



Politecnico
di Bari



LLM-based mapping of natural language statements to knowledge graph updates

Project Report in Formal Languages and Compilers

MATTIA TRITTO
ROSSELLA TRITTO

DEPARTMENT OF ELECTRICAL AND INFORMATION ENGINEERING

POLYTECHNIC OF BARI

Master's Degree Course in Computer Science Engineering (AI & Data Science)

Contents

1	Introduction	2
2	Code Architecture Overview	3
2.1	Utils	3
2.2	Knowledge Graph	4
3	New Features	5
3.1	Update Individuals	5
3.2	Add Individuals	6
3.2.1	add Function	6
3.2.1.1	get_superclasses Function	7
3.3	Remove Individuals	7
3.4	Remove Assertions	7
4	System Evaluation	9
4.1	How the framework was evaluated	9
4.2	Explanation of the automatic testing code	10
5	Result Analysis	12
5.1	Home Configuration	12
5.2	Table of Test Cases	13
5.3	Overall performances	16
5.3.1	Group A, retrieve information about specific devices	17
5.3.2	Group B, retrieve information about a room	18
5.3.3	Group C, retrieve information about the entire house	18
5.3.4	Group D, queries for setting specific devices	18
5.3.5	Group E, queries for setting group of devices	19
5.3.6	Group F, queries for modifying the configuration	19
5.3.7	Group G, advanced queries conditioned by sensor data	20
6	Conclusions and Future Work	21
	Bibliography	22

1

Introduction

This project aims to develop CRUD functions to interact with the knowledge graph and use appropriate prompts to guide an open source LLM to transform specific natural language statements into a sequence of Web Ontology Language (OWL) knowledge graph updates. The adopted strategy is to register a custom function that the agent can invoke for each functionalities. Thanks to this, we can obtain effectively interactions between LLM and the knowledge graph (KG). To achieve this goal, we use ollama, an open-source platform that facilitates LLM usage, and Python for interfacing with LLM and managing knowledge graph operations. The functionalities implemented and evaluated in this report include adding new individuals to the knowledge graph, removing individuals or property assertions, and updating existing individuals. To do this, we have employed a zero-shot strategy so that the code can be usable for future purposes beyond just the context of domotics.

The report is structured as follows:

1. **Introduction;**
2. **Code Architecture Overview:** provides an overview of the system's architecture;
3. **New Features:** describes newly added functionalities;
4. **System Evaluation:** explains the approach used to evaluate system performance;
5. **Results Analysis:** presents and analyzes the results of the evaluation;
6. **Conclusion and Future Work.**

In the next sections, we will provide a detailed view of all modifications implemented in the code and an accurate explanation of our chosen methodology for evaluating the agent's performance with very promising results in the testing phase.

2

Code Architecture Overview

In this chapter, we outline the structure of the project’s code, highlighting the main directories: `data`, `test`, and `src`.

Folder `data`

The `data` folder contains the necessary file to configure the home environment, namely `home.owl`. This file is formatted in OWL (Web Ontology Language) and represents a description of the elements in the house, their properties, divided into object and data properties, and the relationships between different classes. The configuration of the home used for testing is provided in 5.1.

Folder `test`

The `test` folder includes the main file `home.py` for executing and interacting with the agent. It provides all the functionalities to load the home configuration, register functions, and respond to user queries. Specifically, it defines a `SmartHome` class that represents the assistant and provides an interface to interact with the knowledge graph.

Folder `src`

The `src` folder contains the main source code of the project. Within this folder, there are 10 files, among which two are particularly important for the newly implemented functionalities:

- `_kg.py`: Handles the management of the Knowledge Graph. It provides functionalities to create, update, query, and manipulate the agent’s knowledge base;
- `_utils.py`: Contains general utility functions such as string transformation between different formats.

The other files in the `src` folder contain various system components, each with specific responsibilities. However, `_kg.py` and `_utils.py` are the files we modified for implementing the new functionalities.

2.1 Utils

In `_utils.py`, we added two new methods:

- `space_to_snake`: Transform a string containing spaces by replacing them with underscores;
- `label_to_iri`: Converts an input string into an IRI. We insert `:` to the

beginning of the string, apply the previously defined `space_to_snake` method, and capitalize the first letter.

For example: "*Living room*" is transformed in "*: Living_room*".

These methods manage the creation of new individuals, ensuring their names are coherent with the ontology's required format.

2.2 Knowledge Graph

We used several methods of `_kg.py` to handle the addition, removal, and update of individuals or assertions within the ontology:

- `add_axioms`: Add axioms to the ontology via an asynchronous POST request. It manages both ontology and associated index updates;
- `axioms_for_entity`: Retrieve axioms for a specific entity. The entity can be represented as either a string or an `Entity` object. It uses an HTTP GET request to obtain axioms, which are returned as a dictionary filtered by entity type and IRI;
- `add_individual`: Add a new individual to the ontology with specified classes. First, an `Individual` object is created using the provided IRI, followed by the creation of the list of assertions to be added. These assertions are added to the ontology using `add_axioms`;
- `add_functional_property_assertions`: This method adds multiple functional properties assertions to the ontology. It first transforms the input into a collection and then adds the assertions to the ontology using the `add_axioms` method;
- `remove_entity` / `remove_property_assertions`: `remove_entity` removes an entity from the ontology, while `remove_property_assertions` removes property assertions from the ontology. Both perform an HTTP DELETE request to the server. They are similar to `axioms_for_entity` but aim to remove data rather than retrieve it.

3

New Features

In this chapter, we describe some functionalities introduced to manage operations on the Knowledge Graph. These operations include adding new individuals, updating existing ones, removing individuals, and removing specific assertions within the graph. The adopted strategy is to implement the KG functionality as an additional function available to the agent via `llm.register_function`. Using the register function, we can register a custom function that the agent can invoke. The **description** field is crucial as it describes the function's operation. This description helps the agent understand the purpose of the function and when to use it. We used a zero-shot strategy to generalize as much as possible, avoiding restricting ourselves only to the smart home context. Thanks to this function, we can return actions in JSON format that can be converted into operations to the KG.

3.1 Update Individuals

The `update_individuals` function enables transforming specific user queries into updates to the knowledge graph.

The function parameters of the `register_function` and the description field are:

- **subjects:** A list of subjects to update;
- **predicate:** The predicate to update;
- **object:** The new object of the axiom that will be updated for the specified subjects.

```
description =  
    """  
    Updates an array of subjects within the knowledge graph.  
    This function allows users to  
    modify the graph based on specific queries.  
    """
```

The `update_individuals` function allows modifying one or more individuals. The main steps are:

1. **Subject check:** First, we verify that the subject exists in the KG using `axioms_for_entity` to retrieve a set of axioms related to a specific entity;
2. **Predicate check:** Next, we check that the axiom containing the subject has the predicate specified by the query. If the predicate is present, we add the individual to the `subjects_to_update` list;
3. **Manage the subject list:** Creating this list allows efficient management of subjects, so if a check fails for a particular subject, the method does not stop

but continues with the other subjects in the list;

4. **Update subjects:** Finally, we update all subjects that passed all checks using `add_functional_property_assertion` in `_kg.py`.
5. **Result:** The returned response varies depending on the number of modified subjects. If all subjects were successfully updated (meaning the length of the modified subjects list is the same as the initial subjects), the function returns a success message; otherwise, it indicates which subjects were not updated.

3.2 Add Individuals

The `add_individuals` function enables the addition of new individuals to the knowledge graph. Regarding the `register_function`, the parameters for registration are:

- `individual_name`: The name of the new individual to be added to the KG;
- `class_name`: The class to which the new individual belongs.

```
description =
    """
    Adds a new individual to the knowledge graph by specifying
    the class it belongs to within the graph.
    """
```

The steps of the function are:

1. **Transform Name into IRI:** We transform both the individual name and the class name into their corresponding IRIs using `label_to_iri` (2.1);
2. **Check Class:** We use the `entity` function to verify that the class exists in the KG. The `entity` function retrieves an entity from the KG using an IRI or a label;
3. **Manage Classes with a Final 's':** After some tests, we noticed that the agent sometimes considered the class name with a final 's'. Therefore, if the class ends with an 's', we remove the 's' and recheck the class's existence. If it exists, we proceed with the addition; otherwise, we return an error.

3.2.1 add Function

The `add` function performs the actual addition of the individual to the KG. It includes:

1. **Retrieve Superclasses:** Recursively retrieves all superclasses of the specified class using `get_superclasses`;
2. **Assign Default Properties:** Set default properties for various classes using a predefined dictionary;
3. **Add Individual:**
 - Add the individual to the KG, associating it with all classes obtained in the `superclasses` list;
 - Assigns default properties to the individual with `add_functional_property_assertions` (2.2).

3.2.1.1 `get_superclasses` Function

The `get_superclasses` function in `_kg.py`, given a class, returns a list of strings containing all superclasses. Specifically, as long as there are assertions for the current class:

1. It retrieves the assertions for the current class (`axioms_for_entity`);
2. It retrieves the superclass from the assertion using the key "subclass_of";
3. It updates the current class name with the name of the new found superclass;
4. It adds the superclass to the `class_list`.

3.3 Remove Individuals

The `remove_individuals` function enables the removal of existing individuals from the knowledge graph. Using `register_function`, the parameter for registration is:

- `individual_names`: A list of names of the individuals to be removed.

```
description =
    """
    Removes an array of individuals from the knowledge graph.
    Forget the name of the individual specified.
    """
```

In the field `description` we use the keyword "forget" to enables the correct management of queries such as "Forget about lamp 1".

The steps followed by the function are:

1. **Check Individuals:**
 - For each individual in the `individual_names` list, we verify their existence in the KG using the `entity` function;
 - If the individual exists, we add it to the `individuals_to_remove` list. This strategy is similar to the one used in the `add_individuals` function, where we proceed with the removal of existing individuals without stopping the entire process if one of the individuals does not exist;
2. **Remove Individuals:**
 - We use the `remove_entity` (2.2) function to delete the entity and all axioms connected to it;
3. **Result:**
 - The function returns a success message if all individuals in the list have been removed;
 - If some individuals cannot be removed, the function returns the list of individuals that have not been deleted.

3.4 Remove Assertions

The `remove_assertions` function allows the removal of specific axioms related to existing individuals. The parameters for `register_function` are:

- **Subject:** The subject of the assertion to be removed;
- **Predicate:** The predicate of the assertion to be removed.

```
description =  
    """  
    Removes an assertion from the knowledge graph.  
    Forget the relationship between subject and object.  
    """
```

The function is implemented as follows:

1. **Subject Check:** Uses `axioms_for_entity` (2.2) to retrieve axioms associated with the specified subject;
2. **Predicate Check:** Verifies if the axioms related to the subject contains the specified predicate. We use `kg.entity(predicate).iri` to obtain the IRI of the predicate. If the predicate is not present in the axioms, a warning is returned indicating that the predicate does not exist for this subject;
3. **Remove Assertion:** If both subject and predicate are valid and present in the KG, the function proceeds to remove the assertion using `remove_property_assertions` from the KG (2.2).
4. **Result:** After successfully removing the assertion, the function returns a message confirming that the operation has been completed successfully.

4

System Evaluation

4.1 How the framework was evaluated

To evaluate the framework performances, the plan was to test it with 100 different queries to observe its responses and verify how well the natural language sentences were translated into correct function calls.

Each query has been evaluated based on the following criteria:

- **Function call:** 1 if the function name is correct, -1 if it is not;
- **Function parameters:** 1 if the parameters are correct, 0 if they are correct but some are missing, -1 if they are incorrect;
- **Chat output:** 1 if it is correct, 0 if it is correct but some information is missing, -1 if it is incorrect.

The dataset containing queries was divided into these 7 groups:

- **Group A)** 15 queries for retrieving information about specific devices (e.g., *"Is lamp 1 on?"*, *"What's the setting of air conditioner 1?"*, *"What's the temperature in the living room?"*, *"Is someone in the kitchen?"*);
- **Group B)** 15 queries for retrieving information about a room (e.g., *"Is the light on in the kitchen?"*, *"List all lights in the living room"*);
- **Group C)** 15 queries for retrieving information about the entire house (e.g., *"How many rooms are there?"*, *"List all devices"*);
- **Group D)** 15 queries for setting specific devices (e.g., *"Turn on the robot vacuum"*, *"Set air conditioner 1 to 20 degrees"*);
- **Group E)** 15 queries for setting groups of devices (e.g., *"Turn off everything in the bathroom"*, *"Turn on all lights"*);
- **Group F)** 10 queries for modifying the configuration (e.g., *"Move lamp 1 to the living room"*, *"Forget where lamp 2 is located"*, *"Add a new room named ..."*);
- **Group G)** 15 advanced queries conditioned by sensor data (e.g., *"Set air conditioner 1 to 5 degrees lower than the current room temperature"*, *"Turn off the light in all rooms with no people"*).

For groups A, B, and C the LLM must always call the **respond** function for all the queries. It is crucial that the LLM chat output is sensible and provides correct information to the user.

For groups D, E, and F the LLM must call the correct functions with the correct

parameters. Additionally, the LLM chat output must be coherent and accurate.

Group G is more of an experiment to evaluate the capacity of the LLM. Given the complexity of these queries, it is challenging for a small LLM running on a PC to fully understand intricate sentences. This group helps to explore the boundaries and limitations of the framework.

After automatically posing these queries to the system and manually evaluating the correctness of the function calls, parameters, and chat output, the metrics used to evaluate the system are the following:

- **Function call accuracy;**
- **Parameter accuracy** (fully correct only, and including partially correct);
- **Chat output accuracy** (fully correct only, and including partially correct).

4.2 Explanation of the automatic testing code

The testing has been designed to be:

- **Reproducible:** each time a query is asked, the ontology is restored to its original state, which is particularly useful when testing queries that changes the configuration (in order to do that, we call `home.configure()` before a new query is asked);
- **Automatic:** this is crucial because we don't have GPUs on our personal laptops, so each inference took about 4 minutes to produce the output. It would be very annoying to wait every 4 minutes to execute the next query;
- **Automatically stored:** the results are saved automatically in a CSV file, so they can be analyzed later.

In the `test` folder, we can store all the queries in a file named `queries.csv`. This CSV file should contain only one column, where each query is separated by the new line character. For running the automated test, it is necessary to run the `automated_testing.py` file. All the chat outputs are stored in the CSV file named `chat_outputs.csv`, which consists of two columns:

- The first column contains the queries asked;
- The second column contains all the chat outputs generated by the LLM.

To retrieve the standard output, we have redirect the standard output in this way:

```
with io.StringIO() as buf, redirect_stdout(buf):
```

- `io.StringIO()` creates an in-memory buffer (`buf`) that behaves like a file object but stores its data in a string buffer;
- `redirect_stdout(buf)` redirects the standard output (`stdout`) to `buf`. This means any output that would normally be printed to the console will be captured into `buf` instead.

So after calling the asynchronous function `print_response(home, query)`, the `chat_output` is retrieved using `buf`:

```
chat_output = buf.getvalue().strip()
```

- `buf.getvalue()` retrieves the contents of the `StringIO` buffer `buf`. Since `redirect_stdout()` redirected `print()` output to `buf`, `buf` now contains whatever was printed by `print_response()`;
- `strip()` removes any leading and trailing white space characters from the captured output.

```
writer = csv.writer(outfile)
writer.writerow(['Query', 'Chat output'])
writer.writerow([query, chat_output])
```

Then, we write the query and the chat output as a new row in the file `chat_outputs.csv`. To automate the retrieval of function calls and their parameters, we've done a workaround due to the current lack of API support. After running `automated_testing.py`, we analyzed the `owllama.log` file using CLI tools to automatically extract the JSON data related to the function calls.

In order to do that, it is necessary to open a terminal (valid for MacOS and Linux users only), move to the root directory of the `owllama` project and run the following command:

```
awk '/====\[ Start response \]====/, /====\[ End response
    \]====/{if (!/====.*====/) print}'
owllama.log > test/functions_called.json
```

`awk` is a command-line utility for text processing. In this case, it matches lines between `====[Start response]====` and `====[End response]====` in the `owllama.log` file.

The command `{if (!/====.*====/) print}` filters and prints lines that do not match `====.*====`, specifically extracting JSON function calls, and then redirects this output to the `functions_called.json` file in the test directory.

5

Result Analysis

5.1 Home Configuration

The home configuration used for testing includes 20 devices distributed across 5 rooms: bedroom, bathroom, study, kitchen, and living room. The devices are divided into two main types: toggleable devices and sensors. Each device has the object property `located_in` indicating its room, and the `state` property indicating whether it is on or off.

Toggleable Devices

Toggleable devices are further divided into Lights and Appliances.

Lights

- **Lights:** Can be turned on or off.
- **Dimmable Lights:** Have an additional data property `setting` that indicates the current light level.

Appliances

- Some appliances may have the property `setting` and `unit` depending on the specific device.

Sensors

Sensors, in addition to `located_in` and `state` properties, provide information through the `reading` property. The types of sensors include:

- **Temperature Sensors:** Provide the temperature of the room they are located in.
- **Occupancy Sensors:** Indicate if there is a person in the room (Boolean property).
- **Smoke Sensors:** Indicate the presence or not of smoke (Boolean property).
- **Humidity Sensors:** Provide the humidity percentage of the room.








Togglable Device		Class	Located In	State	Setting	Unit
	Lamp 1	Light	Bedroom	Off	20	-
	Lamp 2	Light	Study	On	50	-
	Lamp 3	Light	Living Room	Off	-	-
	Lamp 4	Light	Kitchen	On	-	-
	Lamp 5	Light	Bathroom	Off	-	-
	Air Conditioner 1	Appliance	Living Room	On	20	°C
	Air Conditioner 2	Appliance	Bedroom	Off	16	°C
	Robot Vacuum 1	Appliance	Bathroom	Off	-	-
	Oven 1	Appliance	Kitchen	On	180	°C
	Coffee Machine 1	Appliance	Kitchen	Off	-	-
	Television 1	Appliance	Living Room	On	-	-
	Washing Machine 1	Appliance	Bathroom	On	-	-

Figure 5.1: List of Togglable Devices with their properties.


Sensors		Located In	State	Reading	Unit
	Temperature Sensor 1	Bedroom	On	20	°C
	Occupancy Sensor 1	Living Room	Off	-	-
	Occupancy Sensor 2	Kitchen	Off	-	-
	Occupancy Sensor 3	Bedroom	On	True	-
	Occupancy Sensor 4	Study	On	True	-
	Occupancy Sensor 5	Bathroom	Off	-	-
	Smoke Sensor 1	Kitchen	On	True	-
	Humidity Sensor 1	Bathroom	On	20	-

Figure 5.2: List of Sensors with their properties.

5.2 Table of Test Cases

The following table provides a complete overview of the system's performance. For further details, please refer to the attached `test.pdf` which contains comprehensive information about the testing, including our comments on each executed query.

Table 5.1: Accuracy Table*Legend:*

1 = Totally correct,

0 = Partially correct

-1 = Incorrect

ID	Invoked Function Correct	Arguments Correct	Chat Output Correct
A1	1	1	1
A2	1	1	1
A3	1	1	1
A4	1	1	1
A5	1	1	1
A6	1	1	1
A7	1	1	1
A8	1	1	1
A9	1	1	1
A10	1	1	1
A11	1	1	1
A12	1	1	1
A13	1	1	1
A14	1	1	1
A15	1	1	1
B16	1	1	1
B17	1	1	-1
B18	1	1	1
B19	1	1	1
B20	1	1	-1
B21	1	1	-1
B22	1	1	-1
B23	1	1	-1
B24	1	1	-1
B25	1	1	1
B26	1	1	-1
B27	1	1	-1
B28	1	1	-1
B29	1	1	-1
B30	1	1	-1
C31	1	1	1
C32	1	1	1
C33	1	1	1
C34	1	1	-1
C35	1	1	1
C36	1	1	1
C37	1	1	1

ID	Invoked Function Correct	Arguments Correct	Chat Output Correct
C38	1	1	1
C39	1	1	0
C40	1	1	1
C41	1	1	1
C42	1	1	1
C43	1	1	1
C44	1	1	-1
C45	1	1	-1
D46	1	1	1
D47	1	1	1
D48	1	1	1
D49	1	1	1
D50	1	1	1
D51	1	1	1
D52	-1	-1	-1
D53	1	1	1
D54	1	1	1
D55	1	1	1
D56	1	1	1
D57	1	1	1
D58	1	1	1
D59	1	1	1
D60	1	1	1
E61	-1	-1	1
E62	1	1	1
E63	-1	-1	1
E64	-1	-1	1
E65	1	1	1
E66	1	1	1
E67	1	1	1
E68	-1	-1	-1
E69	1	1	1
E70	1	0	1
E71	1	1	1
E72	1	1	1
E73	1	1	1
E74	1	1	1
E75	1	1	1
F76	-1	-1	1
F77	1	1	1
F78	1	1	1
F79	1	1	1
F80	1	1	1
F81	1	1	1
F82	1	1	1

ID	Invoked Function Correct	Arguments Correct	Chat Output Correct
F83	1	1	1
F84	1	-1	1
F85	1	1	1
G86	-1	-1	1
G87	-1	-1	-1
G88	1	-1	-1
G89	-1	-1	1
G90	-1	-1	1
G91	-1	-1	1
G92	1	-1	-1
G93	-1	-1	1
G94	1	-1	-1
G95	-1	-1	1
G96	1	1	1
G97	-1	-1	-1
G98	-1	-1	-1
G99	-1	-1	-1
G100	-1	-1	1

5.3 Overall performances

If we exclude group G from the queries (which was more of an experiment), the results are really promising:

	Fully correct (%)	Considering also partially correct (%)
Function call accuracy	92,9	92,9
Parameters accuracy	90,6	91,8
Chat output accuracy	80,0	81,2

Table 5.2: Overall performances without group G

It must be said that the results are really good, considering that this is an open-source LLM model with relatively few billion parameters, and all the function descriptions were very generic. We didn't even use few-shot strategies, which would certainly have further improved performance (but with a slight trade-off on inference speed and loss of generalization).

Anyway, these are the results if we consider also group G:

	Fully correct (%)	Considering also partially correct (%)
Function call accuracy	83,0	83,0
Parameters accuracy	78,0	79,0
Chat output accuracy	76,0	77,0

Table 5.3: Overall performances including group G

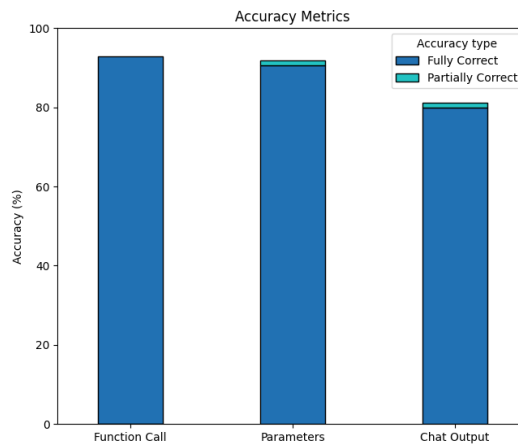


Figure 5.3: Accuracy metrics without group G

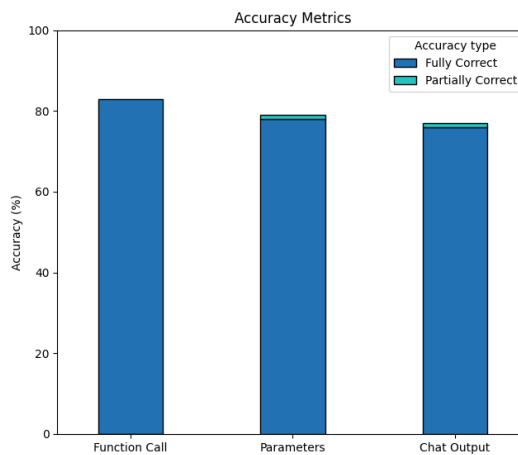


Figure 5.4: Accuracy metrics including group G

In the next subsections, it will be analyzed separately by each group, and also groups A, B and C together (retrieval queries) and D, E, and F together (setting and modifying queries).

5.3.1 Group A, retrieve information about specific devices

	Fully correct (%)	Considering also partially correct (%)
Function call accuracy	100,0	100,0
Parameters accuracy	100,0	100,0
Chat output accuracy	100,0	100,0

Table 5.4: Group A performances

In these queries, the most important accuracy to consider is the chat output accuracy.

This is a brilliant result, if you ask for information about a single device, it will reply with correct information, also if you ask for information about something that does not exist (like A3).

If we want to be very critical, in 4 cases it responded with correct information but included sentences that were superfluous to the question asked (A9, A12 and A14).

5.3.2 Group B, retrieve information about a room

	Fully correct (%)	Considering also partially correct (%)
Function call accuracy	100,0	100,0
Parameters accuracy	100,0	100,0
Chat output accuracy	26,7	26,7

Table 5.5: Group B performances

About this specific set of queries, the chat output accuracy is bad. The results here are justified because we’ve posed many queries that ask information about a device in a room that exists in the configuration but not in that specific room (B20, B22, B23 and B29).

Furthermore, when you ask queries like: *List all <category of device> in the <specific room>* tends to hallucinate about which devices are actually in that room (B17, B23, B24, B27, B29 and B30).

5.3.3 Group C, retrieve information about the entire house

	Fully correct (%)	Considering also partially correct (%)
Function call accuracy	100,0	100,0
Parameters accuracy	100,0	100,0
Chat output accuracy	73,3	80,0

Table 5.6: Group C performances

The results here are acceptable. Nothing particular has emerged here, the errors are all about hallucination that the LLM had when it has to list all devices, or all devices that have a particular setting (C34, C39, C44 and C45).

5.3.4 Group D, queries for setting specific devices

Here of course it is fundamental that the LLM has called the correct function with the correct parameters. The results are pretty good, considering that the 93,3% it calls correctly the function (and so, it performs what it is supposed to do). In this particular case, just one time it has called the `respond` function instead of the `update_individuals` (D52).

	Fully correct (%)	Considering also partially correct (%)
Function call accuracy	93,3	93,3
Parameters accuracy	93,3	93,3
Chat output accuracy	93,3	93,3

Table 5.7: Group D performances

5.3.5 Group E, queries for setting group of devices

	Fully correct (%)	Considering also partially correct (%)
Function call accuracy	73,3	73,3
Parameters accuracy	66,7	73,4
Chat output accuracy	93,3	93,3

Table 5.8: Group E performances

Here results tend to be lower than the previous case, because many times it has called the `respond` function instead of the `update_individuals` (E61, E63, E64 and E68).

Chat output accuracy is bigger than the parameters accuracy, this means that as a user that is using the system, for him the system is a black box. For him the 93,3% has done what it is supposed to do, but in reality it hasn't (because only the 73,4% of the cases it has really done what it was asked to do).

5.3.6 Group F, queries for modifying the configuration

	Fully correct (%)	Considering also partially correct (%)
Function call accuracy	90,0	90,0
Parameters accuracy	80,0	80,0
Chat output accuracy	100,0	100,0

Table 5.9: Group F performances

Results are pretty good. Just one time it has called the wrong function (F76), and just one time it has instantiated the wrong arguments (F84).

The chat output accuracy is bigger than the parameters accuracy for the same reasons as before.

5.3.7 Group G, advanced queries conditioned by sensor data

	Fully correct (%)	Considering also partially correct (%)
Function call accuracy	26,7	26,7
Parameters accuracy	6,7	6,7
Chat output accuracy	53,3	53,3

Table 5.10: Group G performances

Results are pretty bad. For now, the system is not able to handle this type of advanced queries. The reason could be the lack of capacity of the LLM used, it is too small to understand this type of queries. Just four times it has called the right function, and of these 4 just one time it has called also the right arguments.

6

Conclusions and Future Work

The system does what is supposed to do 90.6% of the time. This is a very astonishing result.

This is the future of our smart homes. The results are promising, showing that the use of offline LLMs is good for these tasks. Offline models ensure better privacy, as data isn't sent to external servers, and they significantly reduce costs. By handling these tasks locally, smart homes can operate more securely and cost-efficiently. The combination of high performance and cost-effectiveness is a game-changer for the smart home industry.

As mentioned earlier in this report, it could be beneficial to include examples of function usage (using a few-shot prompting strategy) to further improve accuracy. However, this should be approached cautiously due to potential loss of generality and increased time required to produce results.

Furthermore, handling queries like those in the G group will be quite challenging. Potential solutions could involve developing specific functions aiming to address these advanced queries, or leveraging more powerful open source LLMs if they become available in the future. It will be intriguing to observe the performance in the future when tasked with such complex queries.

Bibliography

- [1] *OWL 2 Web Ontology Language Primer (Second Edition)*. (n.d.).
<https://www.w3.org/TR/owl2-primer/>
- [2] *Prompt Engineering Guide – Nexta*. (n.d.). <https://www.promptingguide.ai/>