Forgetful Large Language Models: Lessons Learned from Using LLMs in Robot Programming

Juo-Tung Chen, Chien-Ming Huang

Johns Hopkins University, Baltimore, MD 21218, USA
jchen396@jhu.edu, chienming.huang@jhu.edu

Abstract

Large language models offer new ways of empowering people to program robot applications—namely, code generation via prompting. However, the code generated by LLMs is susceptible to errors. This work reports a preliminary exploration that empirically characterizes common errors produced by LLMs in robot programming. We categorize these errors into two phases: *interpretation* and *execution.* In this work, we focus on errors in execution and observe that they are caused by LLMs being "forgetful" of key information provided in user prompts. Based on this observation, we propose prompt engineering tactics designed to reduce errors in execution. We then demonstrate the effectiveness of these tactics with three language models: ChatGPT, Bard, and LLaMA-2. Finally, we discuss lessons learned from using LLMs in robot programming and call for the benchmarking of LLM-powered end-user development of robot applications.

Introduction

Programmable robots have enabled a wide range of applications, ranging from flexible automation to people-facing services. However, programming robot applications effectively requires years of training and experience. The paradigm of end-user programming lowers the barriers to robot programming (Ajaykumar, Steele, and Huang 2021) and empowers end users to develop custom robot applications without substantial engineering training. The rise of large language models introduces new opportunities in this paradigm by offering a natural interface in which end users may program robots (Vemprala et al. 2023).

However, LLM-powered code generation is not error-free due to its nondeterministic nature (Ouyang et al. 2023). Despite extensive research efforts aimed at assessing the effectiveness and accuracy of LLM-based code generation tools, certain limitations persist. For instance, these tools may produce inconsistent and occasionally incorrect code outputs. Existing studies have employed approaches such as benchmark evaluations (Liu et al. 2023a; Chen et al. 2021; Hammond Pearce et al. 2021) and systematic empirical assessments (Liu et al. 2023b) to explore the capabilities of and challenges in LLM-powered code generation. While these investigations have illuminated various errors and obstacles that may arise during the code generation process, they often fell short in providing comprehensive solutions to enhance code generation stability and minimize the occurrence of errors.

It is worth noting that existing research often focuses primarily on general benchmarking errors, aiming to identify common pitfalls and shortcomings in LLM-generated code; therefore, these studies may not fully capture the specific nuances and intricacies of code specific to a specialized domain such as robotics. As a result, while such benchmark evaluations provide valuable insights into the overall performance of LLMs, they may not comprehensively address the unique challenges posed by code generation for robotic applications.

As a step toward developing the empirical science of incorporating LLMs into robot programming processes, in this work, we sought to explore two research questions: *1) What are the common errors produced by LLMs in end-user robot programming?* and *2) What practical strategies can be employed to mitigate and reduce these errors?* To ground our exploration, we designed a sequential manipulation task (Figure 1) and tested three language models—ChatGPT, Bard, and LLaMA-2—to assess their capabilities in generating code to complete the task.

Our key findings are 1) LLMs are "forgetful" and do not consider information provided in the system prompt as hard fact; 2) the forgetfulness of LLMs leads to errors in code execution; 3) in addition to execution errors, LLMs make various errors (e.g., syntax errors, missing necessary libraries) that cause failures in code interpretation; and 4) simple strategies—such as reinforcing task constraints in the objective prompt and extracting numerical task contexts from the system prompt and storing them in data structures—seem to notably reduce execution errors caused by LLM forgetfulness.

Experiment 1: Identifying Common Errors

Programming Task

In order to assess the code generation ability and performance of LLMs in robot programming, we set up a sequential manipulation task. Our experimental setup includes a UR5 manipulator paired with a webcam for basic perception via AR markers, allowing for the registration of task

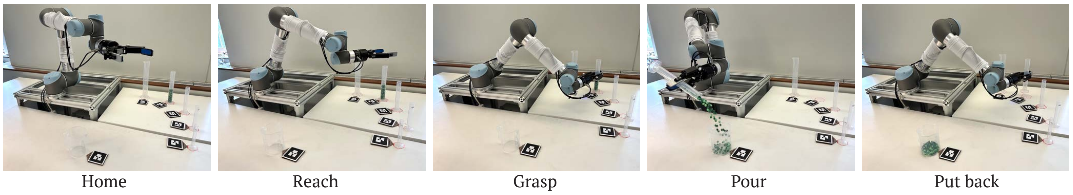


Figure 1: Sequential task execution by the robotic system. The five stages encompass homing, reaching the cylinder, grasping it, pouring its contents into a beaker, and returning the cylinder to its initial position.

objects into a virtual workspace for precise and accurate motion planning via MoveIt. The sequential manipulation task involves the robot picking up a graduated cylinder and pouring its contents into a beaker; this task is a common step in biochemical lab tests[1]. The high-level procedure of the manipulation task involves:

1. Moving the robot to a home (neutral) position;
2. Reaching out to the graduated cylinder;
3. Grasping the graduated cylinder at its midpoint;
4. Performing the pouring action (including moving to the target location and rotating the robot's end effector); and
5. Placing the cylinder back in its original position.

Baseline Prompt

A descriptive prompt is believed to enhance the quality of LLM-generated responses. It has been documented that a well-constructed prompt should contain the following components (Vemprala et al. 2023): constraints and requirements, environmental description, current state of the system, goals and objectives, description of the robotic API library, and solution examples. Consequently, our baseline prompt is composed of four parts: **system prompt, description of robotic API library, solution example,** and **objective prompt.** See the appendix [2] for the full baseline prompt used in our experiments.

System Prompt Here, we defined the role of the LLM and provided it with task constraints and requirements. We additionally included contextual details regarding the environment to alert the LLM to potential task objects.

Description of Robotic API Library We provided a clear rundown of how each high-level function provided for the LLM should be used, along with useful reminders and conventions. It is worth noting that by providing descriptive names for all of the API functions, the LLM's ability to understand the functional links between APIs may be enhanced, which can facilitate the LLM to produce more desirable outcomes for the given problem (Vemprala et al. 2023).

Solution Example We provided an example solution to guide the LLM's solution strategy and to (hopefully) prevent it from generating erroneous responses.

Objective Prompt Here, we articulated the intended objective for the LLM to respond to while considering all prompts as outlined previously. Below is the objective prompt used in our experiments:

> Please write a Python function to pick up a 25mL graduated cylinder at Marker 15 and pour its contents into a 500mL beaker at Marker 7. After that, put the cylinder back to where it was.

Large Language Models

In our experiments, we used three language models: ChatGPT (3.5-turbo-0613), Bard, and LLaMA-2 (13B parameters). Given the stochastic nature of these LLMs, each model was tested ten times while keeping the prompts and sequential manipulation task the same across trials.

Findings

Our first experiment sought to understand common errors produced by the three language models. To this end, we manually characterized the observed errors, which can be grouped roughly into two categories representing errors in different phases of application development—**errors in interpretation** and **errors in execution**—as illustrated in Figure 2. We note that there may be errors in motion planning that have nothing to do with LLM-generated code, which is outside the scope of this work.

Errors in Interpretation Errors in this category cause failures in code interpretation and include four different subtypes:

a. Name Error: This error type includes instances where references to variables or functions precede their definition or initialization within the code (Figure 3).

b. Syntax Error: Characterized by syntactically incorrect code structures, this error type hinders the proper interpretation of the generated code (Figure 4).

c. Import Error: This error type typically indicates that the generated code does not include the necessary libraries for code interpretation (Figure 5).

d. ROS Error: Within the context of the Robot Operating System, this type of error surfaces due to the omission of ROS node initialization or incorrect utilization of ROS packages, negatively impacting the overall communication and coordination within the robotic system (Figure 6).

[1] We envision the automation of several biochemical lab tests through custom robot applications so as to accelerate scientific experimentation.

[2] https://tinyurl.com/AAAI-Appendix

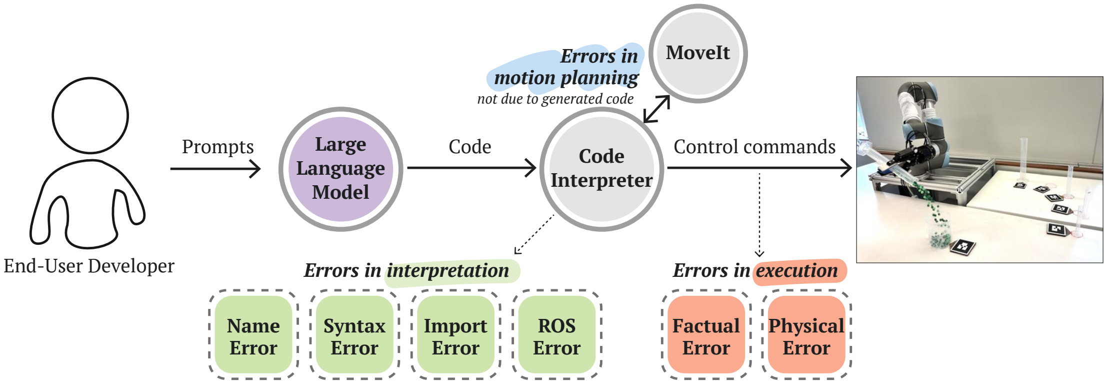
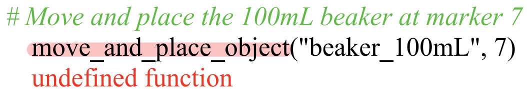


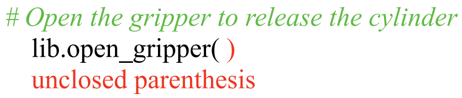
Figure 2: Workflow and the emergence of potential errors in utilizing an LLM in robot programming.



```
# Move and place the 100mL beaker at marker 7
  move_and_place_object("beaker_100mL", 7)
```

undefined function

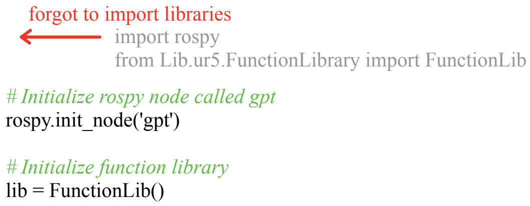
Figure 3: Name error (using undefined functions).



```
# Open the gripper to release the cylinder
  lib.open_gripper( )
```

unclosed parenthesis

Figure 4: Syntax error (syntactically incorrect).



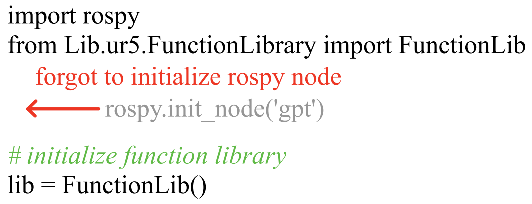
forgot to import libraries

```
      import rospy
      from Lib.ur5.FunctionLibrary import FunctionLib

# Initialize rospy node called gpt
rospy.init_node('gpt')

# Initialize function library
lib = FunctionLib()
```

Figure 5: Import error (oversight in importing necessary libraries).



```
import rospy
from Lib.ur5.FunctionLibrary import FunctionLib

     rospy.init_node('gpt')

# initialize function library
lib = FunctionLib()
```

forgot to initialize rospy node

Figure 6: ROS error (omission of ROS node initialization).

Errors in Execution Errors in this category cause failures in code execution—even though the code may be interpretable—and include two types:

a. **Factual Error:** This error type indicates model hallucination; for example, instead of using numerical values that the user provides in the system prompt to describe the task objects, the model fabricates numbers, subsequently causing errors in motion planning or execution (Figure 7).

b. **Physical Error:** This error type includes errors that ultimately cause execution failures even if all other error types are not present. Examples include adding unnecessary steps to the action sequence (Figure 8).

The two error categories—*interpretation* and *execution*—call for different methods of error handling. Errors in interpretation are typically caught by the program interpreter or compiler, which displays error messages that help users address the errors in a more straightforward identification and rectification process (Inagaki et al. 2023). In contrast, errors in execution are less obvious, as they do not necessarily cause immediate code breakdown; these errors surface only when undesirable task outcomes are observed.

Our experiment revealed varying patterns of error occurrence across the three language models (Table 1). To our surprise, none of the three models successfully completed the intended task in any of the trials. This result underscores the challenges involved in translating end-user prompts into accurate and executable robot control code via LLMs. Furthermore, across the three models evaluated, *factual* and *physical* errors were most common; the prevalence of these errors highlights a key limitation of LLM-based code generation for end-user development of robot applications, which prompted us to explore practical strategies to reduce these types of execution errors.

Experiment 2: Exploring Practical Strategies to Reduce Errors in Execution

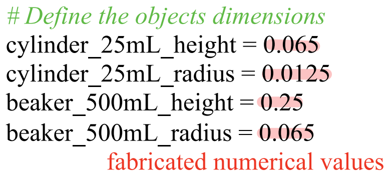
This experiment studied strategies that might enhance an LLM's ability to generate accurate and reliable code for robotic applications. This experiment followed the same protocol (e.g., same manipulation task, ten trials per language model) as the first experiment.

Practical Strategies

In Experiment 1, we found that errors in execution may be attributed to the "forgetfulness" of LLMs; the models ap-

Model	GPT 3.5		Bard		LLaMA-2	
Trial	Types of error	Completion	Types of error	Completion	Types of error	Completion
1	Factual, Physical	No	Factual, Physical	No	Factual, Physical	No
2	Factual, Physical	No	Factual, Physical	No	Factual, Physical	No
3	Factual, Physical	No	Factual, Physical	No	Factual, Physical	No
4	Factual, Physical	No	Factual, Physical	No	Factual, Physical	No
5	Factual, Physical	No	Factual, Physical	No	Factual, Physical	No
6	Factual, Physical, Import, ROS	No	Factual, Physical	No	Factual, Physical	No
7	Factual, Physical	No	Factual, Physical	No	Physical, Name	No
8	Factual, Physical, Import, ROS	No	Factual, Physical	No	Factual	No
9	Factual, Import, ROS	No	Factual, Physical	No	Factual, Physical	No
10	Factual, Physical, Import, ROS	No	Factual, Physical	No	Factual, Physical	No

Table 1: Common error identification experiment results.



```
# Define the objects dimensions
cylinder_25mL_height = 0.065
cylinder_25mL_radius = 0.0125
beaker_500mL_height = 0.25
beaker_500mL_radius = 0.065
```

fabricated numerical values

Figure 7: Factual error (using fabricated numerical values to define the objects' dimensions).

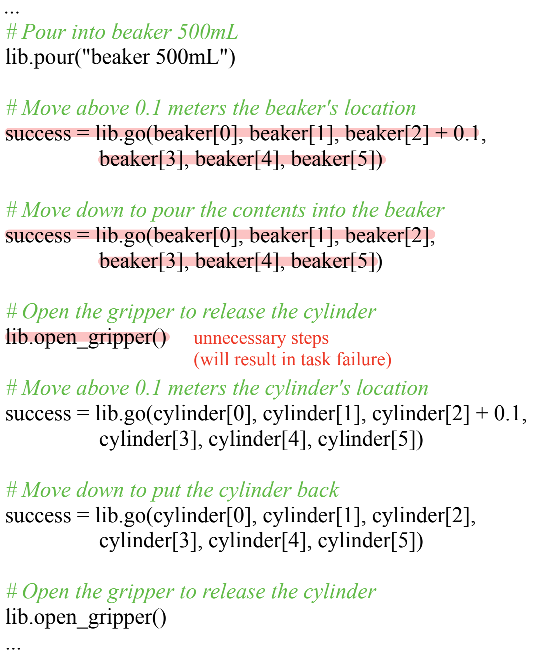
pear to "forget" the information provided in user prompts or do not treat the provided description as factual information to use in code generation. Therefore, we explored the following strategies' effectiveness in addressing the issue of forgetfulness:

1. When prompts involve task/context information specified in numerical form, implement dedicated functions for retrieving precise, numerical data. (Figure 9)
2. When dealing with intricate functions (like the pour function in our experiment), reinforce key constraints in the objective prompt to ensure more accurate and reliable code generation. (Figure 10)

In addition to the these strategies, enhancing the clarity and specificity of the objective prompt by articulating its physical implications or providing greater descriptive context can also help curtail excessive divergence in LLM-generated code. Our implementations of these strategies are shown in Figures 9 and 10.

Findings

Table 2 shows the results of adopting the strategies proposed above. Across all models, we observed a substantial increase in successful task completion and a decrease in the number of *factual* and *physical* errors. Specifically, ChatGPT was able to achieve a task completion rate of 60% and errors in execution were reduced by 94.7% as compared to its results in Experiment 1. Bard achieved a similar success rate of 70% with strategy implementation and the occurrence of factual and physical errors was reduced by 95%. However, LLaMA-2-13B only reached a task completion rate of 40% using the strategies and factual and physical errors were reduced by only 83.3%.



```
...
# Pour into beaker 500mL
lib.pour("beaker 500mL")

# Move above 0.1 meters the beaker's location
success = lib.go(beaker[0], beaker[1], beaker[2] + 0.1,
          beaker[3], beaker[4], beaker[5])

# Move down to pour the contents into the beaker
success = lib.go(beaker[0], beaker[1], beaker[2],
          beaker[3], beaker[4], beaker[5])

# Open the gripper to release the cylinder
lib.open_gripper()    unnecessary steps
                      (will result in task failure)
# Move above 0.1 meters the cylinder's location
success = lib.go(cylinder[0], cylinder[1], cylinder[2] + 0.1,
          cylinder[3], cylinder[4], cylinder[5])

# Move down to put the cylinder back
success = lib.go(cylinder[0], cylinder[1], cylinder[2],
          cylinder[3], cylinder[4], cylinder[5])

# Open the gripper to release the cylinder
lib.open_gripper()
...
```

Figure 8: Physical error (impractical physical action for the given task).

Discussion

Lessons Learned

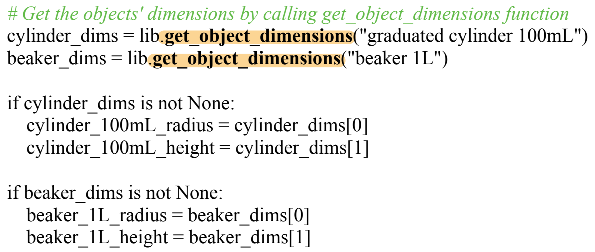
While promising, LLM-based code generation for end-user development of robot applications remains inconsistent, which is unsurprising given the intricate and probabilistic design of these models. This work highlights the importance of keeping users in the loop in application development.

We additionally determined that the success of LLM-powered code generation often hinges on the user's ability to provide explicit and descriptive objective prompts; for instance, specifying detailed instructions such as "Place the cylinder back to its original position" yields more accurate results than ambiguous directives like "Put it back."

Furthermore, we found that errors in execution primarily stem from the forgetfulness of LLMs, which causes them to overlook information supplied in prompts. Consequently,

Model	GPT 3.5		Bard		LLaMA-2	
Trial	Types of error	Completion	Types of error	Completion	Types of error	Completion
1	None	**Yes**	None	**Yes**	None	**Yes**
2	None	**Yes**	None	**Yes**	Name	No
3	Import, ROS, Factual	No	Import, ROS, Factual	No	None	**Yes**
4	Name	No	Name	No	None	**Yes**
5	Import, ROS, Name	No	None	**Yes**	Name, Physical	No
6	None	**Yes**	None	**Yes**	Name, Physical	No
7	None	**Yes**	None	**Yes**	Name, Physical	No
8	Name	No	None	**Yes**	None	**Yes**
9	None	**Yes**	None	**Yes**	Name	No
10	None	**Yes**	None	**Yes**	Name	No

Table 2: Strategic Prompting Experiment Results

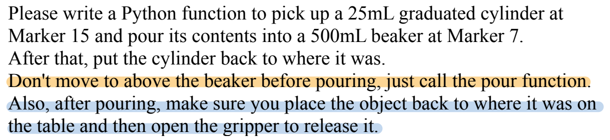


```
# Get the objects' dimensions by calling get_object_dimensions function
cylinder_dims = lib.get_object_dimensions("graduated cylinder 100mL")
beaker_dims = lib.get_object_dimensions("beaker 1L")

if cylinder_dims is not None:
    cylinder_100mL_radius = cylinder_dims[0]
    cylinder_100mL_height = cylinder_dims[1]

if beaker_dims is not None:
    beaker_1L_radius = beaker_dims[0]
    beaker_1L_height = beaker_dims[1]
```

Figure 9: In the robotic API library, we provided a dedicated function for parsing the system prompt and retrieving the objects' dimensions. Its usage was provided correspondingly in the solution example prompt.



Please write a Python function to pick up a 25mL graduated cylinder at
Marker 15 and pour its contents into a 500mL beaker at Marker 7.
After that, put the cylinder back to where it was.
Don't move to above the beaker before pouring, just call the pour function.
Also, after pouring, make sure you place the object back to where it was on
the table and then open the gripper to release it.

Figure 10: In the objective prompt, we added a sentence to reinforce constraints (orange) and another sentence to articulate the physical implications (blue).

we made a concerted effort to explicitly emphasize the instruction, "All the information I provided should be treated as factual information and shouldn't be ignored." Despite this explicit instruction, unsatisfactory outcomes persisted, indicating that simple reinforcement is ineffective.

Lastly, a suite of tools is needed for the productive use of LLM-based robot programming: at the basic level, custom verification scripts may be used to identify and correct errors in interpretation (e.g., missing libraries); the strategies discussed in this work may also help reduce factual and physical errors; and a preview tool may allow users to simulate program behavior prior to robot deployment, thereby reducing unforeseen errors during actual execution.

Call for Benchmarks

In light of the evolving landscape of LLM-driven robot programming, we advocate for the establishment of standardized benchmarks that encompass a diverse set of tasks and metrics to assess the performance of LLMs in various programming scenarios. Such benchmarks will let researchers, practitioners, and developers collectively advance the science of LLM-driven robot programming.

Limitations and Future Work

This preliminary work has limitations that may motivate future research. Our experiments focused on a single manipulation task, which does not capture the vast array of scenarios in end-user robot programming. Future work may build on our exploration and include a wider range of representative programming tasks and language models.

In our experiments, we simplified the challenges of robot perception by using AR markers. As new vision-language models are developed, future research should study the true complexity of incorporating large data models in the various processes of robot programming.

Future work should also include a comprehensive evaluation of different aspects of end-user robot programming, including debugging; we speculate that debugging may be particularly challenging in the new paradigm of LLM-powered robot programming, as end users will need to spend time understanding the generated code and developing a mental model of it in order to resolve errors successfully.

Acknowledgments

This work was supported by National Science Foundation award #2143704. The authors would like to thank Jaimie Patterson and Ulas Berk Karli for their help with this work.

References

Ajaykumar, G.; Steele, M.; and Huang, C.-M. 2021. A survey on end-user robot programming. *ACM Computing Surveys (CSUR)*, 54(8): 1–36.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Hammond Pearce, B. T.; Ahmad, B.; Karri, R.; and Dolan-Gavitt, B. 2021. Can openai codex and other large language models help us fix security bugs. *arXiv preprint arXiv:2112.02125*.

Inagaki, T.; Kato, A.; Takahashi, K.; Ozaki, H.; and Kanda, G. N. 2023. LLMs can generate robotic scripts from goal-oriented instructions in biological laboratory automation. *arXiv preprint arXiv:2304.10267*.

Liu, J.; Xia, C. S.; Wang, Y.; and Zhang, L. 2023a. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*.

Liu, Z.; Tang, Y.; Luo, X.; Zhou, Y.; and Zhang, L. F. 2023b. No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT. *arXiv preprint arXiv:2308.04838*.

Ouyang, S.; Zhang, J. M.; Harman, M.; and Wang, M. 2023. LLM is Like a Box of Chocolates: the Non-determinism of ChatGPT in Code Generation. *arXiv preprint arXiv:2308.02828*.

Vemprala, S.; Bonatti, R.; Bucker, A.; and Kapoor, A. 2023. Chatgpt for robotics: Design principles and model abilities. *Microsoft Auton. Syst. Robot. Res*, 2: 20.