CAPSULE: RUNTIME CONTROL AND HOT PATCHING FOR CODE-AS-POLICY ROBOTIC AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Code-as-Policy (CaP) robotic agents generate executable programs to orchestrate perception, planning, and control for robotic manipulation. However, relying solely on rollout-level feedback makes it difficult to associate execution failures with the relevant code segments. Recovery is further complicated by physical changes that persist after a failed execution, such as an upright book being knocked flat, which can make rerunning the program unsafe or ineffective. We introduce CAPSULE, a runtime control layer for CaP agents that supports group-level execution and effect-aware program repair without requiring the control layer itself to call existing robot APIs. CAPSULE partitions each program into source-local, effect-bounded execution groups and maintains a persistent namespace. It links execution traces and task feedback with the corresponding source regions to support inspection, local patching, and recovery. When recovery is needed after physical effects have occurred, CAPSULE avoids replaying groups whose physical effects have already occurred and generates forward recovery patches conditioned on updated environment observations. Across eight backbone LLMs on Robosuite and LIBERO, CAPSULE significantly outperforms the CaP-X multi-turn baseline, boosting macro-averaged success rates from 36.7% to 49.7% and 8.2% to 16.8%, respectively. Ablation studies show that local patching and physical-state forward recovery provide complementary gains. Furthermore, we propose CAPSULE-Critique-GRPO (CC-GRPO), a post-training scheme that leverages execution-grounded repair trajectories to enhance standalone program generation. In a two-task study, CC-GRPO outperforms vanilla GRPO, increasing task success rates from 17.5% to 31.3% on Cube Stack and from 16.3% to 28.8% on Spill Wipe.

1 INTRODUCTION

Code-as-Policy (CaP) advances embodied control from selecting fixed action primitives toward generating executable programs that dynamically combine perception, geometric computation, motion planning, and control application programming interfaces (APIs) (Liang et al., 2023). By expressing robot policies as code, CaP provides an interpretable and debuggable interface between the reasoning of large language models (LLMs) and embodied execution. Benchmarks such as CaP-Bench (Fu et al., 2026) have formalized a systematic evaluation of this paradigm across diverse manipulation tasks. However, existing execution feedback is provided only at the coarse granularity of complete code blocks or full rollouts. Because a synthesized program heavily interleaves perception and computation with physical actions, such monolithic feedback often indicates that an execution failed without associating failures with the relevant code segments. More critically, by the time an execution failure is detected, preceding robot actions may have already changed the physical environment irreversibly. Although recent embodied agents increasingly leverage execution feedback for program refinement and continual improvement (Chen et al., 2026; Lu et al., 2026; Xiao et al., 2026; Zhang et al., 2026c;a), associating runtime feedback with actionable program segments while accounting for irreversible physical changes remains a challenge.

To address this challenge, we introduce CAPSULE, an effect-aware runtime control layer that exposes execution feedback at a fine-grained program granularity while accounting for physical effects that have already occurred. As illustrated in Figure 1, CAPSULE partitions a generated CaP program into effect-bounded execution groups and incrementally associates runtime feedback with

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

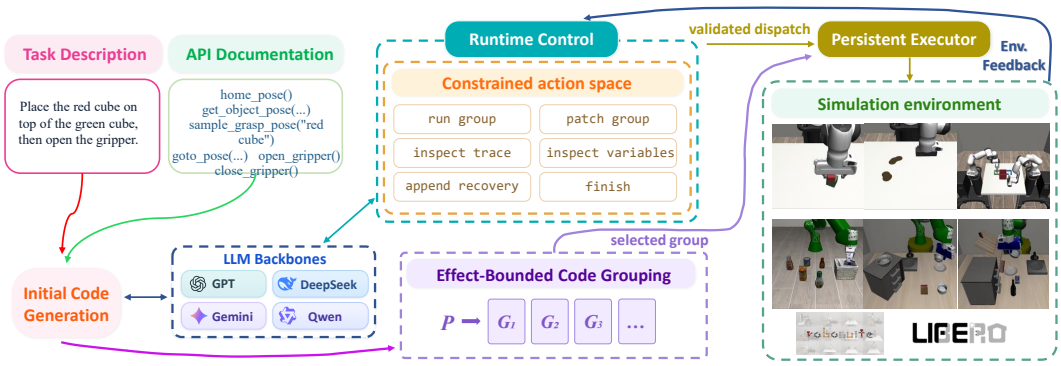

Figure 1: **CAPSULE architecture overview.** CAPSULE decomposes a synthesized CaP program into effect-bounded execution groups and dynamically associates runtime feedback with the corresponding source region. A persistent executor maintains computational state across groups. Before physical actions change the environment, the controller applies local patching to repair faulty code in place. After an effect-inducing group executes, CAPSULE enforces safe execution by switching to forward recovery, generating continuation code conditioned on fresh sensory feedback.

the corresponding source regions. Before the current group produces any physical effects, the runtime controller inspects intermediate results and locally patches the problematic group in place. We refer to this mechanism as *hot patching*, an in-place program repair procedure that modifies code during an ongoing rollout while preserving the active runtime namespace, thereby allowing execution to continue from the current physical state without reinitializing the program or resetting the environment. Conversely, once an effectful primitive invocation has been recorded in the runtime trace, CAPSULE strictly avoids unsafe patching and replay behaviors, switching instead to forward recovery by synthesizing continuation code conditioned on the execution context and grounded in updated environment observations. Across eight backbone LLMs, compared with the CaP-X multi-turn baseline, CAPSULE improves average task success from 36.7% to 49.7% on Robosuite and from 8.2% to 16.8% on LIBERO. Ablation studies further show that both local patching and forward recovery contribute to these improvements.

We also observe that CAPSULE yields limited or inconsistent runtime gains on smaller backbones such as Qwen3-8B. This may reflect difficulties of resource-constrained models in interpreting runtime diagnostics and generating effective corrections, occasionally introducing spurious patches or consuming execution budgets on repeated revisions. These limitations motivate us to investigate whether the rich repair trajectories collected by CAPSULE can guide online reinforcement learning to strengthen the base policy directly. Specifically, we propose Capsule-Critique-GRPO (CC-GRPO), a post-training scheme that integrates runtime execution critiques (Zhang et al., 2025) and structured correction traces into Group Relative Policy Optimization (GRPO) (Shao et al., 2024). In an empirical study on two tasks, CC-GRPO substantially outperforms vanilla GRPO under standalone inference without any runtime intervention, improving task success rates from 17.5% to 31.3% on Cube Stack and from 16.3% to 28.8% on Spill Wipe. Together, these results demonstrate that CAPSULE can improve task success through runtime program repair and that its repair trajectories can also serve as a high-value data generator to train more effective standalone policies.

2 METHODS

CAPSULE is a runtime control layer for CaP robotic agents that executes generated robot programs in effect-bounded groups while preserving native perception, planning, and control interfaces. As illustrated in Figure 1, CAPSULE analyzes the raw abstract syntax tree (AST) of an LLM-generated program to construct these groups, while a persistent executor maintains the computational state across groups and monitors physical environment interactions. A runtime controller uses this execution feedback to steer program flow, diagnose failures, and apply targeted recoveries. Furthermore, localized repair trajectories collected during execution serve as scaffolding to train a stronger standalone program generation policy through CC-GRPO.

2.1 Problem Formulation and Scope

Let x denote a natural language task instruction, $\mathcal{E}$ the robotic environment, and $\mathcal{A}$ the set of public environment APIs exposed to generated programs. Given x, an LLM generates a CaP-style Python program P that interacts with $\mathcal{E}$ through API calls in $\mathcal{A}$.

We partition the public APIs $\mathcal{A}$ into two disjoint subsets: effectful APIs $\mathcal{A}_{\text{eff}} \subseteq \mathcal{A}$ that actuate the robot or alter physical objects (such as trajectory execution, grasping, or gripper release), and observation APIs $\mathcal{A}_{\text{obs}} \subseteq \mathcal{A}$ that query sensory observations, refresh scene states, or retrieve object poses without changing the physical environment. A source region is statically categorized as effectful if its statements or resolvable helper functions invoke primitives in $\mathcal{A}_{\text{eff}}$, with physical effects verified at runtime through execution traces.

At each step t, the persistent executor maintains a computational namespace Γ_t and records execution evidence e_t, including return values, standard output streams, exception stacks, and traced API invocations. We use s_t to denote the state of the physical environment when describing state transitions without assuming privileged access, keeping all observations available to CAPSULE strictly confined to standard benchmark interfaces.

2.2 Effect-Bounded Execution Groups

Statement-level runtime control can incur excessive runtime overhead and fragmented feedback, whereas executing an entire program as a single group makes it difficult to associate failures with the relevant code segments. To balance control overhead and feedback granularity, CAPSULE parses program P into an ordered sequence of top-level source regions $(r_1, r_2, \ldots, r_n)$ based on its AST structure, and partitions them into *effect-bounded execution groups*:

$$G = (g_1, g_2, \ldots, g_m), \quad \text{where} \quad g_j = (r_{a_j}, \ldots, r_{b_j}), \quad j = 1, \ldots, m, \tag{1}$$

subject to the contiguous partition constraints:

$$a_1 = 1, \qquad b_m = n, \qquad 1 \le a_j \le b_j \le n, \qquad a_{j+1} = b_j + 1 \quad (j = 1, \ldots, m-1). \tag{2}$$

Each group g_j serves as a unit of runtime execution and repair, with boundaries determined by program structure and potential physical effects. This construction requires neither predefined subgoal annotations nor manually specified pre- and post-conditions.

Static Region Profiling. For each source region r_i, CAPSULE extracts a static profile:

$$\phi(r_i) = (\text{span}_i, D_i, U_i, C_i, F_i, \hat{\eta}_i)\,, \tag{3}$$

where span_i specifies the source code range, D_i and U_i are the sets of variables defined and used in the region, C_i records statically resolved API calls, and F_i captures the enclosing control-flow context. The indicator $\hat{\eta}_i \in \{0, 1\}$ conservatively marks whether the region may produce physical effects. Specifically, we set $\hat{\eta}_i = 1$ if the region directly invokes an effectful robot API or transitively reaches one through user-defined helpers. Effect annotations are propagated along statically resolvable call paths, while unresolved call targets are conservatively treated as effectful.

Deterministic Boundary Construction. Using these profiles, CAPSULE constructs the execution partition G in a single pass over consecutive source regions and creates group boundaries following the three deterministic rules:

1. Effect boundary. A partition boundary is inserted immediately after each region with $\hat{\eta}_i = 1$. Each active execution group therefore contains at most one marked effectful region before yielding control.

2. Control-flow atomicity. Compound statements, including conditionals and loops, are preserved intact within a single execution group to maintain syntactic validity and control-flow integrity.

3. Dependency-aware closure. Consecutive regions with ($\hat{\eta}_i = 0$) are greedily fused with their downstream consumer regions without crossing effect boundaries.

This grouping strategy constructs non-invasive metadata boundaries over the program AST rather than modifying statements or injecting runtime wrappers. Figure 2 illustrates the above rules using a simplified cube-stacking program. Each execution group g_j corresponds strictly to a contiguous slice of the original source text. This structural design inherently preserves the lossless source reconstruction property, satisfying $\text{concat}(g_1.\texttt{src}, \ldots, g_m.\texttt{src}) = P.\texttt{src}$.

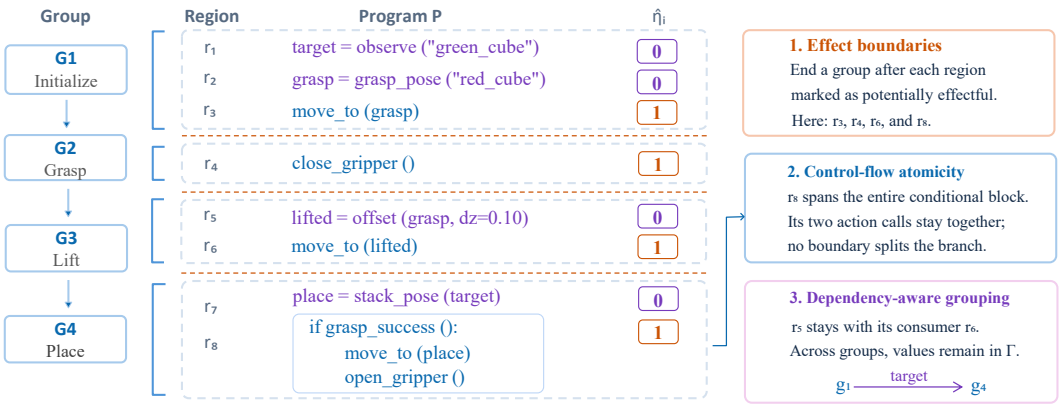

Figure 2: **Effect-bounded grouping of a cube-stacking program.** Eight source regions are partitioned into four execution groups. Dashed lines mark group boundaries after regions with $\hat{\eta}_i = 1$. The complete conditional in r_8 remains within g_4, while variables such as grasp and target remain available across groups through the persistent namespace. API names are simplified for illustration.

2.3 CAPSULE RUNTIME EXECUTION

CAPSULE executes the groups defined in Section 2.2 within a persistent runtime that preserves program state across group boundaries. The generated code retains direct access to the benchmark primitives, such as `get_object_pose`, `goto_pose`, and `open_gripper`. This design preserves the native API surface provided by each benchmark, supporting both non-privileged perception in Robosuite and state-based primitives in LIBERO.

Runtime Control Interface. At each execution step, the runtime exposes six discrete control operations to the high-level controller:

- `run_group`: Executes the next scheduled code group g_j.
- `patch_group`: Modifies an execution group in place, provided that the source group has not yet produced physical effects.
- `inspect_trace`: Retrieves structured execution evidence from recent primitive calls.
- `inspect_variables`: Inspects selected variable states from the active runtime namespace.
- `append_recovery`: Registers new recovery source blocks into the active program.
- `finish`: Ends the execution trial upon task completion or unrecoverable failure.

Persistent Namespace and State Continuity. CAPSULE executes all source groups within a shared, persistent Python namespace whose state transitions sequentially as $\Gamma_0 \to \Gamma_1 \to \cdots \to \Gamma_t$. Intermediate variables computed by earlier groups, such as target object coordinates and grasp poses, remain available to subsequent groups. When errors occur, local modifications or appended recovery blocks update the active program P_t while retaining previously computed valid representations, thereby avoiding repetitive perceptual queries or trajectory re-computations.

Interactive Control and Safe Recovery. At each step, CAPSULE constructs a compact context comprising the active program code, execution progress, and source-grounded feedback. Because physical interactions are irreversible, in-place modification via `patch_group` is restricted to groups that have not executed primitives in $\mathcal{A}_{\text{eff}}$. Once a group produces physical effects, subsequent corrections must be performed by invoking `append_recovery` with new code blocks conditioned on updated sensory feedback. Appending recovery code registers the new logic into P_t for explicit scheduling in subsequent steps.

Primitive Tracing and Source-Bound Feedback. To correlate runtime failures with corresponding source locations, CAPSULE wraps public environment APIs with lightweight tracing decorators that record structured invocation tuples (`name`, `args`, `return`, `error`, `idx`). Before executing a source group, CAPSULE records the current trace position as an execution checkpoint. When execution yields, only events recorded after this checkpoint are attributed to the active group, generating

focused local diagnostic feedback rather than full-trajectory logs. The resulting feedback bundle integrates group identifiers, execution statuses, exception details, local outputs, and rewards.

2.4 POST-TRAINING WITH CAPSULE CRITIQUES

While runtime intervention improves execution success, practical deployment benefits substantially from generating robust programs directly without online controller guidance. Inspired by critique-augmented exploration in reinforcement learning (Zhang et al., 2025), we introduce CC-GRPO, an algorithm that extends GRPO (Shao et al., 2024) by using CAPSULE runtime repair experience to enhance standalone program synthesis.

Critique-Guided Group Construction. Standard GRPO samples candidate programs purely from the unguided policy distribution, which often struggles to produce successful rollouts in sparse-reward environments with complex physical constraints. In contrast, CC-GRPO constructs each exploration group by pairing standard policy rollouts with repaired candidate programs. When an initial rollout fails during training, CAPSULE generates execution-grounded critiques by localizing errors to specific source spans and environment traces, allowing a repair model to propose a revised program. To prevent ungrounded critiques from distorting policy optimization, each repaired program must pass independent verification by executing from the initial state s_0, admitting only successful repaired rollouts into the optimization group.

Policy Optimization and Test-Time Standalone Generation. We optimize the policy network using a dual-source objective over verified candidate groups: standard rollouts are optimized with a clipped surrogate policy-gradient loss, whereas verified repaired rollouts are supervised through an advantage-weighted probability-shaping loss. Both components are evaluated strictly on the original task instruction x without prepending critique tokens or execution histories, regularized by a Kullback-Leibler (KL) divergence penalty relative to reference policy π_{ref} (Appendix C.3.1). This training scheme allows the policy to internalize recovery patterns discovered during interactive execution, enabling the trained model to generate robust and executable CaP programs at test time without runtime intervention.

3 EXPERIMENTS

3.1 EXPERIMENTAL SETUP

Simulation Environments. We conduct online closed-loop evaluations in Robosuite (Zhu et al., 2020) and LIBERO (Liu et al., 2023). On Robosuite, we select six representative manipulation tasks, each evaluated across 20 matched random seeds. The agent interacts with the environment through a library of non-privileged, task-specific control APIs, where object localization, grasp pose estimation, and motion planning rely solely on raw RGB-D observations. For LIBERO, we evaluate three standard suites: `libero_object`, `libero_spatial`, and `libero_goal`. Each suite comprises 10 tasks, and each task is evaluated across 10 official initial scene configurations, yielding 300 evaluation episodes per model (3 suites $\times$ 10 tasks $\times$ 10 initial states). In this benchmark, we evaluate the models using privileged APIs with direct access to simulator ground-truth states, thereby decoupling code generation and runtime recovery from perceptual and pose-estimation errors.

Models and Evaluation Protocol. We evaluate eight backbone LLMs on both benchmarks. Each model is tested with CAPSULE and the CaP-X multi-turn (MT) baseline under matched task instructions, API access, simulator resets, evaluation horizons, and success predicates. We adopt the environment-defined task success rate as the primary evaluation metric. For each model, we compute the unweighted macro-average success rate across tasks within each benchmark, and subsequently average the resulting scores across all eight backbones. For Robosuite, cumulative reward is reported only as a secondary internal metric, since reward formulations and execution horizons are not directly comparable across simulators.

3.2 MAIN RESULTS

For all evaluations, we record the initially generated programs, CAPSULE runtime decisions, robot primitive execution traces, pre- and post-execution state summaries, exception logs, and trial videos.

Table 1: Task success rates (%) on six Robosuite manipulation tasks. Avg. denotes the unweighted mean across all the six tasks. Values in parentheses indicate the absolute percentage-point (pp) gains of CAPSULE over the CaP-X MT baseline.

Model	Setting	Success rate (%)						
		Lift	Stack	Restack	Wipe	2A-L	2A-H	Avg.
DeepSeek-V4-Flash	MT	90.0	55.0	0.0	85.0	15.0	0.0	40.8
	CAPSULE	**95.0**	**85.0**	**20.0**	**90.0**	**55.0**	0.0	**57.5** (+16.7 ↑)
Qwen3.7-Plus	MT	85.0	20.0	0.0	100.0	0.0	0.0	34.2
	CAPSULE	**95.0**	**85.0**	**15.0**	100.0	**10.0**	0.0	**50.8** (+16.7 ↑)
Kimi-K2.5	MT	90.0	25.0	0.0	80.0	10.0	0.0	34.2
	CAPSULE	**95.0**	**45.0**	**15.0**	80.0	**15.0**	0.0	**41.7** (+7.5 ↑)
GLM-4.7	MT	95.0	25.0	10.0	100.0	0.0	0.0	38.3
	CAPSULE	**100.0**	**50.0**	**15.0**	95.0	**10.0**	0.0	**45.0** (+6.7 ↑)
MiniMax-M2.5	MT	70.0	40.0	0.0	10.0	5.0	0.0	20.8
	CAPSULE	**90.0**	40.0	**5.0**	**55.0**	5.0	0.0	**32.5** (+11.7 ↑)
MiniMax-M2.7	MT	95.0	45.0	10.0	55.0	0.0	0.0	34.2
	CAPSULE	95.0	**65.0**	**20.0**	**60.0**	0.0	0.0	**40.0** (+5.8 ↑)
Gemini-3.7-Flash	MT	80.0	60.0	5.0	95.0	65.0	20.0	54.2
	CAPSULE	**100.0**	**95.0**	**35.0**	**100.0**	65.0	**45.0**	**73.3** (+19.2 ↑)
GPT-5.6-Terra	MT	85.0	45.0	10.0	50.0	30.0	0.0	36.7
	CAPSULE	**95.0**	**70.0**	**50.0**	**80.0**	**45.0**	0.0	**56.7** (+20.0 ↑)

These artifacts enable us to trace program modifications to their corresponding physical execution, while systematically analyzing recovery behaviors and failure causes. Given that Robosuite and LIBERO differ substantially in task structure, reward shaping, episode horizons, and access to privileged state information, we report their success rates and rewards separately.

3.2.1 PERFORMANCE COMPARISON

Table 1 reports the per-task success rates on the six ROBOSUITE tasks with the CaP-X MT baseline and CAPSULE. The tasks include *Cube Lift*, *Cube Stack*, *Cube Restack*, *Spill Wipe*, *Two-Arm Lift* (2A-L), and *Two-Arm Handover* (2A-H).

Averaged across all the eight backbones, CAPSULE consistently improves the macro-average success rate on ROBOSUITE from 36.7% to 49.7% (+13.0 pp). A similar trend is observed on LIBERO (Table 2), where the across-model average increases from 8.2% to 16.8% (+8.6 pp).

Despite these overall gains, severe task-level bottlenecks remain. On LIBERO, the average success rate reaches 16.8%, while on Robosuite, seven out of the eight backbones still fail entirely on 2A-H under CAPSULE (Table 1). Such failures reveal cases that runtime control cannot recover from, particularly when a task demands global structural replanning or bimanual coordination, or when irrecoverable prior actions leave no viable execution path under the available APIs.

3.2.2 ABLATION STUDIES

To assess the contribution of each component in CAPSULE, we systematically evaluate:

- **A1**: One-shot execution of the original generated program without CAPSULE controller steps.
- **A2**: CAPSULE with `patch_group` enabled and `append_recovery` disabled.
- **A3**: Full CAPSULE with both `patch_group` and `append_recovery` enabled.

Figure 3 shows that moving from one-shot execution (A1) to patch-enabled runtime control (A2) increases the macro-average success from 26.3% to 34.3% on Robosuite and from 6.0% to 8.0% on LIBERO. Enabling forward recovery (A3) further increases these rates to 49.7% and 16.8%, respectively. The A2-to-A3 gains support the contribution of forward recovery beyond local patching,

Table 2: Task success rates (%) on the LIBERO benchmark. Avg. denotes the unweighted mean across all three suites. Values in parentheses indicate the absolute percentage-point (pp) gains of CAPSULE over the CaP-X MT baseline.

Model	Setting	Success rate (%)			
		Object	Spatial	Goal	Avg.
DeepSeek-V4-Flash	MT	19.0	8.0	6.0	11.0
	CAPSULE	**25.0**	**22.0**	**8.0**	**18.3** (+7.3 ↑)
Qwen3.7-Plus	MT	8.0	5.0	2.0	5.0
	CAPSULE	**11.0**	**7.0**	**4.0**	**7.3** (+2.3 ↑)
Kimi-K2.5	MT	3.0	4.0	4.0	3.7
	CAPSULE	**12.0**	**8.0**	3.0	**7.7** (+4.0 ↑)
GLM-4.7	MT	13.0	7.0	5.0	8.3
	CAPSULE	**22.0**	**14.0**	**6.0**	**14.0** (+5.7 ↑)
MiniMax-M2.5	MT	10.0	3.0	0.0	4.3
	CAPSULE	**15.0**	**9.0**	**8.0**	**10.7** (+6.3 ↑)
MiniMax-M2.7	MT	13.0	5.0	2.0	6.7
	CAPSULE	**16.0**	**13.0**	**5.0**	**11.3** (+4.7 ↑)
Gemini-3.7-Flash	MT	23.0	2.0	21.0	15.3
	CAPSULE	**39.0**	**30.0**	**30.0**	**33.0** (+17.7 ↑)
GPT-5.6-Terra	MT	22.0	7.0	5.0	11.3
	CAPSULE	**47.0**	**26.0**	**22.0**	**31.7** (+20.3 ↑)

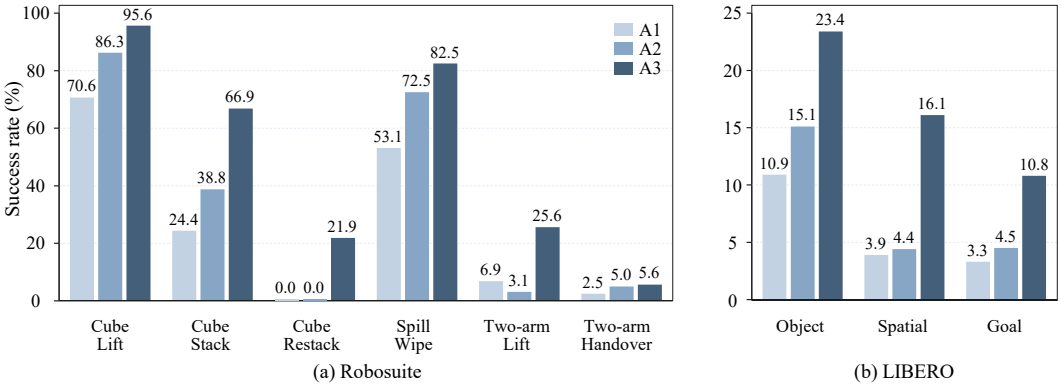

Figure 3: Component ablations of CAPSULE on Robosuite and LIBERO. Success rates are averaged equally across eight model backbones for (a) six Robosuite tasks and (b) three LIBERO suites.

particularly when execution must continue from a physical state already altered by earlier actions. Detailed per-task results are provided in Appendix Tables 5 and 7.

A representative *Cube Stack* execution trace in Appendix A further illustrates how local patching and forward recovery complement each other within a single rollout. The controller first applies an in-place patch to correct the lifting-target computation while reusing intermediate variables preserved in the persistent namespace Γ_t. When the subsequent placement fails, the controller appends a recovery block that refreshes the cube poses, resamples the grasp pose, and recalculates the placement height. Executing this continuation successfully completes the task without discarding earlier progress or restarting the entire rollout.

3.3 CAPSULE-CRITIQUE-GRPO

Despite the overall effectiveness of runtime control, Table 3a reveals that CAPSULE does not consistently benefit smaller-scale models. On *Cube Stack*, success rates drop from 10% to 5% for

Table 3: Performance of smaller-scale models with runtime control and post-training results with CC-GRPO. **Top:** Success rates (%) of MT and CAPSULE across six Robosuite tasks using three smaller-scale backbone models. Arrows show changes in average success rate relative to MT in percentage points. **Bottom:** Post-training comparison on Qwen2.5-Coder-7B across base model, vanilla GRPO, and CC-GRPO (Cube Stack and Spill Wipe), evaluated standalone without the runtime control layer.

(a) Performance of smaller-scale models with MT and CAPSULE

Model	Setting	Success rate (%)						
		Lift	Stack	Restack	Wipe	Arm Lift	Handover	Avg.
Qwen2.5-Coder-7B	MT	**35.0**	**10.0**	0.0	15.0	0.0	0.0	**10.0**
	CAPSULE	20.0	5.0	0.0	**30.0**	0.0	0.0	9.2 (-0.8 ↓)
Qwen3-8B	MT	0.0	**30.0**	0.0	**15.0**	0.0	0.0	7.5
	CAPSULE	**65.0**	10.0	0.0	0.0	0.0	0.0	**12.5** (+5.0 ↑)
Qwen3-30B-A3B	MT	15.0	**25.0**	**10.0**	**55.0**	0.0	0.0	17.5
	CAPSULE	**55.0**	10.0	0.0	45.0	0.0	0.0	**18.3** (+0.8 ↑)

(b) Post-training comparison (Base, Vanilla GRPO, and CC-GRPO) on Qwen2.5-Coder-7B

Method	Cube Stack Success (%)	Δ (pp)	Spill Wipe Success (%)	Δ (pp)
Base (standalone)	10.00	—	13.75	—
Vanilla GRPO	17.50	+7.50 ↑	16.25	+2.50 ↑
CC-GRPO	**31.25**	+21.25 ↑	**28.75**	+15.00 ↑

Qwen2.5-Coder-7B, from 30% to 10% for Qwen3-8B, and from 25% to 10% for Qwen3-30B-A3B. On *Spill Wipe*, runtime control improves Qwen2.5-Coder-7B but degrades performance on the other two models. These observations suggest that runtime control alone may induce performance regressions on smaller-scale models, motivating us to investigate whether the same execution feedback could instead serve as a useful training signal during post-training.

To test this hypothesis, we select *Cube Stack* and *Spill Wipe* as representative tasks, as they capture distinct manipulation challenges (object stacking versus surface wiping) and exhibit pronounced smaller-scale model regressions. We investigate whether CAPSULE critique can guide verified program revisions to improve standalone policy generation via reinforcement learning. Specifically, we benchmark the base Qwen2.5-Coder-7B model against vanilla GRPO and our CC-GRPO. While vanilla GRPO optimizes policy generation using standard rollouts and environment rewards, CC-GRPO incorporates critique-guided, independently verified program revisions into the training objective. All models are evaluated without runtime assistance to isolate the gains in standalone code generation. Detailed training configurations, candidate verification, and evaluation protocols are provided in Appendix C.3.

As shown in Table 3b, CC-GRPO achieves success rates of 31.25% on *Cube Stack* and 28.75% on *Spill Wipe*, exhibiting absolute improvements of 13.75% and 12.50% over vanilla GRPO (17.50% and 16.25%), respectively. These substantial improvements demonstrate that critique-assisted post-training can improve standalone program generation on these tasks. However, since CC-GRPO is evaluated only on these tasks and without reintegration into the full CAPSULE framework, these gains do not establish improved runtime control or transfer to the remaining manipulation domains.

Together with the runtime experiments, these findings demonstrate two complementary uses of execution-grounded feedback across inference and training. During runtime execution, CAPSULE leverages this feedback to guide local patching and forward recovery. During post-training, CC-GRPO incorporates critique-guided, independently verified program revisions to directly improve standalone program generation.

4 Related Work

Embodied code generation and self-improvement. LLMs have emerged as effective policy generators by transforming natural-language instructions into executable robotic programs. Foundation frameworks such as CaP synthesize control flows by composing perception and planning APIs (Liang et al., 2023), while subsequent approaches integrate 3D geometric affordances through value maps (Huang et al., 2023) or relational keypoint constraints (Huang et al., 2024). To improve policy robustness, recent systems incorporate multi-turn execution feedback and synthesize reusable skill libraries (Fu et al., 2026), or leverage historical interaction traces to refine program synthesis across successive trials (Chen et al., 2026; Lu et al., 2026; Xiao et al., 2026; Zhang et al., 2026c;a). While these methods advance program quality across repeated attempts from scratch, they treat each execution as an isolated trial. In contrast, CAPSULE addresses the challenge of intra-execution runtime control within a single ongoing rollout, preserving active variable bindings in a persistent namespace while explicitly tracking physical state changes that cannot be undone.

Execution feedback and runtime control. Incorporating environment feedback during execution is essential for detecting failures and guiding corrective actions. In high-level task planning, systems leverage 3D scene-graph simulators or closed-loop world models to identify discrepancies and adjust subsequent subgoals (Rana et al., 2023; Guo et al., 2024; Ju et al., 2026; Zhang et al., 2026b; Neary et al., 2025). In programmatic reasoning, iterative self-correction paradigms such as Self-Debugging and Reflexion utilize execution errors and verbal critiques to regenerate candidate solutions (Chen et al., 2024; Zhang et al., 2023; Shinn et al., 2023). However, existing methods typically perform coarse-grained, full-program regeneration without binding execution anomalies to specific source-code spans, and they fail to distinguish whether physical effects have already changed the physical environment. CAPSULE resolves this limitation by structuring programs into effect-bounded execution groups. Runtime feedback is mapped directly to the active source span to enable localized patching prior to state modification, while unrecoverable effectful deviations trigger observation-grounded forward recovery to avoid unsafe state rollbacks.

Group-relative policy optimization and post-training. Reinforcement learning from verifiable rewards has significantly advanced mathematical reasoning and programmatic synthesis in foundation models. GRPO eliminates the requirement for a separate critic network by computing advantage estimates across batches of sampled outputs (Shao et al., 2024; DeepSeek-AI, 2025), with subsequent variants improving optimization stability and sample efficiency (Yu et al., 2025). To bridge natural-language feedback and reinforcement learning, Critique-GRPO incorporates verbal critiques to guide candidate proposal generation while anchoring policy updates to verifiable task outcomes (Zhang et al., 2025). Building upon this formulation, our proposed CC-GRPO uses physical repair trajectories generated by CAPSULE to guide policy optimization. Repaired candidate programs are included in the policy optimization objective only after passing full execution verification initialized from the pristine environment state. This design allows repair experience to guide candidate exploration while ensuring that gradient updates are driven only by independently verified task success.

5 Conclusion

We presented CAPSULE, a runtime control layer for CaP agents that supports group-level execution and effect-aware program repair. Across eight backbone LLMs evaluated on Robosuite and LIBERO, CAPSULE consistently improves average task success rates over the CaP-X multi-turn baseline. These results demonstrate the clear advantage of grounding program recovery in fine-grained physical feedback over full-program regeneration. Furthermore, we showed that the collected repair trajectories provide valuable supervision for post-training. CC-GRPO substantially improves standalone program generation over vanilla GRPO on two evaluated tasks without requiring CAPSULE at test time. Together, these findings highlight that execution-grounded feedback serves not only as an effective runtime recovery mechanism, but also as a powerful training signal to advance autonomous policy synthesis. Future work will extend this framework to long-horizon, real-world robotic manipulation and investigate the cross-task and cross-embodiment transferability of repair-derived supervision.

6 AI USE STATEMENT

Generative AI tools were used for language polishing, code editing, figure modification, and literature search. We did not use generative AI tools to generate synthetic datasets, formulate mathematical claims, or support qualitative and thematic data analysis.

We reviewed all AI-assisted work, including inspecting the code and checking the figures and cited literature for correctness. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Kaiyuan Chen, Shuangyu Xie, Letian Fu, Justin Yu, William Pacini, Sandeep Bajamahal, Hudson Kim, Jaimyn Drake, Daehwa Kim, Haoru Xue, Jonathan Francis, Christian Juette, Peter Schaldenbrand, Muhammet Yunus Seker, Ruwan Wickramarachchi, Uksang Yoo, Guanzhi Wang, Adithyavairavan Murali, Balakumar Sundaralingam, S. Shankar Sastry, Spencer Huang, Yuke Zhu, Linxi "Jim" Fan, and Ken Goldberg. GaP: A graph-as-policy multi-agent self-learning harness for variational automation tasks. 2026. URL https://arxiv.org/abs/2607.05369.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/2460396f2d0d421885997dd1612ac56b-Abstract-Conference.html.

DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. 2025. URL https://arxiv.org/abs/2501.12948.

Letian Fu, Justin Yu, Karim El-Refai, Ethan Kou, Haoru Xue, Huang Huang, Wenli Xiao, Guanzhi Wang, Dantong Niu, Fei-Fei Li, Guanya Shi, Jiajun Wu, Shankar Sastry, Yuke Zhu, Ken Goldberg, and Linxi Fan. CaP-X: A framework for benchmarking and improving coding agents for robot manipulation. 2026. URL https://arxiv.org/abs/2603.22435.

Yanjiang Guo, Yen-Jen Wang, Lihan Zha, and Jianyu Chen. DoReMi: Grounding language model by detecting and recovering from plan-execution misalignment. 2024. URL https://arxiv.org/abs/2307.00329.

Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. VoxPoser: Composable 3d value maps for robotic manipulation with language models. 2023. URL https://arxiv.org/abs/2307.05973.

Wenlong Huang, Chen Wang, Yunzhu Li, Ruohan Zhang, and Li Fei-Fei. ReKep: Spatio-temporal reasoning of relational keypoint constraints for robotic manipulation. 2024. URL https://arxiv.org/abs/2409.01652.

Yuanchen Ju, Yongyuan Liang, Yen-Jen Wang, Gireesh Nandiraju, Yuanliang Ju, Seungjae Lee, Qiao Gu, Elvis Hsieh, Furong Huang, and Koushil Sreenath. MomaGraph: State-aware unified scene graphs with vision-language models for embodied task planning. 2026. URL https://arxiv.org/abs/2512.16909.

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. 2023. URL https://arxiv.org/abs/2209.07753.

Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. LIBERO: Benchmarking knowledge transfer for lifelong robot learning. 2023. URL https://arxiv.org/abs/2306.03310.

Runyu Lu, Yubo Wu, Ethan Kou, Letian Fu, Wenli Xiao, Ajay Mandlekar, Yinzhen Xu, Guanya Shi, Ken Goldberg, Ang Chen, Mosharaf Chowdhury, Yuke Zhu, Linxi "Jim" Fan, and Guanzhi Wang. ASPIRE: Agentic skills discovery for robotics. 2026. URL https://arxiv.org/abs/2607.00272.

Cyrus Neary, Omar G. Younis, Artur Kuramshin, Ozgur Aslan, and Glen Berseth. Improving pre-trained vision-language-action policies with model-based search. 2025. URL https://arxiv.org/abs/2508.12211.

Krishan Rana, Jesse Haviland, Sourav Garg, Jad Abou-Chakra, Ian Reid, and Niko Suenderhauf. SayPlan: Grounding large language models using 3d scene graphs for scalable robot task planning. 2023. URL https://proceedings.mlr.press/v229/rana23a.html.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. 2024. URL https://arxiv.org/abs/2402.03300.

Ying Sheng et al. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv:2409.19256*, 2024. URL https://arxiv.org/abs/2409.19256.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. 2023. URL https://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Wenli Xiao, Jia Xie, Tonghe Zhang, Haotian Lin, Letian "Max" Fu, Haoru Xue, Jalen Lu, Yi Yang, Cunxi Dai, Zi Wang, Jimmy Wu, Guanzhi Wang, S. Shankar Sastry, Ken Goldberg, Linxi "Jim" Fan, Yuke Zhu, and Guanya Shi. ENPIRE: Agentic robot policy self-improvement in the real world. 2026. URL https://arxiv.org/abs/2606.19980.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Weinan Dai, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. 2025. URL https://arxiv.org/abs/2503.14476.

Chunhui Zhang, Yuxuan Wang, Aoyang Qin, Yi-Long Lu, Kunlun Wu, Yizhou Wang, and Wei Wang. Bridging values and behavior: A hierarchical framework for proactive embodied agents. 2026a. URL https://arxiv.org/abs/2604.27699.

Jiahan Zhang, Muqing Jiang, Nanru Dai, Taiming Lu, Arda Uzunoglu, Shunchi Zhang, Yana Wei, Jiahao Wang, Vishal M. Patel, Paul Pu Liang, Daniel Khashabi, Cheng Peng, Rama Chellappa, Tianmin Shu, Alan Yuille, Yilun Du, and Jieneng Chen. World-In-World: World models in a closed-loop world. 2026b. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/5b4263be85820683d78675cc18d2efc7-Abstract-Conference.html.

Junyi Zhang, Jiaxin Ge, Hanjun Yoo, Letian Fu, Zihan Yang, Yaowei Liu, Raj Saravanan, Shaofeng Yin, Justin Yu, Dantong Niu, Zirui Wang, Roei Herzig, Ken Goldberg, Yutong Bai, David M. Chan, Ion Stoica, Angjoo Kanazawa, Jiahui Lei, Haiwen Feng, and Trevor Darrell. Playful agentic robot learning. 2026c. URL https://arxiv.org/abs/2606.19419.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. Self-edit: Fault-aware code editor for code generation. 2023. URL https://aclanthology.org/2023.acl-long.45/.

Xiaoying Zhang, Yipeng Zhang, Hao Sun, Kaituo Feng, Chaochao Lu, Chao Yang, and Helen Meng. Critique-GRPO: Advancing LLM reasoning with natural language and numerical feedback. 2025. URL https://arxiv.org/abs/2506.03106.

Yuke Zhu, Josiah Wong, Ajay Mandlekar, Roberto Martín-Martín, Abhishek Joshi, Kevin Lin, Abhiram Maddukuri, Soroush Nasiriany, and Yifeng Zhu. robosuite: A modular simulation framework and benchmark for robot learning. 2020. URL https://arxiv.org/abs/2009.12293.

A CASE STUDY

The following trajectory illustrates how CAPSULE seamlessly coordinates step execution, local patching, and forward recovery on a representative cube-stacking task. The initial program encounters a runtime Python error while computing the lifting trajectory. Instead of discarding the execution history and regenerating the whole plan, CAPSULE isolates and patches only the failing effect-bounded group, resuming execution directly from the persistent namespace and the current physical state. After this local repair succeeds, the subsequent physical placement fails due to unmodeled geometric discrepancies. To resolve this physical failure, CAPSULE synthesizes and appends a forward recovery block that refreshes object poses, resamples the grasp, and recalculates the placement height using the measured cube dimensions. The newly generated recovery code is then executed incrementally, ultimately bringing the task to successful completion.

A Case Study of CAPSULE Runtime Control

Initial Program

The agent initially generates the following program for stacking the red cube on top of the green cube:

```
import numpy as np

home_pose()
open_gripper()

green_pos, green_quat, green_bbox = get_object_pose(
    "green cube",
    return_bbox_extent=True,
)

placement_quat = np.array([0, 0, 1, 0])

grasp_pos, grasp_quat = sample_grasp_pose("red cube")

goto_pose(grasp_pos, grasp_quat, z_approach=0.1)
goto_pose(grasp_pos, grasp_quat)

close_gripper()

lift_pos = grasp_pos
lift_pos[2] += 0.1

goto_pose(lift_pos, grasp_quat)

if green_bbox is not None:
    green_height = green_bbox[2]
    red_height = (
        green_bbox[2]
        if green_bbox is not None
        else 0.05
    )

    placement_z = (
        green_pos[2]
        + green_height / 2
        + red_height / 2
    )
else:
    placement_z = green_pos[2] + 0.05

placement_pos = np.array([
    green_pos[0],
    green_pos[1],
    placement_z,
])

goto_pose(
    placement_pos,
    placement_quat,
    z_approach=0.1,
)
goto_pose(placement_pos, placement_quat)
```

```
open_gripper()
```

Step 1: Initialize the robot
Controller Action: `run_group(group_1)`
Execution:

```
import numpy as np

home_pose()
open_gripper()
```

The robot first moves to its predefined home pose and opens the gripper in preparation for grasping.
Outcome: Both API calls return normally. The reward changes from 0.003441 to 0.002167, while `task_completed=False`. The slight decrease in reward does not indicate an execution failure, since this group only performs initialization.

Step 2: Query the target and attempt the initial grasp
Controller Action: `run_group(group_2)`
Execution:

```
green_pos, green_quat, green_bbox = get_object_pose(
    "green cube",
    return_bbox_extent=True,
)

placement_quat = np.array([0, 0, 1, 0])

grasp_pos, grasp_quat = sample_grasp_pose("red cube")

goto_pose(grasp_pos, grasp_quat, z_approach=0.1)
goto_pose(grasp_pos, grasp_quat)

close_gripper()
```

The group queries the green cube, samples a grasp pose for the red cube, moves to the sampled grasp target, and closes the gripper.

Step 3: Attempt to lift the red cube by 10 cm
Controller Action: `run_group(group_3)`
The original group contains:

```
lift_pos = grasp_pos
lift_pos[2] += 0.1

goto_pose(lift_pos, grasp_quat)
```

Outcome: Execution fails before the lifting motion with

```
ValueError: assignment destination is read-only
```

The failure is therefore localized to the source group responsible for constructing the lifting target.

Step 4: Patch the failed lifting group
Controller Action: `patch_group(group_3, source=...)`
Instead of regenerating the entire program, CAPSULE replaces only the failed group:

```
lift_pos = grasp_pos.copy()
lift_pos[2] += 0.1

goto_pose(lift_pos, grasp_quat)
```

The essential modification is

$$\texttt{lift_pos = grasp_pos} \quad \longrightarrow \quad \texttt{lift_pos = grasp_pos.copy()}.$$

This creates a writable copy before modifying the vertical coordinate.

Step 5: Re-execute the patched lifting group

Controller Action: `run_group(group_3)`
Execution:

```
lift_pos = grasp_pos.copy()
lift_pos[2] += 0.1

goto_pose(lift_pos, grasp_quat)
```

The same execution group can now proceed past the Python error and issue the lifting command without restarting the preceding initialization and grasping groups.

Step 6: Execute the original placement group
Controller Action: `run_group(group_4)`
Execution:

```
if green_bbox is not None:
    green_height = green_bbox[2]
    red_height = (
        green_bbox[2]
        if green_bbox is not None
        else 0.05
    )

    placement_z = (
        green_pos[2]
        + green_height / 2
        + red_height / 2
    )
else:
    placement_z = green_pos[2] + 0.05

placement_pos = np.array([
    green_pos[0],
    green_pos[1],
    placement_z,
])

goto_pose(
    placement_pos,
    placement_quat,
    z_approach=0.1,
)
goto_pose(placement_pos, placement_quat)

open_gripper()
```

Outcome: The group executes, but the stacking task is not completed.
Two properties of the original placement logic are relevant. First, the group continues to use the green-cube position and dimensions stored during Step 2, without refreshing the scene state. Second, the red-cube height is not queried independently. Instead, the program assumes

```
red_height = green_bbox[2]
```

and therefore uses the green-cube height as a proxy for the red-cube height when computing the vertical placement target.

Step 7: Append a forward recovery program
Controller Action: `append_recovery(source=...)`
Since the original execution has already altered the physical scene, CAPSULE does not restart the task from the beginning. Instead, it appends the following recovery program:

```
import numpy as np

green_pos, green_quat, green_bbox = get_object_pose(
    "green cube",
    return_bbox_extent=True,
)

red_pos, red_quat, red_bbox = get_object_pose(
    "red cube",
    return_bbox_extent=True,
)
```

```
print(f"Green cube position: {green_pos}")
print(f"Red cube position: {red_pos}")
print(f"Green bbox: {green_bbox}")

grasp_pos, grasp_quat = sample_grasp_pose("red cube")
print(f"Grasp pose: {grasp_pos}, {grasp_quat}")

goto_pose(grasp_pos, grasp_quat, z_approach=0.1)
goto_pose(grasp_pos, grasp_quat)
close_gripper()

lift_pos = grasp_pos.copy()
lift_pos[2] += 0.1
goto_pose(lift_pos, grasp_quat)

placement_quat = np.array([0, 0, 1, 0])

if green_bbox is not None:
    green_height = green_bbox[2]
    red_height = (
        green_bbox[2]
        if red_bbox is None
        else red_bbox[2]
    )

    placement_z = (
        green_pos[2]
        + green_height / 2
        + red_height / 2
    )
else:
    placement_z = green_pos[2] + 0.05

placement_pos = np.array([
    green_pos[0],
    green_pos[1],
    placement_z,
])

print(f"Placement position: {placement_pos}")

goto_pose(
    placement_pos,
    placement_quat,
    z_approach=0.1,
)
goto_pose(placement_pos, placement_quat)

open_gripper()

retreat_pos = placement_pos.copy()
retreat_pos[2] += 0.1
goto_pose(retreat_pos, placement_quat)

home_pose()
```

Compared with the original attempt, the recovery program introduces three main changes:

1. It re-queries the positions and dimensions of both cubes, replacing the previously stored scene information.
2. It resamples a new grasp pose for the red cube.
3. When the red-cube bounding box is available, the placement calculation uses `red_bbox[2]` rather than assuming that the red and green cubes have the same height.

Step 8: Refresh the states of both cubes
Controller Action: `run_group(group_5)`
Execution:

```
import numpy as np

green_pos, green_quat, green_bbox = get_object_pose(
    "green cube",
```

```
    return_bbox_extent=True,
)

red_pos, red_quat, red_bbox = get_object_pose(
    "red cube",
    return_bbox_extent=True,
)

print(f"Green cube position: {green_pos}")
print(f"Red cube position: {red_pos}")
print(f"Green bbox: {green_bbox}")
```

The recovery trajectory first refreshes the scene state. The newly queried positions and bounding boxes overwrite the stale variables from the original execution and are subsequently used by the remaining recovery groups.

Step 9: Resample the grasp and close the gripper
Controller Action: `run_group(group_6)`
Execution:

```
grasp_pos, grasp_quat = sample_grasp_pose("red cube")
print(f"Grasp pose: {grasp_pos}, {grasp_quat}")

goto_pose(grasp_pos, grasp_quat, z_approach=0.1)
goto_pose(grasp_pos, grasp_quat)

close_gripper()
```

Rather than reusing the initial grasp target, the recovery program invokes the grasp sampler again under the current scene state and initiates a new grasp attempt.

Step 10: Lift, place, and release the red cube
Controller Action: `run_group(group_7)`
Execution:

```
lift_pos = grasp_pos.copy()
lift_pos[2] += 0.1
goto_pose(lift_pos, grasp_quat)

placement_quat = np.array([0, 0, 1, 0])

if green_bbox is not None:
    green_height = green_bbox[2]
    red_height = (
        green_bbox[2]
        if red_bbox is None
        else red_bbox[2]
    )

    placement_z = (
        green_pos[2]
        + green_height / 2
        + red_height / 2
    )
else:
    placement_z = green_pos[2] + 0.05

placement_pos = np.array([
    green_pos[0],
    green_pos[1],
    placement_z,
])

print(f"Placement position: {placement_pos}")

goto_pose(
    placement_pos,
    placement_quat,
    z_approach=0.1,
)
goto_pose(placement_pos, placement_quat)

open_gripper()
```

The program now computes the placement target using the refreshed green-cube state and, when available, the actual height of the red cube. It then lifts the newly grasped red cube, moves to the recomputed placement target, and opens the gripper.
Outcome: The environment reports successful task completion:

$$\texttt{task_completed} : \text{False} \rightarrow \text{True}.$$

Step 11: Finish execution
Controller Action: `finish()`
No additional effectful API call is executed after the task has already been completed.

B MORE EXPERIMENTAL DETAILS

B.1 MORE DETAILS ON ROBOSUITE

B.1.1 AVERAGE REWARD RESULTS

Table 4 reports the per-task average rewards of all evaluated models under the MT and CAPSULE settings.

Table 4: Average rewards across six robosuite tasks under the MT and CAPSULE settings. The two settings of each model are placed in adjacent rows. Within each model, the higher reward for each task is shown in bold, while ties are left unbolded. Avg. denotes the unweighted macro-average over the six tasks.

Model	Setting	Average reward						
		Lift	Stack	Restack	Wipe	2A-L	2A-H	Avg.
DeepSeek-V4-Flash	MT	0.950	0.533	0.004	0.956	0.158	0.062	0.444
	CAPSULE	**0.977**	**0.861**	**0.277**	**0.982**	**0.414**	**0.169**	**0.613**
Qwen3.7-Plus	MT	0.929	0.284	0.005	1.000	0.036	0.053	0.385
	CAPSULE	**0.975**	**0.862**	**0.160**	1.000	**0.123**	**0.067**	**0.531**
Kimi-K2.5	MT	0.956	0.270	0.036	0.955	0.116	0.042	0.396
	CAPSULE	**0.977**	**0.466**	**0.208**	**0.971**	**0.118**	**0.059**	**0.467**
GLM-4.7	MT	0.976	0.481	**0.206**	**1.000**	0.053	0.101	0.470
	CAPSULE	**1.000**	**0.500**	0.154	0.989	**0.137**	**0.124**	**0.484**
MiniMax-M2.5	MT	0.855	0.404	0.013	0.510	**0.096**	**0.117**	0.333
	CAPSULE	**0.951**	**0.449**	**0.065**	**0.753**	0.076	0.063	**0.393**
MiniMax-M2.7	MT	**0.975**	0.460	0.116	**0.814**	**0.022**	**0.114**	0.417
	CAPSULE	0.974	**0.765**	**0.205**	0.782	0.016	0.048	**0.465**
Gemini-3.7-Flash	MT	0.903	0.603	0.067	0.987	**0.679**	0.244	0.581
	CAPSULE	**1.000**	**0.950**	**0.379**	**1.000**	0.650	**0.476**	**0.743**
GPT-5.6-Terra	MT	0.904	0.454	0.105	0.620	0.225	**0.087**	0.399
	CAPSULE	**0.950**	**0.702**	**0.508**	**0.800**	**0.407**	0.003	**0.562**

B.1.2 ABLATION RESULTS

Tables 5 and 6 report per-model and per-task success rates and average rewards for the A1–A3 variants defined in the main text. Avg. denotes the unweighted mean over the six tasks.

Table 5: Detailed Robosuite ablation success rates (%). Notation: A1 (one-shot execution), A2 (patching only), and A3 (full CAPSULE). Avg. is the unweighted mean over six tasks.

Model	Variant	Lift	Stack	Restack	Wipe	Two-arm Lift	Two-arm Handover	Avg.
DeepSeek-V4-Flash	A1	85	20	0	80	0	0	30.8
	A2	85	40	0	80	0	0	34.2
	A3	95	85	20	90	55	0	57.5
Qwen3.7-Plus	A1	90	10	0	90	0	0	31.7
	A2	95	70	0	100	0	0	44.2
	A3	95	85	15	100	10	0	50.8
Kimi-K2.5	A1	80	5	0	75	0	0	26.7
	A2	95	35	0	65	0	0	32.5
	A3	95	45	15	80	15	0	41.7
GLM-4.7	A1	80	30	0	50	0	0	26.7
	A2	90	40	0	90	5	0	37.5
	A3	100	50	15	95	10	0	45.0
MiniMax-M2.5	A1	35	20	0	0	0	0	9.2
	A2	60	20	0	15	5	0	16.7
	A3	90	40	5	55	5	0	32.5
MiniMax-M2.7	A1	30	0	0	5	0	0	5.8
	A2	90	15	0	55	10	0	28.3
	A3	95	65	20	60	0	0	40.0
Gemini-3.7-Flash	A1	80	60	0	95	30	20	47.5
	A2	90	70	0	100	0	40	50.0
	A3	100	95	35	100	65	45	73.3
GPT-5.6-Terra	A1	85	50	0	30	25	0	31.7
	A2	85	20	0	75	5	0	30.8
	A3	95	70	50	80	45	0	56.7

B.2 MORE DETAILS ON LIBERO

B.2.1 ABLATION RESULTS

Table 7 reports success rates for all eight model backbones under the three ablation configurations. Specifically, A1 executes the original generated program once without controller steps, A2 enables only `patch_group`, and A3 enables both operations.

C EXPERIMENTAL PROTOCOL

C.1 MODELS

The main cross-benchmark study evaluates eight models: DeepSeek-V4-Flash, Qwen3.7-Plus, Kimi-K2.5, GLM-4.7, MiniMax-M2.5, MiniMax-M2.7, Gemini-3.7-Flash, and GPT-5.6-Terra. The small-model analysis additionally includes Qwen2.5-Coder-7B, Qwen3-8B, and Qwen3-30B-A3B, resulting in eleven models overall.

For the main cross-benchmark evaluation, we set the temperature to 0.2 and limit generated outputs to 4096 tokens. The RL study uses the separate decoding configuration reported in Appendix C.3.

C.2 OBSERVATION ACCESS AND EXECUTION BUDGETS

We use non-privileged perceptual observations for the Robosuite experiments and privileged perceptual observations for the LIBERO experiments. Within each benchmark, MT and CAPSULE

Table 6: Detailed Robosuite ablation average rewards. Notation: A1 (one-shot execution), A2 (patching only), and A3 (full CAPSULE). Avg. is the unweighted mean over the six tasks, computed from the displayed task-level values.

Model	Variant	Lift	Stack	Restack	Wipe	Two-arm Lift	Two-arm Handover	Avg.
DeepSeek-V4-Flash	A1	0.923	0.202	0.002	0.947	0.077	0.006	0.360
	A2	0.929	0.404	0.002	0.955	0.032	0.034	0.393
	A3	0.977	0.861	0.277	0.982	0.414	0.169	0.613
Qwen3.7-Plus	A1	0.954	0.154	0.010	0.969	0.026	0.043	0.359
	A2	0.975	0.703	0.007	1.000	0.037	0.050	0.462
	A3	0.975	0.862	0.160	1.000	0.123	0.067	0.531
Kimi-K2.5	A1	0.904	0.228	0.040	0.890	0.075	0.157	0.382
	A2	0.976	0.362	0.016	0.876	0.031	0.151	0.402
	A3	0.977	0.466	0.208	0.971	0.118	0.059	0.467
GLM-4.7	A1	0.903	0.304	0.008	0.911	0.033	0.109	0.378
	A2	0.974	0.401	0.007	0.975	0.053	0.032	0.407
	A3	1.000	0.500	0.154	0.989	0.137	0.124	0.484
MiniMax-M2.5	A1	0.662	0.299	0.009	0.031	0.051	0.096	0.191
	A2	0.706	0.200	0.001	0.414	0.075	0.076	0.245
	A3	0.951	0.449	0.065	0.753	0.076	0.063	0.393
MiniMax-M2.7	A1	0.650	0.007	0.007	0.131	0.043	0.093	0.155
	A2	0.952	0.155	0.011	0.987	0.116	0.057	0.380
	A3	0.974	0.765	0.205	0.782	0.016	0.048	0.465
Gemini-3.7-Flash	A1	0.903	0.603	0.018	0.987	0.315	0.253	0.513
	A2	0.952	0.705	0.005	1.000	0.009	0.460	0.522
	A3	1.000	0.950	0.379	1.000	0.650	0.476	0.743
GPT-5.6-Terra	A1	0.850	0.501	0.004	0.386	0.177	0.055	0.329
	A2	0.878	0.204	0.031	0.750	0.102	0.090	0.343
	A3	0.950	0.702	0.508	0.800	0.407	0.003	0.562

Table 7: Detailed LIBERO ablation results. All entries are success rates (%), evaluated over 100 trials per suite.

Model	LIBERO-Object			LIBERO-Spatial			LIBERO-Goal		
	A1	A2	A3	A1	A2	A3	A1	A2	A3
DeepSeek-V4-Flash	18	12	25	6	7	22	4	4	8
Qwen3.7-Plus	0	9	11	0	5	7	0	2	4
Kimi-K2.5	3	11	12	3	3	8	0	2	3
GLM-4.7	13	21	22	7	4	14	5	6	6
MiniMax-M2.5	2	11	15	1	4	9	0	4	8
MiniMax-M2.7	10	12	16	5	6	13	2	4	5
Gemini-3.7-Flash	19	29	39	2	3	30	15	9	30
GPT-5.6-Terra	22	16	47	7	3	26	0	5	22
Suite mean	10.88	15.13	23.38	3.88	4.38	16.13	3.25	4.50	10.75

are provided with the same observation access and environment APIs to ensure a fair comparison. Results are therefore compared within each benchmark under its respective observation setting.

For MT, we allow at most five regeneration attempts after the initial program generation. For CAPSULE, we impose a budget of 30 runtime control steps, where each step corresponds to a control decision, such as executing a code group, inspecting execution feedback, or applying a repair. MT and CAPSULE use different interaction granularities: MT regenerates complete programs, whereas

CAPSULE operates through finer-grained runtime decisions. The two methods are not matched in LLM calls or token consumption, and we do not conduct a matched-compute comparison. We additionally impose the same wall-clock timeout of $400\,\mathrm{s}$ per trial for both methods. A trial terminates when the task is completed, its method-specific budget is exhausted, or the timeout is reached.

C.3 RL TRAINING AND EVALUATION CONFIGURATION

Training setup. We benchmark vanilla GRPO against CC-GRPO using Qwen2.5-Coder-7B-Instruct as the shared trainable program generator. During training, the agent interacts with privileged task APIs to verify execution status and facilitate controller intervention, whereas evaluation strictly relies on non-privileged perceptual APIs operating without runtime assistance. All experiments are conducted on a single NVIDIA A800 80GB GPU using the VeRL framework (Sheng et al., 2024). The detailed sampling hyperparameters and task-specific configurations are summarized in Table 8.

Table 8: Configuration of the RL experiments. Group counts represent cumulative totals per training method. The frozen runtime Controller is utilized exclusively during CC-GRPO training.

Configuration	Cube Stack	Spill Wipe
Program generator model	Qwen2.5-Coder-7B-Instruct	
Controller model	Qwen3.7-Plus	GLM-5
Training groups per method	32	48
Training scene seeds	32	48
Candidate programs per group (G)	8	8
Ordinary samples before critique	7	7
Maximum failed programs selected for repair	2	2
Repair trajectories per selected program	2	2
Program sampling temperature	0.7	0.7
Top-p	0.8	0.8
Top-k	20	20
Repetition penalty	1.1	1.1
Maximum generated tokens	4096	4096
Evaluation scene seeds	20	20
Evaluated rollouts per scene	4	4
Total evaluated programs per policy	80	80
Runtime Controller at evaluation	Disabled	Disabled

Parameter-efficient fine-tuning. For both Cube Stack and Spill Wipe, we fine-tune Qwen2.5-Coder-7B-Instruct via LoRA with rank $r = 16$, scaling factor $\alpha = 32$, and zero dropout. Adapters are attached to all linear projections across both the self-attention block (`q_proj`, `k_proj`, `v_proj`, and `o_proj`) and the feed-forward network (`gate_proj`, `up_proj`, and `down_proj`). All pretrained base parameters and biases remain frozen throughout training, ensuring that only the LoRA adapters are updated. Vanilla GRPO and CC-GRPO share this exact adapter layout.

We optimize the policy using AdamW with a constant learning rate of 10^{-5}, momentum parameters $(\beta_1, \beta_2) = (0.9, 0.999)$, and weight decay of 0.01, without a learning-rate warmup schedule. When continuing across training stages, we restore both the adapter weights and the full optimizer state rather than reinitializing them. For KL divergence regularization, the reference policy π_{ref} is instantiated by disabling the LoRA adapter on the initial pretrained model. During evaluation, each policy activates its trained adapter, whereas the base-model baseline operates without adapters.

C.3.1 OPTIMIZATION OBJECTIVE

Group advantage and policy loss formulation. Given a task specification prompt x, each training group contains $G = 8$ candidate programs denoted by $\{y_i\}_{i=1}^{G}$. Let $R_i \in \{0, 1\}$ represent the binary task completion reward independently verified in simulation for candidate i. We define the empirical

group baseline and sequence-level advantage as

$$\bar{R} = \frac{1}{G}\sum_{j=1}^{G} R_j, \qquad A_i = R_i - \bar{R}. \tag{4}$$

Every candidate in the group, including any verified guided revision, contributes equally to the baseline $\bar{R}$. We assign the scalar advantage A_i uniformly to all valid response tokens in y_i without dividing by the reward standard deviation. If all candidates in a group receive identical rewards, the advantages vanish identically to zero, and the corresponding actor updates along with KL regularization are skipped.

For standard samples generated without guidance, let $p_{i,t} = \pi_\theta(y_{i,t} \mid x, y_{i,<t})$ and $p_{i,t}^{\text{old}}$ denote the current and rollout token probabilities, respectively. Defining the importance ratio $\rho_{i,t} = p_{i,t}/p_{i,t}^{\text{old}}$, the clipped surrogate objective with threshold $\epsilon = 0.2$ is given by $b_{i,t} = \max\left(-A_i\rho_{i,t}, -A_i\,\text{clip}(\rho_{i,t}, 1-\epsilon, 1+\epsilon)\right)$. To preserve optimization stability, we apply dual clipping with parameter $c = 3.0$:

$$\ell_{i,t}^{\text{ord}} = \begin{cases} \min\left(b_{i,t}, -cA_i\right), & A_i < 0, \\ b_{i,t}, & A_i \geq 0, \end{cases} \tag{5}$$

where the log-ratio $\log \rho_{i,t}$ is bounded within $[-20, 20]$ prior to exponentiation.

Guided candidates are synthesized by incorporating critique feedback and interactive repair trajectories. However, their policy likelihoods are evaluated strictly under the original task prompt x to reflect the autonomous deployment distribution. Following the Critique-GRPO formulation (Zhang et al., 2025), we shape the loss for guided tokens as

$$\ell_{i,t}^{\text{guide}} = -A_i \frac{p_{i,t}}{p_{i,t} + \gamma} = -A_i\,\text{sigmoid}\left(\log p_{i,t} - \log \gamma\right), \qquad \gamma = 0.1, \tag{6}$$

which omits importance weighting and PPO clipping to accommodate the off-policy exploration mode. The constant γ serves as a probability scale parameter within the sigmoid function rather than a temporal discount factor. Letting $g_{i,t} \in \{0, 1\}$ denote an indicator variable for response tokens originating from a guided candidate, the combined policy gradient loss is expressed as $\ell_{i,t}^{\text{PG}} = (1 - g_{i,t})\ell_{i,t}^{\text{ord}} + g_{i,t}\ell_{i,t}^{\text{guide}}$.

To prevent catastrophic policy drift, we regularize the policy against the frozen reference model $p_{i,t}^{\text{ref}} = \pi_{\text{ref}}(y_{i,t} \mid x, y_{i,<t})$ using the low-variance KL divergence estimator in VeRL with coefficient $\beta = 10^{-3}$:

$$d_{i,t} = \text{clip}\left(\log p_{i,t}^{\text{ref}} - \log p_{i,t}, -20, 20\right), \qquad k_{i,t} = \text{clip}\left(\exp(d_{i,t}) - d_{i,t} - 1, -10, 10\right). \tag{7}$$

This regularization applies uniformly across all valid tokens for both sample types, with the entropy bonus set to zero. Let $m_{i,t} \in \{0, 1\}$ denote the valid token mask and $T_i = \sum_t m_{i,t}$ be the sequence length. Using a micro-batch size of one program and an accumulation factor of $1/G$, the overall optimization objective averaged across valid groups is

$$\mathcal{L}(\theta) = \frac{1}{G}\sum_{i=1}^{G}\frac{1}{T_i}\sum_t m_{i,t}\left[\ell_{i,t}^{\text{PG}} + \beta k_{i,t}\right]. \tag{8}$$

Vanilla GRPO employs the identical objective structure by setting $g_{i,t} = 0$ across all eight ordinary candidates.

Candidate generation and replay verification. Vanilla GRPO samples all eight candidate programs directly from the generator policy without intermediate controller involvement. In contrast, CC-GRPO initiates each group by collecting and executing seven ordinary programs. If any of these candidates fail during execution, the critique mechanism selects up to two failed programs for revision, exploring up to two repair trajectories per program. The first revised program that successfully completes an independent clean replay starting from the initial state is inserted as the eighth candidate. If all repair attempts fail or if no initial failures occur, an ordinary sample is generated to fill the final slot, ensuring that the seven initial samples remain unchanged. Crucially, online execution recovery during the repair phase does not qualify a candidate on its own. The entire revised script must pass an independent re-execution from scratch to receive a positive reward, guaranteeing that all training signals reflect true environmental outcomes rather than controller-assigned labels.

Training schedule and evaluation protocol. We train policies for Cube Stack and Spill Wipe across two and three sequential stages of 16 groups, respectively. Checkpoint resumption restores policy weights, optimizer states, scheduler steps, and random number generator seeds. While both algorithms share identical scheduled group counts, they differ in effective gradient update steps and total simulation budget.

During evaluation, each trained checkpoint generates four autonomous programs across 20 unseen evaluation scenes, yielding 80 total program executions under the decoding parameters in Table 8. Each program is executed exactly once without critique feedback or controller intervention, utilizing matched scene seeds, generation seeds, and initial physical state configurations. The reported success rates represent the unweighted arithmetic mean over all 80 executions rather than a best-of-four selection or an evaluation across 80 distinct scene seeds.