IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

ModuLoop : Low-Level Code Generation using Modular Synthesizer and Closed-Loop Debugger for Robotic Control

Gina Yoon, *Student Member, IEEE*, Sumin Lee, *Student Member, IEEE*, Joo Yong Sim, *Member, IEEE*

***Abstract*—Large Language Models (LLMs) have demonstrated impressive performance across various domains, including code generation and problem solving. However, their application in robotic control—particularly in low-level tasks that require precise manipulation, real-time feedback, and environment-dependent execution—remains limited. To address this challenge, we propose the Closed-Loop Modular Code Synthesizer framework. This framework leverages a pre-trained LLM without any task-specific fine-tuning to perform modular code planning and generation, and iteratively executes the generated code while inserting debugging probes to observe its behavior. This closed-loop structure facilitates systematic debugging and refinement, ultimately producing executable control programs. We apply the proposed framework to the calibration of an RGB-D camera and a robotic arm, validating its effectiveness in real-world settings. Furthermore, through a subsequent pick-and-place task, we demonstrate not only the accuracy of the calibration but also the potential extensibility of the framework. Across both tasks, the framework achieved high execution accuracy and autonomy, illustrating the practicality and scalability of LLM-based robotic control using our framework.**

***Index Terms*—AI-Enabled Robotics, Calibration and Identification, Task Planning, Code Generation, LLM-based Control**

I. Introduction

LARGE Language Models (LLMs) have demonstrated remarkable capabilities across a range of domains, including natural language understanding, code generation, and symbolic reasoning. Advanced models such as the GPT series [1] have shown the ability to generate executable code from natural language instructions, highlighting their growing utility in industrial and applied settings [2]. In the field of robotics, recent approaches have sought to enhance LLMs' task interpretation and planning capabilities through large scale dataset fine-tuning or few-shot learning [3]–[5].

Against this backdrop, recent LLM-based robotic control studies have generally progressed in two directions: (i) translating natural language commands into high-level plans while relying on external modules for execution [4], [6], and (ii) directly generating executable low-level control or perception code through user-defined APIs from natural language instructions [5], [7], [8]. While the former limits LLM involvement in execution, the latter often suffers from prompt dependency and poor generalization—issues particularly critical in low-level control where precise motion and real-time error handling are required.

To address these challenges and enable LLMs to participate actively in the control loop, we propose **ModuLoop**, a framework that enables LLMs to participate in the full control loop of low-level robotic tasks. Fig. 1 illustrates the overall architecture of the ModuLoop based robotic control system. The pipeline begins with a high-level task command expressed in natural language. ModuLoop decomposes the given instruction into fine-grained subtasks and converts each subtask into executable Python code. Based on execution feedback—including runtime errors and performance metrics—the framework dynamically refines and corrects the generated code, forming a closed-loop structure in which the robot autonomously learns and improves through iterative execution.

We validate ModuLoop in two representative tasks: (1) hand–eye calibration between a camera and robotic manipulator, and (2) a pick-and-place manipulation task requiring perception and motion planning. Although originally designed for calibration without any user-defined APIs, the framework demonstrated extensibility to manipulation tasks with minimal perception guidance.

Our main contributions are threefold:

1) A simulation-based LLM feedback loop that secures collision-free executable coordinates for hand–eye calibration, thereby enhancing data efficiency and accuracy.
2) A modular code synthesizer framework that decomposes high-level natural language commands into executable low-level Python modules for robot control.
3) A closed-loop debugging mechanism that uses execution feedback—such as runtime errors and accuracy metrics—to autonomously revise and improve generated code.

Together, these contributions position ModuLoop as a practical and generalizable framework that elevates LLMs from passive planners to autonomous agents capable of generating, executing, and adapting robot control code in real-world environments.

II. Related Work

A. LLM-Driven Robotic Control

Recent research on integrating Large Language Models (LLMs) into robotic systems has generally followed two

Manuscript received: May, 24, 2025; Revised: August, 29, 2025; Accepted; October, 3, 2025.

This paper was recommended for publication by Editor Chao-Bo Yan upon evaluation of the Associate Editor and Reviewers' comments.

This work was supported by National Research Foundation of Korea(NRF) grant (No. RS-2025-02216282, RS-2025-16070288), Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2022-II220025) and by Ministry of Trade, Industry and Energy (MOTIE RS 2023 00258591). *(Gina Yoon and Sumin Lee are co-first authors.) (Corresponding author: Joo Yong Sim.)*

Gina Yoon, Sumin Lee, and Joo Yong Sim are with the Department of Mechanical Systems Engineering, Sookmyung Women's University, Cheongpa-ro 47-gil 100, Yongsan-gu, Seoul, 04310, South Korea (e-mail: ppippi272@sookmyung.ac.kr, sstone@sookmyung.ac.kr, jysim@sookmyung.ac.kr).

Digital Object Identifier (DOI): see top of this page.

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

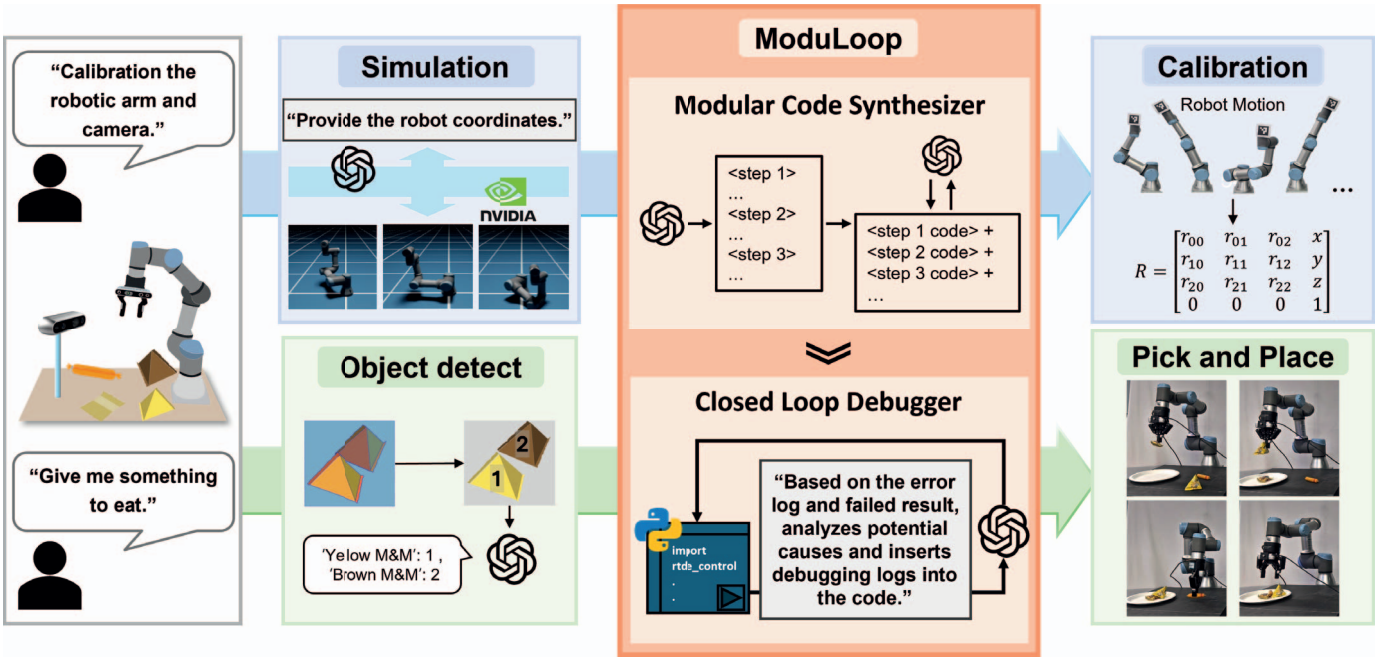

Fig. 1. The architecture of an LLM-based control framework that enables a robotic arm to autonomously execute tasks based on natural language commands. ModuLoop serves as a code synthesis framework that generates executable control code for both low-level hand–eye calibration and high-level pick-and-place manipulation. Before code generation, the system collects task-specific environmental information — simulation-based reachable coordinate verification for calibration, and SAM- and LLM-based object detection for pick-and-place. Using these inputs, ModuLoop performs modular code synthesis and closed-loop debugging, producing verified control code that enables accurate calibration and autonomous manipulation.

directions. The first focuses on high-level action planning, as seen in systems such as *SayCan* [6] and *PaLM-E, RT-1, RT-2* [4], [9], [10], which combine LLMs with affordance models or multimodal inputs to translate language commands into abstract action sequences. However, these approaches do not produce executable low-level control code and instead rely on downstream modules.

The second direction explores directly generating robot control code from natural language. *Code-as-Policies* [5], for example, uses few-shot prompting with API specifications and annotated examples, while *ProgPrompt* [7] employs structured prompts with action primitives, and *RobotGPT* [11] generates and simulates code via ChatGPT with reinforcement learning. Despite producing executable code, most approaches remain limited by their reliance on structured inputs and lack of feedback-based refinement, reducing adaptability across environments. RobotScript [12] also demonstrates LLM-based code generation. It not only integrates perception tools like AnyGrasp [13] for grip pose prediction but also exposes them at the API level for direct invocation within the generated script.

B. Closed-Loop Refinement for Code Generation using LLM

Existing LLM-based code generation systems (e.g., Codex [14]) operate in an open-loop manner, producing static code without execution feedback. *AutoGen* [15] introduces a multi-agent framework, while *Reflexion* [16] allows a single LLM to refine reasoning through self-reflection, both enabling iterative improvement. However, these methods remain confined to abstract programming and lack applicability to robotic control, where continuous sensing and physical feedback are essential. *MCCoder* [17] extends this line by incorporating sensor feedback for microcontroller-based robots, but its reliance on hardware-specific prompts and rigid templates limits generalizability. In contrast, ModuLoop autonomously synthesizes and improves executable control code by directly leveraging real-world feedback.

C. Hand-eye Calibration

Hand-eye calibration [18] estimates the spatial transformation between a robot's end-effector and a mounted camera, enabling vision-based manipulation. It aligns the coordinate frames of the robot and the camera so that perceived object positions can be accurately used for motion control. Recent learning-based methods automate this process, either by regressing extrinsics from RGB or point clouds without markers [19], or by aligning point clouds via registration pipelines [20], [21]. These reduce marker dependence and manual effort but remain sensitive to training data distribution and rely on accurate initial alignment for reliable performance. ModuLoop overcomes these issues by enabling automatic, adaptive calibration without prior data or manual tuning.

III. Low-Level Control with Workspace Validation via Simulation and Closed-Loop Modular Code Generation

A. Iterative Interaction between LLM and Simulator for Physically Valid Motion Planning

To verify the physical feasibility of motion trajectories prior to real-world execution, we employ a simulation-based

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

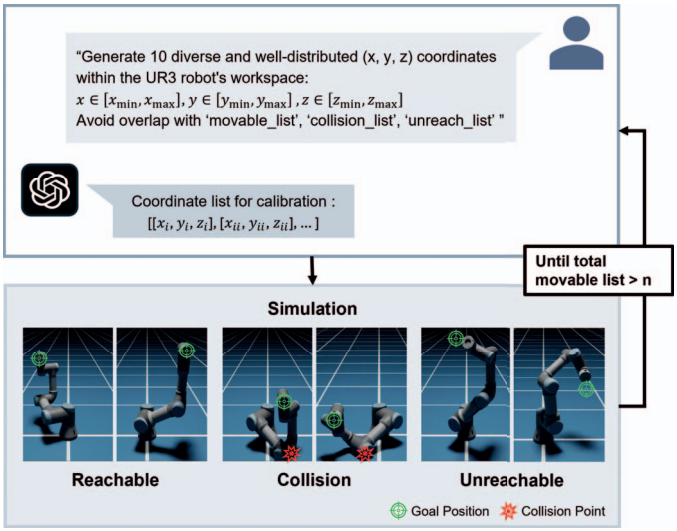

Fig. 2. A pipeline where candidate positions are pre-validated in simulation for reachability and collision before being used in the closed-loop modular code generation process. Candidate positions are tested in simulation and classified as reachable, collision, or unreachable. These results are fed back to refine future generations until a sufficient number of reachable positions is obtained.

validation environment in Isaac Sim [22] to check workspace reachability and potential collisions. As shown in Fig. 2, within each iteration of the LLM–simulator feedback loop, the LLM generates calibration targets—specifically, a set of 3D end-effector position candidates:

$$\mathcal{C}_{\text{gpt}} = G(\mathcal{W}, L_{\text{rea}}, L_{\text{col}}, L_{\text{unr}})$$

where $W \subset \mathbb{R}^3$ is the robot's workspace constraint, $L_{\text{rea}}, L_{\text{col}}, L_{\text{unr}}$ are sets of previously observed reachable, collision, and unreachable positions, respectively, $G(\cdot)$ is a function that calls the LLM with prior outcomes as prompts, acting as a coordinate generator that produces candidate end-effector positions.

The candidate positions C_{gpt} are then filtered through inverse kinematics (IK):

$$C_{\text{gen}} = \{c \in C_{\text{gpt}} \mid IK(c) \neq \emptyset\}$$

where, C_{gen} represents the subset of position candidates from C_{gpt} that are kinematically reachable, i.e., those that pass IK filtering. Each $c \in C_{\text{gen}}$ denotes a single 3D end-effector position that satisfies this condition. IK serves both to reject infeasible samples and to provide executable joint trajectories.

The simulator evaluates each pose $c \in C_{\text{gen}}$ using a function F_{sim}, which classifies them into:

$$(C_{\text{rea}}, C_{\text{col}}, C_{\text{unr}}) = F_{\text{sim}}(C_{\text{gen}})$$

where:
C_{rea}: positions successfully reached without collisions
C_{col}: positions causing self-collision or collisions with the environment
C_{unr}: positions the robot fails to reach

These labeled results are re-fed into the LLM prompt to inform the next generation step, forming a feedback loop.

TABLE I
COMPARISON OF REACHABLE COORDINATE GENERATION ACCURACY.

Metric	Analytical (DH-based)	Analytical + Robot Verification	Simulation-based
Accuracy (%)	56.67	83.33	100.00

The process repeats until enough reachable positions are collected. Through this process, the framework effectively integrates LLM-guided position generation with physics-based validation, enabling efficient collection of valid and executable robot coordinates.

Table I compares three approaches for calculating reachable coordinates in the robot workspace. The first is an analytical method using UR3 Denavit–Hartenberg (DH) parameters and joint limits. The second extends this by verifying inverse kinematics (IK) on the physical UR3 robot, yielding results closer to actual execution. The third is our simulation-based calibration method, where candidate coordinates are validated in simulation to assess reachability.

Results show that the DH-based method has the lowest accuracy due to its reliance on analytical computation, while adding real-robot verification improves performance. Purely analytical approaches are prone to collision or infeasibility, whereas simulation-based validation provides the most reliable outcomes.

In conclusion, the proposed framework integrates LLM-guided candidate generation with simulation-based validation, ensuring that only physically feasible and collision-free coordinates are retained. Simulation is employed exclusively in this validation stage, while the validated coordinates are subsequently used for real robot code generation, thereby enhancing execution fidelity.

B. ModuLoop for Hand-eye Calibration

A closed-loop framework is proposed to automate the calibration process between the robotic arm and the camera using GPT-4o. The methodology consists of two main stages: (1) Modular Code Synthesizer (MCS), where an initial Python script is automatically generated from a natural language prompt, and (2) Closed-Loop Debugger (CLD), in which the script is iteratively refined based on execution results. An overview of the entire workflow is shown in Fig. 3.

1) Environment: The process begins with the definition of the overarching goal of low-level control. In this case, the objective is to calculate the 3D-to-3D transformation matrix between the robot's workspace and the camera's coordinate system. To achieve this, a structured prompt is provided to GPT, containing essential information such as the robot's IP address and sensor specifications. In addition, movable coordinates obtained from simulation are also included in the prompt.

2) Modular Code Synthesizer: The code generation stage systematically decomposes the calibration task and transforms it into automatically executable code. This process consists of three steps.

- **Task Decomposition & Planning:** The LLM decomposes the given calibration task into a series of inde-

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

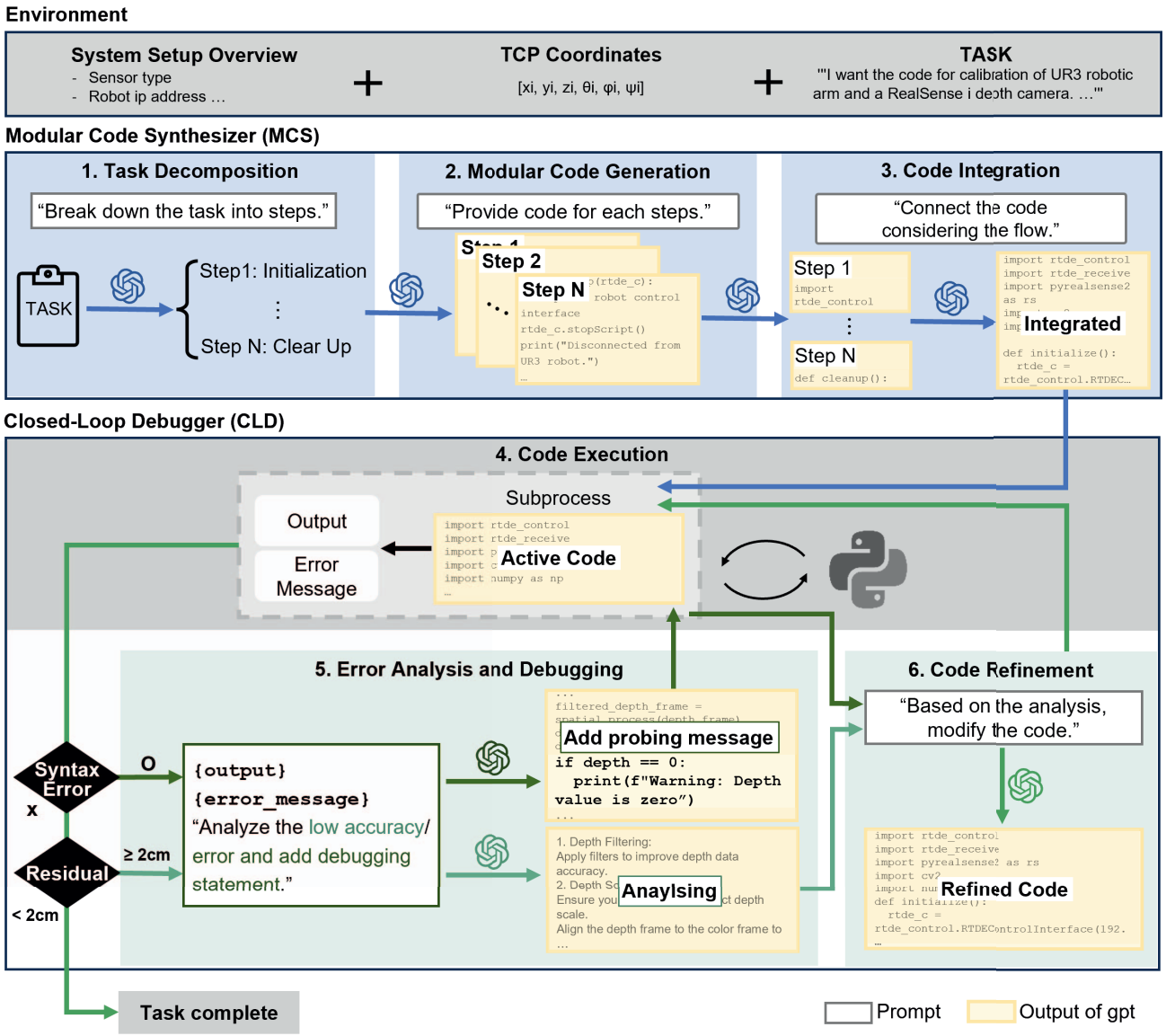

Fig. 3. ModuLoop: A LLM-based control framework architecture for autonomously executing low-level robotic tasks from natural language instructions. Given a task, along with essential information such as reachable TCP coordinates (from simulation), sensor specifications, and robot IP, the framework proceeds through six stages. (1) The task is decomposed into sequential code-generation steps. (2) A code block is generated for each step. (3) The individual blocks are integrated into an initial executable script. (4) The code is executed via a subprocess for real-time interaction. (5) Errors are analyzed: syntax errors are handled through direct debugging, while low accuracy triggers debugging-message insertion and further diagnosis. (6) Based on analysis, the code is refined iteratively until both correctness and task accuracy criteria are met.

pendent subtasks. For example, to compute the transformation matrix, the procedure may include steps such as moving to predefined coordinates, acquiring sensor data, and performing matrix computation. Each subtask is then passed as a structured prompt to the next LLM instance.

- **Modular Code Generation:** Each subtask is implemented as a Python function or code block.
- **Code Integration:** The generated code blocks are assembled into a coherent executable script while maintaining logical consistency. This script is executed via subprocesses, enabling the system to observe robot motions and process camera data in real time.

3) Closed-Loop Debugger: The code generation stage provides an initial Python script, but in real robotic environments, unexpected runtime errors or insufficient accuracy may prevent reliable execution. To address these issues, we propose a closed-loop feedback structure that diagnoses execution outcomes and iteratively refines the code (Algorithm 1). In this process, the LLM acts not only as a code generator but also as an active participant in a continuous loop of execution, analysis, and refinement.

At each iteration, the current code C is executed in the robotic environment, producing an output o and an error message e. If an error occurs, the Analyzer formulates hypotheses about possible causes and re-executes the code with probing messages inserted to validate these hypotheses. The results are then passed to the Refiner, which applies targeted modifications accordingly. This process is exemplified in Fig. 4,

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

Algorithm 1: Closed-Loop Feedback for Calibration

Input: Initial code C_0; Evaluator E; Analyzer A; Refiner R; Max iterations T
Output: Refined calibration code C^*
$C \leftarrow C_0$;
$t \leftarrow 0$;
while $t < T$ **do**
  /* *Execute current code* */
  (output o, error e) $\leftarrow$ Run(C)
  if $e \neq \emptyset$ **then**
    /* *If error occurs, insert probes and fix* */
    $C_d \leftarrow A$.add_probing(C, o, e)
    $(o', e') \leftarrow$ Run(C_d)
    $C \leftarrow R$.apply_fix(C_d, o', e')
  else
    /* *If no error, evaluate calibration quality* */
    result $r \leftarrow E$.evaluate(o) **if** r == *"inaccurate"* **then**
      /* *If inaccurate, diagnose and refine* */
      $diag \leftarrow A$.analyze_accuracy(C, o)
      $C \leftarrow R$.apply_improvement($C, o, diag$)
    else if r == *"calibration success"* **then**
      /* *If successful, return calibrated code* */
      return C
  $t \leftarrow t + 1$;
return *Failure: calibration did not converge*

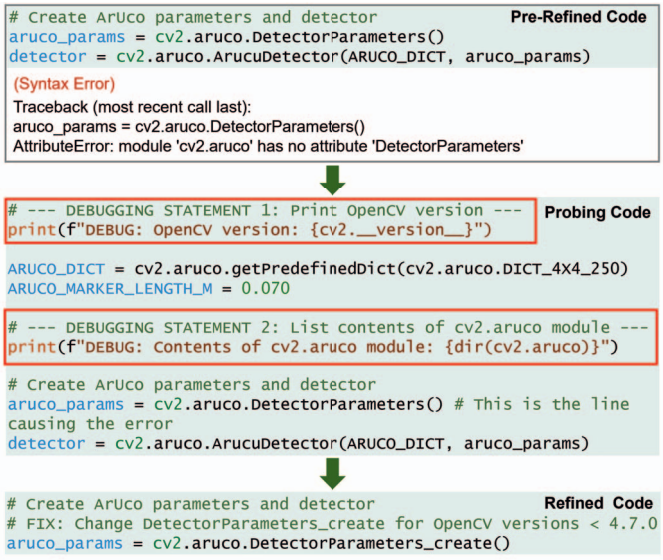

Fig. 4. Syntax Error Debugging in Closed-Loop Process. This case illustrates the process of resolving a version mismatch error in OpenCV's ArUco module. Based on the pre-refined code and its output message, probing code is generated to identify the root cause, and the issue is resolved by refining the code accordingly.

which shows a syntax error case in OpenCV's ArUco module resolved through probing and refinement.

If no runtime error is detected, the Evaluator assesses calibration accuracy based on the output messages. The output includes either "calibration success" or "calibration inaccurate." If classified as "inaccurate," the Analyzer diagnoses potential causes, and the Refiner updates the code to improve accuracy. An example of this process is shown in Fig. 5, where calibration accuracy is improved through additional alignment

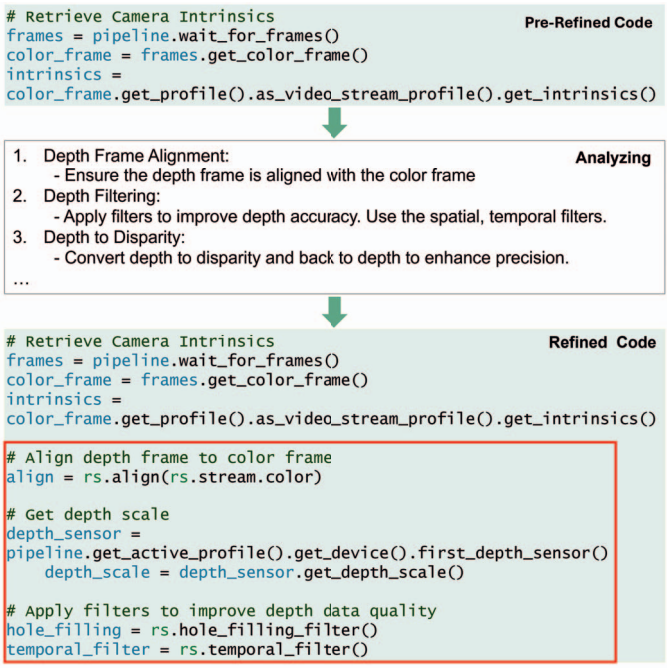

Fig. 5. Accuracy Improvement Debugging in Closed-Loop Process. This case demonstrates how adding depth alignment and filtering to the camera pipeline improves analysis accuracy. Based on the initial code and its output messages, the system analyzes potential causes of low accuracy and recommends refinement strategies.

and filtering strategies.

This loop continues until one of two termination conditions is met: (1) the calibration reaches the required accuracy, in which case the refined script C^* is returned, or (2) the maximum number of iterations T, is reached, in which case the process is considered a failure. Through this mechanism, the LLM functions as a self-correcting agent, capable of iterative adaptation, real-time diagnosis, and execution-driven refinement throughout the calibration process.

IV. Performance of Modular Code Generation and Closed-Loop Debugger

A. Experimental Setup

This experiment was designed to quantitatively evaluate the performance of two key components of the proposed closed-loop calibration framework: MCS and CLD. The objective is to assess their contribution to the robustness of code execution and calibration accuracy.

The experiment was conducted using a UR3 robotic manipulator and an Intel RealSense D435i depth camera. An ArUco marker was attached to the robot's tool center point (TCP), and the camera was positioned 1.3 meters in front of the robot, facing it directly. The robot is controlled through the RTDE (Real-Time Data Exchange) interface, and the control system operates at a frequency of 125 Hz. Communication between the robot and the control system is conducted via TCP/IP over a local network.

To assess the effectiveness of the MCS and CLD, we included an additional model to provide a baseline for comparison. This model, referred to as the Single-Path Generator

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

TABLE II
COMPARISON OF CALIBRATION PERFORMANCE ACROSS DIFFERENT CODE GENERATION AND FEEDBACK STRATEGIES. THE ITERATION WAS TERMINATED AND DEEMED UNSUCCESSFUL AFTER TEN ATTEMPTS. SUCCESS CASES: CALIBRATION ERROR < 0.02 M. ABBREVIATIONS: CAP - CODE AS POLICIES, SPG – SINGLE PATH GENERATOR, MCS – MODULAR CODE SYNTHESIZER.

Model		Code Gen Success (%)	Accuracy Success (%)	Verified Error (Success Cases, m) mean / median	Verified Error (All Cases, m) mean / median
GPT-4o	CaP with Basic Robot Control API	-	-	-	-
	CaP with Full API	90.0	-	-	0.038/ 0.039
	ProgPrompt	66.67	-	-	0.026/ 0.026
	SPG	-	-	-	-
	SPG + Debugger	6.67	-	-	0.802/ 0.802
	SPG + Debugger w/ Probing	10.0	-	-	0.485/ 0.062
	MCS	23.33	6.67	0.029/ 0.029	0.308/ 0.283
	MCS + Debugger	73.33	60.0	0.174/ 0.019	0.074/ 0.014
	MCS + Debugger w/ Probing	96.67	86.67	0.048/ 0.011	0.170/ 0.012
GPT-4.1 mini	MCS + Debugger w/ Probing	73.33	40.0	0.085/ 0.068	0.337 /0.068
Gemini	MCS + Debugger w/ Probing	90.0	73.33	0.064/ 0.064	0.091/ 0.065

(SPG), generates the entire code in a single pass without any task planning, directly producing code from the given task description in a single prompt. In the feedback stage, two additional comparison models were included. The first is an open-loop model that performs no feedback at all, and the second is a limited closed-loop model without probing. The latter follows the closed-loop structure but does not insert diagnostic code to observe runtime behavior, relying solely on the final execution results as feedback. Thus, it represents a minimally applied feedback loop. All experiments were repeated 30 times to ensure statistical reliability.

B. Experimental Results

Each model was evaluated according to the following three criteria:

1) Successful code generation of without syntax errors
2) Satisfaction of a predefined accuracy threshold (< 2 cm)
3) Positional error between the robot's actual TCP position and the transformed position computed from the camera

The experimental results are summarized in Table II. The combination of the Modular Code Synthesizer (MCS) and the Closed-Loop Debugger (CLD) achieved the highest success rate and accuracy, whereas the single-prompt approach showed substantially lower performance, indicating that task decomposition is critical for generating structurally and logically complete programs.

As shown in Fig. 6, integrating the CLD with the MCS improved calibration efficiency. Through probing-based debugging, more initial samples were successfully calibrated, and fewer iterations were required for convergence. This demonstrates that error-hypothesis probing accelerates convergence and enhances robustness.

This study compared ModuLoop with ProgPrompt [7] and Code as Policies (CaP) [5] on the same hand–eye calibration task. ProgPrompt is a prompting technique that leverages code structures instead of natural language, where available robot actions and environmental objects are presented as Python functions and lists. By providing example tasks together with these definitions, the LLM is guided to generate new task plans directly in the form of code.

Code as Policies is a paradigm where the LLM constructs control code around user-defined APIs, generating programs

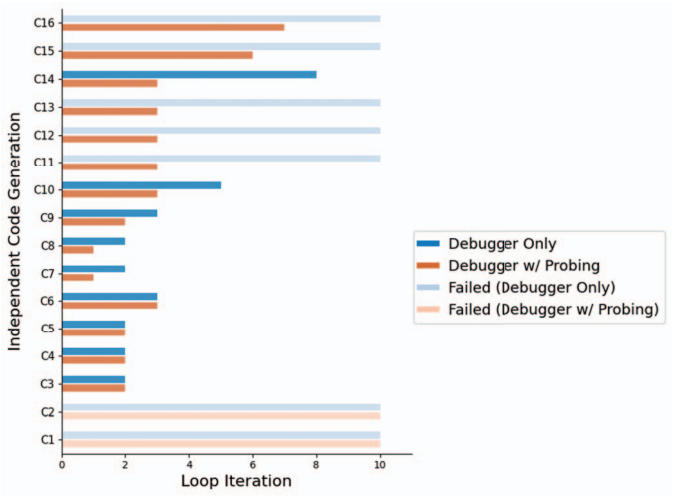

Fig. 6. Comparison of feedback loop iterations between models with and without the proposed Closed-Loop Debugger. Each bar represents the number of refinement iterations required for successful task completion across 16 code generation experiments.

mainly by invoking provided APIs and, when necessary, defining new ones recursively. In our experiments, we evaluated two conditions. In *CaP with Full API*, the LLM was given a rich set of APIs covering not only low-level robot and sensor operations but also calibration procedures, along with hints. In contrast, *CaP with Basic Robot Control API* provided only simple functions for primitive robot control, while essential information as in ModuLoop was supplied for the rest.

ProgPrompt and CaP with Full API achieved relatively strong performance but were constrained by their reliance on predefined APIs and the absence of feedback for error correction. ModuLoop, by comparison, does not depend on predefined APIs and directly generates low-level control code, which is iteratively refined through feedback, thereby demonstrating robustness and intuitive applicability even in precise robotic tasks.

We further evaluated ModuLoop with different LLM variants. GPT-4.1-mini achieved slightly lower success rates than GPT-4o, but demonstrated advantages in cost and response speed. This indicates that it can serve as a cost- and time-efficient alternative, albeit with reduced reliability. Gemini

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

achieved code-generation and accuracy comparable to GPT-4o, but the derived transformation matrices fell below the required accuracy threshold, indicating limited calibration stability.

V. Evaluation of the Generalizability and Practicality of the Closed-Loop Code Generation and Debugging Framework

To validate the calibration-derived transformation matrix and assess ModuLoop's generality, we applied it to a representative pick-and-place task involving object recognition, coordinate alignment, motion planning, and grasp planning—all handled through LLM-based reasoning and code generation. The LLM received natural language instructions, target poses from perception, and motion planning guidelines.

Unlike calibration, where closed-loop debugging was feasible, executing pick-and-place trials risked collisions and environmental changes. We therefore adopted static debugging, supplying GPT-4o with a structured evaluation checklist to anticipate and reason about potential failure cases.

A. Object detection for pick and place

When a natural language instruction is given, the system executes the task through a multi-stage pipeline. RGB-D images from front and gripper-mounted cameras are processed by the Segment Anything Model [23], which segments object masks and extracts their center coordinates.

Each object is assigned a unique label, and the annotated images with the user command are provided to the LLM, which contextually interprets the scene to identify target objects rather than performing simple classification. For example, given "I'm hungry. Can you get me some snacks?", the LLM selects relevant items such as "sausages" or "chocolate," returning their names and IDs from each view (Fig. 7).

The LLM also considers object geometry and context to decide if rotation is needed for precise pick-and-place execution.

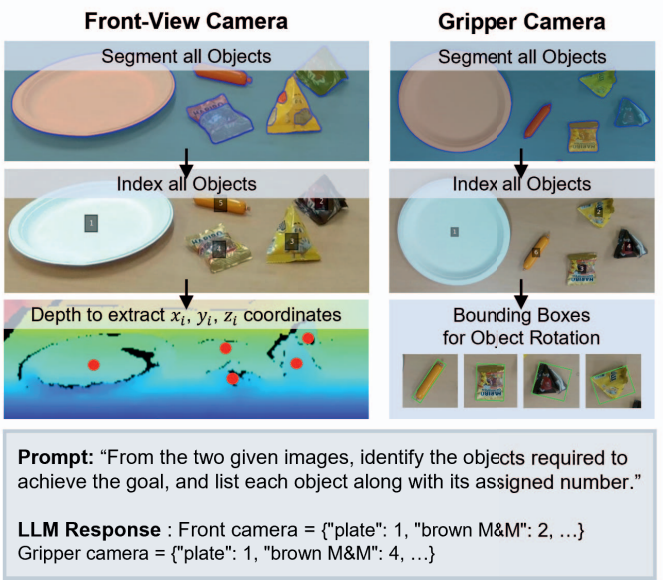

Fig. 7. LLM-based Object Recognition for Pick-and-Place Tasks. RGB-D images from both front and gripper-mounted cameras are processed by the Segment Anything Model (SAM) to segment object masks and extract their center coordinates. The annotated images and the natural-language command are then provided to the LLM, which contextually interprets the scene to identify the target object and determine whether rotation is required for precise manipulation.

B. Quantitative Evaluation of ModuLoop on Pick-and-Place Tasks

We evaluated ModuLoop on five pick-and-place tasks of increasing complexity and linguistic difficulty (see Table III), each executed 25 times using three code generation methods: (1) Single-Prompt Code Generation, (2) Modular Code Synthesizer, and (3) ModuLoop.

The tasks were specifically structured to assess the system's capabilities in perception, reasoning, and control, with increasing complexity. in Fig. 8. shows example scenes of a robotic arm performing each of the five pick-and-place tasks based on natural language instructions. The main characteristics of each task are as follows:

- **Simple 1**: Basic object targeting and orientation control
- **Moderate 1**: Logical reasoning involving multiple objects and inference of color mixing
- **Moderate 2**: Semantic categorization of objects and orientation control
- **Moderate 3**: Distance-based ordering and applying appropriate orientations for identical objects located at different positions
- **Hard 1**: Spatial reasoning, precise relative positioning, and orientation control

As shown in Table IV, ModuLoop achieved the highest success rates, particularly in complex tasks (Tasks 3–5) requiring semantic reasoning, spatial understanding, and precise motion planning, outperforming the Single-Prompt baseline. These results underscore the value of structured command decomposition and GPT-based debugging for translating complex manipulation commands into executable code.

While this study confirms the feasibility of LLM-based low-level code generation in calibration and pick-and-place tasks, we acknowledge a key limitation: ModuLoop currently relies only on minimal APIs that provide object positions, limiting its applicability to relatively simple tasks. Contact-rich manipulation, such as assembly or opening a drawer, requires richer environmental understanding—including perception modules, motion planners, and value map composition provided as APIs as exemplified in works such as VoxPoser [8].

VI. Conclusion

This study verified that pre-trained large language models (LLMs) can autonomously generate and execute low-level robot control code without further training. The ModuLoop pipeline demonstrated high execution accuracy in camera-to-robot calibration and pick-and-place tasks, confirming the feasibility of LLM-based control in physical environments.

By positioning the LLM as the central agent in the control process, this work demonstrates the feasibility of using pre-trained LLMs to translate natural language into robot-executable code. Its ability to handle complex control logic without manual programming or task-specific training un-

IEEE Robotics and Automation Letters (RA-L) paper, presented at ICRA 2026, Vienna, Austria. Cite as RA-L paper.

TABLE III
CAPABILITIES REQUIRED BY EACH TASK TO GUIDE LLM-BASED CODE GENERATION.

Task ID	Task Description	Key Capabilities Required per Task (increasing complexity)					
		Class Reasoning	Logical Inference	Multi Object	Orientation Control	Sequencing	Spatial Reasoning
Simple 1	Pick up the sausage on a plate.				✓		
Moderate 1	Put paints for coloring the Eggplant into the box.		✓	✓			
Moderate 2	I'm hungry, can you put some snacks on a plate?	✓		✓	✓		
Moderate 3	Pick up Mentos in order from farthest to nearest.			✓	✓	✓	
Hard 1	Move the red block next to the blue block.			✓	✓		✓

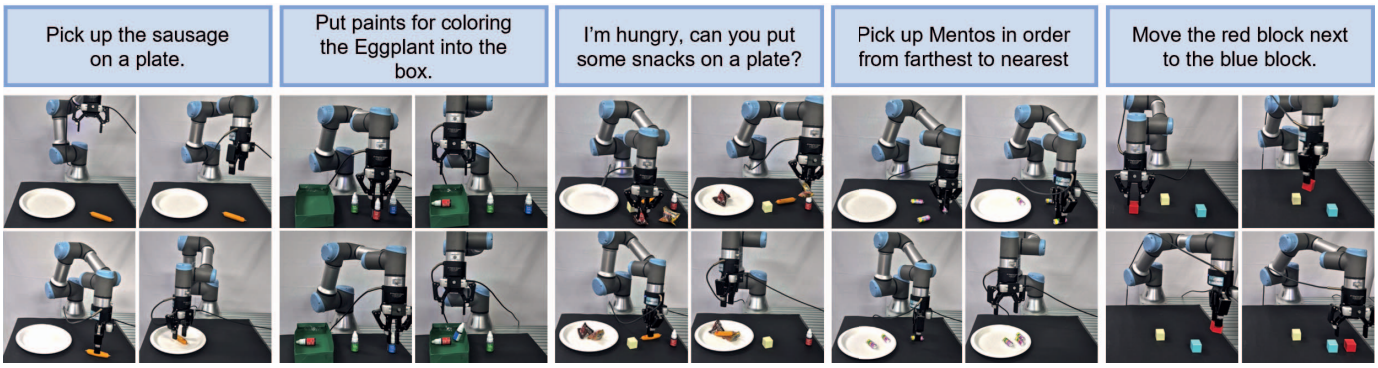

Fig. 8. Pick-and-place tasks performed by a robot based on language instructions. Each task is visualized as a 2×2 image sequence, ordered from top-left to bottom-right to show the temporal progression of execution.

TABLE IV
SUCCESS RATES(%) OF CODE GENERATION METHODS ACROSS TASKS

Task ID	Single Code Generator	Modular Code Synthesizer	ModuLoop
Simple 1	60	88	**96**
Moderate 1	64	80	**96**
Moderate 2	36	60	**92**
Moderate 3	12	76	**92**
Hard 1	24	48	**60**

derscores the approach's practicality. However, due to the inherent structure of LLMs, generating complete control code introduces noticeable latency, which can limit responsiveness.

In future work, we will extend the framework to real-world industrial applications such as process and factory automation, and evaluate its generality and hardware-independence across diverse robotic platforms beyond the UR3.

REFERENCES

[1] OpenAI, "Gpt-4o," https://openai.com/index/gpt-4o, 2024, accessed: 2024-04-10.

[2] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *ACM Trans. Softw. Eng. Methodol.*, Jul. 2025.

[3] T. Brown *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[4] B. Zitkovich *et al.*, "Rt-2: Vision-language-action models transfer web knowledge to robotic control," in *Proceedings of The 7th Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, J. Tan, M. Toussaint, and K. Darvish, Eds., vol. 229. PMLR, 06–09 Nov 2023, pp. 2165–2183.

[5] J. Liang *et al.*, "Code as policies: Language model programs for embodied control," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2023, pp. 9493–9500.

[6] A. Brohan *et al.*, "Do as i can, not as i say: Grounding language in robotic affordances," in *Conference on robot learning*. PMLR, 2023, pp. 287–318.

[7] I. Singh *et al.*, "Progprompt: Generating situated robot task plans using large language models," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2023, pp. 11 523–11 530.

[8] W. Huang, C. Wang, R. Zhang, Y. Li, J. Wu, and L. Fei-Fei, "Voxposer: Composable 3d value maps for robotic manipulation with language models," in *7th Annual Conference on Robot Learning*, 2023.

[9] D. Driess *et al.*, "Palm-e: An embodied multimodal language model," *CoRR*, vol. abs/2303.03378, 2023.

[10] A. Brohan *et al.*, "Rt-1: Robotics transformer for real-world control at scale," in *Robotics: Science and Systems (RSS)*, 2023.

[11] Y. Jin *et al.*, "Robotgpt: Robot manipulation learning from chatgpt," *IEEE Robotics and Automation Letters*, vol. 9, no. 3, pp. 2543–2550, 2024.

[12] J. Chen *et al.*, "Roboscript: Code generation for free-form manipulation tasks across real and simulation," *CoRR*, vol. abs/2402.14623, 2024.

[13] H.-S. Fang *et al.*, "Anygrasp: Robust and efficient grasp perception in spatial and temporal domains," *IEEE Transactions on Robotics (T-RO)*, 2023.

[14] M. Chen *et al.*, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021.

[15] Q. Wu *et al.*, "Autogen: Enabling next-gen LLM applications via multi-agent conversations," in *First Conference on Language Modeling*, 2024.

[16] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 36, pp. 8634–8652, 2023.

[17] Y. Li *et al.*, "Mccoder: Streamlining motion control with llm-assisted code generation and rigorous verification," *CoRR*, vol. abs/2410.15154, 2024.

[18] R. Horaud and F. Dornaika, "Hand-eye calibration," *The international journal of robotics research*, vol. 14, no. 3, pp. 195–210, 1995.

[19] A. Falisse, S. D. Uhlrich, A. S. Chaudhari, J. L. Hicks, and S. L. Delp, "Marker data enhancement for markerless motion capture," *IEEE Transactions on Biomedical Engineering*, 2025.

[20] T. Tang, M. Liu, W. Xu, and C. Lu, "Kalib: Markerless hand-eye calibration with keypoint tracking," *arXiv preprint arXiv:2408.10562*, 2024.

[21] L. Li, X. Yang, R. Wang, and X. Zhang, "Automatic robot hand-eye calibration enabled by learning-based 3d vision," *Journal of Intelligent & Robotic Systems*, vol. 110, no. 3, p. 130, 2024.

[22] NVIDIA, "Isaac Sim," https://developer.nvidia.com/isaac-sim, 2021, accessed: 2025-04-30.

[23] A. Kirillov *et al.*, "Segment anything," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2023, pp. 4015–4026.