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/.**

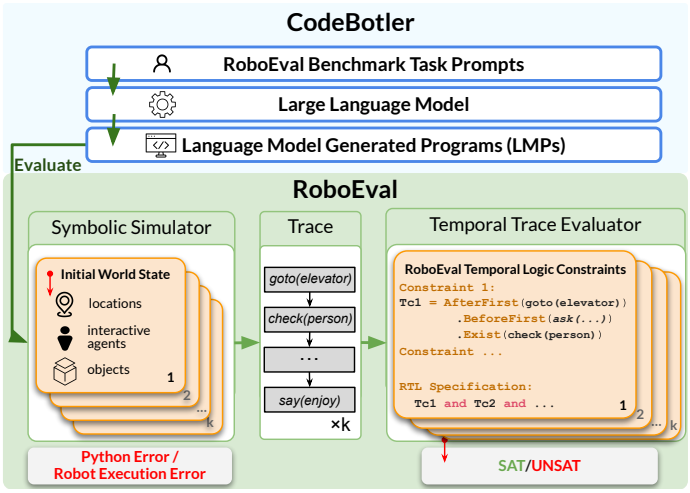

Fig. 1: The system diagram of CODEBOTLER and ROBOEVAL. CODEBOTLER receives a task prompt and queries a large language model (LLM) to generate a robot program. Then ROBOEVAL evaluates the generated programs using a symbolic simulator and a temporal trace evaluator to determine whether each program satisfies the task constraints or not.

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]–[3], planning [4]–[7], and in this work, 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 present CODEBOTLER and ROBOEVAL [8]: 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.

While the capabilities of LLMs at producing robot programs are impressive, they are still susceptible to a variety of failures. To understand the nature of the failures, we need an effective method to evaluate these programs. Existing benchmarks typically rely on either simple input-output unit test functions [1], [9], or they utilize complex high-fidelity 3D simulations [2]. However, checking for input-output pairs is insufficient when it comes to evaluating service robot programs. Consider the task *"Check how many conference rooms have no markers"*. It is insufficient to just check for the number of conference rooms stated by the LLM-generated programs. Rather, the correctness of the program depends on *the sequence of robot actions taken*. In this example, the robot must visit each conference room and check for markers, before arriving at the final answer. Furthermore, the correct sequence of actions may depend on the specific *world state*.

We thus introduce the ROBOEVAL benchmark to address these challenges in evaluating LLM-generated programs for service mobile robots. This benchmark integrates three key components: a symbolic simulator, a trace evaluator, and a comprehensive suite of 16 tasks. Fig. 1 shows the system diagram of CODEBOTLER and ROBOEVAL. When a program created by CODEBOTLER is passed into ROBOEVAL, it undergoes a two-step evaluation process. First, the program is executed within a symbolic simulator, which produces multiple program traces corresponding to different initial world states. Next, these traces are evaluated against a set of temporal specifications. These specifications are designed to define the correct behavior for the given task, tailored to each specific initial world state.

Finally, We use the ROBOEVAL benchmark to evaluate and analyze the performance of five state-of-the-art Large Language Models (LLMs) and propose a rejection sampling strategy based on the ROBOEVAL symbolic simulator to improve the code generation performance of these LLMs.

[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

```python
# 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,…,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)	**RoboEval Temporal Logic (RTL)**
Atomic Proposition	y = yes n = no g = goto(office) a = ask(lunch) s = say(meet)	Trace elements g = goto(office) a = ask(lunch) s = say(meet)
Operator	$\wedge$ = And | $\vee$ = Or | $\neg$ = Not **F** = Finally | **G** = Globally | **U** = Until | **N** = Next	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)
Constraint	$\pi \vDash g \wedge$ **N** [**F** $a \wedge (y \wedge$ **N F** $s \vee n \wedge$ **N G** $\neg s$)]	

(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.

II. 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`).

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

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 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

RoboEval Benchmark Tasks							
ElevatorTour	FindBackpack	GetDrink	GrilledCheese	HalloweenList	Halloween Shopping	LunchBreak	LunchTime
CountSavory	MailDelivery	MovieMessenger	SayGoodDay	SetTemperature	StaplerDelivery	StaplerSupply	WeatherPoll
Task details							
Prompt	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: ...						
Attributes	Navigation, Perception, Commonsense Reasoning, Arithmetic, Conditional Statements				**Number of Initial World States**		4

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

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` [1]. 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 [10] that LLMs may not be as robust as previously thought, and trivial prompt variations could cause significant performance variations for LLMs [11], [12]. 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.

III. RoboEval Results

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 evaluate five state-of-the-art models: 1) GPT-4 [13], 2) GPT-3.5 [14] (`text-davinci-003`), and 3) PaLM2 [15] (`text-bison-001`) as state-of-the-art API-only proprietary models; and 4) CodeLlama [16] (`Python-34b-hf`) and 5) StarCoder [17] as open-access models.. From the experiments, we hope 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?

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

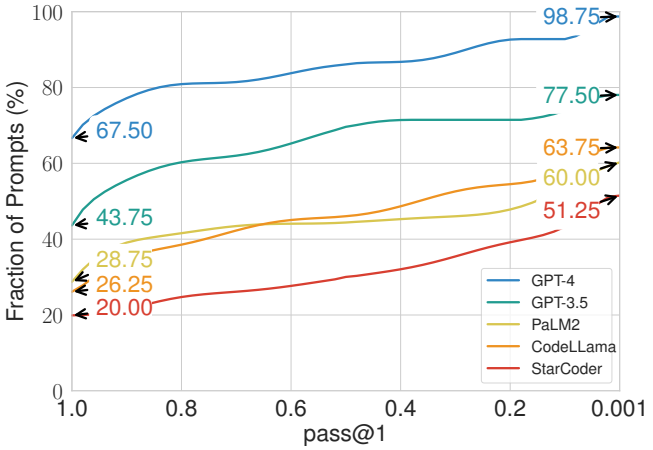

Fig. 4: Cumulative Distribution Function (CDF) curves depict the percentage of prompts for which each LLM can generate correct LMPs at various pass@1 score thresholds. A perfect LLM would show a horizontal line at 100%, indicating it can generate correct LMPs for all prompts with a pass@1 score of 1. To maintain visual clarity, we limit the x-axis to 10^{-3} since all CDF plots eventually reach 100%.

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 [9], 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.

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

B. Causes of Failures of LMPs

We evaluate the failure modes of the LMPs and 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)

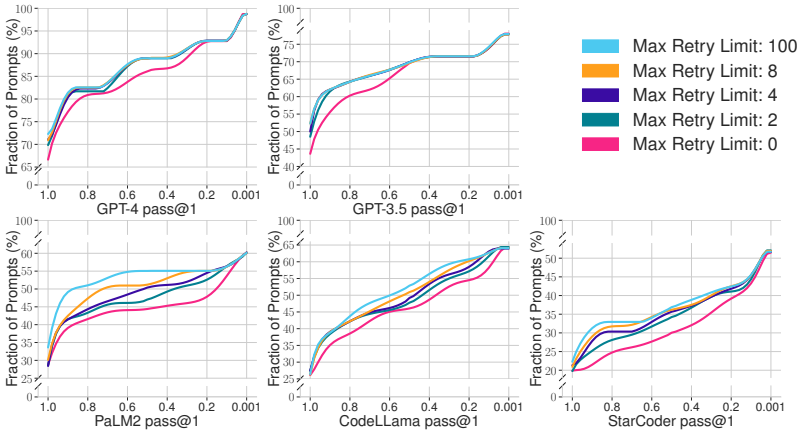

Fig. 5: Cumulative Distribution Function (CDF) of the LLMs' performance across different max retry limits. As the max retry limit increases, all five LLMs improve in performance.

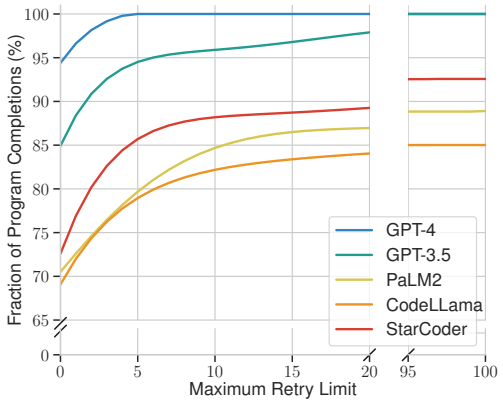

Fig. 6: Cumulative Distribution Function (CDF) of the fraction of program completions that can be executed over different max retry limits.

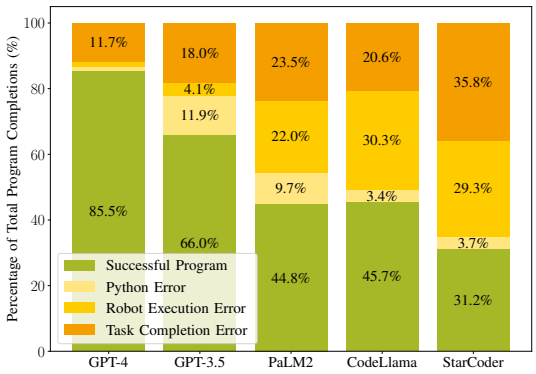

Fig. 7: Causes of failures for LMPs on the RoboEval benchmark.

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. 7shows the breakdown of these failure categories for each LLM.

We observe that despite having fewer parameters, the CodeLLMs (CodeLlama and StarCoder) generally make fewer Python errors, which 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.[2]

IV. Improving Robot Program Generations

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

[2]For a detailed analysis of our study, please refer to our paper available on arXiv at https://arxiv.org/pdf/2311.11183.pdf.

To detect errors in an LMP, the RoboEval symbolic simulator uses the current world state and executes the LMP. If the execution fails, CodeBotler will prompt an LLM for a new program and then execute it in the symbolic simulator again. This process repeats until an LMP can successfully pass in the symbolic simulator for deployment on the robot or until a predefined 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. 5 shows the CDF curves with respect to different maximum retry limits for each LLM. Each curve indicates the fractions of prompts, which an LLM can use to generate successful LMPs, given a threshold of the pass@1 score. We observe that all five LLMs can improve their code generation performance using this strategy. In addition, as the maximum retry limit increases, this improvement becomes more substantial.

We then investigate how effective the rejection sampling strategy is in eliminating the program execution errors. We execute every LMP in the symbolic simulator and compute the percentages of successful executions. Then we plot them over different maximum retry limits as a CDF in Fig. 6 for each LLM. Interestingly, we observe that some LLMs (PaLM2, CodeLlama, and Starcoder) constantly fail to generate successful programs for certain tasks. Thus, while the rejection sampling strategy can improve LLM performance, it does not eliminate all execution errors. In future work, we would like to explore more sophisticated strategies to mitigate execution errors before deploying the LMP on a robot.

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] 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.

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

[5] 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.

[6] 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.

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

[8] Z. Hu, F. Lucchetti, C. Schlesinger, Y. Saxena, A. Freeman, S. Modak, A. Guha, and J. Biswas, "Deploying and evaluating llms to program service mobile robots," *IEEE Robotics and Automation Letters*, vol. PP, pp. 1–8, 03 2024.

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

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

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