base de conocimiento funciona como una capa adicional de validación, evitando que los agentes generen SQL con tablas o columnas inexistentes.

6.6 Registro y Monitoreo

El módulo EnhancedTokenLogger registra los tokens consumidos por cada agente, siguiendo el flujo jerárquico del procesamiento (agente → subagente → subproceso) y almacenando los logs tanto en consola como en archivo. Cada agente actualiza su contexto mediante current_agent.set(), lo que facilita la auditoría y la trazabilidad de la ejecución.

6.7 Configuración del Sistema

La configuración se define mediante archivos YAML:

- agents.yaml: describe los agentes, dominios y tablas asignadas.

- system.yaml: determina el tipo de conexión (local o cloud), canal de notificación, ubicación de la base de datos y ubicación de la base de conocimiento.
- tables.yaml: describe los campos dentro de cada tabla.

La base de conocimiento (knowledge.json) se genera automáticamente a partir de la base de datos y es consultada por todos los agentes para obtener la descripción de tablas y columnas, Validar filtros y datos y construir prompts más precisos para la generación de SQL.

Esta arquitectura permite modificar parámetros o la lógica del sistema sin reescribir código, garantizando flexibilidad y adaptabilidad.

6.8 Interfaz de usuario con Streamlit

El sistema incluye una interfaz web interactiva desarrollada con Streamlit, que permite a los usuarios enviar consultas en lenguaje natural y visualizar los resultados generados por los agentes.

Funcionamiento básico:

1. El usuario abre la aplicación Streamlit (streamlit run app.py).
2. Se muestra un campo de texto para ingresar la consulta (user_query).
3. Al enviar la consulta, el backend de agentes procesa la información siguiendo el flujo descrito en 6.1–6.7.
4. El flujo de la información en cada agente puede verse en la interfaz.
5. Los resultados se presentan en pantalla con un resumen de tiempo y consumo de tokens además de la respuesta final.

En la figura 2 podemos ver una imagen de la interfaz en Streamlit

Fig 2: interfaz en Streamlit

7. Configuración y personalización

El sistema multiagente está diseñado para ser altamente configurable y adaptable a distintos dominios o escenarios, sin necesidad de modificar el código fuente. La configuración central se realiza mediante archivos YAML que permiten definir agentes, plantillas, tablas y parámetros del sistema. Esto facilita la personalización de la arquitectura y su integración con diferentes bases de datos y canales de notificación.

7.1 Configuración de Agentes

Cada agente se configura en el archivo agents.yaml, donde se define su dominio, plantilla y tablas asignadas. Por ejemplo, un agente de dominio de clientes podría configurarse de la siguiente manera:

```yaml
agents:
  router:
    domain: general
    template_file: router

  order:
    domain: specific
    template_file: order
    table_list:
      - encabezado_pedidos
      - items_pedidos

  product:
    domain: specific
    template_file: product
    table_list:
      - productos

  customer:
    domain: specific
    template_file: customer
    table_list:
      - clientes
```

En este archivo se pueden incluir todos los agentes del sistema, incluyendo agentes de dominio, agentes de filtrado, de generación de SQL y de respuesta final. Además, se pueden definir parámetros adicionales como fuentes de datos o plantillas específicas para cada agente, lo que permite modificar la lógica de procesamiento de manera flexible.

7.2 Configuración del Sistema

El archivo system.yaml permite ajustar aspectos globales del sistema, tales como el tipo de conexión a la base de datos (local o en la nube) y el canal de notificación predeterminado para las respuestas finales. Por ejemplo:

```yaml
# Tipo de conexion del sistema: local o en la nube
system:
  connection_type: local   # opciones: local / cloud
  notification_channel: "none"  # opciones para enviar el mensaje de salida: email / slack / telegram / none

# Ubicación de la base de datos local
local_database:
  path: "data/db/fake_db.sqlite"

# Ubicación de la base de conocimiento
knowledge_base:
  path: "src/knowledge/knowledge.json"
```

Estos parámetros determinan cómo se conecta el sistema al motor de base de datos y si las respuestas generadas por FinalResponseAgent deben enviarse automáticamente a canales como email, Slack o Telegram.

7.3 Configuración de Tablas

El archivo **tables.yaml** define las tablas disponibles, sus nombres lógicos, descripciones y mapeo de columnas físicas a nombres lógicos usados por los agentes.

```yaml
tables:
  SB1010:
    logical_name: productos
    description: "Contiene el id del producto, la descripción, el tipo y la familia del producto."
    columns:
      B1_COD: producto_id
      B1_DESC: descripcion
      B1_TIPO: tipo
      B1_FAMILIA: familia

  SC5010:
    logical_name: encabezado_pedidos
    description: "Contiene los datos relacionados con el pedido de venta, incluyendo el id del pedido, el cliente, el vendedor, la fecha de emisión, la u
    columns:
      C5_NUM: pedido_id
      C5_CLIENTE: cliente_id
      C5_VEND1: vendedor_id
      C5_EMISSAO: fecha_emision
      C5_NATUREZ: unidad_negocio
      C5_TXMOEDA: tasa_moneda

  SC6010:
    logical_name: items_pedidos
    description: "Contiene el detalle de los artículos del pedido de venta. Se relaciona con encabezado_pedidos a través de pedido_id. Contiene los artícu
    el id del pedido, el ítem, el producto, la cantidad vendida, la descripción del producto, el precio de venta, el valor total y la fecha de entrega."
```

7.4 Personalización de Plantillas y Prompts

Cada agente utiliza plantillas (.yaml) que definen la instrucción para el modelo de lenguaje y el formato de salida esperado. Estas plantillas pueden personalizarse para distintos dominios o necesidades específicas, permitiendo:

- Ajustar la forma en que se extraen columnas y tablas relevantes.
- Definir la estructura de la SQL generada.
- Modificar el lenguaje y formato de las respuestas finales para diferentes audiencias.

Esta personalización asegura consistencia y trazabilidad, y permite extender el sistema a nuevos dominios o tipos de consultas sin alterar la arquitectura principal.

7.4 Ajustes Avanzados

El sistema también permite configurar parámetros más técnicos, como:

- Tipos de motores de base de datos soportados (SQLite, SQL Server, etc.).
- Opciones de logging y nivel de detalle de los registros.
- Guardrails específicos para entradas y salidas, ajustando los validadores de seguridad según las necesidades del proyecto.
- Reutilización de subgrafos (graph_final) para nuevos agentes de dominio o escenarios de prueba.

Estas opciones brindan flexibilidad completa y facilitan la escalabilidad del sistema sin comprometer la seguridad ni la consistencia de los datos.

8. Conclusiones

El sistema multiagente desarrollado demuestra la eficacia de combinar modelos de lenguaje (LLMs) con arquitecturas modulares basadas en grafos de agentes para procesar consultas en lenguaje natural sobre bases de datos relacionales. Cada agente cumple un rol específico dentro del flujo de procesamiento, desde el enrutamiento inicial de la consulta hasta la generación de la respuesta final en lenguaje natural. Los subgrafos, como graph_final, permiten reutilizar la lógica de extracción de tablas y columnas en distintos dominios sin modificar la arquitectura central.

La implementación de mecanismos de validación y seguridad, mediante InputGuard y OutputGuard, asegura que tanto las entradas como las salidas sean consistentes y seguras, evitando inyecciones o errores en el flujo de procesamiento. Al mismo tiempo, el EnhancedTokenLogger proporciona trazabilidad completa del proceso, registrando cada paso, desde la consulta inicial hasta la respuesta final, incluyendo subprocesos internos de los agentes. Esto facilita auditoría y monitoreo en tiempo real.

El sistema también destaca por su flexibilidad y configurabilidad. Gracias a la centralización de la configuración en archivos YAML (agents.yaml, system.yaml, tables.yaml), es posible ajustar agentes, plantillas, tablas y parámetros globales sin modificar el código fuente. Esto permite adaptar rápidamente el sistema a distintos motores de base de datos y a diversos canales de notificación como correo electrónico, Slack o Telegram.

En cuanto a los resultados, la combinación de LLMs con reglas de plantillas y validadores asegura que las consultas sean interpretadas correctamente y que

las respuestas entregadas sean coherentes y comprensibles. El sistema es capaz de interpretar consultas complejas, transformarlas en SQL válido, ejecutarlas sobre la base de datos y generar respuestas precisas en lenguaje natural, manteniendo seguridad, trazabilidad y facilidad de mantenimiento.

En conjunto, este proyecto evidencia que un enfoque modular basado en agentes especializados, potenciado por LLMs, ofrece una solución escalable, confiable y adaptable a nuevos dominios o aplicaciones empresariales, demostrando la efectividad de la integración entre inteligencia artificial y arquitecturas de software modular.