MADRID, 5 DE MARZO DE 2024

OMMA

Agente LLM de calidad de datos para integración operacional

Aplicación generativa para creación de reglas de calidad

JUAN RAMON GONZALEZ
CTO EN MÁTICA PARTNERS

1. OMMA: una introducción

OMMA es una herramienta avanzada de **calidad de datos** diseñada para **facilitar la creación y gestión de políticas de calidad mediante interfaces no-code.** Desde su concepción, OMMA ha buscado empoderar a los usuarios permitiéndoles **definir reglas** de calidad de datos **sin necesidad de tener un conocimiento técnico exhaustivo.** Este enfoque se basa en la premisa de que el conocimiento profundo de los datos es más crucial que la habilidad de programación para garantizar la integridad y la precisión de los datos.

OMMA simplifica la tarea de crear reglas de calidad de datos, que tradicionalmente se hacía de forma manual y técnica, lo que las hacía difíciles y propensas a errores. Con una plataforma intuitiva, OMMA permite a los usuarios crear políticas basadas en su conocimiento de los datos, haciendo el proceso más fácil y eficiente.

2. Introducción a la integración de LLMs para crear reglas con lenguaje natural en OMMA

Actualmente estamos inmersos en la aplicación de LLMs en multitud de tareas. Pero como todas las nuevas tecnologías, se encuentra en un proceso de evolución, estandarización y madurez. Hoy en día, la gran mayoría de casos de uso que encontramos están basados en la creación de chatbots, de RAGs y similares, pero la aplicación de LLMs en otro tipo de casos prácticos no está todavía tan explotada o documentada.

La **integración de LLMs en procesos productivos**, donde áreas o procedimientos completos puedan ser sustituidos por un interfaz más ágil y amigable para el usuario, supone una gran transformación en la forma de interactuar con sistemas, como lo son los de gestión de datos.

En **Grupo Mática** contamos con una gran ventaja gracias a OMMA, que es nuestro producto propio. Con OMMA podemos crear, testear y desplegar funcionalidades productivas completas, como la integración de LLMs en la creación de políticas de calidad. Esto supone un paso adelante significativo, permitiéndos establecer reglas mediante el uso de lenguaje natural.

Esta integración tiene el potencial de transformar completamente el proceso de definición de políticas de calidad de datos. **Mediante el uso de LLMs, los usuarios de OMMA podrán formular reglas y directrices en lenguaje natural**, que luego serán interpretadas y ejecutadas por el sistema. Esto no solo simplifica el proceso, haciéndolo más accesible para usuarios no técnicos, sino que también acelera el tiempo de implementación y reduce la posibilidad de errores.

La problemática que enfrentamos con esta integración radica en la necesidad de transformar reglas y políticas de calidad de datos que tradicionalmente se han definido de manera manual y técnica, en directrices comprensibles y aplicables mediante lenguaje natural. Esta transformación no solo hará que el proceso sea más accesible, sino que también mejorará la eficiencia y precisión en la gestión de calidad de datos.

En este apartado, exploraremos en detalle los desafíos y oportunidades que conlleva la integración de LLMs en OMMA. Analizaremos cómo las LLMs pueden interpretar y generar reglas de calidad de datos de manera eficiente y precisa, mejorando así la gobernanza de datos. Además, discutiremos las metodologías y enfoques técnicos que hemos adoptado para asegurar que esta integración no solo sea efectiva, sino también escalable y segura.

2.1 Flujo de Creación de Reglas en OMMA

OMMA ha sido diseñado para **simplificar la creación y gestión de políticas de calidad de datos** a través de un proceso intuitivo **y sin necesidad de conocimientos técnicos avanzados.**

El sistema se basa en un **formulario de tres pasos** que permite a los usuarios definir reglas de calidad de datos de manera eficiente y precisa.

- **Basic Rule Data:** En este primer paso, el usuario **selecciona el dataset** sobre el que desea aplicar sus políticas de calidad. Decide el tipo de regla que quiere implementar y le da un nombre descriptivo. Además, elige las columnas específicas del conjunto de datos en las que se aplicará la regla. El usuario también decide aquí si quiere trabajar con el dataset en su forma original o si prefiere aplicar agregaciones o joins para enriquecer los datos. **Este paso inicial establece las bases de la regla y asegura su aplicación en los datos pertinentes.**

- **Specific Rule Data:** En el segundo paso, se **configura el comportamiento específico de la regla seleccionada**. Dependiendo del tipo de regla, el usuario define parámetros detallados. Por ejemplo, si se elige una regla de expresión regular, se introduce la expresión regular correspondiente. Si la regla seleccionada es de rangos de valores, se especifican los límites de los rangos. **Este paso permite una personalización detallada de la regla, asegurando que se ajuste exactamente a las necesidades del usuario y a la naturaleza del dataset.**

- **Execution Conditions:** En el último paso, el usuario **establece las condiciones bajo las cuales la regla se ejecutará**. Esto implica definir qué condiciones deben cumplir los datos para que la regla sea aplicada. Por ejemplo, se pueden especificar condiciones basadas en fechas, valores específicos en otras columnas, o cualquier otro criterio relevante. **Esta etapa final garantiza que las reglas de calidad se apliquen de manera precisa y contextualmente adecuada.**

Este proceso de tres pasos en OMMA es directo y no requiere conocimientos técnicos avanzados, sino un profundo entendimiento de los datos y del tipo de validación que se desea implementar.

Plantearnos modificar este comportamiento sustituyéndolo por una LLM requiere que el sistema sea capaz de imitar este comportamiento de creación de reglas.

2.2 Requerimientos de Aplicación Generativa para OMMA

Para crear un sistema basado en IA Generativa que sustituya los formularios de creación de reglas en OMMA, es necesario asegurar que el flujo de trabajo sea eficiente y replique el proceso actual, y que los outputs o resultados generados sean estructurados y confiables. A continuación, se detallan los componentes y requisitos clave para alcanzar este objetivo.

Definición del flujo de trabajo: pasos a seguir

El sistema debe seguir un flujo nido que permita a los usuarios crear reglas de calidad de datos de manera estructurada y secuencial. Este flujo se compone de los siguientes pasos:

- **Creación de Joins (Opcional):** El sistema debe permitir a los usuarios especificar si desean realizar joins con otras tablas en la base de datos. Esto implica identificar y seleccionar las tablas relevantes y definir las condiciones de join.
- **Creación de Agregaciones (Opcional):** Los usuarios pueden optar por aplicar agregaciones a los datos. El sistema debe ofrecer opciones para seleccionar los tipos de agregaciones (sumas, promedios, etc.) y las columnas sobre las que se aplicarán.
- **Definición de Reglas:** Este es el paso obligatorio en el que los usuarios especifican el tipo de regla que desean crear (por ejemplo, regex, value ranges) y proporcionan los parámetros específicos para cada tipo.
- **Configuración de Condiciones de Ejecución (Opcional):** El sistema debe permitir la configuración de condiciones bajo las cuales las reglas se ejecutarán, asegurando que las reglas solo se apliquen a los datos relevantes.

Structured Output

Para asegurar la confiabilidad del sistema, es vital que la información manejada por el modelo de lenguaje sea estructurada. Esto implica que el output generado por la IA debe tener un formato predefinido que incluya todos los componentes necesarios: agregación, join, listado de reglas y condiciones de ejecución. Y este ha sido uno de los puntos críticos. Hay bastante documentación de cómo implementar chains con salidas estructuradas, e incluso agentes sencillos, pero cuando entramos en sistemas de agentes complejos, como veremos más adelante, la solución técnica se complica.

Componentes del Objeto de Respuesta

El objeto de respuesta del sistema debe estar compuesto de los siguientes elementos:

- **Joins**: Información sobre las tablas a unir y las condiciones de join.
- **Agregaciones**: Detalles sobre los tipos de agregaciones y las columnas sobre las que se aplican.
- **Listado de Reglas**: Definición de las reglas específicas, incluyendo el tipo de regla y sus parámetros.
- **Condiciones de Ejecución**: Condiciones bajo las cuales la regla debe ejecutarse.

Esta será la estructura final que el sistema nos debe proporcionar, que como se puede ver es una composición de la información de salida de cada uno de los pasos

3. Diseño de Aplicación Generativa

En el diseño consideramos dos aproximaciones principales: la **creación de un agente con múltiples tolos encargado del razonamiento y la coordinación del proceso**, y el uso de una librería como LangGraph para definir un mapa de estados y un flujo de ejecución. Optamos por comenzar con la primera opción, debido a nuestra menor experiencia con LangGraph, al ser una librería con menos tiempo en el mercado y por tanto algo menos madura.

3.1 Agente Customizado para la Coordinación del Proceso

Para el desarrollo del sistema, decidimos crear un agente personalizado con dos componentes principales: un **planificador encargado de llevar el control del flujo de ejecución y en base a la ejecución de acciones específicas ir rellenando el objeto de respuesta final.**

Cada action es en sí un chain específico, encargado de ejecutar un paso específico, por ejemplo, creación de una regla de validación de esquema, creación de un join, etc. Este enfoque nos permitió modularizar el código y facilitar la evaluación del rendimiento del sistema, ya que pudimos testear de forma independiente cada action, validar su salida mediante su comparación con la estructura esperada, antes de pasarlo a su integración en el agente principal.

3.2 Estructura de Clases con Pydantic

Lo primero que hicimos fue **diseñar una estructura de clases utilizando Pydantic**, similar a la estructura que manejamos en nuestro backend de OMMA. Esto nos permitió asegurar la consistencia y la validación de datos a lo largo del proceso de desarrollo. La estructura de clases incluyó componentes como agregaciones, joins, condiciones de ejecución y diferentes tipos de reglas.

Es muy importante mantener una coherencia de clases. Por ejemplo, en nuestro caso disponemos de un backEnd en Spring Boot que define las clases y su almacenamiento en base de datos. Estas clases tienen su correspondencia en interfaces en TypeScript en el FrontEnd, y las clases en Pydantic se crearon también con una correspondencia similar. **De esta manera los sistemas pueden trabajar intercambiando datos entre FrontEnd, BackEnd e IA Generativa.**

3.3 Configuración de Salida Estructurada

Para cada regla, configuramos un output estructurado utilizando la siguiente configuración:

```
structured_llm = model.with_structured_output(QualityAggregation)
```

Esta configuración forzaba a la LLM a proporcionar la salida en la clase necesaria, lo cual presentaba dos ventajas principales:

- **Modularización Óptima**: Permitiendo que cada componente del sistema se desarrolle y pruebe de manera independiente.
- **Pruebas Unitarias Eficientes**: Facilitando la evaluación del comportamiento de cada regla mediante la comparación del objeto de salida de la LLM con el objeto esperado.

3.4 Agente Planificador

Una vez desarrollados los componentes fundamentales, nos enfocamos en la creación del planificador. El planificador **actúa como el agente principal, coordinando todas las acciones** necesarias para completar el flujo de creación de reglas basándose en las entradas del usuario. Aquí describimos el proceso de implementación del planificador y cómo se integra con las tools y el modelo de lenguaje.

```
# Definición de las tools de OMMA

omma_tools = [...]  # Lista de tools necesarias para el agente

llm_with_tools = llm.bind_functions([omma_tools + Response]) # Se incluyen las tools definidas más el objeto de respuesta creado en Pydantic
```

3.5 Implementación de Estructura de Salida en el Agente de Planificación utilizando LangChain

Para resolver el problema de cómo añadir una estructura de salida al agente de planificación, nos basamos en las instrucciones de LangChain para agentes con salida estructurada. El enfoque consiste en **crear un parser** que se añadirá a las tools que se enlazan a la LLM del agente de planificación, pero no a las tools que se envían al AgentExecutor.

3.5.1 Definir la Clase de Respuesta:

Definimos la **clase Response** utilizando Pydantic para estructurar la salida deseada.

```
from pydantic import BaseModel, Field

from typing import Optional, Union, List

class Response(BaseModel):

    joins: Optional[QualityJoins] = Field(description="Joins a realizar")

    aggregation: Optional[QualityAggregation] = Field(description="Agregación realizada")

    rules: List[QualityRule] = Field(description="The created rules")

    conditions: Optional[QualityCondition] = Field(description="Condiciones de ejecución de la regla")
```

3.5.2 Crear el Parser:

Utilizamos LangChain para crear un parser que se encargará de formatear la salida del agente de planificación. Para ello, aconsejo seguir la documentación de LangChain:

https://python.langchain.com/v0.1/docs/modules/agents/how_to/agent_structured/

Es importarte que el objeto de Respuesta se enlace en la LLM que usaremos para el agente de planificación, pero no en el Agent Executor. El parser se incluye en el agente de la siguiente manera:

```
agent = (
    {
        "input": lambda x: x["input"],
        # Formatear el scratchpad del agente desde los pasos intermedios
        "agent_scratchpad": lambda x: format_to_openai_function_messages(
            x["intermediate_steps"]
        ),
    }
    | full_prompt
    | llm_with_tools #Incluir Response en las tools con las que se hace bind
    | parser
)
```

Un aspecto importante del código anterior es que es necesario realizar un bind de las tools con la LLM que se va a utilizar, e incluir el objeto de Respuesta.

El AgentExecutor se encarga de ejecutar el agente con las tools necesarias, manteniendo la capacidad de devolver los pasos intermedios para su análisis.

```
# Creación del ejecutor del agente
agent_executor = AgentExecutor(agent=agent, tools=omma_tools, verbose=True,
return_intermediate_steps=True)
```

Es importante no incluir el parser en las tools que se envían al AgentExecutor, solo se debe enlazar a la LLM del agente de planificación.

3.5.2.1 Crear el Prompt

Aunque no podemos incluir el prompt completo por motivos de propiedad intelectual de nuestro producto sí podemos daros un prompt similar algo más genérico para que se pueda ver cómo es la estructura que hemos seguido:

```
Eres un asistente encargado de crear reglas de calidad de datos en base al
statement que te de el usuario.
Primero debes evaluar si tienes que hacer una agregación.
Después debo crear todas las reglas necesarias. Si tienes que crear una
regla para varias columnas, no crees multiples reglas, crea 1 regla para N
columnas
El output de las tool de reglas se debe incluir en  el campo rules de la
respuesta final.
El último paso es crear las condiciones de ejecución necesesarias si las
hubiera. Si algún paso falla, reintenta ejecutarlo un máximo de 3 veces
hasta obtener un resultado correcto
```

3.5.3 Validación y Pruebas

Realizamos pruebas unitarias para asegurar que cada componente del objeto de regla se genera correctamente y se ajusta al formato esperado.

Uno de los problemas que encontramos al utilizar un modelo de lenguaje generativo para ensamblar las piezas de la respuesta final fue la falta de consistencia. Específicamente, el **modelo a veces tomaba "licencias"** con los **nombres de las columnas y otros detalles**, lo que resultaba en errores que arruinaban la creación completa de la regla.

Otros problemas encontrados junto con las soluciones implementadas fueron:

- **Estandarización y Validación de Nombres de Columnas:** Implementamos una validación estricta de los nombres de las columnas en cada paso para asegurar que los nombres usados sean consistentes a lo largo de todo el proceso.

- **Uso de Parsers y Estructuras Fuertemente Tipadas**: Fortalecimos el uso de parsers y estructuras fuertemente tipadas para cada componente del flujo de trabajo, asegurando que los outputs de las LLMs sean conformes a las especificaciones definidas.
- **Reforzar la Lógica del Agente con Validaciones Adicionales**: Introducimos validaciones adicionales en el agente de planificación para detectar y corregir automáticamente cualquier inconsistencia en la salida generada.

Pero a pesar de estas mejoras, **las respuestas, aunque se obtuvieran con el formato deseado, no eran 100% confiables**. Se intentó ajustar el prompt para evitar que creara columnas nuevas inventadas y otras licencias que la LLM tomaba, pero no llegamos a obtener un resultados lo suficientemente confiable como para que concluyéramos que era un modelo implementable en producción.

Por tanto, **cambiamos ligeramente el enfoque**.

3.6 Implementación de Tools Mejoradas para la Generación de Respuestas en OMMA

Para abordar los problemas de consistencia y precisión en la generación de respuestas, modificamos nuestras tolos. En lugar de recibir solo el statement y el dataset y proporcionar una respuesta unitaria, cada tool también recibe el objeto Response. De esta manera, **cada tool se encarga de rellenar programáticamente la sección correspondiente del Response** basado en el resultado de la LLM y devolver el Response actualizado. Esta aproximación ha demostrado ser significativamente más efectiva.

3.6.1 Modificación de las Tools

Cada tool ahora recibe el objeto Response como entrada y lo actualiza con la información relevante antes de devolverlo.

```
def create_join(statement, dataset, response: Response) -> Response:
    # Lógica para crear el join basado en el statement y dataset
    join_result = some_llm_function_to_create_join(statement, dataset)
    # Actualizar el objeto Response
    response.enrichDataformat = EnrichDataformatObject(table=join_result["table"],
on=join_result["on"])
    return response
```

3.6.2 Configuración del Agente

Configuramos el agente de planificación para utilizar estas tools mejoradas y devolver el objeto Response final.

3.6.3 Beneficios

- **Consistencia Mejorada**: Cada tool actualiza programáticamente el objeto Response, asegurando que los detalles sean precisos y consistentes.
- **Salida Estructurada y Utilizable**: El objeto Response final es estructurado y listo para su uso directo, minimizando errores.
- **Validación Continua**: La estructura validada de Response ayuda a detectar y corregir errores en cada etapa del proceso.

3.6.4 Problemática: consumo de tokens

Un aspecto que encontramos con esta aproximación es que, aunque es muy fiable, el consumo de tokens aumentaba. Y nos fue bastante difícil encontrar inicialmente donde estaba el problema. Cuando hacemos un bind de tools en la llm, la información de las tools es serializada y almacenada en la LLM. En nuestro caso como las tools llevaban parámetros que eran clases complejas en pydentic, todo esto iba sumando tokens.

Si utilizábamos los métodos habituales de langchain para analizar el consumo, esta información no aparecía, ya que no son tokens del prompt realmente, pero sí que se contabilizaba en el consumo de la LLM.

Tuvimos que hacer una ingeniería de las tools para minimizar el espacio en tokens que sus definiciones ocupaban y evitar que el consumo se disparara.

3.7 Resumen de Implementación

La solución final fue, por tanto, crear un agente encargado de la coordinación de todas las acciones necesarias para crear las reglas de OMMA. Como punto clave, se le dio la secuencia de pasos que debía dar para crear una regla, tanto opcionales como obligatorios. Cada uno de esos pasos mapea con un tipo de acción. Y su objetivo es ir rellenando iterativamente un objeto Response que incluye toda la información de la regla.

Las tools son en sí mismas una chain con su prompt y su LLM encargadas de, en base a un input y a información extra como el dataset sobre el que trabaja, crear la estructura de la regla o el paso necesario. Aquí es importante destacar que igual que **el planificador ha necesitado usar un modelo GPT-4O** en nuestro caso (cualquier GPT-4 habría funcionado, pero ya que teníamos esta nueva versión cuyo coste es la mitad que GPT-4 turbo por qué no usarlo) mientras que las actions se han implementado con GPT-3.5 turbo a excepción de alguna puntual como la generación de condiciones de ejecución que por su complejidad necesitaban más poder de razonamiento.

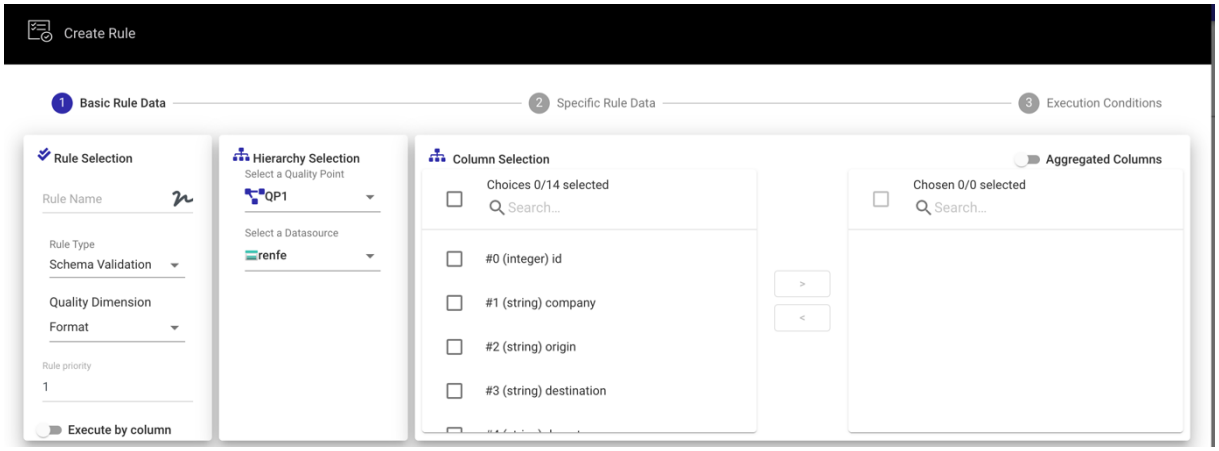
Cada tool ahora procesa su tarea específica (por ejemplo, crear un join, una agregación, una regla específica o establecer condiciones de ejecución) **y actualiza el objeto Response con la información relevante.** Este enfoque asegura que cada detalle sea preciso y coherente en toda la salida generada.

Esta implementación ha demostrado ser mucho más efectiva, ya que **garantiza que cada componente** del proceso de creación de reglas **se maneje de manera estructurada y precisa.** La iteración programática del objeto Response por parte de las tools **asegura que los detalles sean consistentes y correctos,** mejorando significativamente la integridad y fiabilidad del sistema de gestión de datos en OMMA, aunque ha sido necesario una revisión cuidadosa del **tamaño del prompt** debido a que en las llamadas se envía serializados los tools, y si el objeto Response crecía mucho **podía aumentar exponencialmente el consumo**. Este es un **punto de investigación y trabajo a futuro**, a medida que la complejidad del sistema aumente, y su número de tools aumente, puede que sea necesario revisar esta arquitectura para tener un mejor escalado.

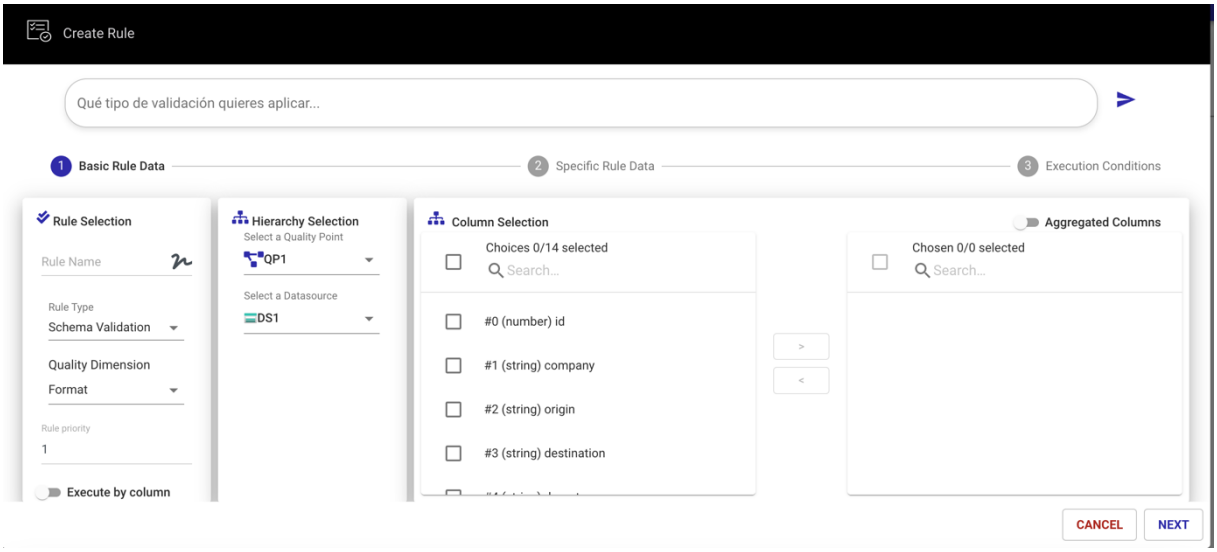
El resultado es un **sistema modular y escalable** que facilita la extensión y el mantenimiento. **Cada tool maneja su parte específica del proceso**, lo que permite una fácil adición de nuevas funciones o adaptaciones a cambios en los requisitos. **El planificador coordina eficazmente todas las acciones** y las tools actualizan el objeto Response de manera estructurada, garantizando la precisión del output y reduciendo significativamente el riesgo de errores. Este enfoque ha mejorado notablemente la fiabilidad general del sistema de **OMMA**.

4. Integración en OMMA

En **OMMA**, como ya explicamos anteriormente, existe un flujo de trabajo en tres pasos para la creación de reglas mediante un formulario intuitivo. Cuando un usuario crea una regla en OMMA, hay cierta información contextual que se genera automáticamente, como el usuario, metadata, y otros detalles específicos del sistema. Esta información es crucial, pero no queríamos que la aplicación generativa se encargara de rellenarla. Por lo tanto, diseñamos un componente inicial, en los formularios de creación de reglas, que interactúa con nuestro agente planificador para obtener el objeto Response y mapearlo en los campos necesarios del front-end.



Este es el formulario base de OMMA para la creación de reglas. Con nuestra integración con el Agente de IA Generativa, creamos un componente así:



Esta integración se ha implementado inicialmente como una prueba de concepto (PoC) para validar toda la arquitectura end-to-end. **A través de este proceso de testing, hemos podido identificar y solucionar problemas potenciales, asegurando que la arquitectura sea robusta y escalable**. Esta validación nos ha permitido refinar nuestro sistema, **mejorando tanto la experiencia del usuario como la eficiencia operativa**.

Para las pruebas, **utilizamos un dataset público de Kaggle, (https://www.kaggle.com/datasets/thegurusteam/spanish-high-speed-rail-system-ticket-pricing).** Este dataset de viajes en tren es bastante completo y lo usamos frecuentemente para diversos tipos de demostraciones en OMMA. La prueba de concepto (PoC) permitió validar toda la arquitectura end-to-end, asegurando que el sistema funcione de manera robusta y escalable.

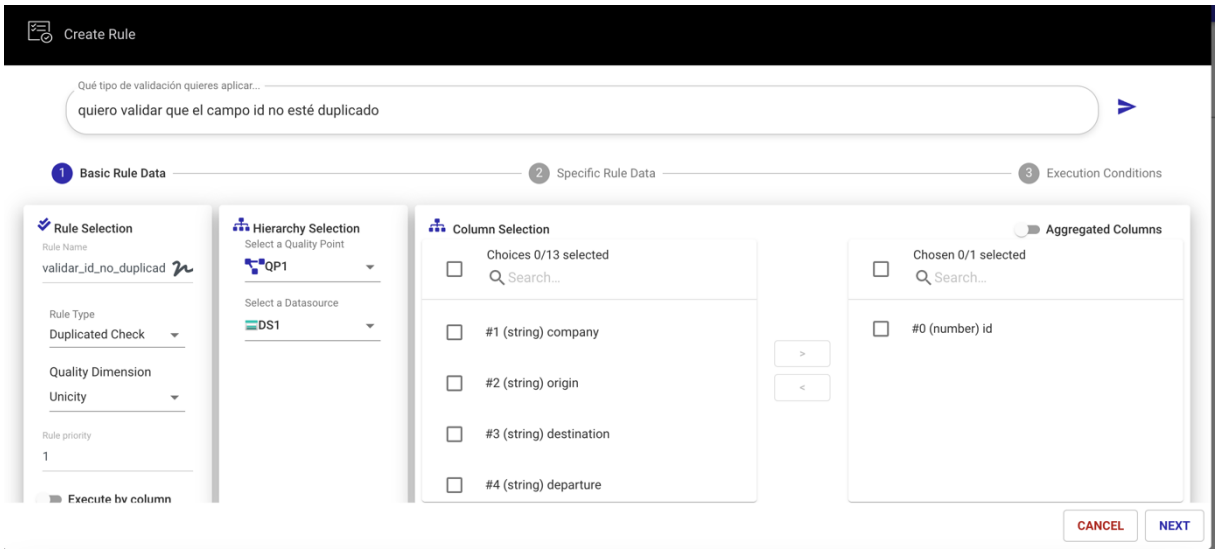
4.1 Pruebas y Validación

Vamos a mostraros algunos ejemplos de creación de reglas:

4.1.1 Validación de la PK del dataset

Normalmente uno de los primeros puntos es validar que la PK de los datos no esté duplicada:

```
quiero validar que el campo id no esté duplicado
```



Como podéis ver, el sistema proporciona tanto un nombre para la regla elige el tipo de regla de Validación de Duplicados, y el campo ID.

Se trata de la prueba más básica, vamos a ir complicando la creación de reglas

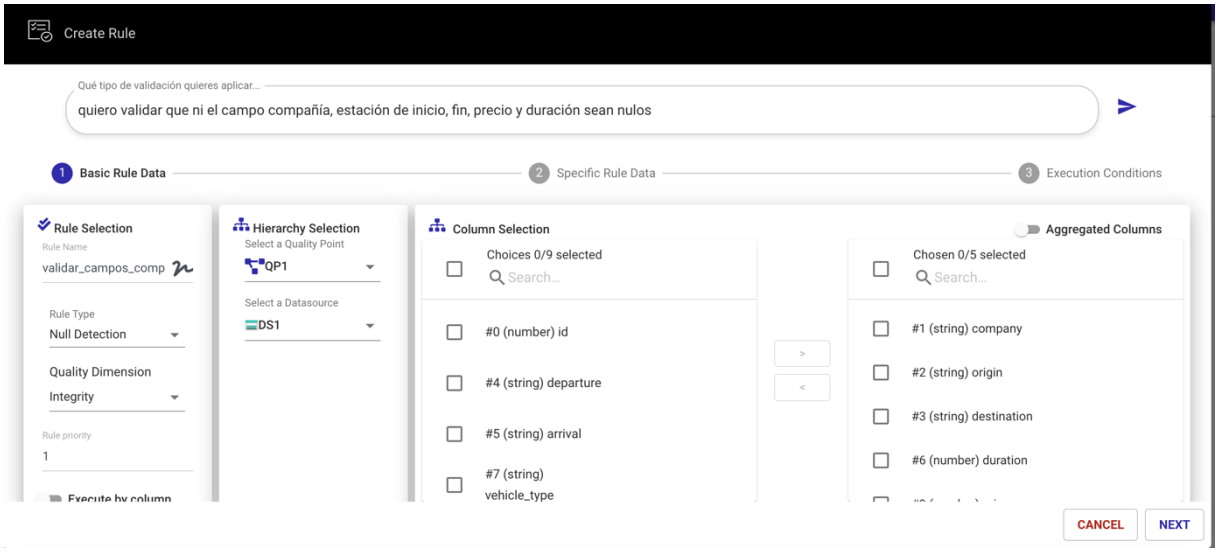
4.1.2 Validación de campos no nulos

El siguiente punto va a ser asegurarme de que las columnas que me interesan no tengan valores nulos:

```
quiero validar que ni el campo compañía, estación de inicio, fin, precio y duración sean nulos
```

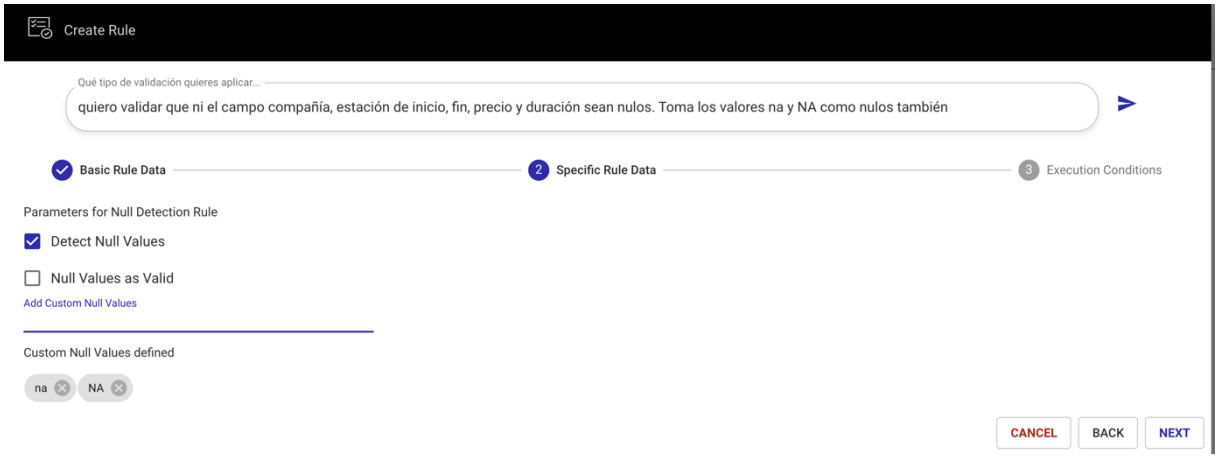
Como puede verse se crea la regla de nulos para los campos indicados:

Es importante remarcar que **hemos definido las columnas de forma algo difusa**, no hablamos de origen y destino sino estación de inicio y fin. Como puede verse el sistema identifica correctamente las columnas deseadas.



Voy a añadir un elemento extra y es marcar algunos datos custom que también quiero que se tomen como nulos, na y NA:

quiero validar que ni el campo compañía, estación de inicio, fin, precio y duración sean nulos. Toma los valores na y NA como nulos también



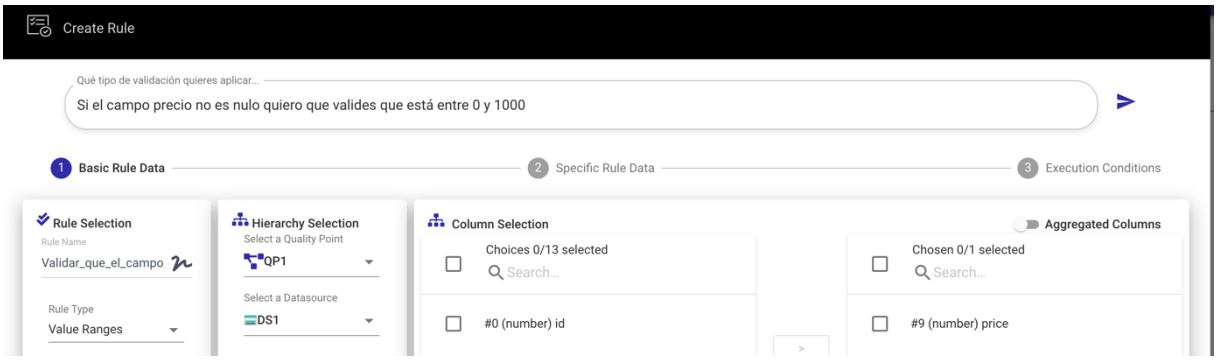
Fijaos cómo en este caso en la configuración específica de la regla ha incluido los valores custom.

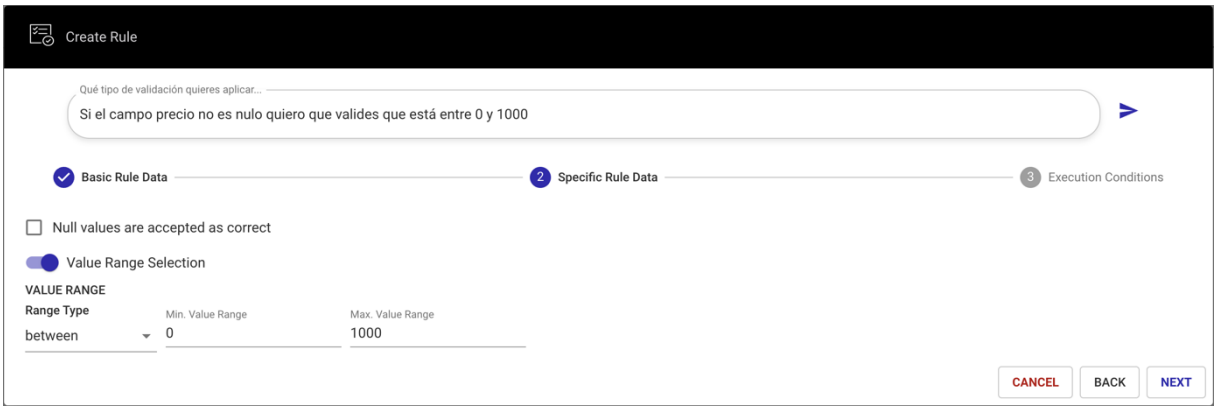
4.1.3 Validación con condiciones de ejecución

Ahora vamos a añadir algún elemento más, como son condicionantes:

Si el campo precio no es nulo quiero que valides que está entre 0 y 1000

En este caso, crea correctamente la regla de value ranges:





Y en el paso 3 de condiciones de ejecución crea la condición que hemos especificado:

Crearemos un video enseñando conceptos más avanzados, como crear agregaciones de los datos, joins y condiciones más complejas, para que veáis la potencia de la solución.

5. Conclusiones y Siguientes Pasos

Con las pruebas realizadas, podemos decir orgullosos que **hemos creado el primer sistema de creación y despliegue de políticas de calidad con IA Generativa**. Pero para ello, hemos tenido que obtener avances en varias áreas, como:

- **Arquitectura de Agentes con Información Estructurada:** Hemos desarrollado una arquitectura de agentes capaz de trabajar en todos sus pasos con información estructurada, asegurando consistencia y precisión en la generación de reglas.
- **Flujo de Ejecución Definido y Confiable:** La arquitectura creada permite la generación de estructuras complejas y por pasos, siguiendo un flujo de ejecución definido de forma fiable. Cada paso está mapeado a una acción específica que garantiza la coherencia y precisión en la creación de reglas.
- **Estructura de Clases y Tools Modular y Escalable:** Hemos diseñado una estructura de clases y tools que facilita el desarrollo y las pruebas unitarias de los diferentes pasos y elementos del flujo de creación de reglas. Esta modularidad permite escalar y evolucionar la solución de manera sencilla y eficiente.
- **Integración Completa End-to-End:** Se ha llevado a cabo una integración completa end-to-end, desde la generación de reglas mediante LLMs hasta su revisión y ajuste por parte del usuario en el front-end. Este enfoque asegura una experiencia de usuario fluida y una implementación robusta.
- **Validación de Flujos Basados en IA Generativa:** Hemos validado la factibilidad de sustituir los flujos habituales de trabajo por flujos basados en IA Generativa. Las pruebas con datasets reales han demostrado que la solución no solo es viable sino también eficiente y precisa.

Con esto, **en este momento disponemos de la integración en OMMA de la creación de reglas mediante IA Generativa**. Lo que nos ha permitido disponer del primer sistema de creación de políticas de calidad mediante IA Generativa. Dentro de OMMA, donde una de nuestras premisas es facilitar la creación de políticas de calidad sin conocimiento técnico, supone un paso enorme, ya **que facilita de forma exponencial la creación de reglas.**

A su vez, **dentro de Mática, esto nos ha permitido validar nuestra idea de evolucionar las interfaces hacia soluciones mucho más amigables y fáciles para el usuario**. Este avance no solo transforma la forma en que los usuarios interactúan con nuestras tools, sino que también nos posiciona en la vanguardia de la innovación en gestión de calidad de datos, al incorporar inteligencia artificial de manera efectiva y práctica en nuestras soluciones.

Pero esto no queda aquí. Se trata del primer paso con éxito dentro de una hoja de ruta mucho más extensa. De momento os hemos enseñado una primera integración que sustituye nuestros formularios de creación de reglas. **Pero el sistema que hemos creado no solamente permite crear una regla cada vez, sino que de una sola instrucción nos permite crear múltiples reglas**, y ese será el siguiente paso. Posteriormente, también trabajaremos en integrar esta tecnología en muchos otros de nuestros interfaces, como el recomendador inteligente de reglas. Además, vamos a explorar nuevos tipos de reglas, donde se pueda introducir una sentencia SQL y el propio sistema la convierta en una tipología de regla utilizable en OMMA.

También estamos avanzando con la implementación de LangGraph para comparar su eficiencia con nuestra arquitectura actual de agentes y decidir cuál es la mejor opción para OMMA.

Finalmente, a través de esta integración con nuestro **Framework de GenAI**, podremos desplegar nuestras soluciones en diversas plataformas LLM, añadiendo capacidades de observabilidad y medición de costes, y asegurando que nuestras soluciones sean robustas, escalables y adaptables a diferentes entornos. Este desarrollo inicial sienta las bases para una transformación continua de nuestras herramientas, haciendo que la gestión de calidad de datos sea cada vez más accesible, eficiente y poderosa.

Este desarrollo inicial sienta las bases para una constante evolución de nuestras herramientas, haciendo que la gestión de calidad de datos sea cada vez más accesible y eficiente.