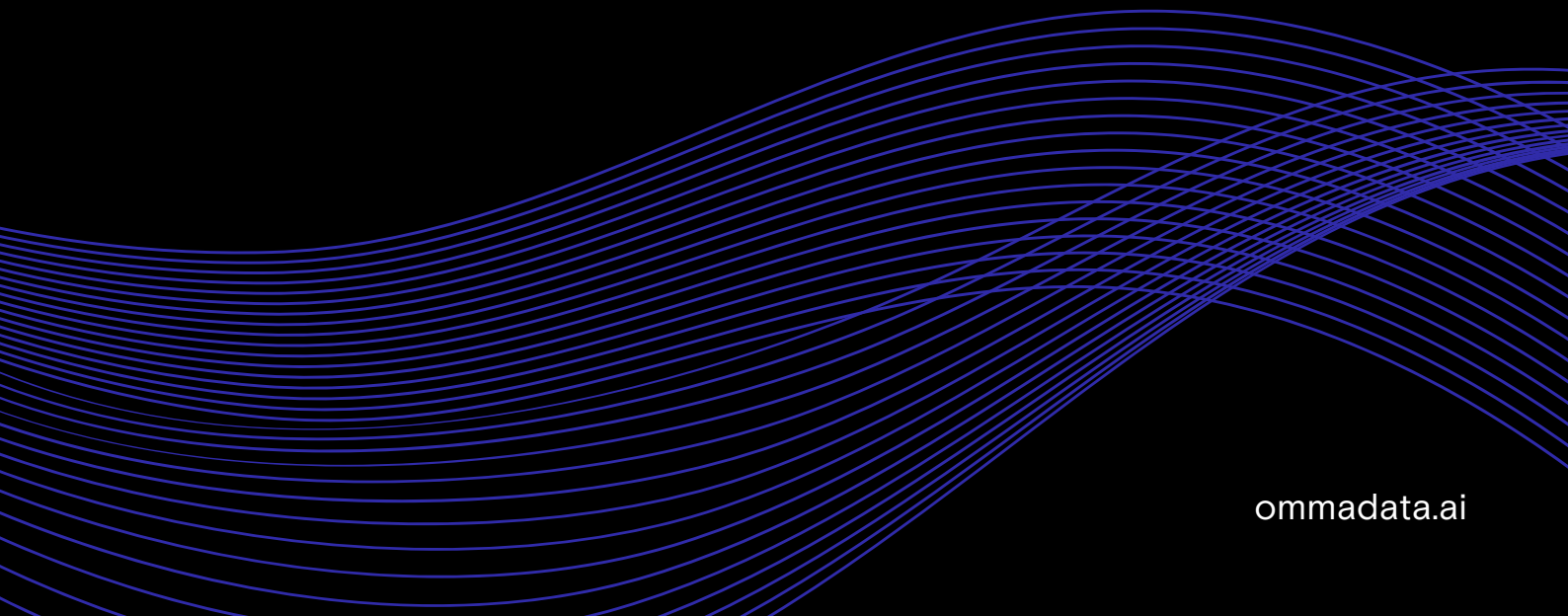
MADRID, MARCH 5, 2024

OMMA

DATA QUALITY LLM AGENT FOR OPERATIONAL INTEGRATION

GENERATIVE APPLICATION FOR CREATING QUALITY RULES

JUAN RAMON GONZALEZ
CTO AT MÁTICA PARTNERS



ommadata.ai

1. OMMA: an introduction

OMMA is an advanced **data quality** tool designed **to facilitate the creation and management of quality policies through no-code interfaces**. From its inception, OMMA has aimed to empower users by allowing them to define data quality rules **without needing extensive technical knowledge**. This approach is based on the premise that a deep understanding of the data is more crucial than programming skills to ensure data integrity and accuracy.

OMMA simplifies the task of creating data quality rules, which was traditionally done manually and technically, making it difficult and error-prone. With an intuitive platform, OMMA enables users to create policies based on their knowledge of the data, making the process easier and more efficient.

2. Introduction to Integrating LLMs for Creating Natural Language Rules in OMMA

We are currently immersed in the application of Large Language Models (LLMs) across a multitude of tasks. However, like all new technologies, they are in a process of evolution, standardization, and maturity. Today, the vast majority of use cases we encounter are based on the creation of chatbots, Retrieval-Augmented Generation (RAG) systems, and similar applications. The application of LLMs in other practical scenarios is not yet as widely exploited or documented.

The integration of LLMs into productive processes, where entire areas or procedures can be replaced by a more agile and user-friendly interface, represents a significant transformation in how we interact with systems, such as data management systems.

At **Grupo Mática**, we have a significant advantage thanks to OMMA, our proprietary product. With OMMA, we can create, test, and deploy complete productive functionalities, such as the integration of LLMs in creating quality policies. This represents a significant step forward, allowing the establishment of rules using natural language.

This integration has the potential to completely transform the process of defining data quality policies. **By using LLMs, OMMA users will be able to formulate rules and guidelines in natural language**, which will then be interpreted and executed by the system. This not only simplifies the process, making it more accessible to non-technical users, but also accelerates implementation time and reduces the possibility of errors.

The challenge we face with this integration lies in the need to transform data quality rules and policies that have traditionally been defined manually and technically, into understandable and applicable guidelines using natural language. This transformation will not only make the process more accessible but also improve efficiency and accuracy in data quality management.

In this section, we will explore in detail the challenges and opportunities that come with integrating LLMs into OMMA. We will analyze how LLMs can efficiently and accurately interpret and generate data quality rules, thereby improving data governance. Additionally, we will discuss the methodologies and technical approaches we have adopted to ensure that this integration is not only effective but also scalable and secure.

2.1 Rule creation workflow in OMMA

OMMA has been designed to **simplify the creation and management of data quality policies** through an intuitive process that requires **no advanced technical knowledge**.

The system relies on a **three-step form** that allows users to define data quality rules efficiently and accurately.

- **Basic Rule Data:** In this first step, the user **selects the dataset** on which they wish to apply their quality policies. They decide the type of rule they want to implement and give it a descriptive name. Additionally, they choose the specific columns of the dataset to which the rule will apply. The user also decides whether to work with the dataset in its original form or apply aggregations or joins to enrich the data. **This initial step establishes the foundations of the rule and ensures its application to the relevant**.

- **Specific Rule Data:** In the second step, **the specific behavior of the selected rule is configured**. Depending on the type of rule, the user defines detailed parameters. For example, if a regular expression rule is chosen, the corresponding regular expression is entered. If the selected rule involves value ranges, the limits of the ranges are specified. **This step allows for detailed customization of the rule, ensuring it fits the user's exact needs and the nature of the dataset.**

- **Execution Conditions:** In the final step, the user **sets the conditions under which the rule will be executed**. This involves defining the conditions that the data must meet for the rule to be applied. For instance, conditions can be specified based on dates, specific values in other columns, or any other relevant criteria. **This final stage ensures that quality rules are applied accurately and in the appropriate context**.

This three-step process in OMMA is straightforward and does not require advanced technical skills, but rather a deep understanding of the data and the type of validation that needs to be implemented.

Considering modifying this workflow by replacing it with an LLM requires the system to be capable of mimicking this rule creation behavior.

2.2 Generative Application Requirements for OMMA

To create a generative AI-based system that replaces the rule creation forms in OMMA, it is necessary to ensure that the workflow is efficient, replicates the current process, and that the outputs generated are structured and reliable. Below are the key components and requirements to achieve this objective.

Workflow Definition: Steps to Follow

The system must follow a nested workflow that allows users to create data quality rules in a structured and sequential manner. This workflow consists of the following steps:

- **Creation of Joins (Optional):** The system should allow users to specify if they want to perform joins with other tables in the database. This involves identifying and selecting the relevant tables and defining the join conditions.

- **Creation of Aggregations (Optional):** Users may choose to apply aggregations to the data. The system should offer options to select the types of aggregations (e.g., sums, averages) and the columns to which they will be applied.
- **Definition of Rules:** This is the mandatory step where users specify the type of rule they want to create (e.g., regex, value ranges) and provide the specific parameters for each type.
- **Configuration of Execution Conditions (Optional):** The system should allow the configuration of conditions under which the rules will be executed, ensuring that the rules are applied only to the relevant data.

Structured Output

To ensure the reliability of the system, it is vital that the information handled by the language model is structured. This means that the output generated by the AI must have a predefined format that includes all the necessary components: aggregation, join, list of rules, and execution conditions. This has been one of the critical points. There is ample documentation on how to implement chains with structured outputs and even simple agents, but when we enter complex agent systems, as we will see later, the technical solution becomes more complicated.

Components of the Response Object

The response object of the system should be composed of the following elements:

- **Joins**: Information about the tables to join and the join conditions.
- **Aggregations**: Details about the types of aggregations and the columns to which they are applied.

- **List of Rules**: Definition of the specific rules, including the type of rule and its parameters.
- **Execution Conditions**: Conditions under which the rule should be executed.

This will be the final structure that the system must provide, which, as can be seen, is a composition of the output information from each of the steps.

3. Design of Generative Application

In designing the application, we considered two main approaches: **creating an agent with multiple tools responsible for reasoning and coordinating the process**, and using a library like LangGraph to define a state map and execution flow. We opted to start with the first option due to our limited experience with LangGraph, as it is a relatively new and less mature library.

3.1 Customized Agent for Process Coordination

For system development, we decided to create a customized agent with two main components: **a planner responsible for controlling the execution flow and, based on specific actions, filling in the final response object.**

Each action is itself a specific chain, responsible for executing a specific step, such as creating a schema validation rule, creating a join, etc. This approach allowed us to modularize the code and facilitate system performance evaluation, as we could independently test each action and validate its output against the expected structure before integrating it into the main agent.

3.2 Class Structure with Pydantic

The first step was to **design a class structure using Pydantic**, similar to the structure we manage in our OMMA backend. This ensured consistency and data validation throughout the development process. The class structure included components such as aggregations, joins, execution conditions, and different types of rules.

Consistency in class definitions is crucial. For instance, in our case, we have a backend in Spring Boot that defines the classes and their storage in the database. These classes have corresponding interfaces in TypeScript in the frontend, and the Pydantic classes were also created with similar correspondences. **This way, systems can exchange data seamlessly between the frontend, backend, and generative AI.**

3.3 Structured Output Configuration

For each rule, we configured a structured output using the following configuration:

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

This configuration forced the LLM to provide output in the necessary class, offering two main advantages:

- **Optimal Modularization**: Allowing each system component to be developed and tested independently.
- **Efficient Unit Testing**: Facilitating the evaluation of each rule's behavior by comparing the LLM's output object with the expected object.

3.4 Planner Agent

Once the fundamental components were developed, we focused on creating the planner. **The planner acts as the main agent, coordinating all the necessary actions** to complete the rule creation flow based on user inputs. Below, we describe the planner implementation process and how it integrates with the tools and language model.

```
# Defining OMMA Tools

omma_tools = [...]  # List of necessary tools for the agent

llm_with_tools = llm.bind_functions([omma_tools + Response]) # Including the defined tools and the response object created in Pydantic
```

3.5 Implementing Structured Output in the Planning Agent using LangChain

To address how to add structured output to the planning agent, we followed LangChain's instructions for agents with structured output. The approach involves **creating a parser** that is added to the tools bound to the planner agent's LLM, but not to the tools sent to the AgentExecutor.

3.5.1 Defining the Response Class:

We define the **Response class** using Pydantic to structure the desired output.

```
from pydantic import BaseModel, Field
from typing import Optional, Union, List
class Response(BaseModel):
    joins: Optional[QualityJoins] = Field(description=" Joins to perform")
    aggregation: Optional[QualityAggregation] = Field(description=" Performed aggregation")
    rules: List[QualityRule] = Field(description="The created rules")
    conditions: Optional[QualityCondition] = Field(description=" Rule execution conditions")
```

3.5.2 Creating the Parser:

We used LangChain to create a parser responsible for formatting the planning agent's output. Detailed instructions can be found in LangChain's documentation:

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

It is important to bind the response object to the LLM used for the planning agent but not to include it in the AgentExecutor tools. The parser is included in the agent as follows:

```
agent = (
    {
        "input": lambda x: x["input"],
        # Formatting the agent scratchpad from the intermediate steps
        "agent_scratchpad": lambda x: format_to_openai_function_messages(
            x["intermediate_steps"] ),
    }
    | full_prompt
    | llm_with_tools # Include Response in bind tools | parser
)
```

An important aspect of the above code is the need to bind the tools with the LLM (Language Model) that will be used and include the Response object.

The AgentExecutor is responsible for executing the agent with the necessary tools, maintaining the capability to return intermediate steps for analysis.

```
# Creation of the Agent Executor
agent_executor = AgentExecutor(agent=agent, tools=omma_tools, verbose=True,
return_intermediate_steps=True)
```

It's important not to include the parser in the tools sent to the AgentExecutor; it should only be bound to the LLM of the planning agent.

3.5.2.1 Creating the Prompt

Although we cannot include the full prompt due to intellectual property reasons for our product, we can provide a somewhat more generic prompt so you can see the structure we have followed:

```
You are an assistant responsible for creating data quality rules based on the user's statement.
First, evaluate if you need to perform an aggregation. Then create all the necessary rules. If
you need to create a rule for multiple columns, do not create multiple rules; create one rule
for N columns. The output of the rule tools should be included in the 'rules' field of the final
response. The last step is to create the necessary execution conditions if any. If any step
fails, retry it up to 3 times until a correct result is obtained.
```

3.5.3 Validation and testing

We conducted unit tests to ensure that each component of the rule object is generated correctly and conforms to the expected format.

One of the issues we encountered when using a generative language model to assemble the pieces of the final response was a lack of consistency. Specifically, the **model sometimes took "liberties"** with the **names of columns and other details**, resulting in errors that ruined the entire rule creation.

Other issues we encountered, along with the implemented solutions, were:

- **Standardisation and Validation of Column Names:** We implemented strict validation of column names at each step to ensure that the names used were consistent throughout the entire process.

- **Use of Parsers and Strongly Typed Structures**: We reinforced the use of parsers and strongly typed structures for each component of the workflow, ensuring that the outputs of the LLMs conformed to the defined specifications.

- **Reinforcing Agent Logic with Additional Validations**: We introduced additional validations in the planning agent to automatically detect and correct any inconsistencies in the generated output.

Despite these improvements, **the responses, although obtained in the desired format, were not 100% reliable.** We attempted to adjust the prompt to prevent the creation of new, invented columns and other liberties the LLM took, but we did not achieve results reliable enough to conclude that it was a model implementable in production.

Therefore, **we slightly changed our approach.**

3.6 Implementation of Improved Tools for Response Generation in OMMA

To address the issues of consistency and accuracy in response generation, we modified our tools. Instead of merely receiving the statement and dataset and providing a unitary response, each tool also receives the Response object. In this way, **each tool programmatically fills in the corresponding section of the Response** based on the LLM's result and returns the updated Response. This approach has proven to be significantly more effective.

3.6.1 Modification of the Tools

Each tool now receives the Response object as input and updates it with relevant information before returning it.

```
def create_join(statement, dataset, response: Response) -> Response:
    # Logic to create the join based on the statement and dataset
    join_result = some_llm_function_to_create_join(statement, dataset)
    # Update the Response object
    response.enrichDataformat = EnrichDataformatObject(table=join_result["table"],
on=join_result["on"])
    return response
```

3.6.2 Agent Configuration

We configured the planning agent to use these enhanced tools and return the final Response object.

3.6.3 Benefits

- **Improved Consistency**: Each tool programmatically updates the Response object, ensuring precise and consistent details.
- **Structured and Usable Output**: The final Response object is structured and ready for direct use, minimizing errors.
- **Continuous Validation**: The validated structure of the Response helps detect and correct errors at each stage of the process.

3.6.4 Token Consumption Issue

One issue we encountered with this approach is that, although highly reliable, token consumption increased. Initially, pinpointing the problem was quite challenging. When we bind tools in the LLM, the information from the tools is serialized and stored in the LLM.

In our case, because the tools included parameters that were complex Pydantic classes, this added to token consumption. When using standard LangChain methods to analyze consumption, this information did not appear as part of the prompt tokens but still counted towards LLM consumption.

We had to engineer the tools to minimize the token space their definitions occupied to prevent consumption from skyrocketing..

3.7 Implementation Summary

The final solution was to create an agent responsible for coordinating all actions necessary to create **OMMA** rules. A key aspect was providing the sequence of steps required to create a rule, both optional and mandatory. Each of these steps mapped to a specific type of action, aiming to iteratively populate a Response object containing all rule information.

The tools themselves form a chain with their prompt and LLM, tasked with creating the rule or necessary step based on input and additional information like the dataset it works on. It's worth noting that while **the planner needed to use a GPT-4O model** in our case (any GPT-4 would have worked, but since we had this newer version whose cost is half that of GPT-4 Turbo, why not use it), the actions were implemented with GPT-3.5 Turbo, except for certain cases like generating execution conditions which required more reasoning power.

Each tool now handles its specific task (e.g., creating a join, an aggregation, a specific rule, or setting execution conditions) and updates the Response object with relevant information. **This approach ensures precision and coherence** across the entire generated output.

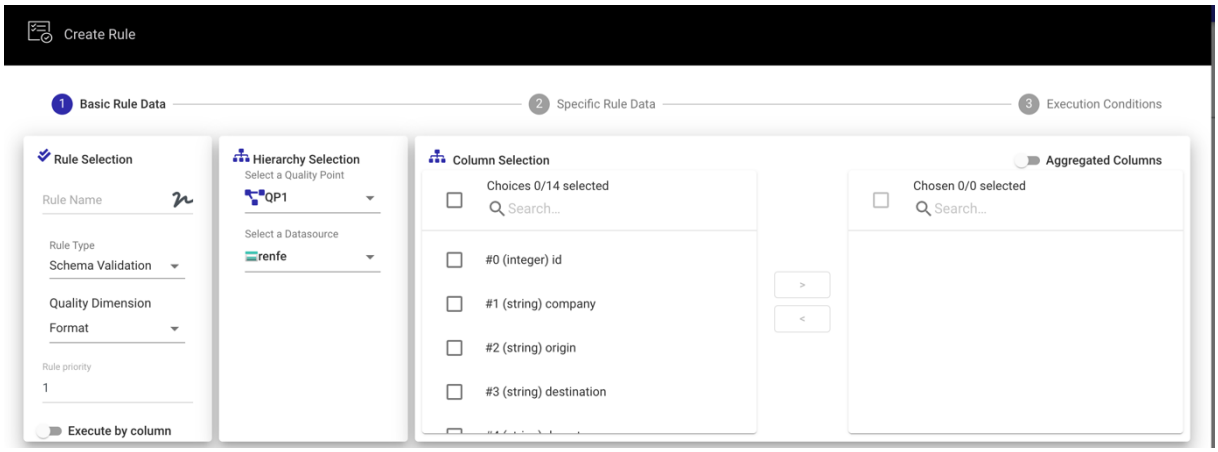
This implementation has proven much more effective by **ensuring each component** of the rule creation process is **handled in a structured and precise manner**. The programmatic iteration of the Response object by the tools **ensures details are consistent and correct**, significantly enhancing the integrity and reliability of the

data management system at OMMA. However, careful review of **prompt size** was necessary because tools are serialized in calls, and if the Response object grew too large, it **could exponentially increase consumption**. This is an **area for future investigation and work** as system complexity and tool count may require revisiting this architecture for better scalability.

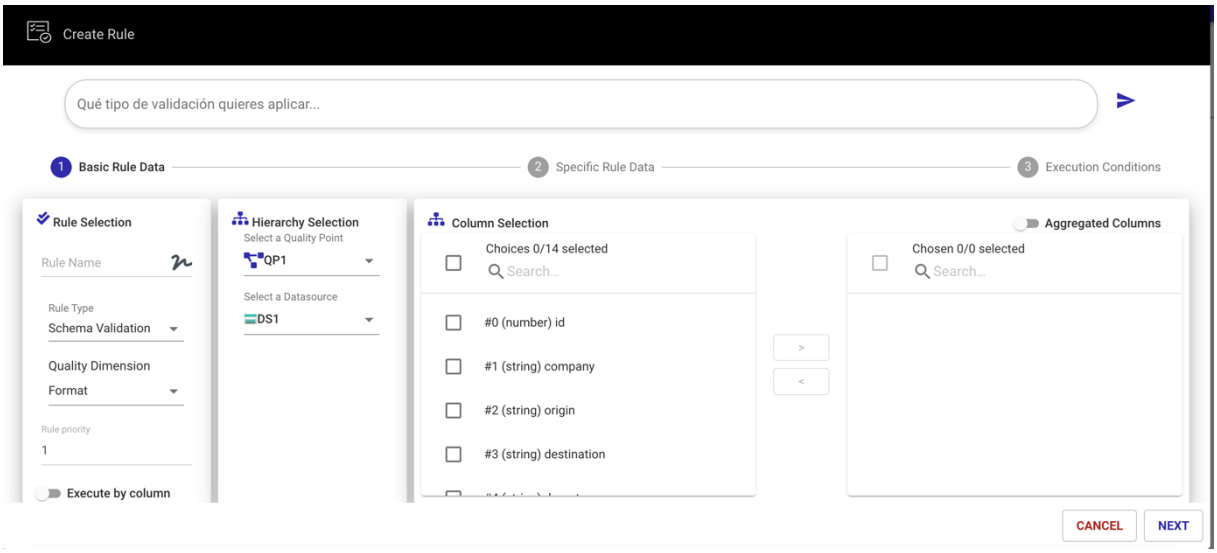
The result is a **modular and scalable system** that facilitates extension and maintenance. **Each tool manages its specific part of the process**, allowing easy addition of new functions or adaptations to changing requirements. **The planner effectively coordinates all actions**, and the tools update the Response object in a structured manner, ensuring output accuracy and significantly reducing error risks. This approach has notably improved the overall reliability of **OMMA's** system.

4. Integration in OMMA

At **OMMA**, as previously explained, there is a three-step workflow for creating rules using an intuitive form. When a user creates a rule in OMMA, certain contextual information is generated automatically, such as the user, metadata, and other system-specific details. This information is crucial, but we did not want the generative application to be responsible for filling it in. Therefore, we designed an initial component in the rule creation forms that interacts with our planning agent to obtain the Response object and map it to the necessary fields in the front end.



This is the base form in OMMA for rule creation. With our integration with the Generative AI Agent, we created a component like this:



This integration was initially implemented as a proof of concept (PoC) to validate the entire end-to-end architecture. **Through this testing process, we have been able to identify and resolve potential issues, ensuring that the architecture is robust and scalable**. This validation has allowed us to refine our system**, improving both the user experience and operational efficiency.**

For testing, **we used a public dataset from Kaggle: (https://www.kaggle.com/datasets/thegurusteam/spanish-high-speed-rail-system-ticket-pricing).** This train travel dataset is quite comprehensive and we frequently use it for various types of demonstrations in OMMA. The proof of concept (PoC) allowed us to validate the entire end-to-end architecture, ensuring that the system operates robustly and is scalable.

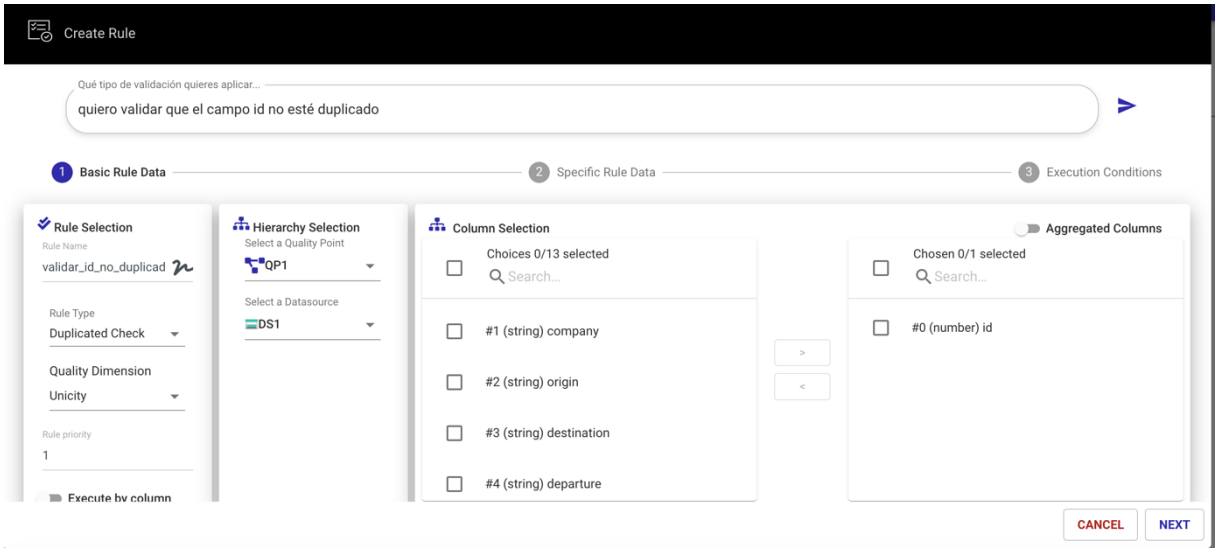
4.1 Testing and Validation

We will show you some examples of rule creation:

4.1.1 Dataset PK Validation

Usually, one of the first steps is to validate that the dataset's PK is not duplicated:

```
I want to validate that the id field is not duplicated
```



As you can see, the system provides both a name for the rule and selects the Duplicate Validation rule type and the ID field.

This is the most basic test, and we will gradually complicate the rule creation process.

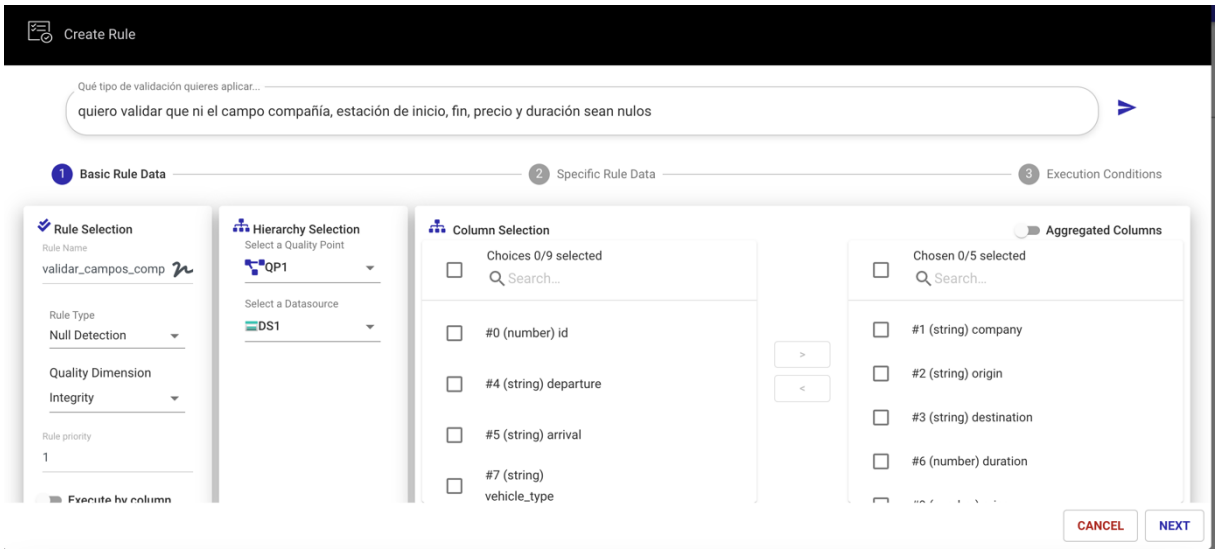
4.1.2 Validation of Non-Null Fields

The next step will be to ensure that the columns of interest do not have null values:

```
I want to validate that neither the company, start station, end station, price, nor duration fields are null
```

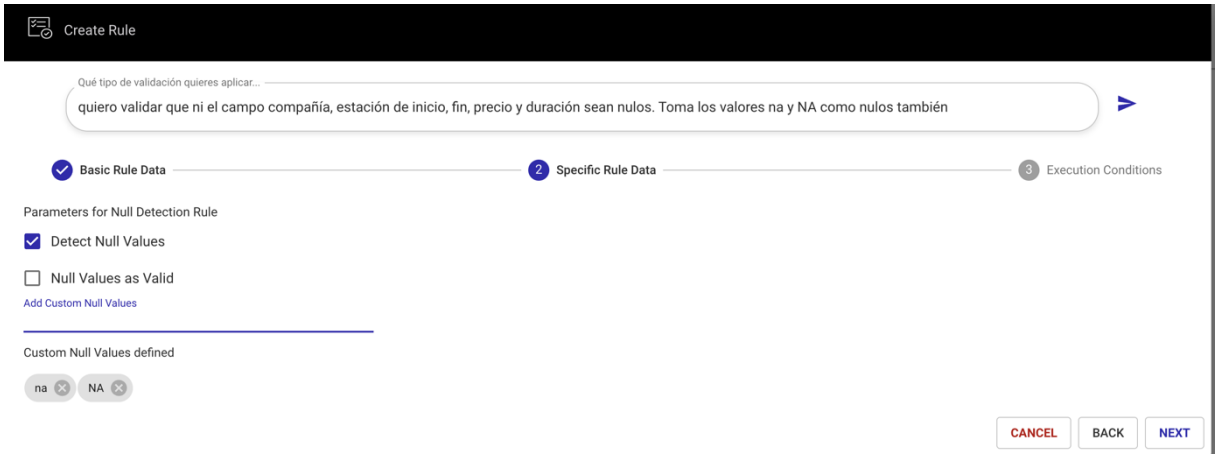
As can be seen, a null rule is created for the indicated fields:

It is important to note that **we have defined the columns somewhat loosely**, referring to start and end stations instead of origin and destination. As can be seen, the system correctly identifies the desired columns.



I will add an extra element, which is to mark some custom data as nulls, such as na and NA:

```
I want to validate that neither the company, start station, end station, price, nor duration fields are null. Treat the values na and NA as null as well
```



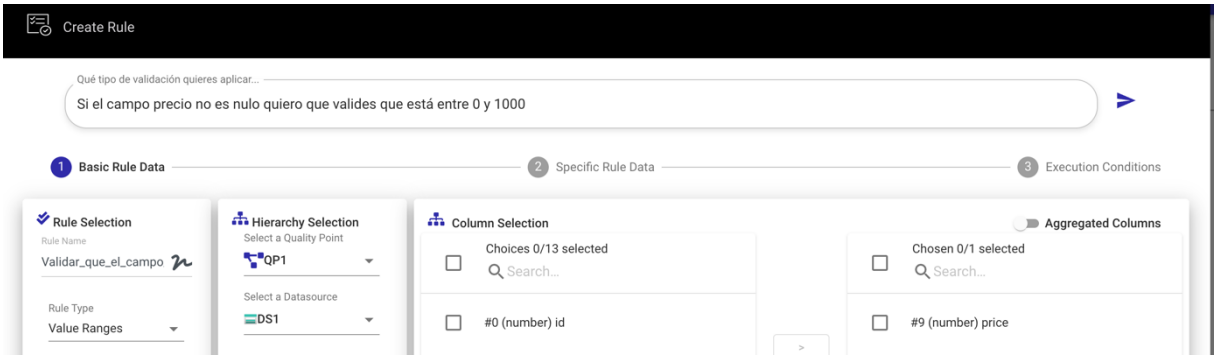
Notice how, in this case, the specific rule configuration includes the custom values.

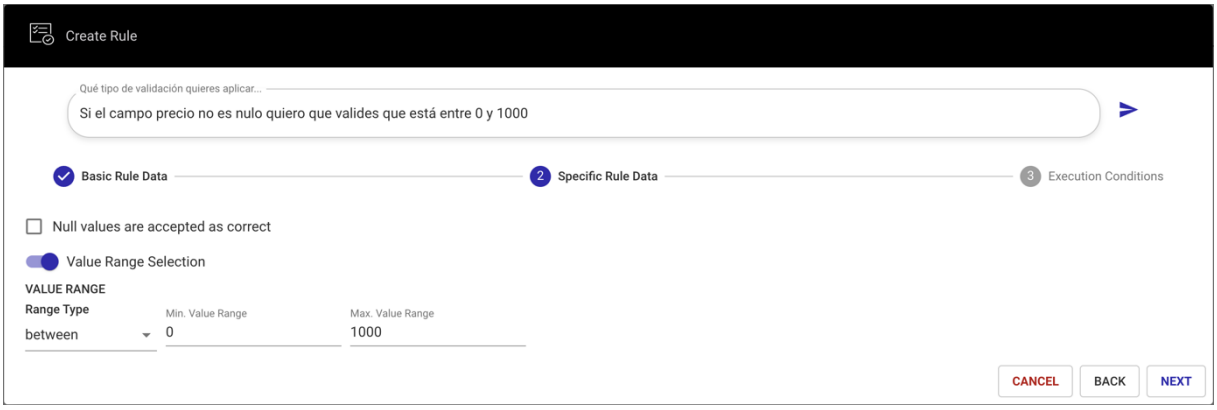
4.1.3 Validation with Execution Conditions

Now we will add an additional element, such as conditions:

```
If the price field is not null, I want you to validate that it is between 0 and 1000
```

In this case, it correctly creates the value ranges rule:





And in step 3 of the execution conditions, it creates the specified condition:

We will create a video demonstrating more advanced concepts, such as data aggregations, joins, and more complex conditions, to showcase the power of the solution.

5. Conclusions and Next Steps

From the tests conducted, we can proudly say that **we have created the first generative AI-driven system for the creation and deployment of quality policies**. However, achieving this required advancements in several areas, such as:

- **Structured Information Agent Architecture:** We developed an agent architecture capable of working with structured information at every step, ensuring consistency and accuracy in rule generation.
- **Defined and Reliable Execution Flow:** The architecture allows the generation of complex structures step-by-step, following a reliably defined execution flow. Each step is mapped to a specific action that ensures coherence and precision in rule creation.
- **Modular and Scalable Class and Tools Structure:** We designed a class and tools structure that facilitates the development and unit testing of the various steps and elements in the rule creation flow. This modularity allows the solution to scale and evolve easily and efficiently.
- **Complete End-to-End Integration:** A full end-to-end integration has been carried out, from rule generation via LLMs to user review and adjustment on the front end. This approach ensures a smooth user experience and a robust implementation.
- **Validation of Generative AI-Based Workflows:** We validated the feasibility of replacing traditional workflows with generative AI-based workflows. Testing with real datasets has shown that the solution is not only viable but also efficient and accurate.

With this, **we now have the integration of rule creation via Generative AI in OMMA**. This has allowed us to develop the first generative AI-driven quality policy creation system. Within OMMA, where one of our core principles is to facilitate the creation of quality policies without technical knowledge, **this represents a significant step forward, exponentially simplifying rule creation**.

Moreover, **within Mática, this has allowed us to validate our idea of evolving interfaces towards much more user-friendly solutions**. This advancement not only transforms how users interact with our tools but also positions us at the forefront of innovation in data quality management by effectively and practically incorporating artificial intelligence into our solutions.

However, this is just the beginning. It marks the first successful step in a much broader roadmap. So far, we have demonstrated an initial integration that replaces our rule creation forms. **But the system we have created not only allows for the creation of one rule at a time but also enables us to create multiple rules from a single instruction, which will be the next step**. Subsequently, we will also work on integrating this technology into many other interfaces, such as the intelligent rule recommender. Additionally, we will explore new types of rules, where a SQL statement can be introduced and the system itself converts it into a usable rule type in **OMMA**.

We are also making progress with the implementation of LangGraph to compare its efficiency with our current agent architecture and decide which is the best option for OMMA.

Finally, through this integration with our **GenAI Framework**, we will be able to deploy our solutions across various LLM platforms, adding observability and cost measurement capabilities, and ensuring that our solutions are robust, scalable, and adaptable to different environments. This initial development lays the foundation for the continuous transformation of our tools, making data quality management increasingly accessible, efficient, and powerful.

This initial development lays the foundation for the ongoing evolution of our tools, making data quality management more accessible and efficient.