University of Padova

Department of Mathematics "Tullio Levi-Civita"

Master Thesis in Data Science

Natural Language Interaction for a BI Platform and Data Integration

Task-Oriented, Retrieval-Augmented Agent Using LLMs

Supervisor
Prof. Gian Antonio Susto
University of Padova

Co-supervisor
Dr. David Dandolo
Statwolf

Master Candidate
FELICE FRANCARIO

Student ID
2097475

Academic Year
2024-2025

Abstract

This thesis focuses on the development of a chatbot designed to facilitate interaction with a BI platform using natural language commands. The primary objective was to create a system capable of interpreting user requests and translating them into database operations, thereby simplifying data access and management. The project utilized the LangChain library in Python to build an agent capable of executing queries and commands on a database. An incremental development approach was adopted, starting with a foundational model and progressively enhancing it with advanced functionalities to align with the structure of Statwolf's status queries.

The chatbot was developed during an internship at Statwolf, specifically tailored to interact with their datasources. It was designed to transform natural language queries into a structured dictionary-like format compatible with Statwolf's data query system. Additional functionalities were incorporated to support the creation of new metrics and dimensions through natural language commands, which were then translated into ClickHouse query definitions.

Contents

Listing of figures

1 Introduction

With the increasing reliance on large datasets, efficient tools to interact with databases are essential for simplifying data analysis and management. Traditional database interaction methods often required advanced technical expertise, creating a barrier for non-expert users. This thesis addresses this challenge by presenting a natural language processing (NLP)-powered chatbot capable of bridging the gap between user queries and database operations.

The core of the project involved using LangChain, a Python library, to develop an intelligent agent capable of translating natural language commands into executable database queries. The chatbot was specifically developed for Statwolf's platform, allowing users to interact seamlessly with their database infrastructure. The initial implementation translated natural language queries into dictionary-like structures for querying the Statwolf database. Subsequently, the project expanded to include the generation of new metrics and dimensions, enabling users to define these in natural language, which the system then converted into ClickHouse query definitions.

The thesis begins with an overview of large language models (LLMs) and agents, exploring their significance and the different types of agents, including the distinction between standard Langchain agents and LangGraph-based agents. The next chapter delves into the specifics of enabling natural language to SQL translation, emphasizing its utility, current practices, and the custom solutions developed for the Statwolf platform. Subsequent chapters explore schema customization, focusing on dimensions and metrics, culminating in a discussion of practical results demonstrated through examples and screenshots.

2

Large Language Models and Agents

2.1 Large Language Models

Large Language Models(LLM) have emerged as revolutionary tools in natural language processing (NLP), powering applications ranging from chatbots to advanced problem solving systems. These models are essentially based on deep learning architectures, specifically transformers, which allow them to process and comprehend text sequences with a high degree of efficiency. Models such as OpenAI's GPT and Google's BERT learn patterns from extensive datasets, capturing linguistic features that range from grammar and syntax to the deeper, nuanced meanings within language.

To explain in practical terms, LLMs are trained on vast amounts of textual data, absorbing patterns and relationships that allow them to predict subsequent words, answer questions, and even generate coherent and contextually relevant content. A critical feature of LLMs is their adaptability; through fine-tuning, they can specialize in specific domains such as medicine, law, or finance, enhancing their relevance to particular use cases. Another enhancement, reinforcement learning with human feedback (RLHF), ensures that their outputs align more closely with human expectations, improving both accuracy and usability.

Another important component of LLMs' success is their scalability. These models can process massive amounts of data, enabling breakthroughs in conversational AI, intelligent search engines, and automated code development—key examples of what agents can achieve. Their

ability to generate text that closely resembles human communication makes them essential for developing user-friendly interfaces, assisting users with complex tasks, and enhancing accessibility to sophisticated systems.

2.2 Agents

LLM Agents are systems designed to take on tasks autonomously by understanding user inputs and turning them into actions. They work as bridges between complex systems and people, making it easier to interact with technology. For example, conversational agents use advanced large language models (LLMs) to understand and respond in natural language, hiding the technical details and making high-level tech accessible to everyone. These agents can handle tasks like querying databases, managing APIs, or even navigating complex workflows, making processes smoother and reducing the need for technical expertise. From personal assistants to automated data management, agents are making a big impact by enhancing customer support, streamlining operations, and making advanced tech more approachable.

2.3 Types of LLM Agents

When discussing Large Language Model (LLM) agents, it's important to understand that they come in many different types, each suited for specific tasks and applications. Here's an overview of some of the most common ones:

2.3.1 Generative Agents

Generative agents are all about creating text based on the input they get. These models can generate everything from essays to poems and even code. They're particularly useful for content creation, whether that's writing blog posts, creating creative pieces, or helping with coding tasks.

2.3.2 Task-Oriented Agents

Task-oriented agents are designed with a specific goal in mind. Whether it's helping you book a flight, troubleshoot an issue, or manage your calendar, these agents are focused on completing concrete tasks. They are commonly seen in virtual assistants and customer support roles.

2.3.3 Retrieval-Augmented Agents

Retrieval-augmented agents are a bit more sophisticated—they combine the power of generative language models with information retrieval. This means they can fetch relevant information from external sources or databases to give more accurate answers. This is especially useful when external knowledge is needed to handle complex queries.

2.3.4 Conversational Agents

These are the agents designed to engage in natural, open-ended conversations. They can switch topics and adapt to the flow of a dialogue, making them great for simulating real human interactions. Whether for casual chatting or more in-depth discussions, conversational agents make communication feel more fluid and dynamic.

2.3.5 Knowledge-Based Agents

Knowledge-based agents are built to have deep expertise in specific domains. They usually connect to a structured knowledge base, allowing them to give highly accurate and reliable responses on specialized topics, from law to medicine to science.

2.3.6 Multi-Agent Systems

In multi-agent systems, you have several agents working together to achieve a shared goal. Each agent usually has its area of specialization, and they cooperate or communicate to solve complex problems. These systems are often used in scenarios like simulations or collaborative tasks that require different kinds of expertise.

2.3.7 Personalized Agents

Personalized agents focus on adapting to individual user preferences. Over time, these agents learn from interactions and provide increasingly customized responses, whether it's through personalized recommendations, tailored shopping advice, or adaptive learning experiences.

2.3.8 Embodied Agents

Embodied agents go beyond just text—they exist in virtual or physical forms, like avatars in virtual environments or robots that can physically interact with their surroundings. These agents

often combine language models with sensory inputs (e.g., sight or hearing) to interact more realistically with the world.

2.3.9 Interactive Agents (Human-in-the-loop)

Interactive agents are unique in that they involve a human in the decision-making process. These agents may ask for feedback or clarifications from users to ensure they're delivering the most accurate response. This is especially important in fields where human judgment is essential, like healthcare or high-stakes decision-making.

2.3.10 Cognitive Agents

Cognitive agents are designed to mimic human reasoning and decision-making. They can process information, make decisions, and adapt to changing conditions without needing constant human input. These agents are especially useful for complex systems where decision-making needs to be autonomous.

2.4 Task-Oriented Retrieval-Augmented Agent for Database Interaction

The chatbot developed in this thesis is a hybrid of task-oriented and retrieval-augmented agent architectures. This combination enabled it to process natural language commands, translate them into structured database queries, and execute these queries effectively within Statwolf's environment.

2.4.1 Task-Oriented Nature

Task-oriented agents are designed with a specific goal: to perform predefined tasks or actions based on user input. In this case, the chatbot's primary task is facilitating seamless database interactions. This is achieved by interpreting user requests in natural language and converting them into a format compatible with Statwolf's query system. The 2 Key features of task orientation in this agent are:

- Translating natural language commands into database queries.
- Supporting database-specific operations, such as creating new metrics or dimensions.

2.4.2 Retrieval-Augmented Capabilities

The retrieval-augmented component of the agent enables it to interface with structured databases and incorporate external knowledge into its operations. The retrieval-augmented design enhances the system's capabilities by querying the database for existing metrics and dimensions to ensure compatibility and validate user-provided inputs against the database schema.

2.5 LangChain

After discussing agents, it's natural to delve into the tools that make their implementation practical. For this project, we used LangChain—a Python framework designed to simplify and extend the functionality of large language models (LLMs). As a customizable wrapper, LangChain enables developers to adapt LLMs to diverse workflows and integrate them seamlessly into various applications. It bridges the gap between general-purpose language models and specific use cases by offering modular components for chaining tasks, managing memory, and connecting to external systems. This adaptability made LangChain the ideal choice for building a chatbot tailored to translating natural language inputs into structured database queries.

2.5.1 Core Functionalities of LangChain

LangChain offers a variety of tools and abstractions to streamline the development of LLM-powered applications. The framework is particularly useful for breaking down complex workflows into manageable, interconnected components. Its key functionalities include [2]:

- **Chains:** LangChain allows developers to create "chains" of operations where the output of one step serves as the input for the next. This modularity is crucial for multi-step workflows like parsing user input, retrieving data, and formatting results.
- **Memory Management:** LangChain includes memory components to enable context retention across interactions. This feature is essential for conversational agents, allowing them to maintain a sense of continuity during user interactions.
- **Integration with External Tools:** LangChain provides utilities for connecting LLMs to external data sources, APIs, and databases. This functionality ensures that applications can incorporate up-to-date and contextually relevant information.
- **Prompt Templates:** The framework simplifies the process of crafting reusable and dynamic prompts, which can be adapted based on the task or user input.

- **Agent Framework:** LangChain supports the creation of agents—dynamic systems that can decide which action to take based on user input and environmental factors. This aligns perfectly with the requirements of this project, where the chatbot must dynamically interpret natural language commands and execute database operations.

LangChain's modular design and extensive library of tools provided a solid foundation for the development of the chatbot, enabling efficient handling of natural language queries and seamless interaction with Statwolf's database infrastructure.

2.5.2 LangChain vs. LangGraph for Building Agents

Agents represent a significant evolution in this space, allowing LLMs to not only process data but also decide the sequence of steps in an application. Unlike a traditional chain that follows a predefined workflow, an agent provides a degree of autonomy, enabling it to dynamically route tasks, select appropriate tools, or even evaluate whether a response is sufficient or requires further refinement. The architecture of agents can vary widely, ranging from simple routers that choose between predefined options to fully autonomous systems capable of determining complex, multi-step workflows. However, this flexibility introduces a trade-off: as agents gain more control, the reliability of the system often decreases due to the inherent non-deterministic nature of LLMs and the potential for errors in decision making.

LangGraph, a more recent extension to Langchain, introduced in June 2024 [3], takes the above mentioned capabilities a step further. Designed as an independent framework, LangGraph emphasizes precision and flexibility, addressing challenges in creating workflows for real-world tasks. Unlike the legacy LangChain AgentExecutor, which abstracts many underlying details for simplicity, LangGraph offers developers granular control. This makes it particularly suitable for scenarios where the complexity of tasks demands intricate cognitive architectures. It allows the flow of an application to be expressed as a graph of interconnected nodes and edges, where each node represents a discrete action and can access or modify shared memory. The edges between nodes define the control flow, which can be deterministic or guided by conditional logic. This structured approach ensures that even as agents handle increasingly complex workflows, developers retain oversight and can mitigate errors.

Furthermore, LangGraph supports persistent graph states through short-term or long-term memory, enabling robust task tracking and facilitating human-agent collaboration. With features like conditional branching, looping, and hierarchical decision making, LangGraph allows for customized agentic behaviors tailored to specific domains [1]. In addition, its support for

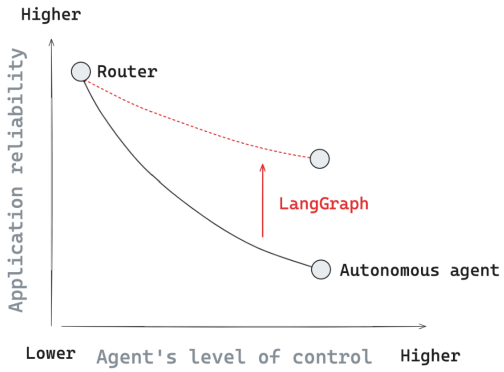

Figure 2.1: Reliability of LangGraph Agents [1]

human-agent collaboration, including intervention points and workflow rewiring, ensures reliability even in the most sensitive tasks. The persistence layer is particularly valuable, as it allows for interactive human oversight. Developers can pause an agent, review and edit its state, and approve subsequent steps before execution. This human-in-the-loop design ensures that agents remain aligned with user expectations, even in high-stakes or sensitive tasks. By balancing control and flexibility, LangGraph provides a powerful framework for building reliable agentic systems suited to real-world complexities.

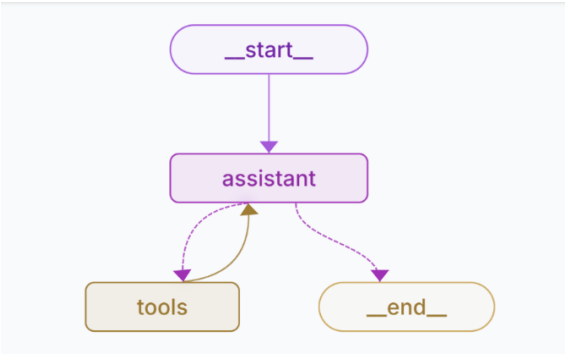

Figure 2.2: Node Graph of a basic agent

Despite the potential advantages of LangGraph, this project opted to use LangChain's legacy AgentExecutor. This decision was guided by practical considerations within Statwolf's infrastructure. LangGraph introduced dependencies that were incompatible with the existing packages on Statwolf's servers, and reconfiguring the codebase to accommodate these changes would have been resource intensive. Furthermore, the enhanced functionalities of LangGraph, while impressive, were not essential for the scope of this project. The tasks required—such

as translating natural language queries into structured database operations—were effectively handled by the AgentExecutor, which provided the necessary balance of simplicity and functionality.

Using LangChain's legacy AgentExecutor, the chatbot was successfully developed to facilitate database interactions in a manner that aligned with Statwolf's technical ecosystem. It enabled natural language queries to be converted into actionable commands without compromising efficiency or reliability. Although LangGraph offers exciting possibilities for future development, the decision to rely on LangChain's established tools ensured a streamlined and effective implementation within the constraints of this project. Future iterations may explore LangGraph as system requirements evolve, potentially unlocking new dimensions of agentic workflows.

3

Agents for Natural Language Database Queries

3.1 Natural Language to SQL

The transformation of natural language into Structured Query Language (SQL) is an important step in making databases more accessible and user-friendly. By allowing people to ask questions and retrieve insights using everyday language, this approach breaks down barriers for non-technical users. It provides a way to interact with data that feels natural and intuitive, opening up new possibilities for decision-making and exploration. It also leads to a significant reduction in the time and effort required to gain insights from data. Additionally, this approach helps experts focus on interpreting results and solving problems rather than getting bogged down by syntax. Overall, natural language to SQL bridges the gap between technical complexity and ease of use.

This technology has many practical applications. In business intelligence, for example, it allows analysts to quickly generate custom reports and explore data without needing to rely on IT specialists. Customer support systems also benefit from this capability, as it enables them to resolve issues faster by automatically querying databases based on user input. Even in education, natural language to SQL is being used to help students and professionals learn how to interact with data in a hands-on way.

3.1.1 How Does Natural Language to SQL Work?

Understanding how natural language to SQL converters work requires breaking down their functionality into distinct stages, each playing a critical role in translating user input into actionable database queries.

The process begins with input understanding. When a user provides a natural language query, the system analyzes it to identify key elements such as keywords, entities, and relationships. This step relies heavily on natural language processing techniques like tokenization, part-of-speech tagging, and named entity recognition. For instance, in a query like "Show me the sales for last quarter," the system must recognize "sales" as the primary subject, "last quarter" as a temporal condition, and "show me" as an indicator of data retrieval.

Once the input is parsed, the next step is query mapping. Here, the natural language components are aligned with corresponding SQL elements. Nouns in the query often map to database table names or columns, while verbs correspond to SQL operations like SELECT, INSERT, UPDATE, or DELETE. Other parts of the query, such as adjectives and adverbs, might define conditions or parameters. For example, the word "highest" in a query could signal the need for an ORDER BY clause with a descending sort. This stage requires a deep understanding of both the database schema and the semantics of the user's query.

The third stage is query generation. After mapping the elements, the system constructs a valid SQL query that faithfully represents the user's intent. This involves combining the mapped components into a syntactically correct SQL statement. For example, the query "What were the top three products last month?" could be translated into something like:

```
SELECT product_name, SUM(sales) AS total_sales
FROM sales_data
WHERE sale_date BETWEEN '2024-12-01' AND '2024-12-31'
GROUP BY product_name
ORDER BY total_sales DESC
LIMIT 3;
```

Finally, the generated SQL query is executed against the target database. The system retrieves the results and presents them to the user in a readable format, completing the interaction. Depending on the implementation, these results might be visualized in tables, charts, or even narrative summaries.

This process highlights the complex interplay between language understanding, database schema knowledge, and SQL generation. As these systems evolve, their ability to interpret

nuanced queries and adapt to diverse schemas will continue to improve, making data more accessible to everyone.

3.1.2 State of the Art in Natural Language to SQL

The development of Natural Language to SQL (NL2SQL) systems has undergone significant evolution over the years, moving from rule-based approaches to advanced solutions powered by large language models (LLMs). Early methods relied heavily on predefined templates and handcrafted rules to interpret queries and generate SQL statements. These approaches, though foundational, were limited in their adaptability and scalability.

With the advent of neural network-based methods, a new era of NL2SQL began. These methods leveraged large-scale benchmark datasets like Spider and WikiSQL to train sequence-to-sequence models. For instance, models like Seq2SQL[4] and IRNet [5] introduced mechanisms to encode both natural language queries and database schemas, enabling a more dynamic mapping between the two. However, challenges such as handling complex queries with nested SQL constructs remained a significant hurdle.

The introduction of pre-trained language models (PLMs) like BERT and T5 marked a turning point in NL2SQL development. These models, fine-tuned on task-specific datasets, significantly improved the system's ability to understand and generate SQL queries. More recently, large language models (LLMs) like GPT-4 have further revolutionized the field. These models bring advanced language understanding and reasoning capabilities, allowing for context-aware query generation and handling of complex operations such as JOINs and subqueries. Prompt engineering has become a crucial technique for leveraging these models effectively. For instance, methods like DAILSQL and DINSQL use carefully designed prompts to guide the model's SQL generation process, achieving state-of-the-art performance on benchmarks like Spider.

Despite these advancements, there is no clear winner among NL2SQL methods[6], as different approaches excel in different scenarios. LLM-based methods, while powerful, often require substantial computational resources and are sensitive to domain-specific nuances. On the other hand, PLM-based methods offer a balance of efficiency and accuracy, particularly when fine-tuned on specific datasets.

Future research in this field aims to address several challenges, such as improving domain adaptation, handling ambiguous queries, and reducing computational costs. Looking ahead, the future of natural language to SQL lies in enhancing contextual understanding and integrating more metadata and schema information into these systems. Researchers are also working

on better evaluation methods to measure the accuracy and efficiency of these models.

3.2 The Statwolf Platform

Statwolf is a data analytics platform founded in 2014 in Dublin, Ireland, with the goal of making data science an integral part of business decision-making. With offices in Padua, Italy, the platform has grown to include a team of 19 members comprising data scientists and software developers. Statwolf provides cloud-based AI systems and tools for transitioning to data-driven solutions in cloud and IoT scenarios.

At its core, Statwolf offers an end-to-end proprietary data and AI platform designed to integrate seamlessly with diverse data sources and existing software systems. This flexibility is a cornerstone of its design, enabling businesses to adopt its solutions without overhauling their current infrastructure. Additionally, the platform is big-data and IoT-ready, leveraging advanced technologies for data retrieval, storage, processing, and handling.

One of the platform's strengths is its modularity. Statwolf emphasizes interoperability, ensuring that its solutions complement existing workflows and tools within an organization. This modular approach empowers users to harness data solutions fully, tailoring them to specific business needs.The platform has been successfully applied in various domains, including manufacturing, digital marketing, and IoT-connected appliances. For example, Statwolf has enabled predictive maintenance systems that optimize intervention schedules, reducing service costs and minimizing unexpected downtimes.

Statwolf's functionalities span several key areas:

3.2.1 Data Integration

Data integration is one of Statwolf's most critical functionalities, serving as the backbone for generating meaningful insights. The platform is designed to gather, harmonize, and manage data from multiple sources efficiently. It supports a variety of data formats and ensures smooth integration with external databases, APIs, and IoT devices. This project focused on the task of transform natural language inputs into structured dictionaries, which can then be processed by Statwolf's SQL generation systems. While traditional NL2SQL systems focus on generating SQL queries directly, Statwolf's approach involves creating structured dictionaries that feed into its existing SQL generation methods. This distinction simplifies the integration process and ensures compatibility with the platform's established workflows.

3.2.2 Data Modeling

Statwolf excels in data modeling by providing tools to define and manage metrics and dimensions tailored to specific business needs. Data modeling is essential for structuring raw data into meaningful formats that support advanced analysis and visualization. This capability is particularly relevant to this thesis, where generating personalized metrics and dimensions through natural language was also a core part of the task.

3.2.3 Analytics and Visualization

Statwolf's analytics and visualization capabilities enable users to explore their data interactively. Through intuitive dashboards, users can create custom reports, identify trends, and monitor key performance indicators(KPI's). The visualization components are tightly integrated with the data integration and modeling layers, providing a cohesive and streamlined analytical experience.

3.2.4 Decision Support Systems

Statwolf's Decision Support Systems offer functionalities such as interactive reports with customizable alerts, keeping users informed about critical metrics or anomalies without the need for constant monitoring. The platform also provides self-service analytics, empowering end-users to independently explore data and generate insights tailored to their needs, reducing reliance on IT or data teams. Additionally, collaboration tools with fine-grained permissions enable secure sharing of dashboards, reports, and datasets, promoting teamwork while ensuring data security.

3.3 Natural Language to Statwolf status: Query Generator

This section delves into the creation of an agent designed to transform natural language queries into a structured format. This structured output is subsequently converted into ClickHouse SQL queries by the existing Statwolf infrastructure. This intermediate format, known as a "Statwolf status" captures the core components of a query (dimensions, metrics, filters, and timeframe) in a JSON structure.

Below is an example of a Statwolf status:

```
{
  "table": "all_devices_predictions",
  "filter": {
    "field_0": "pred",
    "selector_0": "AND",
    "operator_0": ">",
    "value_0": 20
  },
  "timeframe": {
    "period": "last7days",
    "granularity": "day"
  },
  "granularity": "overall",
  "dimensions": ["device_name"],
  "metrics": ["avg_pred"],
  "crosstab": [],
  "sort": [],
  "take": "5000",
  "testing": {},
  "params": {}
}
```

The following sections outline the steps implemented to create this agent, detailing each aspect of the process.

3.3.1 Parsing User query

The initial step involves understanding and extracting key components from the user's query and clearly expand on the user's intent before passing it into main agent. A key feature of this step is ambiguity handling, where the agent makes reasonable assumptions and documents them explicitly.

The following is a section of the first prompt template used to expand on the query:

```
Task:
```

```
You are an interpretation agent whose job is to thoroughly analyze the user's
     query and capture every detail, including any specific dimensions,
    metrics, filters, and timeframe elements. Ensure nothing is overlooked,
    and clarify any potentially ambiguous aspects.

Input Query: {query}

Steps:

1. **Analyze the Query in Full**:
Thought: Carefully read through the user's query to understand the full
    context and extract all essential details, even if they are implied
    rather than explicit.
Action: Only if the user's query is impossible or nonsense, correct it by
    giving your best inuitive guess to what the user's actual request is in
    the Query description part of your output.

2. **Describe the Key Information**:
   In your response, explain each element you have identified, including:
   - Key entities or subjects in the query
   - All The possible metrics that seem relevant, if any
   - All The possible dimensions that seem relevant
   - Any potential filters, including for dimensions, values, and timeframes
   - Any clarifications or assumptions you made to fill in details if the
       query is ambiguous.
```

For the full prompt template refer to the Appendix.

Another important aspect was the clarification of the definitions of "Dimensions" and "Metrics" in the context of Statwolf's platform:

```
Important: Make sure to not confuse dimensions with metrics.
Dimensions represent individual data values that categorize, identify, or
    provide context without any aggregation. They capture the foundational
    details of each data instance.
Example: pred (predictions) is a dimension because it holds the raw,
    individual values generated by a model for each data point. Treating pred
```

```
    as a dimension allows each prediction to stand alone and be organized,
    filtered, or grouped alongside other dimensions in the dataset.
Metrics are aggregated calculations applied to one or more dimensions.
    Metrics summarize or quantify aspects of the data through operations like
    averaging, summing, or counting.
Example: avg_pred (average of predictions) is a metric because it represents
    the computed mean value of all pred instances. 'Its derived from pred and
    provides a consolidated, higher-level view of the data rather than a
    single instance.
```

3.3.2 Initial Selector Agent

After parsing the user query, it is passed to an agent which extracts information from the expanded query and a database schema to create an initial structured output highlighting dimensions, metrics, filters and the timeframe with the following structure:

```
{
  "dimensions": ["dimension_1", "dimension_2", ...],
  "metrics": ["metric_1", "metric_2", ...],
  "filter": [
    ["dimension_name_to_filter", "operator", "value"]
  ],
  "timeframe": {
    "granularity": "day",
    "period": "last7days"
  }
}
```

This rough structured output will then be corrected/refined to match the expected statwolf status structure.

Here are all steps taken by the initial agent, highlighting the relevant parts of its prompt template and all its tools:

Retrieving the Dataset Schema

To tailor the output to the available data, the agent starts by retrieving the dataset schema using the *describe_source* tool(a function that is executable by the agent). This action provides a list of available dimensions and metrics and their respective data types. This ensures that the subsequent steps are aligned with the data source's constraints. The full code of the tools can be found in the Appendix. Here is the first step present in the agents prompt template:

```
Step 1: Get the complete list of dimensions and metrics of the dataset
Action: Use the describe_source tool to retrieve the schema.
Action Input: 'source_name':{source_name},'host':{host}, 'log_directory_path
    ': {log_directory_path}, username: {username}, password: {password} .
Observation: The output of the describe_source tool is a list of dictionaries
     where each dictionary represents a data field. Each dictionary has three
     keys: 'name' (field name), 'data_type' (the type of data), and 'type' (
    either dimension or metric).
```

Understanding the User's Query

In this step the agent analyzes the user's query to extract the core objective. This involves identifying any dimensions and metrics of interest, any explicit or implicit filters and the desired timeframe.

Here is the second step present in the agents prompt template:

```
Step 2: Understand the user's request
Action: Analyze the user's query carefully and determine the core objective.
Thought: I will now assess which dimensions, metrics and filters are
    necessary to respond to the query effectively.
```

Selecting Dimensions and Metrics

Using the schema retrieved in Step 1, the agent selects only the relevant and corresponding dimensions and metrics necessary to answer the query.

Here is the third step present in the agents prompt template:

```
Step 3: Select dimensions and metrics
```

```
Thought: Select only the necessary dimensions and metrics from the dataset.
Consider the following:
Only include dimensions and metrics that are directly relevant to answering
    the question.
Use the the given query and the reasoning provided in the Query description
    to select the most likely corresponing dimensions/metrics from the
    dataset.
Do not include unnecessary dimensions or metrics.
Ensure ALL selected dimensions and metrics exist in the schema you have.
In the schema dictionary, you have a clear indication of whether a specific
    field is a dimension or a metric in the "type" field.
Be careful to consider only the fields where the "type" is marked as "metric"
     when selecting metrics, and only those marked as "dimension" when
    selecting dimensions.
Ensure that you respect these classifications strictly.
```

Validating Dimensions and Metrics

The agent validates the selected dimensions and metrics by calling a manually created *check_dimensions_and_metrics* tool. For the full code of the tool, refer to the Appendix. A verification step ensures that all selected fields exist in the schema and the classifications (dimension/metric) are respected.This ensures compatibility with the dataset. If validation fails, the agent re-evaluates the query and refines its selections.

Here is the fourth step present in the agents prompt template:

```
Step 4: Check dimensions and metrics
To be sure that the dimensions and the metrics are acceptable you must use
    the check_dimensions_and_metrics tool which require as input the json.
    dumps of a dictioanry with:
- 'list_of_dimensions':the list of dimension you selected
- 'list_of_metrics':the list of metrics you selected (You may set it as an
    empty list if no metrics were selected)
- 'source_name':{source_name}
- 'host': {host}
- 'log_directory_path': {log_directory_path}
```

```
If the check_dimensions_and_metrics tool returns True it means that you have
    selected dimensions and metrics that are acceptable in the data and you
    may proceed with step 5. Only if check_dimensions_and_metrics tool
    returns False go back to step 2 and reflect again on the appropriate list
     of dimensions and metrics to select based on the original query.
```

Filtering

Once the relevant dimensions and metrics are selected, the agent creates a list of filters based on the prompt indications and the user's query.

Here is the fifth step present in the agents prompt template:

```
Step 5: Filtering
Thought: Apply filters based on the user's request.

You must capture and apply all relevant filters based on the user's question,
     including those for time periods and dimensions.
Filters related to dimensions and variables should be structured as follows:
    ['dimension_name_to_filter', 'operator', 'value/values'].
Time-related filters must be managed separately from dimension filters. Time-
    related filters should be placed in a second dictionary called "timeframe
    " with the following structure:
"granularity": specifies whether the granularity is day or month.
"period" : the period of the analysis, as example : "last 7 days", "last year
    ", "last 3 months", "from datetime_start to datetime_end" where
    datetime_start is the starting datetime of the analysis and datetime_end
    is the ending datetime for the analysis.

The allowed operators depend on the data type of the field being filtered.
    The following are the acceptable operators for each data type:
- 'String': ['contains', 'in', 'notIn', 'regex', 'notRegex']
- 'Array(String)': ['contains', 'containsAny', 'notContainsAny']
- 'Date': ['timeframe', 'dateIsGreaterOrEqual', 'dateIsGreater', '
    dateIsLowerOrEqual', 'dateIsLower', 'dateIsEqual']
- 'Float32', 'Float64', 'UInt32', 'UInt64', 'Int32', 'Int64': ['==', '>=',
    '<=', '>', '<', '!=']
```

Structuring the Output

The agent prepares the final output as a structured dictionary.

Here is the sixth step present in the agents prompt template:

```
Step 6:
Thought: Prepare the output.
Provide the final list of dimensions, metrics, and filters as a dictionary
    with the following structure:
"dimensions": a list of relevant dimensions (excluding 'timestamp' unless
    explicitly asked for in the user's query)
"metrics": a list of requested metrics.
"filter": a list of filters to be applied.
"timeframe": description of the timeframe for the analysis.
Ensure the final result is formatted as a clean dictionary as a single string
     in JSON format.
```

Finalizing and Returning the Output

It ensures compliance with JSON standards using a manually created *json_correction* tool. For the full code, refer to the Appendix. Examples of input-output pairs were also provided at the end of the prompt substantially increasing the accuracy by few-shot prompting.

Here is the last step present in the agents prompt template:

```
Notes:
You must capture all relevant filters specified by the user, both for
    dimensions/metrics and time periods.
Use the json_correction tool before returning the final answer to ensure
    correct JSON formatting.
Please return ONLY the transformed string without any additional comment or
    explanation.
```

The output of the first agent has the following structure:

```
{
  "dimensions": ["dimension_1", "dimension_2", ...],
  "metrics": ["metric_1", "metric_2", ...],
  "filter": [
    ["dimension_name_to_filter", "operator", "value"]
  ],
  "timeframe": {
    "granularity": "day",
    "period": "last7days"
  }
}
```

3.3.3 Filter corrector

In this section, the process of refining and structuring the filters extracted from the output of the initial selector agent is detailed. The goal of this step is to transform the raw filters into a format that adheres to strict syntactic and logical rules of statwolf's statuses, ensuring compatibility with downstream processing. This refinement is achieved by passing the filters into another LLM with a carefully crafted prompt template. The LLM receives as input the raw filters generated by the initial agent and the original expanded query for context. For the full prompt template, refer to the Appendix.

Here are some sections of the prompt template that was used:

```
All filters must be included in a dictionary called 'filter'.
For each filter, create a unique key using a progressive index (indexcount),
    starting from 0 and incrementing by 1 for each new filter.
Each filter must follow this structure:
field_indexcount: The name of the dimension/metric to filter.
operator_indexcount: The operator to apply based on the data type of the
    field.
value_indexcount: The value(s) for filtering (could be a single value or a
    list).
selector_indexcount: Logical selector ('AND' or 'OR') to specify how the
    current filter connects to the next one.
```

```
For the operator_indexcount, the allowed operators depend on the data type of
     the field being filtered. Use the following list of acceptable operators
     for each data type:

- 'String': ['contains', 'in', 'notIn', 'regex', 'notRegex']
- 'Array(String)': ['contains', 'containsAny', 'notContainsAny']
- 'Date': ['timeframe', 'dateIsGreaterOrEqual', 'dateIsGreater', '
    dateIsLowerOrEqual', 'dateIsLower', 'dateIsEqual']
- 'Float32', 'Float64', 'UInt32', 'UInt64', 'Int32', 'Int64': ['==', '>=',
    '<=', '>', '<', '!=']

For the selector_indexcount, make sure to use the original query as context.
```

An example of the expected output is also provided in addition to some important notes that must be followed in order to receive an ideal output.

```
Important Notes:
Ensure that each filter is properly structured using the progressive
    indexcount.
Apply the correct operator based on the field's data type.
Use the orginal query information only as context only for determining the
    selector associated to each field.
YOU ARE NOT ALLOWED TO CHANGE THE NAME OR VALUE OF ANY OF ANY OF THE fields
    or values. You may only modify the opertor if necessary according to the
    above mentioned rules.
Return ONLY the filters dictionary in a format that can be directly loaded
    using the Python json library (without any additional quotes, backticks,
    or JSON formatting).
```

Since *selector_indexcount* specifies how the current filter connects to the next one, through either 'OR' or 'AND'. Another important correction is applied manually by ensuring the last *selector* among the filters is always set to 'AND' to avoid system errors.

3.3.4 Timeframe corrector

This section focuses on refining and structuring the timeframe outputed by the initial selector agent. The aim is to standardize the timeframe structure, ensuring it aligns with predefined

rules for granularity and periods. The transformation is carried out by passing the timeframe field through another LLM using a tailored prompt template.

The prompt template used is the following:

```
You will be given a dictionary with various fields. Our task is to modify the
     timeframe field, which is a dictionary.
We need to modify the timeframe field, creating a new one with the following
    structure:
"granularity" :  "day"|"month"
"period": With the following values,
if granularity is "month", it can take one of "alltime","currentYear","
    lastYear","months","lastXMonths","months",
if granularity is "day" it can take "last3days","last7days","last14days","
    last30days","last90days","currentMonth","lastMonth","lastXDays","days".

If the input dictionary is empty assign the following default values:
    "period": "alltime",
    "granularity": "month"

Example of output corresponding to the last 7 days:
    "period": "last7days",
    "granularity": "day"

Only in the cases where "period": "days"|"months", where specific date time
    intervals are needed, you add the following keys:

"dateFrom": "YYYY-MM-DD", #start date
"dateTo": "YYYY-MM-DD", #end date
"timeFrom": "hh:mm", #start time
"timeTo": "hh:mm" #end time

In cases where "period":"lastXDays"|"lastXMonths", add the following key
"auxval": Number of months or days if "lastXMonths" or "lastXDays" is
    selected

The period field in the input dictionary determines the time range. This can
```

```
    either be:
A start date and end date: use these values directly for dateFrom and dateTo.
A text-based description: map the input value to one of the predefined
    periods.

The granularity field in the input dictionary specifies the analysis level
    and should be used to fill both the granularity field.
If necessary,use the period field of the input dictionary to set:
"dateFrom": start date
"dateTo": end date
"timeFrom": start time
"timeTo": end time

Note:
Return ONLY the dictionary in a format that can be directly loaded using the
    Python json library (without any additional quotes, backticks, or JSON
    formatting).
Today datetime is {day}. Return the input dictionary filters with the new
    modified timeframe field.

The input dictionary is the following:
```

To make sure that the structure and values are respected, the model was programmed to return a structured dictionary with predefined keys and values using langchain's *with_structured_output()* method. For more details refer to the code in the Appendix.

3.3.5 Validation of Filter values

The validation of filter values is a crucial step in ensuring the compatibility of user-provided inputs with the predefined categorical fields of the Statwolf system. This step addresses potential discrepancies in the filter values, such as spelling errors, capitalization inconsistencies, or minor formatting issues, which might otherwise hinder the smooth execution of queries.

The process begins by defining a function that extracts the unique values of the categorical fields present in the filters derived from the previous step. These extracted values are then passed into an LLM for validation and correction.

The prompt used for validation is designed to ensure that the corrected output list preserves

the original length and order of the input values. Each input value is replaced by its most similar counterpart from the list of distinct values, determined based on semantic similarity or contextual relevance. Importantly, the distinct values list remains unaltered, even if it contains apparent anomalies, such as extra spaces or unusual capitalization. This approach guarantees that the corrected values strictly conform to the predefined valid categories, ensuring their usability within the Statwolf system.

The prompt used is the following:

```
Given the following input list of values: {value}, identify and replace each
    value
with the most similar value from the provided list of distinct values: {
    distinct_values}.
The replacement must follow these rules:
1. The output list must have the **same number of values** as the input list
    and
   maintain the **original order** of the input values.
2. For each value in the input list, select the **most similar value** from
    the distinct
   values list. Similarity should be based on semantic meaning, context, or
       other criteria.
3. The output values must strictly match the **exact format** of the selected
     values
   from the distinct values list, including capitalization, spacing, and any
       formatting quirks.
4. **Do not modify or "fix"** the values in the distinct values list under
    any circumstances,
   even if they appear to contain errors such as extra spaces.

Example:

Input list of values:
    ['up', 'down']
List of distinct values to select from:
    [' alta', 'bassa', ' lato']
Output:
    [' alta', 'bassa']
```

```
IMPORTANT:
- Ensure the output list maintains the **same length and order** as the input
    list.
- Each input value must be replaced by its most similar match from the
   distinct values list.
-DON'T ADD ANY ADDITIONAL TEXT, JUST RETURN THE OUTPUT LIST AND NOTHING ELSE
```

This validation approach offers several advantages. Firstly, it handles minor input errors efficiently, ensuring compatibility without manual intervention. Secondly, it maintains the original order of the input values, preserving the user's intent. Lastly, it outputs results that are directly usable by the system without requiring further adjustments, thus streamlining the workflow and reducing the likelihood of errors in subsequent processes.

Another important feature that was added to this step was manually removing dimensions from the output dimension list in cases when filtered to only one value. This standardizes the output by removing unnecessary columns from the final table.

By incorporating this validation mechanism, the filters become more robust and aligned with the system's requirements. This ensures that user inputs, even when imperfect, are transformed into formats that facilitate seamless processing and analysis, enhancing both the accuracy and reliability of the overall pipeline.

3.4 Performance results

3.4.1 Testing method

The performance of the initial query generator was evaluated using a structured testing approach. A dataset of seven natural language queries of varying complexity was created. Each query was rewritten into three alternate forms by a large language model (LLM) using the following prompt:

```
Given the following query:

{input}

Return a list of 3 alternate ways to rewrite the same query. Return only the
    list [] with queries in " " separated by commas. Make sure that the
```

```
returned list of queries make the same logical sense as the input query.
Don't return any query that has less information or that may be
misinterpreted compared to the original query.
```

The prompt instructed the model to provide a list of alternate phrasings that preserved the original query's intent while avoiding oversimplification or ambiguity.

The generated outputs were evaluated against predefined structured dictionary manually created in Statwolf's platform. The test queries included a diverse range of requirements, such as complex filters, metric aggregation, and specific temporal constraints. The accuracy of the query generator was assessed by comparing its outputs to the expected dictionaries. Additionally, query generation time was recorded to evaluate performance efficiency.

3.4.2 Results

The query generator's accuracy varied depending on whether query expansion was incorporated. With expansion(parsing), the generator achieved an accuracy of approximately 90%, occasionally reaching 95%. Without this step, accuracy dropped to a maximum of 85%. Query expansion clarified ambiguous or underspecified inputs by transforming them into more explicit forms, thereby enhancing the likelihood of generating correct outputs. Common errors in the absence of query expansion included misidentifying metrics as dimensions (e.g., selecting *avg_pred* instead of *pred*) and selecting incorrect dimensions with similar names (e.g., choosing *compressor_status_changed*, *state*, or *compressor_1on1* instead of the intended *compressor_state*). These issues persisted, albeit less frequently, even with query expansion, suggesting that further refinements, such as including descriptive metadata for metrics and dimensions, are essential for mitigating such errors.

Step 4 of the main agent prompt, the dimension and metric check, had to be refined numerous times to achieve the above mentioned results. With the tool not being properly defined, the initial accuracies were around 40%, due to an incorrect selection of dimensions and metrics not present in the schema. The prompt template was also modified numerous times, in addition to setting a maximum number of iterations to 10 to avoid cases of infinite loops. The evaluation of filter values, the last step of the pipeline, also had a significant impact on the overall accuracy by fixing possible incorrect value assignments by retrieving the closest semantically similar values directly through an SQL directly from the datasource. This step was crucial depending on the format of the input queries given as input, giving users more freedom to not specify the exact value but anything semantically similar or related. Without this step even a

small spelling mistake in the value would create obvious errors. Initially the selection of values was done through calculating the most similar word embeddings, but directly using an LLM proved to be much more effective. Embedding comparison was abandoned due to some obvious discrepancies between the closest embedding similarities and the expected values.

In terms of performance, the query generator required an average of 18 seconds per query when query expansion was included, compared to 13 seconds without it. While query expansion significantly improved accuracy, it introduced additional computational overhead, increasing query generation times.

3.4.3 Failure cases

Out of the 21 queries generated (7 queries, each with 3 rewritten forms), 2 failed to produce the correct structured dictionaries, resulting in an overall accuracy of 90.48%. The following examples illustrate these failures in detail.

The first failure involved the query written in Italian:

"Mostrami la richiesta media di potenza e la percentuale di anomalie per ogni tipo di dispositivo con compressione accesa negli ultimi 2 mesi."

In English, this translates to:

"Show me the average power request and the anomaly percentage for each type of device with compression on in the last 2 months."

The expected output was a dictionary that included metrics for average power request and average anomaly percentage, with a filter specifying devices with their compression state set to "On." The correct output should have been:

```
{'filter': {'compressor_state': [{'selector': 'AND', 'operator': 'in', 'value
    ': ['On']}]}, 'timeframe': {'period': 'lastXMonths', 'granularity': '
    month'}, 'dimensions': ['device_name'], 'metrics': ['avg_is_anomaly', '
    avg_power_request']}
```

However, the generated output included the wrong metric:

```
{'dimensions': ['device_name'], 'metrics': ['avg_anomaly_score', '
    avg_power_request'], 'filter': {'compressor_state': [{'selector': 'AND',
    'operator': 'in', 'value': ['On']}]}, 'timeframe': {'period': '
    lastXMonths', 'granularity': 'month'}}
```

Although the filter and timeframe were correct, the generator selected *avg_anomaly_score* instead of the expected *avg_is_anomaly* metric. This is the most common issue of the agent. Fully addressing it would require adding detailed metadata descriptions to the schema to help differentiate between similarly named metrics.

The second failure involved another Italian query:

"Mostrami la media delle predizioni, la percentuale di anomalie e il punteggio medio di anomalia da gennaio ad agosto 2024, per ognuno i dispositivi con la prima causa 'bassa portata in CH'."

This translates to:

"Show me the average predictions, the anomaly percentage, and the average anomaly score from January to August 2024, for each device with the primary cause 'bassa portata in CH'."

The expected dictionary included a filter for devices where the primary cause was "bassa portata in CH" metrics for average predictions, anomaly percentage, and average anomaly score, as well as device_name as a dimension. The correct output should have been:

```
{'filter': {'Cause1': [{'selector': 'AND', 'operator': 'in', 'value': ['
    bassa portata in CH']}]}, 'timeframe': {'period': 'months', 'granularity
    ': 'month'}, 'dimensions': ['device_name'], 'metrics': ['
    avg_anomaly_score', 'avg_is_anomaly', 'avg_pred']}
```

However, the generated dictionary missed the device_name dimension:

```
{'dimensions': [], 'metrics': ['avg_anomaly_score', 'avg_is_anomaly', '
    avg_pred'], 'filter': {'Cause1': [{'selector': 'AND', 'operator': 'in', '
    value': [' bassa portata in CH']}]}, 'timeframe': {'period': 'months', '
    granularity': 'month'}}
```

While the filter, metrics, and timeframe were accurate, the omission of the dimension field resulted in an incomplete output. This could be because of the wording of the prompt and shows that even if we add metadata descriptions to the schema, the agent will still not be 100% perfect due to its probabilistic nature. It is worth noting that for this project 'gpt-4o-mini', openAI's most cost efficient small model, was used due to it being an order of magnitude more affordable than previous frontier models and more than 60% cheaper than GPT-3.5 Turbo[7].

—

In conclusion, these failure cases illustrate where the query generator can struggle, particularly in selecting the correct metrics and accurately mapping dimensions among many similarly

named metrics/dimensions. Enhancing the schema with detailed metadata and fine-tuning the input prompts could mitigate these issues and improve its overall performance.

3.4.4 Agent in action

Here the complete mechanism of the query generator will be shown. For the examples we will be working on a devices predictions datasource.

Example 1: Suppose we want to know the average of all the predictions greater than 10 of the variable 'fan' in the last 30 days. We could input the following natural language query:

Give me the average of all predictions greater than 10 of the variable 'fan' in the last 30 days

The query generator outputs the following structured statwolf Status:

```
{  "filter": {
  "field_0": "pred",
  "selector_0": "AND",
  "operator_0": ">",
  "value_0": 10,
  "field_1": "variable",
  "selector_1": "AND",
  "operator_1": "in",
  "value_1": ["fan"]
},
"timeframe": {
  "period": "last30Days",
  "granularity": "day",
  "dateFrom": "2025-01-09",
  "dateTo": "2025-02-07",
  "timeFrom": "00:00",
  "timeTo": "23:59"
},
"dimensions": [],
"metrics": ["avg_pred"]}
```

As you can see it doesn't select a dimension to be shown, since its not requested but it uses the dimensions *variable* and *pred* for the creation of the filters. This structured status is then integrated in statwolf's platform creating a filter, as can be seen in figure Figure 3.1. The actual query response can be seen in figure Figure 3.2.

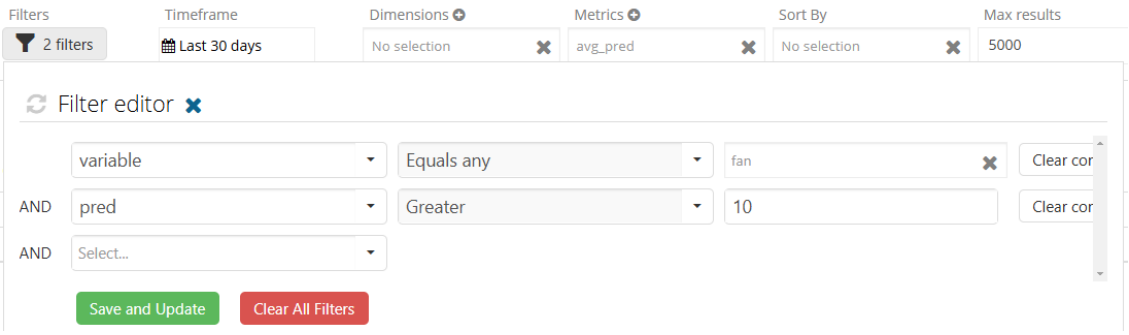

Figure 3.1: Creation of a filter for the first example query in Statwolf's interface

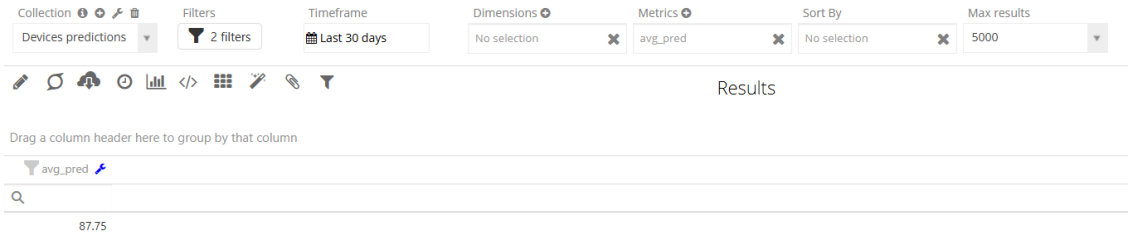

Figure 3.2: Final result of the first example query in Statwolf's Interface

Example 2: Suppose, instead of just wanting one specific value, we want to see certain metrics applied to all the devices in a certain time period.

We could input a query like:

Show me the average predictions, outlier rate and average anomaly score from January through August of 2024, for each of the devices where the 'compressor state' is active

The query generator outputs the following structured statwolf Status:

```
{  "filter": {
    "field_0": "compressor_state",
    "selector_0": "AND",
    "operator_0": "in",
    "value_0": ["On"]
  },
  "timeframe": {
    "period": "months",
    "granularity": "month",
    "dateFrom": "2024-01-01",
    "dateTo": "2024-08-01",
```

```
    "timeFrom": "00:00",
    "timeTo": "23:59"
  },
  "dimensions": ["device_name"],
  "metrics": [
    "avg_pred",
    "avg_is_outlier",
    "avg_anomaly_score"]}
```

As you can see, this time multiple metrics were selected and also *device_name* was selected as a dimension. The filter value verification step makes sure to select the best corresponding real value 'On'(instead of 'active') among the possible unique values of the the dimension *compressor_state*.

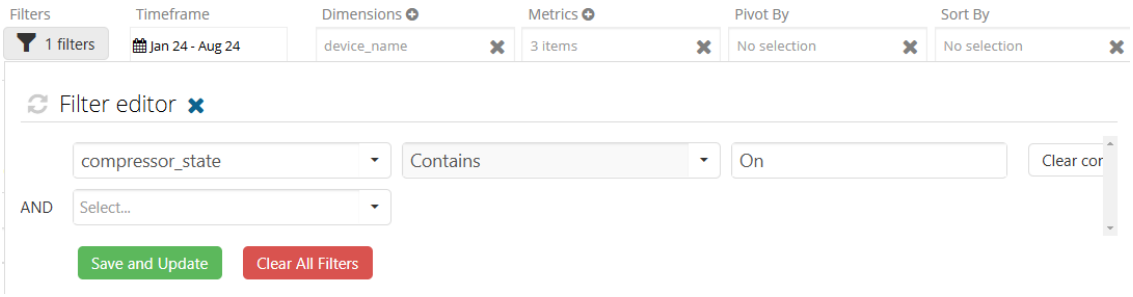

Figure 3.3: Creation of a filter for the second example query in Statwolf's interface

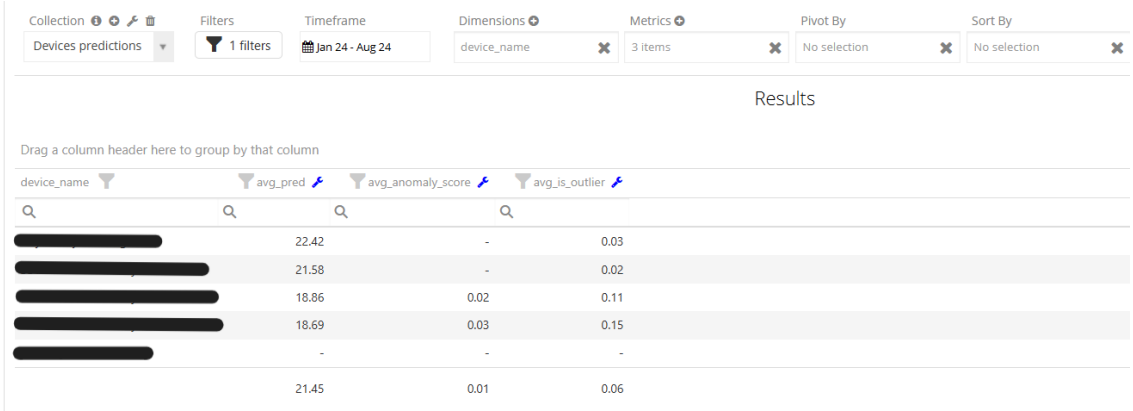

Figure 3.4: Final result of the second example query in Statwolf's Interface

4

Database Schema Customization

In this chapter, we explore the process of database schema customization, a critical aspect of enabling dynamic interactions with a database through natural language inputs. Schema customization focuses on generating metrics and dimensions from user-provided natural language descriptions, bridging the gap between unstructured queries and the structured requirements of a database.

Metrics represent aggregated calculations, summarizing data into key performance indicators (e.g., averages, counts, or totals) that provide quantitative insights. In contrast, dimensions categorize and contextualize data, serving as descriptive attributes (e.g., time periods, locations, or categories) that allow for deeper analysis and segmentation. Together, these components form the foundation for extracting, structuring, and presenting meaningful insights tailored to the needs of end users.

While the previous chapter addressed query generation in the context of a structured dictionary format called 'status', integrated into Statwolf's platform, schema customization extends these capabilities by focusing on creating entirely new features—whether dimensions or metrics—from natural language descriptions. This involves generating SQL queries tailored to the ClickHouse database, requiring a different design approach compared to the query generator.

Key distinctions between schema customization and the query generator lie in the agent's functionality and objectives. While the query generator translates natural language into operations for existing database schemas, the schema customization agent is tasked with interpreting

user intentions to create new schema elements. This requires advanced logic to parse user inputs, map them to valid SQL constructs, and validate their feasibility within the database.

This chapter outlines the methods and tools employed to achieve schema customization, including the design and implementation of specialized prompts and agents.

4.1 ClickHouse Database Systems

ClickHouse is a columnar database management system designed for high-performance analytical queries, making it particularly well-suited for tasks involving dynamic schema customization. Unlike traditional row-oriented SQL databases, ClickHouse's columnar storage format enables efficient query execution by allowing operations to focus on relevant columns rather than scanning entire rows of data. This approach significantly improves performance for analytical workloads, especially when dealing with large-scale datasets [8].

One key difference between ClickHouse and conventional SQL databases lies in its architecture. While traditional SQL systems are often optimized for transactional workloads, ClickHouse is purpose-built for analytical processing. Its capabilities include advanced aggregation functions, array manipulations, and support for nested data structures, all of which enable highly flexible and efficient schema definitions. Additionally, ClickHouse's distributed nature allows it to scale horizontally, handling massive amounts of data and query throughput without compromising speed or reliability.

In the context of schema customization, ClickHouse provides the tools necessary to define complex metrics and dimensions dynamically. This flexibility is crucial for adapting schemas to meet evolving analytical requirements, whether through manual adjustments or automated processes, such as those driven by a language model agent. By leveraging these unique characteristics, ClickHouse serves as a foundation for creating a dynamic, user-driven approach to database interaction.

The next section explores how this adaptability is harnessed to generate metric and dimension definitions, illustrating the process of translating user queries into ClickHouse SQL using a language model agent.

4.2 Feature Generation Overview

The core functionality for schema customization was encapsulated in a *generate_feature* method of a general Statwolf Agent class. For the full code refer to the Appendix.

4.2.1 Feature Identification

First of all, The *generate_feature* method operates as a central point for identifying whether the input query pertains to a metric or a dimension, subsequently routing the query to the appropriate generation process. When the feature type is unspecified, the method first invokes an LLM-based classification model to determine whether the query refers to a metric or dimension. Once identified, the method proceeds with the generation process corresponding to the identified feature type.

The following is an initial section of the prompt that was used for feature identification:

```
You are an advanced interpretation model whose role is to analyze the input
    query and classify it as either a 'dimension' or a 'metric.' Assign it to
     the most likely of the two options, based on the definitions provided.
Definitions:
1. **Dimensions**:
   - Dimensions are individual data values used to categorize, identify, or
       provide context. They are **non-aggregated** and can involve
       transformations, categorizations, or validations that operate at the
       level of each individual data instance.
   - Dimensions do not summarize or consolidate multiple data points but
       rather describe characteristics or attributes of the data.
   Example: `pred` (predictions) is a dimension because it holds raw,
       individual values for each data point.

2. **Metrics**:
   - Metrics are **aggregated calculations** derived from one or more
       dimensions. They summarize, quantify, or provide a higher-level view
       of the data using operations such as averaging, summing, counting, etc
       .
   - Metrics consolidate data across multiple instances, providing insights
       about the dataset as a whole or subsets of it.
   Example: `avg_pred` (average predictions) is a metric because it
       calculates the mean value of all individual predictions.

**Key Rules**:
- Any description involving **aggregation operations** (e.g., sum, count,
    average) or consolidations is a metric.
```

```
- Descriptions involving **extraction, correction, validation, or
    classification of single instances** are dimensions, even if they involve
     some computation.
```

4.2.2 Generating Metrics and Dimensions

The method begins by computing a unique hash for logging purposes, which allows for accurate tracking of operations. It utilizes a suite of tools, including *describe_source_dimensions*, *check_dimensions*, *sql_correction*, and *values_extractor*, to ensure the accuracy and relevance of the generated features. Depending on the identified or provided feature type, the method invokes the corresponding agent to generate Clickhouse SQL definitions, logs each step, and ultimately names the feature using an additional LLM-powered chain.

In addition,`generate_metric` and `generate_dimension` methods were also implemented to create SQL definitions for metrics and dimensions, respectively. Each of these methods uses a specialized LLM prompt to interpret the natural language input, extract relevant dimensions, and produce accurate ClickHouse SQL definitions through an agent.

The process for generating a metric through a langchain agent involves the following steps:

Retrieve Dataset Dimensions

The process begins with retrieving the dimensions from complete schema of the dataset since only dimensions are required in defining new metric or dimension SQL definitions. The *describe_source_dimensions* tool was created for this purpose. For the full code, refer to the Appendix.

Here is the corresponding section from the prompt template that was used:

```
Step 1: Get the complete list of dimensions of the dataset
Action: Use the describe_source_dimensions tool to retrieve the schema.
Action Input: 'source_name':{source_name},'host':{host}, 'log_directory_path
    ': {log_directory_path}, username: {username}, password: {password} .
Observation: The output of the describe_source_dimensions tool is a list of
    dictionaries where each dictionary represents a data field. Each
    dictionary has three keys: 'name' (field name), 'data_type' (the type of
    data), and 'type' (will always be 'dimension' since we are extracting
    dimensions).
```

Analyze the User's Query

The next step involves analyzing the user's query to understand its core objective. This includes identifying the dimensions required, as well as any implicit or explicit filtering or aggregation criteria. A thorough understanding of the query ensures that the generated SQL definition aligns with the user's intent.

Here is the corresponding section in the prompt template:

```
Step 2: Understand the user's request
Action: Analyze the user's query carefully and determine the core objective.
Thought: I will now assess which dimensions are necessary to generate the
    ClickHouse metric definition based on the user's query.
```

Select Relevant Dimensions

Based on the user's query, relevant dimensions are selected for the metric/dimension definition. These dimensions include those directly involved in the computation and categorical dimensions necessary for applying conditional filters.

Here is the corresponding section in the prompt template:

```
Step 3: Select dimensions
Thought: Consider also dimensions that may be important for conditional
    statements in the final ClickHouse SQL metric definition. For example for
     the following input:
'Calcola la proporzione media di outlier su tutti i timestamp.'
The expected output is: 'uniqIf(timestamp, is_outlier != 'Normal' and
    prediction_done = 'yes')/uniq(timestamp)',
Where in this case it filters over values of categorical dimensions '
    is_outlier' and 'prediction_done'.
Action: Select all the relevant dimensions from the dataset including
    categorical dimensions that maybe important for filtering in the final
    definition.
```

Validate Dimensions

The selected dimensions are then validated against the dataset schema to ensure their availability. For this a *check_dimensions* tool was created, which confirms whether the dimensions are present and acceptable in the data. If the validation fails, the dimensions are re-evaluated based on the original query.

Here is the corresponding section in the prompt template:

```
Step 4: Check dimensions
Thought:Ensure that all selected dimensions are present in the schema.
Action:To be sure that the dimensions are acceptable you must use the
    check_dimensions tool which require as input the json.dumps of a
    dictionary with:
- 'list_of_dimensions':the list of dimension you selected
- 'source_name':{source_name}
- 'host': {host}
- 'log_directory_path': {log_directory_path}

If the check_dimensions tool returns True it means that you have selected
    dimensions that are acceptable in the data and you may proceed with step
    5. Only if False go back to step 2 and reflect again on the appropriate
    list of dimensions to select based on the original query.
```

Extract Categorical Dimension Values

For categorical dimensions, it is essential to extract their possible values to incorporate accurate filtering conditions in the SQL definition. The *values_extractor* tool was created for this purpose, taking the selected dimensions as input along with contextual information such as the source name and host. The tool outputs a mapping of categorical dimensions to their distinct values.

Here is the corresponding section in the prompt template:

```
step 5: Dimension Value Extraction
Thought: It's important to know the values of categorical dimensions when
    creating a ClickHouse SQL definition
```

```
Action: Extract the possibile values of relevant dimensions using the the
    values_extractor tool which requires as input:
- 'list_of_dimensions':the list of relevant selected dimensions in the
    previous step
- 'source_name':{source_name}
- 'host': {host}
- 'log_directory_path': {log_directory_path}

This tool will return a dictionary with lists of possible distinct values
    associated to dimensions that satisfy a certain criteria in the tool.
Note:Don't assume you know the values, make sure you use the values_extractor
     tool to extract all the possibile values. Don't skip this step even if
    no categorical dimensions are selected.
```

Generate ClickHouse SQL Definition

With the validated dimensions and extracted categorical values, a ClickHouse SQL command is constructed. This involves translating the natural language query into SQL operations and incorporating filters based on the extracted categorical values when necessary. The resulting SQL definition reflects the user's intent and the structure of the data.

Here is the corresponding section in the prompt template:

```
Step 6: ClickHouse DEFINITION
Action: Using the dimensions selected after step 4 and the dictionary of
    values associated to each of the relevant categorical dimensions
    extracted in step 5 as context, generate a ClickHouse command based on
    the user's natural language query. Don't forget to consider categorical
    dimension values when generating the definition.
```

Ensure SQL Formatting

To ensure correctness, the generated SQL definition is validated using the *sql_correction* tool. This step guarantees that the definition adheres to ClickHouse syntax and is free of errors.

Here is the corresponding section in the prompt template:

```
STEP 7: Output the final ClickHouse definition.
Notes:
Use the sql_correction tool before returning the final output to ensure
    correct ClickHouse formatting.
Please return ONLY the transformed string without any additional comment or
    explanation.
```

Output the Final Definition

After giving some examples to increase the models understanding by the effect of few shot prompting, the final SQL definition is presented as a response to the user's query.

A similar workflow is applied for dimension generation, with adjustments made to align with its descriptive nature. The agent incorporates user queries, schema details, and extracted dimension values to produce precise and contextually appropriate ClickHouse definitions.

4.2.3 Naming Features

The final step in the schema customization process is naming the generated feature. A *name_generator* function was created to employ a structured LLM-based chain to assign meaningful names to the features based on their SQL definition. It ensures that the name is not only descriptive but also adheres to the conventions required for seamless integration into the broader schema. The generated name, alongside the feature's SQL definition and type, is logged for traceability and future reference.

Here is a section of the name generator template that is used:

```
I need you to generate a name for a {feature} based on its SQL definition.
    The output should follow this JSON structure:
{{
    "name": "generated_name",
    "definition": "provided_query",
    "type": "metric_or_dimension"
}}

Guidelines:
```

```
- The `name` should be descriptive and succinct, using lowercase words
    separated by underscores.
- Incorporate the key components of the SQL definition (e.g., functions,
    conditions, columns) into the `name`.
- Avoid overly long or ambiguous names.
- The `definition` should directly use the input SQL query without changes.
- The `type` should directly use the input type
```

4.3 Performance Results

4.3.1 Testing method

To evaluate the performance of the metric and dimension generation, we conducted tests on 12 natural language metric definitions and 15 natural language dimension definitions, for a total of 27 definitions. The metric and dimension definitions used for testing can be found in the appendix. The outputs generated by the model were confronted with the expected SQL outputs to determine their equivalence. Additionally, since there multiple ways one can write equivalent SQL queries, an LLM was employed to assess whether the generated outputs were semantically equivalent to the expected ones. Despite some outputs being functionally acceptable, the model occasionally failed to recognize equivalence due to syntactic differences. Such cases were marked as incorrect, although they would be valid in practical scenarios.

4.3.2 Feature Identification Results

On a dataset of 32 queries, containing 16 dimensions and 16 metrics, the feature identifier had an accuracy of 93.67% on dimensions and 100% on metrics, leaving a total accuracy of 96.87%.

4.3.3 Metric generation Results

With the strictest evaluation, the metric generator performed the test with an accuracy of 58.33%. This considers only cases where the output was perfectly compatible with the expected output. Since, in generating metric definitions only dimensions need to be extracted from the schema, adding an initial query parsing step was unnecessary. Instead the final metric prompt template used was determined by the feature identification step. All the lessons learned in creating the

Query generator, discussed in chapter chapter 3, made the transition to this generator smooth with immediate positive results after some fine-tuning of the prompt template.

Limit cases

These are cases where they are practically acceptable but not completely equivalent. In our test, out of 12 metric definitions tested, there were 2 that were branded as incorrect but are acceptable in practice. These didn't include additional conditions in SQL logic (e.g.,*prediction_done = 'yes'*) in the final ClickHouse Sql query generated.

For example:

For the query written in Italian:

"Calcola la media del limite inferiore previsto e se questa media è negativa, moltiplicala per 1.15; se è positiva, moltiplicala per 0.85.In caso contrario restituisci 0"

In English, this translates to:

"Calculate the average of the predicted lower bound and if this average is negative, multiply it by 1.15; if it is positive, multiply it by 0.85. Otherwise return 0"

The correct output should have been:

```
case
 when avgIf(pred_lower,prediction_done='yes')<0 then avgIf(pred_lower,
     prediction_done='yes')*1.15
 when avgIf(pred_lower,prediction_done='yes')>0 then avgIf(pred_lower,
     prediction_done='yes')*0.85
 else 0
 end
```

However, the generated output doesn't filter by the *prediction_done* condition:

```
avg(pred_lower) * (case when avg(pred_lower) < 0 then 1.15 when avg(
    pred_lower) > 0 then 0.85 else 0 end)
```

We view this as acceptable since there was no mention of *prediction_done* in the input definition and in cases when *prediction_done = 'no'* the *pred* variables are set to NULL and would be automatically ignored in any aggregation function.

Accuracy for practical use

In practical scenarios, Considering these functionally equivalent limit cases as correct, the accuracy would increase to 75%. Overall the generation of ClickHouse SQL metric definitions posed slightly more challenges in comparison to the structured query generator discussed in chapter chapter 3, due to the unpredictable and free nature of LLM's in generator in written SQL commands compared to generating a query with a predefined JSON structure. In the following section contains examples of some failure cases, providing a good overview of the agent's weaknesses.

Failure cases

Out of the 12 queries tested, 3 critically failed to generate the correct metrics or SQL queries, resulting in an overall accuracy of 75%. Below, we detail these failure cases and their implications.

The first failure case involved the query written in Italian:

"Conta il numero di avvii del ciclo di sbrinamento."

In English, this translates to:

"Count the number of defrost cycle starts."

The expected output was a metric representing the sum of defrost cycle starts, with the corresponding ClickHouse SQL:

```
sum(starting_defrost)
```

However, the generated output instead provided:

```
count(defrost_cycle)
```

This error highlights the generator's inability to differentiate between ambiguously named dimensions. Such mistakes can arise from ambiguities in natural language prompts and may be mitigated by fine-tuning the generator or enriching the schema with additional semantic information.

—

The second failure case involved this query:

"Calcola la proporzione media di outlier giornalieri contrassegnati come guasti."

In English:

"Calculate the average proportion of daily outliers marked as faults."

The expected output was a metric for the average proportion of daily outliers flagged as faults, with the SQL query:

```
avgIf(1, is_daily_outlier = 'Fault')
```

However, the generated output was:

```
uniqIf(day, is_outlier != 'Normal' AND prediction_done = 'yes') / uniq(day)
    AS outlier_proportion
```

Here, the generator selected a metric and SQL structure more suited to calculating unique proportions of outliers, rather than the required average proportion for fault-specific cases. This again underscores the model's challenges in distinguishing similar metrics with overlapping semantic meanings.

—

The third failure case involved this query:

"Calcola la media dei residui proporzionali, escludendo i valori normali e N/A."

In English:

"Calculate the mean of proportional residuals, excluding normal and N/A values."

The expected output should have included a metric for average proportional residuals, with the SQL:

```
abs(avgIf(true_val - pred, is_outlier NOT IN ('Normal', 'N/A'))) / abs((max(
    true_val - pred) - min(true_val - pred)))
```

However, the generated output was:

```
avg(residuals_interval) WHERE is_outlier != 'Normal' AND prediction_done = '
    yes'
```

This failure occurred because the generator confused the user-specified "proportional residuals" with the existing dimension ‘residuals_interval‘ extracted from the schema. The ‘residuals_interval‘ is defined as follows:

```
CASE
    WHEN true_val > pred_upper_boundries*1.15 AND prediction_done = 'yes' AND
         pred_upper > 0 THEN ABS(ABS(true_val) - ABS(pred_upper*1.15))
    WHEN true_val > pred_upper_boundries*0.85 AND prediction_done = 'yes' AND
         pred_lower < 0 THEN ABS(ABS(true_val) - ABS(pred_upper*0.85))
```

```
    WHEN true_val < pred_lower*0.85 AND prediction_done = 'yes' AND
        pred_lower > 0 THEN ABS(ABS(true_val) - ABS(pred_lower*0.85))
    WHEN true_val < pred_lower*1.15 AND prediction_done = 'yes' AND
        pred_lower < 0 THEN ABS(ABS(true_val) - ABS(pred_lower*1.15))
    ELSE 0
END
```

While this dimension is valid for other contexts, it does not align with the requested proportional residuals metric. The mention of ”residuals” in the query led the generator to assume ‘residuals_interval‘ was appropriate, despite its distinct calculation logic. This issue stems from the lack of metadata descriptions in the schema, leaving the generator unable to assess the actual properties or intended use of dimensions and metrics when constructing new metrics. The problem is exacerbated by overlapping or vague naming conventions, which the generator defaults to when faced with ambiguity.

—

In conclusion, these failure cases reveal areas where the ClickHouse SQL metric generator struggles, particularly in interpreting nuanced metric definitions and generating the correct SQL expressions. Addressing these challenges could involve schema enrichment, refining the input query structure, and incorporating domain-specific training examples. Explicit metadata descriptions for dimensions and metrics would significantly reduce such errors by enabling the model to make informed decisions about appropriate metrics and their usage. Despite these limitations, the model demonstrates strong overall accuracy and reliability in handling a majority of test cases.

4.3.4 Dimensions generation results

With the strictest evaluation, the dimension generator performed the test with an accuracy of 60%. This considers cases where the output was perfectly compatible with the expected output. For dimension generation, there were no limit cases but a new type of failure case was observed.

Failure Cases

Out of the 15 queries tested, 6 failed to generate the correct metrics or SQL queries, resulting in an overall accuracy of 60%. Below, we detail these failure cases and their implications.

—

The first failure case involved the query written in Italian:

Calcola il punteggio di anomalia in base al metodo di isolamento e alla soglia dove restituisce 1 se un valore negativo, moltiplicato per -3, supera 1; il prodotto se resta sotto 1; altrimenti 0.

In English, this translates to:

Calculate the anomaly score based on the isolation method and the threshold where it returns 1 if a negative value, multiplied by -3, exceeds 1; the product if it remains below 1; otherwise 0.

The expected output was a dimension definition of:

```
metric: anomaly_score,
ClickHouse SQL: case
    when anomaly_score_isolation < 0 and -1*anomaly_score_isolation * 3 > 1
        then 1
    when anomaly_score_isolation < 0 and -1*anomaly_score_isolation * 3 < 1
        then -1*anomaly_score_isolation*3
    else 0
end
```

However, the generated output instead provided:

```
metric: true_val_adjustment,
ClickHouse SQL: CASE
    WHEN true_val < 0 AND (true_val * -3) > 1 THEN 1
    WHEN true_val < 0 AND (true_val * -3) <= 1 THEN (true_val * -3)
    ELSE 0
END
```

The discrepancy in the output demonstrates a misunderstanding of the variable context. While the expected output uses *anomaly_score_isolation* as the core variable, the generated output substitutes this with *true_val*, which lacks specific meaning in the given query. This may be solved by being more specific in the input definition. This discrepancy points to the model's struggle with understanding contextual relationships and accurately mapping them to the correct metric and corresponding SQL.

—

The second failure case involved the query: *Correggi lo stato del compressore in base al cambiamento del suo stato.*

Translated to English: *Correct the compressor state based on the change in its state.*

The expected output was:

```
metric: compressor_state_corrected,
```

```
ClickHouse SQL: case
    when compressor_state = 'On' and compressor_status_changed = 'true' then
        'TurnOff'
    when compressor_state = 'Off' and compressor_status_changed = 'true' then
         'TurnOn'
    else compressor_state
end
```

Instead, the generated output was:

```
metric: compressor_state_change,
ClickHouse SQL: CASE
    WHEN compressor_state = 'On' AND compressor_state_changed = true THEN '
        TurnOff'
    WHEN compressor_state = 'Off' AND compressor_state_changed = true THEN '
        TurnOn'
    ELSE compressor_state
END
```

In this case, it gets confused between *compressor_state_changed* and *compressor_status_changed*. An understandable result considering the lack of descriptive metadata assigned to the schema.

—

The third failure involved the query: *Determina se il valore di delta_tw è fuori dai limiti in base allo stato operativo: restituisce 'Out' se è maggiore di 5 con stato 'Heat Pump' o minore di 5 con stato 'Chiller'. Altrimenti, assegna 'Normal'.*

Translated to English: *Determine if the delta_tw value is out of bounds based on the operating state: return 'Out' if it is greater than 5 with state 'Heat Pump' or less than 5 with state 'Chiller'. Otherwise, assign 'Normal'.*

The expected output was:

```
metric: delta_tw_out_of_bounds,
ClickHouse SQL: case
    when variable = 'delta_tw' and true_val > 5 and status= 'Heat Pump' then
        'Out'
    when variable = 'delta_tw' and true_val < 5 and status= 'Chiller' then '
        Out'
    else 'Normal'
end
```

Instead, the generated output was:

```
metric: outlier_state_condition,
ClickHouse SQL: CASE
    WHEN true_val > 5 AND state = 'Heat Pump' THEN 'Out'
    WHEN true_val < 5 AND state = 'Chiller' THEN 'Out'
    ELSE 'Normal'
END
```

In this failure, the logic is mostly correct, but the model didn't match the specific input requirements, such as using *delta_tw* for the variable and confusing *status* with *state*.

—

The fourth failure involved the query: *Restituisci il nome del mese con il codice formattato in base al timestamp.*

Translated to English: *Return the month name with the code formatted according to the timestamp.*

The expected output was:

```
metric: inline_month,
ClickHouse SQL: CASE
 WHEN toMonth(timestamp) = 1 THEN '(01) January'
 WHEN toMonth(timestamp) = 2 THEN '(02) February'
 WHEN toMonth(timestamp) = 3 THEN '(03) March'
 WHEN toMonth(timestamp) = 4 THEN '(04) April'
 WHEN toMonth(timestamp) = 5 THEN '(05) May'
 WHEN toMonth(timestamp) = 6 THEN '(06) June'
 WHEN toMonth(timestamp) = 7 THEN '(07) July'
 WHEN toMonth(timestamp) = 8 THEN '(08) August'
 WHEN toMonth(timestamp) = 9 THEN '(09) September'
 WHEN toMonth(timestamp) = 10 THEN '(10) October'
 WHEN toMonth(timestamp) = 11 THEN '(11) November'
 WHEN toMonth(timestamp) = 12 THEN '(12) December'
 ELSE 'N/A'
END
```

Instead, the generated output was:

```
metric: month_of_timestamp,
ClickHouse SQL: month(timestamp)
```

The output uses a standard SQL function, *month()*, which is not supported in ClickHouse, indicating inadequate training of the model on ClickHouse-specific syntax.

—

The fifth failure involved the query: *Determina se l'anomalia è stata corretta in base al suo stato, assegnando i valori 'Fault' o 'Normal'.*

Translated to English: *Determine whether the anomaly has been corrected based on its status by assigning the values 'Fault' or 'Normal'.*

The expected output was:

```
metric: is_anomaly_corrected,
ClickHouse SQL: case
 when is_anomaly = 1 then 'Fault'
 else 'Normal'
 end
```

Instead, the generated output was:

```
metric: compressor_state_normal_fault,
ClickHouse SQL: case when compressor_state = 'On' and state = 'Normal' then '
    Normal' else 'Fault' end
```

This is completely wrong. It seems that the word "state" in the input definition confuses the agent.

—

The last failure case involved the query: *Quando la variabile è 'eev1_perc', limita il valore del confine superiore previsto a 100 se supera tale valore.*

Translated to English: *When the variable is 'eev1_perc', limit the predicted upper bound value to 100 if it exceeds that value.*

The expected output was:

```
metric: pred_upper_boundaries,
ClickHouse SQL: case
 when variable == 'eev1_perc' and pred_upper > 100 then 100
 else pred_upper
 end
```

Instead, the generated output was:

```
metric: adjusted_true_value,
ClickHouse SQL: CASE WHEN variable = 'eev1_perc' AND true_val > 100 THEN 100
    ELSE true_val END
```

One important factor to consider is that the wording of the input significantly affects the final result. In this case, most online translation services would translate the input as: *When the variable is 'eev1_perc', limit the* ***expected*** *upper bound value to 100 if it exceeds that value.*

This translation uses *expected upper bound* instead of *predicted upper bound*. The issue arises because the Italian word *previsto* can mean both *predicted* and *expected*, depending on the context. This ambiguity may cause the model to misinterpret the input and generate an incorrect output.

—

For testing purposes minimum schema knowledge was used in creating the feature definitions. In practical scenarios, many ambiguities can be resolved by specifying the exact name in quotation marks of the necessary dimensions or metrics.

5 Conclusion

The objective of this project was to develop a chatbot capable of enabling natural language interaction with Statwolf's BI platform. This was achieved through the implementation of LLM agents using the LangChain library. The chatbot was designed with two fundamental capabilities: query generation and database schema customization.

The query generation component, utilizing a structured pipeline of LLMs and agents, produces a JSON-formatted 'Status' that integrates seamlessly with Statwolf's existing software to generate ClickHouse SQL queries. This approach demonstrated strong performance during testing, proving to be robust against spelling errors and incomplete queries due to the presence of custom agent tools designed for feature and value selection verification.

For database schema customization, a similar process was followed, but with a key distinction: rather than generating structured outputs, the system directly produces ClickHouse SQL commands. Due to the increased complexity of this task, its accuracy was slightly lower than that of the query generator. The most frequent failure cases in both components involved misinterpretation or confusion between similarly named features (dimensions or metrics) extracted from the schema.

Future improvements could focus on reducing feature misinterpretation by incorporating descriptive metadata into the schema, which could help the model differentiate between features more effectively. This refinement could lead to further improvements in accuracy and reliability, making the chatbot a more effective tool for interacting with large-scale analytical databases.

Ultimately, this project demonstrates the potential of LLM-driven solutions in enhancing database interaction, bridging the gap between natural language understanding and complex query generation.

References

[1] LangGraph. (2024) Langgraph- conceptual guide. [Online]. Available: https://langchain-ai.github.io/langgraph/concepts/

[2] Langchain. (2024) Langchain- conceptual guide. [Online]. Available: https://python.langchain.com/docs/concepts/

[3] LANGCHAIN. (2024) Announcing langgraph v0.1 langgraph cloud. [Online]. Available: https://blog.langchain.dev/langgraph-cloud/

[4] V. Zhong, C. Xiong, and R. Socher, "Seq2sql: Generating structured queries from natural language using reinforcement learning," 2017. [Online]. Available: https://arxiv.org/abs/1709.00103

[5] J. Guo, Z. Zhan, Y. Gao, Y. Xiao, J.-G. Lou, T. Liu, and D. Zhang, "Towards complex text-to-sql in cross-domain database with intermediate representation," in *Proceeding of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 2019.

[6] B. Li, Y. Luo, C. Chai, G. Li, and N. Tang, "The dawn of natural language to sql: Are we fully ready?" 2024. [Online]. Available: https://arxiv.org/abs/2406.01265

[7] OPENAI. (2024) Gpt-4o mini: advancing cost-efficient intelligence. [Online]. Available: https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/

[8] ClickHouse. What is clickhouse? [Online]. Available: https://clickhouse.com/docs/en/intro

Acknowledgments

I want to thank Dr. David Dandolo for guiding me throughout my internship at Statwolf.

A Code

A.1 Main Agent Class

Listing A.1: Statwolf Agent

```
import json
import hashlib
import pandas as pd
from agent_tools import (
    describe_source,
    describe_source_dimensions,
    check_dimensions_and_metrics,
    check_dimensions,
    json_correction,
    sql_correction,
    values_extractor,
)
from agent_functions import (
    query_simplifier,
    selector_agent,
    filter_corrector,
    timeframe_corrector,
```

```
    validate_filter_values,
    metric_generator_agent,
    dimension_generator_agent,
    name_generator,
    feature_generator_agent,
    feature_identification,
)
from prompt_templates import (
    Instructions,
    query_simplifier_prompt_template,
    filter_prompt_template,
    timeframe_prompt,
    value_template,
    metric_generator_template,
    dimension_generator_template,
    name_generator_template,
    feature_identification_template,
)
from logger import Agent_Logger



class STATWOLF_LLM_AGENT:
    def __init__(self, username, password, log_directory_path):
        """
        Initialize the STATWOLF_LLM_AGENT.

        Args:
            username (str): The username for authentication.
            password (str): The password for authentication.
            log_directory_path (str): The directory path for logging.
        """
        self.username = username
        self.password = password
        self.log_directory_path = log_directory_path
```

```
        self.logger = Agent_Logger(log_path=f"{log_directory_path}/
            activity_log.csv")

    def generate_query(self, query, source_name, host):
        """
        Process the query on the specified data source.

        Args:
            query (str): The query text to process.
            source_name (str): The name of the data source.
            host (str): The data source host.

        Returns:
            dict: The final processed output.
        """
        data_to_hash = f"{query}{source_name}{host}"
        hash_code = hashlib.sha256(data_to_hash.encode("utf-8")).hexdigest()
        self.logger.call_hash = hash_code

        tools = [describe_source, check_dimensions_and_metrics,
            json_correction]

        self.logger.log_activity(
            activity="QUERY GENERATION START",
            input=query,
            output=None,
            llm_prompt=None,
            source_name=source_name,
            host=host,
        )
        try:
        # Analyze query
            query_desc = query_simplifier().invoke({"query": query})
            self.logger.log_activity(
                activity=self.logger.QUERY_SIMPLIFIER,
                input=query,
```

```
                output=query_desc,
                llm_prompt=query_simplifier_prompt_template,
                source_name=source_name,
                host=host,
            )

            # Selector Agent
            agent_dict = selector_agent(query_desc, source_name, host, tools,
                self.log_directory_path, self.username, self.password)
            self.logger.log_activity(
                activity=self.logger.SELECT_DIMENSIONS_AND_METRICS,
                input=query,
                output=agent_dict,
                llm_prompt=Instructions.format(source_name=source_name, host=
                    host, log_directory_path=self.log_directory_path,
                    username="REDACTED", password="REDACTED"),
                source_name=source_name,
                host=host,
            )

            # Filter Correction
            agent_dict_fc = filter_corrector(agent_dict, query)
            self.logger.log_activity(
                activity=self.logger.FILTER_CORRECTOR,
                input=agent_dict,
                output=agent_dict_fc,
                llm_prompt=filter_prompt_template,
                source_name=source_name,
                host=host,
            )

            # Timeframe Correction
            agent_dict_tc = timeframe_corrector(agent_dict_fc)
            self.logger.log_activity(
                activity=self.logger.TIMEFRAME_CORRECTOR,
                input=agent_dict_fc,
```

```
                output=agent_dict_tc,
                llm_prompt=timeframe_prompt,
                source_name=source_name,
                host=host,
            )

            # Value Validation
            final_agent_dict = validate_filter_values(agent_dict_tc,
                source_name, host,self.log_directory_path)
            self.logger.log_activity(
                activity=self.logger.VALUE_VALIDATION,
                input=agent_dict_tc,
                output=final_agent_dict,
                llm_prompt=value_template,
                source_name=source_name,
                host=host,
            )

            self.logger.log_activity(
                activity="QUERY GENERATION END",
                input=None,
                output=final_agent_dict,
                llm_prompt=None,
                source_name=source_name,
                host=host,
            )

            self.logger.return_log(save=True)

            # Save output to file
            #with open("output.json", "w") as f:
                #json.dump(final_agent_dict, f)

            return final_agent_dict

        except Exception as e:
```

```
        # Log the error details internally
            self.logger.log_activity(
                activity="ERROR",
                input=query,
                output=f"Error: {str(e)}",
                source_name=source_name,
                host=host,
            )
            self.logger.return_log(save=True)

            return {"StatwolfError": "We were unable to process your request.
                 Please try again with a different query or contact support
                if the issue persists."}
            #return e




    def get_logs(self):
        """
        Retrieve the activity logs as a pandas DataFrame.

        Returns:
            pd.DataFrame: DataFrame containing the logged activities.
        """
        return pd.read_csv(f"{self.log_directory_path}/activity_log.csv")

    #def generate_dimension(self, query, source_name, host):




    def generate_metric(self, query, source_name, host):


        data_to_hash = f"{query}{source_name}{host}"
```

```
        hash_code = hashlib.sha256(data_to_hash.encode("utf-8")).hexdigest()
        self.logger.call_hash = hash_code

        tools = [describe_source_dimensions, check_dimensions, sql_correction
            , values_extractor]

        self.logger.log_activity(
            activity="METRIC GENERATION START",
            input=query,
            output=None,
            llm_prompt=None,
            source_name=source_name,
            host=host,
        )
        try:
        # Analyze query

            # Selector Agent
            metric_sql_def = metric_generator_agent(query, source_name, host,
                 tools,self.log_directory_path, self.username, self.password)
            self.logger.log_activity(
                activity=self.logger.METRIC_DEFINITION,
                input=query,
                output=metric_sql_def,
                llm_prompt=metric_generator_template.format(source_name=
                    source_name, host=host, log_directory_path=self.
                    log_directory_path, username="REDACTED", password="
                    REDACTED"),
                source_name=source_name,
                host=host,
            )
            metric_def=name_generator(metric_sql_def,"metric")
            self.logger.log_activity(
                activity=self.logger.NAME_GENERATION,
                input=metric_sql_def,
                output=metric_def,
```

```
                llm_prompt=name_generator_template,
                source_name=source_name,
                host=host,
            )



            self.logger.log_activity(
                activity="METRIC GENERATION END",
                input=None,
                output=metric_def,
                llm_prompt=None,
                source_name=source_name,
                host=host,
            )

            self.logger.return_log(save=True)

            # Save output to file
            #with open("output.json", "w") as f:
                #json.dump(final_agent_dict, f)

            return metric_def

        except Exception as e:
        # Log the error details internally
            self.logger.log_activity(
                activity="ERROR",
                input=query,
                output=f"Error: {str(e)}",
                source_name=source_name,
                host=host,
            )
            self.logger.return_log(save=True)
```

```
            return {"StatwolfError": "We were unable to process your request.
                 Please try again with a different query or contact support
                if the issue persists."}
            #return e

    def generate_dimension(self, query, source_name, host):


        data_to_hash = f"{query}{source_name}{host}"
        hash_code = hashlib.sha256(data_to_hash.encode("utf-8")).hexdigest()
        self.logger.call_hash = hash_code

        tools = [describe_source_dimensions, check_dimensions, sql_correction
            , values_extractor]

        self.logger.log_activity(
            activity="DIMENSION GENERATION START",
            input=query,
            output=None,
            llm_prompt=None,
            source_name=source_name,
            host=host,
        )
        try:
        # Analyze query

            # Selector Agent
            dimension_sql_def = dimension_generator_agent(query, source_name,
                 host, tools,self.log_directory_path, self.username, self.
                password)
            self.logger.log_activity(
                activity=self.logger.DIMENSION_DEFINITION,
                input=query,
                output=dimension_sql_def,
                llm_prompt=dimension_generator_template.format(source_name=
                    source_name, host=host, log_directory_path=self.
```

```
                log_directory_path, username="REDACTED", password="
                REDACTED"),
            source_name=source_name,
            host=host,
        )
        dimension_def=name_generator(dimension_sql_def,"dimension")
        self.logger.log_activity(
            activity=self.logger.NAME_GENERATION,
            input=dimension_sql_def,
            output=dimension_def,
            llm_prompt=name_generator_template,
            source_name=source_name,
            host=host,
        )



        self.logger.log_activity(
            activity="DIMENSION GENERATION END",
            input=None,
            output=dimension_def,
            llm_prompt=None,
            source_name=source_name,
            host=host,
        )

        self.logger.return_log(save=True)

        # Save output to file
        #with open("output.json", "w") as f:
            #json.dump(final_agent_dict, f)

        return dimension_def

    except Exception as e:
    # Log the error details internally
```

```
            self.logger.log_activity(
                activity="ERROR",
                input=query,
                output=f"Error: {str(e)}",
                source_name=source_name,
                host=host,
            )
            self.logger.return_log(save=True)

            return {"StatwolfError": "We were unable to process your request.
                 Please try again with a different query or contact support
                if the issue persists."}


    def generate_feature(self, query, source_name, host, feature='unknown'):
        ''' feature should be metric or dimension'''

        data_to_hash = f"{query}{source_name}{host}"
        hash_code = hashlib.sha256(data_to_hash.encode("utf-8")).hexdigest()
        self.logger.call_hash = hash_code

        tools = [describe_source_dimensions, check_dimensions, sql_correction
            , values_extractor]

        self.logger.log_activity(
            activity=f"{feature.upper()} GENERATION START",
            input=query,
            output=None,
            llm_prompt=None,
            source_name=source_name,
            host=host,
        )
        try:
        # Analyze query

            if feature=='unknown':
```

```
            feature=feature_identification(query)
            print(f"Note: 'feature' type was not provided among the input
                 arguments. After analyzing the query, we set the feature
                 type as '{feature}' .")

        # Selector Agent
        feat_sql_def = feature_generator_agent(query, source_name, host,
            tools,self.log_directory_path, self.username, self.password,
            feature)

        if feature.lower()=='dimension':
            self.logger.log_activity(
                activity=self.logger.DIMENSION_DEFINITION,
                input=query,
                output=feat_sql_def,
                llm_prompt=dimension_generator_template.format(
                    source_name=source_name, host=host,
                    log_directory_path=self.log_directory_path, username=
                    "REDACTED", password="REDACTED"),
                source_name=source_name,
                host=host,
            )

        if feature.lower()=='metric':
            self.logger.log_activity(
                activity=self.logger.METRIC_DEFINITION,
                input=query,
                output=feat_sql_def,
                llm_prompt=metric_generator_template.format(source_name=
                    source_name, host=host, log_directory_path=self.
                    log_directory_path, username="REDACTED", password="
                    REDACTED"),
                source_name=source_name,
                host=host,
            )
```

```
        feat_def=name_generator(feat_sql_def,feature)
        self.logger.log_activity(
            activity=self.logger.NAME_GENERATION,
            input=feat_sql_def,
            output=feat_def,
            llm_prompt=name_generator_template,
            source_name=source_name,
            host=host,
        )



        self.logger.log_activity(
            activity="METRIC GENERATION END",
            input=None,
            output=feat_def,
            llm_prompt=None,
            source_name=source_name,
            host=host,
        )

        self.logger.return_log(save=True)

        # Save output to file
        #with open("output.json", "w") as f:
            #json.dump(final_agent_dict, f)

        return feat_def

    except Exception as e:
    # Log the error details internally
        self.logger.log_activity(
            activity="ERROR",
            input=query,
```

```
                output=f"Error: {str(e)}",
                source_name=source_name,
                host=host,
            )
            self.logger.return_log(save=True)

            return {"StatwolfError": "We were unable to process your request.
                 Please try again with a different query or contact support
                if the issue persists."}
            #return e



#funzione generate_metric_dim( type='not_set') ->if type=='not_set' llm
    determina il type. if type='metric'->metric_generatore_agent,ecc.
```

A.2 Agent Tools

Listing A.2: Agent tools

```
#from langchain.tools import tool
from langchain_core.tools import tool
import os
import json
import statwolf
import pandas as pd
from langchain_openai import ChatOpenAI



@tool
def describe_source(source_name: str, host: str, log_directory_path: str,
    username: str, password: str):
    """
    Call this to retrieve and saves the schema, dimensions, and metrics from
        a specified datasource.
```

```
    """


    host_name= host.replace("https://", "").strip("/").replace("/", "_").
        replace(".com", "").replace("statwolf", "").replace(".","") .replace(
        "__", "_")

    schema_dir = os.path.join(log_directory_path, 'source_schema', host_name,
         source_name)
    # Definisci il percorso del file schema json
    schema_path = os.path.join(schema_dir, f"{source_name}_schema.json")

    # Create the directory if it does not exist
    os.makedirs(schema_dir, exist_ok=True)


    #schema_path = f'./source_schema/{source_name}_schema.json'#source_schema
        ->host->source_name
    # Crea la directory se non esiste
    # os.makedirs('./source_schema', exist_ok=True)

    # Controlla se il file esiste già
    if os.path.exists(schema_path):
        print(f"Schema already exists at {schema_path}. Loading from file.")
        with open(schema_path, 'r') as schema_file:
            # Carica e restituisci il file esistente
            return json.load(schema_file)
    else:
        # Connetti al datasource e scarica lo schema
        #datasource = statwolf.create({'host': host, 'username': username, '
            password': password}, 'datasource')

        try:
            datasource = statwolf.create(
                {'host': host, 'username': username, 'password': password},
                'datasource'
```

```
            )
            print("[Info] Connected to datasource.")
        except Exception as e:
            print(f"[Error] Failed to connect to datasource: {e}")
            return {"error": str(e)}


        print('[Download] Schema...')
        schema = datasource.explore(sourceid=source_name).schema()
        print('[Download] Dimensions...')
        dimensions = datasource.explore(sourceid=source_name).dimensions()
        print('[Download] Metrics...')
        metrics = datasource.explore(sourceid=source_name).metrics()

        print('[Download] Generate complete schema object...')
        dimensions_str = [str(d) for d in dimensions]
        metrics_str = [str(m) for m in metrics]

        #print(f"[Debug] Schema: {schema}")
        #print(f"[Debug] Dimensions: {dimensions_str}")
        #print(f"[Debug] Metrics: {metrics_str}")

        #final_schema = [
        #    {'name': d, 'data_type': schema[d], 'type': 'dimension'} for d
            in dimensions_str
        #] + [
        #    {'name': m, 'data_type': schema[m.split('_timecomparison')[0].
            split('_vs')[0]], 'type': 'metric'}
        #    for m in metrics_str
        #]
        final_schema = [
            {'name': d, 'data_type': schema.get(d, 'NA'), 'type': 'dimension'
                } for d in dimensions_str
        ]
        processed_base_keys = set()
        for m in metrics_str:
```

```
            # Process metric name to get base key
            base_key = m.split('_timecomparison')[0].split('_vs')[0]
            #print(f"[Debug] Processing metric '{m}' with base key '{base_key
                }'")
            if base_key in processed_base_keys:
                continue

            processed_base_keys.add(base_key)
            # Assign 'NA' if base key is not found in schema
            data_type = schema.get(base_key, 'NA')

            # Add metric to final schema
            final_schema.append({
                'name': base_key,
                'data_type': data_type,
                'type': 'metric'
            })



        # Save schema to file
        try:
            with open(schema_path, 'w') as schema_file:
                json.dump(final_schema, schema_file, indent=4)
            print(f"[Save] Schema saved to {schema_path}.")
        except Exception as e:
            print(f"[Error] Failed to save schema: {e}")
            return {"error": str(e)}

        return final_schema



@tool
def describe_source_dimensions(source_name: str, host: str,
    log_directory_path: str, username: str, password: str):
```

```
    """
    Call this to retrieve and save the dimensions from a specified datasource
        .
    """


    host_name= host.replace("https://", "").strip("/").replace("/", "_").
        replace(".com", "").replace("statwolf", "").replace(".","") .replace(
        "__", "_")

    schema_dir = os.path.join(log_directory_path, 'source_schema', host_name,
         source_name)
    # Definisci il percorso del file schema json
    schema_path = os.path.join(schema_dir, f"{source_name}_schema.json")

    # Create the directory if it does not exist
    os.makedirs(schema_dir, exist_ok=True)


    #schema_path = f'./source_schema/{source_name}_schema.json'#source_schema
        ->host->source_name
    # Crea la directory se non esiste
   # os.makedirs('./source_schema', exist_ok=True)

    # Controlla se il file esiste già
    if os.path.exists(schema_path):
        print(f"Schema already exists at {schema_path}. Loading from file.")
        with open(schema_path, 'r') as schema_file:
            # Carica e restituisci il file esistente
            full_schema = json.load(schema_file)
            # Filter for type='dimension'
            dimensions_only = [item for item in full_schema if item.get('type
                ') == 'dimension']
            # Return the filtered schema
            return dimensions_only
    else:
```

```
        # Connetti al datasource e scarica lo schema
        datasource = statwolf.create({'host': host, 'username': username, '
            password': password}, 'datasource')
        print('[Download] Schema...')
        schema = datasource.explore(sourceid=source_name).schema()
        print('[Download] Dimensions...')
        dimensions = datasource.explore(sourceid=source_name).dimensions()
        print('[Download] Metrics...')
        metrics = datasource.explore(sourceid=source_name).metrics()

        print('[Download] Generate complete schema object...')
        dimensions_str = [str(d) for d in dimensions]
        metrics_str = [str(m) for m in metrics]

        final_schema = [
            {'name': d, 'data_type': schema.get(d, 'NA'), 'type': 'dimension'
                } for d in dimensions_str
        ]
        processed_base_keys = set()
        for m in metrics_str:
            # Process metric name to get base key
            base_key = m.split('_timecomparison')[0].split('_vs')[0]
            #print(f"[Debug] Processing metric '{m}' with base key '{base_key
                }'")
            if base_key in processed_base_keys:
                continue

            processed_base_keys.add(base_key)
            # Assign 'NA' if base key is not found in schema
            data_type = schema.get(base_key, 'NA')

            # Add metric to final schema
            final_schema.append({
                'name': base_key,
                'data_type': data_type,
                'type': 'metric'
```

```
            })


        dimensions_only = [item for item in final_schema if item['type'] == '
            dimension']
        # Salva lo schema in un file json
        try:
            with open(schema_path, 'w') as schema_file:
                json.dump(final_schema, schema_file, indent=4)
            print(f"[Save] Schema saved to {schema_path}.")
        except Exception as e:
            print(f"[Error] Failed to save schema: {e}")
            return {"error": str(e)}

        return dimensions_only

@tool
def check_dimensions_and_metrics(list_of_dimensions, list_of_metrics,
    source_name, host, log_directory_path):
    """
    Call this to check the provided dimensions and metrics against the schema
         of the specified datasource.
    """


    # Definisci il percorso del file schema json
    host_name= host.replace("https://", "").strip("/").replace("/", "_").
        replace(".com", "").replace("statwolf", "").replace(".","") .replace(
        "__", "_")

    schema_dir = os.path.join(log_directory_path, 'source_schema', host_name,
         source_name)
    # Definisci il percorso del file schema json
    schema_path = os.path.join(schema_dir, f"{source_name}_schema.json")
    #schema_path = f'./source_schema/{source_name}_schema.json'
```

```
    # Controlla se il file esiste
    if os.path.exists(schema_path):
        print(f"Loading schema from {schema_path}.")
        with open(schema_path, 'r') as schema_file:
            # Carica e restituisci lo schema dal file json
            schema = json.load(schema_file)
    else:
        print(f"Error: Schema file {schema_path} does not exist.")
        return False  # Restituisci None o un altro valore predefinito se il
            file non esiste

    schema_df = pd.DataFrame(schema)
    dimension_df = schema_df.loc[schema_df['name'].isin(list_of_dimensions)]
    metrics_df = schema_df.loc[schema_df['name'].isin(list_of_metrics)]

    length_check = dimension_df.shape[0] + metrics_df.shape[0] == len(
        list_of_dimensions + list_of_metrics)
    #metrics_check = (len(metrics_df['type'].unique()) == 1 and metrics_df['
        type'].unique()[0]=='metric')
    metrics_check = (metrics_df.empty or (len(metrics_df['type'].unique()) ==
         1 and metrics_df['type'].unique()[0] == 'metric'))
    #dimension_check = (len(dimension_df['type'].unique()) == 1 and
        dimension_df['type'].unique()[0]=='dimension')
    dimension_check = (dimension_df.empty or (len(dimension_df['type'].unique
        ()) == 1 and dimension_df['type'].unique()[0] == 'dimension'))

    check_output = metrics_check and dimension_check and length_check

    print(f'[Dimension and Metric Check] {check_output}')

    return check_output



@tool
```

```
def check_dimensions(list_of_dimensions,source_name, host, log_directory_path
    ):
    """
    Call this to check the provided dimensions against the schema of the
        specified datasource.
    """

    # Definisci il percorso del file schema json
    host_name= host.replace("https://", "").strip("/").replace("/", "_").
        replace(".com", "").replace("statwolf", "").replace(".","") .replace(
        "__", "_")

    schema_dir = os.path.join(log_directory_path, 'source_schema', host_name,
         source_name)
    # Definisci il percorso del file schema json
    schema_path = os.path.join(schema_dir, f"{source_name}_schema.json")

    # Controlla se il file esiste
    if os.path.exists(schema_path):
        print(f"Loading schema from {schema_path}.")
        with open(schema_path, 'r') as schema_file:
            # Carica e restituisci lo schema dal file json
            schema = json.load(schema_file)
    else:
        print(f"Error: Schema file {schema_path} does not exist.")
        return False  # Restituisci None o un altro valore predefinito se il
            file non esiste

    schema_df = pd.DataFrame(schema)# souce_schema-->schema
    dimensions_df = schema_df[schema_df["type"] == "dimension"]
    dimensions_only = dimensions_df['name'].tolist()

    # Match input dimensions with schema dimensions
    matched_dimensions = dimensions_df[dimensions_df["name"].isin(
        list_of_dimensions)]
```

```
    length_check = matched_dimensions.shape[0] == len(list_of_dimensions)

    dimension_check = not matched_dimensions.empty

    check_output = dimension_check and length_check

    print(f'[Dimension Check] {check_output}')

    return check_output



@tool
def json_correction(input_str):
    """
    Call this to correct and format a given string into a valid JSON format.
    """
    turbo_llm = ChatOpenAI(
        temperature=0,
        model_name='gpt-4o-mini')
    prompt = f"""
    Please check this string and make it a string in a correct json format
        that could be easily read using the python command json.loads.

    The string to check and transform is {input_str}.
    Please return ONLY the transformed string without any additional comment
        or explanation.

    """

    out = turbo_llm.invoke(prompt)
    print('[Json Correction] Done')
    return out.content

@tool
def sql_correction(input_str,query):
```

```
    """
    Call this to correct an  ClickHouse sql query given as input.
    """
    turbo_llm = ChatOpenAI(
        temperature=0,
        model_name='gpt-4o-mini')


    prompt = f"""
    Please check and correct the syntax of the following ClickHouse command:

    {input_str}

    Make sure that it is compatibile with the following orginal natural
        language query: {query}.


    Note: Please return ONLY the corrected query without any additional
        comment or explanation. Make sure that it can be easily read as a
        command by a ClickHouse database. Don't change any variable names.

    """

    out = turbo_llm.invoke(prompt)
    print('[SQL Correction] Done')
    return out.content

@tool
def values_extractor(list_of_dimensions, source_name, host,log_directory_path
    ):
    """
    Extracts the distinct values of a list of dimensions.
    """
    host_name= host.replace("https://", "").strip("/").replace("/", "_").
        replace(".com", "").replace("statwolf", "").replace(".","") .replace(
        "__", "_")
```

```
schema_dir = os.path.join(log_directory_path, 'source_schema', host_name,
     source_name)

schema_path = os.path.join(schema_dir, f"{source_name}_schema.json")

with open(schema_path, 'r') as schema_file:
        # Carica e restituisci il file esistente
    full_schema = json.load(schema_file)
    # Filter for type='dimension'
    categorical_dimensions_only = [item['name'] for item in full_schema
        if (item.get('type') == 'dimension' and item.get('data_type')!='
        Float32' and item.get('data_type')!='NA' and item.get('data_type'
        )!='DateTime')]



dim_values={}
# Loop through filters, identifying fields and values by index
for dim in list_of_dimensions:

    if dim in categorical_dimensions_only:
    # Check if the file with distinct values already exists
        unique_file_path = os.path.join(schema_dir, f"{source_name}
            _unique", f"{dim}.json")

        if os.path.exists(unique_file_path):
            # If the file exists, load the distinct values from the file
            try:
                with open(unique_file_path, 'r') as file:
                    distinct_values = json.load(file)
            except Exception as e:
                print(f"Error reading the file {unique_file_path}: {e}")
                return dim_values
        else:
```

```
qp = {
      'timeframe': ('1990-01-01','2030-01-01'), # (date from,
          date to)
      'metrics':['row_check'],
      'dimensions':[dim],
      'take':10000
}
datasource = statwolf.create({
    'host': host,
    'username': 'boss',
    'password': 'test123'
}, 'datasource')




try:

    #data = pd.read_parquet(sql.query(query, format='Parquet
        ', returnDatasetPath=True, forceDatasetRefresh=False)
        ['data']['path'])
    data = datasource.query({'source': source_name,'query':
        qp}, format='Parquet', returnDatasetPath=True,
        forceDatasetRefresh=False)
    #distinct_values = data[field_name].dropna().tolist()
    distinct_values = pd.read_parquet(data['path'])[dim].
        unique().tolist()
    # Save the distinct values to a file for future use
    os.makedirs(os.path.dirname(unique_file_path), exist_ok=
        True)
    with open(unique_file_path, 'w') as file:
        json.dump(distinct_values, file)

except Exception as e:
    print(f"SQL query failed: {e}")
```

```
                    return None

            dim_values[dim]=distinct_values

    print('[Dimension values extraction] Done')
    return dim_values
```

A.3 Agent Functions

Listing A.3: Agent functions

```
import statwolf
import datetime
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers.string import StrOutputParser
from langchain.prompts import ChatPromptTemplate
from langchain_core.utils.function_calling import convert_to_openai_function
#from langchain.tools import tool
from langchain_core.tools import tool
from langchain.chains.conversation.memory import
    ConversationBufferWindowMemory
import json
import os
from langchain.agents.output_parsers import OpenAIFunctionsAgentOutputParser
from langchain.output_parsers.openai_functions import
    JsonOutputFunctionsParser
from langchain_core.output_parsers.json import JsonOutputParser
from langchain.output_parsers.structured import StructuredOutputParser
from langchain_core.prompts import PromptTemplate
from langchain_core.output_parsers.list import CommaSeparatedListOutputParser

from langchain.prompts import MessagesPlaceholder
from langchain_core.messages import HumanMessage, SystemMessage
from langchain_core.prompts import SystemMessagePromptTemplate,
    HumanMessagePromptTemplate
```

```
from langchain.agents.format_scratchpad import format_to_openai_functions
from langchain.schema.runnable import RunnablePassthrough
from langchain_core.runnables import RunnableLambda
from langchain.memory import ConversationBufferMemory
from langchain.agents import AgentExecutor

from langchain_core.prompts import PromptTemplate
import datetime

from langchain_openai import OpenAIEmbeddings

import pandas as pd
#from utils import cosine_similarity

from prompt_templates import Instructions,query_simplifier_prompt_template,
    filter_prompt_template,timeframe_prompt,value_template,
    metric_generator_template, name_generator_template,
    dimension_generator_template, feature_identification_template
from typing_extensions import Annotated, TypedDict
from typing import Literal
from typing import Optional

from pydantic import BaseModel, Field



#---Langraph
#from langgraph.prebuilt import create_react_agent
#from langgraph.checkpoint.memory import MemorySaver



def query_simplifier():
    query_simplifier_prompt = PromptTemplate.from_template(template =
        query_simplifier_prompt_template)
    query_simplifier_model=ChatOpenAI(temperature=0,model_name='gpt-4o-mini')
```

```
    query_simplifier_chain=query_simplifier_prompt|query_simplifier_model|
        StrOutputParser()
    return query_simplifier_chain

def selector_agent_with_qs(query, source_name, host,tools,log_directory_path,
     username, password):

    query_simplifier_chain=query_simplifier()
    functions = [convert_to_openai_function(f) for f in tools]
    model_1 = ChatOpenAI(temperature=0, model_name='gpt-4o-mini').bind(
        functions=functions)
    formatted_instructions = Instructions.format(source_name=source_name,
        host=host, log_directory_path=log_directory_path, username=username,
        password=password)

    memory = ConversationBufferMemory(return_messages=True,memory_key="
        chat_history")
    #main agent prompt
    prompt = ChatPromptTemplate.from_messages([

        SystemMessage(content=formatted_instructions),

        MessagesPlaceholder(variable_name="chat_history"),

        HumanMessagePromptTemplate.from_template("{input}"),

        MessagesPlaceholder(variable_name="agent_scratchpad")
    ])

    chain_1 = prompt | model_1 | OpenAIFunctionsAgentOutputParser()

    agent_chain = RunnablePassthrough.assign(
        agent_scratchpad= lambda x: format_to_openai_functions(x["
            intermediate_steps"])
    ) | chain_1
```

```
    first_agent_executor = AgentExecutor(agent=agent_chain, tools=tools,
        verbose=False, memory=memory,max_iterations=10)

    #composing agent chain with simplified query
    composed_agent_chain = (
    query_simplifier_chain
    | (lambda input: {"input": input})
    | first_agent_executor
    )


    first_agent_output=composed_agent_chain.invoke({"query":query})
    output_dict=json.loads(first_agent_output['output'])

    return output_dict

def selector_agent_lg(query, source_name, host,tools):

    #query_simplifier_chain=query_simplifier()
    #functions = [convert_to_openai_function(f) for f in tools]
    model_1 = ChatOpenAI(temperature=0, model_name='gpt-4o-mini')#.bind(
        functions=functions)
    formatted_instructions = Instructions.format(source_name=source_name,
        host=host)


    memory = MemorySaver()
    langgraph_agent_executor = create_react_agent(
        model_1, tools, state_modifier=formatted_instructions, checkpointer=
            memory
    )
    config = {"configurable": {"thread_id": "test-thread"}}
```

```
    #composing agent chain with simplified query
    #composed_agent_chain = (
    #query_simplifier_chain
    #| (lambda input: {"input": input})
    #| langgraph_agent_executor
    #)
    first_agent_output=langgraph_agent_executor.invoke({"input":query},config
        )["messages"][-1].content
    output_dict=json.loads(first_agent_output)

    #first_agent_output=composed_agent_chain.invoke({"query":query},config)["
        messages"][-1].content
    #output_dict=json.loads(first_agent_output['output'])

    return output_dict

def selector_agent(query, source_name, host,tools,log_directory_path,
    username, password):

    functions = [convert_to_openai_function(f) for f in tools]
    model_1 = ChatOpenAI(temperature=0, model_name='gpt-4o-mini').bind(
        functions=functions)
    #formatted_instructions = Instructions_without_qs.format(source_name=
        source_name, host=host)
    formatted_instructions = Instructions.format(source_name=source_name,
        host=host, log_directory_path=log_directory_path, username=username,
        password=password)

    memory = ConversationBufferMemory(return_messages=True,memory_key="
        chat_history")
    #main agent prompt
    prompt = ChatPromptTemplate.from_messages([

        SystemMessage(content=formatted_instructions),
```

```
        MessagesPlaceholder(variable_name="chat_history"),

        HumanMessagePromptTemplate.from_template("{input}"),

        MessagesPlaceholder(variable_name="agent_scratchpad")
    ])

    chain_1 = prompt | model_1 | OpenAIFunctionsAgentOutputParser()

    agent_chain = RunnablePassthrough.assign(
        agent_scratchpad= lambda x: format_to_openai_functions(x["
            intermediate_steps"])
    ) | chain_1




    first_agent_executor = AgentExecutor(agent=agent_chain, tools=tools,
        verbose=False, memory=memory,max_iterations=15)


    first_agent_output=first_agent_executor.invoke({"input":query})
    output_dict=json.loads(first_agent_output['output'])

    return output_dict

def filter_corrector(input_dict,query):
    output_dict=input_dict.copy()
    filter_prompt=PromptTemplate.from_template(template=
        filter_prompt_template)
    model_3 = ChatOpenAI(temperature=0,model_name='gpt-4o-mini')
    filter_chain=filter_prompt| model_3 | JsonOutputParser()
    # Process the filter with the chain
    filter_result = (filter_chain.invoke({"input": input_dict['filter'], "
        query": query}))['filter']
```

```
    # Set the last selector to 'AND' manually
    if isinstance(filter_result, dict):

        selector_indices = [int(k.split('_')[-1]) for k in filter_result if k
            .startswith('selector_')]

        if selector_indices:
            last_index = max(selector_indices)
            filter_result[f'selector_{last_index}'] = 'AND'
        # Perform actions with last_index if needed
        else:
        # Do nothing if selector_indices is empty
            pass
    # Update the input dictionary
    output_dict['filter'] = filter_result
    print('[Filter correction] Done')
    return output_dict




def timeframe_corrector_without_struct(input_dict):
    output_dict=input_dict.copy()
    day = str(datetime.datetime.now()).split('.')[0:-1][0]
    model_2 = ChatOpenAI(temperature=0,model_name='gpt-4o-mini')
    timeframe_chain = RunnableLambda(lambda x: timeframe_prompt.format(day=
        day) + x ) | model_2 | JsonOutputParser()
    output_dict['timeframe']=timeframe_chain.invoke(json.dumps(input_dict['
        timeframe']))
    return output_dict


def timeframe_corrector(input_dict):


    class time_frame_structure(BaseModel):
```

```
#         """Structure of timeframe"""



        period: Literal["alltime","currentYear","lastYear","months","
            lastXMonths","months",
                                    "last3days","last7days","last14days","
                                        last30days","last90days","currentMonth"
                                        ,"lastMonth","lastXDays","days"]=Field(
                                        default="alltime",description=
                            "If granularity is 'month', it can take one of: '
                                alltime','currentYear','lastYear','months','
                                lastXMonths','months', if granularity is 'day'
                                it can take one of: 'last3days','last7days','
                                last14days','last30days','last90days','
                                currentMonth','lastMonth','lastXDays','days'.")
        granularity: Literal["day","month"]=Field( default="month",
            description="This indicates the analysis level of the timeframe
            filter.")

        dateFrom: Optional[str]=Field(description=" Start date in the format
            'YYYY-MM-DD'. Only use this key if 'period' is set to 'days' or '
            months' ")
        dateTo: Optional[str]=Field(description=" End date in the format '
            YYYY-MM-DD'. Only use this key if 'period' is set to 'days' or '
            months' ")
        timeFrom: Optional[str]=Field(description=" Start time in the format
            'hh:mm'. Only use this key if 'period' is set to 'days' or '
            months' ")
        timeTo: Optional[str]=Field(description=" End time in the format 'hh:
            mm'. Only use this key if 'period' is set to 'days' or 'months' "
            )
        auxval: Optional[int]=Field(description= " This indicates the Number
            of months or days if 'lastXMonths' or 'lastXDays' is assigned to
            the 'period' in this dictionary. Only use if 'period' is set to '
            lastXMonths' or 'lastXDays'")
```

```
    output_dict=input_dict.copy()
    day = str(datetime.datetime.now()).split('.')[0:-1][0]
    model_2 = ChatOpenAI(temperature=0,model_name='gpt-4o-mini').
        with_structured_output(time_frame_structure)
    timeframe_chain = RunnableLambda(lambda x: timeframe_prompt.format(day=
        day) + x ) | model_2# | JsonOutputParser()
    output_dict['timeframe']=timeframe_chain.invoke(json.dumps(input_dict['
        timeframe'])).dict(exclude_unset=True)
    print('[TimeFrame correction] Done')
    return output_dict



# Main function to validate and replace filter values
def validate_filter_values(input_dict, source_name, host,log_directory_path):
    output_dict=input_dict.copy()
    filters = output_dict["filter"]

    host_name= host.replace("https://", "").strip("/").replace("/", "_").
        replace(".com", "").replace("statwolf", "").replace(".","") .replace(
        "__", "_")

    schema_dir = os.path.join(log_directory_path, 'source_schema', host_name,
         source_name)


    # Loop through filters, identifying fields and values by index
    for index in range(len(filters) // 4):  # Adjust range if structure
        changes
        field_key = f"field_{index}"
        value_key = f"value_{index}"
```

```
        # Check if both field and value exist
        if field_key in filters and value_key in filters and isinstance(
            filters[value_key], list):
            field_name = filters[field_key]
            value_list = filters[value_key]

            # Check if the file with distinct values already exists
            unique_file_path = os.path.join(schema_dir, f"{source_name}
                _unique", f"{field_name}.json")

            if os.path.exists(unique_file_path):
                # If the file exists, load the distinct values from the file
                try:
                    with open(unique_file_path, 'r') as file:
                        distinct_values = json.load(file)
                except Exception as e:
                    print(f"Error reading the file {unique_file_path}: {e}")
                    return output_dict
            else:
                # If the file doesn't exist, execute SQL query to get
                    distinct values



                qp = {
                  'timeframe': ('1990-01-01','2030-01-01'), # (date from,
                      date to)
                  'metrics':['row_check'],
                  'dimensions':[field_name],
                  'take':10000
                }
                datasource = statwolf.create({
                'host': host,
                'username': 'boss',
                'password': 'test123'
                }, 'datasource')
```

```
        try:

            #data = pd.read_parquet(sql.query(query, format='Parquet
                ', returnDatasetPath=True, forceDatasetRefresh=False)
                ['data']['path'])
            data = datasource.query({'source': source_name,'query':
                qp}, format='Parquet', returnDatasetPath=True,
                forceDatasetRefresh=False)
            #distinct_values = data[field_name].dropna().tolist()
            distinct_values = pd.read_parquet(data['path'])[
                field_name].unique().tolist()
            # Save the distinct values to a file for future use
            os.makedirs(os.path.dirname(unique_file_path), exist_ok=
                True)
            with open(unique_file_path, 'w') as file:
                json.dump(distinct_values, file)

        except Exception as e:
            print(f"SQL query failed: {e}")
            return output_dict


    # Replace the original value with the modified list of closest
        matches
    value_llm=ChatOpenAI(model='gpt-4o-mini')


    value_prompt=PromptTemplate.from_template(value_template)
    value_chain=value_prompt|value_llm|StrOutputParser()#|
        CommaSeparatedListOutputParser()
```

```
            filters[value_key] = eval(value_chain.invoke({'value':value_list,
                'distinct_values':distinct_values}))


            # New Step: Remove field_name from dimensions if it has only one
                value in value_list
            if len(value_list) == 1 and field_name in output_dict["dimensions
                "]:
                output_dict["dimensions"].remove(field_name)

    print('[Value Validation] Done')
    return output_dict


#creare funzione agente per metric

def metric_generator_agent(query, source_name, host,tools,log_directory_path,
     username, password):

    functions = [convert_to_openai_function(f) for f in tools]
    model_1 = ChatOpenAI(temperature=0, model_name='gpt-4o-mini').bind(
        functions=functions)
    #formatted_instructions = Instructions_without_qs.format(source_name=
        source_name, host=host)
    formatted_instructions = metric_generator_template.format(source_name=
        source_name, host=host, log_directory_path=log_directory_path,
        username=username, password=password)

    memory = ConversationBufferMemory(return_messages=True,memory_key="
        chat_history")
    #main agent prompt
    prompt = ChatPromptTemplate.from_messages([

        SystemMessage(content=formatted_instructions),

        MessagesPlaceholder(variable_name="chat_history"),
```

```
        HumanMessagePromptTemplate.from_template("{input}"),

        MessagesPlaceholder(variable_name="agent_scratchpad")
    ])

    chain_1 = prompt | model_1 | OpenAIFunctionsAgentOutputParser()

    agent_chain = RunnablePassthrough.assign(
        agent_scratchpad= lambda x: format_to_openai_functions(x["
            intermediate_steps"])
    ) | chain_1




    first_agent_executor = AgentExecutor(agent=agent_chain, tools=tools,
        verbose=False, memory=memory,max_iterations=10)


    first_agent_output=first_agent_executor.invoke({"input":query})
    sql_def=first_agent_output['output']

    return sql_def

def dimension_generator_agent(query, source_name, host,tools,
    log_directory_path, username, password):

    functions = [convert_to_openai_function(f) for f in tools]
    model_1 = ChatOpenAI(temperature=0, model_name='gpt-4o-mini').bind(
        functions=functions)
    #formatted_instructions = Instructions_without_qs.format(source_name=
        source_name, host=host)
    formatted_instructions = dimension_generator_template.format(source_name=
        source_name, host=host, log_directory_path=log_directory_path,
        username=username, password=password)
```

```
memory = ConversationBufferMemory(return_messages=True,memory_key="
    chat_history")
#main agent prompt
prompt = ChatPromptTemplate.from_messages([

    SystemMessage(content=formatted_instructions),

    MessagesPlaceholder(variable_name="chat_history"),

    HumanMessagePromptTemplate.from_template("{input}"),

    MessagesPlaceholder(variable_name="agent_scratchpad")
])

chain_1 = prompt | model_1 | OpenAIFunctionsAgentOutputParser()

agent_chain = RunnablePassthrough.assign(
    agent_scratchpad= lambda x: format_to_openai_functions(x["
        intermediate_steps"])
) | chain_1




first_agent_executor = AgentExecutor(agent=agent_chain, tools=tools,
    verbose=False, memory=memory,max_iterations=10)


first_agent_output=first_agent_executor.invoke({"input":query})
sql_def=first_agent_output['output']

return sql_def
```

```
def name_generator(sql_def, feature):

    class feature_definition(TypedDict):
        """Structure of feature definition"""

        name: Annotated[str,...,"The name generated for the metric or
            dimension"]
        definition: Annotated[str,...,"The query provided as input"]
        type: Annotated[Literal['metric','dimension'], ..., "The type
            provided as input"]

    name_llm=ChatOpenAI(model='gpt-4o-mini').with_structured_output(
        feature_definition)


    name_prompt=PromptTemplate.from_template(name_generator_template)
    name_chain=name_prompt|name_llm#|JsonOutputParser()

    name=name_chain.invoke({'input':sql_def,'feature':feature})


    return name




def feature_generator_agent(query, source_name, host,tools,log_directory_path
    , username, password, feature):

    functions = [convert_to_openai_function(f) for f in tools]
    model_1 = ChatOpenAI(temperature=0, model_name='gpt-4o-mini').bind(
        functions=functions)
```

```
    #formatted_instructions = Instructions_without_qs.format(source_name=
        source_name, host=host)

    if feature.lower()=='metric':
        formatted_instructions = metric_generator_template.format(source_name
            =source_name, host=host, log_directory_path=log_directory_path,
            username=username, password=password)
    elif feature.lower()=='dimension':
        formatted_instructions = dimension_generator_template.format(
            source_name=source_name, host=host, log_directory_path=
            log_directory_path, username=username, password=password)
    else:
        return {'Error':f"{feature} is not among the recognised features.
            Please set it to either 'metric' or 'dimension'" }

    memory = ConversationBufferMemory(return_messages=True,memory_key="
        chat_history")
    #main agent prompt
    prompt = ChatPromptTemplate.from_messages([

        SystemMessage(content=formatted_instructions),

        MessagesPlaceholder(variable_name="chat_history"),

        HumanMessagePromptTemplate.from_template("{input}"),

        MessagesPlaceholder(variable_name="agent_scratchpad")
    ])

    chain_1 = prompt | model_1 | OpenAIFunctionsAgentOutputParser()

    agent_chain = RunnablePassthrough.assign(
        agent_scratchpad= lambda x: format_to_openai_functions(x["
            intermediate_steps"])
    ) | chain_1
```

```
    first_agent_executor = AgentExecutor(agent=agent_chain, tools=tools,
        verbose=False, memory=memory,max_iterations=15)


    first_agent_output=first_agent_executor.invoke({"input":query})
    sql_def=first_agent_output['output']

    return sql_def




def feature_identification(query):

        # TypedDict
    class feature_select(TypedDict):
        """Select feature:Dimension or Metric"""

        feature: Annotated[Literal['metric','dimension'], ..., "Assign the
            description to metric or dimension"]


    feature_prompt=PromptTemplate.from_template(template=
        feature_identification_template)
    llm=ChatOpenAI(temperature=0,model_name='gpt-4o-mini')
    structured_llm = llm.with_structured_output(feature_select)
    chain=feature_prompt|structured_llm

    return chain.invoke({'query':query})['feature']
```

B

Agent Testing Datasets

B.1 Queries used for testing the structured Query generator

Listing B.1: Italian Queries used for testing the structured Query generator

```
[
    "Calcola la media delle previsioni superiori a 10 per la variabile 'fan'
        negli ultimi 10 giorni",
    "Forniscimi la media delle previsioni della variabile 'fan' che superano
        10 negli ultimi 10 giorni",
    "Mostrami la media delle previsioni della variabile 'fan' negli ultimi 10
         giorni, considerando solo quelle superiori a 10",
    "Mostrami la media delle previsioni per ciascuna variabile negli ultimi
        30 giorni che superano il valore di 20",
    "Forniscimi la media delle previsioni di tutte le variabili degli ultimi
        30 giorni che sono superiori a 20",
    "Dammi la media delle previsioni per ogni variabile negli ultimi 30
        giorni, considerando solo quelle che hanno valori maggiori di 20",
    "mostrami le previsioni per la variabile 'eev1_perc' negli ultimi 7
        giorni che superano 50 o sono inferiori a 10",
```

```
    "fammi vedere le previsioni con variabile 'eev1_perc' negli ultimi 7
       giorni che sono superiori a 50 o inferiori a 10",
    "visualizza le previsioni per la variabile 'eev1_perc' negli ultimi 7
       giorni che sono maggiori di 50 o minori di 10",
    "Mostrami la media delle previsioni per ciascuna variabile negli ultimi
       30 giorni",
    "Puoi mostrarmi la media delle previsioni per ognuno delle variabili
       negli ultimi 30 giorni?",
    "Fammi vedere la media delle previsioni di ogni variabile negli ultimi 30
        giorni",
    "mostrami i residui totali delle ultime 2 settimane per i Titan Sky Hi R0
        5.1 1P su tutte le variabili",
    "puoi mostrarmi i residui totali delle ultime 2 settimane relativi ai
       Titan Sky Hi R0 5.1 1P per ogni variabile?",
    "fammi vedere i residui totali per tutte le variabili dei Titan Sky Hi R0
        5.1 1P nelle ultime 2 settimane",
    "Mostrami la richiesta media di potenza e la percentuale di anomalie per
       ogni tipo di dispositivo con compressione accesa negli ultimi 2 mesi
       ",
    "Forniscimi la media della richiesta di potenza e la percentuale di
       anomalie per ognuno dei dispositivi con compressione 'ON' negli
       ultimi 2 mesi",
    "Visualizza la richiesta media di potenza e la percentuale di anomalie
       per ognuno dei dispositivi che hanno la compressione accesa negli
       ultimi 2 mesi",
    "Mostrami la media delle predizioni, la percentuale di anomalie e il
       punteggio medio di anomalia da gennaio ad agosto 2024, per ognuno i
       dispositivi con la prima causa 'bassa portata in CH'",
    "Forniscimi la predizione media, la percentuale di anomalie e il
       punteggio medio di anomalia per il periodo da gennaio ad agosto 2024,
        per ognuno dei dispositivi in cui la causa principale \u00e8 'bassa
       portata in CH'",
    "Dammi la media delle predizioni, la percentuale di anomalie e il
       punteggio medio di anomalia da gennaio a agosto 2024, per ognuno dei
       dispositivi dove la prima causa \u00e8 'bassa portata in CH'"
]
```

B.2 Feature Queries used for testing the feature generator

Listing B.2: Italian Feature Queries used for testing the feature generator

```
{
    "dimensions": {
        "anomaly_score": "Calcola il punteggio di anomalia in base al metodo
            di isolamento e alla soglia dove restituisce 1 se un valore
            negativo, moltiplicato per -3, supera 1; il prodotto se resta
            sotto 1; altrimenti 0.",
        "month": "Estrai il mese dal timestamp.",
        "calendar_week": "Estrai la settimana ISO dal timestamp.",
        "compressor_state_corrected": "Correggi lo stato del compressore in
            base al cambiamento del suo stato.",
        "corrected_external_air_temperature": "Correggi la temperatura
            esterna 'dellaria, assicurandoti che non superi i 200.",
        "day_of_month": "Estrai il giorno del mese dal timestamp.",
        "delta_t3_too_low": "Classifica i valori di delta_t3 in base alla
            soglia: assegna 'Critical' se il valore è inferiore a 5, 'Low' se
             è tra 5 e 10 (escluso), e 'Normal' se è pari o superiore a 10.
            Restituisce 'N/A' per altri casi.",
        "delta_tw_out_of_bounds": "Determina se il valore di delta_tw è fuori
             dai limiti in base allo stato operativo: restituisce 'Out' se è
            maggiore di 5 con stato 'Heat Pump' o minore di 5 con stato '
            Chiller'. Altrimenti, assegna 'Normal'.",
        "delta_tw_too_high": "Classifica il valore di delta_tw: restituisce '
            High' se supera 5, 'Normal' se è pari o inferiore a 5, e 'N/A'
            per altri casi.",
        "inline_month": "Restituisci il nome del mese con il codice
            formattato in base al timestamp.",
        "is_anomaly_corrected": "Determina se 'lanomalia è stata corretta in
            base al suo stato,assegnando i valori 'Fault' o 'Normal'",
        "is_daily_outlier": "Verifica se il numero di guasti giornalieri
            supera 5760*0.25 per essere classificato come 'Fault' o 'Normal
            '.",
```

```
        "is_outlier": "Verifica se il valore è un outlier in base ai limiti
            previsti e se la previsione è stata effettuata.",
        "pred_upper_boundaries": "Quando la variabile è 'eev1_perc', limita
            il valore del confine superiore previsto a 100 se supera tale
            valore.",
        "rule_violated": "Verifica se una delle regole è violata('true')
            assegnando 'true' o 'false'",
        "timestamp_hour": "Estrai 'lora dal timestamp e arrotondala
            'allinizio 'dellora.",
        "residual_interval": "Calcola 'lintervallo residuo per i valori che
            superano i limiti previsti, indicando un outlier."
    },
    "metrics": {
        "avg_anomaly_score": "Calcola la media dei punteggi di anomalia.",
        "avg_combined_score_timestamp_residuals_outlier": "Calcola il
            punteggio combinato medio considerando 'lintervallo residuo e
            'lunicità dei timestamp.",
        "avg_correct_external_temperature": "Calcola la media della
            temperatura esterna corretta 'dellaria.",
        "sum_starting_defrost": "Conta il numero di avvii del ciclo di
            sbrinamento.",
        "sum_is_anomaly": "Conta il numero totale di anomalie rilevate.",
        "avg_defrost_duration_seconds": "Calcola la durata media del ciclo di
             sbrinamento in secondi.",
        "avg_is_daily_outlier": "Calcola la proporzione media di outlier
            giornalieri contrassegnati come guasti.",
        "avg_is_outlier": "Calcola la proporzione media di outlier su tutti i
             timestamp.",
        "avg_outlet_common_user": "Calcola la media del valore 'dellutente
            comune 'alluscita, limitato a 200.",
        "avg_pred_lower": "Calcola la media del limite inferiore previsto e
            se questa media è negativa, moltiplicala per 1.15; se è positiva,
             moltiplicala per 0.85.In caso contrario restituisci 0",
        "avg_pred_upper": "Calcola la media del limite superiore previsto e
            se questa media è positivo, moltiplicala per 1.15; se è negativa,
             moltiplicala per 0.85.In caso contrario restituisci 0",
```

```
        "avg_proportional_residuals": "Calcola la media dei residui
            proporzionali, escludendo i valori normali e N/A.",
        "consecutive_time_super_heating_too_low": "Calcola il tempo massimo
            trascorso 'dallultimo cambiamento di riscaldamento eccessivo
            troppo basso.",
        "max_time_from_latest_compressor_status_changed": "Calcola il tempo
            massimo trascorso 'dallultimo cambiamento dello stato del
            compressore.",
        "number_of_defrost": "Conta il numero di cicli di sbrinamento unici
            eseguiti.",
        "sum_defrost_duration_seconds": "Calcola la somma totale della durata
             dei cicli di sbrinamento in secondi."
    }
}
```

Listing B.3: Feature definitions

```
{
        "dimensions":{
                "anomaly_score": "case\n    when anomaly_score_isolation < 0
                    and -1*anomaly_score_isolation * 3 > 1 then 1\n    when
                    anomaly_score_isolation < 0 and -1*
                    anomaly_score_isolation * 3 < 1 then -1*
                    anomaly_score_isolation*3\n    else 0\nend",
                "month": "toMonth(timestamp)",
                "calendar_week": "toISOWeek(timestamp)",
                "compressor_state_corrected": "case \n    when
                    compressor_state = 'On' and compressor_status_changed = '
                    true' then 'TurnOff'\n    when compressor_state = 'Off'
                    and compressor_status_changed = 'true' then 'TurnOn'\n
                       else compressor_state\nend",
                "corrected_external_air_temperature": "case\n  when
                    external_air_temperature < 200 then
                    external_air_temperature\n   else 200\n  end",
                "day_of_month": "toDate(timestamp)",
                "delta_t3_too_low": "case\n    when variable = 'delta_t3' and
                     true_val < 5 then 'Critical'\n    when variable = '
```

```
                    delta_t3' and true_val < 10 then 'Low'\n    when variable
                     = 'delta_t3' and true_val >= 10 then 'Normal'\n    else
                    'N/A'\nend",
                "delta_tw_out_of_bounds": "case \n    when variable = '
                    delta_tw' and true_val > 5 and status= 'Heat Pump' then '
                    Out'\n    when variable = 'delta_tw' and true_val < 5 and
                     status= 'Chiller' then 'Out'\n    else 'Normal'\nend\n
                       ",
                "delta_tw_too_high": "case\n    when variable = 'delta_tw'
                    and true_val > 5 then 'High'\n    when variable = '
                    delta_tw' and true_val <= 5 then 'Normal'\n    else 'N/A
                    '\nend",
                "inline_month": "CASE\n WHEN toMonth(timestamp) = 1 THEN
                    '(01) January'\n WHEN toMonth(timestamp) = 2 THEN '(02)
                    February'\n WHEN toMonth(timestamp) = 3 THEN '(03) March
                    '\n WHEN toMonth(timestamp) = 4 THEN '(04) April'\n WHEN
                    toMonth(timestamp) = 5 THEN '(05) May'\n WHEN toMonth(
                    timestamp) = 6 THEN '(06) June'\n WHEN toMonth(timestamp)
                     = 7 THEN '(07) July'\n WHEN toMonth(timestamp) = 8 THEN
                    '(08) August'\n WHEN toMonth(timestamp) = 9 THEN '(09)
                    September'\n WHEN toMonth(timestamp) = 10 THEN '(10)
                    October'\n WHEN toMonth(timestamp) = 11 THEN '(11)
                    November'\n WHEN toMonth(timestamp) = 12 THEN '(12)
                    December'\n ELSE 'N/A'\n END",
                "is_anomaly_corrected": "case\n when is_anomaly = 1 then '
                    Fault'\n else 'Normal'\n end",
                "is_daily_outlier": "case\n when fault_count_daily >
                    5760*0.25 then 'Fault'\n else 'Normal'\n end",
                "is_outlier": "case\n when true_val > pred_upper_boundries
                    *1.15 and prediction_done = 'yes' and
                    pred_upper_boundries > 0 then 'Upper Outlier'\n when
                    true_val > pred_upper_boundries*0.85 and prediction_done
                    = 'yes' and pred_upper_boundries < 0 then 'Upper Outlier
                    '\n when true_val < pred_lower*0.85 and prediction_done =
                     'yes' and pred_lower > 0 then 'Lower Outlier'\n when
                    true_val < pred_lower*1.15 and prediction_done = 'yes'
```

```
                    and pred_lower < 0 then 'Lower Outlier'\n else 'Normal'\n
                     end",
                "pred_upper_boundaries": "case\n when variable == 'eev1_perc'
                     and pred_upper > 100 then 100\n else pred_upper\n end",
                "rule_violated": "case\n when rule_1 = 'true' or rule_2 = '
                    true' or rule_3 = 'true' or rule_4 = 'true' or rule_5 = '
                    true' or rule_6 = 'true' then 'true'\n  else 'false'\n
                    end",
                "timestamp_hour": "toStartOfHour(timestamp)",
                "residual_interval": "case\n when true_val >
                    pred_upper_boundries*1.15 and prediction_done = 'yes' and
                     pred_upper > 0 then abs(abs(true_val) - abs(pred_upper
                    *1.15))\n when true_val > pred_upper_boundries*0.85 and
                    prediction_done = 'yes' and pred_lower < 0 then abs(abs(
                    true_val) - abs(pred_upper*0.85))\n when true_val <
                    pred_lower*0.85 and prediction_done = 'yes' and
                    pred_lower > 0 then  abs(abs(true_val) - abs(pred_lower
                    *0.85))\n when true_val < pred_lower*1.15 and
                    prediction_done = 'yes' and pred_lower < 0 then  abs(abs(
                    true_val) - abs(pred_lower*1.15))\n else 0\n end"
        },
        "metrics":{
                "avg_anomaly_score": "avg(anomaly_score)",
                "avg_combined_score_timestamp_residuals_outlier": "avg(
                    residuals_interval) * uniq(timestamp)",
                "avg_correct_external_temperature": "avg(
                    correct_external_air_temperature)",
                "sum_starting_defrost": "sum(starting_defrost)",
                "sum_is_anomaly": "sum(is_anomaly)",
                "avg_defrost_duration_seconds": "avg(defrost_duration_seconds
                    )",
                "avg_is_daily_outlier": "avgIf(1,is_daily_outlier='Fault')",
                "avg_is_outlier": "uniqIf(timestamp,  is_outlier != 'Normal'
                    and prediction_done = 'yes')/uniq(timestamp)",
                "avg_outlet_common_user": "avg(case\n when oulet_common_user
                    < 200 then oulet_common_user\n else 200\n end\n )",
```

```
                "avg_pred_lower": "case\n when avgIf(pred_lower,
                    prediction_done='yes')<0 then avgIf(pred_lower,
                    prediction_done='yes')*1.15\n when avgIf(pred_lower,
                    prediction_done='yes')>0 then avgIf(pred_lower,
                    prediction_done='yes')*0.85\n else 0\n end",
                "avg_pred_upper": "case\n  when avgIf(pred_upper_boundries,
                    prediction_done='yes')>0 then avgIf(pred_upper_boundries,
                    prediction_done='yes')*1.15\n when avgIf(
                    pred_upper_boundries,prediction_done='yes')<0 then avgIf(
                    pred_upper_boundries,prediction_done='yes')*0.85\n else
                    0\n end",
                "avg_proportional_residuals": "abs(avgIf(true_val-pred,
                    is_outlier not in ('Normal','N/A')))/abs((max(true_val-
                    pred)-min(true_val-pred)))",
                "consecutive_time_super_heating_too_low": "maxIf(timestamp -
                    parseDateTimeBestEffortOrZero(
                    super_heating_too_low_change), \n
                    super_heating_too_low_change not in ('1970-01-01
                    00:00:00','N/A','',' ') \n and super_heating_too_low != '
                    Normal')",
                "max_time_from_latest_compressor_status_changed": "maxIf(
                    timestamp - parseDateTimeBestEffortOrZero(
                    latest_compressor_status_changed), \n
                    latest_compressor_status_changed!='1970-01-01 00:00:00')
                    ",
                "number_of_defrost": "uniq(defrost_cycle)",
                "sum_defrost_duration_seconds": "arrayReduce('sum',arrayMap(x
                    -> x.2, groupUniqArray((defrost_cycle,
                    defrost_duration_seconds))))"
        }
}
```