.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

El presente trabajo desarrolla un sistema multiagente basado en modelos de lenguaje (LLMs) para procesar consultas expresadas en lenguaje natural sobre bases de datos relacionales.

El proyecto, enmarcado en la materia *Procesamiento del Lenguaje Natural III*, busca demostrar la integración de técnicas de procesamiento del lenguaje natural (PLN), razonamiento automatizado y generación dinámica de consultas SQL dentro de una arquitectura modular e inteligente.

El sistema propuesto permite que un usuario formule preguntas en lenguaje natural, las cuales son interpretadas y transformadas en consultas SQL válidas que se ejecutan sobre la base de datos. Los resultados son luego devueltos en lenguaje natural, completando un ciclo automatizado de comprensión, ejecución y comunicación.

La arquitectura divide el proceso en agentes especializados responsables de etapas específicas como el enrutamiento de la consulta, la extracción de información relevante, la validación, la generación del SQL y la elaboración de la respuesta final. Este enfoque modular aporta escalabilidad, trazabilidad y reutilización de componentes, elementos fundamentales en sistemas basados en LLMs.

Además, el sistema incorpora una base de conocimiento generada automáticamente a partir de la estructura real de la base de datos. Este archivo describe tablas, columnas, tipos de datos y ejemplos de valores, lo que permite a los agentes razonar con información precisa y coherente, reduciendo errores durante la generación de consultas SQL.

Finalmente, se incluye una interfaz web desarrollada con Streamlit, que permite a los usuarios interactuar con el sistema de manera intuitiva, visualizar el flujo de procesamiento y observar los resultados generados por los agentes en tiempo real.

2. Objetivos

El desarrollo de este sistema multiagente tiene como propósito principal demostrar cómo los modelos de lenguaje pueden integrarse de manera efectiva en entornos empresariales, permitiendo la interacción natural entre usuarios y bases de datos estructuradas.

2.1 Objetivo general

Diseñar e implementar un sistema multiagente impulsado por LLMs capaz de interpretar consultas en lenguaje natural, generar consultas SQL correctas y

devolver respuestas interpretables, basándose en una base de conocimiento dinámica generada a partir de la estructura real de la base de datos.

2.2 Objetivos específicos

Para alcanzar este objetivo general, se plantean las siguientes metas técnicas y funcionales:

- Diseñar una arquitectura multiagente donde cada agente cumpla un rol específico dentro del flujo de procesamiento (análisis de intención, extracción de entidades, generación de SQL, validación y presentación de resultados).
- Automatizar la generación de la base de conocimiento, creando un archivo estructurado (JSON) que describa las tablas, columnas, relaciones y ejemplos de datos de la base, facilitando el razonamiento de los agentes.
- Entrenar y configurar los agentes para que colaboren y se comuniquen entre sí de forma coherente, minimizando errores de interpretación y redundancias.
- Desarrollar una interfaz interactiva con Streamlit, que permita al usuario escribir consultas en lenguaje natural, visualizar las respuestas y analizar el flujo interno de agentes y decisiones.
- Integrar la ejecución real de consultas SQL sobre una base de datos (ej. SQL Server o SQLite) garantizando la correspondencia entre el lenguaje natural y los resultados estructurados.
- Validar el sistema mediante pruebas controladas, evaluando precisión, consistencia y eficiencia en las respuestas.
- Documentar y analizar los resultados obtenidos, destacando los aportes y limitaciones del enfoque multiagente en comparación con soluciones monolíticas.

3. Arquitectura general

El sistema multiagente propuesto se compone de varios módulos interconectados que colaboran para interpretar y responder consultas en lenguaje natural. La arquitectura está diseñada para ser modular, escalable y fácilmente configurable, permitiendo adaptar cada componente a distintos entornos o bases de datos.

3.1 Componentes principales

La estructura general se organiza en torno a un grafo principal de agentes (MainGraph) que coordina las tareas de comprensión, generación de consultas y validación.

Cada agente se especializa en un rol dentro del proceso, favoreciendo la trazabilidad y el control de calidad de las respuestas.

Principales componentes:

- RouterAgent: determina qué tipo de consulta está realizando el usuario (por ejemplo, sobre clientes, productos o pedidos).
- RouteRequestAgent: canaliza la solicitud hacia el subgrafo correspondiente al dominio identificado.
- Agentes de dominio: como *CustomerAgent*, *OrdersAgent* y *ProductAgent*, que contienen las reglas, plantillas y prompts específicos para su contexto de datos.
- Agentes de filtrado y validación:
    - *FilterCheckAgent* y *FilterConditionAgent* determinan si la consulta requiere filtros adicionales.
    - *FuzzFilterAgent* refina las condiciones de búsqueda para mejorar la precisión.
    - *QueryValidationAgent* garantiza que la consulta SQL generada sea sintáctica y semánticamente válida.
- QueryGenerationAgent: construye la consulta SQL a partir de la información estructurada de la base de conocimiento.
- ExecuteSQLAgent: ejecuta la consulta directamente sobre la base de datos.
- FinalResponseAgent: formatea y presenta el resultado en lenguaje natural, listo para mostrarse en la interfaz de usuario.

3.2 Capa de soporte

Además del flujo principal de agentes, el sistema cuenta con una capa de soporte que provee infraestructura y utilidades para su funcionamiento:

- Módulo de configuración dinámica (config_system.yaml, tables.yaml y config_agents.yaml): define los parámetros operativos del sistema y de cada agente.
- Gestor de conocimiento (knowledge.json): contiene la representación estructurada de la base de datos (tablas, columnas, tipos de datos, descripciones y ejemplos). Este archivo es generado automáticamente por un script que analiza la base real, asegurando que los agentes trabajen sobre una fuente actualizada y confiable.
- Sistema de logging y monitoreo: registra las decisiones y resultados de los agentes, permitiendo auditoría y análisis de desempeño.
- Interfaz de usuario con Streamlit: ofrece un entorno visual para ejecutar consultas, mostrar los resultados y seguir el flujo interno de los agentes de manera transparente.

3.3 Diagrama de arquitectura

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

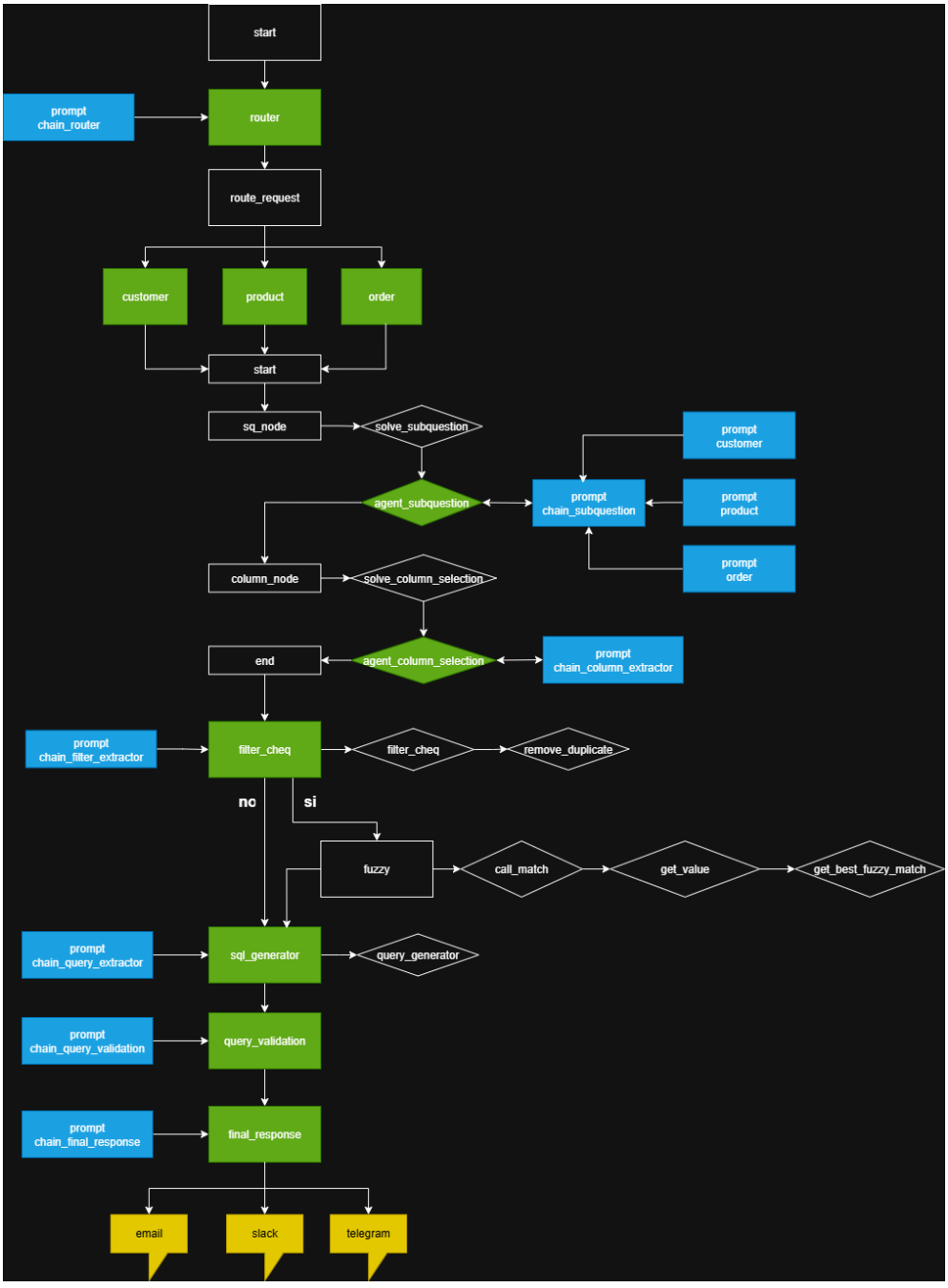

Fig 1: arquitectura del sistema multiagente.

En la figura se observa la arquitectura del sistema multiagentes como está definido actualmente. Al ser modular la arquitectura, es simple agregar agentes específicos al sistema para ampliar las capacidades de respuestas.

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: divide la pregunta principal en subpreguntas que puedan ser respondidas con las tablas asociadas basada en su descripción (table_description) usando LLM.
        - Output: table_extract (lista de tablas con descripciones relevantes)
    - column_node: selecciona columnas relevantes de las tablas asociadas para responder la subpregunta.
        - 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 mayor de un 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 = [ID máximo de pedido, "encabezado_pedidos"]
    - column_node: identifica las columnas relevantes: id_pedido, cliente, fecha_pedido, monto_total.
        - Output: column_extract = ["id_pedido", "Identificador único del pedido de venta (formato con ceros a la izquierda)", ……]
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 MAX(id_pedido) max_id_pedido

        FROM encabezado_pedidos;

    - 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": 000200}]
    - 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 de pedido más alto registrado es 000200."

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.

6.1 Base de conocimiento

El archivo knowledge.json contiene información detallada sobre las tablas y columnas disponibles en la base de datos, incluyendo:

- Descripciones de tablas y columnas (table_description y columns)
- Tipos de datos (datatype)
- Valores de ejemplo (sample_values)

Los agentes consultan esta base de conocimiento para:

- Validar la existencia de columnas y tablas antes de generar SQL
- Construir prompts más precisos para los LLMs
- Garantizar filtros y condiciones consistentes con los datos reales

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.
- Etapas principales del subgrafo:
    - sq_node: genera subpreguntas basadas en las descripciones de las tablas
    - column_node: selecciona las columnas relevantes para la consulta principal

Esto asegura que los datos utilizados sean consistentes y que la lógica de selección de columnas se pueda reutilizar en distintos dominios.

6.3 Comunicación entre agentes

Cada agente escribe su salida en el diccionario state bajo claves específicas ({nombre_del_agente}_out) y consulta simultáneamente el knowledge.json para:

- Verificar existencia de columnas y tablas
- Construir prompts precisos para los LLMs
- Validar filtros y condiciones según los valores posibles de cada columna

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 YAML

Cada agente ejecuta un LLM mediante plantillas YAML especializadas que incluyen:

- Información del knowledge.json sobre tablas y columnas
- Instrucciones y formato de salida esperado
- Parsers específicos para interpretar la respuesta del modelo

Ventajas:

- Consistencia y formato uniforme de las respuestas
- Reducción de errores de columnas inexistentes
- Mantenimiento y personalización más sencillos por dominio

6.5 Mecanismos de validación y seguridad

El sistema incorpora guardrails para 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 consistencia esperados

La base de conocimiento también funciona como capa adicional de validación, evitando la generación de 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 (agente → subagente → subproceso) y almacenando los logs en consola y archivo.

Cada agente actualiza su contexto mediante current_agent.set(), facilitando auditoría y trazabilidad en tiempo real.

6.7 Configuración del sistema

La configuración se realiza mediante archivos YAML:

- agents.yaml: define agentes, dominios y tablas asignadas
- system.yaml: determina tipo de conexión, canal de notificación y ubicación de la base de datos
- tables.yaml: describe los campos de cada tabla

Esto permite modificar parámetros y lógica del sistema sin reescribir código, garantizando flexibilidad y adaptabilidad a distintos dominios o motores de base de datos.

6.8 Interfaz de usuario con Streamlit

El sistema incluye una interfaz web interactiva desarrollada con Streamlit:

1. El usuario abre la aplicación (streamlit run main_streamlit.py)
2. Se presenta un campo de texto para ingresar consultas en lenguaje natural (user_query)
3. Al enviar la consulta, el backend de agentes procesa la información siguiendo el flujo del sistema
4. Los resultados se presentan en pantalla, incluyendo:
    - Respuesta final
    - Consumo de tokens
    - Tiempo de ejecución

Esta interfaz permite interactuar con el sistema de forma amigable y visualizar en tiempo real el procesamiento de cada agente.

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 modificar el código fuente. La centralización de la configuración en archivos YAML permite definir agentes, plantillas, tablas y parámetros del sistema de forma clara y flexible.

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. Resultados

8.1 Ejecución de consultas

Se realizaron pruebas enviando distintas consultas en lenguaje natural, evaluando la capacidad del sistema para:

- Interpretar correctamente la intención del usuario.
- Seleccionar las tablas y columnas adecuadas.
- Generar SQL válida y compatible con el motor de base de datos.
- Devolver resultados precisos y coherentes en lenguaje natural.

Ejemplo de consulta:

Usuario: ¿Cuál es el id más alto de un pedido?

Flujo observado: Router → OrdersAgent → QueryGenerationAgent → ExecuteSQLAgent → FinalResponseAgent

Resultado: "El id de pedido más alto registrado es 000200."

Las preguntas generadas fueron con intención de activar distintos agentes principales, forzar a que deban hacer join de la información que cada uno de ellos posee, etc.

8.2 Evaluación de seguridad

Se verificó que las consultas con intentos de inyección SQL fueran bloqueadas por InputGuard, garantizando que el sistema no ejecute operaciones inseguras. De la misma manera, OutputGuard asegura la consistencia de la salida.

8.3 Métricas de desempeño

Las métricas de desempeño son fuertemente dependiente de la pregunta del usuario, de la cantidad de agentes que se activan para poder responderla, del modelo en cada uno de los agentes utilizado (en todas las pruebas se utilizó el modelo gpt-5-nano), de los prompts, etc.

- Tiempo promedio de respuesta: 60 s
- Tokens promedios consumidos por prompts: 10000
- Tokens promedios consumidos por respuesta: 25000
- Tokens promedios consumidos por consulta: 35000
- Porcentaje de consultas simples (un solo agente involucrado) correctamente interpretadas: 95%
- Porcentaje de consultas complejas (múltiples agentes involucrados) correctamente interpretadas: 70%

Estos resultados indican que hay margen de mejora, tanto en la optimización de la arquitectura como en la elección de modelos y el refinamiento de los prompts. Se trata de un camino de desarrollo por recorrer.

9. 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.