IEEE Robotics and Automation Letters (RAL) paper presented at
2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 14-18, 2024. Abu Dhabi, UAE

Deploying and Evaluating LLMs to Program Service Mobile Robots

Zichao Hu[1], Francesca Lucchetti[2], Claire Schlesinger[2], Yash Saxena[1], Anders Freeman[3],
Sadanand Modak[1], Arjun Guha[2], Joydeep Biswas[1]

Abstract*—Recent advancements in large language models (LLMs) have spurred interest in using them for generating robot programs from natural language, with promising initial results. We investigate the use of LLMs to generate programs for service mobile robots leveraging mobility, perception, and human interaction skills, and where *accurate sequencing and ordering* of actions is crucial for success. We contribute CODEBOTLER, an open-source robot-agnostic tool to program service mobile robots from natural language, and ROBOEVAL, a benchmark for evaluating LLMs' capabilities of generating programs to complete service robot tasks. CODEBOTLER performs program generation via few-shot prompting of LLMs with an embedded domain-specific language (eDSL) in Python, and leverages skill abstractions to deploy generated programs on any general-purpose mobile robot. ROBOEVAL evaluates the correctness of generated programs by checking execution traces starting with multiple initial states, and checking whether the traces satisfy temporal logic properties that encode correctness for each task. ROBOEVAL also includes multiple prompts per task to test for the robustness of program generation. We evaluate several popular state-of-the-art LLMs with the ROBOEVAL benchmark, and perform a thorough analysis of the modes of failures, resulting in a taxonomy that highlights common pitfalls of LLMs at generating robot programs. We release our code and benchmark at https://amrl.cs.utexas.edu/codebotler/.

I. INTRODUCTION

We are interested in deploying service mobile robots to perform arbitrary user tasks from natural language descriptions. Recent advancements in large language models (LLMs) have shown promise in related applications involving visuomotor tasks [1], [2], planning [3]–[6], and in this paper, we investigate the use of LLMs to generate programs for *service mobile robots* leveraging mobility, perception, and human interaction skills, where accurate *sequencing and ordering of actions* is crucial for success. We contribute CODEBOTLER and ROBOEVAL: CODEBOTLER is an open-source robot-agnostic tool to generate general-purpose service robot programs from natural language, and ROBOEVAL is a benchmark for evaluating LLMs' capabilities of generating programs to complete service robot tasks.

CODEBOTLER leverages an embedded domain-specific language (eDSL) in Python to abstract key robot skills, and includes robot-agnostic bindings for such tasks using ROS Actions [7]. Given few-shot examples, CODEBOTLER uses LLMs to convert natural language task descriptions into programs in the eDSL, which are then executed on real robots using a lightweight interpreter to interface with robot-specific skills.

While the capabilities of LLMs at producing robot programs are impressive, they are still susceptible to a variety of failures. To understand empirically the failure modes of LLMs in producing robot programs, we introduce the ROBOEVAL benchmark. Given a program generated by CODEBOTLER, ROBOEVAL first executes it in a symbolic simulator to generate multiple program traces from different initial world states, and then checks these traces against a set of temporal checks that define correct behavior for the task for each initial world state.

Existing code completion benchmarks tackle simple data processing functions [8] or low-level robot skills [1], which are amenable to simple input-output unit tests. However, code generation for general-purpose service robot programs cannot be evaluated just on input/output sequences. For example, given a task *"Check how many conference rooms have no markers"*, it is insufficient to just test whether the LLM-generated program executes to state the correct answer — to ensure correctness, a correct program must first visit all conference rooms and check for markers there before arriving at the result. We also observe that there are significant variations in the correctness of generated programs with small variations in the phrasing of the natural language task descriptions [9]. We thus contribute three key ideas to test for both correctness and robustness of LLM-generated robot programs: 1) we evaluate the *execution traces* of programs; 2) we check whether the execution traces satisfy *temporal logic* properties that encode correctness for each task; and 3) we *vary the prompts* and to test for robustness. Fig. 1 shows the system diagram of CODEBOTLER and ROBOEVAL.

We further categorize the types of failures of different LLMs in the ROBOEVAL benchmark and find several common categories of failures, including Python run-time errors, errors in executing infeasible robot actions, and errors in satisfying task requirements. We analyze the types of errors in each category and find several common modes of failures across LLMs. We believe this analysis will be invaluable in furthering research on LLM-guided robot program generation. Driven by our initial findings, we include a simple rejection sampling procedure that shows immediate improvements in reducing robot execution errors of LLM-generated robot programs.

In summary, this paper contributes:

1) CODEBOTLER, an open-source tool to generate robot programs from natural language using LLMs, and to enable robot-agnostic deployment of such programs;
2) ROBOEVAL, a benchmark to evaluate LLM-generated robot programs for service mobile robots;
3) a comprehensive analysis and taxonomy of failures of LLM-generated robot programs; and

This work is partially supported by the National Science Foundation (CCF-2006404 and CCF-2102291) and JP Morgan. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

[1]Department of Computer Science, University of Texas at Austin, {zichao, yash.saxena, sadanandm, joydeepb}@utexas.edu

[2]Khoury College of Computer Sciences, Northeastern University, {lucchetti.f, schlesinger.e, a.guha}@northeastern.edu

[3]Department of Computer Science, Wellesley College, af103@wellesley.edu

Copyright ©2024 IEEE

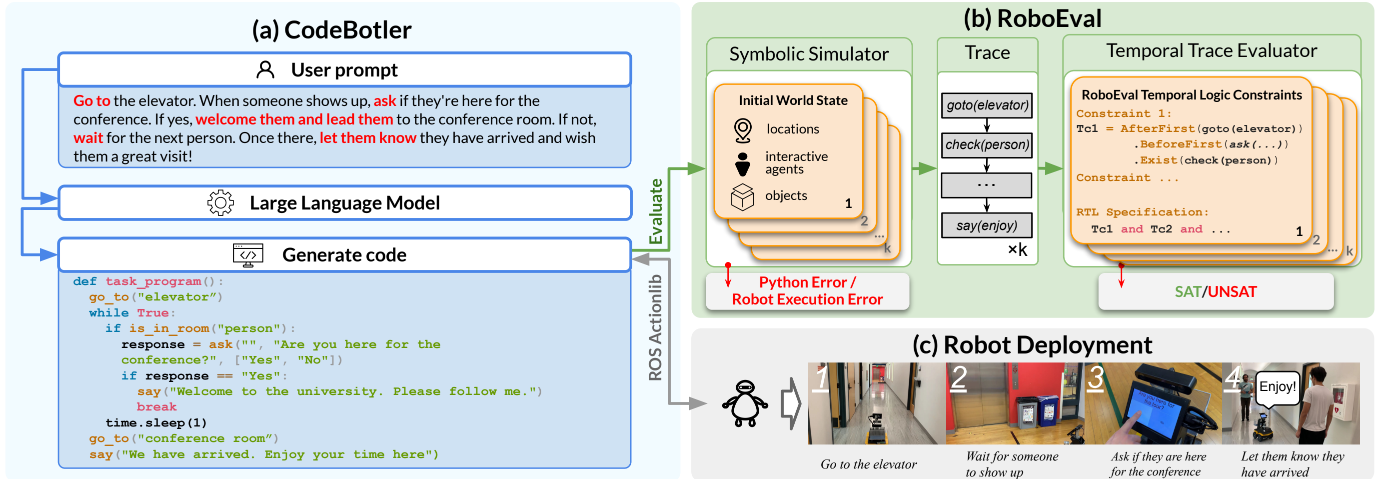

Fig. 1: The system diagram of CodeBotler and RoboEval. CodeBotler receives a user prompt and queries a large language model (LLM) to generate a robot program (**a**). It can execute the program to send instructions to a robot via ROS Actionlib (**c**). Separately, RoboEval evaluates programs generated from its benchmark tasks using a symbolic simulator and a temporal trace evaluator to determine whether each program satisfies the task constraints or not (**b**).

4) a rejection sampling mechanism to reduce robot execution failures of LLM-generated robot programs.

II. Related Work

LLMs for Robotics Applications. Using LLMs to perform robotics tasks [1]–[6], [10]–[21] has attracted a lot of attention because of their impressive out-of-box and commonsense reasoning capabilities [22]. One approach uses LLMs as task planners to break down a free-form natural language task description into multiple sub-goals. Language Models as Zero-Shot Planners [4] expresses these sub-goals in the form of natural language and builds an interpreter to convert these subgoals into robot actions. LLM+p [6] outputs these sub-goals in the form of the well-defined planning domain definition language (PDDL). Another approach leverages the code-writing capabilities of LLMs to generate programs for the robot to execute. Code-as-Policies [1] defines a robot-centric formulation of language model-generated programs (LMPs). It proposes a hierarchical method to query an LLM and generate executable Python programs that invoke parameterized robot primitives. Voxposer [13] builds on the Code-as-Policies' LMP formulation and defines a set of primitives that enables the LLMs to generate Python programs to create voxel cost maps. Then, it plans on the voxel cost maps and carries out the specified manipulation tasks. CodeBotler builds on the LMP formulation, in which we define a set of 8 robot primitives specific to service mobile robots.

Evaluating LLMs. The evaluation of LLMs is an ongoing effort due to the scope and variability of model generations [23]–[25]. Much progress has been made in evaluating code-writing LLMs [8], [9], [26], [27]. While these programming tasks assess language understanding, reasoning, algorithms, and basic mathematics, they do not address the skills of embodied agents. In the domain of embodied agents, ProgPrompt [2] creates a dataset of 70 household high-fidelity 3D simulation tasks to evaluate the code-writing capabilities of LLMs. High-fidelity 3D simulations are useful for capturing complex agent-environment dynamics but not essential for verifying program logic, and creating them can be time-consuming. To address this limitation, we create RoboEval, a lightweight benchmark that uses a *symbolic* simulator to evaluate the temporal correctness of a robot program.

III. CodeBotler: Robot-Agnostic Deployments

CodeBotler is a lightweight open-source tool that 1) defines robot-agnostic skills; 2) provides a user-friendly web interface to accept instructions and generate language model programs (LMPs) using LLMs; and 3) executes LMPs independently of robot platforms by sending commands to robots via ROS Actionlib. To illustrate the capabilities of CodeBotler, we present its design in the following subsections.

A. CodeBotler *Language Design*

CodeBotler leverages an embedded domain-specific language (eDSL) in Python to abstract 8 commonly available service robot skills. Fig. 2a shows the definition of each skill. The design of the CodeBotler eDSL encompasses skills such as: 1) `get_current_location` and `get_all_rooms` that inspect the robot's state and the world configurations, so they do not need to be hard-coded or manually specified in the prompt, as in previous work [1], [2], [13], 2) `is_in_room` that utilizes the zero-shot visual-language models (VLMs) for perceptual reasoning, 3) `ask` that provides a structured interface for human interaction through multiple-choice questions, facilitating both the LMP in processing human responses and interaction with a robot via touch-screen or audio input. 4) `pick`, `place`, and `go_to` that represent core robot manipulation and navigation abilities. These abstractions allow CodeBotler to be used for robot-agnostic deployments. An LMP can often be reused across different maps and robots through the user interface.

B. *User Interface (UI) and Robot Program Generation*

The CodeBotler UI includes a task input pane, a program preview pane, and a status monitor to track the generation of

```
# Get the current location of the robot.
def get_current_location() -> str

# Get a list of all rooms.
def get_all_rooms() -> list[str]

# Check if an object is in the current room.
def is_in_room(object : str) -> bool

# Go to a specific named location.
def go_to(location : str) -> None

# Ask a person a question, and offer a set of
specific options for the person to respond.
Returns the response selected by the person.
def ask(person : str, question : str,
        options: list[str]) -> str

# Say the message out loud.
def say(message : str) -> None

# Pick up an object if you are not already holding
one. You can only hold one object at a time.
def pick(obj: str) -> None

# Place an object down if you are holding one.
def place(obj: str) -> None
```

(a) *CodeBotler* Robot Skills

Trace Elements

e ::= goto(regex)
| say(regex)
| ask(regex[1], regex[2])
| check(regex)
| pick(regex)
| place(regex)

RTL Constraint

π ::= tc | tc and π | tc or π | not π

Trace

tr ::= $[e^1, e^2, \ldots, e^n]$	List of trace elements
| tr.BeforeFirst(e)	Return the trace before the first matching element
| tr.BeforeLast(e)	Return the trace after the first matching element
| tr.AfterFirst(e)	Return the trace before the last matching element
| tr.AfterLast(e)	Return the trace after the last matching element

Trace Constraint

tc ::= tr.Exists(e)	Returns True if the trace contains the trace element

(b) *RoboEval* Temporal Logic (RTL) Formula

Task: *Tell Alice in her office to meet me in the lobby if she would like to have lunch now.*

Linear Temporal Logic (LTL)

Atomic Proposition	y = yes n = no g = goto(office) a = ask(lunch) s = say(meet)
Operator	$\wedge$ = And | $\vee$ = Or | $\neg$ = Not **F** = Finally | **G** = Globally | **U** = Until | **N** = Next
Constraint	$\pi \vDash g \wedge \mathbf{N}\,[\,\mathbf{F}\,a \wedge (y \wedge \mathbf{N}\,\mathbf{F}\,s \vee n \wedge \mathbf{N}\,\mathbf{G}\neg s)]$

RoboEval Temporal Logic (RTL)

Trace elements	g = goto(office) a = ask(lunch) s = say(meet)

Constraints	if yes:
	$\pi \vDash tr$.AfterFirst(g).AfterFirst(a).Exists(s)
	if no:
	$\pi \vDash$ not tr.AfterFirst(g).AfterFirst(a).Exists(s) and tr.AfterFirst(g).Exists(a)

(c) LTL *vs.* RTL Example

Fig. 2: CodeBotler robot skills (**a**), RoboEval temporal logic (RTL) formula (**b**), and the LTL specifications *vs.* the RTL specifications of an example task (**c**). In section (**c**), the terms *office*, *meet*, and *lunch* are used to represent the regex patterns. The RTL specifications are simpler to express and have improved readability.

programs on the robot. When given a user task, CodeBotler combines it with a prompt prefix containing robot skills and a few example programs, and then queries an LLM for program generation. In addition, CodeBotler supports many LLM interfaces, including the OpenAI API [28], the Google PaLM API [29], and HuggingFace models (`AutoModel` and `Text-Generation-Inference`).

C. Robot Deployment With CodeBotler

CodeBotler is designed to work with the ROS system and acts as a client by utilizing the ROS Actionlib [7]. When CodeBotler executes an LMP and encounters a statement that invokes a robot skill (*e.g.,* `go_to("lobby")`), it publishes a goal (*e.g.,* `"lobby"`) to the appropriate remote topic (*e.g.,* `"/go_to_server"`) for a ROS action server on the robot to pick up. This approach makes CodeBotler independent of any specific robot platform and permits CodeBotler to operate both onboard and externally to the robot. Additionally, it provides robot deployers with the flexibility to customize the ROS action server to accommodate their specific needs for these primitives[1].

IV. The RoboEval Benchmark

RoboEval consists of a simulator, an evaluator, and a benchmark suite of tasks. Given P, the space of natural language prompts describing service mobile tasks, and Π, the set of possible LMPs, CodeBotler generates LMPs $\pi \in \Pi$ given a prompt $p \in P$. The symbolic simulator accepts a world state $w \in W$ and an LMP $\pi \in \Pi$, and produces a program trace $r \in R$. The evaluator accepts a trace and a temporal constraint $c \in C$, and returns whether the trace satisfies the constraint or not (SAT/UNSAT).

$$\text{CodeBotler} : P \rightarrow \Pi$$
$$\text{Simulator} : \Pi \times W \rightarrow R$$
$$\text{Evaluator} : R \times C \rightarrow \{\texttt{SAT}, \texttt{UNSAT}\}$$

[1]The code for our implementation of the robot skills can be found at https://github.com/ut-amrl/codebotler_amrl_impl.

The results derived from traces over multiple world states and multiple task prompts yield the success rate for an LLM on a particular task. The RoboEval benchmark thus consists of tasks $T_i (i \in [1, N])$, where each task consists of M prompts, and K world states. Each world state has a corresponding temporal check. Each RoboEval task thus consists of a tuple of prompts and multiple world-states to check against a constraint (one constraint per world state):

$$T_i = \left\langle \{p_i^j | j \in [1, M]\}, \{\langle w_i^k, c_i^k \rangle | k \in [1, K]\} \right\rangle$$

We present next 1) the RoboEval simulator, 2) the RoboEval evaluator, and 3) the tasks in the RoboEval benchmark.

A. The RoboEval Simulator

For each task T_i, the RoboEval benchmark includes multiple world states to check against. Each world state $w_i^k \in W$ consists of 1) a list of rooms in the world that `GetAllRooms()` returns, and which `GoTo()` is valid for; 2) a list of objects in the world that `IsInRoom()` returns true for; 3) a list of objects that can be manipulated using `Pick()` and `Place()`; and 4) a list of responsive humans, their locations, and regular expressions that define their responses to `Ask()`. Thus, a single LMP may produce very different traces when simulated with different initial world states. The simulator consists of a Python interpreter and a symbolic simulation of each robot skill, and the result of running an LMP π is recorded as a trace r as a sequence of robot skills that were executed, along with the parameters (*e.g.,* the location parameter of a `GoTo` call). All Python errors or robot execution errors are logged during simulation.

B. The RoboEval Evaluator

Given a trace r_i^k produced by simulating an LMP π with an initial world state w_i^k, the RoboEval evaluator checks whether r_i^k satisfies the temporal check c_i^k that defines correct execution of the task for that world state. c_i^k may consist

<table>
<tr><th colspan="8">RoboEval Benchmark Tasks</th></tr>
<tr><td>ElevatorTour</td><td>FindBackpack</td><td>GetDrink</td><td>GrilledCheese</td><td>HalloweenList</td><td>Halloween Shopping</td><td>LunchBreak</td><td>LunchTime</td></tr>
<tr><td>CountSavory</td><td>MailDelivery</td><td>MovieMessenger</td><td>SayGoodDay</td><td>SetTemperature</td><td>StaplerDelivery</td><td>StaplerSupply</td><td>WeatherPoll</td></tr>
<tr><th colspan="8">Task details</th></tr>
<tr><td>Prompt</td><td colspan="7">Paraphrase 1: Go to every office, and if there is someone there, ask them whether they'd like a cupcake, ham sandwich, donut, or beef jerky. Come back and tell me how many people chose a savory option.
Paraphrase 2: Visit all offices. If anyone is present; ask them to choose from the options of cupcake, ham sandwich, donut, or beef jerky. Let me know how many people selected a savory option when you return.
Paraphrase 3: … Paraphrase 4: … Paraphrase 5: …</td></tr>
<tr><td>Attributes</td><td colspan="4">Navigation, Perception, Commonsense Reasoning, Arithmetic, Conditional Statements</td><td colspan="2">Number of Initial World States</td><td>4</td></tr>
</table>

Fig. 3: The RoboEval benchmark includes 16 tasks, each with 5 prompt paraphrases. The figure displays these tasks' names and a detailed example of the task `CountSavory`.

of multiple conditions, expressed in conjunctive normal form over multiple temporal constraints. We review Linear Temporal Logic, which is well-suited to codifying such constraints in order to check for correctness.

Linear Temporal Logic. An LTL formula follows the grammar shown in Fig. 2c — it composes atomic propositions $\pi \in \Pi$ with logical operators $\neg, \wedge, \vee$ and temporal operators $\mathcal{F}, \mathcal{G}, \mathcal{U}, \mathcal{N}$. Given LTL formulas ϕ_1, ϕ_2 defined over a temporal sequence, $\mathcal{F}\phi_1$ is true iff ϕ_1 is true eventually at some point along the sequence, $\mathcal{G}\phi_1$ is true iff ϕ_1 is true over the entire sequence, and $\phi_1 \mathcal{U} \phi_2$ is true iff ϕ_1 for a sub-sequence and ϕ_2 is true for the remainder of the sequence after that. $\mathcal{N}\phi$ is true for a sequence iff the next element in the sequence satisfies ϕ.

RoboEval ***Temporal Logic*** While LTL suffices for writing robot task specifications, these LTL formulas can become complex as task complexity increases. For example, consider an example task $\mathcal{T}_1$ where a user asks the robot, *"tell Alice in her office to meet me in the lobby if she agrees to lunch"*. To complete this task, the robot 1) *first* needs to go to Alice's office; 2) *then* ask Alice whether she would like to have lunch; and 3) *finally* if she agrees, tell her to meet in the lobby. Fig. 2c shows the complete LTL specification for this task. Declaring such specifications is quite tedious and error-prone. To address this challenge, we observe that 1) specifying temporal logic is easier and less error-prone for specific scenarios (*e.g.,* one scenario for if Alice says yes, and a different scenario for no), and 2) the temporal formulas for robot tasks necessarily depend on the robot skills. We thus introduce the RoboEval *Temporal Language* (RTL), a language derived from LTL that is particularly well-suited to specifying temporal logic formulas for robot tasks. Fig. 2b shows the grammar of RTL, and Fig. 2c shows the corresponding RTL formula for task $\mathcal{T}_1$. An additional advantage of the condition expressed in RTL *vs.* LTL is improved readability.

C. The RoboEval Benchmark Tasks

The RoboEval benchmark contains a suite of 16 tasks. Fig. 3 shows the names of the these tasks, along with a detailed example of the task `CountSavory` [2]. These tasks are designed to check whether an LMP can 1) ground language instructions to correct function calls to robot primitives; 2) perform accurate sequencing of robot actions; 3) handle complex control flows based on different world configurations; 4) solve arithmetic problems; 5) comprehend open-world knowledge. In addition, research has shown [30] that LLMs may not be as robust as previously thought, and trivial prompt variations could cause significant performance variations for LLMs [9], [31]. For this reason, we provide 5 different paraphrases of the task prompt to evaluate the robustness of an LLM in dealing with slight prompt variations.

V. Benchmark Results And Analysis

To gain insights into the capabilities and limitations of different state-of-the-art LLMs for generating service mobile robot LMPs, we use the RoboEval benchmark to empirically answer the following questions:

1) First, how do different LLMs perform in generating programs for tasks in the RoboEval benchmark?
2) Second, when a generated service robot LMP fails, what are the causes?

To investigate these two questions, we evaluate five LLMs: 1) GPT-4 [32], 2) GPT-3.5 [22] (`text-davinci-003`), and 3) PaLM2 [33] (`text-bison-001`) as state-of-the-art API-only proprietary models; and 4) CodeLlama [34] (`Python-34b-hf`) and 5) StarCoder [35] as open-access models. When evaluating code generation models, we use standard values of $T = 0.2$ for temperature and $p = 0.95$ for nucleus sampling [8]. In the following subsections, we discuss our analysis of each question in detail.

A. Performance Of LLMs On The RoboEval Benchmark

The RoboEval benchmark consists of 16 tasks, each with 5 prompt paraphrases, totaling 80 different prompts. For each prompt, we generate 50 program completions and calculate the pass@1 score [8], a common metric for LMP evaluation. This score indicates the probability of an LMP being correct if an LLM generates only one LMP for a given prompt.

Overall Performance Analysis. We first investigate the overall performance of each LLM in generating LMPs. Since each LLM gets a pass@1 score for every prompt, we compute the percentage of prompts that have a pass@1 score greater than or equal to a threshold value, which ranges from 1 to 0. We present this information in Fig. 5 as a Cumulative Distribution Function (CDF). Although relaxing the pass@1 score threshold for each LLM increases prompt coverage, there are still certain prompts (ranging from 48.75% for StarCoder

[2]A comprehensive list of the task descriptions can be found at https://amrl.cs.utexas.edu/codebotler/

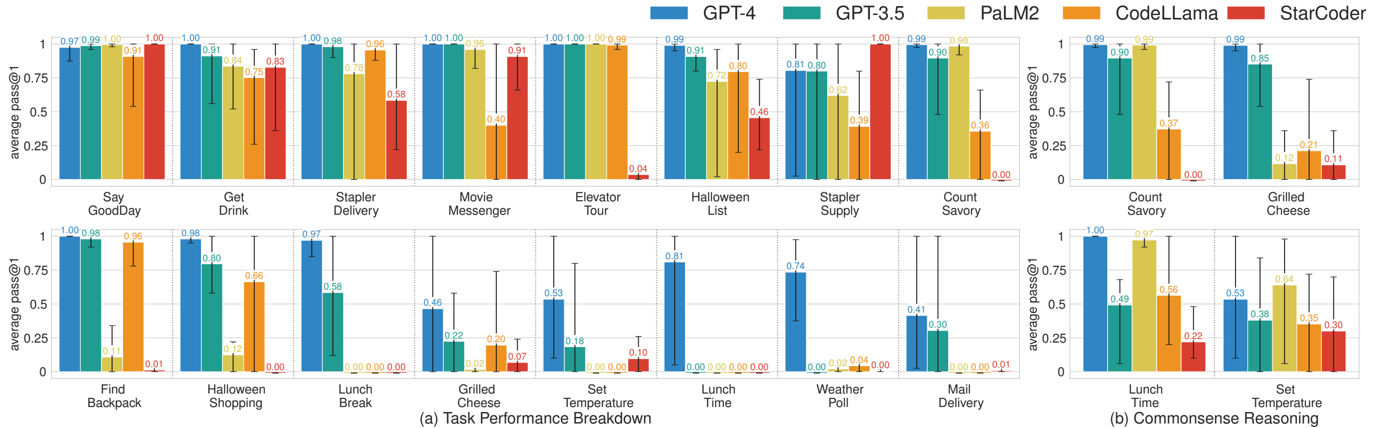

Fig. 4: 5 LLMs are evaluated on the ROBOEVAL benchmark. Each benchmark task contains 5 different prompt paraphrases and each bar represents the average pass@1 score of an LLM for generating responses across all 5 prompts within a given ROBOEVAL benchmark task. Each error bar indicates the range from the highest to the lowest pass@1 score across all prompts. On the left side (a), the performance of LLMs on each task of the ROBOEVAL benchmark is displayed. On the right side (b), the chart shows the performance of LLMs on tasks that have been adapted to exclusively evaluate the models' proficiency in performing commonsense reasoning.

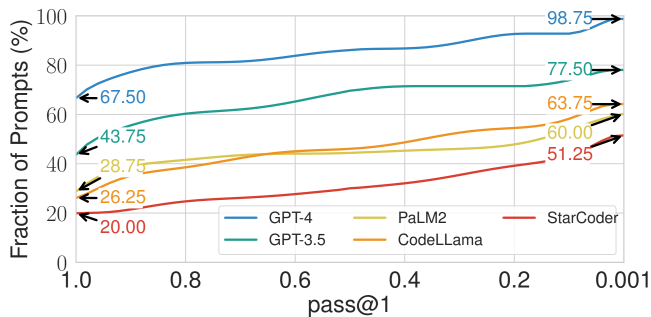

Fig. 5: Cumulative Distribution Function (CDF) curves depict the percentage of prompts for which each LLM can generate correct LMPs at various pass@1 score thresholds. A perfect LLM would show a horizontal line at 100%, indicating it can generate correct LMPs for all prompts with a pass@1 score of 1. To maintain visual clarity, we limit the x-axis to 10^{-3} since all CDF plots eventually reach 100%.

to 1.25% for GPT-4) where LLMs consistently fail to generate correct LMPs.

Performance of LLMs on Individual Tasks. We then look into the performance of each LLM for specific tasks. We present this information in Fig. 4a. Since each task has 5 different prompt paraphrases and a pass@1 score is calculated for each prompt, we average the pass@1 score across 5 prompts. Additionally, we plot an error bar to visualize the variation between the highest to the lowest pass@1 score over all prompts for each task.

From the error bars, we note a considerable disparity between a model's best and worst pass@1 scores for a given task. This suggests that models are not robust to changes in the phrasing of the prompt. Single-word changes can result in substantial performance variations. A common example is changing the verb "ask" to "inquire"; "Ask him about his available ingredients" in the `GrilledCheese` task thus becomes "inquire about his available ingredients". This seems to affect some code model's ability to call the robot `ask` function.

From this chart, we further note a high variation in performance across tasks. The top row contains the high-performing tasks, where four or more models score over 0.7 on pass@1, while the bottom row contains the low-performing tasks where three or more models score below 0.2 pass@1. To identify why such a high variation exists, we run an ablation experiment. We notice that tasks involving commonsense reasoning (`CountSavory`, `GrilledCheese`, `LunchTime`, `SetTemperature`) tend to underperform. Hence, we ablate all but commonsense RTL checks on these tasks. For example, the full `CountSavory` checks require that the robot navigates to every office as well as understanding that *beef jerky* and *ham sandwich* are savory options; for the ablation, we remove these navigation checks. We plot the resulting average pass@1 scores of this experiment on Fig. 4b. If models are failing commonsense reasoning, we expect the performance to be unchanged after ablation. However, we notice that models are largely improving in performance. This suggests that the problem lies elsewhere, which we will analyze next.

B. *Causes of Failures of LMPs*

Given that LLMs still have room for improvement on the ROBOEVAL benchmark, we want to understand the causes of failures for LLMs to generate robot programs. We classify these failures into three categories: 1) Python Errors, including syntax, runtime, and timeout errors; 2) Robot Execution Errors, that occurs when a program attempts to execute an infeasible action, such as navigating to a non-existent (hallucinated) location; and 3) Task Completion Errors, where the program runs correctly in the simulator but fails RTL checks for task completion. We use ROBOEVAL's symbolic simulator to detect and classify Python Errors and Robot Execution Errors, and we use ROBOEVAL's evaluator to capture the Task Completion Errors. Fig. 6a shows the breakdown of these

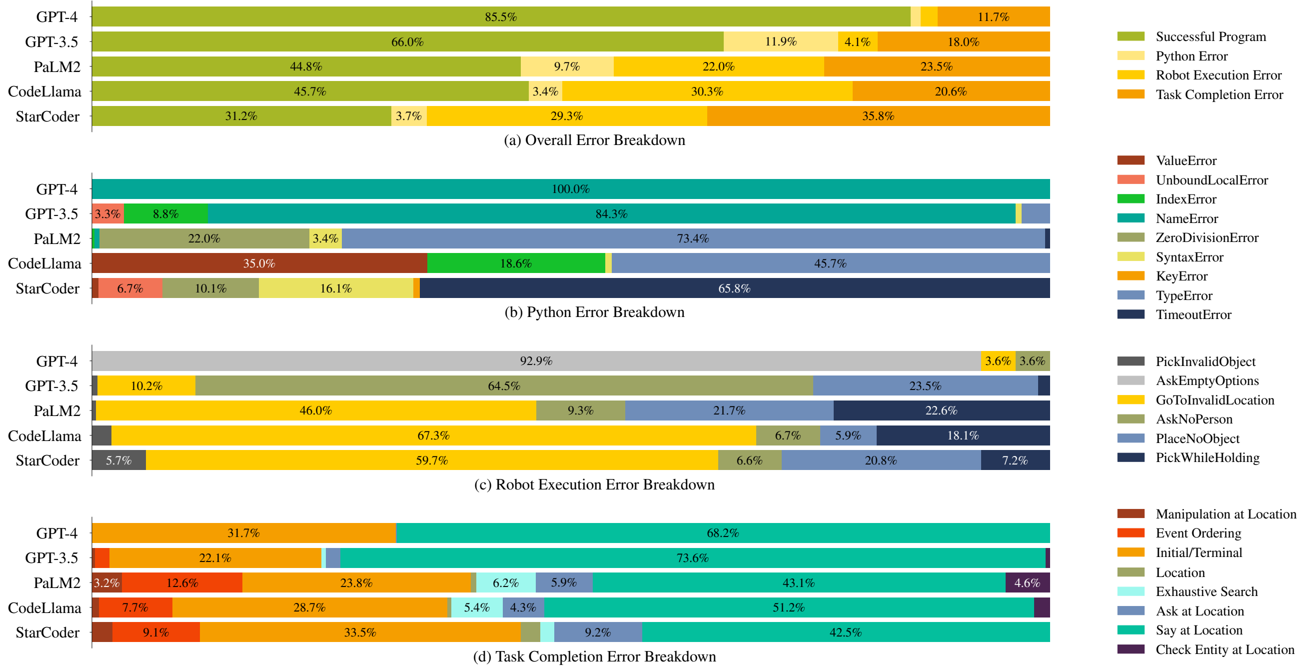

Fig. 6: Causes of failures for LMPs on the RoboEval benchmark.

failure categories for each LLM. We observe that despite having fewer parameters, the CodeLLMs (CodeLlama and StarCoder) generally make fewer Python errors. This suggests that LLMs trained on a larger proportion of code may be more adept at generating successful completions in the DSL defined in the prompt.

In the following subsections, we will analyze each error in detail.

Python Error Analysis. Fig. 6b shows the specific error breakdown of the Python Error. The Python Error encompasses a multitude of errors, but their distribution exhibits a long-tailed pattern. In GPT-3.5 and GPT-4, *NameError* is predominant because of undefined variables. PaLM2 and CodeLLama, on the other hand, often generate *TypeError* due to the misuse of data types, while Starcoder commonly encounters *TimeoutError* when the LMPs get stuck in loops.

Robot Execution Error. Fig. 6c shows the error breakdown of the Robot Execution Error. There are 6 root causes of robot execution errors:

1) GoToInvalidLocation/PickInvalidObject: the program calls `go_to` or `pick` with a hallucinated argument;
2) PlaceNoObject/PickWhileHolding: the program tries to pick/place an object when it is not/already holding one;
3) AskNoPerson: the program calls `ask` at a location with no person nearby; and
4) AskEmptyOptions: the program calls `ask` with an empty list of options for the person to choose from.

We first observe that hallucination plays a substantial role in causing errors, Specifically, the GoToInvalidLocation and PickInvalidObject errors contribute to 33.7% of the total robot execution errors in GPT-3.5, 67.7% in PaLM2, 73.2% in CodeLlama, and 80.5% in StarCoder. We also notice that another important source of errors arises from the PlaceNoObject and PickWhileHolding errors, as well as AskNoPerson errors for GPT-3.5. These errors require the program to be aware of the internal state of the robot or keep track of the external world state. The prevalence of these errors suggests a gap in LLM's ability to keep track of changing states by binding agents to current states. Some of these errors also result from a failure to respect the behavior of our robot functions. For example, our `pick` function only allows the robot to pick an object if it is not already holding one. However, the model may not include this information in its completion and attempt to pick up multiple objects.

Task Completion Error. Fig. 6d shows the breakdown of the Task Completion Errors. We classify every temporal check in the RoboEval benchmark into one of the following categories:

1) {Say/Ask/Manipulation/CheckEntity} AtLocation: The task requires executing a specific action (`say`, `ask`, `pick`, `place` or `check`) at a specific location, but the program fails to do so;
2) Initial/Terminal: The program does not accurately perform an initial or final action;
3) EventOrdering: The program does not carry out actions in the prescribed sequence or has redundant navigation;
4) Location: The program commands the robot to visit a location irrelevant to the task; and,
5) ExhaustiveSearch: The program does not visit all locations required to complete the task.

We notice that the most common task completion errors in each LLM are the SayAtLocation error and the Initial/Terminal error. These two types of errors constitute the majority of task completion errors, making up 100% of the errors in GPT-4, 95.7% in GPT-3.5, 66.9% in PaLM2, 79.9% in CodeLlama, and 76.0% in StarCoder.

After analyzing the programs that produce these errors, we find that most are related to a recurring statement in many

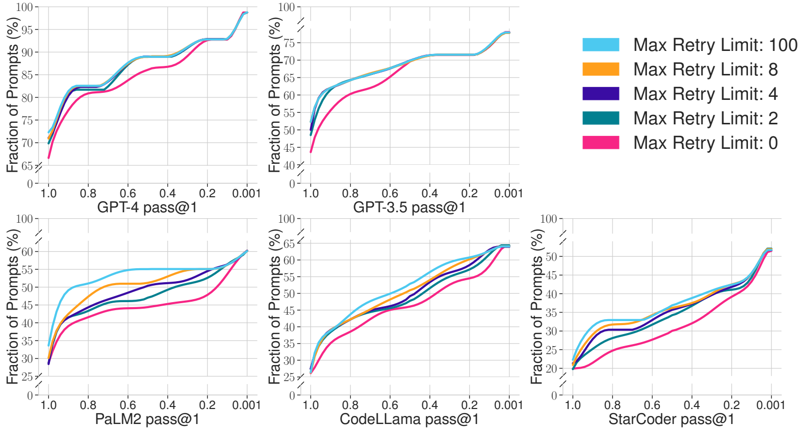

Fig. 7: Cumulative Distribution Function (CDF) of the LLMs' performance across different max retry limits. As the max retry limit increases, all five LLMs improve in performance.

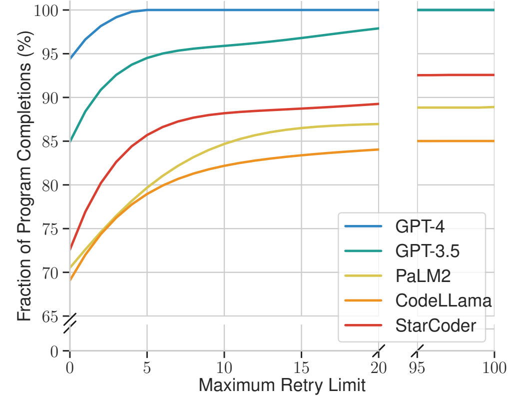

Fig. 8: Cumulative Distribution Function (CDF) of the fraction of program completions that can be executed over different max retry limits.

```
Task: Compile a list of ingredients for grilled cheese. Go to Zarko's office, and
ask him which of them he has. Come back and tell me what he does not have.
Successful Completion:
def task_program():
    missing_ingredients = []
    start_loc = get_current_location() # <---- record the starting location
    go_to("Zarko's office")
    for ingredient in ["bread", "butter", "cheese"]:
        response = ask("Zarko", "Do you have " + ingredient + "?", ["Yes", "No"])
        if response == "No":
            missing_ingredients.append(ingredient)
    go_to(start_loc) # <---- return to the starting location
    say("Zarko does not have " + ",".join(missing_ingredients))
-------------------------------------------------------------------------------------
Failed Completion:
def task_program():
    missing_ingredients = []
    go_to("Zarko's office")
    for ingredient in ["bread", "butter", "cheese"]:
        response = ask("Zarko", "Do you have " + ingredient + "?", ["Yes", "No"])
        if response == "No":
            missing_ingredients.append(ingredient)
    go_to(get_current_location()) # <---- does not return to the starting location
    say("Zarko does not have " + ",".join(missing_ingredients))
```

Fig. 9: Examples of successful (top) and failed (bottom) program completions related to the *"come back"* instruction.

of our tasks: *"Come back and tell me..."* (Fig. 9 shows an example). This statement is often used in tasks in which we want a service robot to go to a different location, complete a task, and then return to us with an update on its progress. This is the most common instruction that fails across models, and also the error that is directly responsible for many of our low-performing tasks. Programs that fail the *"come back"* instruction often do not use the `get_current_location` primitive to record the robot's starting location, and as a result, cannot refer to the starting location at the end of the program. This also causes LLMs to hallucinate locations and produce the GoToInvalidLocation errors — rather than referencing a non-existent `start_loc` variable, some LLMs generate a location based on context clues and send the robot to that location (for example, returning to the kitchen in `GrilledCheese`, although no kitchen is mentioned in the prompt). We observe that in tasks without a *"come back"* instruction, but an explicit return statement like *"return to the mail room"* do not suffer from the same error.

VI. Improving Robot Program Generations

Based on the analysis of the failures of LMPs in RoboEval, we are interested in understanding how to improve service robot program generation using LLMs. Recognizing the breadth of potential improvements, this study focuses on an initial step: we propose a *rejection sampling strategy* to identify and reduce LMP failures (Python Errors and Robot Execution Errors) before deploying the LMP on the robot.

To detect errors in an LMP, the RoboEval symbolic simulator takes in the current world state and executes the LMP. If an error is identified (Python Errors and Robot Execution Errors), CodeBotler will prompt an LLM for a new program and submit the program to the symbolic simulator to execute again. This cycle repeats until an LMP successfully passes in the symbolic simulator for deployment on the robot or until a maximum retry limit is reached.

This proposed strategy has one limitation: the symbolic simulator may not know a-priori the true state of the world, including the current locations of humans and movable objects, or how humans might respond to the robot's questions. We address this limitation by proposing a task-agnostic world state. This world state contains the permanent entities (*e.g.,* known rooms) and employs *state sampling* to simulate random potential world states for non-static entities, such as possible human locations, movable objects, and human responses. Subsequently, each LMP undergoes multiple simulation runs (we chose 5 in our experiments) in the symbolic simulator to ensure statistical reliability when identifying LMP failures.

We evaluate this strategy on all five LLMs with four different maximum retry limits $(2, 4, 8, 100)$ and compare them with the baseline (without rejection sampling). Fig. 7 shows the CDF curves of the percentage of prompts that can be successfully generated given a threshold of the pass@1 score for each LLM with respect to different maximum retry limits. We observe an improvement in performance across all LLMs as the maximum retry limit is increased.

We then investigate how effective the rejection sampling strategy is in eliminating the program execution errors. We compute the percentage of total program completions that can be eventually executed over different maximum retry limits and plot it as a CDF in Fig. 8. Interestingly, we observe that it is possible for some LLMs (PaLM2, CodeLlama, and Starcoder) to never generate successful completions for certain tasks. As a result, while the rejection sampling strategy can improve the performances of LLMs, it is not enough to resolve

all program execution errors.

This observation, coupled with prior findings of LLMs' consistent failures in generating correct LMPs as detailed in Section V-A, points to a systemic challenge in LMP generation. It highlights the issue of abstraction matching [36], which entails aligning ambiguous natural language expressions of user intent with their precise, unambiguous code representations. In future work, we would like to explore more sophisticated strategies, such as utilizing grounded abstraction matching, to solve this problem.

VII. Conclusion, Limitations, & Future Works

In this work, we present CodeBotler and RoboEval. CodeBotler is an open-source robot-agnostic tool to generate general-purpose service robot programs from natural language, and RoboEval is a benchmark for evaluating LLMs' capabilities of generating programs to complete service robot tasks. We evaluate the performance of five LLMs in generating robot programs and perform an analysis of the causes of failures. Our analysis reveals that the errors exhibit a long-tail distribution, with LLMs predominantly struggling with hallucination issues and grounding the phrase *"come back"*. Finally, we propose a rejection sampling strategy to handle program failures. This method has led to improved performance for all five LLMs.

This work has several limitations that could be addressed in future research. Firstly, CodeBotler currently does not support low-level behaviors, such as *"follow Alice to her office"*. Secondly, CodeBotler generates LMPs in an open-loop fashion, rendering it incapable of reacting to unexpected changes in the environment. Thirdly, this study does not consider the strategies for crafting prompts that could improve the performance of LLMs in generating service robot programs. Finally, although the RTL constraints are designed to reduce the workloads of writing specifications compared to LTL constraints, the users still need to manually specify constraints for each task. Therefore, investigating the Tree-of-Thoughts concept [37] to dynamically generate RTL checks with LLMs might be valuable.

References

[1] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng, "Code as Policies: Language Model Programs for Embodied Control," in *arXiv:2209.07753*, 2022.

[2] I. Singh, V. Blukis, *et al.*, "ProgPrompt: Generating Situated Robot Task Plans using Large Language Models," in *ICRA 2023*, 2023.

[3] M. Ahn, A. Brohan, *et al.*, "Do As I Can, Not As I Say: Grounding Language in Robotic Affordances," 2022.

[4] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch, "Language Models as Zero-Shot Planners: Extracting Actionable Knowledge for Embodied Agents," *International Conference on Learning Representations*, 2022.

[5] D. Driess, F. Xia, *et al.*, "PaLM-E: An Embodied Multimodal Language Model," *Proceedings of the 40th International Conference on Machine Learning (ICML), no. 340, pp. 8469–8488*, Jul. 2023.

[6] B. Liu, Y. Jiang, *et al.*, "LLM+P: Empowering Large Language Models with Optimal Planning Proficiency," *arXiv:2304.11477*, 2023.

[7] "ROS Actionlib," accessed 2023-10-19. [Online]. Available: http://wiki.ros.org/actionlib

[8] M. Chen, J. Tworek, H. Jun, Q. Yuan, *et al.*, "Evaluating large language models trained on code," *arXiv:2107.03374*, 2021.

[9] H. M. Babe, S. Nguyen, Y. Zi, A. Guha, M. Q. Feldman, and C. J. Anderson, "StudentEval: A benchmark of student-written prompts for large language models of code," *arXiv:2306.04556*, 2023.

[10] C. Huang, O. Mees, A. Zeng, and W. Burgard, "Visual Language Maps for Robot Navigation," *IEEE International Conference on Robotics and Automation (ICRA), pp. 10608-10615*, 2022.

[11] Y. Ding, X. Zhang, C. Paxton, and S. Zhang, "Task and Motion Planning with Large Language Models for Object Rearrangement," *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 2086-2092*, 2023.

[12] H. Wang, G. Gonzalez-Pumariega, Y. Sharma, and S. Choudhury, "Demo2Code: From Summarizing Demonstrations to Synthesizing Code via Extended Chain-of-Thought," *37th Conference on Neural Information Processing Systems*, 2023.

[13] W. Huang, C. Wang, *et al.*, "VoxPoser: Composable 3D Value Maps for Robotic Manipulation with Language Models," *Proceedings of The 7th Conference on Robot Learning, PMLR vol. 229, pp. 540-562*, 2023.

[14] A. Brohan, N. Brown, *et al.*, "RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control," *Proceedings of The 7th Conference on Robot Learning, PMLR vol. 229, pp. 2165–2183*, 2023.

[15] Y. Tang, W. Yu, *et al.*, "SayTap: Language to Quadrupedal Locomotion," *Proceedings of The 7th Conference on Robot Learning, PMLR vol. 229, pp. 3556-3570*, 2023.

[16] W. Yu, N. Gileadi, C. Fu, *et al.*, "Language to Rewards for Robotic Skill Synthesis," *Proceedings of The 7th Conference on Robot Learning, PMLR vol. 229, pp. 374-404*, 2023.

[17] D. Shah *et al.*, "LM-Nav: Robotic Navigation with Large Pre-Trained Models of Language, Vision, and Action," *Proceedings of The 6th Conference on Robot Learning, PMLR vol. 205, pp. 492-504*, 2022.

[18] W. Huang, F. Xia, *et al.*, "Inner Monologue: Embodied Reasoning through Planning with Language Models," *Proceedings of The 6th Conference on Robot Learning, PMLR vol. 205, pp. 1769-1782*, 2022.

[19] J. Wu, R. Antonova, *et al.*, "TidyBot: Personalized Robot Assistance with Large Language Models," *Autonomous Robots, vol. 47, no. 8, pp. 1087–1102*, 2023.

[20] B. Chen, F. Xia, *et al.*, "Open-vocabulary Queryable Scene Representations for Real World Planning," *IEEE International Conference on Robotics and Automation (ICRA), pp. 11509-11522*, 2022.

[21] C. H. Song, J. Wu, *et al.*, "LLM-Planner: Few-Shot Grounded Planning for Embodied Agents with Large Language Models," *Proceedings of IEEE/CVF International Conference on Computer Vision (ICCV)*, 2023.

[22] T. Brown, B. Mann, N. Ryder, *et al.*, "Language Models are Few-Shot Learners," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 1877–1901.

[23] Y. Chang *et al.*, "A survey on evaluation of large language models," *ACM Transactions on Intelligent Systems and Technology*, 2024.

[24] P. Liang, R. Bommasani, T. Lee, *et al.*, "Holistic Evaluation of Language Models," *Transactions on Machine Learning Research*, 2023.

[25] X. Liu, H. Yu, *et al.*, "AgentBench: Evaluating LLMs as Agents," *The 12th International Conference on Learning Representations*, 2023.

[26] Y. Lai *et al.*, "DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation," *Proceedings of the 40th International Conference on Machine Learning, PMLR vol. 202, pp.18319-18345*, 2023.

[27] J. Austin, A. Odena, *et al.*, "Program Synthesis with Large Language Models," *in arXiv:2108.07732*, 2021.

[28] "OpenAI Platform," accessed: 2023-9-10. [Online]. Available: https://platform.openai.com/docs/guides/gpt

[29] "Generative AI for Developers," accessed: 2023-9-10. [Online]. Available: https://developers.generativeai.google/

[30] A. V. Miceli-Barone, F. Barez, I. Konstas, and S. B. Cohen, "The Larger They Are, the Harder They Fail: Language Models do not Recognize Identifier Swaps in Python," *61st Annual Meeting of the Association for Computational Linguistics*, 2023.

[31] K. D. Dhole, V. Gangal, S. Gehrmann, *et al.*, "NL-Augmenter: A Framework for Task-Sensitive Natural Language Augmentation," *Northern European Journal of Language Technology*, 2021.

[32] OpenAI, "GPT-4 Technical Report," *arXiv:2303.08774*, 2023.

[33] R. Anil, A. M. Dai, O. Firat, *et al.*, "PaLM 2 Technical Report," *arXiv:2305.10403*, 2023.

[34] B. Rozière, J. Gehring, F. Gloeckle, *et al.*, "Code Llama: Open Foundation Models for Code," *arXiv:2308.12950*, 2023.

[35] R. Li, L. B. Allal, Y. Zi, *et al.*, "StarCoder: may the source be with you!" *Transactions on Machine Learning Research*, 2023.

[36] M. X. Liu, A. Sarkar, *et al.*, ""what it wants me to say": Bridging the abstraction gap between end-user programmers and code-generating large language models," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023.

[37] S. Yao, D. Yu, *et al.*, "Tree of Thoughts: Deliberate Problem Solving with Large Language Models," *Proceedings of the 37th International Conference on Neural Information Processing Systems*, 2023.