2026 IEEE International Conference on Robotics and Automation (ICRA 2026)
June 1-5, 2026. Vienna, Austria

LLM-Driven Corrective Robot Operation Code Generation with Static Text-Based Simulation

Wenhao Wang[1], Yi Rong[1], Yanyan Li[2], Long Jiao[1], Jiawei Yuan[1]

***Abstract*— Recent advances in Large language models (LLMs) have demonstrated their promising capabilities of generating robot operation code to enable LLM-driven robots. To enhance the reliability of operation code generated by LLMs, corrective designs with feedback from the observation of executing code have been increasingly adopted in existing research. However, the code execution in these designs relies on either a physical experiment or a customized simulation environment, which limits their deployment due to the high configuration effort of the environment and the potential long execution time. In this paper, we explore the possibility of directly leveraging LLM to enable static simulation of robot operation code, and then leverage it to design a new reliable LLM-driven corrective robot operation code generation framework. Our framework configures the LLM as a static simulator with enhanced capabilities that reliably simulate robot code execution by interpreting actions, reasoning over state transitions, analyzing execution outcomes, and generating semantic observations that accurately capture trajectory dynamics. To validate the performance of our framework, we performed experiments on various operation tasks for different robots, including UAVs and small ground vehicles. The experiment results not only demonstrated the high accuracy of our static text-based simulation but also the reliable code generation of our LLM-driven corrective framework, which achieves a comparable performance with state-of-the-art research while does not rely on dynamic code execution using physical experiments or simulators.**

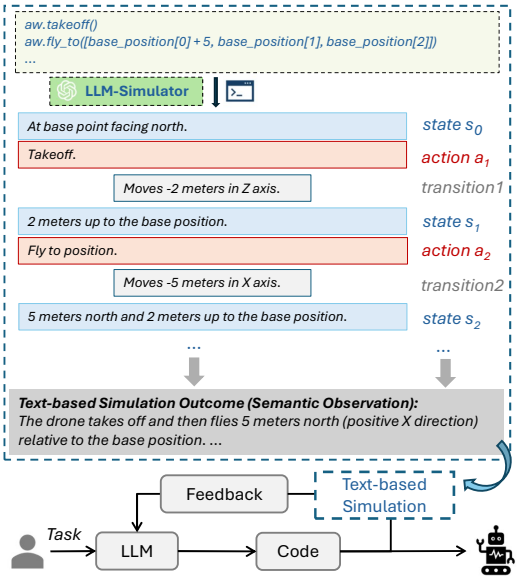

Fig. 1. An overview of LLM-driven corrective robot operation code generation with static text-based simulation.

I. INTRODUCTION

Robots are being increasingly deployed to execute tasks based on human instructions. However, designing a robot that has intelligence to perform complex tasks reliably remains challenging, as it demands both robust instruction interpretation and executing tasks with robust reasoning. Recent advances in LLMs [1]–[3] have demonstrated remarkable proficiency in robotic areas such as control [4], planning [5], and navigation [6]. Their strong context understanding and generation capabilities enable the robot to comprehend human instructions and generate corresponding robot operation code [7]–[10], thereby greatly simplifying the process of robot programming.

Unlike text generation applications, where semantic-level accuracy is sufficient, executing logically inconsistent or syntactically incorrect code on a robot can lead to unexpected outcomes and even unsafe robot behaviors, such as UAV crashes. To improve the reliability of LLMs' robot operation code generation, recent studies have incorporated a corrective process that leverages the observation of executing LLM-generated code in a physical experiment or simulator to identify issues and perform refinement [11]–[13]. While these corrective designs have shown their effectiveness in enhancing the reliability of LLM-generated operation code, they still face challenges from different aspects. Specifically, the configuration of a physical or simulation environment for robot code execution requires specialized expertise to support the tasks (e.g., designing scenes and modeling a robot's replica), especially for custom-built robots. For example, the code execution in ref [13] requires a specific design of mapping that transforms numerical state representation to the semantic description, which needs to be customized case-by-case. In addition, existing corrective designs require multiple rounds of interactive refinements, which can lead to a long execution time in the experiment or simulation when the task is complicated or involves time-consuming operations (e.g., monitor the area for 15 minutes). Therefore, removing the dependence on dynamic execution in a physical experiment or simulator while still maintaining the corrective feedback and refinement features for LLM-driven robot operation code generation becomes a challenging gap to close.

In this paper, we explore the possibility of directly em-

[1]Department of CIS, University of Massachusetts Dartmouth {wwang5, yrong, ljiao, jyuan}@umassd.edu
[2]Department of CSIS, California State University San Marcos. yali@csusm.edu

This work was partially supported by the US National Science Foundation awards 2318710 and 2318711, and UMass Dartmouth Internal Research Seed Funding Program.

Project is available at https://github.com/ai-uavsec/LLM-Driven-Static-Simulation

ploying LLMs for static text-based simulation over robot operation code and obtaining effective semantic observation to enable the refinement of LLM-driven robot operation code generation. Specifically, we propose a novel static text-based simulation solution powered by LLM to statically simulate the execution of robot operation code and generate accurate simulation outcomes. As shown in Fig. 1, our static simulation solution emulates the code execution by interpreting the actions encoded in the text of the code, reasoning about the corresponding state and environment-driven transitions, analyzing the next robot state after the action execution, and ultimately producing semantic observations that capture the robot's trajectory dynamics. On top of that, we propose our corrective robot code generation with static text-based simulation that unfolds as follows: 1) An LLM configured as a code generator first generates the initial version of robot operation code based on the task description from the user; 2) The LLM-based simulator "executes" the code and produces a semantic observation of the robot's trajectory; 3) An evaluator LLM analyzes the observation and identifies mismatches between the task description and the robot's trajectory, and then provides feedback that depicts the mismatched actions; 4) Guided by the feedback, the code generator corrects the mismatches. This iterative correction process continues until the evaluation confirms alignment between the code and task objectives, after which the final version of code is deployed to the robot for task execution.

We extensively evaluated our proposed framework for the operation code generation of UAV tasks with various levels of task complexities. Our results show that our static text-based simulation achieves over 97.5% of simulation accuracy compared with the widely adopted UAV simulators, i.e., AirSim [14] and PX4-Gazebo [15]. In addition, our corrective code generation framework delivers comparable robot execution performance as the state-of-the-art (SOTA) method relying on dynamic code execution in physical experiment or simulator, with an $85\%+$ success rate and $96.9\%+$ completeness on different UAV systems, i.e., $96.9\%+$ of required actions in all evaluated tasks are completed correctly and $85\%+$ of tasks are entirely completed without any error. To demonstrate the adaptability of our framework and its performance on real-world robot deployment, we also evaluated it for the operation code generation for UAVs and ground robots. Our experiment results show that our framework can also achieve high success rates ($87.5\%+$) and completeness ($96.9\%+$).

II. RELATED WORK

A. LLM-Driven Corrective Robot Code Generation

Using LLMs to generate operation codes for robot tasks has become a prevalent trend in recent research. By configuring LLMs with appropriate system prompting techniques, existing research has shown that LLMs have the potential to generate robot operation codes [7]–[10]. To further enhance the reliability of LLMs' output and support more complicated robot tasks, recent efforts have adopted corrective code generation such that the errors or mismatches are iteratively detected and corrected; therefore, the robot could perform the desired task accurately when executing the final code [16]–[18]. However, the physical execution during the correction process may increase the hardware cost and raise safety risks, as executing code with errors could cause irreparable damage to robots, such as UAV crashes. More recent studies proposed a simulation-based pipeline that eliminates the potential risks during physical execution [11], [13], [19]. For example, [13] leverages the AirSim [14] simulator to iteratively correct the UAV operation code until the code is ready for deployment. The authors also develop semantic observations rather than numerical representation [12] to describe the UAV trajectory to further improve performance. Robo-Instruct [19] fine-tuning LLM that checks LLM-generated robot programs with a simulator and revises them until correct.

B. LLM-based Simulation

Recent advances in LLMs [1]–[3] have demonstrated their exceptional capabilities in world modeling. Previous studies have explored the integration of LLMs into agent-based modeling and simulation within the social [20] and planning [21] domains. For example, BeSimulator [22] built an LLM-powered framework towards behavior simulation on the behavior tree. However, studies on text game show that current LLMs are not yet able to reliably act as text world simulators as they are likely to make errors when arithmetic, common-sense, or scientific knowledge is needed [23]. This limitation highlights the need for further research to examine the use of LLMs for the robot code simulation domain and develop methods to enhance the ability of LLMs to perform accurate and reliable simulations of robot operation code.

C. Prompt Engineering

Prompt engineering can effectively communicate and interact with LLM-driven tools [24]. Recent studies have increasingly used prompt engineering to improve the reliability of LLM-driven robotic systems. For example, PromptBook proposes a prompt framework as a system prompt to enhance the code generation [25]. On the one hand, prompt engineering leverages in-context learning [26] and few-shot learning [27] to enable LLM to learn knowledge within the given context and identify patterns from a limited set of examples. These techniques facilitate the generation of code that adheres to robot policy and how to ground task descriptions from a few examples [9]. On the other hand, CoT [28] prompts the LLM to articulate intermediate inference steps, which is valuable for robotic tasks that require sequential, stepwise decision-making. CoT encourages the production of code that aligns with each stage of the intended action plan. Previous studies have embedded CoT within the examples to guide LLMs through reasoning step-by-step [25], [29]. In this study, we utilize prompt engineering strategies to facilitate our text-based simulation.

III. METHOD

A. Overview

Fig. 2 presents the overall framework of our corrective code generation with text-based simulation. When given a

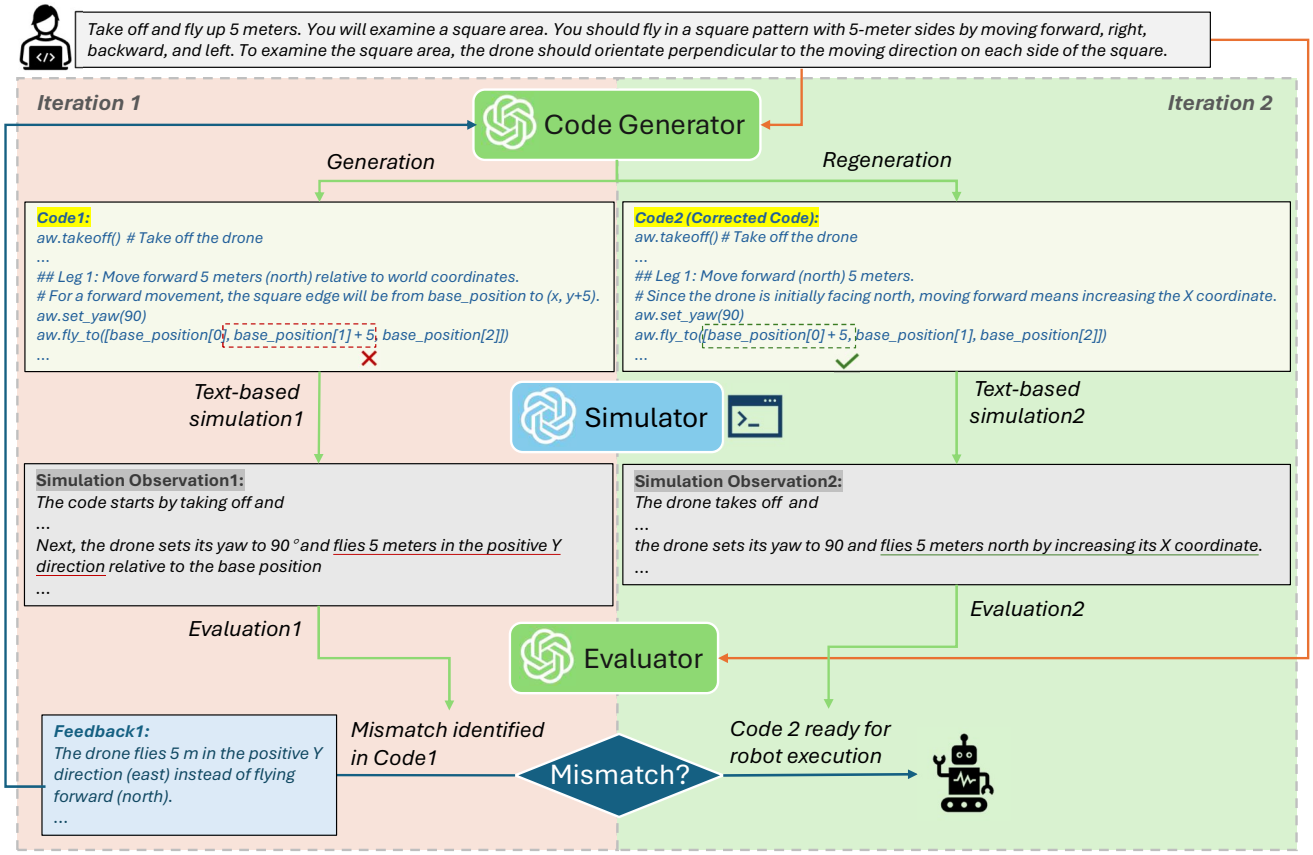

Fig. 2. An illustrative example of corrective code generation with text-based simulation. In the first iteration, the LLM-based simulator accurately produces an observation of UAV actions, while the evaluator identifies the mismatch and constructs feedback. Based on the feedback, the code generator corrects the mismatch and produces a valid code for robot operation in the second iteration.

task by the user, the code generator first produces the initial version of the robot operation code. Then our LLM-based simulator produces an observation of the robot's trajectory through text-based simulation of the code. After that, the evaluator analyzes the observation together with the task and generates feedback that depicts the mismatches (if any) in the robot's actions. Based on the feedback, the code generator regenerates the code that corrects the mismatches. This iterative "generation-simulation-evaluation" loop continues until the evaluation confirms task objectives are achieved or the maximum number of iterations is reached. The following sections present the detailed design of our code generation, text-based simulation, and evaluation.

B. Code Generation

The code generation in our method is achieved by configuring an LLM agent (code generator) with a robot operation-related system prompt. We adopt the strategies in the GSCE framework [10], which enhance the reasoning capabilities of LLMs in generating UAV operation code that aligns with task instructions. Additionally, the code generator regenerates the code that resolves the mismatches according to the feedback from the evaluator in section III-D. In detail, given a task description or evaluation feedback, the code generator produces a robot operation code that aims to accomplish the task or resolve the mismatches.

C. Static Text-Based Simulation using LLM

Given the robot operation code generated by the code generator, the goal of our static text-based simulation is to accurately interpret the robot's actions, reason about the state transitions, predict the robot states, and generate an observation of the robot's trajectory. Specifically, we formulate our text-based simulation using LLM as $O = \langle S, A, T, C \rangle$, where O denotes the robot trajectory observation produced by the simulation, $S = (s_0, s_1, s_2, \ldots, s_n)$ denotes the finite set of discrete states, A denotes the finite set of robot actions, $T : S \times A \to S$ denotes the state transition, and C denotes the text of robot operation code. The trajectory observation O captures a sequence of robot actions $l = (a_1, a_2, \ldots, a_n)$ that transitions robot from the initial state s_0 through intermediate states, to the final state s_n. O also embeds the history of the robot's trajectory dynamics for each robot action l and their transitions T after executing code C. This observation enables the subsequent evaluation to identify the mismatches and correct them in the next code generation. Therefore, the reliability of LLM-driven robotics depends critically on the accuracy of O.

To generate O, our design leverages the semantic reasoning capabilities of LLMs to simulate the execution of C

by interpreting its textual content rather than executing the code in a real simulator. The LLM implements a function $F : C \times S \times A \rightarrow S$ as a simulator that maps from a given code, current state, and action to the next state. Specifically, upon receiving a code script, the LLM-simulator simulates the execution of code, interprets the actions in the code, reasons about the corresponding state transitions, and predicts the next state. When the simulation completes, the LLM outputs its observation of the robot's trajectory as the simulation outcome.

To further enhance the reliability and accuracy of our text-based simulation, we further design a system prompt framework. The system prompt of LLM-simulator is composed of *role*, *APIs*, *policies*, and *examples* [1].

- **Role:** Defines the LLM agent responsible for simulating code execution by analyzing the provided code, inferring the intended actions, and outputting a description of the robot's trajectory.
- **APIs:** Provide definitions of robot action that guide the LLM in understanding the code's intent, enabling inference actions l and their corresponding state transitions T encoded within the text of the code C.
- **Policies:** Instruct the LLM with the code execution policies where the LLM lacks prior knowledge. The policies clarify the APIs usage, state transition rules, and environmental settings.
- **Examples:** Consist of pairs of code and trajectory that demonstrate how a robot operation code script C should be interpreted into semantic observation O. These examples guide the LLM via few-shot learning, enabling the LLM-simulator to generate observations O that follow the same style and structure.

As demonstrated in Fig 2, the structured system prompt enables the LLM simulation to accurately produce an observation of the UAV actions, which facilitates the subsequent evaluation to identify the mismatches.

D. Evaluation

In evaluation, the evaluator identifies the mismatches (if any) between the observation O and the task description, and then provides feedback specifying the mismatched actions. To ensure evaluation accuracy, we adopted the evaluator design in [13], which has demonstrated effectiveness in identifying the deviations between the UAV's trajectory and task description. The feedback from the evaluation provides the code generator with a clearer understanding of the objectives implied by the task description, thereby steering the generation of corrected code to better align with the task.

IV. EXPERIMENT

A. Experiment Setup

*1) **Experimental Environment**:* We implement our proposed method and comparison method using OpenAI "o3-mini" (o3-mini-2025-01-31) [30] and "o4-mini" (o4-mini-2025-04-16) [31] as the foundational LLMs. To measure the performance of the methods, the experiment is conducted on a quadcopter on both the "simple_flight" flight controller in AirSim [14] and the PX4 flight controller [15] in Gazebo [32] [2]. During the experiment, UAV state information was accessible from the simulators and utilized for performance measurement purposes. Furthermore, all experiments are averaged over three repetitions to mitigate the randomness of LLM generation [33].

*2) **Task Dataset**:* For the experiments, we adopt the Advanced task set from [13] as the benchmark dataset to measure the performance. The Advanced task set contains 20 UAV operation tasks with varying levels of complexity, each involving 6-19 actions to reach the goal state. The tasks are designed to emulate real-world UAV operation scenarios that require complex reasoning about the UAV's state in the world environment, such as flying complex geometric patterns with scenario requirements. To ensure the authenticity of the result, all tasks are manually validated in both AirSim and Gazebo to avoid potential simulation-induced errors that could affect the experiment result.

*3) **Compared Method**:* We compare our method against four methods:

- **Direct Analysis (Direct):** Uses the semantic checker from [11], where an LLM agent directly analyzes whether the generated code aligns with the task and provides feedback for code correction.
- **Simulator-based (Numerical):** Dynamically executes the generated code in a simulator to obtain numerical state observations, which are then evaluated to provide feedback for code correction [12].
- **Simulator-based (Semantic):** A SOTA method extends the Numerical method [12] by transforming numerical state observations produced from the simulator into semantic trajectory descriptions [13] to improve the code correction efficiency. During the experiments, we modified the transformation algorithm to accommodate different robots and simulators.

B. Evaluation Setup

*1) **Evaluation Metrics**:* Following [13], we evaluate performance using **Completeness** and **Success Rate (SR)**.

Completeness measures the proportion of actions in a given task that are executed correctly. It is computed as the ratio between the number of correctly executed actions and the total number of actions in the ground-truth sequence. This metric provides insight into performance throughout the intermediate execution process. For a given task i, the completeness is defined as:

$$\text{Completeness}_i = \frac{|a_i^{\text{correct}}|}{|l_i^{\text{gt}}|} \tag{1}$$

where (a_i^{correct}) denotes the count of actions correctly executed for task i, and (l_i^{gt}) is the total number of actions in the task's ground truth. The overall completeness is averaged over n tasks.

[1]The detailed design of the system prompt is provided in Appendix A

[2]Simulator configurations and setups are provided in Appendix B

SR reflects task-level reliability by measuring whether the robot successfully reaches the final goal state while following the correct sequence of actions that produce the intended state transitions. A task is considered successful only if the complete trajectory is executed without errors ($SR_i = 1,$ if $\text{Completeness}_i \equiv 1$).

*2) **Ground Truth**:* The ground truth is represented by a list of state transitions. Each state transition is a vector of four elements: $[x, y, z, \theta]$, where x, y, and z denote the robot's position changes in the North, East, and Down axes, and θ represents yaw rotation.

C. Result and Analysis

*1) **Overall Result**:* The overall performance of our proposed method and comparison methods on both simple_flight controller in AirSim and PX4 controller in Gazebo are summarized in Table I and Table II. For both LLM models, our LLM-simulator consistently supports reliable corrective code generation across different robot configurations, achieving success rates above 85% on the simple_flight controller and 86.7% on the PX4 controller. These results demonstrate both the reliability of our text-based simulation and its adaptability across diverse robot systems.

In particular, our method achieves performance comparable to the SOTA Semantic method [13] without requiring dynamic code execution in simulators that are explicitly designed to support robot simulations. This highlights that the proposed LLM-simulator can reliably conduct static text-based simulation of robot code to support corrective code generation. Furthermore, our method outperforms the Numerical method [12], indicating that the semantic observations generated by our LLM-simulator capture richer semantics about the robot's trajectory dynamics than the numerical representations. This enables an equally effective correction process as the SOTA Semantic method [13], but without the need for customizing algorithms for different robot configurations to transform numerical states into semantic descriptions. In contrast, the Direct method [11] yields unreliable performance, proving the limitations of directly configuring LLMs to analyze code and highlighting the necessity to strengthen LLM's capabilities for LLM-based simulation.

TABLE I
RESULTS OF UAV WITH SIMPLE_FLIGHT CONTROLLER

	o3-mini		o4-mini	
	SR	Completeness	SR	Completeness
Direct [11]	43.3%	70.9%	33.3%	57.6%
Numerical [12]	73.3%	92.4%	81.7%	96.1%
Semantic [13]	**85.0%**	**98.5%**	88.3%	98.1%
Ours	**85.0%**	97.0%	**90.0%**	**98.3%**

*2) **Text-Based Simulation Accuracy**:* To further evaluate the reliability of our text-based simulation, we compare the accuracy of the trajectory observations generated by the LLM-simulator against those obtained using the Semantic method [13] in the dynamic simulator. For each task in the Advanced task set, we construct a corresponding ground truth code (C^{correct}) that implements the task. The LLM-simulator is then used to statically simulate the execution of each C_i^{correct} and produce trajectory observations. The accuracy of the simulation is computed as $TP/N \times 100\%$, where TP denotes the number of observations that accurately capture the robot trajectory dynamics, and N is the total number of simulated code in C^{correct}.

TABLE II
RESULTS OF UAV WITH PX4 CONTROLLER

	o3-mini		o4-mini	
	SR	Completeness	SR	Completeness
Direct [11]	55.0%	77.5%	25.0%	50.9%
Numerical [12]	83.3%	97.1%	86.7%	96.9%
Semantic [13]	**88.3%**	**98.3%**	**93.3%**	**98.6%**
Ours	86.7%	97.7%	**93.3%**	96.9%

As shown in Table III, the LLM-simulator achieves 97.5% accuracy on "o3-mini" and 100% on "o4-mini" in capturing robot trajectory dynamics. These results demonstrate that the proposed LLM-simulator can reliably simulate the execution of robot actions, reason over the state transitions, and predict subsequent robot states. Therefore, the observation from the text-based simulation accurately reflects the intended robot's trajectory dynamics from the code, enabling the evaluator to effectively detect mismatched actions in the code.

TABLE III
TEXT-BASED SIMULATION OBSERVATION ACCURACY

	o3-mini	o4-mini
Semantic [13]	100.0%	100.0%
Ours	97.5%	100.0%

TABLE IV
EVALUATION ACCURACY OVER OBSERVATIONS FROM LLM-SIMULATOR

	o3-mini		o4-mini		Avg.
	C^{correct}	$C^{\text{incorrect}}$	C^{correct}	$C^{\text{incorrect}}$	
Semantic [13]	90.0%	91.7%	93.3%	93.3%	92.1%
Ours	91.7%	90.0%	93.3%	91.7%	91.7%

*3) **Text-Based Simulation Evaluation Accuracy**:* We further analyze whether the observations generated by the LLM-simulator can effectively support the evaluation process. Specifically, we compare the evaluation accuracy when using observations generated by our LLM-simulator against those produced from the Semantic method [13]. This experiment leverages the ground-truth code set C^{correct} in section IV-C.2, along with an additional set $C^{\text{incorrect}}$ that contains at least one error action in code for each task. The evaluation accuracy is computed as $(TP + TN)/N \times 100\%$, where TP denotes the number of correct evaluations on C^{correct} and $C^{\text{incorrect}}$ (i.e., correctly identifying whether the trajectory

matches with the task and providing a detailed explanation if mismatches are identified), and N is the total number of evaluated code scripts.

As presented in Table IV, our method achieves over 90% accuracy on "o3-mini" and 91.7% accuracy on "o4-mini". The results are comparable to the Semantic method [13] without the need for modifying algorithms for each robot and simulator to transform numerical states into semantic descriptions. The results also demonstrate that the observations from the LLM-simulator are sufficient to support effective evaluation as the SOTA Semantic method [13] and are adaptable to different robot systems.

However, due to the non-determinism of LLM generation [34], identical actions may yield semantically equivalent but syntactically different observations. For example, the action "fly 5 meters south" from the text-based simulation may be produced as "followed by 5 meters back (returning in the north-south direction)". While such descriptions remain interpretable to humans, their implicit ambiguity can cause misinterpretations for LLMs [35]. Consequently, observations generated by the Semantic method [13] exhibit slightly higher evaluation accuracy, as they are deterministic and free from linguistic variations.

D. Ablation Study on LLM-Simulator Design

We conduct an ablation study on the design of our LLM-simulator. Specifically, we investigate how different components of the LLM's system prompt affect the overall system performance. As specified in Section III-D, the *role* defines the LLM to be a simulator, and the *APIs* provide definitions of robot action APIs, they serve as the essential component to ensure the LLM performs the text-based simulation. Thus, they are retained in the ablation study. For the remaining components, we remove *policies*, *examples*, and both *policies* and *examples* to measure their impact on the overall system performance. The results in Table V show that removing either *policies* or *examples* leads to degradation in performance, while removing both yields the largest decline. Furthermore, removing *examples* produces a larger negative impact than removing *policies*, which is consistent with prior evidence that LLMs exhibit strong few-shot learning capabilities [27]. The results of the ablation study highlight that both instructing the LLM with code execution rules (*policies*) and providing demonstrations (*examples*) are essential for enhancing the performance of the LLM-simulator, and their combination yields the best overall performance.

TABLE V
OVERALL SYSTEM PERFORMANCE OVER LLM-SIMULATOR DESIGN

	Ours	w/o policies	w/o examples	w/o policies & examples
		o3-mini		
SR	**85.0**%	83.3%	81.7%	71.7%
Completeness	**97.0**%	94.9%	94.0%	90.8%
		o4-mini		
SR	**91.7**%	83.3%	83.3%	68.3%
Completeness	**97.7**%	96.1%	95.1%	93.1%

V. REAL-WORLD DEPLOYMENT

We validate our method through the deployment of physical robotic platforms. For consistency, the deployment is conducted using OpenAI's "o3-mini" and "o4-mini" models, and the performance of robot deployment is measured using the same metrics defined in section IV-A.

A. Robot Setup

*1) **UAV**:* We deploy our method on the Holybro X500 V2 quadcopter equipped with a Pixhawk 6X flight controller running PX4 firmware. For the task set, we select two tasks from each complexity level of the Advanced task set, resulting in a total of 8 tasks for deployment. In the deployment, the user types the task description into a ground station computer that runs our method to generate the UAV operation code (the computer is connected to the Internet for accessing the OpenAI API). The generated code is then transmitted to the flight controller via MAVSDK [36] for task execution. During the flight, the UAV's state information is collected through MAVSDK for performance measurement.

*2) **Ground Vehicle**:* We further evaluate the method on a ROSMASTER X3 ground vehicle controlled by ROS [37]. We then design 8 tasks for the ground vehicle to form patterns when driving on the ground. In the deployment, the user accesses the onboard computer remotely and types in the task descriptions, then the onboard computer (connected to the Internet for accessing the OpenAI API) runs our method to generate the ground vehicle operation code and then executes the code to control the vehicle's movement.

B. Result and Analysis

The results of the real-world deployment are summarized in Table VI. Overall, our method demonstrates consistent performance across both UAV and ground vehicle platforms, validating the effectiveness of the proposed LLM-simulator in real-world settings. On both the UAV and ground vehicle tasks, our approach achieves high success rates and completeness, confirming the reliability and adaptability of our framework to different robot systems. These findings highlight that the static text-based simulation framework is reliable in supporting corrective code generation in real-world robot execution.

TABLE VI
ROBOT DEPLOYMENT PERFORMANCE

	o3-mini		o4-mini	
	SR	Completeness	SR	Completeness
UAV	87.5%	98.6%	91.7%	97.9%
Ground Vehicle	87.5%	96.9%	87.5%	97.2%

VI. CONCLUSIONS

This paper presented an enhanced LLM-Driven corrective robot operation code generation framework. Different

from existing solutions that require dynamic execution in a physical or simulation environment for code feedback and refinement, our framework is designed with a novel static text-based simulation solution powered by LLM, and hence addresses the challenges brought by the configuration of a dynamic code execution environment and potential long execution time for refinement. The experiment results on both different UAV systems validated the performance of our simulation solution in terms of both simulation accuracy and the reliability of robot operation code generation. Moreover, real-world deployments on physical robots further demonstrated the adaptability of our framework across different configurations and environments.

APPENDIX

A. LLM-Simulator System Prompt

Role: You will analyze and infer the intention of the provided Python drone control code, then generate a description of the drone actions in one paragraph.
You should focus on the code, not the comments. Because code will be the actual actions of the drone.

APIs: Here are the available functions for the drone when you infer the drone actions and their state transitions:
aw.takeoff() - takes off the drone.
aw.land() - lands the drone.
aw.fly_to([x, y, z]) - flies the drone to the position specified as a list of three arguments corresponding to world XYZ coordinates.
aw.get_yaw() - returns the current yaw of the drone in degrees.
aw.set_yaw(yaw) - sets the yaw of the drone to the specified value in degrees.
aw.get_drone_position() - returns the current position of the drone as a list of 3 floats corresponding to world XYZ coordinates.

Policies: Important drone coordinate directional information and action policies:
1. The horizontal axes are Y and X, the vertical axis is Z.
2. When rotating the drone, turning right or clockwise means positive, the yaw angle should increase.
3. *aw.fly_to([x, y, z])* function uses NED coordinate system (world coordinates), positive X axis is North/forward, positive Y axis is East/right, positive Z axis is Down. When flying up, the Z value should decrease. When flying down, the Z value should increase.
4. The drone is initialized facing north (Yaw = 0 degrees).
5. Map Yaw angle degree from -180 to 180, for example: map 270 to -90.

Examples: Below are some examples; you should follow the output format in these examples in your answers.
Query: "
current_position = aw.get_drone_position()
aw.fly_to([current_position[0], current_position[1], current_position[2] - 10]) "
Answer:
"The drone flies 10 meters up."
Query: "
aw.takeoff()
current_position = aw.get_drone_position()
aw.fly_to([current_position[0], current_position[1], current_position[2] - 5])
current_position = aw.get_drone_position()
aw.set_yaw(90)
aw.fly_to([current_position[0] + 5, current_position[1], current_position[2]])
current_position = aw.get_drone_position()
current_yaw = get_yaw()
aw.set_yaw(90 + current_yaw)
aw.fly_to([current_position[0], current_position[1] + 5, current_position[2]])
current_position = aw.get_drone_position()
current_yaw = get_yaw()
aw.set_yaw(90 + current_yaw)
aw.fly_to([current_position[0] - 5, current_position[1], current_position[2]])
current_position = aw.get_drone_position()
current_yaw = get_yaw()
aw.set_yaw(90 + current_yaw)
aw.fly_to([current_position[0], current_position[1] - 5, current_position[2]]) "
Answer:
"First, the drone takes off and climbs 5 m. Next, it rotates 90 degrees clockwise (now facing east) and flies to a point 5 m north along the world X axis. It then rotates 90 degrees clockwise (now facing south), and flies 5 m west along the world Y axis. Then rotate 90 degrees clockwise (now facing west), fly 5 m south along world X axis. Then rotate 90 degrees clockwise (now facing north) and fly 5 m east along world Y axis."

B. Simulator Configurations

In our experiments, we employ two widely used simulators: AirSim [14] and Gazebo [32].

1) AirSim: Developed by Microsoft, AirSim provides a high-fidelity virtual environment for research in autonomous systems such as UAVs, self-driving cars, and ground robots. Built on Unreal Engine [38], it supports realistic rendering of robot dynamics. In our experiments, we use the pre-built "block" [39] environment, which provides a clear and open airspace suitable for UAV testing.

2) Gazebo: Gazebo [32] models physical dynamics such as gravity, friction, and contact forces, making it applicable to both aerial and ground robots. It integrates seamlessly with the ROS operating system [37]. For our experiments, we use the PX4 [15] software-in-the-loop (SITL) setup to control a

simulated quadcopter within a virtual world environment of Gazebo.

REFERENCES

[1] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[2] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican, *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.

[3] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[4] A. Tagliabue, K. Kondo, T. Zhao, M. Peterson, C. T. Tewari, and J. P. How, "Real: Resilience and adaptation using large language models on autonomous aerial robots," in *2024 IEEE 63rd Conference on Decision and Control (CDC)*, 2024, pp. 1539–1546.

[5] M. Mohanan and A. Salgoankar, "A survey of robotic motion planning in dynamic environments," *Robotics and Autonomous Systems*, vol. 100, pp. 171–185, 2018.

[6] B. Yu, H. Kasaei, and M. Cao, "L3mvn: Leveraging large language models for visual target navigation," in *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2023, pp. 3554–3560.

[7] S. H. Vemprala, R. Bonatti, A. Bucker, and A. Kapoor, "Chatgpt for robotics: Design principles and model abilities," *IEEE Access*, vol. 12, pp. 55 682–55 696, 2024.

[8] G. Chen, X. Yu, N. Ling, and L. Zhong, "Typefly: Flying drones with large language model," *arXiv preprint arXiv:2312.14950*, 2023.

[9] 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 *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 9493–9500.

[10] W. Wang, Y. Li, L. Jiao, and J. Yuan, "Gsce: a prompt framework with enhanced reasoning for reliable llm-driven drone control," in *2025 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2025, pp. 441–448.

[11] Y. Chen, J. Arkin, C. Dawson, Y. Zhang, N. Roy, and C. Fan, "Autotamp: Autoregressive task and motion planning with llms as translators and checkers," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 2024, pp. 6695–6702.

[12] L. Sun, D. K. Jha, C. Hori, S. Jain, R. Corcodel, X. Zhu, M. Tomizuka, and D. Romeres, "Interactive planning using large language models for partially observable robotic tasks," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 2024, pp. 14 054–14 061.

[13] W. Wang, Y. Li, L. Jiao, and J. Yuan, "Large language model-driven closed-loop UAV operation with semantic observations," *IEEE Internet of Things Journal*, 2025.

[14] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and Service Robotics: Results of the 11th International Conference*. Springer, 2018, pp. 621–635.

[15] L. Meier, D. Honegger, and M. Pollefeys, "PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015, pp. 6235–6240.

[16] F. Joublin, A. Ceravola, P. Smirnov, F. Ocker, J. Deigmoeller, A. Belardinelli, C. Wang, S. Hasler, D. Tanneberg, and M. Gienger, "Copal: Corrective planning of robot actions with large language models," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 2024, pp. 8664–8670.

[17] S. S. Raman, V. Cohen, I. Idrees, E. Rosen, R. Mooney, S. Tellex, and D. Paulius, "Cape: Corrective actions from precondition errors using large language models," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 2024, pp. 14 070–14 077.

[18] A. Curtis, N. Kumar, J. Cao, T. Lozano-Pérez, and L. P. Kaelbling, "Trust the PRoc3s: Solving long-horizon robotics problems with LLMs and constraint satisfaction," in *8th Annual Conference on Robot Learning*, 2024, pp. 1362–1383.

[19] Z. Hu, J. J. Li, A. Guha, and J. Biswas, "Robo-instruct: Simulator-augmented instruction alignment for finetuning code LLMs," in *Second Conference on Language Modeling*, 2025.

[20] C. Gao, X. Lan, N. Li, Y. Yuan, J. Ding, Z. Zhou, F. Xu, and Y. Li, "Large language models empowered agent-based modeling and simulation: A survey and perspectives," *Humanities and Social Sciences Communications*, vol. 11, no. 1, pp. 1–24, 2024.

[21] C. Yang, X. Wang, J. Jiang, Q. Zhang, and X. Huang, "Evaluating world models with llm for decision making," *arXiv preprint arXiv:2411.08794*, 2024.

[22] J. Wang, B. Li, X. Wang, F. Li, Y. Wu, J. Chen, and X. Yi, "Besimulator: A large language model powered text-based behavior simulator," *arXiv preprint arXiv:2409.15865*, 2024.

[23] R. Wang, G. Todd, Z. Xiao, X. Yuan, M.-A. Côté, P. Clark, and P. Jansen, "Can language models serve as text-based world simulators?" *arXiv preprint arXiv:2406.06485*, 2024.

[24] G. Marvin, N. Hellen, D. Jjingo, and J. Nakatumba-Nabende, "Prompt engineering in large language models," in *International conference on data intelligence and cognitive informatics*. Springer, 2023, pp. 387–402.

[25] M. G. Arenas, T. Xiao, S. Singh, V. Jain, A. Ren, Q. Vuong, J. Varley, A. Herzog, I. Leal, S. Kirmani, M. Prats, D. Sadigh, V. Sindhwani, K. Rao, J. Liang, and A. Zeng, "How to prompt your robot: A promptbook for manipulation skills with code as policies," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 2024, pp. 4340–4348.

[26] B. Y. Lin, A. Ravichander, X. Lu, N. Dziri, M. Sclar, K. Chandu, C. Bhagavatula, and Y. Choi, "The unlocking spell on base llms: Rethinking alignment via in-context learning," *arXiv preprint arXiv:2312.01552*, 2023.

[27] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[28] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[29] Z. Zhou, J. Song, K. Yao, Z. Shu, and L. Ma, "Isr-llm: Iterative self-refined large language model for long-horizon sequential task planning," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 2024, pp. 2081–2088.

[30] OpenAI Models. (2025) o3-mini. Accessed: 25-May-2025. [Online]. Available: https://platform.openai.com/docs/models/o3-mini

[31] OpenAI Models. (2025) o4-mini. Accessed: 08-Sep-2025. [Online]. Available: https://platform.openai.com/docs/models/o4-mini

[32] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004, pp. 2149–2154 vol.3.

[33] B. Atil, A. Chittams, L. Fu, F. Ture, L. Xu, and B. Baldwin, "Llm stability: A detailed analysis with some surprises," *arXiv preprint arXiv:2408.04667*, 2024.

[34] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "An empirical study of the non-determinism of chatgpt in code generation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, pp. 1–28, 2025.

[35] A. Keluskar, A. Bhattacharjee, and H. Liu, "Do llms understand ambiguity in text? a case study in open-world question answering," in *2024 IEEE International Conference on Big Data (BigData)*, 2024, pp. 7485–7490.

[36] MAVSDK. (2025) Mavsdk: The mavlink sdk. Accessed: 08-Sep-2025. [Online]. Available: https://github.com/mavlink/MAVSDK

[37] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, 2009, p. 5.

[38] Epic Games, "Unreal Engine," 2018. [Online]. Available: https://www.unrealengine.com

[39] Microsoft Research, "Setup blocks environment for airsim," 2025, accessed: 2-Feb-2025. [Online]. Available: https://microsoft.github.io/AirSim/unreal_blocks/