FORMAL LANGUAGES AND COMPILERS
PROJECT REPORT

MANAGING LAST-MILE WAREHOUSE LOGIC USING OWLLAMA AGENT

STUDENTS: Alessio Mattiace, Claudio Saponaro

INDEX

ABSTRACT

This project aims to create a chatbot interface used by users in our system to know information quickly and easily about a last mile delivery warehouse and perform specific changes.

There are two types of users we can log in as: warehouse operator and administrator. The operator has certain restrictions and does not have access to all information and functions, while the admin has full control.

Knowledge graph will be queried via the LLM to obtain information about the warehouse or modified through specific functions using a custom-designed logic when requested by an authorized user.

We use an OWLlama agent to manipulate the knowledge graph which used Llama 3.1 as LLM and the COWL API as an interface that integrates the ontology into the LLM's available knowledge base.

From the final tests conducted and documented on specific queries, the great potential of this system has emerged.

SECTION 1 – State of the Art in Large Language Models and Retrieval-Augmented Generation

Intro

This section examines the current state of the art by presenting Llama 3.1, analyzing the limitations of classical LLMs, and introducing RAG as a solution, culminating with an overview of its pipeline.

Model Decision

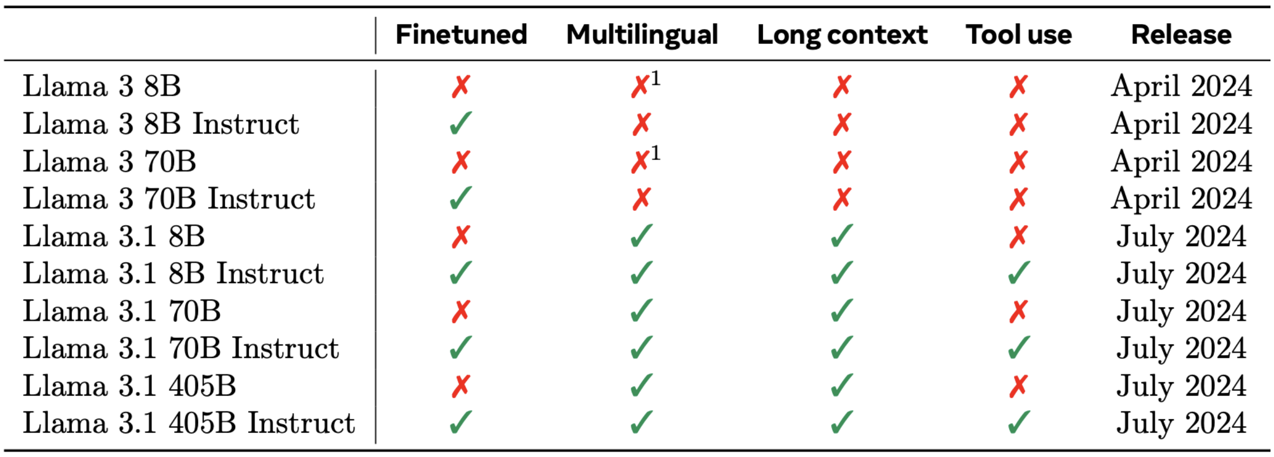
Firstly, we have decided to use Llama 3.1 since it represents the state of art of open-source large language model in comparison to the closed-source LLM such as GPT-4 or Claude.

To utilize Llama 3.1 in a local environment we have used the Ollama platform to pull the manifest of llama3.1, this has been done running the command :

ollama pull llama3.1

Typically, by default this command pulls the most recent and smaller-size model which is optimized for local deployment and performance.

In particular launching the command: ollama show llama3.1, it's possible to explore all the details of the model installed locally.



	Finetuned	Multilingual	Long context	Tool use	Release
Llama 3 8B	✗	✗[1]	✗	✗	April 2024
Llama 3 8B Instruct	✓	✗	✗	✗	April 2024
Llama 3 70B	✗	✗[1]	✗	✗	April 2024
Llama 3 70B Instruct	✓	✗	✗	✗	April 2024
Llama 3.1 8B	✗	✓	✓	✗	July 2024
Llama 3.1 8B Instruct	✓	✓	✓	✓	July 2024
Llama 3.1 70B	✗	✓	✓	✗	July 2024
Llama 3.1 70B Instruct	✓	✓	✓	✓	July 2024
Llama 3.1 405B	✗	✓	✓	✗	July 2024
Llama 3.1 405B Instruct	✓	✓	✓	✓	July 2024

Fig. 1 – Comparison between Llama models [1].

As illustrated in the table, we have employed the Llama3.1 model with 8B parameters updated to July 2024 which exhibits the following characteristics:

- Architecture: llama
- Parameters: 8.0B
- Context length: 131072 tokens
- Embedding dimensionality: 4096
- Quantization: Q4_0

Despite its advanced capabilities, models like this one excel at general language tasks but have trouble answering specific questions for several reasons such as:

- Absence of domain-specific knowledge (without fine-tuning)
- Susceptibility to hallucination (not correct responses)
- Lack of real-time knowledge integration
- Lack of explainability

RAG based architecture

To address this problem, we have used a variant technique of traditional RAG called GraphRAG (Retrieve Augmented Generation) which allowed us to connect the knowledge graph, of our scenario (the warehouse), to the built-in knowledge of Llama3.1 to obtain precise responses, however we will see in the following section how we have built this scenario.

In general RAG is a way to improve the response of a Large Language Model by retrieving external sources of information from external data stores to augment generated responses, in our case the relevant part of the knowledge graph where nodes and edges correspond to the OWL's constructs.

At high level, RAG architecture involves three key important steps which are: query understanding, retrieving information and generating responses.

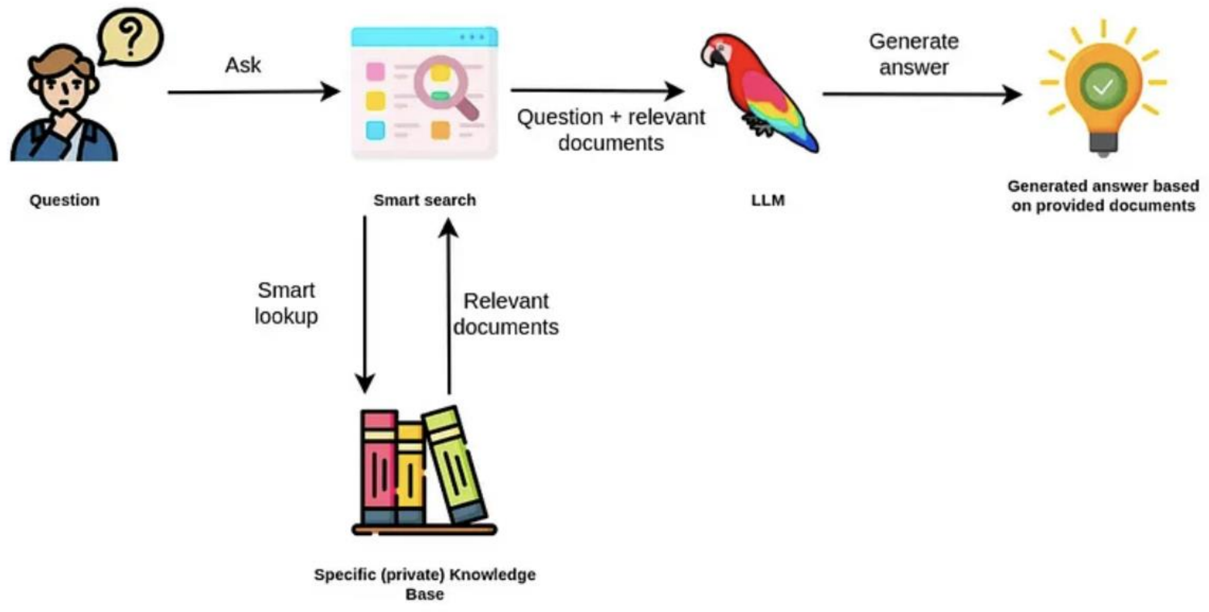

Question
Ask
Smart search
Question + relevant documents
LLM
Generate answer
Generated answer based on provided documents
Smart lookup
Relevant documents
Specific (private) Knowledge Base

Fig. 2 – Pipeline of a RAG system [2].

The first phase is query understanding where the prompt defined by the user goes into the LLM API, in our case we have used the COWL API, developed by SisInf Lab to connect the LLM to the RAG application known as Owllama (which has been developed as well by SisInf Lab).

The second step is the retrieval where the application (Owllama) search for the key elements in the query to retrieve, from the ontology, which are relevant for the response.

In order to obtain the relevant information complex algorithms are involved, in the Owllama's application has been used the FAISS library (Facebook AI Similarity Search) for calculating in an efficient way the similarity between the query and relevant element of the ontology.

The last step is the response generation which combines the retrieved information with the user's original prompt to create a more detailed and context-rich prompt.

SECTION 2 – Ontology structure

In the first phase of our work, we have designed suitable ontology for our purpose: emulating the structure of a real warehouse. It must contain all the necessary entities correctly related to each other to perform the required functions.

We have used OWL (Ontology Web Language): it is a W3C standard language used to create and manage ontologies on the web. It allows the representation of complex knowledge by defining classes, properties, and relationships between concepts, facilitating the sharing and integration of semantic data.
OWL is a formal language that uses symbols and rules to define well-formed strings and structures, describing real-world models and scenarios precisely and unambiguously.

Among the available syntaxes, we chose to use functional syntax because it is more fluent and readable.

2.1 – Ontology structure and classes configuration

In particular, the scenario we have crafted required several key entities:

- Sectors into which the warehouse is divided (Sector_A, Sector_B, Sector_C)
- Shelves on which the products are located (Shelf_A1, Shelf_C6, etc.)
- Types of products that can be added to the shelves (Computers, Books, etc.)
- Vehicles for order delivery (Vehicle_1, etc.)
- Orders to be prepared and loaded onto a vehicle for delivery (Order_1, etc.)

These entities were modelled as classes in our ontology, but without any specific class-subclass hierarchy among them: it was unnecessary for our purpose.

Below is an illustration of the general structure of the ontology, including the relationships between instances of the various classes:

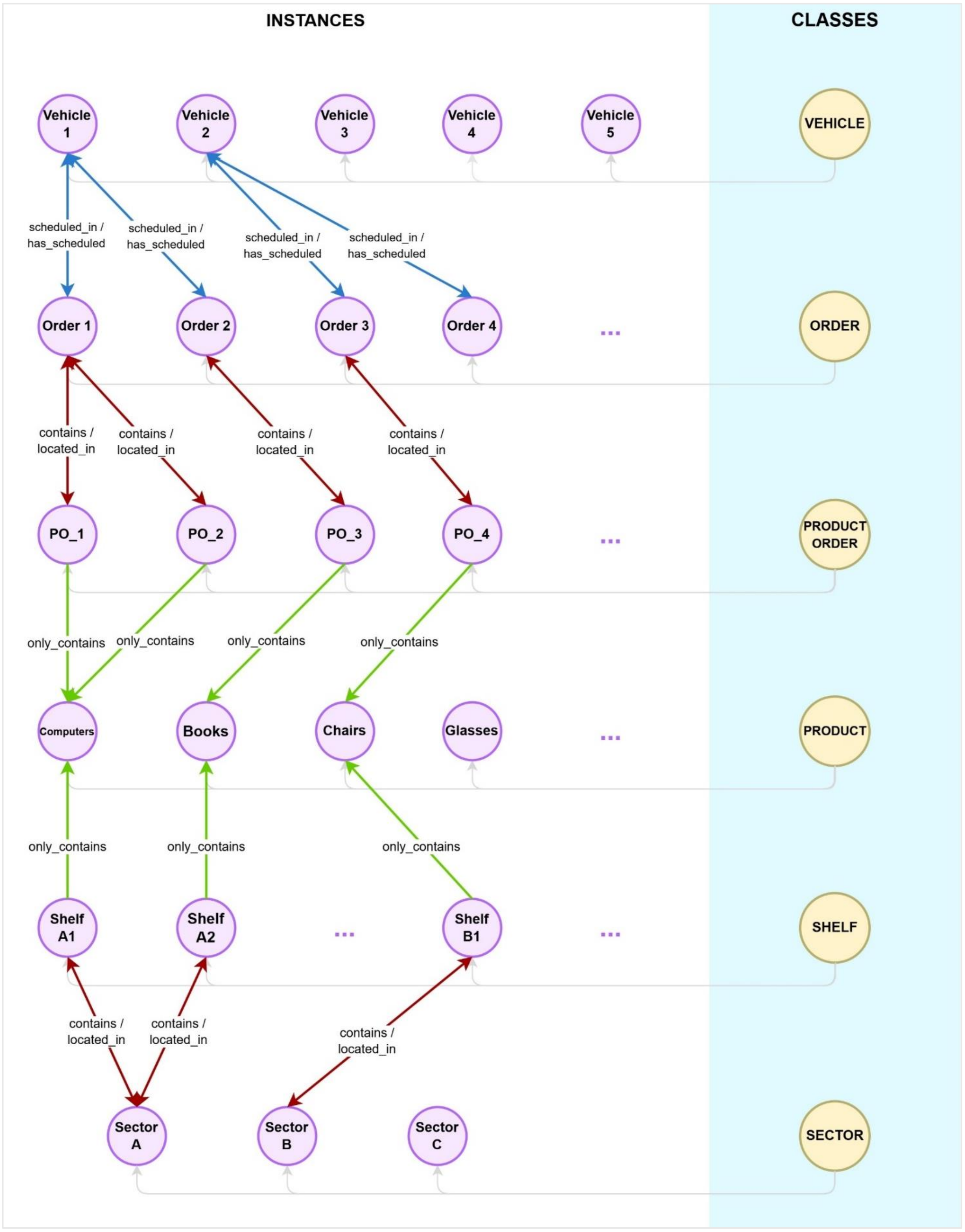


Fig. 3 – Visual representation of the ontology structure with classes, instances and object properties.

It is important to specify that we have used an additional class called Product_order, where each instance is linked to a specific order and is characterized by a unique product type.

In this way, an order does not contain different products directly but rather different Product_orders, each uniquely referring to a specific product (in a certain quantity). An alternative to this structural choice would be associating different product types with each order, specifying a quantity for each (e.g., computers_quantity, books_quantity, etc.). However, this approach seemed inefficient.

2.2 – Instances schema

Each modelled class contains several instances with a specific role and structure in terms of data properties and object properties. Below a detailed explanation:

Vehicle

There are ten vehicles on which orders can be loaded or simply scheduled for delivery. Each vehicle has a current number of orders, and a maximum capacity based on its load capacity. Note that the current number includes both scheduled orders and those already loaded onto the vehicle.

The scheduled orders for a particular vehicle are those that have been assigned but not yet loaded by an operator. The scheduling algorithm is external and therefore beyond the scope of our responsibility.

Object properties
has_scheduled
contains

Data properties
quantity
max_quantity

Order

To test the scalability of the system, several dozen orders were inserted. These orders could either be scheduled or already loaded onto a vehicle and contain a certain number of Product_orders.

Additionally, they have properties related to the buyer's personal information, the shipping address, and the expected delivery date; that information will not be available for warehouse operator users.

Object properties
scheduled_in
located_in
contains

Data properties
buyer_data
address
delivery_date

Note that an order maintains the scheduled_in relationship even if it has already been loaded onto the vehicle.

Product_order

Each instance of this class will simply contain the unique product type, a relationship with the order that contains it, and the corresponding quantity of the product.

Object properties
located_in
only_contains

Data properties
quantity

Product

They are not treated as actual products, but as product types, and are contained in product orders and shelves; however, we chose not to include "outgoing relationships" for them.

Shelf

Each instance of this class contains references to the location within the warehouse (sector) and the unique product type it holds. Additionally, it includes the current quantity and the maximum quantity.

Object properties
located_in
only_contains

Data properties
quantity
max_quantity

Sector

There are three sectors, each containing about ten shelves.

Object properties
contains

SECTION 3 – Logic behind Smart Warehouse features

As previously mentioned, the primary goal of our work is to enable users (both warehouse operators and administrators) to ask the LLM questions to obtain information about the warehouse status or make modifications to the knowledge.

Among the first type of queries, where no special function management was required, we included:

- Requesting information about a specific product (location in the warehouse, available quantity, etc.) – [accessible to warehouse operators and administrators].
- Viewing detailed information for each order (delivery address, estimated delivery date, buyer information, etc.) – [administrator only].
- Viewing detailed information about the warehouse (vehicles, sectors, shelves) and each sector/shelf – [accessible to warehouse operators and administrators].

For these queries, responses were documented in Section 4.

Below is an explanation of the logic behind the three functions we implemented to modify the knowledge graph: move_products, load_order, and reschedule_order.

3.1 – Move product function

Function that allows an operator or an administrator to move a certain quantity of products of the same type from a source shelf to a target shelf; therefore, among the input parameters, we have the quantity of products, the source shelf, and the target shelf (the product type can be implied).
This can be useful when an operator has performed a transfer and wants to update the warehouse status in the system as well.

We expect the system to work correctly if it can call the move_product function and pass the correct parameters: in this case, quantity, shelf_src, and shelf_tar.
The query that will trigger the function will look like the following:

- “Move 4 products from Shelf_A3 to Shelf_B6”.
- “Please move 6 computers from Shelf_B11 to Shelf_C1”.

Below is an illustration that helps visualize the function's logic:

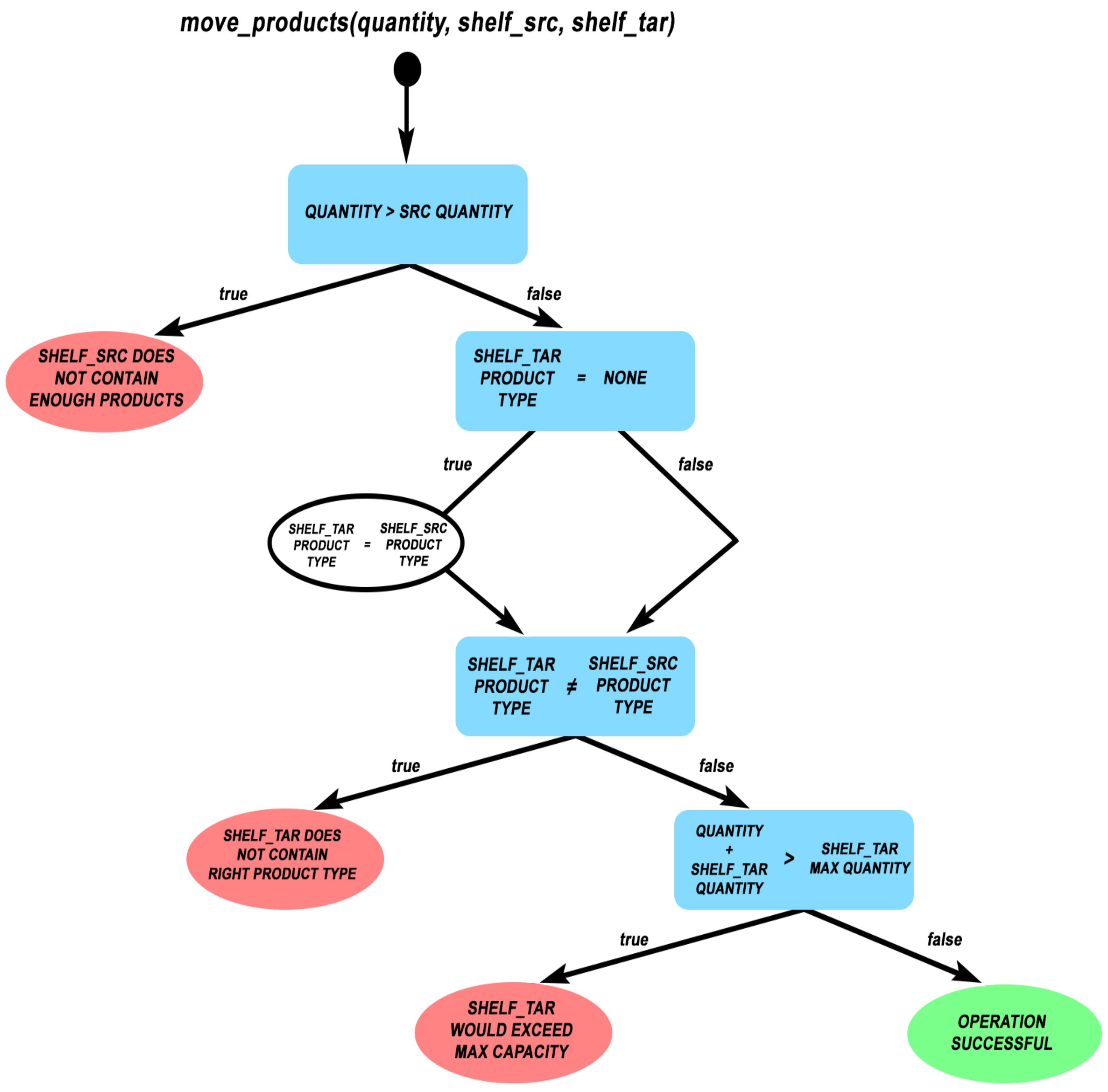

move_products(quantity, shelf_src, shelf_tar)
QUANTITY > SRC QUANTITY
true
false
SHELF_SRC DOES NOT CONTAIN ENOUGH PRODUCTS
SHELF_TAR PRODUCT TYPE = NONE
true
false
SHELF_TAR PRODUCT TYPE = SHELF_SRC PRODUCT TYPE
SHELF_TAR PRODUCT TYPE ≠ SHELF_SRC PRODUCT TYPE
true
false
SHELF_TAR DOES NOT CONTAIN RIGHT PRODUCT TYPE
QUANTITY + SHELF_TAR QUANTITY > SHELF_TAR MAX QUANTITY
true
false
SHELF_TAR WOULD EXCEED MAX CAPACITY
OPERATION SUCCESSFUL

Fig. 4 – move_ products logic visualization.

Given the three parameters and assuming they have been correctly passed to the function, we encounter various scenarios based on the structure of the ontology. Therefore, the operational logic can be represented with a tree-like structure where nodes (in blue) correspond to conditions (if statements) and leaves determine either the success (in green) or failure (in red) of the function.

First, the system checks whether the source shelf has enough products to perform the transfer. If this condition is met, it verifies if the target shelf already has an associated product type (If the property is empty, it will not have one, and we can set it to match the source shelf's product type): In either case, the process can continue.

Next, it checks that the target shelf does not contain a different product type from the source shelf. Finally, if all conditions are satisfied, a final check ensures that the target shelf does not exceed its maximum capacity.

Once the "operation successful" status is obtained, the ontology is updated by modifying the product quantities on each shelf.

3.2 – Load order function

Function that allows an operator or an admin to modify the knowledge graph by stating that an order has been loaded onto the vehicle it was scheduled for. Therefore, the input parameters include a list of orders to be loaded and a list of shelves from which to retrieve the products to fulfil the orders.

We expect the system to work correctly if it can call the load_order function and pass the correct parameters: in this case, all the various orders and shelves.

The query that will trigger the function will have a format like the following:

- "Load Order_5 and Order_13 taking products from Shelf_A1, Shelf_B9, Shelf_B10, and Shelf_C5"

Below is an illustration that helps visualize the logic of the function:

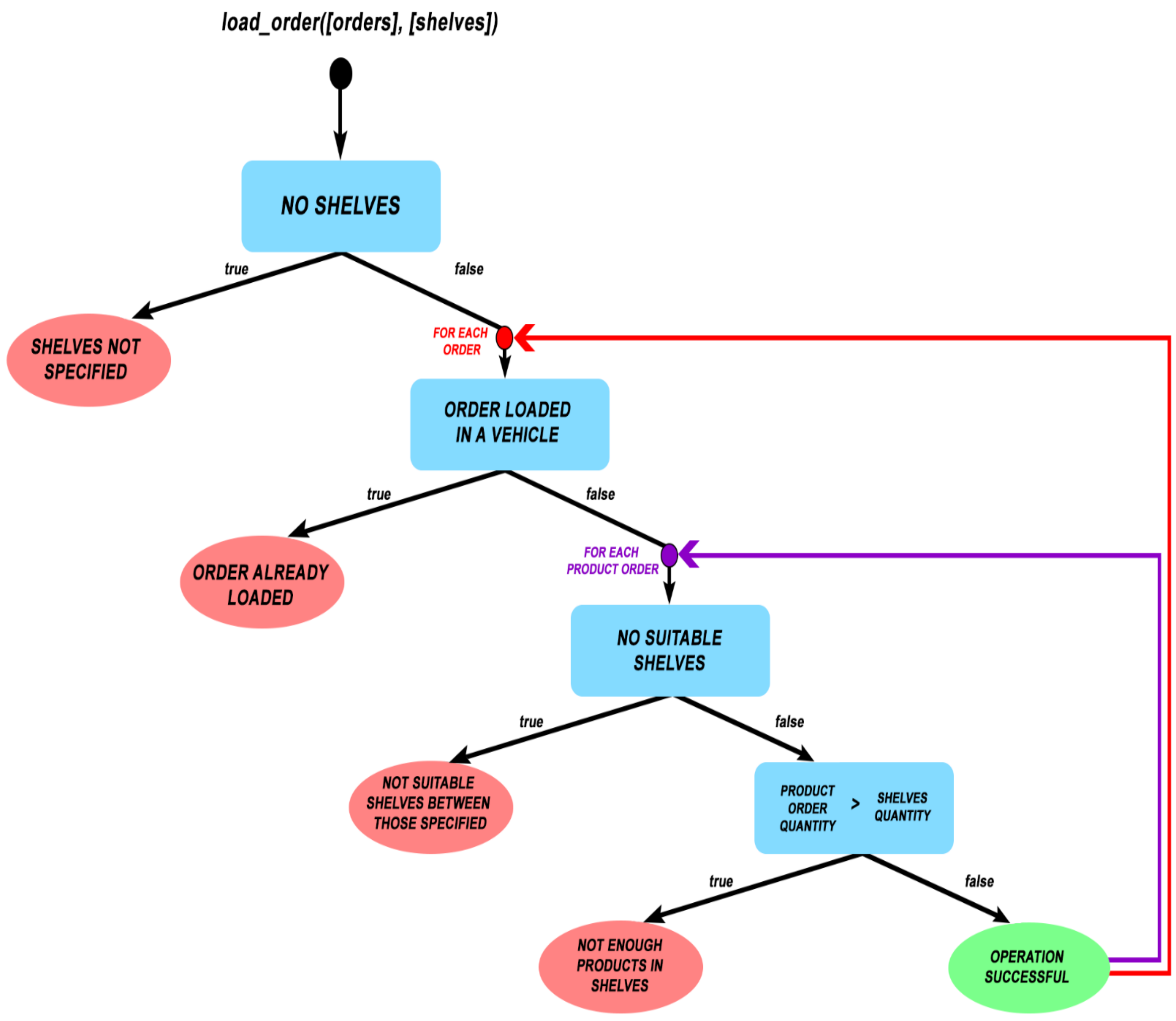

load_order([orders], [shelves])
NO SHELVES
true
false
SHELVES NOT SPECIFIED
FOR EACH ORDER
ORDER LOADED IN A VEHICLE
true
false
ORDER ALREADY LOADED
FOR EACH PRODUCT ORDER
NO SUITABLE SHELVES
true
false
NOT SUITABLE SHELVES BETWEEN THOSE SPECIFIED
PRODUCT ORDER QUANTITY > SHELVES QUANTITY
true
false
NOT ENOUGH PRODUCTS IN SHELVES
OPERATION SUCCESSFUL

Fig. 5 – load_order logic visualization.

Here too, we can represent the logic with a tree-like structure but including loops that iterate over each order and nested within them, over each product order.

First, it is verified that shelves have been specified; if not, the operation fails. Then, for each order, it is checked whether it has already been loaded, and for each product order, it is verified that there are suitable shelves among those specified (in terms of product type) to fulfil the request.

If this condition is met, a final check is performed to assess the availability of products in terms of quantity on the suitable shelves.

If all the conditions are met, the shelves are emptied by iterating over each product order, and then the order and vehicle information is updated.

3.3 – Reschedule order function

Function that allows only administrators to reschedule an order overriding the scheduling software's decision by assigning an order to a different vehicle than the one initially assigned. Therefore, the input parameters include a list of orders to be rescheduled and a single vehicle.

We expect the system to work correctly if it can call the reschedule_order function and pass the correct parameters: in this case, all the various orders and the vehicle.

The query that will trigger the function will have a format like the following:

- "Reschedule Order_7 and Order_19 in Vehicle_7"

Below is an illustration that helps visualize the logic of the function:

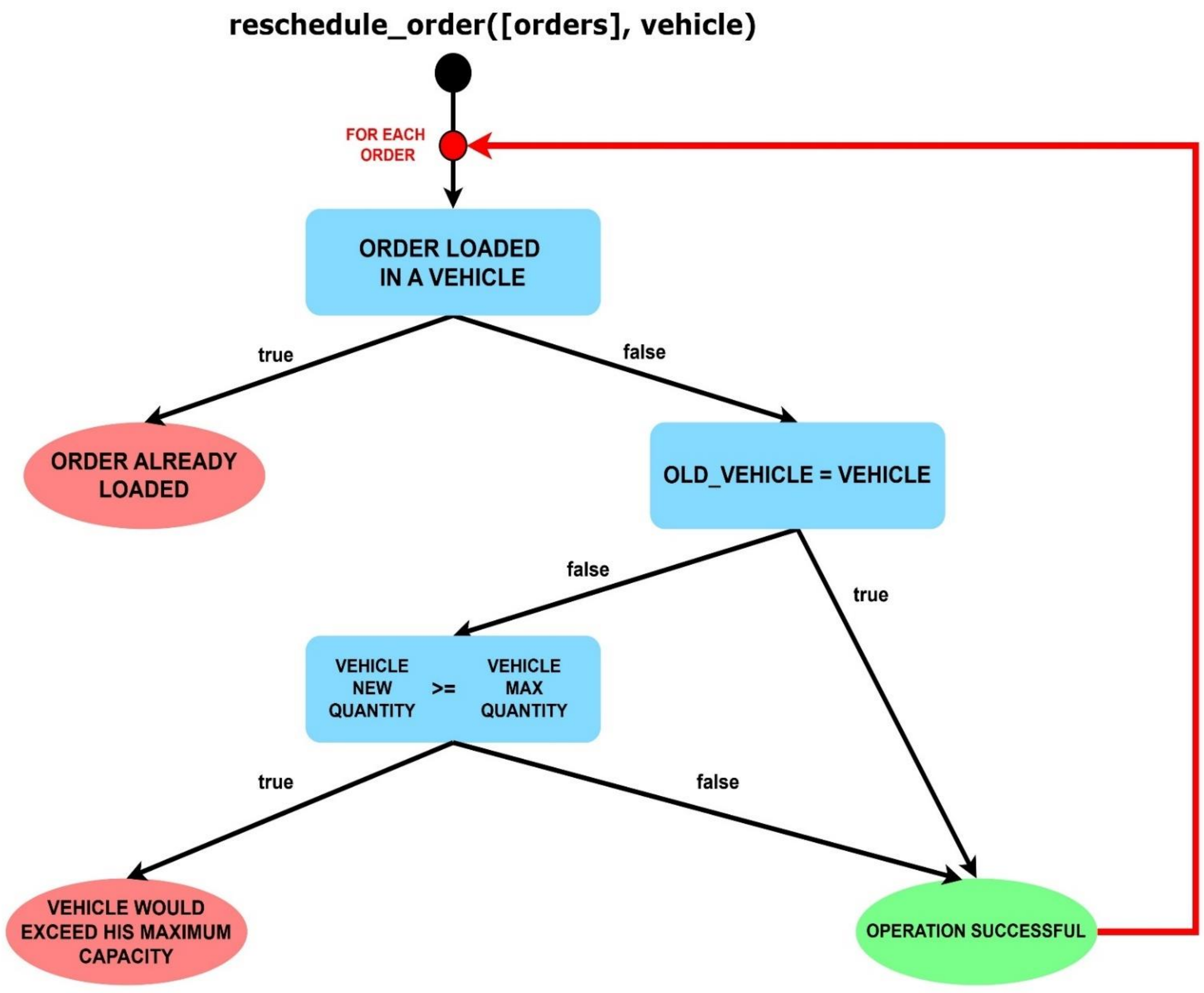

reschedule_order([orders], vehicle)
FOR EACH ORDER
ORDER LOADED IN A VEHICLE
true
false
ORDER ALREADY LOADED
OLD_VEHICLE = VEHICLE
false
true
VEHICLE NEW QUANTITY >= VEHICLE MAX QUANTITY
true
false
VEHICLE WOULD EXCEED HIS MAXIMUM CAPACITY
OPERATION SUCCESSFUL

Fig. 6 – reschedule_order logic visualization.

Here too, the tree structure includes a loop because it is necessary to iterate over the orders. Specifically, for each order, it must be checked that it has not already been loaded onto a vehicle.

Then, if the new vehicle matches the old one, the function can immediately conclude successfully as it is idempotent. However, if the vehicles differ, it is first necessary to check that the new vehicle does not exceed its maximum capacity after the rescheduling operation.

Finally, if the operation is successful, the knowledge graph must be updated by adding the scheduling property to the order and the new vehicle (while removing it from the old vehicle).

3.4 – Knowledge filter function

This last implemented function does not manage changes to the knowledge graph; rather, it allows masking part of the knowledge from unauthorized users: in our case, the warehouse operators.

For the scenario we have modelled, it is unnecessary (and incorrect) to allow the warehouse operator to view private information about orders such as buyer_data, address, and delivery_date. This case has been managed by using this function, passed as a parameter to the query function in the knowledge_mapper field in our test scenario.

This field allows, by passing a filtering function, the execution of the desired masking operation.

Note that, in our system, the login as an operator or administrator is done directly from the terminal when starting the test using one of the two commands:

- *python test/warehouse.py administrator*
- *python test/warehouse.py operator*

SECTION 4 – Results evaluation

Intro

In this final section of our project, we present an evaluation table designed to test the behavior of the agent when handling the queries provided. The table is color-coded to reflect specific groupings: adjacent queries with the same color means correlated queries executed in the same scenario (e.g., query 30 serves as confirmation for query 29).

Table description

The table was created using Excel and is structured with the following columns:

1. **Prompt ID**: A unique identifier for each query executed. A total of 53 formalized queries were tested, though additional queries were used during development to identify and address bugs.

2. **Mode**: Specifies the operational mode of the agent. According to the instruction.txt file, we have defined two roles: *warehouse operator* and *warehouse administrator*. These roles differ in terms of available functions and knowledge.

   - The administrator has full access to the scenario's knowledge and actions.
   - The operator is limited to a subset of knowledge and actions.

3. **Expected Function**: The function anticipated to be called by the agent, basically the ground truth (labels of the dataset)

4. **Expected Parameters**: The parameters expected to be retrieved from the knowledge base, before answering, formatted as a JSON object. (labels of the dataset)

5. **Expected Output**: A ground truth reference for the expected function's outcome (labels of the dataset). The outputs are generally categorized as:

- Successful operation.
- Failed operation - reason of the failure.
- A list of items post-operation (to validate correctness) or the result of a static query that doesn't change the warehouse's scenario.

6. **Predicted Function**: The function called by the agent, formatted like column three.

7. **Predicted Parameters**: The parameters retrieved by the agent, formatted like column four.

8. **LLM Response**: The response from the agent, extracted directly from the terminal

9. **Function Score**: A binary score (0 or 1) that evaluates the alignment between the expected and the predicted functions. A score of 1 indicates the correct function called; otherwise, the score is 0.

10. **Parameters Score**: A metric quantifying the accuracy of the retrieved parameters, calculated as:

$$\text{Score} = \frac{(p\_1 + p\_2 \; + p\_N)\}}{N}$$

where: $p\_1, \ldots\, p\_N$ are scores for each parameter (ranging between 0.0 and 1.0):

For single-string parameters

$$p\_i \;=\; 1.0 \, if \; correct, otherwise \;\; p\_i \;=\; 0.0$$

For list parameters:

$$p\_i = \frac{correct}{correct + missing + incorrect}$$

This score, ranging from 0.0 to 1.0, was assigned without a formalized rule set since it is not a central focus for our project.

11. **Semantic Score**: A manual evaluation of the semantic alignment between the expected response and the actual response. This score, ranging from 0.0 to 1.0, was assigned without a formalized rule set since it is not a central focus of this project.

- A score of 0.5 was assigned if part of the response was correct but included hallucinated information.
- If the response was incomplete, a deduction of 0.n points was applied.
- Additional deductions were applied if the action was correct, but the justification provided by the model was inaccurate.
- An additional column was included to explain the rationale behind each semantic score.

Analysis of Results

Overall, the agent performed well, but there were significant differences between queries that involved static knowledge interrogation and those requiring dynamic function execution.

What we can observe is that the parameters score and function score have obtained most of the times the maximum score, to test this behavior we have produced synonymous queries to the description of the function trying to fool the agent, but the agent handles pretty well all of this cases; moreover when the right function is called the parameters are passed well too from the knowledge to the agent.

The same cannot be said for the static interrogation of the knowledge which sometimes led to hallucination of the agent and/or missing details in the response.

In some cases, while the query was partially correct, key details were missing from the output.

In conclusion, the agent performed reliably when tasked with dynamic function execution, achieving high scores for both function and parameter accuracy.

However, static knowledge queries revealed weaknesses, particularly in handling ambiguous prompts and avoiding hallucination.

CONCLUSIONS AND FUTURE DEVELOPMENTS

In conclusion, the following future developments are proposed to enhance the evaluation framework and improve the model’s applicability to real-world scenarios:

1. **Expanding and Refining the Ontology**:

   Increasing the size and detail of the ontology to better represent a real-world warehouse scenario, allowing the agent to be tested in a more realistic, comprehensive and detailed rich environment.

2. **Leveraging More Parameter-Rich Models**:

   Utilizing an LLM with a larger number of parameters to ensure higher accuracy and reliability in its responses. A more advanced model could improve performance, particularly in scenarios requiring understanding and reasoning.

3. **Testing on a Broader Range of Queries**:

   Expanding the number and variety of queries. Incorporating more complex logic and parameters into the queries would help assess the model’s capability to handle intricate and realistic situations effectively.

4. **Automating Evaluation Metrics**:

   Developing an automated method for scoring both the syntactic and semantic correctness of the model’s responses. This would enhance scalability in building the dataset and in the testing process, enabling more efficient and objective evaluation of the model’s performance.

By implementing these advancements, the project aims to improve both the robustness and scalability of the testing framework, while moving closer to replicating real-world operational conditions for the model.

BIBLIOGRAPHY

[1] Llama team (Meta), “The Llama 3 Herd of Models”, paper

[2] Neo4j, “What is Retrieval-Augumented Generation (RAG)?”, blog

[3] Cowl documentation