ROBOAWARE: LEARNING TO COORDINATE EMBODIED SKILLS FROM COUNTERFACTUAL OUTCOMES

Anonymous authors
Paper under double-blind review

ABSTRACT

Embodied coding agents can combine modular robot skills with frozen end-to-end policies, yet effective composition requires anticipating which policy family will succeed in the current physical state. We present **RoboAware**, which builds on coding agents' skill orchestration by learning only a state-conditioned responsibility coordinator from counterfactual outcomes. Inspired by the success of REPL, we propose the P^5 schema and formulate a hierarchical MDP based on it. P^5 organizes skills uniformly into five semantic stages, defining where responsibility can be compared. To address the lack of counterfactual branch outcomes in existing work, we introduce **State-Locked Counterfactual Branching (SCB)**, which restores the same training state to generate and execute a code block from each admissible family, exposing outcomes that selected-branch experience leaves unobserved. Building on this, we propose **Execution-Aware Learning (EAL)**, which combines Monte Carlo tree search with Q-learning to distill these outcomes into family-conditioned values. At deployment, the coordinator selects the policy family according to observable context, and the frozen coding agent generates the next local code block. Comprehensive single-episode evaluations on 100 tasks show that RoboAware reaches a **77.0%** overall success rate, with SOTA averages of 90.0% on RoboSuite, 73.8% on diverse LIBERO-Pro task clusters, and 90.0% on challenging RoboTwin bimanual tasks, outperforming existing code-as-policy and VLA-harness baselines.

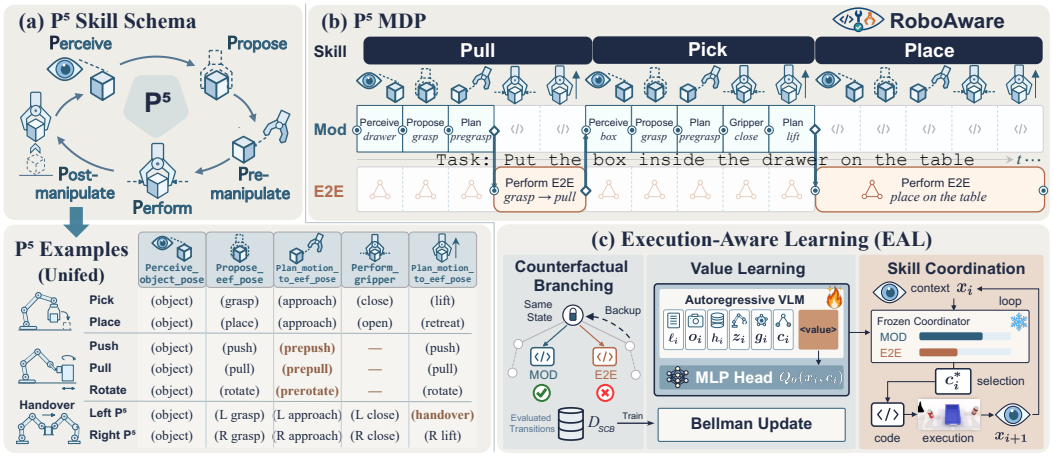


Figure 1: Overview of RoboAware. (a) The P^5 skill schema decomposes each skill into five stages—perceive, propose, pre-manipulate, perform, and post-manipulate—and unifies primitive skills (*e.g.*, pick, place, push, pull) and compositional skills (*e.g.*, handover) under one representation. (b) A hierarchical MDP selects a policy family $c_i \in \{\text{Mod}, \text{E2E}\}$ at each P^5 stage decision boundary; a coding agent then generates each corresponding code block a_i. (c) EAL learns family-conditioned coordination: SCB evaluates modular and end-to-end branches from the same restored state into $\mathcal{D}_{\text{SCB}}$, Q-learning yields $Q_\theta(x, c)$, and at test time a frozen coordinator selects best c_i to condition code generation and execution.

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.

1 INTRODUCTION

Performing general-purpose embodied manipulation in diverse and unstructured environments remains challenging in the absence of low-level executor adaptation. Recent progress toward this goal has followed two major paradigms, namely end-to-end policies and coding agents, each with distinct strengths and limitations. Within the end-to-end paradigm, task-specific visuomotor controllers excel at contact-rich control but typically lack native support for open-vocabulary rebinding (Zhao et al., 2023; Chi et al., 2023; Ze et al., 2024). Vision-language-action (VLA) and world-action models (WAMs) support language-conditioned execution (Brohan et al., 2023; Zitkovich et al., 2023; Kim et al., 2024; Black et al., 2025), yet become unreliable under compositional and state-distribution shifts. Coding agents, by contrast, enable Code-as-Policy (Liang et al., 2023; Fu et al., 2026; Huang et al., 2023a) by flexibly composing program modules for long-horizon tasks. Their physical competence, however, is bounded by the exposed primitives, limiting fine-grained, closed-loop correction during contact-rich interactions. These complementary advantages inspire hybrid frameworks in which coding agents generate modular API compositions or invoke frozen end-to-end policies, most commonly VLAs, as contact-rich primitives (Zhang et al., 2026d; Hu et al., 2026; Huang et al., 2026). Nevertheless, high-level coding agents lack inherent awareness of the divergent consequences of invoking different skill executors. They struggle to choose among an explicit modular composition, any available self-contained policy, and their possible switching patterns. For instance, modular execution can remain stable under task-distribution shift, whereas end-to-end execution excels in case of unexpected interruption. This raises a critical research question: *How can we endow coding agents with outcome-aware execution responsibility over embodied skills?*

We address this question with **RoboAware**, a framework that learns a responsibility boundary between modular composition and end-to-end execution from counterfactual outcomes. Here, a *policy family* $c \in \{\mathrm{Mod}, \mathrm{E2E}\}$ is the sole high-level coordination action; the *API library* comprises typed APIs together with their bound low-level backends; *responsibility* denotes which family should control the next semantically meaningful skill transition. The first challenge is to define where responsibility can be compared. The space of low-level code generation is too expansive and unstructured to provide a tractable basis for responsibility assignment. Inspired by the success of REPL-style programming (Liu et al., 2025), we propose the $\mathbf{P}^5$ **Skill Schema** to address this limitation. The P^5 schema organizes embodied skills into semantically meaningful stages from perception to execution in 3D space. We formulate a finite-horizon hierarchical MDP that selects a policy family at each P^5 stage. At the lower level, we curate an API library combining typed modular interfaces for composability and controllability with end-to-end policies for contact-rich local execution. Conditioned on the selected family and current context, the frozen coding agent π_{code} generates a complete local code block through the fixed API library. Learning is confined to the responsibility coordinator, while skill implementation leverages the coding agent's existing code-generation capability.

The second challenge is to expose the missing outcome of the skills that were not selected. Many approaches execute open-loop skill sequences or code programs without attending to outcomes (Liang et al., 2023; Singh et al., 2023; Huang et al., 2023a). Methods that obtain outcomes typically rely on VLM-based evaluations limited to the predicted outcome of the executed trajectory, and therefore inherit selection bias (Huang et al., 2026; Zhang et al., 2026a). We introduce **State-Locked Counterfactual Branching (SCB)**, which leverages simulator (1) save-restore mechanisms and (2) oracle outcome judgments to execute every admissible family from an identical physical state, producing a same-state set of potential outcomes at each P^5 decision point. To convert these set-wise outcomes into deployable family selection, we formulate **Execution-Aware Learning (EAL)**, in which Monte Carlo tree search (MCTS) treats restorable P^5 decision states as nodes and admissible policy families as edges, while Q-learning distills the explored outcomes into a family-conditioned value function Q_θ. The family-wise values encode a relative responsibility preference over modular composition and the end-to-end family, reflecting expected remaining skill- and task-level return under an observation. During deployment, the RoboAware coordinator assigns responsibility by maximizing ρ_θ induced by Q_θ, and a coding agent generates executable code blocks accordingly to transition from one P^5 state to the next, without state restoration or privileged simulator state. See Figure 1 for an overview of the proposed framework. Our contributions are as follows:

- We formulate RoboAware as a hierarchical MDP that selects a policy family c_i at each P^5 stage to produce code block a_i. We curate its lower-level API library around two complementary families, namely composable modular components and end-to-end policies for contact-rich local skills.

- We introduce SCB to acquire paired locked-state outcomes, and EAL to learn family-conditioned Q_θ through MCTS-guided exploration and Q-learning.
- We recover the coordinator from Q_θ and evaluate it on 100 tasks across three benchmarks, obtaining a SOTA 77.0% overall success rate. Ablations confirm the effectiveness of our designs.

2 METHODOLOGY

2.1 PROBLEM FORMULATION

We consider language-conditioned manipulation in a physics environment $\mathcal{E}$, where the observation at time t is $o_t = (I_t^{\text{rgb}}, I_t^{\text{d}}, q_t)$ from an agent-view RGB image, a metric depth map, and proprioception. We formulate the task as a finite-horizon MDP $\mathcal{M} = (\mathcal{X}, \mathcal{C}, \mathcal{P}, r, \gamma)$. Decision point i occurs at the start of a P^5 stage, where the observation $x_i = (\ell, o_i, h_i, z_i, g_i) \in \mathcal{X}$ is received. Here ℓ denotes the language instruction, h_i the observable execution history, z_i the current P^5 stage, and g_i the active skill schema. The high-level action is a policy-family choice $c_i \in \mathcal{C}_i(z_i, g_i) \subseteq \mathcal{C} = \{\text{Mod}, \text{E2E}\}$. A coding agent generates a complete executable code block $a_i \sim \pi_{\text{code}}(\cdot \mid x_i, c_i)$ with $a_i \in \mathcal{A}_{c_i}(x_i, z_i, g_i)$, where $\mathcal{A}_{\text{Mod}}$ and $\mathcal{A}_{\text{E2E}}$ are the modular- and end-to-end-family code-block spaces. Executing a_i in $\mathcal{E}$ until the next decision point induces the transition $\mathcal{P}(x_{i+1} \mid x_i, c_i) = \mathbb{E}_{a_i \sim \pi_{\text{code}}(\cdot \mid x_i, c_i)}[\mathcal{P}_{\mathcal{E}}(x_{i+1} \mid x_i, a_i)]$. A binary reward r is received at task termination. $\gamma = 1$. Only the coordinator's value model Q_θ is trained; the coding agent, API interfaces, backend bindings, and low-level backend parameters remain fixed.

2.2 ROBOAWARE FRAMEWORK

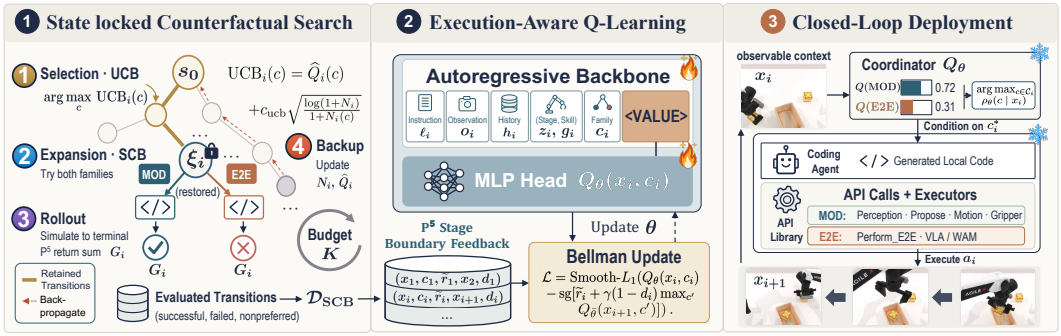


Figure 2: Method overview of EAL in RoboAware. (1) State-locked counterfactual search repeats Selection (UCB), Expansion (SCB paired Mod/E2E branches from a restored ξ_i), Rollout (suffix return G_i), and Backup within budget K, retaining successful, failed, and nonpreferred transitions in $\mathcal{D}_{\text{SCB}}$. (2) A VLM backbone with an MLP head maps context–family tokens ending in <VALUE> to $Q_\theta(x_i, c_i)$, trained by Bellman updates on skill-completion and terminal-task feedback. (3) At deployment, frozen Q_θ induces ρ_θ and selects $c_i^* = \arg\max_c \rho_\theta(c \mid x_i)$; a frozen coding agent generates a local code block against the fixed API library and executes it, looping on observable feedback at each P^5 boundary.

RoboAware formulates heterogeneous policy-family orchestration as stage-conditioned responsibility learning. The pipeline is visualized in Figure 2 and elaborated in Algorithm 1.

2.2.1 SKILL PREPARATION

P^5 Skill Schema. We instantiate local embodied skills through either interface-based modular composition or a self-contained end-to-end controller. The P^5 skill schema decomposes each skill into five stages: *perceive*, *propose*, *pre-manipulate*, *perform*, and *post-manipulate*. As shown in Figure 1, this schema provides a unified representation for arbitrary skills, including primitive skills (*e.g.* pick, place, push, pull) and compositional skills (*e.g.* handover). Given the task instruction ℓ and initial observation o_0, the coding agent constructs a task-level plan $\mathcal{G}$ comprising P^5 skill schemas. Responsibility is reconsidered at each P^5 stage.

Table 1: Offline screening of the selected VLA and WAM backends (successful / total). **Full**: complete task; **G–L**: grasp and lift.

Method	LIBERO-Pro		RoboSuite		RoboTwin 2.0	
	Full	G–L	Full	G–L	Full	G–L
$\pi_{0.5}$	**10/14**	**13/14**	0/4	**2/4**	0/10	2/10
FastWAM	4/14	11/14	0/4	0/4	**10/10**	**10/10**

Table 2: High-level embodied APIs under the P^5 skill schema; ✓ marks stages covered by each API.

API	*perc.*	*prop.*	*pre*	*perf.*	*post*
`Perceive_object_pose`	✓				
`Propose_eef_pose`		✓			
`Plan_to_eef_pose`			✓		✓
`Perform_gripper`				✓	
`Perform_E2E`	✓		✓	✓	✓

API Library Construction. Building on the P^5 stages, we expose an API library that supports the modular and end-to-end policy families. The Mod family uses an explicit intermediate interface for composability and controllability, whereas the E2E family relies on an implicit internal loop for local contact competence. We assemble the library by offline screening of open-source modular perception, grasping, and motion components alongside released VLA and world-action-model (WAM) policies. For the Mod family, screening retains **InstructSAM** (Yuan et al., 2026b), our augmented **NeuGraspNet+** from NeuGraspNet (Jauhri et al., 2024)—which expands sparse geometric candidates and selects Top-1 only among planner-feasible grasps—and **cuRoboV2** (Sundaralingam et al., 2026) as the core backends for perception, pose proposal, and motion planning, respectively. Together with the deterministic gripper API, these components define the modular calls available to π_{code}. For the E2E family, offline evaluation reveals limited zero-shot performance on complete tasks, yet these policies perform well on local skills such as grasping and lifting (see Section 3.2). We therefore position VLA and WAM policies as local executors at grasp–lift granularity rather than as end-to-end policies for complete tasks. Effectiveness and efficiency screening selects $\pi_{0.5}$ (Physical Intelligence et al., 2025) and FastWAM (Yuan et al., 2026a) for integration into `Perform_E2E` (see Table 1). Screening details appear in Appendix B. Each stage admits both families except the *propose* stage (see Table 2). `Perform_E2E` spans the *perceive*, *pre-manipulate*, *perform*, and *post-manipulate* stages. A call remains atomic with respect to family selection until it meets its registered postcondition, reports failure, or exhausts its step budget. The API library, including its typed interfaces, backend bindings, and checkpoints, remains frozen during training and evaluation.

2.2.2 Execution-Aware Learning

Existing coding agents cannot reliably anticipate the physical consequences of executing a generated skill. To address this limitation, we propose Execution-Aware Learning (EAL) as illustrated in Figure 2, which learns family-conditioned outcome values from controlled simulator experience.

State-Locked Counterfactual Branching (SCB) is the training-time operator that makes the otherwise missing policy-family outcomes observable. On a fixed set of simulator training tasks, EAL builds a Monte Carlo search tree from each sampled P^5 stage-entry context. Nodes are restorable decision states, and outgoing edges correspond to admissible policy families. Each node stores a snapshot $\xi_i = (x_i, m_i, p_i, r)$ containing the simulator and execution state required for exact restoration. Beyond the deployment-visible context x_i, it provides oracle masks m_i, ground-truth object poses p_i, and access to the terminal reward evaluator r. These privileged signals support the training rewards including pose-based assessment under threshold ϵ_p. The search repeats four search phases within a budget of K completed suffix simulations per root. See Appendix A.2 for details.

Selection. Starting from the root, the search follows stored tree paths by selecting family edges with UCB at already expanded nodes, where N_i and $N_i(c)$ are node and family-edge visit counts, and $\widehat{Q}_i(c)$ is the empirical return estimate:

$$c_i^{\text{tree}} = \arg\max_{c \in \mathcal{C}_i} \left[\widehat{Q}_i(c) + c_{\text{ucb}} \sqrt{\frac{\log(1+N_i)}{1+N_i(c)}} \right]. \tag{1}$$

Expansion. At a new node, SCB evaluates every admissible family at least once from the same physical state and observable-history prefix. For each $c \in \mathcal{C}_i$, it restores ξ_i, samples $a_i^c \sim \pi_{\text{code}}(\cdot \mid x_i, c)$, and executes the block to the next decision point or termination. The resulting transition contains a successor snapshot ξ_{i+1}^c, terminal indicator d_i^c, and training reward $\tilde{r}_i^c$. Reevaluating an existing family edge repeats restoration, code generation, and execution with a fresh code sample.

Rollout. From each nonterminal successor, the codes are generated using API library to complete the suffix. The suffix ends at task termination, unrecoverable failure, or the episode horizon T. Rollout-only states are not added as expanded tree nodes during this simulation. Each completed suffix yields $G_i = \sum_{j=i}^{T} \gamma^{j-i} \tilde{r}_j$. $\tilde{r}_i^c$ is defined in Equation (3).

Backpropagation. This tree-statistics backup propagates the suffix return G_i along the selected tree path, including each ancestor's recorded prefix rewards. Each visited node and family-edge count is incremented, and $\widehat{Q}$ is updated by empirical averaging. These statistics guide subsequent selection; the parameters of Q_θ are optimized only after data collection. Within budget $K = 20$, all evaluated tree edges, including failed and nonpreferred branches, contribute transitions to the training dataset:

$$\mathcal{D}_{\mathrm{SCB}} = \bigcup_i \left\{ (x_i, c, \tilde{r}_i^c, x_{i+1}^c, d_i^c, \xi_{i+1}^c) : c \in \mathcal{C}_i \right\}. \tag{2}$$

Execution-Aware Q-learning. A task comprises several P^5 skill blocks. At the end of each P^5 stage, oracle signals in ξ_{i+1}^c yield a perception score $r_{\mathrm{p},i}^c \in [0,1]$ from agreement with mask m_i, and a manipulation score $r_{\mathrm{m},i}^c \in [0,1]$ from object-pose error against p_i under threshold ϵ_p. The binary task-success reward $r \in \{0,1\}$ is received only when $d_i^c = 1$. The transition reward used in both tree search and Q-learning is the linear combination with non-negative weights:

$$\tilde{r}_i^c = w_{\mathrm{p}} r_{\mathrm{p},i}^c + w_{\mathrm{m}} r_{\mathrm{m},i}^c + w_{\mathrm{t}} d_i^c r, \quad \text{where } w_{\mathrm{p}} + w_{\mathrm{m}} + w_{\mathrm{t}} = 1. \tag{3}$$

The learning target is $Q_\theta(x_i, c) \approx \max_\pi \mathbb{E}_\pi \left[\sum_{j=i}^{T} \gamma^{j-i} \tilde{r}_j \mid x_i, c_i = c \right]$, where π governs subsequent family choices under the fixed coding agent and API library. The value $Q_\theta(x_i, c)$ estimates the remaining skill- and task-level return conditional on choosing family c next. After collecting $\mathcal{D}_{\mathrm{SCB}}$ across the training tasks, we optimize Q_θ on this fixed transition dataset using Bellman targets rather than directly regressing the rollout returns G_i or tree estimates $\widehat{Q}_i(c)$. All evaluated tree edges, including failed and nonpreferred branches, contribute to this update. For transitions sampled from $\mathcal{D}_{\mathrm{SCB}}$, we optimize θ by gradient-based backpropagation through Q_θ, minimizing the following Smooth-L_1 Bellman loss:

$$\mathcal{L}_{\mathrm{EAL}}(\theta) = \mathbb{E}_{(x_i,c)\sim\mathcal{D}_{\mathrm{SCB}}} \ell \left[Q_\theta(x_i, c) - \left(\tilde{r}_i^c + \gamma(1 - d_i^c) \max_{c' \in \mathcal{C}_{i+1}} Q_{\bar{\theta}}(x_{i+1}^c, c') \right) \right], \tag{4}$$

where $Q_{\bar{\theta}}$ is a stop-gradient target copy and $\ell(\epsilon)$ refers to the Smooth-L_1 loss, which maintains stable gradients in the regime of large errors as the L_1 loss does and facilitates smooth optimization for small errors analogous to the L_2 loss:

$$\ell(\epsilon) = \begin{cases} \frac{1}{2}\epsilon^2 & \text{if } |\epsilon| < 1, \\ |\epsilon| - \frac{1}{2} & \text{otherwise.} \end{cases} \tag{5}$$

Q-Modeling. $Q_\theta(x_i, c)$ is parameterized with a lightweight VLM backbone (*e.g.* Qwen3.5-0.8B). The state $x_i = (\ell, o_i, h_i, z_i, g_i)$ and family c form a multimodal token sequence ending in a <VALUE> token. An MLP head maps its final-layer hidden state to the scalar Q-value, optimized with the Bellman objective in Equation (4).

2.2.3 Test-Time Skill Coordination

During training, repeated access to ξ_i and oracle outcome judgments supervises values, whereas neither is included at test time. The learned values induce a relative responsibility preference over admissible families:

$$\rho_\theta(c \mid x_i) = \frac{\exp(Q_\theta(x_i, c))}{\sum_{c' \in \mathcal{C}_i} \exp(Q_\theta(x_i, c'))}. \tag{6}$$

The RoboAware coordinator selects $c_i^* = \arg\max_{c \in \mathcal{C}_i} \rho_\theta(c \mid x_i)$ (equivalently $\arg\max_c Q_\theta(x_i, c)$; see Appendix A.3), and the coding agent generates $a_i \sim \pi_{\mathrm{code}}(\cdot \mid x_i, c_i^*)$. The coding agent executes the selected block until the next semantic boundary or task termination, then updates (h_i, z_i, g_i) from observable feedback. Family selection repeats at each subsequent boundary with Q_θ fixed. Both family selection and local code generation respond to updated observations without parameter updates or changes to executor interfaces.

3 EXPERIMENTS

3.1 EXPERIMENTAL PROTOCOL

Benchmarks. Our evaluation contains 100 tabletop-manipulation tasks across three benchmarks: LIBERO-Pro (Zhou et al., 2025), RoboSuite (Zhu et al., 2020), and RoboTwin 2.0 (Chen et al., 2026d). We use scene seed 0 only to collect privileged SCB branches and train Q_θ. The learned coordinator is then frozen and evaluated on seeds 1–10, giving 1,000 test episodes in total.

Baselines. We compare RoboAware with the following baselines:

- *End-to-end policies*: π_0 (Black et al., 2025), $\pi_{0.5}$ (Physical Intelligence et al., 2025), FastWAM (Yuan et al., 2026a), FasterWAM (Zhao et al., 2026), as standalone controllers;
- *Code-as-policy*: CaP (Liang et al., 2023), CaP-X (Fu et al., 2026), RATS (Zhang et al., 2026b), and ASPIRE (Lu et al., 2026), without learned family coordination.
- *VLA with harness*: Pigey (Galanti et al., 2026), VLS (Liu et al., 2026a), and Harness-VLA (Zhang et al., 2026d), which invoke a frozen policy.

3.2 BENCHMARK PERFORMANCE

Table 3: Aggregate success rates (%) on LIBERO-Pro (instruction-redirection **T** / position-swap **S**), RoboSuite, and RoboTwin. RoboSuite and RoboTwin use their full task pools under seeds 1–10. Overall is the mean over available columns. On RoboTwin, π_0 and $\pi_{0.5}$ are incompatible with RoboTwin's dual-arm setup, so results involving these models are marked "/" because of the embodiment mismatch. For fair comparison, global cross-episode memory is disabled for baselines that rely on memory. All seeds are tested in **one trial**.

Method	LIBERO-Pro								RoboSuite	RoboTwin	Overall
	Spat-T	Spat-S	Obj-T	Obj-S	Goal-T	Goal-S	L10-T	L10-S			
End-to-end policies											
π_0 (Black et al., 2025)	0.0	0.0	0.0	2.0	0.0	0.0	0.0	0.0	0.0	/	0.2
$\pi_{0.5}$ (Physical Intelligence et al., 2025)	1.0	20.0	1.0	17.0	2.0	38.0	1.0	8.0	0.0	/	9.8
FastWAM (Yuan et al., 2026a)	30.0	0.0	10.0	0.0	10.0	0.0	10.0	0.0	0.0	81.9	14.2
FasterWAM (Zhao et al., 2026)	50.0	0.0	10.0	10.0	10.0	10.0	10.0	0.0	0.0	85.6	18.6
Code as policy											
CaP (Liang et al., 2023)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	70.0	5.6	7.6
CaP-X (Fu et al., 2026)	0.0	0.0	10.0	0.0	0.0	0.0	0.0	0.0	50.0	6.9	6.7
RATS (Zhang et al., 2026b)	2.0	18.0	10.0	8.0	8.0	8.0	0.0	0.0	45.0	0.0	9.9
ASPIRE (Lu et al., 2026)	60.0	51.0	**95.0**	**98.0**	45.0	81.0	38.0	22.0	80.0	0.0	57.0
VLA with harness											
Pigey (Galanti et al., 2026)	**80.0**	66.0	54.0	54.0	22.0	44.0	28.0	12.0	15.0	/	41.7
VLS (Liu et al., 2026a)	54.0	42.0	41.0	45.0	33.0	38.0	25.0	15.0	0.0	/	32.6
Harness-VLA (Zhang et al., 2026d)	50.0	**70.0**	50.0	90.0	40.0	60.0	40.0	30.0	20.0	/	50.0
RoboAware (ours)	68.0	68.0	92.0	86.0	**66.0**	**84.0**	**64.0**	**62.0**	**90.0**	**90.0**	**77.0**

End-to-end policies struggle with whole-task distribution shifts. For both VLAs and WAMs, unfamiliar skill combinations, scene types, and perturbations may deviate substantially from their training distributions, leading to low zero-shot SRs in Table 3. In comparison, RoboAware generates composable modular alternatives under such conditions while retaining end-to-end policies for local skills suited to their capabilities.

Code-as-policy methods are constrained by contact precision and accumulated experience. Code-as-policy methods offer stronger long-term planning and composability for long-horizon bimanual tasks such as those in RoboTwin. However, within-skill contact-rich performance depends on available primitives: CaP and CaP-X generate detailed code at test time, yet may lack the precision needed for tasks such as placing a mug inside a microwave in LIBERO-Pro (see Figure 3). Single-turn evaluation without cross-episode adaptation or multi-turn memory further limits the benefits of accumulated experience for memory-based methods like CaP-X, RATS, and ASPIRE. RoboAware addresses these limitations by combining modular composition with local end-to-end control and learning state-dependent family responsibility from training-time outcomes.

VLA harnesses remain limited by outcome anticipation. We observe that VLS (Liu et al., 2026a) can be misled by early-stage behavior in long-horizon tasks, where initial progress may not reflect readiness for later interactions. Harness-VLA is closely related to our work and achieves competitive performance. However, once the selected family changes the physical state, the alternative family's outcome at the original state remains unobserved, so family switching relies on experiential judgments. RoboAware addresses this gap through SCB and EAL, which expose same-state alternatives for family selection.

3.3 Experimental Analysis

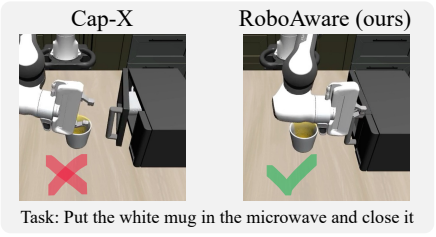


Figure 3: Comparison between CaP-X and RoboAware on a contact-rich LIBERO-Pro task "Put the white mug in the microwave and close it."

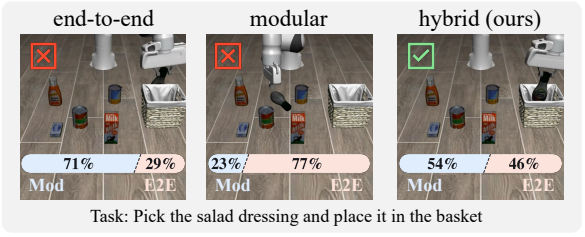


Figure 4: A representative LIBERO-Pro task comparison and visualization of the responsibility preference distribution ρ_θ for RoboAware, RoboAware-EO, and RoboAware-MO.

Takeaway 1: Neither policy family alone provides broad high-success task coverage. We restrict code generation to modular composition (RoboAware-MO) or end-to-end blocks (RoboAware-EO), keeping the same API library and tasks as RoboAware. Success rate (SR) reductions in group A of Table 4 show that both families must be coordinated by state. In a LIBERO-Pro task illustrated in Figure 4, RoboAware-EO and RoboAware-MO fail at different stages, whereas RoboAware succeeds by switching policy families across P^5 boundaries. The ρ_θ visualization assigns lower preference to the families used by the ablation variants at the illustrated states, consistent with their failures.

Table 4: Ablation success rates (%) on the three benchmarks, grouped by conclusions A–C in this analysis. RoboAware is the common reference. LIBERO-Pro aggregates its eight T/S evaluation splits. Abbreviations: MO = modular only; EO = end-to-end only; UO = unpaired outcomes; TF = training free; FH = fixed handoff; ES = episode selection.

Variants	LIBERO-Pro	RoboSuite	RoboTwin 2.0
A. Policy Family			
RoboAware-MO	54.8	67.5	41.9
RoboAware-EO	49.5	60.0	45.6
B. Counterfactual Outcome			
RoboAware-UO	58.5	75.0	53.8
RoboAware-TF	51.0	60.0	40.0
C. Coordination Mode			
RoboAware-FH	64.5	82.5	55.6
RoboAware-ES	59.3	70.0	51.9
RoboAware	**73.8**	**90.0**	**90.0**

Table 5: Group B ablations of coordinator training and same-state outcomes. ✓: present; ✗: absent.

Variant	Training	Counterfactual
ours	✓	✓
ours-UO	✓	✗
ours-TF	✗	✗

Table 6: Group C ablations of when family responsibility may change. When: decision points; Family: modular / end-to-end.

Variant	When	Family
ours	Each stage	Mod / E2E
ours-FH	Fixed handoff	Mod + E2E
ours-ES	Episode start	Mod / E2E

Takeaway 2: Paired same-state outcomes improve Execution-aware learning. To evaluate the importance of SCB, we ablate coordinator training and counterfactual supervision (Table 5): RoboAware uses both, RoboAware-UO trains Q_θ only on executed trajectories without state-locked branching, and RoboAware-TF skips coordinator training so the coding agent chooses directly from the hybrid API library. Higher SR for RoboAware-UO than RoboAware-TF in group B of Table 4

indicates that the coding agent lacks inherent execution awareness and that coordinator training still helps without counterfactuals. Yet executed-only branches inherit the agent's experience distribution and introduce selection bias. A modest SR decrease for RoboAware-UO relative to RoboAware therefore supports the necessity of paired counterfactual executions from the same ξ_i.

Takeaway 3: RoboAware benefits from fine-grained coordination at P^5 stages. Following the same training protocol to obtain Q_θ, we compare three coordination strategies that differ in when family responsibility may change. RoboAware reselects a policy family at every P^5 stage. RoboAware-FH (fixed handoff) follows a fixed protocol: modular APIs for *perceive* and *pre-manipulate*, then an E2E call for subsequent execution. RoboAware-ES (episode selection) chooses one family at episode start and retains it for the entire task. Lower SRs for both variants in group C of Table 4 suggests that neither a fixed handoff nor a single episode-level decision tracking changing execution responsibility as well as per-stage coordination (Table 6).

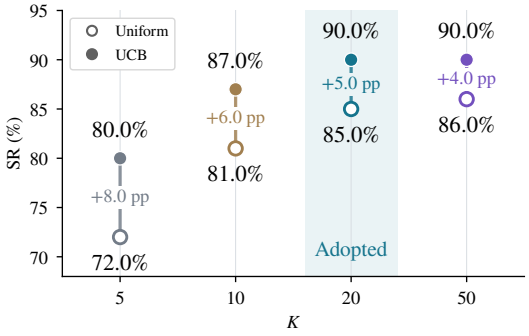


Figure 5: Average RoboSuite success rates (%) across budgets $K \in \{5, 10, 20, 50\}$. Hollow & filled circles denote RoboAware-UA & RoboAware. Shading marks adopted K=20.

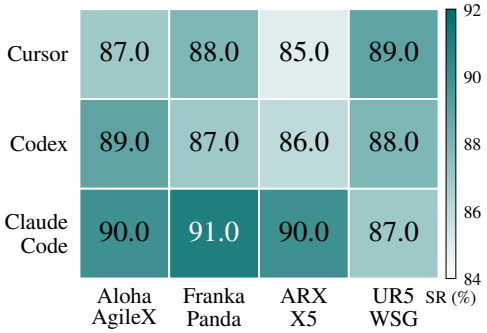


Figure 6: RoboTwin 2.0 success rates (%) on a four-task subset across coding-agent backends (rows) and embodiments (columns). Color bar shows the SR range.

Takeaway 4: UCB-guided exploration uses a limited branching budget more effectively. To test how a finite simulation budget is allocated during branching, we compare RoboAware with RoboAware-UA, which allocates evaluations uniformly over admissible families instead of the UCB rule in Equation (1). Figure 5 reports both under $K \in \{5, 10, 20, 50\}$ in RoboSuite. SR gains diminish with K, so we use the elbow K=20. Under matched budgets, RoboAware exceeds RoboAware-UA, indicating that UCB concentrates evaluations on promising families more effectively than uniform allocation.

Takeaway 5: RoboAware is less sensitive to coding-agent and embodiment variations. To assess sensitivity to changes in the coding agent and robot embodiment, we evaluate RoboAware with three coding-agent backends (Claude Code as default, Cursor, and Codex) across four embodiments (Aloha-AgileX as default, Franka-Panda, ARX-X5, and UR5-WSG) on four tasks in RoboTwin 2.0 (See Appendix Appendix C.1 for details), retaining the same API interfaces. Figure 6 shows a small average SR gap across coding-agent backends and embodiments. We attribute the small backend gap to P^5, which limits each generation to one code block $a_i \in \mathcal{A}_{c_i}(x_i, z_i, g_i)$. Notably, the cuRoboV2 backend of `Plan_to_eef_pose` inherently supports different robots through the same API interface across embodiments.

4 Related Work

4.1 Embodied Coding Agents

Embodied coding agents have progressed along three lines. Early language planners translated goals into sequences over closed skill APIs and improved grounding through semantic matching, affordance scoring, program-structured prompting, and execution feedback (Song et al., 2023; Ahn et al., 2023; Singh et al., 2023; Huang et al., 2023b). Executable-policy methods such as CaP (Liang et al., 2023), VoxPoser (Huang et al., 2023a), RoboCodeX (Mu et al., 2024), and CodeDiffuser (Yin

et al., 2025) then generated Python controllers or geometric intermediates, gaining compositionality, spatial reasoning, and interpretability while remaining bounded by designer-provided abstractions and API coverage. Recent systems make the program itself adaptive. CaP-X (Fu et al., 2026) and AOR (Kumar, 2026) repair code from multimodal rollouts, RHO (Elmaaroufi et al., 2026) and GaP (Chen et al., 2026a) optimize repository- or graph-structured policies through execution, VLCP (Naouali et al., 2026) rewrites failed controllers within an episode, PhysCaP (Lin et al., 2026a) actively probes latent physical properties, and ASPIRE (Lu et al., 2026) compounds validated repairs into a reusable skill library. This line strengthens modular program execution, yet its evidence is conditional on generating a modular code block. Neither success nor code-only repair reveals whether an end-to-end block performs better at the same state. We instead use SCB to generate and execute modular and end-to-end coding blocks from the same restored state, and EAL learns their family-conditioned values from paired outcomes.

4.2 End-to-End Policy Orchestration and Execution Harnesses

End-to-end policy orchestration has likewise progressed along three lines: generalist VLAs including RT-1 (Brohan et al., 2023), RT-2 (Zitkovich et al., 2023), Octo (Ghosh et al., 2024), OpenVLA (Kim et al., 2024), and π_0 (Black et al., 2025) broaden local language-conditioned control, while hierarchical agents move planning, memory, grounding, and progress tracking to a VLM or code layer, exposing VLAs as bounded tools. Recent variants add tool-aligned post-training, persistent semantic state, tractability-aware interfaces, interruptible physical orchestration, and modular monitor, planner, and controller wrappers (Hu et al., 2026; Lei et al., 2026; Galanti et al., 2026; Zhang et al., 2026d; Vo et al., 2026; Peng et al., 2026; Chen et al., 2026c; Yoon et al., 2026). VLS (Liu et al., 2026a) instead steers the sampling of a frozen generative policy with VLM-synthesized rewards. Hybrid and heterogeneous-policy harnesses including RoboRouter (Chen et al., 2026e), RoboHarness (Huang et al., 2026), and RouterVLA (Ren et al., 2026) then infer applicability from execution traces, semantic retrieval, capability memory, outcome-disjoint probes, or transition structure. Related methods including Reset (Dai et al., 2025), SwitchVLA (Li et al., 2025), SemHandoff (Rui et al., 2026), and BICPO-VLA (Shang et al., 2026) prepare out-of-distribution initial states, condition switching on execution phase, diagnose semantic handoffs, or reduce asynchronous takeover mismatch. Finally, VLM oracles, learned rewards, frozen-feature success probes, counterfactual relabeling, and self-evolving critics detect, rank, or repair attempts (Qi et al., 2026; Saurabh et al., 2026; Chen et al., 2026b; Wu et al., 2026; Kwok et al., 2025; Li et al., 2026a; Glossop et al., 2026; Zhang et al., 2026a; Kondoh et al., 2026; Ding et al., 2026). These methods estimate absolute success, compare within-policy alternatives, or infer cross-policy suitability from retrospective traces and probes. Our distinction is to use simulator-restored, same-state executions to obtain paired outcomes for modular and end-to-end families within a common skill-stage interface.

5 Conclusion and Limitations

We presented RoboAware, which learns state-dependent responsibility between modular composition and end-to-end control while keeping the coding agent and API library fixed. The P^5 skill schema defines the semantic boundaries at which responsibility is assigned, and SCB exposes both family outcomes at such a boundary by executing paired code blocks from identical restored states. From this paired experience, EAL combines UCB-guided MCTS with Q-learning into family-conditioned estimates of remaining skill- and task-level return. At deployment, these estimates select a family before code generation from observable context alone, and responsibility is reconsidered at each subsequent boundary without simulator restoration, privileged inputs, or online search. Across 100 tasks in LIBERO-Pro, RoboSuite, and RoboTwin 2.0, this makes anticipated physical outcomes an explicit basis for composing complementary frozen skills.

We outline two directions for future work. First, RoboAware currently integrates only modular composition and end-to-end policies. Extending it to generalizable reinforcement learning policies, retrieval-based policies, and policies learned from demonstrations may require new counterfactual outcome data and further coordinator training. Second, responsibility also changes hands only at P^5 boundaries, so a single atomic end-to-end invocation may span several stages and leave transient failures to its internal controller until a postcondition, failure signal, or timeout returns control. Finer-grained interruption under ambiguous observations therefore remains an open problem.

AI Assistance Statement

In this work, we used generative AI tools to draft sections of the paper, to aid or polish writing for readability, and for retrieval and discovery of related work. Separately, LLM-based coding-agent backends appear as experimental subjects in our evaluation protocol and are described in the paper; that use is part of the studied system rather than undisclosed author assistance. We have reviewed all AI-assisted text and literature suggestions: authors verified technical claims, notation, equations, and experimental descriptions against the methods and results, and checked suggested related work against the cited sources. The authors take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

References

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, et al. Do as I can, not as I say: Grounding language in robotic affordances. In *Proceedings of The 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pp. 287–318, 2023.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Robert Equi, et al. π_0: A vision-language-action flow model for general robot control. In *Robotics: Science and Systems (RSS)*, 2025.

Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Joseph Dabis, et al. RT-1: Robotics transformer for real-world control at scale. In *Robotics: Science and Systems (RSS)*, 2023.

Jun Cen, Siteng Huang, Yuqian Yuan, Kehan Li, Hangjie Yuan, Chaohui Yu, et al. RynnVLA-002: A Unified Vision-Language-Action and World Model. *arXiv preprint arXiv:2511.17502*, 2025.

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 Fan, and Ken Goldberg. GaP: A graph-as-policy multi-agent self-learning harness for variational automation tasks. *arXiv preprint arXiv:2607.05369*, 2026a.

Shirui Chen, Cole Harrison, Ying-Chun Lee, Angela Jin Yang, Zhongzheng Ren, et al. TOPReward: Token probabilities as hidden zero-shot rewards for robotics. *arXiv preprint arXiv:2602.19313*, 2026b.

Siang Chen, Pengwei Xie, Wei Tang, Dingchang Hu, Yixiang Dai, and Guijin Wang. Region-aware grasp framework with normalized grasp space for efficient 6-DoF grasping. In *Proceedings of The 8th Conference on Robot Learning*, volume 270 of *Proceedings of Machine Learning Research*, pp. 1834–1850, 2025.

Siyi Chen, Hugo Hadfield, Alex Zook, Mikaela Angelina Uy, Chan Hee Song, Erwin Coumans, Xuning Yang, Faisal Ladhak, Qing Qu, Stan Birchfield, Jonathan Tremblay, and Valts Blukis. VoLo: A physical orchestrator for open-vocabulary long-horizon manipulation. *arXiv preprint arXiv:2606.07723*, 2026c.

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. In *International Conference on Machine Learning (ICML)*, 2026d.

Yiteng Chen, Zhe Cao, Hongjia Ren, Chenjie Yang, Wenbo Li, et al. RoboRouter: Training-free policy routing for robotic manipulation. *arXiv preprint arXiv:2603.07892*, 2026e.

Cheng Chi, Siyuan Feng, Yilun Du, Zhenjia Xu, Eric Cousineau, Benjamin Burchfiel, and Shuran Song. Diffusion policy: Visuomotor policy learning via action diffusion. In *Robotics: Science and Systems (RSS)*, 2023.

Yinlong Dai, Andre Keyser, and Dylan P. Losey. Prepare before you act: Learning from humans to rearrange initial states. *arXiv preprint arXiv:2509.18043*, 2025.

Xin Ding, Liang Mi, Mingzhe Huang, Zixuan Wang, Chao Zhang, Zixu Hao, Fu Chen, Xiangyu Li, Yikai Zheng, Yaoyu Guo, Weijun Wang, Kun Li, Hao Wu, Yunxin Liu, and Ting Cao. Zetta ζ: An efficient closed-loop embodied harness for self-evolving physical intelligence. *arXiv preprint arXiv:2608.16590*, 2026.

Karim Elmaaroufi, Justin Svegliato, Sarunas Kalade, Graham Schelle, Sanjit A. Seshia, and Matei Zaharia. RHO: Your coding agent is secretly a roboticist. *arXiv preprint arXiv:2606.16458*, 2026.

Hao-Shu Fang, Chenxi Wang, Hongjie Fang, Minghao Gou, Jirong Liu, Hengxu Yan, et al. AnyGrasp: Robust and efficient grasp perception in spatial and temporal domains. *IEEE Transactions on Robotics*, 39(5):3929–3945, 2023. doi: 10.1109/TRO.2023.3281153.

Haoquan Fang, Jiafei Duan, Donovan Clay, Sam Wang, Shuo Liu, Weikai Huang, et al. MolmoAct2: Action Reasoning Models for Real-world Deployment. *arXiv preprint arXiv:2605.02881*, 2026.

Adam Fishman, Aaron Walsman, Mohak Bhardwaj, Wentao Yuan, Balakumar Sundaralingam, Byron Boots, and Dieter Fox. Avoid everything: Model-free collision avoidance with expert-guided fine-tuning. In *Proceedings of The 8th Conference on Robot Learning*, volume 270 of *Proceedings of Machine Learning Research*, pp. 1925–1948, 2025.

Letian Fu, Justin Yu, Karim El-Refai, Ethan Kou, Haoru Xue, et al. CaP-X: A framework for benchmarking and improving coding agents for robot manipulation. *arXiv preprint arXiv:2603.22435*, 2026.

Liane Galanti, Dhruv Shah, and Tri Dao. Addressing the orchestration gap in generalist robots via physical agency. *arXiv preprint arXiv:2607.21725*, 2026.

Dibya Ghosh, Homer Rich Walke, Karl Pertsch, Kevin Black, Oier Mees, et al. Octo: An open-source generalist robot policy. In *Robotics: Science and Systems (RSS)*, 2024.

Catherine Glossop, William Chen, Arjun Bhorkar, Dhruv Shah, and Sergey Levine. CAST: Counterfactual labels improve instruction following in vision-language-action models. *IEEE Robotics and Automation Letters*, 11(10):11785–11792, 2026. doi: 10.1109/LRA.2026.3726383.

Jiaheng Hu, Mohit Shridhar, Caden Lu, Dhruv Shah, Hao-Tien Lewis Chiang, Jie Tan, and Annie Xie. What matters in orchestrating robot policies: A systematic study of hierarchical VLA agents. *arXiv preprint arXiv:2606.10267*, 2026.

Jinbang Huang, Yuanzhao Hu, Zhiyuan Li, Ran Qi, Yixin Xiao, et al. RoboHarness: Memory-driven orchestration of heterogeneous robot policies for long-horizon planning. *arXiv preprint arXiv:2607.18060*, 2026.

Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Fei-Fei Li. VoxPoser: Composable 3D value maps for robotic manipulation with language models. In *Conference on Robot Learning (CoRL)*, 2023a.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, et al. Inner monologue: Embodied reasoning through planning with language models. In *Conference on Robot Learning (CoRL)*, 2023b.

Snehal Jauhri, Ishikaa Lunawat, and Georgia Chalvatzaki. Learning any-view 6DoF robotic grasping in cluttered scenes via neural surface rendering. In *Robotics: Science and Systems (RSS)*, 2024.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, et al. OpenVLA: An open-source vision-language-action model. In *Conference on Robot Learning (CoRL)*, 2024.

Moo Jin Kim, Chelsea Finn, and Percy Liang. Fine-tuning vision-language-action models: Optimizing speed and success. In *Robotics: Science and Systems (RSS)*, 2025. doi: 10.15607/RSS.2025.XXI.017.

Haru Kondoh, Kei Ota, Asako Kanezaki, and Yueh-Hua Wu. CounterAlign: Counterfactual supervision for vision-language-action models. *arXiv preprint arXiv:2608.21740*, 2026.

Vaishak Kumar. Act-observe-rewrite: Multimodal coding agents as in-context policy learners for robot manipulation. *arXiv preprint arXiv:2603.04466*, 2026.

Jacky Kwok, Christopher Agia, Rohan Sinha, Matt Foutter, Shulu Li, Ion Stoica, Azalia Mirhoseini, and Marco Pavone. RoboMonkey: Scaling test-time sampling and verification for vision-language-action models. In *Proceedings of The 9th Conference on Robot Learning*, volume 305 of *Proceedings of Machine Learning Research*, pp. 3200–3217, 2025.

Zixing Lei, Changxing Liu, Yichen Xiong, Minhao Xiong, Yuanzhuo Ding, Zhipeng Zhang, Weixin Li, and Siheng Chen. Towards long-horizon embodied agents with tool-aligned vision-language-action models. *arXiv preprint arXiv:2605.13119*, 2026.

Dayou Li, Jiuzhou Lei, Hao Wang, Lulin Liu, Yunhao Yang, et al. Learning actionable manipulation recovery via counterfactual failure synthesis. *arXiv preprint arXiv:2603.13528*, 2026a.

Lin Li, Qihang Zhang, Yiming Luo, Shuai Yang, Ruilin Wang, Fei Han, et al. Causal world modeling for robot control. In *Robotics: Science and Systems (RSS)*, 2026b. doi: 10.15607/RSS.2026.XXII.016.

Meng Li, Zhen Zhao, Zhengping Che, Fei Liao, Kun Wu, et al. SwitchVLA: Execution-aware task switching for vision-language-action models. *arXiv preprint arXiv:2506.03574*, 2025.

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 *IEEE International Conference on Robotics and Automation (ICRA)*, 2023.

Chen-Yu Lin, Jing-Wen Chen, Hsueh-En Chang, Hung-An Chen, Sheng-Hsun Chang, Chi-Pin Huang, Fu-En Yang, Min-Hung Chen, Yi-Ting Chen, Yu-Chiang Frank Wang, and Shao-Hua Sun. PhysCaP: Grounding code-as-policy agent with physics-informed exploration. *arXiv preprint arXiv:2608.21031*, 2026a.

Tao Lin, Yilei Zhong, Yuxin Du, Jingjing Zhang, Jiting Liu, Yinxinyu Chen, et al. Evo-1: Lightweight vision-language-action model with preserved semantic alignment. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 13397–13406, 2026b.

Anthony Z. Liu, Xinhe Wang, Jacob Sansom, Yao Fu, Jongwook Choi, Sungryull Sohn, Jaekyeom Kim, and Honglak Lee. Interactive and expressive code-augmented planning with large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 20330–20354, 2025. doi: 10.18653/v1/2025.acl-long.994.

Shuo Liu, Ishneet Sukhvinder Singh, Yiqing Xu, Jiafei Duan, and Ranjay Krishna. VLS: Steering pretrained robot policies via vision-language models. *arXiv preprint arXiv:2602.03973*, 2026a.

Songming Liu, Bangguo Li, Kai Ma, Lingxuan Wu, Hengkai Tan, Xiao Ouyang, et al. RDT2: Exploring the Scaling Limit of UMI Data Towards Zero-Shot Cross-Embodiment Generalization. *arXiv preprint arXiv:2602.03310*, 2026b.

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. *arXiv preprint arXiv:2607.00272*, 2026.

Haoxiang Ma, Junhao Cai, Xiaoxu Xu, Hao Li, Yuyin Yang, Yang Tian, et al. InternVLA-A1.5: Unifying Understanding, Latent Foresight, and Action for Compositional Generalization. *arXiv preprint arXiv:2607.04988*, 2026.

Yao Mu, Junting Chen, Qing-Long Zhang, Shoufa Chen, Qiaojun Yu, Chongjian Ge, Runjian Chen, Zhixuan Liang, Mengkang Hu, Chaofan Tao, Peize Sun, Haibao Yu, Chao Yang, Wenqi Shao, Wenhai Wang, Jifeng Dai, Yu Qiao, Mingyu Ding, and Ping Luo. RoboCodeX: Multimodal code generation for robotic behavior synthesis. In *International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pp. 36434–36454, 2024.

Adithyavairavan Murali, Balakumar Sundaralingam, Yu-Wei Chao, Wentao Yuan, Jun Yamada, Mark Carlson, et al. GraspGen: A diffusion-based framework for 6-DOF grasping with on-generator training. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2026.

Dhia Naouali, Minghan Wu, Claudia Wong, Abhinav Puthran, and Omar G. Younis. VLCP: Vision language control policy closed-loop code replanning for robot manipulation. *arXiv preprint arXiv:2608.16978*, 2026.

Soroush Nasiriany, Abhiram Maddukuri, Lance Zhang, Adeet Parikh, Aaron Lo, et al. RoboCasa: Large-scale simulation of household tasks for generalist robots. In *Robotics: Science and Systems (RSS)*, 2024.

Jiaqi Peng, Xiqian Yu, Delin Feng, Yuqiang Yang, Wenzhe Cai, et al. Cortex: A bidirectionally aligned embodied agent framework for long-horizon manipulation. *arXiv preprint arXiv:2607.05377*, 2026.

Physical Intelligence, Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, et al. $\pi_{0.5}$: A vision-language-action model with open-world generalization. In *Proceedings of The 9th Conference on Robot Learning*, volume 305 of *Proceedings of Machine Learning Research*, pp. 17–40, 2025.

Carl Qi, Xiaojie Wang, Silong Yong, Stephen Sheng, Huitan Mao, et al. Self-refining vision language model for robotic failure detection and reasoning. In *International Conference on Learning Representations (ICLR)*, 2026.

Delin Qu, Haoming Song, Qizhi Chen, Yuanqi Yao, Xinyi Ye, Yan Ding, et al. SpatialVLA: Exploring spatial representations for visual-language-action models. In *Robotics: Science and Systems (RSS)*, 2025. doi: 10.15607/RSS.2025.XXI.011.

Xingyu Ren, Chugang Yi, and Youran Sun. RouterVLA: Budgeted commissioning and expert onboarding for growing VLA pools. *arXiv preprint arXiv:2606.27355*, 2026.

Ke Rui, Yushen Zuo, Jiawei Wang, Haoran Jia, Jinming Ma, Weitao Zhou, and Minglei Li. Diagnosing semantic handoff failures in agent-orchestrated vision-language-action skill composition. In *RSS SemRob Workshop*, 2026.

Prasun Saurabh, Pablo Valle, Aitor Arrieta, Shaukat Ali, and Paolo Arcaini. VISOR: A vision-language model-based test oracle for testing robots. In *Proceedings of the 3rd ACM International Conference on AI-Powered Software*, pp. 197–207, 2026. doi: 10.1145/3805760.3814910.

Ming Shang, Yuchen Huang, Jiaoyang Chen, Haoyuan Hu, Han Yu, Liping Song, Luyun Feng, Shuo Bao, Wei Dong, Xinzhou Wang, and Fuchun Sun. BICPO-VLA: Behavior-identified continuation preference optimization for smooth asynchronous vision-language-action control. *arXiv preprint arXiv:2608.13924*, 2026.

Mustafa Shukor, Dana Aubakirova, Francesco Capuano, Pepijn Kooijmans, Steven Palma, Adil Zouitine, et al. SmolVLA: A Vision-Language-Action Model for Affordable and Efficient Robotics. *arXiv preprint arXiv:2506.01844*, 2025.

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 *IEEE International Conference on Robotics and Automation (ICRA)*, 2023.

Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M. Sadler, Wei-Lun Chao, and Yu Su. LLM-Planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 2998–3009, 2023. doi: 10.1109/ICCV51070.2023.00280.

Sanjana Srivastava, Chengshu Li, Michael Lingelbach, Roberto Martín-Martín, Fei Xia, et al. BEHAVIOR: Benchmark for everyday household activities in virtual, interactive, and ecological environments. In *Conference on Robot Learning (CoRL)*, 2021.

StarVLA Community. StarVLA: A Lego-like Codebase for Vision-Language-Action Model Developing. *arXiv preprint arXiv:2604.05014*, 2026.

Balakumar Sundaralingam, Adithyavairavan Murali, and Stan Birchfield. cuRoboV2: Dynamics-aware motion generation with depth-fused distance fields for high-DoF robots. *arXiv preprint arXiv:2603.05493*, 2026.

Wil Thomason, Zachary Kingston, and Lydia E. Kavraki. Motions in microseconds via vectorized sampling-based planning. In *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 8749–8756, 2024. doi: 10.1109/ICRA57147.2024.10611190.

Khoa Vo, Sieu Tran, Taisei Hanyu, Yuki Ikebe, Duy Nguyen, et al. CodeGraphVLP: Code-as-planner meets semantic-graph state for non-markovian vision-language-action models. *arXiv preprint arXiv:2604.22238*, 2026.

Xiao-Ming Wu, Jia-Feng Cai, Jian-Jian Jiang, Dian Zheng, Yi-Lin Wei, and Wei-Shi Zheng. An economic framework for 6-DoF grasp detection. In *European Conference on Computer Vision (ECCV)*, pp. 357–375, 2024. doi: 10.1007/978-3-031-73383-3_21.

Yanru Wu, Weiduo Yuan, Ang Qi, Vitor Guizilini, Jiageng Mao, and Yue Wang. Large reward models: Generalizable online robot reward generation with vision-language models. *arXiv preprint arXiv:2603.16065*, 2026.

Jiahui Yang, Jason Jingzhou Liu, Yulong Li, Youssef Khaky, Kenneth Shaw, and Deepak Pathak. Deep reactive policy: Learning reactive manipulator motion planning for dynamic environments. In *Proceedings of The 9th Conference on Robot Learning*, volume 305 of *Proceedings of Machine Learning Research*, pp. 5056–5071, 2025.

Yandan Yang, Shuang Zeng, Tong Lin, Xinyuan Chang, Dekang Qi, Junjin Xiao, et al. ABot-M0: VLA Foundation Model for Robotic Manipulation with Action Manifold Learning. *arXiv preprint arXiv:2602.11236*, 2026a.

Yi Yang, Zhihong Liu, Siqi Kou, Yiyang Chen, Yanzhe Hu, Jianbo Zhou, et al. World-Language-Action Model for Unified World Modeling, Language Reasoning, and Action Synthesis. *arXiv preprint arXiv:2606.05979*, 2026b.

Guang Yin, Yitong Li, Yixuan Wang, Dale McConachie, Paarth Shah, Kunimatsu Hashimoto, Huan Zhang, Katherine Liu, and Yunzhu Li. CodeDiffuser: Attention-enhanced diffusion policy via VLM-generated code for instruction ambiguity. In *Robotics: Science and Systems (RSS)*, 2025.

Sihyung Yoon, Minjong Yoo, Sanghyun Ahn, Seojeong Choi, and Honguk Woo. RoboBRIDGE: A modular framework for bridging policies to robust real-world robotic agents. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2026.

Chao Yu, Yuanqing Wang, Zhen Guo, Hao Lin, Si Xu, Hongzhi Zang, Quanlu Zhang, Yongji Wu, Chunyang Zhu, Junhao Hu, Zixiao Huang, Mingjie Wei, Yuqing Xie, Ke Yang, Bo Dai, Zhexuan Xu, Jiakun Du, Xiangyuan Wang, Xu Fu, Letong Shi, Zhihao Liu, Kang Chen, Weilin Liu, Gang Liu, Boxun Li, Jianlei Yang, Zhi Yang, Guohao Dai, and Yu Wang. RLinf: Flexible and efficient large-scale reinforcement learning via macro-to-micro flow transformation. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 829–846, 2026.

Tianyuan Yuan, Zibin Dong, Yicheng Liu, and Hang Zhao. Fast-WAM: Do world action models need test-time future imagination? *arXiv preprint arXiv:2603.16666*, 2026a.

Yuqian Yuan, Wentong Li, Zhaocheng Li, Yutong Lin, Juncheng Li, Siliang Tang, Jun Xiao, Yueting Zhuang, and Wenqiao Zhang. InstructSAM: Segment any instance with any instructions. *arXiv preprint arXiv:2605.26102*, 2026b.

Yanjie Ze, Gu Zhang, Kangning Zhang, Chenyuan Hu, Muhan Wang, and Huazhe Xu. 3D diffusion policy: Generalizable visuomotor policy learning via simple 3D representations. In *Robotics: Science and Systems (RSS)*, 2024.

Jiachen Zhang, Junnan Nie, Junyi Lao, Wei Cheng, Chenghao Liu, Jiaxin Jiang, and Songfang Huang. What frozen VLAs already know about success: A probing study of value-like structure in foundation robot policies. *arXiv preprint arXiv:2605.28527*, 2026a.

Junyi Zhang, Jiaxin Ge, Hanjun Yoo, Letian Fu, Zihan Yang, Yaowei Liu, Raj Saravanan, Shaofeng Yin, Justin Yu, Dantong Niu, et al. Playful agentic robot learning. *arXiv preprint arXiv:2606.19419*, 2026b.

Yang Zhang, Jiangyuan Zhao, Chenyou Fan, Fangzheng Yan, Tian Li, Haitong Tang, et al. PRTS: A Primitive Reasoning and Tasking System via Contrastive Representations. *arXiv preprint arXiv:2604.27472*, 2026c.

Yixian Zhang, Huanming Zhang, Feng Gao, Xiao Li, Zhihao Liu, et al. Harness VLA: Steering frozen VLAs into reliable manipulation primitives via memory-guided agents. *arXiv preprint arXiv:2607.08448*, 2026d.

Tony Z. Zhao, Vikash Kumar, Sergey Levine, and Chelsea Finn. Learning fine-grained bimanual manipulation with low-cost hardware. In *Robotics: Science and Systems (RSS)*, 2023.

Weiheng Zhao, Haoyi Jiang, Xin Shi, Liu Liu, Fan Huang, Zhizhong Su, Wei Sui, and Xinggang Wang. Faster-WAM: Efficient inference-time future conditioning for robust world action models. *arXiv preprint arXiv:2608.04404*, 2026.

Jinliang Zheng, Jianxiong Li, Zhihao Wang, Dongxiu Liu, Xirui Kang, Yuchun Feng, et al. X-VLA: Soft-prompted transformer as scalable cross-embodiment vision-language-action model. In *International Conference on Learning Representations (ICLR)*, 2026.

Xueyang Zhou, Yangming Xu, Guiyao Tie, Yongchao Chen, Guowen Zhang, Duanfeng Chu, Pan Zhou, and Lichao Sun. LIBERO-PRO: Towards robust and fair evaluation of vision-language-action models beyond memorization. *arXiv preprint arXiv:2510.03827*, 2025.

Yuke Zhu, Josiah Wong, Ajay Mandlekar, Roberto Martín-Martín, Abhishek Joshi, Kevin Lin, Soroush Nasiriany, and Yifeng Zhu. robosuite: A modular simulation framework and benchmark for robot learning. *arXiv preprint arXiv:2009.12293*, 2020.

Brianna Zitkovich, Tianhe Yu, Sichun Xu, Peng Xu, Ted Xiao, et al. RT-2: Vision-language-action models transfer web knowledge to robotic control. In *Conference on Robot Learning (CoRL)*, 2023.

APPENDIX CONTENTS

A Methodological Details **17**
A.1 P^5 Stage Contracts . . . 17
A.2 RoboAware: Training and Deployment . . . 18
A.3 Responsibility Preference . . . 20
A.4 Q-Value Model . . . 21
A.5 Reward Function . . . 22

B API Library Instantiation **23**
B.1 Typed API Contracts . . . 23
B.2 Object Perception . . . 23
B.3 Pose Proposal . . . 24
B.4 Motion Planning . . . 24
B.5 Gripper Control . . . 25
B.6 End-to-End Policy Execution . . . 25

C Experiment Details **29**
C.1 Benchmark Tasks . . . 29
C.2 Baseline Implementations . . . 30
C.3 Success and Runtime Metrics . . . 30
C.4 Hyperparameters and Experimental Settings . . . 31
C.5 Coding-Agent Backends . . . 31
C.6 Code-Execution Protocol . . . 31
C.7 Computation and Software . . . 32

D Additional Empirical Results **32**
D.1 Ablation of Harness-VLA . . . 32
D.2 Visualizations . . . 34
D.3 Failure Modes . . . 34

E Social Impact **35**

A METHODOLOGICAL DETAILS

A.1 P^5 STAGE CONTRACTS

Each P^5 stage declares an entry predicate, the typed inputs it requires, an admissible-family set $\mathcal{C}_i(z_i, g_i)$, the APIs a family-constrained code block may call, a success postcondition, and the failure or timeout feedback that chooses the successor stage. Table 1 summarizes these contracts; typed API I/O appears in Table 4. The coordinator selects a family only when the coding agent has control and the current stage's entry predicate holds. Once an atomic API call begins, that selection remains fixed until the call returns after satisfying its registered postcondition, reporting a failure, or exhausting its step budget.

Table 1: P^5 stage contracts and admissible families. At stage z_i of skill g_i, the coordinator selects $c_i \in \mathcal{C}_i(z_i, g_i)$ before code generation; APIs must match (z_i, c_i) (Table 2 in the main text). Columns: ϕ^{in}—entry predicate; $\mathcal{C}_i$—admissible families; ϕ^{out}—registered success postcondition; Fail—successor after failure/timeout (retry z_i, return upstream $\uparrow$, or terminate $\perp$). Under Mod, ϕ^{out} is the typed modular return (e.g. pose at *perceive*, eef pose at *propose*); under E2E, it is covered by the atomic call postcondition. *propose* is Mod-only; an E2E call may complete the skill without visiting it.

z_i	ϕ^{in}	$\mathcal{C}_i$	ϕ^{out}	Fail
perceive	target ID/pose $\notin h_i$	{Mod, E2E}	typed target info	retry / $\perp$
propose	pose/mask $\in h_i \wedge$ eef target $\notin h_i$	{Mod}	valid eef pose	retry / $\uparrow$*perceive* / $\perp$
pre-manipulate	approach target available	{Mod, E2E}	pre-contact reached	replan / $\uparrow$ / $\perp$
perform	pre-manip. ϕ^{out}	{Mod, E2E}	contact/gripper post-cond.	retry / $\rightarrow$*post-manip.* / $\perp$
post-manipulate	perform ϕ^{out}	{Mod, E2E}	retreat/release/stabilize	retry / $\uparrow$ / $\perp$

Required inputs and permitted APIs. Under Mod, the permitted API depends on the current stage: `Perceive_object_pose` at *perceive*, `Propose_eef_pose` at *propose*, `Plan_to_eef_pose` at *pre-manipulate* and *post-manipulate*, and `Perform_gripper` at *perform*. A modular code block may compose only APIs admissible for the current z_i (and any immediately chained modular helpers that do not cross a new responsibility boundary). Under E2E, the sole permitted pattern is `Perform_E2E(route_id,...)`, whose declared coverage is *perceive*, *pre-manipulate*, *perform*, and *post-manipulate* (Table 2 in the main text). After the coordinator selects E2E, the coding agent supplies `route_id` and the required typed observations in a_i. Thus, Q_θ selects the family, while the coding agent selects the backend within that family.

Why *propose* admits only Mod. `Propose_eef_pose` is the only API assigned to *propose*, where it produces the end-effector target required by modular motion planning. End-to-end backends determine grasp and approach targets internally and do not expose them through this interface, so `Perform_E2E` does not cover *propose*. Consequently $\mathcal{C}_i = \{\text{Mod}\}$ whenever z_i is *propose*, so no E2E comparison is issued at that boundary. A skill enters *propose* only when it requires an explicit end-effector target. An E2E call beginning at *perceive* or a later admissible stage can therefore complete the skill without entering *propose*.

Multi-stage atomic returns and stage advancement. A single `Perform_E2E` call may satisfy several consecutive stage postconditions before returning, but its family assignment remains fixed throughout the call. When control returns, the coding agent updates (h_i, z_i, g_i) from observable feedback only: typed returns, registered postcondition bits, failure codes, and timeout flags. The next decision stage z_{i+1} is the earliest stage in the active skill whose entry predicate still holds and whose success postcondition is unmet. Stages whose postconditions have already been satisfied are skipped, so the coordinator makes no additional family selections for them unless execution later reaches an entry where those postconditions are unmet. If the call fails or times out after partial progress, z_{i+1} is likewise the earliest unmet stage that remains enterable under the returned

feedback (often the stage at which the postcondition failed), enabling retry or upstream fallback as in Table 1. If every stage postcondition of g_i is met, the skill completes and the plan advances to the next schema in $\mathcal{G}$. If execution must continue but $\mathcal{C}_{i+1}$ is empty, the episode terminates with failure, as specified in Algorithm 1.

A.2 RoboAware: Training and Deployment

Algorithm 1: RoboAware: MCTS data collection, Q-learning, and deployment

Input: training tasks $\mathcal{T}_{\text{train}}$; test instruction ℓ_{test}; simulator $\mathcal{E}$; π_{code} and API library; budget K
Output: trained coordinator Q_θ; test execution history h and terminal reward r

Freeze π_{code} and the API library; initialize θ, $\bar{\theta} \leftarrow \theta$, and $\mathcal{D}_{\text{SCB}} \leftarrow \emptyset$;
`// Phase 1. Offline training: prepare search roots`
Collect roots $\mathcal{R}$ from training-task P^5 executions using the neighboring-state sampling protocol described below, with matching histories;
foreach *sampled root* $\xi_t \in \mathcal{R}$ **do**
  Initialize tree $\mathcal{T}$ at ξ_t and remaining budget $b \leftarrow K$;
  while $b > 0$ **do**
    `// 1. Selection`
    Follow stored paths by UCB (Equation (1)) to a nonterminal node j; select a new node only if $b \geq |\mathcal{C}_j|$, otherwise reevaluate an expanded node;
    `// 2. Expansion`
    if *node j is new* **then**
      Set $N_j = N_j(c) = 0$, $\widehat{Q}_j(c) = Q_\theta(x_j, c)$, and $A_j \leftarrow \mathcal{C}_j$;
    else
      $A_j \leftarrow \{c_j^{\text{tree}}\}$ by Equation (1);
    foreach $c \in A_j$ **do**
      Restore ξ_j and its history; generate $a_j^c \sim \pi_{\text{code}}(\cdot \mid x_j, c)$ and execute to the next boundary or termination;
      Record $(x_{j+1}^c, \xi_{j+1}^c, d_j^c)$ and compute $\tilde{r}_j^c$ from the skill/task oracle scores;
      Attach the successor to $\mathcal{T}$; append $(x_j, c, \tilde{r}_j^c, x_{j+1}^c, d_j^c, \xi_{j+1}^c)$ to $\mathcal{D}_{\text{SCB}}$;
      `// 3. Rollout`
      Continue a nonterminal successor to a stopping condition without tree expansion; obtain G_{j+1}, or set $G_{j+1} = 0$ if $d_j^c = 1$;
      $G_j \leftarrow \tilde{r}_j^c + \gamma G_{j+1}$;
      `// 4. Backpropagation`
      Back up G_j and prefix rewards along the selected path; increment visits, average returns into $\widehat{Q}$, and set $b \leftarrow b - 1$;
`// Q-learning after data collection`
Fit Q_θ on all of $\mathcal{D}_{\text{SCB}}$ for M minibatch updates using Equations (4) and (5), learning rate η, and target refresh interval J;
`// Phase 2. Deployment with fixed parameters`
Observe o_0; generate test plan $\mathcal{G}$ from $(\ell_{\text{test}}, o_0)$; initialize $h_0 \leftarrow \emptyset$, (z_0, g_0), and $i \leftarrow 0$;
while *the task is nonterminal and the episode horizon remains* **do**
  Build $x_i = (\ell_{\text{test}}, o_i, h_i, z_i, g_i)$ and admissible families $\mathcal{C}_i(z_i, g_i)$;
  if $\mathcal{C}_i = \emptyset$ **then** Terminate with failure and exit the loop;
  $c_i^\star \leftarrow \arg\max_{c \in \mathcal{C}_i} \rho_\theta(c \mid x_i)$; generate $a_i \sim \pi_{\text{code}}(\cdot \mid x_i, c_i^\star)$;
  Execute a_i to the next boundary or termination, respecting atomic APIs;
  Observe feedback; update x_{i+1} and $\mathcal{G}$; $i \leftarrow i + 1$;
return *success indicator: terminal task reward r*

Algorithm 1 separates offline training from deployment. During training, we collect P^5 search roots, run MCTS from those roots, and fit Q_θ on the resulting transitions. During deployment, the trained coordinator selects a family at each eligible stage boundary and updates its input from execution feedback. Only Q_θ is optimized. The coding agent π_{code} and API library are fixed before data collection and remain frozen during training and deployment.

Trajectory collection and neighboring-state sampling. For each training task, π_{code} constructs a task plan $\mathcal{G}$ comprising P^5 skill schemas, with g_i identifying the active skill and z_i its current stage. A fixed sampling rule with support on all admissible families selects the family before each code block is generated. During the resulting simulator rollout τ, we record observations, execution history, schema states, and saved simulator states, together with the start time t_i of each visited stage. For each t_i, let $\mathcal{W}_i(\Delta)$ contain the nonterminal saved times within Δ steps of t_i at which the same stage-entry conditions hold and control is available to the coding agent. It includes t_i and excludes intermediate states inside an atomic API invocation. We draw S times independently and uniformly from this set, with replacement, and form a separate root snapshot for each draw. Each root uses its own matching context $x_t = (\ell, o_t, h_t, z_t, g_t)$. The snapshot, denoted by $\xi_t = (x_t, m_t, p_t, r)$, also stores the simulator and execution state needed to restore that root exactly. Here, r denotes access to the task-success evaluator rather than a future reward available to the coordinator. Thus, neighboring roots may differ physically, but all counterfactual branches of one root share the same physical state and observable-history prefix.

Search and learning contracts. Each root receives a budget of K completed suffix simulations, including the initial family comparisons. UCB descent follows stored tree paths using Equation (1), stopping before terminal nodes. At a new node, counts start at zero and $\widehat{Q}_i(c)$ is initialized from $Q_\theta(x_i, c)$, whose parameters remain fixed during data collection. Descent stops at an unexpanded nonterminal node when the remaining budget covers every admissible family. Otherwise, it stops at an expanded nonterminal node and selects one family for reevaluation. Every evaluation restores the selected snapshot, samples a fresh family-constrained code block, executes it to the next semantic boundary, and saves the resulting transition and successor snapshot. From each nonterminal successor, the fixed coding agent and API library continue physics execution under the same family-sampling continuation rule to task termination, unrecoverable failure, the episode horizon, or an empty admissible-family set. Reaching any of these stopping conditions sets $d = 1$ and ends the suffix. States visited only during rollout are not added to the search tree. The suffix return is backed up along the selected path using the recorded prefix rewards, with visit counts incremented and $\widehat{Q}$ updated by empirical averaging. All expanded edges enter $\mathcal{D}_{\text{SCB}}$, including failed and nonpreferred branches. Scores r_{p} and r_{m} are emitted at skill completion and are zero at other transitions, while r is emitted only at task termination. Q-learning then uses Equations (4) and (5), with no bootstrap at terminal states and a target copy refreshed every J updates. The sampling parameters (S, Δ), family-sampling rule, optimization budget M, learning rate η, target-update interval J, and related budgets are listed in Table 12.

Deployment contract. Only the trained Q_θ, frozen coding agent, and API library are retained. At every decision point, the coordinator builds x_i from observable information, selects an admissible family, and conditions code generation on that choice. Each atomic API call runs until its postcondition is satisfied, a failure is reported, or its step budget is exhausted. Once the call returns, the coding agent uses the feedback to update the history and remaining plan and to determine whether to advance or retry a stage. Task termination, an unrecoverable failure, or the episode horizon ends execution. An empty admissible-family set is treated as failure. Neither state restoration, tree search, oracle shaping rewards, nor parameter updates are used at deployment. Only the terminal task reward r is reported.

Training data and task pool. We construct $\mathcal{D}_{\text{SCB}}$ using the procedure in Algorithm 1 and the model in Appendix A.4, with the sampling and optimization settings given below. Data collection with privileged simulator access and subsequent Q_θ updates use only scene seed 0 on the coordinator training pool (80 LIBERO-Pro, 4 RoboSuite, and 16 RoboTwin 2.0 tasks; Appendix C.1). Each training task contributes (i) one or more physics trajectories used to mark P^5 stage-entry times, (ii) S neighboring roots per stage entry drawn from $\mathcal{W}_i(\Delta)$ as specified above, and (iii) up to K completed suffix simulations per root, with every expanded family edge written into $\mathcal{D}_{\text{SCB}}$.

Root and family sampling. During trajectory collection, we sample uniformly from the families admissible at the current (z_i, g_i), so every family in $\mathcal{C}_i$ has nonzero probability. Neighboring-state sampling draws S=4 roots independently and uniformly with replacement from $\mathcal{W}_i(\Delta)$ using the default window Δ=10 saved control steps. Within each root budget of K=20 completed suffix simulations (Table 12), the first expansion of a new node evaluates every $c \in \mathcal{C}_i$ once via SCB;

later visits reevaluate a single UCB-selected family (Equation (1)). Rollout continuation after expansion reuses the same family-sampling rule and does not add rollout-only states as tree nodes. All evaluated edges, including failed and nonpreferred branches, remain in $\mathcal{D}_{\mathrm{SCB}}$.

Q initialization before data collection. Before data collection, we load the released Qwen3.5-0.8B backbone weights, randomly initialize the <VALUE> embedding and MLP head, and set $\bar{\theta} \leftarrow \theta$ (Algorithm 1). We then hold θ fixed throughout collection. Each new tree node starts with zero visits and initializes $\widehat{Q}_i(c)$ from $Q_\theta(x_i, c)$, which provides the initial value estimate for UCB selection. No return regression or supervised fine-tuning is performed before collection, and the model receives no privileged oracle features. Bellman updates begin only after $\mathcal{D}_{\mathrm{SCB}}$ is complete, following the single collection-and-fitting cycle used in the reported experiments.

Optimization protocol. After collection, we optimize θ on the fixed offline set with the Smooth-L_1 Bellman loss in Equations (4) and (5), minibatches drawn from $\mathcal{D}_{\mathrm{SCB}}$, a stop-gradient target network $Q_{\bar{\theta}}$, and no bootstrap at terminal states. Table 2 lists the fine-tuning method, thinking setting, optimizer, batch size, learning-rate schedule (cosine with warmup), update budget M, target refresh interval J, numeric precision, and checkpoint rule. We select a checkpoint on the validation split described below, following the checkpoint rule in Table 2, and freeze it for all evaluations on seeds 1–10.

Table 2: Coordinator Q-fitting settings. Symbols match Algorithm 1.

Setting	Value
Fine-tuning method	full-FT (backbone + MLP value head)
Thinking	Disabled
Optimizer	AdamW
Minibatch size	16
Learning rate η / schedule	10^{-5}, cosine, 5% warmup
Gradient updates M	10,000
Target refresh every J updates	500 (hard copy $\bar{\theta} \leftarrow \theta$)
Weight decay / grad clip	10^{-4} / 1.0
Numeric precision	bf16
Checkpoint selection	lowest validation Bellman loss

Grouped training and validation split. The Mod and E2E branches from a given root share both a physical state and an observable-history prefix. Roots sampled from the same neighborhood $\mathcal{W}_i(\Delta)$ are also closely related because they come from nearby states in one trajectory. A random split of individual transitions could therefore place branches from the same root, or transitions from neighboring roots, in both training and validation. To keep these related samples together, we group all transitions from each root and all roots drawn from the same stage-entry neighborhood $\mathcal{W}_i(\Delta)$. Groups are assigned entirely to training or validation using a $90/10$ split, optionally stratified by benchmark and by whether $\mathcal{C}_i$ is singleton or binary. No group contributes transitions to both splits; evaluation seeds 1–10 remain unused for fitting or model selection.

A.3 Responsibility Preference

The normalization in Equation (6) defines a relative preference over the nonempty admissible-family set $\mathcal{C}_i$. When $\mathcal{C}_i = \{\mathrm{E2E}, \mathrm{Mod}\}$, the value gap compares the same ordered pair at every decision point:

$$\Delta_\theta(x_i) = Q_\theta(x_i, \mathrm{E2E}) - Q_\theta(x_i, \mathrm{Mod}). \tag{7}$$

Thus, $\Delta_\theta(x_i)$ depends on the context, while the family identities and their order are fixed by definition. The normalized preferences can then be expressed as sigmoids of this gap. Dividing the E2E

numerator and denominator by $\exp(Q_\theta(x_i, \text{E2E}))$ gives

$$\begin{aligned}\rho_\theta(\text{E2E} \mid x_i) &= \frac{\exp(Q_\theta(x_i, \text{E2E}))}{\exp(Q_\theta(x_i, \text{E2E})) + \exp(Q_\theta(x_i, \text{Mod}))} \\ &= \frac{1}{1 + \exp(Q_\theta(x_i, \text{Mod}) - Q_\theta(x_i, \text{E2E}))} \\ &= \sigma(\Delta_\theta(x_i)), \\ \rho_\theta(\text{Mod} \mid x_i) &= 1 - \rho_\theta(\text{E2E} \mid x_i) = \sigma(-\Delta_\theta(x_i)).\end{aligned} \tag{8}$$

The common normalization cancels in the ratio, so taking its logarithm yields

$$\begin{aligned}\log \frac{\rho_\theta(\text{E2E} \mid x_i)}{\rho_\theta(\text{Mod} \mid x_i)} &= \log \frac{\exp(Q_\theta(x_i, \text{E2E}))}{\exp(Q_\theta(x_i, \text{Mod}))} \\ &= Q_\theta(x_i, \text{E2E}) - Q_\theta(x_i, \text{Mod}) = \Delta_\theta(x_i).\end{aligned} \tag{9}$$

Positive gaps favor E2E, negative gaps favor modular composition, and a zero gap gives equal preference to both families. The larger preference equals $\sigma(|\Delta_\theta(x_i)|)$ and increases monotonically with $|\Delta_\theta(x_i)|$, which therefore measures preference strength.

More generally, exponentiation is strictly increasing and the denominator in Equation (6) is positive and shared by all admissible families. Consequently, the transformation preserves family ordering, including ties, and yields the same maximizing families:

$$\arg\max_{c \in \mathcal{C}_i} \rho_\theta(c \mid x_i) = \arg\max_{c \in \mathcal{C}_i} Q_\theta(x_i, c). \tag{10}$$

When only one family is admissible, its normalized preference is one; the binary sigmoid identities apply only when both families are admissible. An empty admissible-family set terminates execution with failure, as specified in the deployment phase of Algorithm 1. The deployment algorithm therefore selects via ρ_θ (equivalently via Q_θ), then generates and executes the corresponding block before reconsidering responsibility at the next semantic boundary.

A.4 Q-Value Model

We instantiate $Q_\theta(x_i, c)$ with Qwen3.5-0.8B as the multimodal backbone and a scalar MLP value head. Only this coordinator value model is optimized under Equation (4); the coding agent π_{code}, the typed API library, and all low-level backend checkpoints remain frozen throughout EAL and deployment.

Observation encoding. At decision point i, the visual stream uses the same agent-view camera as the modular perception API. We resize the agent-view RGB frame I_i^{rgb} and a visualization of the metric depth map I_i^{d} to a shared resolution $R \times R$ (Table 3) and encode both through the backbone's native image pathway. We serialize proprioception q_i, including joint positions, end-effector pose, and gripper opening, as structured text. Thus, all components of o_i enter Q_θ through its image and text pathways, without a separate numeric depth input, depth adapter, or proprioceptive MLP encoder.

History length and truncation. The execution history h_i retains the most recent H stage transitions in full, each recording the chosen family, typed API returns or failure codes, stage advances or retries, and remaining budgets. Older transitions are dropped and replaced by a fixed-length summary of completed skills/stages, persistent object bindings, cumulative failure counts, and the residual step/wall-clock budget. The summarizer uses only information already present in the observable history: training-time oracle fields never enter h_i.

Token order. For a queried family c, inputs are packed into one multimodal sequence:

$$\ell \;\rightarrow\; (I_i^{\text{rgb}}, I_i^{\text{d}}) \;\rightarrow\; q_i \;\rightarrow\; h_i \;\rightarrow\; g_i \;\rightarrow\; z_i \;\rightarrow\; c \;\rightarrow\; \texttt{<VALUE>},$$

i.e., instruction, images, proprioception, execution history, current skill, current P^5 stage, family identity, then the special `<VALUE>` token. For example, consider

RoboTwin `place_bread_basket` at the start of the *perform* stage for picking up the bread. The instruction ℓ is "Place the bread into the basket," and the image inputs show the current $R\times R$ RGB frame and depth colormap of the bread, basket, and two arms. The proprioceptive input q_i lists joint positions, end-effector poses, and gripper openings. The history h_i records the preceding stage transitions. These may include Mod returning the bread pose at *perceive*, producing an end-effector target at *propose*, and reaching the pre-grasp pose at *pre-manipulate*. It also contains the remaining step and wall-clock budgets and a fixed-length summary of earlier completed skills. The active schema g_i identifies the bread-picking skill and its unmet postconditions, while $z_i = \textit{perform}$. Both families are admissible at this point, so the sequence ends with the queried $c \in \{\mathrm{Mod}, \mathrm{E2E}\}$ followed by `<VALUE>`. The backbone produces the final-layer hidden state at `<VALUE>`; an L_{mlp}-layer MLP with hidden width d_{mlp} maps that state to the scalar $Q_\theta(x_i, c)$. Table 3 lists the quantitative settings used in our experiments.

Table 3: Q-value model settings.

Setting	Value
Backbone	Qwen3.5-0.8B
Image viewpoint	agent-view RGB + depth visualization
Image resolution $R\times R$	224×224
Proprioception encoding	structured text
Depth encoding	colormap image
History window H (full transitions)	4
MLP depth L_{mlp}	2
MLP width d_{mlp}	512
Frozen parameters	π_{code}, API library, backend checkpoints

A.5 Reward Function

This subsection specifies the training-only oracle rewards in Equation (3). The coordinator never observes m_i, p_i, r_{p}, r_{m}, or r at deployment. As in Appendix A.2, $r^c_{\mathrm{p},i}$ and $r^c_{\mathrm{m},i}$ are emitted when a P^5 skill completes under family c and are zero on non-completion transitions; the binary task reward r is emitted only when the episode terminates ($d^c_i = 1$).

Perception score r_{p}. Let m_i be the oracle binary mask of the language-referenced target object in the agent-view image at the skill-entry snapshot ξ_i. When the executed block exposes an explicit mask $\hat{m}^c_i$ (modular `Perceive_object_pose`), mask agreement is the intersection-over-union

$$r^c_{\mathrm{p},i} = \mathrm{IoU}(\hat{m}^c_i, m_i) = \frac{|\hat{m}^c_i \cap m_i|}{|\hat{m}^c_i \cup m_i|} \in [0,1], \tag{11}$$

with the convention $\mathrm{IoU} = 0$ if both masks are empty. End-to-end blocks do not return a mask. For those branches we score target-object identity rather than pixel overlap: after execution, the simulator identifies the object instance that was contacted or whose pose changed under the skill; $r^c_{\mathrm{p},i} = 1$ if that instance matches the object labeled by m_i, and 0 otherwise. Thus both families receive a perception score in $[0,1]$, but only modular perception is judged by mask IoU; E2E perception is judged by whether the call acted on the oracle-referenced object, consistent with the E2E postcondition coverage in Table 1.

Manipulation score r_{m}. The manipulation score measures whether the object's simulator ground-truth pose after skill execution meets the skill's geometric goal. Write p^c_{i+1} for the privileged 6-DoF object pose read from ξ^c_{i+1}, and $p^\star_i$ for the skill goal pose supplied by the same oracle pose channel (placement target, handover target, or other stage goal recorded with the task). Let $t(\cdot)$ and $R(\cdot)$ denote translation and rotation. We measure translation error in meters using the Euclidean distance $\|t(p^c_{i+1}) - t(p^\star_i)\|_2$, and rotation error in radians using the geodesic angle on $\mathrm{SO}(3)$, $\angle\big(R(p^c_{i+1}), R(p^\star_i)\big) = \arccos\big((\mathrm{tr}(R(p^c_{i+1})^\top R(p^\star_i)) - 1)/2\big)$. For skills defined by a contact or grasp postcondition, such as a successful lift, $r^c_{\mathrm{m},i}$ is the corresponding binary oracle predicate. For skills with a 6-DoF placement goal, we instead apply the pose thresholds

$$r^c_{\mathrm{m},i} = \mathbb{I}\big[\|t(p^c_{i+1}) - t(p^\star_i)\|_2 \le \epsilon_p \ \wedge\ \angle\big(R(p^c_{i+1}), R(p^\star_i)\big) \le \epsilon_R\big], \tag{12}$$

where ϵ_p is the translational threshold in meters and ϵ_R is the rotational threshold in radians (Table 12). The main-text phrase "object-pose error against p_i" refers to this privileged pose channel used to form $(p^c_{i+1}, p^\star_i)$, not to scoring the modular pose estimate against the entry state.

Terminal reward and stopping. The evaluator returns $r = 1$ iff the task success predicate holds at termination, and $r = 0$ otherwise. An episode sets $d^c_i = 1$ and stops on task success, unrecoverable failure, empty admissible-family set, or the episode horizon; only the success case can yield $r = 1$. The shaped transition reward remains $\tilde{r}^c_i = w_{\mathrm{p}} r^c_{\mathrm{p},i} + w_{\mathrm{m}} r^c_{\mathrm{m},i} + w_{\mathrm{t}} d^c_i r$ with $w_{\mathrm{p}} + w_{\mathrm{m}} + w_{\mathrm{t}} = 1$.

B API Library Instantiation

B.1 Typed API Contracts

Each embodied API in Table 2 in the main text owns one or more P^5 stages and declares typed inputs, outputs, coordinate frames, postconditions, and failure states (Table 4). The family-specific rules in Table 5 determine which APIs a generated code block may call. The coding agent specifies the semantic target and route, whereas embodiment adapters resolve frames, execute the request, and report measured completion. This separates an incorrect target from a failure to reach a correct target and prevents an internal fallback from silently changing the meaning of a call.

Within $\mathcal{A}_{\mathrm{E2E}}$, `route_id` selects a retained VLA, video-action, or world-action checkpoint inside the code block; it is not a separate action scored by Q_θ. The coordinator observes only typed API returns, not privileged simulator state. We freeze the coding agent, API contracts, backend bindings, and checkpoints before coordinator training.

Table 4: Typed input/output contracts for the high-level embodied APIs in Table 2 in the main text. The E2E route uses `route_id` to select one frozen backend.

API	Stage	Input	Output
`Perceive_object_pose`	*perceive*	task instruction + agent-view RGB-D + calibration	target pose + mask/geometry provenance
`Propose_eef_pose`	*propose*	RGB-D geometry + target mask + robot state + task context	ranked end-effector target + screening provenance
`Plan_to_eef_pose`	*pre/post-manipulate*	scene observation, robot state, target pose, optional attachment	continuous, smooth, collision-aware qpos sequence
`Perform_gripper`	*perform*	gripper open/close command	updated gripper state
`Perform_E2E`	*perceive*, *pre-manipulate*, *perform*, *post-manipulate*	route ID + task instruction + RGB-D + proprioception	closed-loop control until postcondition, failure, or budget

Table 5: Policy-family bindings for generated code blocks. Mod composes stage-admissible APIs; E2E invokes one frozen backend through `route_id`.

Policy family	Code-block space	Permitted API pattern	Backend choice
Mod	$\mathcal{A}_{\mathrm{Mod}}$	stage-admissible modular API composition	fixed selected modular backends
E2E	$\mathcal{A}_{\mathrm{E2E}}$	`Perform_E2E(route_id,...)`	retained frozen backend selected inside a_i

B.2 Object Perception

`Perceive_object_pose` grounds a language reference in an agent-view RGB-D observation and combines the resulting mask with camera calibration and depth to recover a robot-frame target. Valid masked depth pixels are back-projected with the camera intrinsics, transformed through the camera extrinsics, filtered for invalid or isolated points, and summarized by a robust centroid and oriented extent. The API returns the pose together with the mask, supporting points, confidence, and source observation. If the mask is empty, the depth is inconsistent, or the reference is ambiguous, it reports a failure instead of returning a pose.

We use InstructSAM (Yuan et al., 2026b) as the grounding backend because its instruction-conditioned interface directly accepts the detailed free-form prompt produced by the coding agent from the current task context. The agent can therefore preserve task-specific attributes and spatial relations in a single perception call. The resulting mask is lifted through the shared geometric pipeline, so downstream proposal and motion APIs retain the same typed outputs and failure checks.

B.3 POSE PROPOSAL

`Propose_eef_pose` consumes the frozen RGB-D observation, target mask, and derived geometry and returns a grasp target together with its score, rank, predicted width, coordinate transform, screening result, and provenance. We compare proposal methods using the same cuRoboV2 downstream planner and retain each baseline's original candidate ranking. With the planner held fixed, Success@k in Table 6 reflects differences in candidate coverage and ordering.

This API owns grasp proposal, not placement-pose generation. For placement, the task supplies a target object pose; the motion API resolves it to a TCP target through the held-object transform and ranks routes by feasibility, continuity, and the declared approach or contact constraints.

Our NeuGraspNet+ backend addresses two failure modes of native NeuGraspNet (Jauhri et al., 2024). When native geometric sampling produces too few candidates, a *RobustSampler* derives deterministic top-down, side, and oblique seeds from the target point cloud and its principal directions. The released NeuGraspNet checkpoint still predicts their quality and gripper width, so the fallback expands coverage without another grasp network or task-specific proposals.

NeuGraspNet+ then uses cuRoboV2 to identify up to five candidates with feasible approach, grasp, and lift trajectories, and selects the top-ranked grasp from this feasible set. Its confidence gate preserves the network ranking when scores are informative, uses geometry to break near-ties, and gives geometry greater weight when all feasible scores are low. This avoids globally replacing a useful learned ranking with a fixed geometric prior.

NeuGraspNet+ reaches $18/18$ Success@1, Success@3, and Success@5 on the 18-task Panda screen (Table 6) and is frozen as the grasp backend. These selection rules prioritize first-choice grasp success over speed and do not involve retraining NeuGraspNet. Because they were developed using the benchmark, their performance on new object distributions requires further validation.

Table 6: Grasp proposal results on the 18-task Panda benchmark. All baseline methods use their native proposal rankings; NeuGraspNet uses our final selection pipeline. All methods share the same downstream cuRoboV2 motion planner. Success@k indicates whether a successful grasp is found within the first k evaluated proposals. Proposal and screening times are averaged over all 18 tasks; Time@1 is averaged over successful Top-1 trials.

Method	Success@1 ↑	Success@3 ↑	Success@5 ↑	Proposal time (s) ↓	Screening time (s) ↓	Time@1 (s) ↓
AnyGrasp (Fang et al., 2023)	13/18	16/18	17/18	7.98	3.46	36.39
GraspGen (Murali et al., 2026)	11/18	12/18	12/18	**4.54**	3.16	24.84
NeuGraspNet (Jauhri et al., 2024)	12/18	13/18	13/18	12.90	1.56	39.05
EconomicGrasp (Wu et al., 2024)	10/18	12/18	12/18	5.03	**1.11**	**20.68**
RNGNet (Chen et al., 2025)	7/18	8/18	8/18	18.89	1.34	46.33
NeuGraspNet+ (ours)	**18/18**	**18/18**	**18/18**	14.36	2.23	49.04

B.4 MOTION PLANNING

`Plan_to_eef_pose` converts a caller-specified end-effector or held-object target into the robot's command frame, executes a trajectory, and verifies arrival. The fixed-grasp protocol in Table 7 supplies every planner with the same target and therefore isolates planning, whereas planner-native results measure the complete native route. We retain cuRoboV2 (Sundaralingam et al., 2026) as the core global planner and trajectory checker; it is also shared by the proposal comparison in Table 6.

For each checked call, the API freezes the current RGB-D frame, robot state, camera calibration, and target mask. It filters robot-image residuals and excludes the grasp target where contact is intended. Closing a successful grasp creates a held-object state containing the measured hand–object transform and attachment geometry; later paths check both robot and attachment against the environment. With `target_frame=''tcp''`, the public TCP goal is converted to the native hand frame. With `target_frame=''held_object''`, the saved transform determines the hand goal that realizes a desired object pose.

Explicit motion modes. `goto_pose` provides three motion modes with different planning costs and collision checks. The default `auto` mode uses cuRoboV2 first, collision-checked multi-seed IK second, and the existing bounded recovery route last; it is appropriate for grasping, long

transport, and cluttered motion. `ik_checked` skips global planning but captures the scene and collision-checks dense IK branches, including the attachment, for short contact-free adjustments. `ik_unchecked` skips scene capture and scene-collision checking for deliberate contact such as staged insertion, drawer closing, or door pushing, while retaining multi-seed IK, strict joint-continuity limits, dense execution, tracking, and arrival checks. The "unchecked" mode therefore omits scene-collision checks while retaining these control and execution checks.

The mode is selected before execution so that intended contact does not consume the budget in repeated collision-avoidance attempts. This also ensures that a call requested with collision checking cannot silently switch to unchecked execution. For example, a microwave program can use `auto` for grasp and transport, `ik_checked` for final free-space alignment, and `ik_unchecked` for insertion or a door-pushing arc.

Approach and execution acceptance. `z_approach` adds a waypoint along the declared tool frame. For held-object placement, `contact_target` marks only a bounded terminal segment as expected contact, leaving the preceding transport checked. Before returning, every route monitors joint tracking and verifies joint and end-effector arrival. The typed result records the resolved frame, motion mode, trajectory source, performed checks, timing, residuals, and a specific failure category, allowing history h_i to distinguish planning, execution, and semantic-target failures.

Table 7: Motion-planning results on the 18-task Panda benchmark. The planner-native fresh protocol is the main end-to-end comparison: each planner independently resolves endpoints, screens proposals, plans, and executes. Pipeline time includes candidate screening, endpoint resolution, and planning; Plan time measures the final selected trajectory only. The fixed-grasp protocol uses a shared grasp target. The conditional diagnostic uses cuRobo-success anchors and should not be ranked directly against the planner-native results. Avoid Everything was evaluated only in the conditional pose-goal diagnostic.

	Planner-native fresh (main)			Fixed grasp		Conditional diagnostic	
Planner	Plannable ↑	Task success ↑	Pipeline / Plan (s) ↓	Task success ↑	Plan (s) ↓	Grasp success ↑	Plan (s) ↓
cuRoboV2 (Sundaralingam et al., 2026)	**18/18**	**18/18**	**3.82 / 0.254**	**15/18**	**0.192**	**18/18**	**0.935**
DRP/IMPACT (Yang et al., 2025)	16/18	14/18	6.23 / 0.418	11/18	<u>0.576</u>	16/18	<u>1.526</u>
VAMP (Thomason et al., 2024)	12/18	11/18	73.30 / 9.320	6/18	0.931	14/18	1.598
Avoid Everything (Fishman et al., 2025)	–	–	–	–	–	2/18	3.677

B.5 Gripper Control

`Perform_gripper` implements binary open or close actions. A command returns only after the observed opening or joint signal remains stable for a fixed dwell window; failure to settle within the budget is reported explicitly. A successful close also initializes the held-object state described above from the measured hand pose.

The coding agent uses the same gripper interface across Franka, RoboSuite, and RoboTwin 2.0, with platform-specific adapters mapping commands to native gripper coordinates. An arm selector is omitted for single-arm platforms and explicit on bimanual systems, preserving the selected arm through motion and release. Because this operation is deterministic and embodiment-local, we do not screen alternative gripper controllers, and Q_θ does not score the adapter.

B.6 End-to-End Policy Execution

`Perform_E2E` maps an explicit `route_id` to a frozen VLA, video-action, or world-action checkpoint. It supports three mutually exclusive instruction modes. *Task mode* forwards the environment task instruction when neither an object nor a prompt is provided. *Grasp–lift mode* takes an object name, generates a local grasp-and-lift instruction, and updates the held-object state after perception confirms an increase in the object's height. *Bounded-prompt mode* forwards an explicit prompt verbatim and requires a step budget because arbitrary local instructions have no common success detector. Invalid argument combinations and missing required task language are rejected.

Execution ends on task completion, environment termination, a supported model stop signal, or budget exhaustion; grasp–lift mode also ends when its local completion predicate is met. Each call

records the instruction and its source, route, executed steps, termination reason, and measured post-condition. Full-task success and grasp–lift success use the task and local predicates, respectively.

We screen released routes on 14 LIBERO-Pro tasks, four RoboSuite tasks, and ten RoboTwin 2.0 tasks. The screen specifies one initial state per task, but RoboTwin's full-task evaluator rejects states that fail expert validation; realized initial states may therefore differ across methods. Full-task and grasp–lift success are reported in Table 8, and execution statistics in Tables 9 to 11. Successful-trial completion times are reported separately from all-trial runtime statistics. Retained routes receive stable `route_id`s and are frozen before coordinator training.

FastWAM and FasterWAM use their LIBERO checkpoints on LIBERO-Pro and RoboSuite, and their RoboTwin checkpoints on RoboTwin. Harness-VLA uses the same LIBERO-trained checkpoint on all three suites. The RynnVLA-002 results reflect a loader that reinitializes part of the VQ encoder, rather than an unmodified checkpoint deployment. JointWAM is excluded because no runnable checkpoint was available in the audited release. These results characterize the specified checkpoint–suite configurations.

Table 8: Success counts across three evaluation suites. **Full** measures successful completion of the entire task, whereas **G–L** measures whether the policy successfully grasps and lifts the target object, providing a partial-task measure of manipulation capability. Each entry reports the number of successful evaluations over the total number evaluated; the best result in each column is shown in bold. ⋆The π0.5-RLinf-SFT policy uses the same LIBERO-trained checkpoint across all three suites, without suite-specific adaptation. ‡For RynnVLA-002, the current public loader reinitializes part of the VQ encoder when loading the released checkpoint.

	LIBERO-Pro		RoboSuite		RoboTwin 2.0	
Method	Full	G–L	Full	G–L	Full	G–L
Vision-Language-Action Policies						
OpenVLA-OFT (Kim et al., 2025)	4/14	5/14	0/4	0/4	0/10	1/10
PRTS (Zhang et al., 2026c)	6/14	7/14	**1/4**	1/4	0/10	4/10
RDT2 (Liu et al., 2026b)	0/14	0/14	0/4	0/4	0/10	0/10
SpatialVLA (Qu et al., 2025)	0/14	0/14	0/4	0/4	0/10	1/10
StarVLA (StarVLA Community, 2026)	4/14	3/14	0/4	0/4	0/10	2/10
RynnVLA-002‡ (Cen et al., 2025)	1/14	3/14	0/4	0/4	0/10	1/10
EVO1 (Lin et al., 2026b)	4/14	3/14	0/4	0/4	0/10	3/10
ABot-M0 (Yang et al., 2026a)	4/14	3/14	**1/4**	0/4	7/10	9/10
InternVLA-A1.5 (Ma et al., 2026)	5/14	7/14	0/4	0/4	9/10	9/10
SmolVLA (Shukor et al., 2025)	3/14	4/14	0/4	0/4	0/10	3/10
WLA-0 (Yang et al., 2026b)	4/14	4/14	**1/4**	1/4	8/10	**10/10**
X-VLA (Zheng et al., 2026)	2/14	4/14	0/4	0/4	0/10	0/10
MolmoAct2 (Fang et al., 2026)	**10/14**	11/14	**1/4**	**2/4**	0/10	0/10
π0.5-RLinf-SFT⋆ (Physical Intelligence et al., 2025)	**10/14**	**13/14**	0/4	**2/4**	0/10	2/10
Video-Action Models						
LingBot-VA (Li et al., 2026b)	0/14	3/14	0/4	**2/4**	9/10	9/10
World-Action Models						
FastWAM (Yuan et al., 2026a)	4/14	11/14	0/4	0/4	**10/10**	**10/10**
FasterWAM (Zhao et al., 2026)	4/14	11/14	0/4	0/4	8/10	**10/10**

Table 9: Execution efficiency on LIBERO-Pro under the **Full** and **G–L** criteria. **Full** denotes end-to-end task success, while **G–L** denotes successful grasp and lift of the target object. For each criterion, **All trials** aggregates both successful and failed episodes and reports the average number of executed steps, average execution time, and effective control frequency. **Successful only** reports the mean and median completion time over successful episodes only. A dash indicates that the method achieves no successful trial under that criterion and therefore has no successful-only timing statistic. Lower all-trial steps or execution time do not necessarily indicate greater efficiency, since failed episodes may terminate before completing the full rollout horizon; these quantities should therefore be interpreted jointly with the corresponding success counts in Table 8.

	Full					G–L				
	All trials			Successful only		All trials			Successful only	
Method	Steps	Time (s)	Hz	Mean (s)	Median (s)	Steps	Time (s)	Hz	Mean (s)	Median (s)
Vision–Language–Action policies										
OpenVLA-OFT	547.4	34.2	6.171	7.8	7.5	503.1	117.0	6.368	41.0	35.6
PRTS	459.8	39.5	3.017	11.7	11.5	429.2	105.6	3.506	41.5	30.9
RDT2	720.0	126.6	0.885	—	—	720.0	287.1	0.771	—	—
SpatialVLA	720.0	135.4	1.608	—	—	720.0	390.7	1.074	—	—
StarVLA	542.4	48.0	2.338	12.9	12.8	580.5	144.9	3.871	29.0	31.4
RynnVLA-002[‡]	673.9	81.9	1.313	11.1	11.1	591.9	1005.2	0.873	191.5	145.9
EVO1	542.0	68.5	2.008	18.4	15.9	581.8	246.1	1.906	32.5	32.7
ABot-M0	540.9	119.4	2.392	25.3	24.6	580.1	168.8	2.402	42.2	41.5
InternVLA-A1.5	535.5	157.8	0.914	67.2	43.6	407.6	166.4	1.375	38.9	32.1
SmolVLA	590.1	113.3	3.175	29.9	28.4	557.0	242.9	2.360	55.5	43.4
WLA-0	541.6	138.5	1.192	34.6	32.1	533.4	235.7	1.522	27.3	25.6
X-VLA	654.7	134.8	3.269	71.1	71.1	533.6	223.2	6.201	38.2	40.2
MolmoAct2	301.1	56.3	3.649	28.5	26.6	213.6	89.0	2.841	41.7	40.6
π0.5-RLinf-SFT	303.8	60.4	4.876	30.8	29.5	128.9	30.9	4.870	23.1	20.0
Video–Action models										
LingBot-VA	720.0	354.7	2.315	—	—	625.7	708.1	1.622	364.9	227.6
World–Action models										
FastWAM	542.6	120.0	1.451	31.7	31.0	177.7	51.7	1.459	10.6	7.1
FasterWAM	542.0	115.5	2.201	28.3	27.5	177.3	47.3	1.835	10.7	6.5

Table 10: Execution efficiency on RoboSuite under the **Full** and **G–L** evaluation criteria. **Full** requires successful completion of the entire task, whereas **G–L** requires successful grasp and lift of the target object. For each criterion, **All trials** aggregates both successful and failed evaluations and reports the average number of executed steps, average execution time, and effective control frequency. **Successful only** reports the mean and median execution time computed only over evaluations satisfying the corresponding success criterion. A dash denotes that no evaluation is successful under that criterion, so successful-only timing statistics are unavailable; **NR** denotes a statistic that was not recorded. Lower all-trial step counts or execution times do not necessarily indicate higher execution efficiency, since failed evaluations may terminate early rather than complete the full rollout horizon. All-trial efficiency should therefore be interpreted jointly with the corresponding success counts reported in Table 8, where method references are also provided.

	Full					G–L				
	All trials			Successful only		All trials			Successful only	
Method	Steps	Time (s)	Hz	Mean (s)	Median (s)	Steps	Time (s)	Hz	Mean (s)	Median (s)
Vision–Language–Action policies										
OpenVLA-OFT	720.0	179.2	2.337	—	—	720.0	111.2	7.042	—	—
PRTS	575.5	160.0	1.779	37.6	37.6	561.5	94.5	3.092	18.2	18.2
RDT2	720.0	261.7	0.726	—	—	720.0	202.5	0.856	—	—
SpatialVLA	720.0	313.1	1.024	—	—	720.0	207.1	1.546	—	—
StarVLA	200.0	46.7	3.636	—	—	720.0	124.9	3.971	—	—
RynnVLA-002[‡]	220.0	192.3	1.478	—	—	720.0	590.4	1.462	—	—
EVO1	720.0	128.6	2.516	—	—	720.0	128.9	2.343	—	—
ABot-M0	576.8	100.1	3.739	26.2	26.2	720.0	87.7	3.754	—	—
InternVLA-A1.5	720.0	159.0	1.460	—	—	720.0	104.4	1.278	—	—
SmolVLA	720.0	164.4	*n.r.*	—	—	720.0	135.4	2.958	—	—
WLA-0	661.2	139.5	1.717	97.0	97.0	559.2	127.4	1.564	20.2	20.2
X-VLA	720.0	101.6	7.382	—	—	720.0	103.4	7.431	—	—
MolmoAct2	556.8	123.5	1.103	15.6	15.6	404.0	80.9	2.746	17.3	17.3
π0.5-RLinf-SFT	720.0	129.7	4.780	—	—	424.8	74.6	4.952	24.1	24.1
Video–Action models										
LingBot-VA	720.0	377.8	2.887	—	—	526.8	459.2	1.303	372.1	372.1
World–Action models										
FastWAM	720.0	166.5	1.456	—	—	720.0	160.5	1.451	—	—
FasterWAM	720.0	142.4	1.520	—	—	720.0	149.1	1.820	—	—

Table 11: Execution efficiency on RoboTwin 2.0 under the **Full** and **G–L** evaluation criteria. **Full** requires successful completion of the entire task, whereas **G–L** requires successful grasp and lift of the target object. For each criterion, **All trials** reports execution statistics computed over the complete evaluation set, including both successful and failed trials: **Steps**, **Time (s)**, and **Hz** characterize rollout length, execution time, and effective control frequency, respectively. **Successful trials** reports the mean and median execution time over trials satisfying the corresponding success criterion. A dash denotes that no trial is successful under that criterion, so successful-trial timing statistics are unavailable. Lower all-trial step counts or execution times do not necessarily indicate greater execution efficiency, since failed trials may terminate before task completion and therefore incur shorter rollouts. These efficiency statistics should therefore be interpreted jointly with the corresponding success counts reported in Table 8, where method references are also provided.

	Full					G–L				
	All trials			Successful trials		All trials			Successful trials	
Method	Steps	Time (s)	Hz	Mean (s)	Median (s)	Steps	Time (s)	Hz	Mean (s)	Median (s)
Vision–Language–Action policies										
OpenVLA-OFT	792.0	1273.4	3.879	—	—	675.6	1292.2	3.434	51.3	51.3
PRTS	790.0	1620.3	1.370	—	—	464.5	654.0	2.139	78.9	53.0
RDT2	792.0	1372.8	0.347	—	—	790.0	1359.1	0.334	—	—
SpatialVLA	790.0	2242.1	0.669	—	—	739.7	1565.8	0.797	234.1	234.1
StarVLA	790.0	708.5	3.938	—	—	594.7	621.8	3.321	96.4	96.4
RynnVLA-002	790.0	1343.2	0.905	—	—	734.8	1258.9	0.979	121.3	121.3
EVO1	806.6	641.9	1.096	—	—	535.2	399.3	1.626	111.7	61.3
ABot-M0	470.0	245.5	3.610	144.0	139.6	139.8	117.7	3.397	44.2	43.8
InternVLA-A1.5	326.0	188.0	1.534	162.1	153.9	144.8	78.7	1.511	40.7	42.5
SmolVLA	790.0	405.9	2.392	—	—	574.6	332.6	2.011	48.1	38.8
WLA-0	368.0	265.6	1.295	206.3	202.4	81.4	65.2	1.318	65.2	54.0
X-VLA	798.0	481.6	7.936	—	—	790.0	493.5	7.557	—	—
MolmoAct2	792.0	670.4	3.983	—	—	790.0	359.7	0.871	—	—
π0.5-RLinf-SFT	790.0	616.1	5.029	—	—	666.8	483.8	5.042	32.6	32.6
Video–Action models										
LingBot-VA	344.5	203.8	0.948	150.0	159.3	138.4	168.4	0.331	101.1	103.8
World–Action models										
FastWAM	284.8	164.1	0.835	164.1	153.6	73.2	40.0	1.453	40.0	38.7
FasterWAM	431.2	239.8	1.173	178.2	179.3	80.1	44.3	1.863	44.3	45.6

C Experiment Details

C.1 Benchmark Tasks

The evaluation suite comprises 100 tabletop-manipulation tasks across three benchmarks with different scene and control requirements: LIBERO-Pro (Zhou et al., 2025) (single-arm tasks under instruction-redirection **T** and position-swap **S** perturbations), RoboSuite (Zhu et al., 2020) (single-arm pick–place in clean scenes), and RoboTwin 2.0 (Chen et al., 2026d) (long-horizon bimanual manipulation). The task pool contains 80 LIBERO-Pro, 4 RoboSuite, and 16 RoboTwin 2.0 tasks. Coordinator training uses scene seed 0 for these tasks, while evaluation uses seeds 1–10. This split tests generalization to new scene seeds within the same task pool. LIBERO-Pro columns in Table 3 in the main text follow the eight T/S splits (Spat, Obj, Goal, and L10), each aggregating 10 tasks $\times$ 10 seeds; RoboSuite and RoboTwin columns aggregate their full task pools under the same seed range. Mobile-manipulation suites such as BEHAVIOR (Srivastava et al., 2021) and RoboCasa (Nasiriany et al., 2024) fall outside the P^5 tabletop-manipulation scope and are not included.

Cross-agent and cross-embodiment sensitivity. The study described in Takeaway 5 uses a fixed 3×4 backend–embodiment grid on four RoboTwin 2.0 task instances: `blocks_ranking_rgb` (arrange the red, green, and blue blocks), `place_cans_plasticbox` (place the cans into the plastic box), `place_bread_basket` (place the bread into the basket), and `stack_blocks_two`

(stack two blocks). Claude Code and Aloha–AgileX are the default backend and embodiment, respectively; the comparison swaps in Cursor or Codex, and Franka–Panda, ARX–X5, or UR5–WSG, while retaining the same task instructions, P^5 stage contracts, and typed API interfaces. Cursor and Claude Code both use Claude Opus 4.8 as the underlying generator, while Codex uses GPT-5.5. The backend sweep keeps the API library and trained coordinator fixed, whereas each embodiment uses its corresponding native adapter and compatible low-level controller under the shared API contract; client and decoding details are given in Appendix C.5. The supplementary videos provide one demonstration for each task–embodiment pair, giving 16 clips organized as `<task>/<embodiment>.mp4`. These clips illustrate execution on the different embodiments, while Figure 6 in the main text reports the aggregate success rates.

C.2 Baseline Implementations

All methods in Table 3 in the main text share the single-episode evaluation protocol of Appendix C.3: seeds 1–10, one trial per seed, terminal SR only, and *no* global cross-episode memory. For implementations that support memory across episodes, retrieval over prior tasks, or persistent tip stores, we disable those channels to keep the available experience limited to the current episode. RoboTwin columns marked "/" indicate embodiment mismatch for single-arm VLA checkpoints that are incompatible with the bimanual RoboTwin controllers.

End-to-end policies. π_0 (Black et al., 2025) and $\pi_{0.5}$ (Physical Intelligence et al., 2025) run as frozen open-loop policies under their released checkpoints for the LIBERO / RoboSuite control stacks; they are not re-fine-tuned on our coordinator training pool. FastWAM (Yuan et al., 2026a) and FasterWAM (Zhao et al., 2026) likewise use released world-action checkpoints as full-episode controllers (not as `Perform_E2E` routes). In RoboAware, the retained $\pi_{0.5}$ and FastWAM routes appear only inside typed `Perform_E2E` calls at local skill granularity (Appendix B.6).

Code-as-policy. CaP (Liang et al., 2023) and CaP-X (Fu et al., 2026) follow their public code-generation loops over language-conditioned robot APIs; we evaluate them under the shared episode horizon and without their optional cross-episode repair memory. RATS (Zhang et al., 2026b) and ASPIRE (Lu et al., 2026) keep their published planner / skill-library interfaces, again with global memory disabled. These baselines use their published execution-selection rules and do not receive the learned Q_θ family coordinator.

VLA with harness. Pigey (Galanti et al., 2026) and VLS (Liu et al., 2026a) use their public harness wrappers around a frozen generative policy, with early-stop and reward-steering settings taken from the authors' default configs under the shared single-episode protocol. Harness-VLA (Zhang et al., 2026d) is evaluated by us using the RLinf-released `pi05_libero130_fullshot` $\pi_{0.5}$-RLinf-SFT checkpoint (Yu et al., 2026), exposed through the VLA_ACT primitive; harness component ablations appear in Appendix D.1. Controlled routing ablations of RoboAware (RoboAware-UO, RoboAware-TF, RoboAware-UA) keep the same coding agent and API library and change only coordinator training or family allocation, as listed in Tables 4 and 5 in the main text.

C.3 Success and Runtime Metrics

This subsection defines the success-rate and runtime measures used in the main evaluation. Each table specifies its task pool, and comparisons within a table use the same success criterion and timing convention unless stated otherwise. The Harness-VLA ablation in Appendix D.1 uses a separate task and seed schedule and reports success among valid completions, together with execution coverage.

Success-rate denominator. An evaluation episode is one `(task, scene seed)` pair under seeds 1–10. The binary success indicator is the terminal task reward $r \in \{0, 1\}$ emitted by the suite's registered evaluator when the episode ends by task completion, unrecoverable failure, empty $\mathcal{C}_i$, or the horizon T; partial stage progress never counts as success. The episode-level success rate (SR) is therefore

$$\mathrm{SR}_{\mathrm{ep}} = \frac{1}{N} \sum_{n=1}^{N} \mathbf{1}[r_n = 1],$$

where N is the number of evaluated episodes in the stated pool (for example, N=100 for a 10 tasks $\times$ 10 seeds column in Table 3 in the main text, and N=1,000 for the full 100 tasks $\times$ 10 seeds suite). In the main evaluation, episodes that abort remain in the denominator, including crashes, timeouts, and failures caused by an empty admissible-family set.

Task- and suite-level weighting. To distinguish equal weighting of tasks from pooling individual episodes, we define two averages. *Task macro-average* first computes a per-task SR over that task's seeds, then averages those task rates with equal weight:

$$\mathrm{SR}_{\mathrm{macro}} = \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} \mathrm{SR}_{\mathrm{ep}}(t).$$

Episode-weighted (micro) average is $\mathrm{SR}_{\mathrm{ep}}$ over the pooled episodes and therefore weights tasks by their episode counts. Because every task in the main evaluation has the same number of seeds, the task macro-average and episode-weighted average coincide, both within each suite and across all 1,000 episodes. They differ only when tasks contribute unequal numbers of episodes. An unweighted average of the three suite-level rates is a different measure. It gives equal weight to LIBERO-Pro, RoboSuite, and RoboTwin despite their unequal task counts, whereas pooling all episodes gives each task equal weight under this protocol. We therefore identify whether each cross-suite result is an equal-weight suite average or a pooled episode average. Main-text columns use the episode-weighted measure specified in their captions.

Execution-time scope. Reported times are wall-clock seconds on the evaluation host unless a table names a different clock. *End-to-end episode time* starts at the post-reset observation used to build the task plan and stops when the episode terminates under the success predicate above; simulator reset and offline model loading are excluded. Averages over all trials include failures; successful-trial means or medians are computed only on episodes with r=1 and are marked undefined when that set is empty, matching Table 11.

C.4 Hyperparameters and Experimental Settings

Table 12 lists the reward weights, search and learning settings, and step budgets used by RoboAware. We distinguish four budgets because they count different parts of execution: *environment / control steps* advance the physics or low-level controller inside an API; an *API step budget* caps how long an atomic call may retain control before returning success or a typed failure; *semantic decision steps* count P^5 boundaries at which the coordinator may select a family; and the MCTS budget K counts *completed suffix simulations* per search root, not environment steps or semantic decisions. Root-collection and rollout suffixes use a fixed family-sampling rule with support on every admissible family; tree Selection uses UCB (Equation (1)) instead. The main experiments use K=20. The RoboSuite ablation compares $K \in \{5, 10, 20, 50\}$ and shows diminishing success-rate gains beyond K=20 (Figure 5 in the main text). RoboAware-UA uses uniform family allocation under the same K values.

C.5 Coding-Agent Backends

The coding agent π_{code} is frozen before coordinator training and is shared by RoboAware and the controlled routing baselines. Cursor, Claude Code, and Codex are the client toolchains through which the coding models are accessed. In the reported runs, Cursor and Claude Code both use Claude Opus 4.8, while Codex uses GPT-5.5. We fix the client/CLI versions and decoding settings and record them in the run logs. Sensitivity results in Figure 6 in the main text swap only this backend while keeping the API library, P^5 contracts, and Q_θ fixed.

C.6 Code-Execution Protocol

Execution sandbox, budgets, and retries. Generated blocks run in a restricted Python process that can import only the registered API library and allowlisted Python standard-library helpers. Each block inherits an API-call budget, a control-step budget, and a 30 s wall-clock timeout (see Table 12). Syntax errors, undefined names, disallowed imports, and API schema violations trigger an automatic repair round that appends the error text and regenerates under the same family constraint,

Table 12: Hyperparameters of RoboAware.

Parameter	Symbol	Setting
Discount factor	γ	1
Perception reward weight	w_{p}	0.2
Manipulation reward weight	w_{m}	0.3
Task-success reward weight	w_{t}	0.5
Smooth-L_1 threshold	—	1
Neighboring-root draws	S	4
Neighboring-time window	Δ	10 saved control steps
UCB coefficient	c_{ucb}	1.0
Suffix-simulation budget	K	20
Episode horizon	T	100 decision transitions
Q-learning updates	M	10,000
Learning rate	η	10^{-5}
Target-copy interval	J	500 updates
Translational pose threshold	ϵ_p	0.02 m
Rotational pose threshold	ϵ_R	$\pi/12$ rad
Atomic API step budget	—	100 control steps
Block wall-clock timeout	—	30 s
Code-repair attempts	R_{repair}	2
Coding-agent history window	H_{code}	4 full stage transitions

up to R_{repair}=2 attempts; exhausted repairs mark the branch as failed for SCB or terminate the deployment stage according to Table 1. Runtime exceptions raised inside an allowed API are returned as typed failure codes to the agent rather than as raw simulator dumps.

Episode reset versus memory. Between evaluation episodes (and between independent training trajectories), the coding-client session is reset: conversation state, tool caches, and any product-level memory are cleared so that episode $n{+}1$ cannot read traces from episode n. Within an episode, feedback is limited to the observable channel already present in x_i: typed API returns, postcondition bits, failure codes, timeouts, and the truncated history h_i. We do *not* enable cross-episode product memory, retrieval over prior tasks, or persistent "global tips" stores for main results; any ablation that reintroduces such memory is labeled separately and is not part of the default RoboAware protocol. Family selection at deployment is supplied by Q_θ; the coding agent does not choose c_i, and prompts never reveal the unchosen family's counterfactual outcome.

C.7 Computation and Software

The reported coordinator run used four NVIDIA A100 GPUs with 80 GB memory each. Fitting Q_θ took approximately six hours, corresponding to approximately 24 GPU-hours (4×6 h). Trajectory and counterfactual data collection took approximately one additional day on the same hardware pool. Data collection therefore accounts for most of the wall-clock time and must be included when reporting the training cost. Supplementary qualitative clips for the cross-embodiment study follow the naming `<task>/<embodiment>.mp4` described in Appendix C.1.

D Additional Empirical Results

D.1 Ablation of Harness-VLA

Experimental setup. We evaluate the harness components in two stages. All six configurations in Table 13 use the same $\pi_{0.5}$ SFT backbone and a common neutral interface prompt. We first screen them on 24 tasks with seed 42, then evaluate Full, w/o Memory, and w/o Memory & SAM3 on 80 tasks with seeds $\{41, 42, 43\}$. Removing Memory also disables the seed-0 reference, so we use *memory package* to denote these two components together. The w/o Reference variant removes

Table 13: Component configurations for the Harness VLA ablation study. All variants use the same $\pi_{0.5}$ SFT backbone.

Method	Planner	Memory	Seed-0 reference	SAM3	Scripted hybrid	VLA
A: Full Harness VLA	✓	✓	✓	✓	✓	$\pi_{0.5}$ SFT
B: w/o Memory	✓	×	×	✓	✓	$\pi_{0.5}$ SFT
C: w/o Reference	✓	✓	×	✓	✓	$\pi_{0.5}$ SFT
D: w/o SAM3	✓	✓	✓	×	✓	$\pi_{0.5}$ SFT
E: w/o Memory & SAM3	✓	×	×	×	✓	$\pi_{0.5}$ SFT
F: Bare SFT	×	×	×	×	×	$\pi_{0.5}$ SFT

Table 14: Paired ablation screening on 24 tasks using seed 42. Drop is defined as the success rate of Full minus that of the corresponding variant, in percentage points (pp). Confidence intervals are 95% task-bootstrap intervals for the paired drop. S→F counts tasks solved by Full but failed by the ablated variant; F→S counts the reverse. All 24 episodes completed without errors in each variant.

Method	Success / valid	Success rate (%)	Drop (pp)	95% CI (pp)	S→F	F→S
A: Full Harness VLA	15/24	**62.5**	–	–	–	–
B: w/o Memory	11/24	45.8	16.7	$[-8.3,\ 41.7]$	7	3
C: w/o Reference	12/24	50.0	12.5	$[-12.5,\ 37.5]$	6	3
D: w/o SAM3	14/24	58.3	4.2	$[-8.3,\ 20.8]$	2	1
E: w/o Memory & SAM3	9/24	37.5	**25.0**	$[4.2,\ 45.8]$	7	1
F: Bare SFT	10/24	41.7	20.8	$[0.0,\ 41.7]$	6	1

only the reference while retaining Memory, allowing us to examine that component separately in the screening stage.

Single-seed screening. Full succeeds on 15 of 24 tasks (62.5%; Table 14). Removing both the memory package and SAM3 produces the largest observed decrease, to 37.5%. This is a drop of 25.0 percentage points (pp), with a 95% task-bootstrap confidence interval of $[4.2, 45.8]$ pp. Removing the memory package alone, the seed-0 reference alone, or SAM3 alone reduces success by 16.7, 12.5, and 4.2 pp, respectively, although all three confidence intervals include zero. The larger drop from joint removal does not by itself establish a positive interaction. The estimated additive interaction, $\mathrm{SR}_A - \mathrm{SR}_B - \mathrm{SR}_D + \mathrm{SR}_E$, is -4.2 pp and provides no evidence of a positive additive interaction.

Bare SFT achieves 41.7%, which is 20.8 pp below Full. Because this variant also removes the planner and scripted hybrid execution, its comparison with Full measures the contribution of the overall harness rather than the planner alone.

Multi-seed confirmation. Table 15 separates execution errors from completed episodes in which the task fails. Among valid completions, Full achieves 56.6% success, compared with 16.9% for w/o Memory and 20.9% for w/o Memory & SAM3. Because execution errors leave different sets of valid episodes for each variant, we also compare variants on task–seed pairs that completed under both configurations. On these paired subsets, removing the memory package yields a signed difference of -38.9 pp ($n = 113$, 95% CI $[-50.0, -27.4]$ pp), while removing both the memory package and SAM3 yields -35.8 pp ($n = 109$, 95% CI $[-48.2, -23.4]$ pp). Both intervals exclude zero, supporting a contribution from the memory package on the jointly completed episodes.

Scope and limitations. Service-connection errors leave only 47.1–49.2% of the 240 scheduled episodes per configuration with valid completions. The paired estimates therefore describe those

jointly completed episodes and cannot establish the effect over the full scheduled set. We report execution coverage separately and do not count these service errors as model failures in this ablation. The confirmation stage also omits w/o SAM3, so it does not independently measure the standalone contribution of SAM3. The historical whole-system result of 65/80 (81.25%) used a different prompt and protocol and is therefore not directly comparable with Full in this study.

D.2 Visualizations

Figure 1 shows further qualitative rollouts in which responsibility switches between the two families at P^5 boundaries, extending the cases in Figure 4 in the main text.

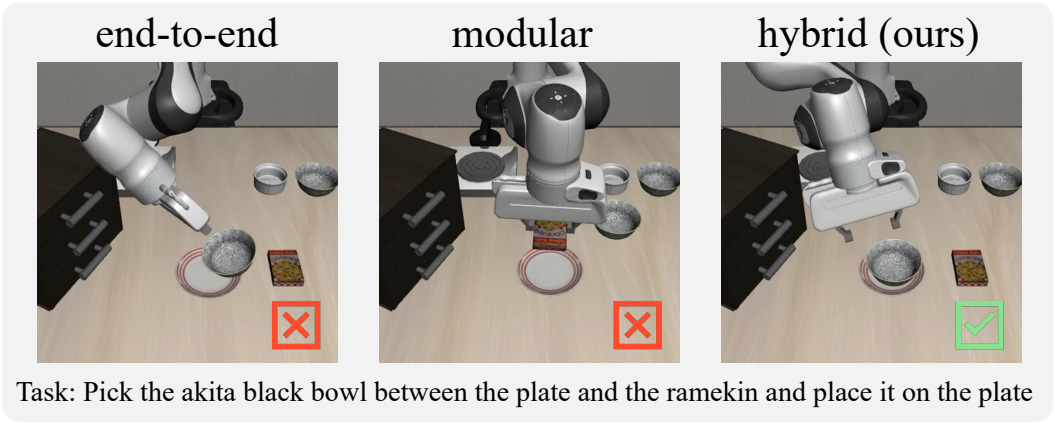


Figure 1: Additional representative cases of family handoffs under P^5 boundaries, complementing Figure 4 in the main text.

D.3 Failure Modes

Figure 2 summarizes the failures observed in LIBERO-Pro and RoboTwin. Grasping and object-holding instability is the largest identified category, accounting for 29.8% of failed rollouts. Constrained contact, insertion, and mechanism actuation account for another 16.7%. Together, these categories represent 46.5% of all failed rollouts, indicating that low-level interaction and contact-rich execution are major sources of failure. Target grounding (5.3%), transport and placement (6.1%), and multi-object sequencing (4.4%) account for smaller identified shares.

However, 37.7% of failures remain in *Others* because the saved evaluation records lack the detailed traces needed to distinguish perception, planning, control, and success-check errors. This limits how strongly the identified categories can characterize the full failure distribution. The categories summarize the observed outcomes. During deployment, recovery and termination instead follow the typed API failures, timeouts, and stage conditions defined in Table 1 and Algorithm 1.

Table 15: Multi-seed confirmation on 80 tasks with seeds $\{41, 42, 43\}$, yielding 240 scheduled episodes per variant. Valid SR is the number of successful episodes divided by the number of successful and failed episodes, so execution errors are excluded. Coverage is the fraction of the 240 scheduled episodes that completed without errors. Paired differences compare each variant against Full on task–seed pairs that completed without errors in both conditions: $\Delta = \mathrm{SR}_{\mathrm{variant}} - \mathrm{SR}_{\mathrm{Full}}$ on that jointly valid subset of size n_{pair}, which need not equal the gap between the separately reported valid success rates.

Method	Success	Failure	Error	Valid / scheduled	Valid SR (%)	Coverage (%)	n_{pair}	Paired Δ (pp)	95% CI (pp)
A: Full Harness VLA	64	49	127	113/240	**56.6**	47.1	–	–	–
B: w/o Memory	20	98	122	118/240	16.9	49.2	113	**−38.9**	[−50.0, −27.4]
E: w/o Memory & SAM3	24	91	125	115/240	20.9	47.9	109	**−35.8**	[−48.2, −23.4]



Figure 2: Failure modes of the unsuccessful rollouts. Grasping instability is the largest identified failure mode (29.8%), followed by constrained contact or insertion failures (16.7%). Grounding, placement, and multi-object sequencing account for 5.3%, 6.1%, and 4.4%, respectively. *Others* (37.7%) denotes failed rollouts for which the saved evaluation records are insufficient to identify a specific cause (29.8% from LIBERO-Pro and 7.9% from RoboTwin).

E Social Impact

RoboAware uses execution outcomes to learn when to select each family of robot skills. Its explicit stage-level choices make it possible to inspect which family was selected and how execution proceeded. This may help developers adapt robotic assistants to different tasks and investigate failures. Deployment in human environments still requires task-specific validation, failure monitoring, and appropriate human oversight.