HyCodePolicy: Hybrid Language Controllers for Multimodal Monitoring and Decision in Embodied Agents

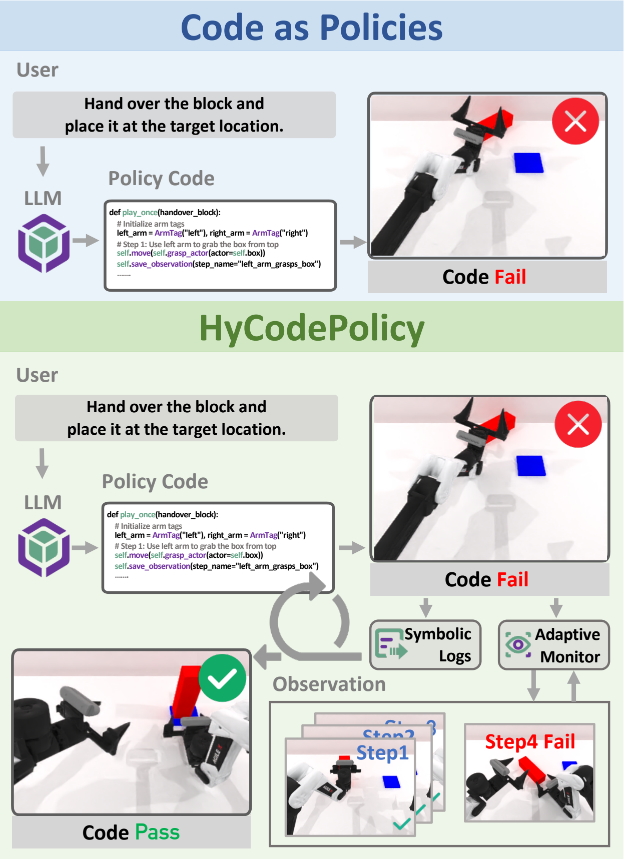
Yibin Liu[3,4*] Zhixuan Liang[2,5*‡] Zanxin Chen[5,6*] Tianxing Chen[2,6] Mengkang Hu[2]
Wanxi Dong[7] Congsheng Xu[1] Zhaoming Han[4] Yusen Qin[4,8] Yao Mu[1,5†]
[1]SJTU ScaleLab [2]HKU MMLab [3]NEU [4]D-Robotics [5]Shanghai AI Lab
[6]SZU [7]SUSTech [8]THU

[*]Equal Contribution [‡]Project Lead [†]Corresponding Author

liuyibin@stumail.neu.edu.cn, zxliang@cs.hku.hk, yaomarkmu@gmail.com

Abstract

Recent advances in multi-modal large language models (MLLMs) offer powerful perceptual grounding for code policy generation in embodied agents. However, most existing systems lack effective mechanisms to adaptively monitor execution and iteratively repair policies in response to failures. In this work, we introduce HyCodePolicy, a hybrid language-based control framework that closes the loop between code synthesis, geometry-aware grounding, perceptual monitoring, and targeted repair. Given a natural language instruction, our system first decomposes it into hierarchical sub-goals and generates an initial program grounded in object-centric geometric primitives. HyCodePolicy then executes the program in simulation, with a vision-language model (VLM) monitoring designated checkpoints to identify and localize failures and inferring their underlying causes. By integrating structured execution logs that capture program-level events with VLM-derived perceptual feedback, HyCodePolicy pinpoints root causes of failures and applies targeted code repairs. This hybrid dual feedback mechanism enables self-correcting program synthesis with minimal human intervention. Our results demonstrate that HyCodePolicy significantly enhances the robustness and sample efficiency of robot manipulation policies, offering a scalable strategy for incorporating multi-modal reasoning into autonomous decision-making pipelines.



```
def play_once(handover_block):
    # Initialize arm tags
    left_arm = ArmTag("left"), right_arm = ArmTag("right")
    # Step 1: Use left arm to grab the box from top
    self.move(self.grasp_actor(actor=self.box))
    self.save_observation(step_name="left_arm_grasps_box")
    .......
```

```
def play_once(handover_block):
    # Initialize arm tags
    left_arm = ArmTag("left"), right_arm = ArmTag("right")
    # Step 1: Use left arm to grab the box from top
    self.move(self.grasp_actor(actor=self.box))
    self.save_observation(step_name="left_arm_grasps_box")
    .......
```

Figure 1. Overview figure of **HyCodePolicy**, a closed-loop framework for language-conditioned manipulation with hybrid program synthesis, monitoring, and repair.

1. Introduction

The burgeoning capabilities of large language models (LLMs) and their multi-modal counterparts (MLLMs) are rapidly transforming the landscape of artificial intelligence, opening unprecedented avenues for robot planning. From this broad perspective, we focus on language-grounded manipulation, where robots leverage LLMs to interpret high-level natural language instructions, reason about complex tasks, and execute them in physical environments [1, 14]. This paradigm provides an opportunity to democratize robot programming, moving beyond tedious explicit coding to intuitive linguistic instruction. However, a fundamental challenge persists that effectively bridging the rich semantic expressiveness of natural language with precise, structured, and physically grounded representations for reliable robot execution.

Prior works have made considerable strides in synthesizing robot actions from language, ranging from direct LLM-generated plans to formal logic representations (*e.g.* code-as-policy) approaches [17, 21, 30]. These methods typically commit to a one-shot generation of the complete behavior plan, relying entirely on that single attempt being correct. Yet, real-world robotic tasks are inherently uncertain due to perception noise, execution errors, and dynamic environments. Current systems frequently lack robust mechanisms to adaptively monitor task execution, detect and diagnose failures, and then repair the robot's behavior in a closed loop [13]. This limitation undermines the robustness, efficiency, and real-world reliability of language-conditioned robot policies, often requiring extensive human intervention for debugging and recovery.

To address this critical gap, we introduce **HyCodePolicy**, a novel hybrid language-based control framework designed for robust, self-correcting robotic manipulation. HyCodePolicy systematically unifies code synthesis, geometric grounding, multi-modal monitoring within a closed-loop programming cycle. Rather than treating generated code as a static output, it treats each program as an evolving hypothesis that can be actively validated, evaluated, and corrected via perceptual cues and symbolic reasoning.

HyCodePolicy comprises four synergistic components: (1) High-level language intent grounding through hierarchical subgoal decomposition and geometrically informed program synthesis; (2) Simulated execution coupled with symbolic logging and concurrent vision-language model (VLM) observations; (3) Hybrid failure attribution fusing symbolic and perceptual diagnostics to infer causal error hypotheses; and (4) Iterative program repair, achieving the closed-loop control via targeted and interpretable code updates. This hybrid feedback mechanism enables self-correcting program synthesis with minimal human supervision.

We conduct extensive experiments on RoboTwin Platform [26] demonstrating the effectiveness of HyCodePolicy and showing the improvement of task success rates from 47.4% to 63.9% and from 62.1% to 71.3% in different settings. Additionally, HyCodePolicy reduces convergence iterations from 2.42 to 1.76, highlighting its ability to improve both robustness and efficiency in dynamic, real-world environments. These results underscore the practical impact of our framework in enhancing robotic manipulation tasks.

Our key contributions are:

- **A Novel Closed-Loop Control Framework:** We propose **HyCodePolicy**, a pioneering architecture that seamlessly integrates language-conditioned program synthesis with adaptive multimodal monitoring and iterative repair, improving the robustness and self-correction of robot policies.
- **Hybrid Grounded Feedback for Causal Repair:** We develop a unique hybrid feedback mechanism that fuses symbolic execution logs with Vision-Language Model (VLM)-based perceptual observations. This enables precise, causally-grounded failure attribution and drives targeted code repair, supported by geometric primitives for physically executable policies.
- **Demonstrated Robustness and Efficient Interface:** We empirically demonstrate that HyCodePolicy significantly enhances the robustness and sample efficiency of robot manipulation policies across diverse tasks. This is further demonstrated in **Bi2Code**, a re-engineered modular interface designed to optimize structured prompting and multimodal tracing for effective deployment.

The remainder of this paper is organized as follows: Section 2 reviews related work on language grounding for robotics and LLM-guided program repair. Section 3 details the design and implementation of HyCodePolicy. Section 4 presents our experimental setup and empirical results. Finally, Section 5 concludes and discusses future directions. Our code is open-sourced at RoboTwin-Platform.

2. Related Work

2.1. Robotic Manipulation Planning with Language Grounding

The integration of large language models (LLMs) into robotic manipulation has led to significant advancements in language-conditioned planning [1, 4, 10–12, 19, 20, 24, 25, 28]. Bridging the gap between the semantic expressiveness of language and the structured representations remains a critical challenge for LLM-based task planning.

Recent research focuses on symbolic and embedding-based approaches. Firstly, symbolic methods, such as those discussed by Cohen et al. [7], provide interpretability and enforce constraints, while embedding-based approaches offer generality but lack transparency. The Embodied Agent Interface [16] introduces a standardized framework for integrating LLMs with robotic agents. Text-to-plan techniques like SayCan [1] and Lang2LTL [22] focus on converting language into structured plans, while Code-as-Symbolic-Planner [6] and GenCHiP [6] focus on constraint-compliant policy generation.

Moreover, language-to-program pipelines, such as Code-as-Policies [17] and ProgPrompt [30], aim to enhance robot code generation for grounded executability. Additionally, multi-modal methods, like VIMA [14] and EmbodiedGPT [2, 24, 25], combine vision and language to enable generalization across diverse tasks.

HyCodePolicy extends these approaches by integrating structured program synthesis with symbolic-perceptual feedback for closed-loop planning and self-correction, as described in Section 3.

2.2. MLLM-Guided Failure Diagnosis and Program Repair

Studies on MLLM-guided failure attribution focus on feedback-driven model refinement. Approaches like Self-Debugging [5] and Self-Refine [18, 23] explore iterative correction through self-generated explanations, but they are tested often only under idealized conditions [13].

Executable program interfaces have been developed for failure attribution and repair. CodeAct [32] and INTERVENOR [31] enhance multi-turn repair by modeling agent reasoning as code. Safety-focused methods, such as SafetyChip [34] and SAFER [15], incorporate formal reasoning to enforce task constraints and ensure safety.

HyCodePolicy enhances failure localization and repair by fusing multimodal perceptual feedback with symbolic state, supporting interpretable, robust and adaptive behavior in real-world tasks. Unlike prior work relying on introspection or static logic, our method integrates dynamic feedback to drive targeted repair.

3. Method

In contrast to prior Code-as-policy frameworks that treat program generation as a one-shot synthesis process, HyCodePolicy introduces a flexible, closed-loop architecture in which code becomes not only an execution medium but also a vehicle for perception, self-monitoring, and autonomous refinement. The core insight is to reinterpret the generated program as an evolving hypothesis—one that is subject to empirical validation and continuous revision.

This perspective enables two key properties: *falsifiability* and *evolvability*. A program is falsifiable in the sense that its execution within a simulated environment exposes its limitations, such as geometric infeasibility or logical contradictions, which can be detected through runtime perceptual signals. It is evolvable because it participates in a monitor-diagnose-repair cycle, allowing it to self-correct and improve over time. This reframing supports a feedback-driven programming loop, where code actively engages in its own refinement, effectively realizing *code-as-monitor* within the broader system.

As shown in Figure 2, HyCodePolicy comprises four tightly integrated phases: (1) **Grounding high-level language intent** through hierarchical subgoal decomposition and geometrically informed program synthesis; (2) **Simulated execution and symbolic-perceptual monitoring** with structured logging and vision-language observation; (3) **Hybrid failure attribution** through fusion of symbolic and visual diagnostics; and (4) **Closed-Loop Autonomy via Adaptive Monitoring and Iterative Code Evolution** that achieves the closed-loop control via targeted and interpretable code updates.

3.1. Grounding High-Level Intent in Code

HyCodePolicy grounds high-level task intent by decomposing natural language instructions into structured subgoals and synthesizing executable Python programs aligned with geometric affordances. Given a language instruction (e.g., *"Handover the block"*), a task name, and optional examples, a language model $\mathcal{L}$ produces a sequence of N semantically coherent subgoals:

$$\mathcal{S} = \{s_1, s_2, \dots, s_N\} = \mathcal{L}(T)$$

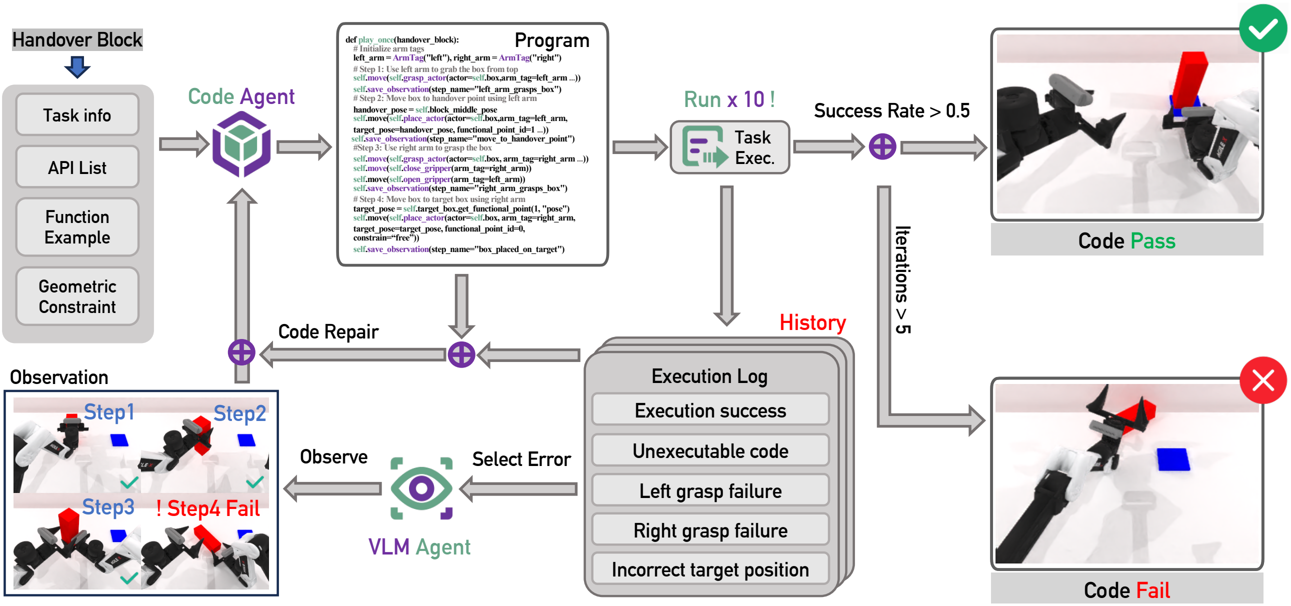
Each subgoal s_i represents a high-level behavioral unit (e.g., "pick up the blue block") and serves as a constraint for subsequent program synthesis.

Codes are generated through structured prompting conditioned on three elements: a general API list, exemplar function calls, and subgoal constraints. This frames the synthesis process as a constrained, structured prediction task over the program space, enabling functional and syntactic validity.

To ensure the resulting programs are not only logically coherent but also physically executable, the system integrates geometric reasoning via a library of **Geometric Operation Primitives**. These primitives abstract physical constraints into two representational categories:

Point Operation Primitives ($\mathcal{P}$): define key spatial targets:

p_{grasp}: Stable grasp point.
p_{place}: Placement support point.



```
def play_once(handover_block):
  # Initialize arm tags
  left_arm = ArmTag("left"), right_arm = ArmTag("right")
  # Step 1: Use left arm to grab the box from top
  self.move(self.grasp_actor(actor=self.box,arm_tag=left_arm ...))
  self.save_observation(step_name="left_arm_grasps_box")
  # Step 2: Move box to handover point using left arm
  handover_pose = self.block_middle_pose
  self.move(self.place_actor(actor=self.box,arm_tag=left_arm,
  target_pose=handover_pose, functional_point_id=1 ...))
 self.save_observation(step_name="move_to_handover_point")
  #Step 3: Use right arm to grasp the box
  self.move(self.grasp_actor(actor=self.box, arm_tag=right_arm ...))
  self.move(self.close_gripper(arm_tag=right_arm))
  self.move(self.open_gripper(arm_tag=left_arm))
  self.save_observation(step_name="right_arm_grasps_box")
  # Step 4: Move box to target box using right arm
  target_pose = self.target_box.get_functional_point(1, "pose")
  self.move(self.place_actor(actor=self.box, arm_tag=right_arm,
  target_pose=target_pose, functional_point_id=0,
  constrain="free"))
  self.save_observation(step_name="box_placed_on_target")
```

Figure 2. **HyCodePolicy: Expert Code Generation Pipeline.** The pipeline integrates language-conditioned program synthesis with multimodal monitoring and iterative repair, enabling adaptive and self-correcting robotic behaviors. It combines high-level task grounding, simulated execution with feedback-driven diagnostics, and a closed-loop repair cycle to refine robot policies over time.

p_{util}: Interaction site for functional use.

Axis Operation Primitives ($\mathcal{A}$): capture directional alignment:

a_{grasp}: Approach axis for grasping.
a_{place}: Orientation axis for placing.
a_{util}: Motion axis for functional tasks.

These abstractions guide the synthesis of subgoal-conditioned code that respects geometric feasibility. For example, relative pose constraints ensure robust grasps and precise placements. By embedding geometric priors into code synthesis, the system reduces execution-time failures and grounds language instructions in the spatial structure of the environment.

3.2. Simulate Execution & Multimodal Monitoring

3.2.1. Program Execution and Symbolic Logging

Once an initial program is synthesized, it is executed within a simulated robotic environment to validate its operational correctness. Each program undergoes ten independent trials to account for stochasticity in robot control, physics simulation, and sensory input. After each batch of executions, the system produces structured symbolic logs that record the outcome (success/-failure) of each run, along with diagnostic error messages categorizing failure types—such as unreachable grasp configurations, invalid function calls, or incorrect placements.

These symbolic logs serve as a low-level feedback channel, capturing the syntactic and functional integrity of the program. However, they are inherently limited in attributing failures that arise from subtle visual or semantic inconsistencies.

3.2.2. Concurrent Multimodal Observation

To supplement symbolic logs with richer perceptual insight, we introduce a vision-language model (VLM) agent that monitors execution in parallel. This component plays a dual role that it observes and records critical state transitions, and analyzes these transitions to assess sub-goal completion.

Observation points are strategically inserted by analyzing the program structure for each subgoal $s_i \in \mathcal{S}$. For each subgoal, we identify a set of operations $\mathcal{O}^i = \{o_1^i, \ldots, o_{K_i}^i\}$, and apply a filtering function ϕ defined as:

$$\phi(o_k^i) = \begin{cases} 1 & \text{if } o_k^i \text{ causes a visible state change} \\ 0 & \text{otherwise} \end{cases}$$

Whenever $\phi(o_k^i) = 1$, an observation function `save_camera_images()` is invoked to capture the visual context post-operation. Observations are also collected at the beginning ($t = 0$) and end ($t = T$) of execution. The complete set of visual observations is:

$$\mathcal{V} = \{v_0\} \cup \{v_k^i \mid \phi(o_k^i) = 1\} \cup \{v_T\}$$

Each v includes RGB-D images, timestamps, step identifiers, and the associated program context, enabling fine-grained alignment between visual evidence and symbolic execution steps.

3.3. Hybrid Feedback and Failure Attribution

3.3.1. VLM-based Perceptual Verification

Following program execution, the VLM agent analyzes the sequence of collected observations to determine whether each subgoal was successfully completed. For each s_i, the model evaluates the corresponding visual frames $v^i_{1:K_i}$ and returns a binary success signal:

$$\hat{y}_i = \text{Observation}(v^i_{1:K_i}) \in \{0, 1\}$$

If $\hat{y}_i = 1$, the subgoal is considered complete. Otherwise, the model initiates a failure analysis routine.

In the case of a failure, the VLM identifies the precise point of deviation t_i^* within the subgoal's execution and infers a high-level causal hypothesis $c_i \in \{\text{logic error}, \text{API misuse}, \text{execution failure}, \ldots\}$. This diagnosis provides a semantically meaningful interpretation of the error, rooted in perceptual context.

3.3.2. Fusing Symbolic and Perceptual Feedback for Diagnosis

By fusing the VLM's perceptual diagnostics with the symbolic logs obtained during simulation, the system produces a joint interpretation of the failure. Symbolic traces provide procedural integrity checks, while visual diagnostics localize failures in space and time and characterize their nature (e.g., incorrect grasp angle, missing object alignment).

This hybrid diagnosis is critical for transitioning from mere detection to causal understanding. The fused feedback is encoded as a structured signal that conditions the next stage of program revision. It enables the system to isolate problematic operations and prioritize repairs according to semantic relevance and execution risk.

3.4. Closed-Loop Autonomy via Adaptive Monitoring and Iterative Code Evolution

3.4.1. Adaptive Monitoring via Selective Observation and Log-Guided Re-inspection

A defining feature of **HyCodePolicy** is its ability to make adaptive decisions on when and where to deploy multimodal perception. This adaptivity unfolds along two axes: *program-level observation insertion* and *execution-level trial selection*.

First, during initial synthesis, HyCodePolicy selectively inserts observation hooks based on the structure and semantics of the generated code. Specifically, operations that are likely to induce visually observable changes—such as object displacement, alignment-sensitive placements, or grasp transitions—are tagged for post-execution image capture. This avoids unnecessary monitoring overhead while ensuring coverage of visually informative transitions.

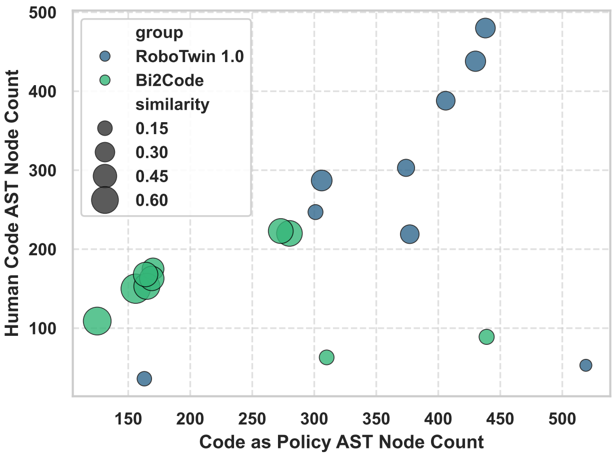


Figure 3. **Distribution of AST similarity and node counts comparing robotic manipulation code generated by RoboTwin 1.0 and Bi2Code with human-written code.** Dot size indicates structural similarity; color denotes source group.

Second, across ten stochastic executions of the same candidate program, HyCodePolicy aggregates symbolic logs to identify the trial with the most diagnostically salient failure. Rather than uniformly analyzing all executions, the system selects a single representative trial for multimodal diagnosis:

$$i^* = \arg\max_i \, \psi(\text{FailureSeverity}_i, \text{TraceDivergence}_i)$$

where ψ is a scoring function that prioritizes executions exhibiting severe failure modes and divergent symbolic traces. Visual inspection is then triggered exclusively for this most informative instance, ensuring efficient use of perceptual resources.

Together, these mechanisms constitute an adaptive attention mechanism over both code structure and execution history, allocating diagnostic effort to the spatiotemporal loci of maximum uncertainty.

3.4.2. Closed-Loop Repair and Policy Evolution

Upon fusing symbolic and visual diagnostics, HyCodePolicy initiates a targeted repair cycle. Faulty operations are localized, and the code-generation agent proposes structured edits based on the failure mode, ranging from logic rewrites and API substitutions to geometric parameter retuning. Repairs are constrained by a symbolic grammar and subgoal template, preserving compatibility and ensuring downstream executability.

Cond.	**ASR**	**Top5-ASR**	**CR-Iter**
Code as Policies w. RoboTwin 1.0	47.4%	57.6%	1.00
CodeAct w. RoboTwin 1.0	60.4%	71.4%	2.46
HyCodePolicy w. RoboTwin 1.0	63.9%	**74.2%**	2.42
Code as Policies w. Bi2Code	62.1%	68.0%	1.00
CodeAct w. Bi2Code	66.7%	73.6%	1.89
HyCodePolicy w. Bi2Code	**71.3%**	**78.6%**	**1.76**

Table 1. **Overall Performance Comparison of Language-Conditioned Robotic Policy Generation Frameworks.** Average Success Rate (ASR), Top-5 ASR, and Mean Code Revision Iterations (CR-Iter) are reported for different policy generation and repair frameworks (Code as Policies, CodeAct, HyCodePolicy) across RoboTwin 1.0 and Bi2Code interfaces.

Metric	**RoboTwin 1.0**	**Bi2Code**
Prompt Token Length ↓	5901.0	**4719.1**
Code Token Length ↓	1236.6	**569.4**
Parallelism Control ↑	✗	✓
AST Similarity [33] ↑	23.72%	**44.78%**
CodeBLEU Similarity [29] ↑	17.18%	**18.53%**
CodeBERT Similarity [8] ↑	97.72%	**98.80%**
Unixcoder Similarity [9] ↑	76.24%	**82.21%**

Table 2. **Code Generation Efficiency and Quality Comparison.** Evaluation of prompt and generated code characteristics, along with code similarity metrics (AST Structural Similarity, CodeBERT, and Unixcoder cosine similarity) against expert-written human code, for RoboTwin 1.0 and Bi2Code in zero-shot generation.

This diagnosis-driven correction process is iterated in a closed loop. Each revised program is re-executed, re-monitored, and re-diagnosed, forming a feedback-driven refinement pipeline. Over multiple iterations, policies evolve into stable, interpretable, and perceptually grounded solutions. Crucially, this evolution is not static fine-tuning but an active restructuring of the program in response to empirical failures.

The result is a form of *policy evolvability*: the ability of generated programs to improve autonomously through multimodal self-assessment and revision. Unlike static one-shot approaches, HyCodePolicy yields adaptive controllers that grow more robust through iterative experience, embodying a scalable strategy for long-horizon task acquisition under uncertainty.

4. Experiment

4.1. Experimental Setup

We evaluate our framework on a shared suite of 10 robotic manipulation tasks supported by both RoboTwin 1.0 [26, 27] and our redesigned Bi2Code interface, which is built upon RoboTwin2.0[3]. Each task is defined via a natural language instruction and executed in a physics-based simulation environment. For each configuration, the code-generation agent synthesizes 10 candidate programs per task, each executed 10 times. Results are averaged to mitigate stochasticity in perception and physics.

Our approach integrates *DeepSeek-V3* for program synthesis and *moonshot-v1-32k-vision-preview* for multimodal observation and diagnosis. We consider three hierarchical configurations. The detailed prompt structures, code templates, and environment metadata used in each configuration are provided in Appendix A.2.

- **Code as Policies:** One-shot generation with no feedback. This baseline reflects a static mapping from instruction to program.
- **CodeAct:** Symbolic feedback and trace-driven repair.
- **HyCodePolicy:** Our full closed-loop pipeline that integrates both symbolic and vision-language feedback for perceptually grounded repair.

To enable effective deployment of HyCodePolicy, we reengineered RoboTwin1.0 into **Bi2Code**—a modular task execution interface with four key capabilities: (1) dual-arm API support, (2) decomposable and structured prompts, (3) standardized symbolic logging, and (4) embedded observation hooks for multimodal tracing. Notably, Bi2Code extends task coverage from 14 to 50, but we restrict evaluation to 10 overlapping tasks to ensure a fair comparison. The complete set of environment functions available in Bi2Code, along with a representative usage example, is provided in AppendixA.4 and A.5, respectively.

We report three metrics to capture both one-shot accuracy and iterative repair efficiency:

- **ASR** (Average Success Rate): Average task completion rate across all candidate executions.
- **Top5-ASR**: Success rate among the top-5 performing candidates.
- **CR-Iter**: Mean number of repair iterations to exceed 50% success.

They jointly evaluate program quality, repair effectiveness, and convergence efficiency across varying sys-

Task	RoboTwin 1.0			Bi2Code		
	Code as Policies	CodeAct	HyCodePolicy	Code as Policies	CodeAct	HyCodePolicy
Beat Block Hammer	16%	48%	**56%**	23%	34%	53%
Handover Block	2%	41%	45%	17%	**50%**	27%
Pick Diverse Bottles	**65%**	**65%**	64%	60%	60%	62%
Pick Dual Bottles Easy	99%	99%	**100%**	**100%**	**100%**	**100%**
Place Container Plate	66%	79%	**91%**	84%	84%	82%
Place Dual Shoes	19%	22%	**25%**	0%	2%	22%
Place Empty Cup	90%	90%	**100%**	61%	61%	85%
Place Shoe	72%	90%	90%	**100%**	**100%**	**100%**
Stack Blocks Three	1%	2%	4%	76%	76%	**82%**
Stack Blocks Two	44%	68%	64%	**100%**	**100%**	**100%**

Table 3. **Task-Specific Performance Comparison of Different Feedback Mechanisms.** Average success rates for individual tasks are presented across 'Code as Policies', 'CodeAct', and 'HyCodePolicy' variants, utilizing both RoboTwin 1.0 and Bi2Code interfaces. Bold numbers indicate the best result for each task.

tem architectures.

4.2. Q1: How Efficient is Bi2Code Compared to Baseline RoboTwin 1.0?

We first quantify the architectural impact of Bi2Code in a one-shot setting (*i.e.*, Code as Policies). Table 2 shows that Bi2Code yields significantly shorter programs (569.4 vs. 1236.6 tokens), with reduced prompt length and higher structural similarity to human-written code. Crucially, it enables dual-arm parallelism via a unified API abstraction, which is absent in RoboTwin 1.0.

These improvements stem from the structured prompting and geometric API modularization designed into Bi2Code. Higher AST similarity (+21.06%), CodeBERT similarity (+1.08%), and Unixcoder alignment (+5.97%) indicate that Bi2Code not only reduces code size but also improves semantic clarity and functional alignment. These properties are essential for subsequent feedback-based refinement, as modular and interpretable code facilitates localized repair. A detailed case study comparing HyCodePolicy-generated and human-written code under the Bi2Code interface is provided in Appendix A.1.

4.3. Q2: Do Feedback and Multimodal Repair Improve Performance? A Hierarchical Ablation Perspective

To systematically assess the impact of feedback modalities, we adopt a hierarchical variant structure that lends itself naturally to ablation-style analysis. Each system variant—*Code as Policies*, *CodeAct*, and *HyCodePolicy*—incrementally augments its predecessor with richer feedback capabilities, ranging from no feedback (one-shot execution) to symbolic correction and finally vision-language-grounded diagnosis. This layered design isolates the contribution of each feedback mechanism to both repair effectiveness and convergence speed. Detailed prompts and configurations for multimodal observation and iterative feedback are provided in Appendix A.3.

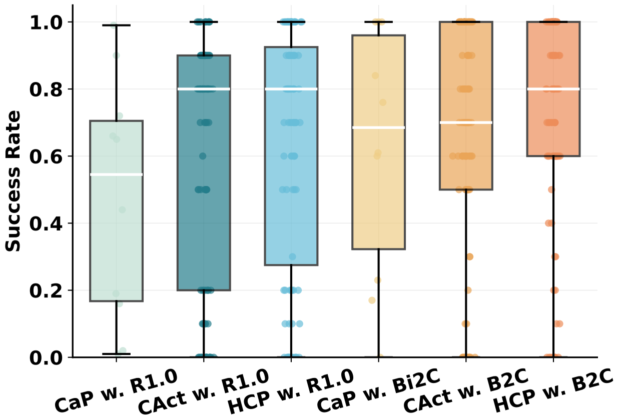


Figure 4. **Distribution of Task Success Rates.** The figure shows the distribution of success rates across all tasks for RoboTwin 1.0 and Bi2Code under different feedback configurations. 'HyCodePolicy' in Bi2Code results in compact distributions centered above 80%, with stronger worst-case performance.

As shown in Figure 4, the task success rate distribution highlights the impact of different feedback mechanisms. In the Code as Policies setting for RoboTwin 1.0, there is high variance and a low median, indicating inconsistent performance. The introduction of CodeAct reduces this variance, improving overall consistency and shifting the central tendency upwards. The most notable improvement is observed with HyCode-

Policy in Bi2Code, where success rates become more concentrated above 80%.

Table 1 presents our main results. Moving from Code as Policies to CodeAct improves RoboTwin 1.0's ASR from 47.4% to 60.4%, while HyCodePolicy further improves it to 63.9%. On Bi2Code, the effect is even more pronounced, with ASR rising from 62.1% to 66.7% with CodeAct, and to 71.3% with HyCodePolicy.

Importantly, HyCodePolicy reduces CR-Iter, indicating faster convergence. Bi2Code reaches functional success in 1.76 iterations, compared to 2.42 under RoboTwin 1.0. These gains highlight that HyCodePolicy not only provides richer feedback but also that Bi2Code's structured code and logging better expose the failure loci for targeted correction.

From a system design perspective, this demonstrates a tight coupling between modular task APIs (Bi2Code), perceptual traceability (VLM hooks), and causally grounded repair. The observed improvements are not just empirical, but emerge directly from design choices that expose richer internal structure to the feedback loop.

4.4. Q3: Which Part of the Planning Does Multimodal Feedback Matter Most?

To dissect performance at a finer granularity, Table 3 compares per-task success rates under CodeAct and HyCodePolicy. Tasks like *stack blocks three*, *place empty cup*, and *handover block* show large gains under HyCodePolicy. These tasks require accurate spatial reasoning, precise object alignment, and nuanced perception—factors poorly captured by symbolic logs.

By contrast, tasks such as *pick dual bottles easy*, which rely on deterministic logic, show near-identical performance across all variants. This confirms that symbolic feedback suffices in low-ambiguity domains, while HyCodePolicy's multimodal loop is crucial for disambiguating failure in visually complex settings.

These trends reinforce that HyCodePolicy's strength lies in its perceptual introspection capability—leveraging visual observations not just for detection, but for actionable error attribution. This "why it failed" signal is critical to enabling effective revision in cases where symbolic traces are silent.

Performance differences between RoboTwin 1.0 and Bi2Code are not directly comparable in Table 3 , as they rely on distinct motion planning backends. While RoboTwin 1.0 uses a deterministic planner, Bi2Code adopts `Curobo`, whose planning process exhibits inherent stochasticity. This variability affects not only generated policies but also expert-authored programs, introducing confounding factors in direct cross-platform comparison.

4.5. Q4: How Well Does HyCodePolicy Generalize Across Diverse Tasks?

To evaluate the generalization capability of our proposed HyCodePolicy framework, we extend evaluation from the 10 core tasks (shared by RoboTwin1.0 and Bi2Code) to the full 50-task suite supported by Bi2Code. Crucially, the framework architecture, feedback logic, and prompting structure were jointly tuned only on the shared subset reported in Tab. 3. No additional task-specific adaptation, hyperparameter change, or manual prompt adjustment was introduced when scaling to the remaining 40 tasks in Tab. 4.

HyCodePolicy demonstrates strong zero-shot generalization, performing well in tasks involving structured placement, stacking, and planar manipulation (e.g., *place mouse pad*, *stack blocks two*, *adjust bottle*), confirming that its framework abstractions, such as compositional prompts and perceptual repair, remain effective across diverse tasks.

However, notable failures occur in tasks requiring non-rigid object handling (*place bread basket*), articulated motion (*open microwave*), or temporal sequencing (*press stapler*, *scan object*). These shortcomings stem from limitations in the action API, world modeling, and trajectory-level reasoning, highlighting areas for future extension. Specifically, tasks like *press*, *scan*, *shake*, *pull*, and *pour* have 0% success due to the absence of these skills in the current API. While HyCodePolicy attempts to simulate these actions by adjusting parameters or combining actions, it often results in suboptimal performance. This occurs because human-designed policies typically involve specific poses, a capability HyCodePolicy currently lacks.

The skill success rate chart (Fig. 5) shows that tasks requiring complex or uncommon skills, such as *press* (0%) and *scan* (0%), perform poorly compared to more common skills like *stack* (70.7%) and *pick* (58.8%). This underscores HyCodePolicy's strength in basic manipulation tasks but reveals challenges in advanced, non-rigid manipulation and precise object control.

Overall, HyCodePolicy exhibits robust zero-shot generalization but its limitations suggest key areas for future improvement, particularly in expanding the action API to support more specialized skills for complex tasks.

5. Conclusions and Limitations

This paper introduced **HyCodePolicy**, a novel closed-loop architecture for language-grounded robotic manipulation. By integrating hierarchical subgoal de-

Task	Rate	Task	Rate	Task
Adjust Bottle	100%	Beat Block Hammer	53%	Blocks Ranking Rgb
Blocks Ranking Size	80%	Click Alarmclock	0%	Click Bell
Dump Bin Bigbin	0%	Grab Roller	74%	Handover Block
Handover Mic	0%	Hanging Mug	0%	Lift Pot
Move Can Pot	30%	Move Pillbottle Pad	50%	Move Playingcard Away
Move Stapler Pad	100%	Open Laptop	0%	Open Microwave
Pick Diverse Bottles	62%	Pick Dual Bottles	100%	Place A2B Left
Place A2B Right	60%	Place Bread Basket	0%	Place Bread Skillet
Place Can Basket	0%	Place Cans Plasticbox	100%	Place Container Plate
Place Dual Shoes	22%	Place Empty Cup	85%	Place Fan
Place Burger Fries	100%	Place Mouse Pad	100%	Place Object Basket
Place Object Scale	80%	Place Object Stand	90%	Place Phone Stand
Place Shoe	100%	Press Stapler	0%	Put Bottles Dustbin
Put Object Cabinet	0%	Rotate Qrcode	80%	Scan Object
Shake Bottle	0%	Shake Bottle Horizontally	0%	Stack Blocks Three
Stack Blocks Two	100%	Stack Bowls Three	20%	Stack Bowls Two
Stamp Seal	20%	Turn Switch	0%	**Avg Success Rate**

Table 4. **Per-Task Success Rates of HyCodePolicy on the Full Bi2Code Task Suite.** This table summarizes the average success rates for all 50 tasks supported by the Bi2Code interface when executed using our proposed HyCodePolicy framework.

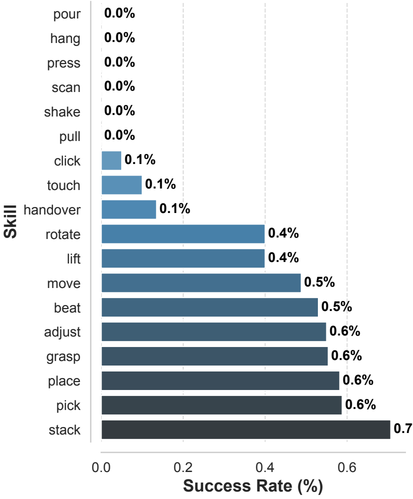


Figure 5. **Skill success rates**, where each skill's success rate is calculated as the average success rate of tasks that utilize the skill.

composition, geometrically grounded program synthesis, multimodal monitoring, and iterative program repair, HyCodePolicy transforms code into a dynamic tool for perception, self-monitoring, and autonomous refinement. Our experimental evaluation using the **Bi2Code** interface demonstrated significant improvements in code conciseness, success rates, and convergence speed. The multimodal feedback, combining symbolic logs and VLM-based diagnostics, enhanced performance in tasks requiring spatial reasoning and perceptual disambiguation, marking a significant step toward more robust, interpretable, and autonomous robotic systems with reduced human supervision.

Despite these advancements, **HyCodePolicy** faces challenges in **articulated object manipulation**, **fine-grained parameter adjustments**, and **deformable object dynamics**. Tasks requiring precise arm poses and granular spatial understanding remain difficult, as do those involving complex temporal reasoning. Future work will focus on improving these areas, with an emphasis on enhancing temporal reasoning, integrating external knowledge, and developing lifelong learning mechanisms.

References

[1] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as i can and not as i say: Grounding language in robotic affordances. In *arXiv preprint arXiv:2204.01691*, 2022. 2, 3

[2] Junting Chen, Yao Mu, Qiaojun Yu, Tianming Wei, Silang Wu, Zhecheng Yuan, Zhixuan Liang, Chao Yang, Kaipeng Zhang, Wenqi Shao, et al. Roboscript: Code generation for free-form manipulation tasks across real and simulation. *arXiv preprint arXiv:2402.14623*, 2024. 3

[3] Tianxing Chen, Zanxin Chen, Baijun Chen, Zijian Cai, Yibin Liu, Qiwei Liang, Zixuan Li, Xianliang Lin, Yiheng Ge, Zhenyu Gu, et al. Robotwin 2.0: A scalable data generator and benchmark with strong domain randomization for robust bimanual robotic manipulation. *arXiv preprint arXiv:2506.18088*, 2025. 6

[4] Tianxing Chen, Kaixuan Wang, Zhaohui Yang, Yuhao Zhang, Zanxin Chen, Baijun Chen, Wanxi Dong, Ziyuan Liu, Dong Chen, Tianshuo Yang, et al. Benchmarking generalizable bimanual manipulation: Robotwin dual-arm collaboration challenge at cvpr 2025 meis workshop. *arXiv preprint arXiv:2506.23351*, 2025. 2

[5] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023. 3

[6] Yongchao Chen, Yilun Hao, Yang Zhang, and Chuchu Fan. Code-as-symbolic-planner: Foundation model-

based robot planning via symbolic code generation. *arXiv preprint arXiv:2503.01700*, 2025. 3

[7] Vanya Cohen, Jason Xinyu Liu, Raymond Mooney, Stefanie Tellex, and David Watkins. A survey of robotic language grounding: Tradeoffs between symbols and embeddings. *arXiv preprint arXiv:2405.13245*, 2024. 2

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020. 6

[9] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022. 6

[10] Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. Hiagent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. *arXiv preprint arXiv:2408.09559*, 2024. 2

[11] Mengkang Hu, Tianxing Chen, Yude Zou, Yuheng Lei, Qiguang Chen, Ming Li, Yao Mu, Hongyuan Zhang, Wenqi Shao, and Ping Luo. Text2world: Benchmarking large language models for symbolic world model generation, 2025.

[12] Wenlong Huang, Chen Wang, Yunzhu Li, Ruohan Zhang, and Li Fei-Fei. Rekep: Spatio-temporal reasoning of relational keypoint constraints for robotic manipulation. *arXiv preprint arXiv:2409.01652*, 2024. 2

[13] Dongwei Jiang, Alvin Zhang, Andrew Wang, Nicholas Andrews, and Daniel Khashabi. Feedback friction: Llms struggle to fully incorporate external feedback. *arXiv preprint arXiv:2506.11930*, 2025. 2, 3

[14] Yunfan Jiang, Agrim Gupta, Zichen Zhang, Guanzhi Wang, Yongqiang Dou, Yanjun Chen, Li Fei-Fei, Anima Anandkumar, Yuke Zhu, and Linxi Fan. Vima: General robot manipulation with multimodal prompts. *arXiv preprint arXiv:2210.03094*, 2(3):6, 2022. 2, 3

[15] Azal Ahmad Khan, Michael Andrev, Muhammad Ali Murtaza, Sergio Aguilera, Rui Zhang, Jie Ding, Seth Hutchinson, and Ali Anwar. Safety aware task planning via large language models in robotics. *arXiv preprint arXiv:2503.15707*, 2025. 3

[16] Manling Li, Shiyu Zhao, Qineng Wang, Kangrui Wang, Yu Zhou, Sanjana Srivastava, Cem Gokmen, Tony Lee, Li Erran Li, Ruohan Zhang, et al. Embodied agent interface: Benchmarking llms for embodied decision making. In *NeurIPS 2024*, 2024. 2

[17] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *arXiv preprint arXiv:2209.07753*, 2022. 2, 3

[18] Zhixuan Liang, Xingyu Zeng, Rui Zhao, and Ping Luo. Mean-ap guided reinforced active learning for object detection. *arXiv preprint arXiv:2310.08387*, 2023. 3

[19] Zhixuan Liang, Yao Mu, Hengbo Ma, Masayoshi Tomizuka, Mingyu Ding, and Ping Luo. Skilldiffuser: Interpretable hierarchical planning via skill abstractions in diffusion-based task execution. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16467–16476, 2024. 2

[20] Zhixuan Liang, Yao Mu, Yixiao Wang, Tianxing Chen, Wenqi Shao, Wei Zhan, Masayoshi Tomizuka, Ping Luo, and Mingyu Ding. Dexdiffuser: Interaction-aware diffusion planning for adaptive dexterous manipulation. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 1745–1755, 2025. 2

[21] Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. Llm+ p: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*, 2023. 2

[22] Jason Xinyu Liu, Ziyi Yang, Benjamin Schornstein, Sam Liang, Ifrah Idrees, Stefanie Tellex, and Ankit Shah. Lang2ltl: Translating natural language commands to temporal specification with large language models. In *Workshop on Language and Robotics at CoRL 2022*, 2022. 3

[23] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36: 46534–46594, 2023. 3

[24] Yao Mu, Qinglong Zhang, Mengkang Hu, Wenhai Wang, Mingyu Ding, Jun Jin, Bin Wang, Jifeng Dai, Yu Qiao, and Ping Luo. Embodiedgpt: Vision-language pre-training via embodied chain of thought. *Advances in Neural Information Processing Systems*, 36:25081–25094, 2023. 2, 3

[25] Yao Mu, Junting Chen, Qinglong Zhang, Shoufa Chen, Qiaojun Yu, Chongjian Ge, Runjian Chen, Zhixuan Liang, Mengkang Hu, Chaofan Tao, et al. Robocodex: Multimodal code generation for robotic behavior synthesis. *arXiv preprint arXiv:2402.16117*, 2024. 2, 3

[26] Yao Mu, Tianxing Chen, Shijia Peng, Zanxin Chen, Zeyu Gao, Yude Zou, Lunkai Lin, Zhiqiang Xie, and Ping Luo. Robotwin: Dual-arm robot benchmark with generative digital twins (early version). *arXiv preprint arXiv:2409.02920*, 2024. 2, 6

[27] Yao Mu, Tianxing Chen, Zanxin Chen, Shijia Peng, Zhiqian Lan, Zeyu Gao, Zhixuan Liang, Qiaojun Yu, Yude Zou, Mingkun Xu, Lunkai Lin, Zhiqiang Xie, Mingyu Ding, and Ping Luo. Robotwin: Dual-arm robot benchmark with generative digital twins. In *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR)*, pages 27649–27660, 2025. 6

[28] Fei Ni, Jianye Hao, Yao Mu, Yifu Yuan, Yan Zheng, Bin Wang, and Zhixuan Liang. Metadiffuser: Diffusion model as conditional planner for offline meta-rl. In *International Conference on Machine Learning*, pages 26087–26105. PMLR, 2023. 2

[29] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020. 6

[30] Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11523–11530. IEEE, 2023. 2, 3

[31] Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. Intervenor: Prompt the coding ability of large language models with the interactive chain of repair. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024. 3

[32] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*, 2024. 3

[33] Wu Wen, Xiaobo Xue, Ya Li, Peng Gu, and Jianfeng Xu. Code similarity detection using ast and textual information. *International Journal of Performability Engineering*, 15(10):2683, 2019. 6

[34] Ziyi Yang, Shreyas S Raman, Ankit Shah, and Stefanie Tellex. Plug in the safety chip: Enforcing constraints for llm-driven robot agents. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 14435–14442. IEEE, 2024. 3

A. Appendix

This appendix provides the detailed prompt structure and components used for guiding the LLM in generating robotic control code. The prompt is constructed from several distinct modules, each serving a specific purpose in defining the task, available tools, and contextual information. For detailed usage instructions, refer to https://robotwin-platform.github.io/doc/usage/expert-code-gen.html.

A.1. LLM-Generated Code and Human-Written Code Case Study

LLM-Generated Code (gpt_place_shoe)

```python
class gpt_place_shoe(place_shoe):
    def play_once(self):
        # Initial observation
        self.save_camera_images(task_name="place_shoe", step_name="step1_initial_scene_state", generate_num_id="generate_num_0")

        # Get the shoe's position to determine which arm to use
        shoe_pose = self.shoe.get_pose()
        shoe_position = shoe_pose.p
        arm_tag = ArmTag("left" if shoe_position[0] < 0 else "right")

        # Grasp the shoe
        self.move(self.grasp_actor(actor=self.shoe, arm_tag=arm_tag, pre_grasp_dis=0.1, grasp_dis=0))
        self.save_camera_images(task_name="place_shoe", step_name="step2_shoe_grasped", generate_num_id="generate_num_0")

        # Lift the shoe up to avoid collision
        self.move(self.move_by_displacement(arm_tag=arm_tag, z=0.07, move_axis='world'))
        self.save_camera_images(task_name="place_shoe", step_name="step3_shoe_lifted", generate_num_id="generate_num_0")

        # Get target pose from the top functional point of the target block
        target_pose = self.target_block.get_functional_point(1, "pose")

        # Place the shoe on the target block with head towards left side
        self.move(self.place_actor(actor=self.shoe, arm_tag=arm_tag, target_pose=target_pose, functional_point_id=0, pre_dis=0.1, dis=0.02, is_open=True, constrain="align", pre_dis_axis='fp'))
        self.save_camera_images(task_name="place_shoe", step_name="step4_shoe_placed", generate_num_id="generate_num_0")

        # Lift the gripper up after placing
        self.move(self.move_by_displacement(arm_tag=arm_tag, z=0.07, move_axis='world'))

        # Return arm to origin
        self.move(self.back_to_origin(arm_tag=arm_tag))

        # Final observation
        self.move(self.save_camera_images(task_name="place_shoe", step_name="step5_final_scene_state", generate_num_id="generate_num_0"))
```

Listing 1. LLM-Generated Code for gpt_place_shoe Task

Human-Written Code (place_shoe)

```python
class place_shoe(base_task):
    def play_once(self):
        # Get the shoe's position to determine which arm to use
        shoe_pose = self.shoe.get_pose().p
        arm_tag = ArmTag("left" if shoe_pose[0] < 0 else "right")

        # Grasp the shoe with specified pre-grasp distance and gripper position
        self.move(self.grasp_actor(self.shoe, arm_tag=arm_tag, pre_grasp_dis=0.1, gripper_pos=0))

        # Lift the shoe up by 0.07 meters in z-direction
        self.move(self.move_by_displacement(arm_tag=arm_tag, z=0.07))

        # Get target_block's functional point as target pose
        target_pose = self.target_block.get_functional_point(0)

        # Place the shoe on the target_block with alignment constraint and specified pre-placement distance
        self.move(self.place_actor(self.shoe, arm_tag=arm_tag, target_pose=target_pose, functional_point_id=0, pre_dis=0.12, constrain="align"))

        # Open the gripper to release the shoe
        self.move(self.open_gripper(arm_tag=arm_tag))
```

Listing 2. Human-Written Code for place_shoe Task

The AI-generated code tends to be more verbose, explicitly logging intermediate visual states and detailing parameters (e.g., pre_dis_axis='fp', is_open=True), while human-written scripts are more minimal, omitting intermediate steps and favoring compact execution. Despite functional similarity, the structural differences illustrate that **MLLM-generated programs are not only executable but emphasize step-by-step clarity**, contributing to more robust feedback and repair.

A.2. Prompt Templates, Code Template and Basic Info

A.2.1. Overall Prompt Template

The complete prompt is constructed by concatenating several key components, as shown in the template below. This structure ensures all necessary information—basic environment details, task description, available actors, API functions, and current code—is provided to the language model for generating or repairing code.

Prompt Template

```
Prompt = (
f"#Basic Info:\n{BASIC_INFO}\n"
f"#Task Description:\n{task_description}\n"
f"#Actor List:\n{actor_list}\n"
f"#Available API:\n{available_env_function}\n"
f"#Function Example:\n{function_example}\n"
f"#Current Code:\n{current_code}"
)
```

A.2.2. Basic Info

The BASIC_INFO string provides fundamental details about the simulation environment, including units of measurement, pose representation, coordinate system conventions, and how to access functional points of actors. This foundational information is crucial for the language model to understand the operational context and correctly interpret geometric and interaction-related instructions.

BASIC_INFO Constant

```
BASIC_INFO = ”’
In this environment, distance 1 indicates 1 meter long.
Pose is representated as 7 dimention, [x, y, z, qw, qx,
qy, qz].
For a 7-dimensional Pose object, you can use Pose.p to
get the [x, y, z] coordinates and Pose.q to get the
[qw, qx, qy, qz] quaternion orientation.
All functions which has parameter actor, and all of
actor should be in the Actor object.
In the world coordinate system, the positive directions
of the xyz coordinate axes are right, front, and upper
respectively, so the direction vectors on the right,
front,
and upper sides are [1,0,0], [0,1,0], [0,0,1]
respectively. In the same way, we can get the unit
vectors of the left side, back side and down side.
Each actor in the environment has one or more functional
points, which are specific locations designed for
interactions.
Access functional points using
actor.get_functional_point(point_id, return_type),
where return_type can be ”pose”, ”p”, or ”q”.
”’
```

A.2.3. CODE_TEMPLATE

The CODE_TEMPLATE provides a basic Python class structure that the generated policies must adhere to. This template includes necessary imports and defines a ‘gpt_$TASK_NAME$‘ class inheriting from ‘$TASK_NAME$‘, ensuring the generated code integrates seamlessly into the existing simulation framework. The ‘play_once‘ method is intended to be filled with the generated policy logic.

```
CODE_TEMPLATE = '''
from envs._base_task import Base_Task
from envs.$TASK_NAME$ import $TASK_NAME$
from envs.utils import *
import sapien

class gpt_$TASK_NAME$($TASK_NAME$):
    def play_once(self):
        pass
'''
```

Listing 3. **CODE_TEMPLATE Constant**

A.3. Observation Agent Prompt

This section details the prompts used to guide the AI observation agent. The agent's role is twofold: first, to strategically insert observation (camera image capture) function calls into the robot task code at critical points; and second, to analyze the captured images to provide feedback on task execution. A third component outlines how multimodal observation feedback is incorporated into iterative code generation.

A.3.1. Insert Observation Function Calls

This prompt is designed to instruct the LLM to augment existing robot task code with calls to a camera observation function. The goal is to capture significant visual scene changes, providing a mechanism for step-by-step monitoring of the robot's execution.

Prompt for Inserting Observation Calls

```
You are an expert in robot programming. I have a robot task code that needs observation functions added for monitoring.
Task information: {task_info}
I need you to:
1. Identify ONLY the main logical steps in this task implementation that cause SIGNIFICANT SCENE CHANGES
2. After each such logical step in the code, insert a camera observation function with this format:
    self.save_camera_images(task_name="{task_name}", step_name="stepX_descriptive_name", generate_num_id="generate_num_{generate_num_id}")
3. Provide a numbered list of all the steps you've identified in the task
4. ADD AN OBSERVATION AT THE BEGINNING OF THE TASK to capture the initial scene state
5. ADD AN OBSERVATION AT THE END OF THE TASK to capture the final scene state
Here's the current code: python {task_code}
IMPORTANT CONSTRAINTS:
- ADD FEWER THAN 10 OBSERVATION POINTS in total
- ONLY add observations after operations that cause VISIBLE SCENE CHANGES
- Do NOT add observations for planning, calculations, or any operations that don't visibly change the scene
- Focus on key state changes like: robot arm movements, gripper operations, object manipulations
- Skip observations for intermediate movements, planning steps, or calculations
- The observation function is already defined in the code
- Give each step a descriptive name like "gripper_closed", "move_to_target", etc.
- The step number (X in stepX) should increase sequentially
- DO NOT MODIFY ANY EXISTING ROBOT OPERATION CODE – only insert observation function calls after existing code without changing the original functionality
Format your response as follows:
STEP_LIST: 1. First step description 2. Second step description ...
MODIFIED_CODE: python <the entire modified code with observation functions inserted>
```

A.3.2. Observe Task Execution by Analyzing Step-by-Step Images

This prompt guides the AI to analyze visual feedback (step-by-step images) from the robot's execution. The analysis focuses on identifying successful and failed steps, and providing detailed reasoning for any task failures.

Prompt for Image-Based Execution Analysis

```
Analyze the execution of the following robot task:
Task name: {task_name}
Task description: {task_info.get('description', 'No description provided')}
Task goal: {task_info.get('goal', 'No goal provided')}
You will be shown images from each step of the task execution. Please analyze:
1. Whether each step was executed successfully.
2. If any step failed, identify which one and explain why.
3. Whether the overall task was successfully completed.
4. If the task failed, provide detailed reasoning.
You will see execution images for the following steps: {', '.join(step_names)}
```

A.3.3. Iterative Correction with Multimodal Observation Feedback

When the initially generated code is unsuccessful, this prompt demonstrates how multimodal feedback (last error message and visual observation feedback) is incorporated to guide the iterative refinement of the robot task code.

Prompt for Iterative Correction with Feedback

```
if observation_feedback:
  prompt = (
    f"The code is unsuccessful, \n# Last Error Message: \n{last_error}\n\n"
    f"# Visual Observation Feedback: \n{observation_feedback}\n\n"
    f"# Task Description: \n{task_description}\n\n"
    f"# Actor List: \n{actor_list}\n\n"
    # Additional components like available API, function example, and current code would follow
    # f"#Available API:\n{available_env_function}\n"
    # f"#Function Example:\n{function_example}\n"
    # f"#Current Code:\n{current_code}"
  )
```

A.4. Available Environment Functions

The following `AVAILABLE_ENV_FUNCTION`, details the various robotic control functions exposed to the language model. Each entry provides the function's signature, a brief description of its purpose, and a comprehensive list of its parameters. These functions form the foundational API for generating and executing robot manipulation policies within the simulation environment.

Available API

The following functions are available for the AI model to use for controlling the robotic arms and interacting with the environment:

- **`open_gripper(self, arm_tag: ArmTag, pos=1.) -> tuple[ArmTag, list[Action]]`**
  Opens the gripper of the specified arm. Returns: tuple[ArmTag, list[Action]] containing the gripper-open action. Args: arm_tag: Which arm’s gripper to open; pos: Gripper position (1 = fully open).
- **`close_gripper(self, arm_tag: ArmTag, pos=0.) -> tuple[ArmTag, list[Action]]`**
  Closes the gripper of the specified arm. Returns: tuple[ArmTag, list[Action]] containing the gripper-close action. Args: arm_tag: Which arm’s gripper to close; pos: Gripper position (0 = fully closed).
- **`move(self, actions_by_arm1: tuple[ArmTag, list[Action]], actions_by_arm2: tuple[ArmTag, list[Action]] = None)`**
  Executes action sequences on one or both robotic arms simultaneously. No Return. Args: actions_by_arm1: Action sequence for the first arm, formatted as (arm_tag, [action1, action2, ...]); actions_by_arm2: Optional, action sequence for the second arm.
- **`move_by_displacement(self, arm_tag: ArmTag, z=0., move_axis=’world’) -> tuple[ArmTag, list[Action]]`**
  Moves the end-effector of the specified arm along relative directions and sets its orientation. Returns: tuple[ArmTag, list[Action]] containing the move-by-displacement actions. Args: arm_tag: The arm to control; z: Displacement along the z-axis (in meters); move_axis: ’world’ means displacement is in world coordinates, ’arm’ means displacement is in local coordinates.
- **`grasp_actor(self, actor: Actor, arm_tag: ArmTag, pre_grasp_dis=0.1, grasp_dis=0, gripper_pos=0., contact_point_id=None) -> tuple[ArmTag, list[Action]]`**
  Generates a sequence of actions to pick up the specified Actor. Returns: tuple[ArmTag, list[Action]] containing the grasp actions. Args: actor: The object to grasp; arm_tag: Which arm to use; pre_grasp_dis: Pre-grasp distance (default 0.1 meters), the arm will move to this position first; grasp_dis: Grasping distance (default 0 meters), the arm moves from the pre-grasp position to this position and then closes the gripper; gripper_pos: Gripper closing position (default 0, fully closed); contact_point_id: Optional list of contact point IDs; if not provided, the best grasping point is selected automatically.
- **`place_actor(self, actor: Actor, arm_tag: ArmTag, target_pose: list | np.ndarray, functional_point_id: int = None, pre_dis=0.1, dis=0.02, is_open=True, **kwargs) -> tuple[ArmTag, list[Action]]`**
  Places a currently held object at a specified target pose. Returns: tuple[ArmTag, list[Action]] containing the place actions. Args: actor: The currently held object; arm_tag: The arm holding the object; target_pose: Target position/orientation, It is recommended to use the return value of actor.get_functional_point(..., ’pose’) or pose in actor_list as target_pose; functional_point_id: Optional ID of the functional point; if provided, aligns this point to the target, otherwise aligns the base of the object; pre_dis: Pre-place distance (default 0.1 meters), arm moves to this position first; dis: Final placement distance (default 0.02 meters), arm moves from pre-place to this location, then opens the gripper; is_open: Whether to open the gripper after placing (default True), Set False if you need to keep gripper closed to maintain hold of the object; **kwargs: Other optional parameters: constrain : ’free’, ’align’, ’auto’, default=’auto’ Alignment strategy: ’free’: Only forces the object’s z-axis to align with the target point’s z-axis, other axes are determined by projection. ’align’: Forces all axes of the object to align with all axes of the target point. ’auto’: Automatically selects a suitable placement pose based on grasp direction (vertical or horizontal). pre_dis_axis : ’grasp’, ’fp’ or np.ndarray or list, default=’grasp’. Specifies the pre-placement offset direction.
- **`back_to_origin(self, arm_tag: ArmTag) -> tuple[ArmTag, list[Action]]`**
  Returns the specified arm to its predefined initial position. Returns: tuple[ArmTag, list[Action]] containing the return-to-origin action. Args: arm_tag: The arm to return to origin.

A.5. Function Example

`FUNCTION_EXAMPLE` provides practical examples and guidelines for using the available API functions. It demonstrates how to interact with actors, control grippers, and execute complex manipulation sequences, including single-arm and dual-arm operations. This section is essential for understanding the practical application of the API in generating robot control policies.

Function Examples

You can directly use the actors provided in the `actor_list`:

```
# For example, if actor_list contains ["self.object1", "self.object2"]
# You can directly use:
object1 = self.hammer
object2 = self.block
```

Using `ArmTag` class to represent arms:

```
arm_tag = ArmTag("left")  # Left arm
arm_tag = ArmTag("right") # Right arm
```

Example of selecting an arm based on conditions:

```
arm_tag = ArmTag("left" if actor_position[0] < 0 else "right")
```

Each actor in the environment may have multiple functional points that are useful for different interactions. Functional points provide precise locations for interactions like grasping, placing, or aligning objects.
To get a functional point from an actor:

```
functional_point_pose = actor.get_functional_point(point_id, "pose")  % Returns a complete 7-dimensional Pose object
position = functional_point_pose.p  % [x, y, z] position
orientation = functional_point_pose.q % [qw, qx, qy, qz] quaternion orientation
```

Note: The pose from a functional point is already aligned for the task. Do NOT manually construct or rotate a quaternion.
When stacking one object on top of another (e.g., placing blockA on top of blockB):

```
target_pose = self.last_actor.get_functional_point(point_id, "pose")
self.move(
    self.place_actor(
        actor=self.current_actor,
        target_pose=target_pose,
        arm_tag=arm_tag,
        functional_point_id=0,
        pre_dis=0.1,
        dis=0.02,
        pre_dis_axis="fp"
    )
)
```

For actors already of type Pose, such as actor_pose, you do **not** need to call .get_pose() again:

```
self.move(
    self.place_actor(
        self.box,
        target_pose=self.actor_pose,
        arm_tag=grasp_arm_tag,
        functional_point_id=0,
        pre_dis=0,
        dis=0,
        is_open=False,
        constrain="free",
        pre_dis_axis='fp',
    )
)
```

Note: For target_actor, you must use get_pose() or get_functional_point().
To select an arm and grasp based on actor position:

```
actor_pose = self.actor.get_pose()
actor_position = actor_pose.p

arm_tag = ArmTag("left" if actor_position[0] < 0 else "right")

self.move(
    self.grasp_actor(actor=self.actor, arm_tag=arm_tag)
)
```

Example grasping and lifting:

```
self.move(
    self.grasp_actor(
        actor=self.actor,
        arm_tag=arm_tag,
        pre_grasp_dis=0.1,
        grasp_dis=0
    )
)

self.move(
    self.move_by_displacement(
        arm_tag=arm_tag,
        z=0.07,
        move_axis='world'
    )
)
```

Gripper control:

```
self.move(self.open_gripper(arm_tag=arm_tag, pos=1.0))  # Fully open
self.move(self.open_gripper(arm_tag=arm_tag, pos=0.5))  # Half open
self.move(self.close_gripper(arm_tag=arm_tag, pos=0.0)) # Fully close
self.move(self.close_gripper(arm_tag=arm_tag, pos=0.5)) # Half close
```

Example placing:

```
self.move(
    self.place_actor(
        actor=self.actor,
        arm_tag=arm_tag,
        target_pose=self.target_pose,
        functional_point_id=0,
        pre_dis=0.1,
        dis=0.02,
        is_open=True,
        pre_dis_axis='fp',
    )
)

self.move(
    self.move_by_displacement(
        arm_tag=arm_tag,
        z=0.07,
        move_axis='world'
    )
)
```

Aligning a functional point with a target:

```
self.move(
    self.place_actor(
        actor=self.actor,
        arm_tag=arm_tag,
        target_pose=target_pose,
        functional_point_id=0,
        pre_dis=0.1,
        dis=0.02,
        pre_dis_axis='fp'
    )
)
```

Move both arms simultaneously:

```
left_arm_tag = ArmTag("left")
right_arm_tag = ArmTag("right")
self.move(
    self.grasp_actor(actor=self.left_actor,
        arm_tag=left_arm_tag),
    self.grasp_actor(actor=self.right_actor,
        arm_tag=right_arm_tag)
)

self.move(
    self.move_by_displacement(arm_tag=
        left_arm_tag, z=0.07),
    self.move_by_displacement(arm_tag=
        right_arm_tag, z=0.07)
)
```

Place left object while returning right arm to origin:

```
move_arm_tag = ArmTag("left")
back_arm_tag = ArmTag("right")
self.move(
    self.place_actor(
        actor=self.left_actor,
        arm_tag=move_arm_tag,
        target_pose=target_pose,
        pre_dis_axis="fp"
    ),
    self.back_to_origin(arm_tag=back_arm_tag)
)
```

Return arms to initial positions:

```
self.move(self.back_to_origin(arm_tag=arm_tag
    ))

left_arm_tag = ArmTag("left")
right_arm_tag = ArmTag("right")
self.move(
    self.back_to_origin(arm_tag=left_arm_tag)
        ,
    self.back_to_origin(arm_tag=right_arm_tag
        )
)
```