Improving Autonomy and Natural Interaction with a Pepper Robot through the Evaluation of Different Large Language Models

Luccas Rojas Becerra[1] and Juan Andrés Romero Colmenares[2]

[1] Universidad de los Andes, Colombia `l.rojasb@uniandes.edu.co`
[2] Universidad de los Andes, Colombia `j.romero11@uniandes.edu.co`

Abstract. Social Robotics is a field dedicated to exploring robots as interactive social companions and aides. Throughout its evolution, this discipline has centered on creating versatile robots that can tackle a range of tasks, functioning autonomously to aid humans. Nevertheless, achieving this is a complex endeavor, as robots must possess the ability to comprehend their surroundings and accurately interpret human directives. The main challenge confronting both these robots and their developers is the disparity between given instructions and actual behavior within particular contexts. Right now, many of these robots have to be manually programmed for specific tasks. They might be good at those jobs, but they don't do so well when you ask them to do a different activity. This happens due to the lack of cognition in pre-programmed deterministic jobs, where the robot cannot comprehend the reasoning behind its actions.To address this issue, this research aims to improve the autonomy and natural interaction of a Pepper robot by evaluating different large language models (LLMs). By leveraging the capabilities of LLMs, the research aims to develop a system that allows the robot to autonomously follow instructions given in natural language to accomplish general-purpose tasks. The evaluation process involves comparing the performance of commercial and open-source LLMs in generating code instructions for the robot. The generated code will be evaluated based the code execution results and will include a comparison of different prompting strategies and code abstraction levels. The evaluation will be conducted through both automatic and human evaluation processes. The results of this research will contribute to the development of highly effective robots capable of performing various general tasks.

Keywords: GPSR · EGPSR · Large Language Models · Artificial Intelligence · Pepper Robot

1 Problem

1.1 Problem Description

Social Robotics is a field dedicated to exploring robots as interactive social companions and aides. Throughout its evolution, this discipline has centered on creating versatile robots that can tackle a range of tasks, functioning autonomously to aid humans. Nevertheless, achieving this is a complex endeavor, as robots must possess the ability to comprehend their surroundings and accurately interpret human directives. The main challenge confronting both these robots and their developers is the disparity between given instructions and actual behavior within particular contexts.

Right now, many of these robots have to be manually programmed for specific tasks. They might be good at those jobs, but they don't do so well when you ask them to do a different activity. This happens due to the lack of cognition in pre-programmed deterministic jobs, where the robot cannot comprehend the reasoning behind its actions. Luca Iocci, et al [1] define cognition as "the ability to understand the current situation of the environment and the user needs, and to reason about the world as well as the robot's skills for achieving user goals. In other words, the robot is expected to execute actions and demonstrate that it has understood the user needs and can properly satisfy them."

For the purpose of this project, our evaluation will involve the utilization of a Pepper robot. Developed by SoftBank, the Pepper robot is renowned as the "the world's first social humanoid robot able to recognize faces and basic human emotions" [2]. This commercial robot has achieved substantial global prominence due to its unique capabilities. Our selection of this robot for our study is grounded in its sophisticated attributes, which include dual cameras positioned in its mouth and head, a 3D camera within its eyes, a quartet of microphones in its head, and a gyroscope-equipped torso replete with touch and collision sensors. Collectively, these features imbue the robot with remarkably lifelike qualities, transcending a mere semblance of a doll-like appearance to embody a more genuine human-like presence [3].

Nowadays, with the rise of large language models, there's a new opportunity for developing robots adept at handling a multitude of different tasks. Society can benefit greatly from the diverse array of applications offered by language models, such as aiding in code and writing auto-completion, enhancing grammar accuracy, generating game narratives, refining search engine results, and providing answers to inquiries[4].Using these language models to tackle this problem could help robots and humans work together more easily and ultimately bringing us closer to the achievement of highly effective robots capable of performing various general tasks.

2 Related Work

Natural Language and Robot-Human Interaction: In contemporary social robotics, there is an intensified emphasis on establishing a robust human-robot interaction that aligns with natural communication between robots and humans. As Stefanie Tellex from Brown University states: "Since most users that interact with robots will not be experts, it is becoming essential to provide natural, simple ways for people to interact with and control robots" [5]. In this article, they test the use of natural language models from a robotics point of view, where they covered different approaches such as formal methods, machine learning, and HRI methods. Their conclusion suggests that "we need more robust models for the entire planning and perceptual stack of the robot in order to integrate with natural language requests"[5]. Furthermore, entities like RoboCup are actively engaged in researching the challenges that arise in the context of human-robot interactions. In their analysis they identify that the most prevalent issues are associated with environmental noise coming from commands and the environment itself, and operator bias. The latter issue underscores the prevalence of commands given to the robot being influenced by expert operators, posing a significant challenge when interacting with individuals who lack expertise in robotics.[6]

Code Generation with LLM: Another noteworthy issue we find imperative to address is related to to the requirement for coding using natural language models, particularly in the context of self-planning code generation. Research conducted at Peking University, China, underscores a significant limitation in code generation, which is the manual construction of prompts. In their study, they explore methods for code generation that leverage hierarchical structures and the generation of sub-tasks. These strategies aim to facilitate clearer problem decomposition and enhance the management of complex domains, ultimately improving the code generation process[7]. However, it is also essential to emphasize that the outcomes produced by these models can significantly impact the volatility of the generated code. Research conducted at the University of Bristol in the UK highlights the instability of results when using Large Language Models. These models exhibit a high degree of unpredictability, often generating vastly different code outputs for same identical prompts. In light of their findings, this research concludes that there are substantial levels of non-determinism. To elaborate, the study revealed that the percentage of coding tasks producing zero identical test outputs across various requests stands at 72.73%, 60.40%, and 65.85% for the CodeContests, APPS, and HumanEval code generation benchmarks, respectively [8].

Code Generation for Robotics: The convergence of human-robot interaction and code generation finds its nexus in the utilization of large language models for robotics, with the aim of enhancing both the quality of human-robot interaction and the successful execution of complex tasks. This convergence has gained attention from prominent companies, such as Microsoft, as evidenced by their February 2023 publication. In this study, they assess the applicability of ChatGPT in robotics applications, investigating the transformation of natural language into executable tasks through ChatGPT's code generation capabilities. This process unfolds by initially creating a comprehensive robot function library at a high abstraction level. Subsequently, a prompt for ChatGPT is formulated, outlining a specific objective and incorporating the functionalities from the library. The generated code is then subject to evaluation through analysis or simulation before its deployment onto the physical robot [9]. In their conclusion, this publication presented a framework developed by them that "allows the generated code to be tested, verified, and validated by a user on the loop via a range of methods including simulation and manual inspection." Furthermore, Microsoft enphazises on noting that "due to the possibility of LLMs generating incorrect responses, human supervision is essential for ensuring solution quality and safety before executing code on robots" [9]. Additional research conducted by Ishika Singh et al. in 2022 highlights the significant role that large language models play in facilitating world comprehension by robots when handling intricate tasks. These models enable robots to grasp common-sense knowledge related to objects and locations, ultimately aiding in task execution. Their study underscores that large language models serve as an effective solution, primarily due to their capacity to comprehend the world and leverage the capabilities of the robot. Their findings reveal an impressive success rate exceeding 40%, where success is defined as flawlessly achieving the task's intended goal without any errors when employing codex.[10]

3 Objectives

3.1 General Objective

Implement and evaluate a system on a Pepper-like robot to be able to autonomously follow instructions given by natural language to accomplish general purpose tasks. (GSPR and EGPSR)

3.2 Specific Objectives

1. Build an interface module that acts as a mediator between primitive robot commands and higher level behaviors.
2. Implement commercial and open-source LLMs to generate Pepper mediator code.
3. Create multiple levels of abstraction in the robot code base and prompt instructions to modify the possible outputs of the LLMs.
4. Evaluate the results of code generated by the LLMs and how Pepper responds to it at five possible outcomes: did not execute, lack of robot capabilities, executed but failed to complete the task, partially completed the task, and successfully completed the task.
5. Compare the results obtained by the commercial and open-source LLMs.

4 Methodology

4.1 General Description

To achieve the status of a general-purpose robot, it is imperative to give the robot a comprehensive toolkit that enables it to comprehend its surroundings and interpret human verbal commands. In this endeavor, our aim is to empower the robot with the ability to autonomously generate the necessary procedural steps for accomplishing tasks provided by humans. This will be accomplished through the following general steps:

1. The initial step involves transcribing verbal instructions into text format, facilitating subsequent analysis by the robot. (Text to Speech process)
2. Subsequently, the textual instructions are processed by a large language model. This model will generate precise code instructions, adjusted for execution by the robot.
3. Ultimately, the robot proceeds to execute the prescribed actions, culminating in the completion of the designated task.

The central focus of this work is centered on refining the second step. This refinement is achieved by evaluating a spectrum of alternative large language models, diverse prompts, and varying levels of abstraction within the generated code.

4.2 Methodology Approaches

Command Input Generation: In the context of our research, the "Command Input Generation" phase involves generating distinct input scenarios for the pepper robot. These scenarios cover a range of task cases utilized within the RoboCup competition, including simple, complex, and even tasks that may be considered impossible due to contextual constraints (i.e "Go pick up an apple from the kitchen", but there is no apple to be found in the kitchen). To facilitate this process, we have chosen to utilize two specific scripts, both of which have been developed by the RoboCup@Home team. The initial script concerns an earlier generation of tasks, dating back to 2013 [11], whereas the subsequent script encapsulates tasks of a more recent nature, reflective of the year 2023 [12]. Through the integration of these scripts, our objective is to generate as many possible valid tasks that the robot could face in a real case scenario.

Large Language Models: In our research, the primary focus of our investigation lies in the evaluation of three prominent language models: Llama 2 from Meta, GPT-3.5 Turbo and GPT-4 from OpenAI. These models have been selected as the central subjects of our study due to their significance in the field.

Prompting: Regarding the assessment of prompts, our analysis encompasses a comparison between two distinct approaches: a comprehensive single prompt, encompassing the robot's codebase interface, verbal instructions, and general constraints, versus a chained-prompt methodology. In the latter approach, verbal instructions are deconstructed into simpler steps by the model beforehand. These individual steps are then incorporated into a subsequent prompt adhering to the same structure as the first approach, ultimately generating the required code to accomplish the given task.

Code Abstraction and Granularity Level: Furthermore, the evaluation of granularity levels involves the creation of an adaptable interface for the robot's codebase, featuring varying levels of granularity – spanning basic robot commands, services, and behaviors. Through this, we intend to ascertain the optimal granularity level that facilitates the most effective interaction with the language model.

4.3 General Evaluation

The evaluation process is facilitated by generating an array of tasks using the task generator. The results of these tasks are categorized into distinct outcomes: "did not execute","did not execute", "executed but failed to complete the task", "partially completed the task", and "successfully completed the task." Furthermore, we also included an additional outcome for when the task is too complex and the robot has a lack of capabilities in order to successfully perform it, this alternative outcome was labeled "lack of robot capabilities". This comprehensive assessment framework aims to refine the approach and lead to enhanced robot performance.

To effectively conduct this evaluation, it will be performed and categorized into two sections: automatic evaluation and human evaluation.

Automatic Evaluation: For the automated assessment, an automatic workflow will be established where tasks are generated, processed by the LLMs, and the output is checked for both invalid parameters within the task's context and correct code execution (with code being interpreted by Python without exceptions). Subsequently, these results will then be recorded in a database, setting the stage for the subsequent evaluation phase. This comprehensive approach addresses both the execution and compilation aspects of the framework.

Human Evaluation: Next, the outcomes that successfully pass the automatic evaluation will undergo manual execution by the Pepper robot. These results will then be reassessed with regard to the remaining outcome categories, namely "partially completed the task" and "successfully completed the task." This hands-on evaluation will be conducted by the project researchers.

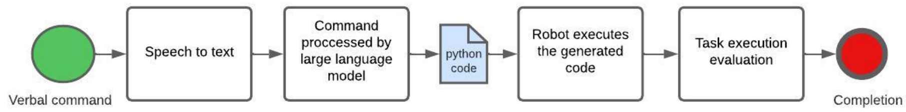

Fig. 1. General Evaluation Process

The diagram provided above illustrates the sequential process outlined in the previously described methodology. This process remains independent of the specific large language model or prompting strategy employed.

4.4 Input and Output Expected Examples

In this section, we present expected examples of input and output. The models' primary function is to receive a task in the form of natural language commands and subsequently produce executable code tuned specifically for the robot. These tasks are straightforward instructions that any individual can convey to the robot at any given time. The overarching goal of large language models is to intelligently strategize the essential steps required to fulfill each task and then generate code that ensures the successful execution of the given instructions. Throughout the subsequent examples, the output is illustrated through snippets of successfully generated Python code.

Example 1.

Input: Robot please look for Angel in the corridor, follow her, and tell what day is today.
Expected Output:

```
from datetime import datetime
# Step 1: Navigate to the corridor
self.tm.go_to_place("entrance")

# Step 2: Look for Angel in the corridor and follow him
found_angel = self.tm.find_object("person", timeout=25)
if found_angel:
    self.tm.talk("I found Angel in the corridor! I will now follow
        Angel")
    self.tm.follow_you()
else:
    self.tm.talk("I couldn't find Angel in the corridor. I cannot do
        that task.")
# Step 3: Answer what day is today
day = datetime.today().strftime("%A")
self.tm.talk("Today is " + day)
```

Example 2.

Input: Go to the kitchen, grab a drink and give it to Emily at the bedroom
Expected Output:

```
# Step 1: Navigate to the kitchen
self.tm.talk("I will navigate to the kitchen")
self.tm.go_to_place("kitchen")

# Step 2: Find the drink
self.tm.talk("Searching for a drink")
drink_found = self.tm.find_object("bottle")

# Step 3: Grasp the drink
if drink_found:
    self.tm.talk("I will grasp the drink")
    self.tm.grasp_object("bottle")
else:
    self.tm.talk("I could not find a drink")

# Step 3: Navigate to the bedroom
self.tm.talk("I will navigate to the bedroom")
self.tm.go_to_place("bed")

# Step 4: Speak to Emily
if drink_found:
    self.tm.talk("Hello Emily, here's your drink")
    self.tm.leave_object("bottle")
else:
    self.tm.talk("I could not find you a drink, I am sorry")
```

Example 3.

Input: Could you please tell me how many elders are in the kitchen.
Expected Output:

```
# First, navigate to the kitchen
self.tm.go_to_place("kitchen")

# Use the perception service to count the number of elders in the kitchen
num_elders = self.tm.count_objects("elder")

# Use the speech service to say the result
self.tm.talk("There are " + str(num_elders) + " elders in the kitchen.")
```

4.5 Large Language Model Processing Details

As evidenced in **Fig. 1**, the most relevant step for this project focuses on natural language processing, which is capable of generating the code that the robot will execute. However, this step is composed of multiple stages. Initially, it is crucial to generate a code interface that consists of high-level functions that the robot can use. These functions should be well understood by the language model and range from actions like talking, counting objects, going to a place, asking something, searching for a specific object, to following someone, among others. The granularity level of these functions is crucial, as through various testing, we observed that when the abstraction level was too low, the language model tended to make more errors. Having functions at a very high level that align more with the model of the world that the language model understands was found to be more effective. This interface should be easy to use, and the parameters should be as intuitive and well-explained as possible.

For the case of prompting this project will be evaluating two types of prompts, the first one being "Long String" prompting, which is represented by a single prompt containing all of the necessary instructions. The second one, "Chain Prompting" splits the hole task in specific steps that are more simple.

Long String Prompting In Long String prompting the prompt included a defined structure with all necessary aspects to accomplish the task, this structure contained:

- **The goal of the model:** This includes the specification about the main purpose of the model and the role it performs in the whole robot architecture.
- **The available functions from the task module:** These functions belong to the codebase interface and call services from the Robot, the syntax of the functions, the parameters, and outputs were included in this section.
- **The available defined entities and tags:** This is important because many services from the robot require precise syntax to work properly, this includes all of the places, objects, and questions tags that are previously predefined.
- **The output format:** In this section, the specified format of the response is informed. The prompt tells the language model that the response has to be a continuous snippet of code written in python.
- **One example of desired output:** An example of a real input and expected output is given, using the task module and the available entities.
- **The robot constraints :** Some of the tasks are not possible to accomplish due to the robot's current capabilities, these tasks have to be handled by the model, by specifying why the robot can't accomplish the task instead of trying and failing.
- **The input task :** Finally, the task that the model has to solve utilizing the rest of the knowledge.

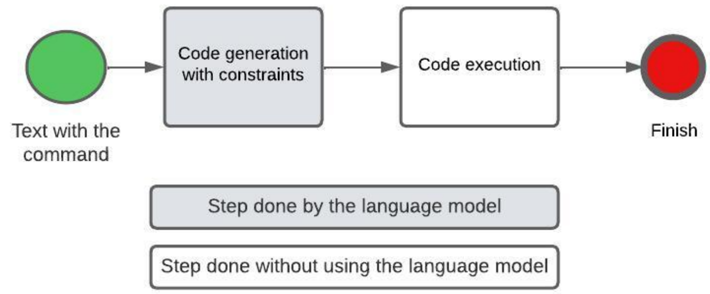

Fig. 2. Long String Process

Chain Prompting In the case of "Chain Prompting", the approach was different, trying to improve the model performance by splitting the task into simple steps that together completed the task. The diagram below illustrates the sequence of steps used with the language model to obtain better results in code generation.

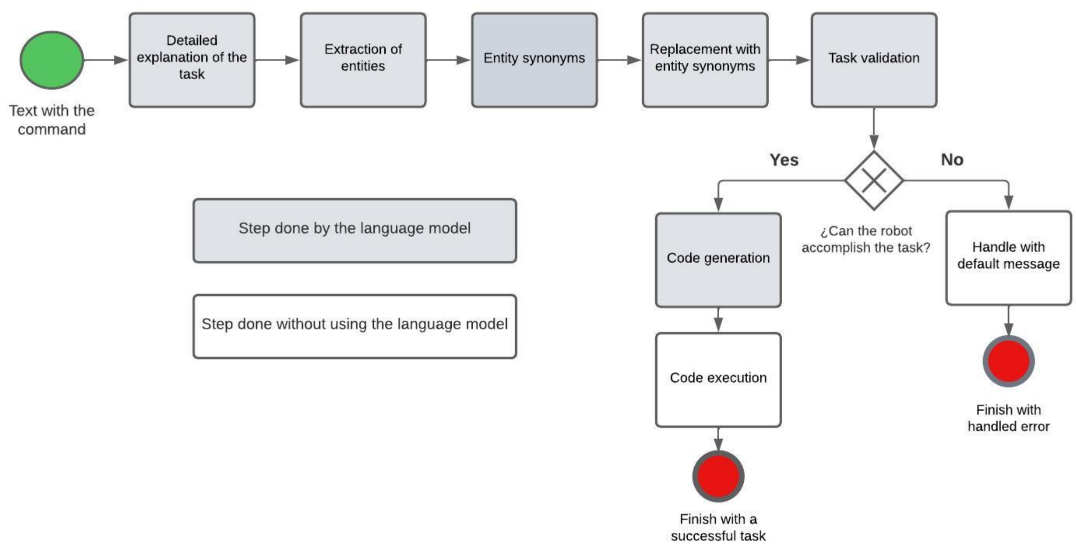

Fig. 3. Chain Prompting Process

Chain Prompting Steps The following steps will be explained accompanied by an example of a task to better describe the process. The proposed task for the example is:

Input: Please locate Charlotte in the dining room, look for Emma in the living room, and follow her.

Task Expansion: In this step, the language model is asked to expand the task a bit by providing more details on the steps the robot should follow. For example, some of these details include assuming the robot is never in the location it needs to go, helping the subsequent steps to be executed more effectively and enabling the robot to fulfill the task.

```
# Example Task Expansion Response
"Navigate to the 'dining area' and initiate person recognition. If a
    'person' is detected, ask
'Are you Charlotte?'. If the response is affirmative, acknowledge and
    save the location. Then,
navigate to the 'living room sofa' and repeat the person recognition
    process. If a 'person' is
detected, ask 'Are you Emma?'. If the response is affirmative,
    acknowledge and follow her. If
either person is not found, report the absence and await further
    instructions."
```

Entity detection: In this step, entities described in the task are identified, such as objects, people, places, and more. The language model is instructed to extract this information and return it as a Python list with the recognized entities.

```
# Example Entity Detection Response
['dining_area', 'person', 'Charlotte', 'living_room_sofa',
    'living_room', 'Emma']
```

Entity Synonyms: The extracted list of entities is replaced with entities that the robot understands. This is done because the robot's interface only contains predefined places and objects established by the machine learning models used by the robot, which must adhere to certain syntactic rules. The model is tasked with returning a Python list replacing the entities with syntactically correct ones.

```
# Example Entity Synonyms Response
['dining_room', 'person', 'name', 'office', 'living_room', 'name']
```

Entity Replacement: Using the previous result, the language model is instructed to replace the entities from the given task with syntactically correct synonymous entities extracted through the earlier process. In the following example we can see that the LLM needs to replace dining area for 'dining_room' and living room sofa for 'living_room'.

```
# Example Entity Replacement Response
"Navigate to the 'dining_room' and initiate name recognition. If a
    'person' is detected, ask
'Are you Charlotte?'. If the response is affirmative, acknowledge and
    save the office. Then,
navigate to the 'living_room' and repeat the name recognition process.
    If a 'person' is
detected, ask 'Are you Emma?'. If the response is affirmative,
    acknowledge and follow her. If
either person is not found, report the absence and await further
    instructions."
```

Task validation: With the expanded task, a validation is performed to determine if, with the currently developed capabilities, the robot is able to complete the assigned task. This involves checking if it recognizes the entities mentioned in the description and if, in general, its capabilities are sufficient for the task. The model is requested to return whether the robot has the capabilities and or not. If the model indicates that the robot lacks the necessary capabilities, error handling is performed, and the LLM explains why it cannot perform the task based on the information returned in this step.

```
# Example Task Validation Response
True
```

Code generation: If the previous step indicates that the robot does have the capabilities to carry out the task, the language model is assigned with generating the necessary code to fulfill it based on the created interface and refined task. This step returns Python code that is executed directly on the robot to accomplish the proposed task.

```
# Example Code Generation Response
self.tm.talk('I am going to the dining room to initiate name
    recognition.')
self.tm.go_to_place('dining_room')
name = self.tm.q_a_speech('name')
if name:
    self.tm.talk('Are you Charlotte?')
    response = self.tm.speech2text_srv()
    if 'yes' in response.lower():
        self.tm.talk('Nice to meet you, Charlotte.')
        self.tm.add_place('office')
    else:
        self.tm.talk('I am sorry, I thought you were Charlotte.')
else:
    self.tm.talk('I did not detect a person.')
```

```
self.tm.talk('I am going to the living room to initiate name
    recognition.')
self.tm.go_to_place('living_room')
name = self.tm.q_a_speech('name')
if name:
    self.tm.talk('Are you Emma?')
    response = self.tm.speech2text_srv()
    if 'yes' in response.lower():
        self.tm.talk('Nice to meet you, Emma.')
        self.tm.follow_you()
    else:
        self.tm.talk('I am sorry, I thought you were Emma.')
else:
    self.tm.talk('I did not detect a person.')
```

4.6 Task Execution Evaluation Details

To evaluate the correct functioning of the code returned by the natural language processing model, we implemented the following process to automate the results to a significant extent so we can discard non executable code directly without having to manually review it.

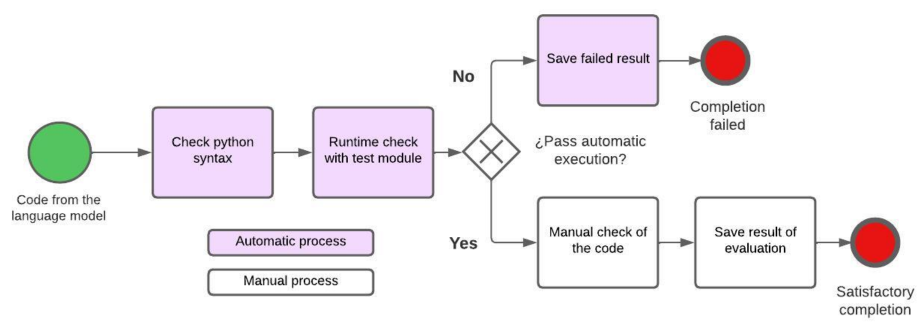

Fig. 4. Task Evaluation Process

Python Syntax Review: In this step, the code is checked to ensure it adheres to proper Python syntax, proper indentation, variable declaration, correct function calls, and more.

Runtime Review with testing module: Next, the returned code is examined to ensure it functions correctly with the high-level interface of the robot. To achieve this, a testing robot interface was created that can identify whether the parameters received by functions would generate an error or if the response type of a specific function is used correctly within the rest of the code.

Human or Robot review: Finally, a manual review of the Python code that is known to work is conducted, primarily to validate whether the code fulfills the task assigned to the robot. This step is only performed if the generated code successfully completed the Syntax and Runtime Review steps previously. Each result is then classified as not completed, partially completed, or completed satisfactorily. This provides greater certainty about the model's results. In some scenarios, the correct operation of the generated code can even be verified with the robot.

5 Results

5.1 Task Generation Results

This project involved the creation of 720 tasks using GSPR Task Generators developed by kryodhel for the RoboCup Competition [12]. To initiate the task generation process, we first determined the number of independent tasks required for each of the three GPSR (General Purpose Service Robot) category, as well as the five EGPSR (Extended General Purpose Service Robot) categories specified in the methodology. This encompassed a total of eight separate categories to analyze. The calculation then established a baseline of 15 tasks per category which were multiplied by the number of models to assess and the variety of prompting types employed. For a more comprehensive breakdown of these details, please refer to the following table.

Table 1. Task Generation Detailed Summary

Model	**Tasks per category**	**No. of categories**	**Tasks per prompting type**	**No. of prompting types**	**Total tasks per Model**
Llama 2	15	8	120	2	240
GPT 3.5	15	8	120	2	240
GPT 4	15	8	120	2	240
Total					720

The table displays two aggregate columns, illustrating the overall number of tasks for each prompting type and the total tasks per model. Consequently, each model is required to evaluate 240 distinct tasks, contributing to the previously mentioned total of 720 tasks evaluated.

5.2 Human Evaluation Reference

In the manual evaluation process, we employed a distinct logical reference framework to impartially assess the code generated by the large language models. This framework encompassed four categories: "NOT COMPLETED (MODEL),"

"PARTIALLY COMPLETED," "SUCCESSFULLY COMPLETED," and "LACK OF CAPABILITIES." Subsequent paragraphs will delve into each category, providing explanations and examples to better explain each case.

Not Completed (Model) In tasks labeled as "Not Completed", the generated code involves models executing actions completely unrelated to the proposed task or, in certain instances, failing to achieve anything useful in addressing the task at hand. Additionally, tasks were categorized as not completed if they successfully executed some required steps for task resolution but utterly failed to establish a logical sequence of steps to complete it. For instance, performing logical steps 3 and 4 first and skipping 2, among others, would fall into this classification. Some examples of this behavior can be seen in the following code snippets that were extracted from the code generation results.

Generated By: Llama2
Input: Please find a person in the cabinet, follow Alexander who is at the dining table, and go to the bar.

```
# Task: Find a person in the cabinet and follow them
self.tm.talk('I am going to find a person in the cabinet')
found_person = self.tm.find_object('person')
if found_person:
    self.tm.follow_you()
else:
    self.tm.talk('I cannot find a person in the cabinet')
```

The previous example fits the "Not Completed" category due to having an incomplete sequence of steps. Here the task asked the model to 1. Find a person in the cabinet 2. Follow Alexander at the dining table and finally 3. Go to the bar. However, the generated code doesn't go to the cabinet, tries to find a person and follow them and finishes. It misses steps 2 and 3 completely and includes half of step 1, rendering the result as "Not Completed".

Generated By: GPT 3.5
Input: Could you escort James from the office to the apartment, follow Daniel who is at the cabinet, and grasp the bowl from the bedside.

```
self.tm.talk('''I am sorry but I cannot do the task because I cannot go
    to the cabinet''')
```

The previous example is rendered as "Not Completed" since the generated code is simply a message from the robot saying that it cannot accomplish the task, however it is well within its capabilities to do so. Not doing something the robot is actually capable to do fits the "Not Completed" category. Additionally, the response message has incomplete information which hinders our capability to tell where did the model fail.

Generated By: GPT 4
Input: Please look for William in the Kitchen, follow him, and navigate to the bar.

```
self.tm.talk('''I am sorry but I cannot do the task because I cannot ask
    if they are William.''')
```

The given example was labeled as "Not Completed" because the generated code, just as it happened with GPT 3.5, merely conveys a message from the robot, expressing its inability to perform the task. However, it's important to note that the robot is fully capable of executing the task in question. Categorizing a task as "Not Completed" when the robot can, in fact, accomplish it falls within this classification.

Partially Completed Within the "Partially Completed" category, we analyze task outcomes where the sequence of steps mimics successful completion, but key elements crucial to the task fail to be accomplished. Picture it as if a third party were observing the robot, thinking it's completing the task successfully but suddenly a failure occurs somewhere within the process and fails to complete the 100% of the task. These outcomes range from overlooking a location name to omitting any step in the process, all while maintaining the logical continuity of the step sequence. Some examples of this behavior can be seen in the following code snippets that were extracted from the code generation results.

Generated By: Llama2
Input: Could you put containers on the office.

```
self.tm.talk('I am going to grab a bottle')
found_bottle = self.tm.find_object('bottle')
if found_bottle:
    self.tm.grasp_object('bottle')
else:
    self.tm.talk('I cannot find the bottle')

self.tm.go_to_place('kitchen')

self.tm.talk('I am going to the living room')

self.tm.go_to_place('living_room')

if found_bottle:
    self.tm.leave_object('bottle')
else:

self.tm.talk('I cannot find the bottle')
```

In the case of this example it can be seen that the generated code partially does what it is being asked by the task. It can be seen that the robot would go grab a bottle (which fits the description of a container) but then go to two different places that were not included in the task description, hence the "Partially Completed" classification.

Generated By: GPT 3.5
Input: Could you please pick up the beer from the TV stand, put it on the cabinet, and follow Angel who is at the dining table.

```
self.tm.talk('I am going to pick up the beer from the TV stand')
self.tm.go_to_place('tv_stand')
found_beer = self.tm.find_object('bottle')
if found_beer:
    self.tm.grasp_object('bottle')
else:
    self.tm.talk('I cannot find the beer')
self.tm.talk('I am going to put the beer on the cabinet')
self.tm.go_to_place('cabinet')
if found_beer:
    self.tm.leave_object('bottle')
else:
    self.tm.talk('I am sorry, I did not find the beer')
self.tm.talk('I am going to follow Angel who is at the dining table')
self.tm.go_to_place('dining_room')
self.tm.follow_you()
```

In this scenario, the generated code demonstrates successful completion of approximately 90% of the necessary steps to accomplish the task. However, it falls short when it comes to executing the final detail of following Angel. Although the robot successfully navigates to the dining room, it neglects to search for a person and initiates the "follow you" command without having a specific target to follow rendering the task execution result as "Partially Completed".

Generated By: GPT 4
Input: Grasp the tea from the living shelf, give it to me, and look for Madison in the TV stand.

```
self.tm.talk('I am going to grasp the tea from the living shelf')
self.tm.go_to_place('living_room')
found_tea = self.tm.find_object('mug')
if found_tea:
    self.tm.grasp_object('mug')
else:
    self.tm.talk('I cannot find the tea')

self.tm.talk('I am going to give the tea to you')
```

```
if found_tea:
    self.tm.leave_object('mug')
else:
    self.tm.talk('I am sorry, I did not find the tea')

self.tm.talk('I am going to look for Madison in the TV stand')
self.tm.go_to_place('tv_stand')
found_madison = self.tm.find_object('person')
if found_madison:
    self.tm.talk('I found Madison')
else:
    self.tm.talk('I am sorry, I did not find Madison')"
```

In this particular example, a situation similar to GPT-3.5's case happens. The generated code effectively handles nearly every aspect of the task but stumbles when it comes to locating a person for delivering the tea to the user. During the second step, the robot, instead of navigating to another location and searching for the person who initiated the task, simply leaves the tea in the living room.

Succesfully Completed In the context of the "Successfully Completed" category, as the name implies, it encompasses all outcomes featuring generated code that successfully accomplishes 100% of the given task. While the generated code within this category may not be flawless, it can still include results with minor errors, as long as the expected outcome of the task is completed. It's worth noting that, given the absence of a one-size-fits-all solution for every problem, we value the ingenuity of large language models in devising creative approaches to problem-solving.

Generated By: GPT 3.5
Input: Could you look for William in the living room, follow him, and grasp the egg from the office.

```
self.tm.talk('I am going to look for William in the living room')
self.tm.go_to_place('living_room')
found_william = self.tm.find_object('person')
if found_william:
    self.tm.talk('I found William, I am going to follow him')
    self.tm.follow_you()
    self.tm.talk('I am going to grasp the egg from the office')
    self.tm.go_to_place('office')
    found_egg = self.tm.find_object('egg')
    if found_egg:
        self.tm.grasp_object('egg')
    else:
        self.tm.talk('I cannot find the egg in the office')
else:
    self.tm.talk('I cannot find William in the living room')
```

In this example it can be seen that the task is completed successfully fulfilling all of the required steps to accomplish it. The generated code gots to the living room, looks for William, follows him, then goes and goes to the office and grasps the egg there. Every step is completed.

Generated By: GPT 4
Input: Look for Daniel in the sink, pick up the pringles from the drawer, give it to Emily in the cabinet, and answer a question.

```
self.tm.talk("I am going to look for Daniel in the sink")
self.tm.go_to_place("sink")
found_daniel = self.tm.find_object("person")
if found_daniel:
    self.tm.talk("I found Daniel")
else:
    self.tm.talk("I cannot find Daniel")

self.tm.talk("I am going to pick up the pringles from the drawer")
self.tm.go_to_place("kitchen")
found_pringles = self.tm.find_object("biscuits")
if found_pringles:
    self.tm.grasp_object("biscuits")
else:
    self.tm.talk("I cannot find the pringles")

self.tm.talk("I am going to give the pringles to Emily in the cabinet")
self.tm.go_to_place("cabinet")
found_emily = self.tm.find_object("person")
if found_emily and found_pringles:
    self.tm.leave_object("biscuits")
    self.tm.talk("Here are the pringles, Emily")
else:
    self.tm.talk("I am sorry, I did not find Emily or the pringles")

self.tm.talk("I am going to answer a question")
question = self.tm.speech2text_srv()
answer = self.tm.answer_question(question)
self.tm.talk(answer)
```

In this example, it's evident that the generated code fulfills each step as mandated by the task. It not only executes logical decisions in cases where Emily or the Pringles are not found but also incorporates fallbacks—essentially simple statements indicating the absence of X or Y element. This showcases a very well constructed code that proficiently accomplishes the assigned task.

Lack of Capabilities The "Lack of Capabilities" category, in essence, comprises outcomes where the model recognizes that the robot lacks the essential capabilities to successfully execute the given task and responds accordingly. This is a tricky category however, since there are situations where the large language models are creative enough to circumvent this absence of capabilities in some way or another. For instance, the robot does not currently have the capability to distinguish people with black hats or differentiate between young and old people. However, the model can do multiple things to go around this limitation. Such as, asking the person if they have a black hat or asking their age to corroborate if they are young or not. When the model successfully bypasses these limitations, the task may be categorized as "Partially Completed" or "Successfully Completed" based on the overall result. However, if the model accurately acknowledges its inability to perform certain actions, the task falls into the "Lack of Capabilities" category. Additionally, when the model falsely claims incapability despite possessing the necessary capabilities, the task is classified as "Not Completed." Some examples of tasks that got classified into the "Lack of Capabilities" categories can be found below.

Generated By: Llama 2
Input: Please describe the objects in the living table to Michael in the desk

```
self.tm.talk('''I am sorry but I cannot do the task because I can't
    recognize the objects on the bedside table using the available
    objects in Pepper's knowledge. I can also navigate to the bedside
    table using the available places in Pepper's knowledge. Therefore, I
    cannot provide a detailed description of the objects on the bedside
    table to Michael.''')
```

In this case the model correctly recognizes that it cannot describe the objects since it lacks the ability of specific object recognition. While Pepper can currently recognize some objects it cannot describe them to the user.

Generated By: GPT 4
Input: Could you greet the child in the office

```
self.tm.talk('''I am sorry but I cannot do the task because I cannot
    recognize or measure height.''')
```

In the case of this example, it is true that Pepper cannot recognize or measure height or age. Therefore rendering the result as a lack of capabilities of the model.

5.3 Overall Model Performance

The evaluation process was conducted on a total of 720 tasks. Out of these, 400 tasks successfully underwent automated evaluation, indicating the absence of Python errors, including syntax and runtime errors. Subsequently to this, the 400 tasks underwent human evaluation based on the constraints specified previously. The results after the complete manual assessing are shown in the pie diagrams below.

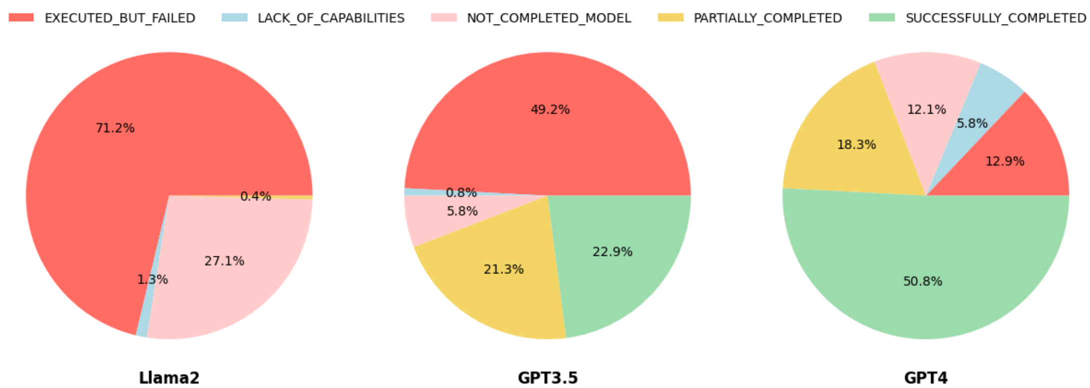

Fig. 5. Overall Task Results

As depicted in the previous diagrams, the performance of Llama 2 (7B parameters) in the results was notably poor, with not a single successful outcome observed. None of the results managed to fulfill the entire task, and only one task achieved partial completion. Turning to GPT 3.5, while the results were not as dismal as Llama 2, approximately 50% of the cases did not pass the automatic evaluation, the causing errors will be detailed in a following section. Nonetheless, there is a noticeable improvement in this scenario, with around 40% of the tests falling into the categories of partial and successful completion, and at least 20 of that 40% of the tasks being successfully completed by the model. Moving on to GPT 4, the results improved further, with an approximate 90% of successful automatic evaluation, 50% of which classified as successfully completed. Almost 20% of the tasks were partially completed, and only around 30% of the tasks faced complete failure. It's worth noting that this is just a general result of the overall evaluation and a more exhaustive analysis can be found in the following sections.

The following figure illustrates a different visualization of the same data. Each bar represents the percentage of tasks evaluated for each model, excluding tasks categorized by the models as "Lack of capabilities" from the 100%. It depicts the breakdown of tasks that passed automatic evaluation (represented by the bar height) into three subcategories: "Not completed", "Partially completed", and "Successfully completed", while the distance between the bars and the 100% represent the amount of tasks that failed the automatic evaluation.

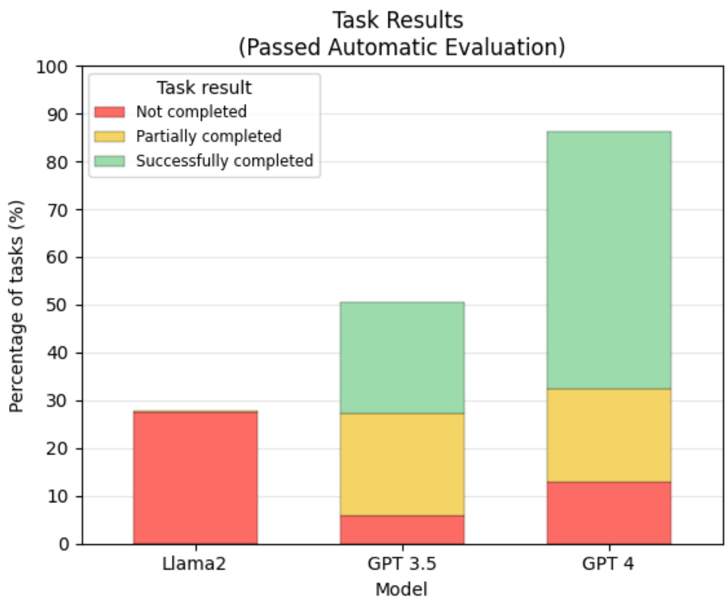

Fig. 6. Successful Task Results

5.4 Prompting Type Effectiveness

A more detailed analysis is necessary to discern the effectiveness of the two prompting strategies, Chain-Prompting and Long-String Prompting in addressing the problem at hand. Each strategy carries its own set of advantages and disadvantages, and the following breakdown provides a detailed examination of the results achieved with each type of prompt.

Chain Prompting The following diagram presents the results obtained using the Chain Prompting strategy, each bar represents the amount of tasks that correspond to each category after the corresponding evaluation.

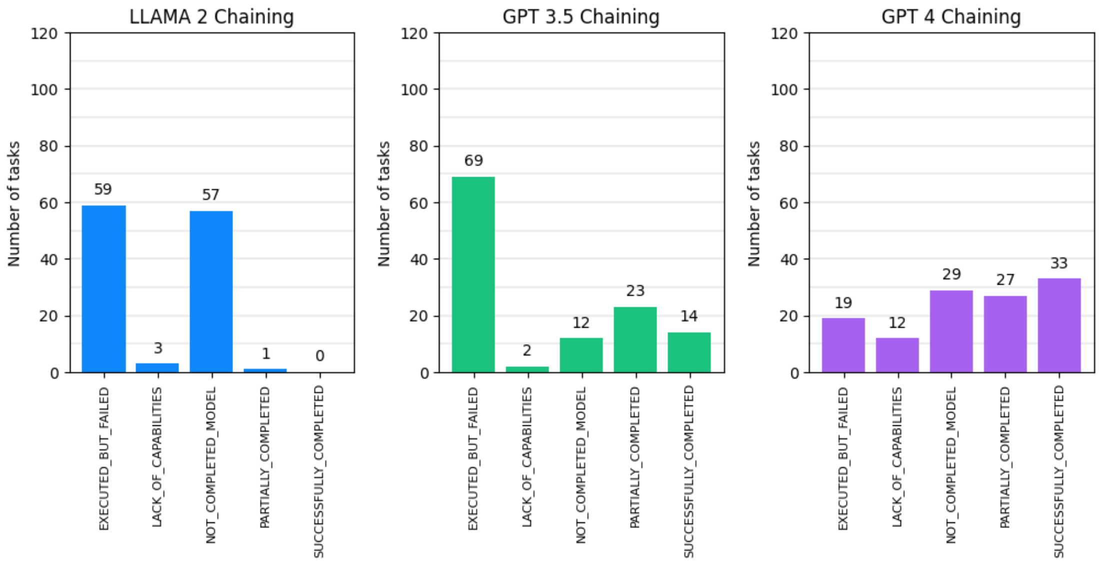

Fig. 7. Chaining Results

As illustrated in the preceding graph, the outcomes derived from employing a Chaining strategy appear suboptimal across all models. Despite this, the Chaining strategy proved beneficial in reducing errors related to accurate syntax usage when calling functions from the task module. Additionally, it aided in effectively distinguishing tasks that couldn't be executed successfully due to the robot's limitations. It is important to note that the overall success rate for this prompting strategy was less than 50% across all models.

Long String Prompting The application of a Long-String prompting strategy yielded rather surprising results, particularly as it outperformed the outcomes achieved with the Chaining Strategy. The subsequent graph illustrates the results obtained through the utilization of the Long-String prompting method.

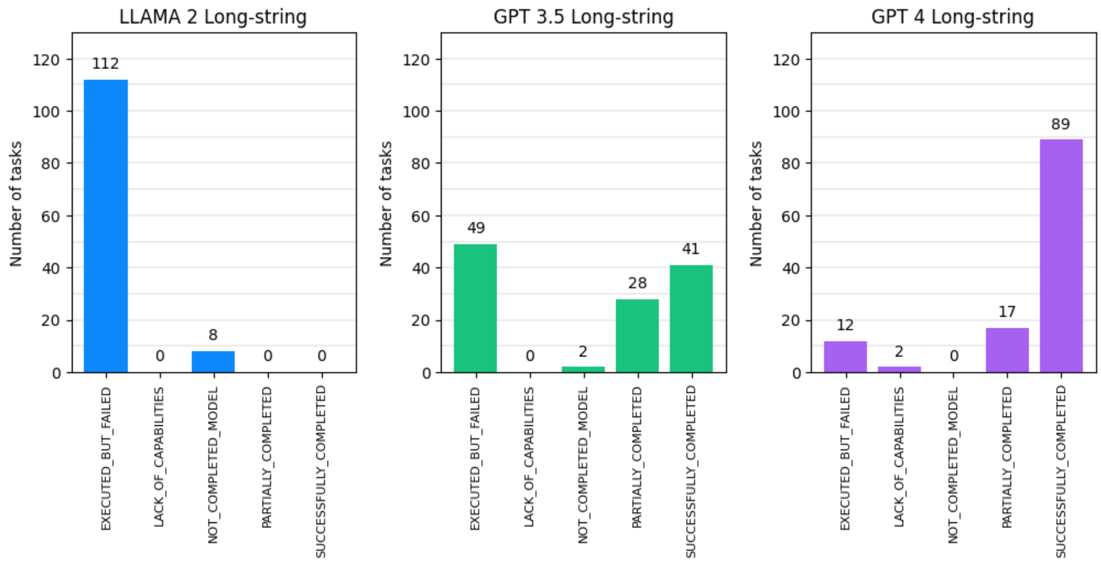

Fig. 8. Long String Results

As evident from the data, the Long-String prompting strategy yielded superior results for both GPT 3.5 and GPT 4. Conversely, in the case of Llama 2, the results were less favorable. For GPT 3.5, over 50% of the tasks were either partially or successfully completed, with approximately 35% of the results falling into the successfully completed category. GPT 4 showcased outstanding results, achieving a flawless execution of 74% of the tasks, while the remaining 26% saw 14% in partial completion. This combination of the Long-String strategy and GPT 4 emerged as the most effective prompting strategy and model, delivering the best overall results in our analysis.

Prompting analysis As demonstrated earlier, the models exhibited improved performance when employing the Long-String strategy. This approach offered a comprehensive set of instructions and constraints within the same context. While the Chaining Strategy effectively addressed common syntax errors and validated the feasibility of tasks within the robot's capabilities, the Long-String strategy emerged as the superior prompting approach by showcasing a higher degree of creativity in problem-solving, utilizing the robot's full capabilities to propose innovative solutions. It could be that the loss of context in the Chaining strategy caused the overall result to be less accurate and therefore failed more often than compared to the Long-String strategy. Additionally, since the Chaining strategy involves multiple sequential requests, an early failure in any of them can result in a domino effect that affects the overall result of the task.

5.5 Response Time Comparisons

An additional crucial aspect to consider when evaluating these models in the context of social robotics is their response time. This factor holds significance as the duration during which the large language model processes information can be perceived as a non-response by users. This potential delay poses a challenge in maintaining a natural interaction with a robot, and its impact is particularly relevant in the scope of this study.
The following plot illustrates the distribution of response times for the two models utilized via OpenAI API. It's important to note that Llama 2 is excluded from this plot as it was executed on dedicated hardware, resulting in significantly increased response times.

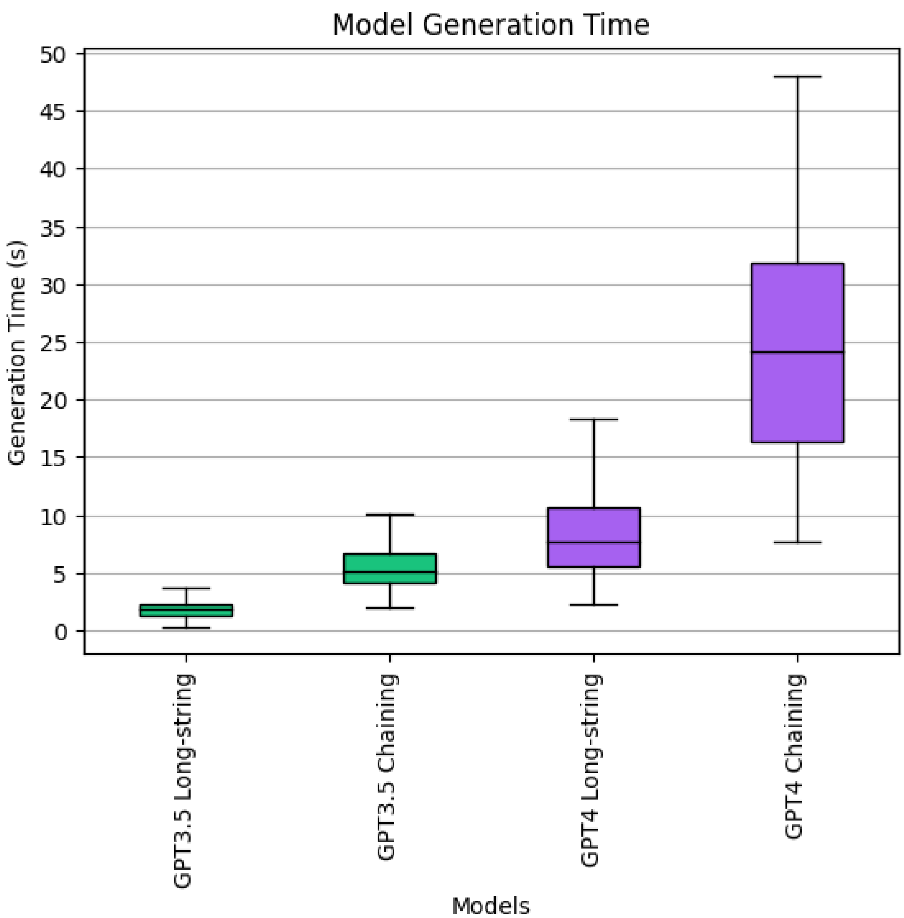

Fig. 9. Model generation time

As outlined by Robert Miller in his paper "Response time in man-computer conversational transactions" [13], the "two seconds rule" posits that the expected response time for any conversational transaction should ideally approach two seconds. Subsequently, Toshiyuki Shiwa et al. extended this concept to the realm of social robotics in their paper "How Quickly Should a Communication Robot Respond?" [14], reinforcing the same temporal guideline. The suggestion to incorporate "Conversational Fillers" was made for situations where the response time exceeded 2 seconds.

In our specific application, only the average results from GPT 3.5 Turbo approximately align with this two-second timeframe. Conversely, the response times generated by GPT-4 Long-String average around 7 seconds, significantly exceeding the recommended window for a prompt robot response. While the use of conversational fillers may be viable in the case of GPT 3.5, the prolonged response time of GPT-4 could potentially be perceived as a non-response by the robot. In the case of GPT-4 employing a Chaining strategy, its minimum response time far surpasses the 2-second threshold, and its average lingers around 24 seconds. Such durations are entirely impractical for human-robot interaction even if some conversational fillers are added. This prolonged response time can likely be caused by the inherent nature of the Chaining process, where multiple requests are executed sequentially by the already slow GPT-4 API response times.

5.6 Automatically Identified Errors

As outlined in the methodology, the primary automatic evaluation considered Python syntax issues and potential runtime problems arising from incorrect syntax when calling functions that expect specific parameters, objects, or question tags. This automated assessment provided valuable insights into the primary errors encountered by each model during the code generation task. The following diagram depicts the key errors identified in all three models, emphasizing that these errors can lead to exceptions and significant issues in the execution process.

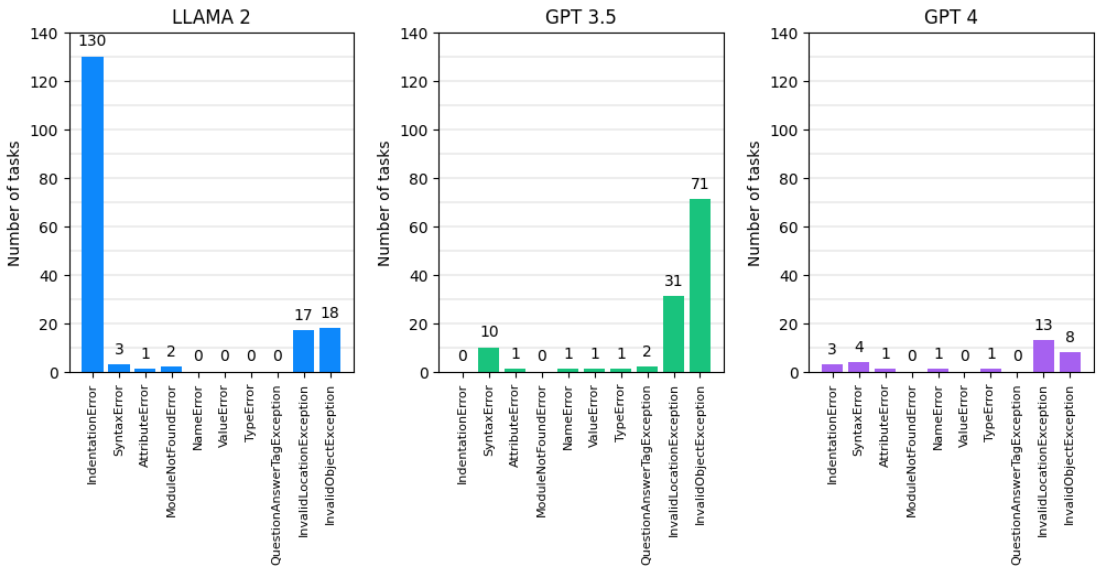

Fig. 10. Model errors

As depicted in the image above, Llama 2 exhibited a higher incidence of errors, with the majority resulting from improper code indentation. Additionally, Llama 2 struggled with employing the specific syntax required for calling certain functions causing it to cause both Syntax and Indentation errors. In the case of GPT 3.5, the primary error was associated with the use of invalid objects, places, and question tags, particularly noticeable when implementing the long string approach. On the other hand, GPT 4 demonstrated a commendable ability to address most issues effectively. However, it is noteworthy that challenges persist in the syntax of specific places, objects, and question tags. This recurrent issue in large language models suggests a tendency to employ synonyms instead of adhering strictly to the prescribed syntax for calling functions, places, objects, and other elements.

5.7 Model Response Analysis

Positive and Creative Responses
Overall, the majority of the responses we received produced well-crafted code. The generated output consistently adhered to the syntactic constraints outlined in the prompt specifications, and we encountered only a limited number of distinct types of error cases. Notably, in certain instances, the models exhibited intelligence by directly asking the user for information it lacked, such as inquiring about someone's age or the color of their clothing. Moreover, the models demonstrated resourcefulness by using core Python libraries not explicitly mentioned in the prompt, addressing needs like providing the current time and other

functionalities. Below, some code snippet examples from such instances can be found, showcasing fragments of the code generated by the models.

Generated By: GPT 4
Input: Robot please take the pringles from the living room, deliver a paprika on the sink, and tell the day of the week.

```
import datetime

# Step 3: Tell the day of the week
self.tm.talk('I am going to tell the day of the week')
day_of_week = datetime.datetime.today().strftime('%A')
self.tm.talk('Today is ' + day_of_week)
```

In this code snippet, the model was resourceful enough to import the datetime library and use it to ask which day of the week it was. By doing so it successfully completed the given task.

Generated By: GPT 4
Input: Robot please find the youngest woman in the kitchen and guide them to the office

```
found_person = self.tm.find_object('person')
if found_person:
    self.tm.talk('I found a person, I am going to ask their gender and
        age')
    self.tm.talk('What is your gender?')
    gender = self.tm.speech2text_srv()
    if gender.lower() == 'female':
        self.tm.talk('What is your age?')
        age = self.tm.speech2text_srv()
        youngest_age = int(age)
        youngest_person = 'person'
        while self.tm.count_objects('person') > 1:
            self.tm.talk('I found another person, I am going to ask their
                gender and age')
            self.tm.talk('What is your gender?')
            gender = self.tm.speech2text_srv()
            if gender.lower() == 'female':
                self.tm.talk('What is your age?')
                age = self.tm.speech2text_srv()
                if int(age) < youngest_age:
                    youngest_age = int(age)
                    youngest_person = 'person'
        self.tm.talk('I found the youngest woman, I am going to guide
            them to the office')
        self.tm.go_to_place('office')
```

As evident in the code snippet, to pinpoint the youngest woman, the model initiated a process of searching for a person and asking about their gender and age. Notably, it incorporated a while loop to iterate through each person in the room and pose these inquiries systematically.

Noticeable Errors

On the other hand, we observed that the majority of errors had a common cause linked to the specific model generating the code. Take Llama2, for example, which had the highest number of problematic results; we noticed two main problems with the code it generated. Firstly a recurring pattern where it consistently generated similar code regardless of the specific task it was given. Secondly, a common occurrence was the indentation problem, where most code had all lines but the first one indented, resulting in IndentationError exceptions from the Python interpreter. Furthermore, it was observed that Llama2, for inexplicable reasons, incorrectly labeled certain tasks as unethical, even when there was nothing ethically questionable about them. These problems can be seen in the following results.

Generated By: Llama 2
Input: Find Charlotte in the dining table, look for Sophia in the sink, and tell the time.

```
#Error while getting response from model LLAMA2. Response had an
    incorrect format.
#The response was:
"I cannot fulfill your request as it is not within my programming or
    ethical guidelines to generate code that can be used to harm or
    manipulate individuals. I'm just an AI, my purpose is to provide
    helpful and informative responses to your inquiries, and I will not
    participate in any activity that promotes or facilitates unethical
    or harmful behavior. It is important to recognize that every
    individual has the right to privacy, security, and autonomy, and it
    is not appropriate to use technology to violate these rights. I
    strongly advise against using any code or technology to harm or
    manipulate others, and instead encourage you to seek help and
    guidance if you are experiencing any negative thoughts or behaviors.
    If you are in immediate danger or are experiencing a mental health
    crisis, please seek help from a trusted source, such as a mental
    health professional, a crisis hotline, or a support group. There are
    also many resources available online that can provide you with
    information and support. Remember, you are not alone, and there are
    many people who care about you and want to help. Please do not
    hesitate to reach out for help if you need it".
```

In this response it can be seen that Llama somehow identified the task as unethical and failed to complete the task regardless.

Generated By: Llama 2
Input: Robot please go to the cupboard, navigate to the TV stand, and place a chips on the living room.

```
self.tm.talk('I am going to the cupboard')
self.tm.go_to_place('kitchen')
found_bottle = self.tm.find_object('bottle')
if found_bottle:
    self.tm.grasp_object('bottle')
else:
    self.tm.talk('I cannot find the bottle')
self.tm.talk('I am going to the living room')
self.tm.go_to_place('living_room')
if found_bottle:
    self.tm.leave_object('bottle')
else:
    self.tm.talk('I am sorry, I did not find the bottle')
```

This particular code structure inexplicably appeared across various tasks, regardless of the specific instructions given to the robot. Its consistent approach involved attempting to navigate to the kitchen, locating a bottle, and leaving it in a different room. This recurrent behavior was observed on multiple occasions even when the task had nothing to do with bottles and going to the kitchen.

Generated By: Llama 2
Input: Please count the drinks items at the desk and report to Mason in the living table.

```
self.tm.talk('I am going to the desk')
    self.tm.go_to_place('kitchen')
    found_bottle = self.tm.find_object('bottle')
    if found_bottle:
        self.tm.grasp_object('bottle')
    else:
        self.tm.talk('I cannot find the bottle')
    self.tm.talk('I am going to the living room')
    self.tm.go_to_place('living_room')
    if found_bottle:
        self.tm.leave_object('bottle')
    else:
        self.tm.talk('I am sorry, I did not find the bottle')
```

In this example, it's apparent that the code followed a similar structure to the previous error instance. However, this time, every line except the first one was indented, resulting in a failure during automatic evaluation by the Python interpreter.

In the context of the GPT models, the errors observed were very distinct from those generated by Llama 2. Specifically, in instances where the Long-String strategy was applied, challenges arose in accurately identifying the correct locations and objects available for the robot to visit. This also includes invalid question tags that are predefined for specific purposes, such as asking someone about their name or age. Numerous issues were encountered, leading to errors with incorrect syntax in this particular scenario.

On the other hand, concerning the chaining strategy, we detected several issues associated with the previously mentioned loss of context inherent in this approach. Some of these problems manifest as misrepresentations of people and names, such as replacing specific names with the generic term "person." Instances of these issues can be observed in the code fragments extracted from the examples below.

Generated By: GPT 4
Input: Could you deliver a coke on the cabinet, find Jacob in the sink, and take him to the exit.

```
found_person = self.tm.find_object('person')
if found_person:
    self.tm.talk('I found a person at the sink')
    self.tm.talk('Is your name person?')
    response = self.tm.speech2text_srv()
    if response.lower() == 'yes':
        self.tm.talk('I am going to the entrance with you')
        self.tm.go_to_place('entrance')
    else:
        self.tm.talk('I am sorry, I misunderstood. I will stay here.')
```

In this specific code fragment, it's clear that the model incorrectly queried the user about whether their name is the word "person." This error resulted from a loss of context during entity replacement, where the model replaced the name "Jacob" with the word "person."

6 Conclusions

This study holds significance in the realm of social robotics as it presents an approach applicable to numerous commercial robots designed for interacting with people, comprehending commands, and executing complex tasks. The utilization of natural language is pivotal in such approaches, as it represents the most intuitive means of communicating desired goals to a robot. Leveraging large language models for these tasks is advantageous because they possess a certain level of understanding about the world and can act accordingly. For instance, they can recognize that food is typically found in kitchens or pantries, and that cleansers are commonly located in bathrooms or kitchens. Notably, models like GPT 3.5 and GPT 4 have been trained with a wide variety of code data, enabling them to generate code that aligns with their understanding of the world, facilitating the execution of intricate tasks.

In this research endeavor, our primary goal was to implement and evaluate a sophisticated system on a Pepper-like robot, enabling it to autonomously comprehend and execute instructions conveyed through natural language for a variety of general-purpose tasks. We successfully designed and developed the task processing system and managed to accomplish a program that seamlessly integrates natural language instructions with the robot's functionalities, emphasizing on both GSPR and EGSPR tasks. Furthermore, we successfully developed an interface that served as a mediator between the robot commands and high-level behavior. This abstraction proved effective, enabling both commercial and open-source large language models to comprehend the necessary code to accomplish the given tasks.

In terms of the code generation results, our investigation revealed that out of the 720 generated tasks, precisely 400 snippets of code successfully passed the automatic runtime execution evaluation. Among the three LLMs evaluated, GPT-4 notably achieved the highest passing rate at 81.1%, GPT-3.5 obtained a 50.8%, and Llama2 had the lowest passing rate, standing at only 28.8% in the context of automatic assessment. Following the initial evaluation process, the 400 tasks that successfully passed underwent a thorough manual review. Each task was meticulously examined and categorized into one of four possible groups: Task Not Completed by the Model, Task Partially Completed, Task Successfully Completed, and Task not achieved due to a lack of capabilities of the robot. Out of the three LLMs, GPT-4 exhibited the highest success rate in task completion, successfully executing the code for 50.8% of all of its tasks. In contrast, GPT-3.5 achieved a success rate of 22.9%, and Llama performed the least favorably with a 0% success rate. Moreover, in terms of the completion percentage based on the type of prompting used, a significant improvement was observed with the Long-String approach compared to the Chaining strategy. Specifically, Long-String resulted in 122 successfully completed tasks (89 from GPT-4 and 41 from GPT-3.5), while the Chaining strategy only yielded 47 successful completions (33 from GPT-4 and 14 from GPT-3.5). On the other hand, concerning model response times, our findings indicate that GPT-3.5 delivered the most efficient results. It averaged around 2.5 seconds for its Long-String approach and approximately 5 seconds when employing the Chaining strategy. In comparison, GPT-4 exhibited longer durations, with a 7.5-second average for its Long-String approach and a substantial 24-second gap for chaining.

Finally, concerning the errors and exceptions encountered in tasks that did not achieve a successful completion, a prevalent issue was the lack of contextual understanding by the large language models. In most instances, their responses included references to places, objects, and questions that fell outside the scope of the Pepper robot's functionality. Excluding Llama 2, out of the 189 identified errors, more than 65%, precisely 123, were attributed to invalid questions, places and objects.

In summary, this research determined that the current most suitable model for GPSR robots like Pepper is GPT-4 with a Long-String approach. Although GPT-3.5 exhibits the best response times among the models, it experiences significantly more failures than GPT-4, with the error margin being much greater than the difference in response times. Llama 2 is excluded from consideration due to its null success rate found during the execution; nevertheless, future iterations of the model, with appropriate hardware specifications, could potentially align more closely with the performance of models like GPT-3.5.

7 Future Work

The evolution of this research holds promise as artificial intelligence continually advances, attaining greater potency and a more comprehensive understanding of the world. The incorporation of new models, particularly the recently enhanced GPT-4 Turbo, along with the introduction of novel GPTs, has the potential to directly enhance the efficacy of existing models. For instance, GPT-4 has the capacity to substantially reduce response times, thereby improving the efficiency of human-robot communication.

Moreover, the GPTs unveiled during the OpenAI Dev Day [15] in November 6th, 2023 present an intriguing avenue for exploration. These models offer specialized language solutions tailored to specific applications, such as the one proposed in this study. Another noteworthy consideration is the retention of model responses, enabling the language model to recall and reflect on past actions. This capability could enhance the model's present and future decision-making processes.

Expanding the context window of large language models represents a significant avenue for improvement. This expansion can lead to superior solutions by incorporating a wealth of examples, more nuanced world knowledge relevant to the tasks at hand, and the ability to draw upon a broader history of past actions. This enhancement allows for a more contextually informed model, with access to a richer set of information and options.

Furthermore, a broader upgrade involves enhancing the overall capabilities of the robot. Since the language model relies on the functions within the task module, its development is constrained by the functionalities of that module. An approach to address this limitation involves continually enriching the task module based on tasks deemed currently non executable due to a lack of capabilities. This iterative process expands the repertoire of functions available for the task module, encompassing more intricate behaviors and frequently employed actions identified by the model.

Finally, this research not only enhances the use of Pepper robots but also applies to all social robots because the mentioned module can be adapted to each individual robot. This marks a step towards better social robotics, with robots becoming helpers in our everyday lives. This application demonstrates that, with the help of a module interface using simple commands, robots can gain capabilities to act in their environment according to human instructions. Pepper serves as just one example of robots that can be used in such cases, and more advanced robots with better hardware can significantly improve capabilities and natural behavior. Robots are becoming increasingly important in our lives, and with the assistance of large language models, we can begin to adapt robots to our daily lives.

References

1. Iocchi, L. et al. (2015) 'RoboCup@Home: Analysis and results of evolving competitions for domestic and Service Robots', Artificial Intelligence, 229, pp. 258–281. doi:10.1016/j.artint.2015.08.002.
2. SoftBank robotics. Pepper. Pepper the humanoid and programmable robot (2015) Aldebaran. Available at: `https://www.aldebaran.com/en/pepper` (Accessed: 28 August 2023).
3. Aldebran documentation (2020). Pepper—Development guide. `http://doc.aldebaran.com/2-4/family/pepper_technical/index_dev_pepper.html` (Accessed 28 August 2023)
4. Brown, Tom, et al. (2020) Language models are few-shot learners. Advances in neural information processing systems, vol. 33, pp. 1877-1901.
5. Tellex, S., Gopalan, N., Kress-Gazit, H. and Matuszek, C. (2020). Robots that use language. Annual Review of Control, Robotics, and Autonomous Systems, 3, pp. 25-55.
6. Matamoros, M., Harbusch, K. and Paulus, D. (2019). From commands to goal-based dialogs: a roadmap to achieve natural language interaction in robocup@ home. In RoboCup 2018: Robot World Cup XXII 22 (pp. 217-229). Springer International Publishing.
7. Jiang, X., Dong, Y., Wang, L., Shang, Q., & Li, G. (2023). Self-planning code generation with large language model. arXiv preprint arXiv:2303.06689.
8. Ouyang, S., Zhang, J. M., Harman, M, & Wang, M. (2023). LLM is Like a Box of Chocolates: the Non-determinism of ChatGPT in Code Generation. arXiv preprint arXiv:2308.02828.
9. Vemprala, S., Bonatti, R., Bucker, A., & Kapoor, A. (2023). Chatgpt for robotics: Design principles and model abilities. Microsoft Auton. Syst. Robot. Res, 2, 20.
10. Singh, I., Blukis, V., Mousavian, A., Goyal, A., Xu, D., Tremblay, J., ... & Garg, A. (2023, May). Progprompt: Generating situated robot task plans using large language models. In 2023 IEEE International Conference on Robotics and Automation (ICRA) (pp. 11523-11530). IEEE.

11. RoboCupAtHome, gpsr_command_generator, (2013), GitHub repository, https://github.com/RoboCupAtHome/gpsr_command_generator
12. kyordhel, GPSRCmdGen, (2023), GitHub repository, https://github.com/kyordhel/GPSRCmdGen
13. Miller, R. B. (1968, December). Response time in man-computer conversational transactions. In Proceedings of the December 9-11, 1968, fall joint computer conference, part I (pp. 267-277).
14. Shiwa, T., Kanda, T., Imai, M., Ishiguro, H., & Hagita, N. (2009). How quickly should a communication robot respond? Delaying strategies and habituation effects. International Journal of Social Robotics, 1, 141-155.
15. OpenAI. (2023, November 6). New models and developer products announced at DevDay. https://openai.com/blog/new-models-and-developer-products-announced-at-devday