REFINING PROGRAMMATIC RUNTIME POLICIES FOR EFFICIENT CONTINGENCY ADAPTATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Robotic systems operating in dynamic environments require continual adaptation of coordination among reusable perception and control capabilities in response to concurrent events during execution. By bringing general-purpose programming abstractions to physical execution, programmatic policy frameworks provide an executable layer for orchestrating these capabilities into behavior, with an explicit organization that can be reconfigured using execution feedback. However, contingencies can alter runtime coordination among concurrently executing skills, while coordination embedded within task logic makes the refinement target ambiguous between skill behavior and coordination and leads to repeated recovery and validation across task compositions. We propose `RaP`, a *compile–execute–refine* framework that externalizes runtime coordination from generated task programs as a persistent refinement target and represents uncertainty over skill behavior and coordination as hypotheses. At *compile* time, `RaP` converts skill invocations into runtime-managed processes and augments the task program with conditional tests of these hypotheses. During *execution*, the runtime enforces coordination relations and records hypothesis-test outcomes, while *refinement* uses this evidence to target revisions to skill behavior or coordination conditions and distills validated repairs into reusable guidance with explicit applicability conditions. On long-horizon multi-arm manipulation tasks under concurrent events that alter coordination requirements, `RaP` achieves higher mean held-out success than task-program repair after three refinement rounds.

1 INTRODUCTION

Adaptive robot systems increasingly require a programmable interface that can construct behavior from task knowledge and revise it through interaction with changing environments. Code-as-policy (CaP) paradigm grounds general-purpose programming knowledge in embodied behavior, with coding agents driving a cycle of program synthesis, execution, and refinement that supports sustained improvement through an explicit and maintainable programmable policy layer (Liang et al., 2023; Elmaaroufi et al., 2026). In this loop, agents *compile* task and environment knowledge into executable programs by composing reusable primitives (Fu et al., 2026). They then *execute* the resulting programs at runtime, coordinating perception and action through structured execution and the orchestration of learned policies (Chen et al., 2026b;c). Finally, agents *refine* this logic by diagnosing execution failures and validating corrected behaviors, retaining reusable skills for subsequent tasks (Lu et al., 2026).

At execution time, multiple events can occur concurrently in dynamic environments and affect ongoing robot behaviors. Different combinations and arrival times of concurrent events can generate a combinatorially large set of contingencies during execution. Each contingency may require a responsive behavior whose execution must be coordinated with behaviors already in progress. For example, while carrying an object, a robot may need to avoid an approaching person while preserving the support and stability required for transport, and changes in the person's motion or additional obstacles may further alter the feasible response. These execution-time interactions motivate a runtime layer that instantiates generated behaviors as processes and manages their evolving execution and coordination.

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

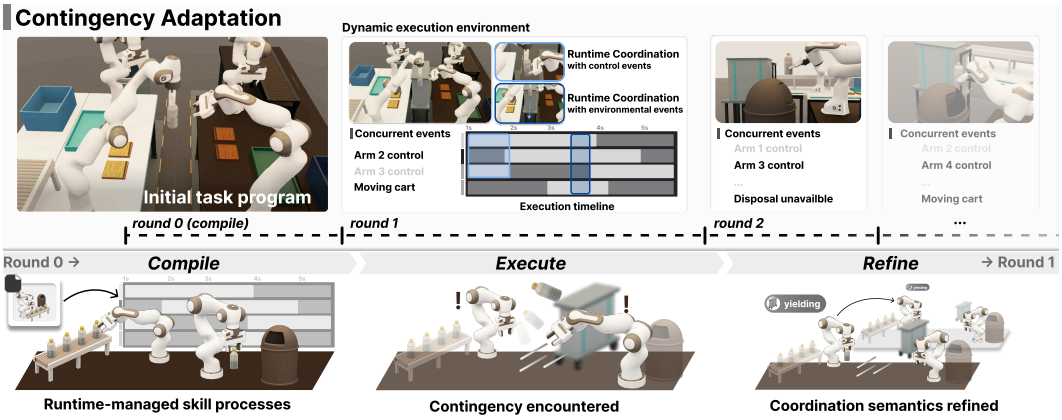

Figure 1: *(top)* Successive *compile–execute–refine* rounds in which concurrent events induce coordination-relevant contingencies that change which skill events may proceed together. *(bottom)* By considering both skill behavior and runtime coordination semantics, `RaP` adapts how ongoing behaviors are coordinated under contingencies and accumulates runtime coordination knowledge for reuse as new contingencies arise.

However, incorporating such a runtime into the program refinement loop introduces two problems: representing execution knowledge for runtime composition and retaining that knowledge through refinement. (i) At compilation, execution relations remain implicit in the generated program, while runtime composition requires these relations independently of any single task-specific control structure. (ii) At refinement, execution provides corrections to these relations, while storing them in the program or its task-specific execution structure binds the corrections to the composition in which they were learned.

We propose `RaP`, **R**untime-**a**daptive **P**olicy refinement, a *compile–execute–refine* framework that adapts to emerging contingencies by making runtime coordination an explicit, persistent refinement target. Runtime coordination determines when skills may start, overlap, pause, or resume in response to other ongoing skills and environmental events. At *compilation*, `RaP` represents runtime coordination rules separately from program control flow and generates hypotheses about skill behavior and runtime coordination. During *execution*, the runtime applies these rules to ongoing skill processes and collects evidence to test the hypotheses. *Refinement* uses this evidence to localize repairs to skills or runtime coordination rules and distills validated repairs into reusable knowledge with explicit applicability conditions. This knowledge accumulates across refinement rounds, supporting adaptation as new contingencies arise and reuse when skills participate in new compositions. Figure 1 illustrates how `RaP` progressively adapts to an expanding set of contingencies by refining and reusing runtime coordination knowledge.

We evaluate `RaP` progressively from runtime compilation through incremental contingency adaptation to real-world transfer and compare it with three baselines with different refinement targets within a common *compile–execute–refine* framework. We observe that runtime compilation can expose concurrent execution from sequentialized programs, while reliably translating task semantics into multiple coordinated runtime events remains sensitive to compiler capability and reasoning effort. After three refinement rounds, `RaP` achieves a mean held-out success rate of 53.3% across the event-information settings, compared with 41.7% for the strongest baseline of task-program repair. We also evaluate `RaP` using a real-world bimanual handover case to examine whether the inferred coordination condition can be reused across skills under the same contingency.

Our contributions are as follows:

- We externalize runtime coordination from program control flow as a persistent refinement target for contingency adaptation.
- We use skill and runtime coordination hypotheses to localize repairs and accumulate reusable runtime coordination knowledge.
- We demonstrate incremental contingency adaptation and transfer of runtime coordination rules to unseen skill contexts in multi-arm manipulation.

2 Related Work

Code-based Policy Synthesis and Refinement. The code-as-policy paradigm uses executable programs to compose reusable perception and control primitives into embodied behavior (Liang et al., 2023; Singh et al., 2023). Recent work brings this programmable interface closer to physical execution through finer-grained control (Burns et al., 2024; Fu et al., 2026), skill orchestration (Chen et al., 2026c), and policy improvement from execution feedback (Xiao et al., 2026). Subsequent refinement frameworks broaden what is retained and updated across interaction, with ASPIRE refining agentic skills, including reusable behaviors and repair knowledge (Lu et al., 2026), GaP revising computation graphs of modular skills (Chen et al., 2026b), and RHO selecting mutations over the executable agent repository (Elmaaroufi et al., 2026). Across these approaches, refinement is organized around the representation each framework exposes. This organization naturally retains execution knowledge within the representation being optimized, while contingencies can change how the same skills must interact, bringing into focus how learned coordination can be reused efficiently across such changes.

Concurrent skill composition. In physical environments, multiple behaviors may execute concurrently while exogenous events change the conditions of ongoing behavior. Coordination therefore determines when each behavior may proceed and how shared resources are used (Colledanchise & Natale, 2022; Liang et al., 2026). Temporal task models learn action dependencies for coordinated manipulation (Dreher & Asfour, 2024; Dreher et al., 2026), while task and motion planning determines feasible temporal and resource relations before execution (Zhang et al., 2025; Garrett & Ramos, 2025). Execution frameworks preserve planned coordination while allowing independent actions to progress asynchronously as their durations vary (Huang et al., 2025). When contingencies change the coordination required during execution, runtime approaches instead resolve these relations from live execution state. They recompute skill compositions as conditions change (Pane et al., 2021) or arbitrate shared resources among stateful processes (Chen et al., 2026a). We explore whether coordination resolved at runtime can also serve as a persistent refinement target, so that execution evidence can improve coordination beyond the composition in which it was observed.

Representations for runtime coordination. Finite state machines and behavior trees organize reactive execution through explicit transitions and compositional control-flow nodes, exposing tradeoffs between reactivity and modularity that affect maintenance as task complexity grows (Iovino et al., 2023; 2025). Concurrent extensions expand these representations to coordination, making progress synchronization and resource exclusion explicit in tree execution (Colledanchise & Natale, 2022) and coordinating multiple trees through cross-tree expansion and intention sharing (Cai et al., 2025a). The construction of these execution representations is increasingly automated, through direct language-model generation (Izzo et al., 2024; Ao et al., 2025) or heuristic guidance for symbolic planning (Cai et al., 2025b). They are also adapted from feedback by expanding trees with newly identified preconditions (Styrud et al., 2025) or revising the action models and control policies underlying execution (Cai et al., 2026). Computation graphs encode modular skill execution in explicit structure and support iterative graph updates for task-specific policy refinement (Chen et al., 2026b). However, contingency-driven repair remains expressed through the representation being adapted, limiting the direct reuse of coordination knowledge across compatible skill compositions.

3 Methods

We propose `RaP`, a *compile–execute–refine* framework for refining runtime coordination by managing skill invocations from generated task programs as processes. To make this coordination an explicit refinement target, we formulate execution as a concurrent event process and factor its policy into control skills and coordination semantics, providing a localized target for coordination adaptation under contingencies.

At *compile* time, `RaP` first generates skill and runtime coordination hypotheses from the provided task program and the perception and control interfaces of the robot, informed by accumulated memory of validated repairs and recoveries. It then compiles the program into a task policy for asynchronous runtime execution, with conditional hypothesis tests. *Execution* records observations, actions, and hypothesis-test outcomes from the compiled policy. *Refinement* updates hypothesis evidence, repairs and validates the task program, and distills reusable knowledge into memory.

3.1 PROBLEM FORMULATION

Event. We model the embodied agent and its environment as a discrete event system, a generalized semi-Markov process $\mathcal{M}$ (Glynn, 1989) over physical time $t \in \mathbb{R}_{\geq 0}$, where multiple events may remain enabled at once and race to change the state,

$$\mathcal{M} = (\mathcal{S},\ \mathcal{E},\ \Gamma,\ G,\ r,\ P)\,. \tag{1}$$

At the event level, this execution is represented by a state $s \in \mathcal{S}$ that changes only when an event $e \in \mathcal{E}$ occurs. For an event e, the enabling function $\Gamma(s) \subseteq \mathcal{E}$ determines whether e is available in s, and the distribution $G_e \in G$ delays it by a remaining time $c_e \in \mathbb{R}_{\geq 0}$ when it enters $\Gamma(s)$. The remaining time c_e counts down in s when the speed $r(s,e) \in \{0,1\}$ is 1 and stays suspended when it is 0. The events whose remaining times reach zero together form the triggering set $\mathcal{E}^* \subseteq \Gamma(s)$ and change the state according to the next-state distribution $p_{\mathcal{E}^*}(\cdot \mid s) \in P$. When triggering events change the state to s', every other event stays enabled with c_e unchanged or is dropped from $\Gamma(s')$.

For example, the state s_t at time t enables three events $\{e_1, e_2, e_3\}$ and evolves as

$$[s_t,\ \Gamma(s_t) = \{e_1, e_2, e_3\}] \ \xrightarrow[\mathcal{E}^* = \{e_1\}]{c_{e_1}} \ \left[s_{t+c_{e_1}},\ \Gamma(s_{t+c_{e_1}}) = \{e_2\}\right]. \tag{2}$$

Here, assume $r(s_t, e_i) = 1$ for all three events and $c_{e_1} < c_{e_2} < c_{e_3}$, so e_1 triggers first after c_{e_1} units of physical time. After the transition, only e_2 remains enabled with its remaining time $c_{e_2} - c_{e_1}$ preserved, while $e_3 \notin \Gamma(s_{t+c_{e_1}})$ and is dropped.

Skill. To model control over event activation, we follow a generalized semi-Markov decision process formulation (Younes & Simmons, 2004), with exogenous events $\mathcal{E}_{\mathrm{x}}$ and controllable events $\mathcal{E}_{\mathrm{c}}$, and extend it with a skill set Ω, where each skill $\omega \in \Omega$ is represented by a set of controllable events $\mathcal{E}_\omega$,

$$\mathcal{E} = \mathcal{E}_{\mathrm{x}} \cup \mathcal{E}_{\mathrm{c}}, \qquad \mathcal{E}_\omega \subseteq \mathcal{E}_{\mathrm{c}}, \quad \omega \in \Omega. \tag{3}$$

The exogenous events $\mathcal{E}_{\mathrm{x}}$ occur independently of control, whereas the controllable events $\mathcal{E}_{\mathrm{c}}$ admit an enable-or-disable decision when available in the current state.

Objective. For a task τ, the runtime objective is to maximize the expected return J_τ with a fixed skill set Ω by composing its skills during execution,

$$J_\tau(\Omega, \pi_\Omega) = \mathbb{E}\Big[\int_0^{T_\tau} R_\tau(s_t)\, dt\Big], \qquad J_\tau^*(\Omega) = \max_{\pi_\Omega}\ J_\tau(\Omega, \pi_\Omega). \tag{4}$$

The *runtime* is a policy $\pi_\Omega : \mathcal{S} \to 2^{\mathcal{E}_{\mathrm{c}}}$ that maps each state s to a subset $\pi_\Omega(s) \subseteq \Gamma(s) \cap \mathcal{E}_\Omega$ of the events $\mathcal{E}_\Omega = \bigcup_{\omega \in \Omega} \mathcal{E}_\omega$ of the skills in Ω that are available in s, so that the speed of a controllable event e is $r(s,e) = 1$ if $e \in \pi_\Omega(s)$ and 0 otherwise. A task τ specifies an initial state s_0, a horizon T_τ, and a reward $R_\tau(s)$ per unit of physical time.

Across K refinement rounds, `RaP` jointly updates the skill library $\boldsymbol{\Omega}^*$ and task-specific policies $\boldsymbol{\pi}^*$ to maximize cumulative task return,

$$(\boldsymbol{\Omega}^*, \boldsymbol{\pi}^*) \in \arg\max_{\boldsymbol{\Omega}, \boldsymbol{\pi}} \sum_{k=1}^{K} \mathbb{E}_{\tau \sim \mathcal{T}_k}\left[J_\tau(\Omega_k, \pi_{\tau,\Omega_k})\right], \tag{5}$$

where $\boldsymbol{\Omega} = (\Omega_1, \ldots, \Omega_K)$, $\boldsymbol{\pi} = (\{\pi_{\tau,\Omega_k}\}_\tau)_{k=1}^K$, and $\mathcal{T}_k$ denotes the task distribution at round k.

We implement this event-driven execution architecture with a runtime kernel that asynchronously dispatches primitive calls according to the active runtime policy, as detailed in Appendix A.7.

3.2 COMPILE

Runtime coordination semantics. At compile time, `RaP` converts each skill call into a *skill process*, exposing its controllable events to the runtime separately from the task-goal progression of the source program. For the active skill processes, an admissibility predicate κ determines whether a joint selection of controllable events may proceed in the current state. We define the *runtime coordination semantics* of a task τ as the estimate $\hat{\kappa}_\tau$ of κ that `RaP` compiles and the runtime enforces,

$$\kappa,\ \hat{\kappa}_\tau : \mathcal{S} \times 2^{\mathcal{E}_{\mathrm{c}}} \to \{0,1\}, \qquad \pi_\Omega(s) \in \{\mathcal{E}' \subseteq \Gamma(s) \cap \mathcal{E}_\Omega \mid \hat{\kappa}_\tau(s, \mathcal{E}') = 1\}\,. \tag{6}$$

Here, s includes task progress and the execution history needed by coordination decisions. We call a transition from s to s' a *coordination-relevant contingency* when the participating skills remain available but the admissibility of a joint selection of their controllable events changes, $\kappa(s, \mathcal{E}') \neq \kappa(s', \mathcal{E}')$ for some joint event selection $\mathcal{E}'$ available in both states. This isolates changes in which controllable events of the same participating skills may proceed jointly from changes in which skills are available. The runtime coordination semantics $\hat{\kappa}_\tau$ agree with κ across such a contingency only when their conditions anticipate the change. `RaP` therefore externalizes the definitions of $\hat{\kappa}_\tau$ from task-specific control flow and retains them as persistent refinement targets of the task.

Task compilation. To assess the validity of the compiled runtime coordination semantics $\hat{\kappa}_\tau$ under contingencies, uncertainty over skill behavior and these semantics is represented as *hypotheses* $h \in \mathcal{H}_\tau$ that are evaluated at runtime from hypothesis-specific information derived from the current state $s_t \in \mathcal{S}$. For a task τ, the compiler constructs

$$\mathcal{H}_\tau = \mathcal{H}_\tau^\Omega \cup \mathcal{H}_\tau^\kappa, \qquad z_{h,t} = m_h(s_t) \in \mathcal{Z}_h, \qquad h(s_t) = \varphi_h(z_{h,t}) \in \{0, 1, \bot\}. \tag{7}$$

Here, $\mathcal{H}_\tau^\Omega$ contains hypotheses over skill execution and $\mathcal{H}_\tau^\kappa$ contains hypotheses that the conditions of $\hat{\kappa}_\tau$ agree with κ. The hypothesis-specific evaluation domain $\mathcal{Z}_h$ contains the possible evaluation contexts, with $z_{h,t}$ extracted from the runtime state by $m_h : \mathcal{S} \to \mathcal{Z}_h$ and evaluated by $\varphi_h : \mathcal{Z}_h \to \{0, 1, \bot\}$, where $\bot$ denotes insufficient information to evaluate the condition.

3.3 Execution and Refinement

Each refinement round k updates the compiled policy $\theta_{\tau,k}$ of every task τ in three steps,

$$\theta_{\tau,k} \xrightarrow{\text{execute}} \mathcal{D}_{\tau,k} \xrightarrow[\mathcal{K}_k]{\text{revise}} \theta'_\tau \xrightarrow{\text{admit}} (\theta_{\tau,k+1},\, \mathcal{K}_{k+1})\,, \tag{8}$$

where compilation produces $\theta_{\tau,0}$ and $\mathcal{K}_k$ is the knowledge shared across tasks. Here, $\theta_{\tau,k} = (c_{\tau,k}, \hat{\kappa}_{\tau,k}, \mathcal{H}_{\tau,k})$, where $c_{\tau,k}$ contains the task program and skill implementations at round k.

Execution evidence. Execution gathers evidence to guide the next refinement of $\theta_{\tau,k}$. For each development case i, the policy runs without agent calls and produces a task outcome y_i and an execution trace ξ_i. The trace records command outcomes and coordination transitions, showing how skill processes advance or wait as the runtime enforces $\hat{\kappa}_{\tau,k}$. The policy also tests hypotheses at designated program points or in response to runtime events, using perception when needed. Each test records the hypothesis h, its evaluation context $z_{h,t}^{(i)}$, and the verdict $h(s_t^{(i)})$ from equation 7. An unavailable or non-Boolean result yields $\bot$. Tests that never run contribute no observation. Each observation remains linked to the executable that produced it, and any response to a verdict is explicitly encoded in $\theta_{\tau,k}$. The task outcomes, execution traces, and hypothesis observations together form the evidence passed to refinement,

$$\mathcal{D}_{\tau,k} = \left\{ \left(y_i,\, \xi_i,\, \left\{ \left(h, z_{h,t}^{(i)}, h(s_t^{(i)}) \right) \right\}_{(h,t)\in\mathcal{I}_i} \right) \right\}_i, \tag{9}$$

where $\mathcal{I}_i$ contains the pairs (h, t) for tests actually invoked in case i.

Hypothesis and policy refinement. Revision reads $\mathcal{D}_{\tau,k}$ in one coding agent session per task. Using this evidence, the agent may revise or add hypotheses in $\mathcal{H}_{\tau,k}^\Omega$ and $\mathcal{H}_{\tau,k}^\kappa$, yielding the candidate hypothesis set $\mathcal{H}'_\tau$. For each testable $h \in \mathcal{H}'_\tau$, it specifies or revises the context extraction m_h and evaluation φ_h of equation 7; hypotheses without an available test carry an explicit reason for remaining unobserved. These changes are made jointly with revisions to $c_{\tau,k}$ and $\hat{\kappa}_{\tau,k}$, producing $\theta'_\tau = (c'_\tau, \hat{\kappa}'_\tau, \mathcal{H}'_\tau)$. Within the refinement budget, the agent may execute θ'_τ on selected development cases and use the resulting outcomes and hypothesis observations to guide further revision.

Admitted candidates become $\theta_{\tau,k+1}$, with changes to executable behavior required to improve development performance without reducing the number of validation successes. Following prior work (Lu et al., 2026), admitted repairs that increase development successes may contribute reusable advice to $\mathcal{K}_{k+1}$ for subsequent revisions. Appendix A.6 details admission and knowledge retention.

4 Experiments

We evaluate runtime coordination as a persistent, reusable refinement target in the *compile–execute–refine* loop through three research questions:

RQ1. What factors affect runtime compilation quality, and how does compilation change concurrency and task performance?

RQ2. Can `RaP` efficiently refine persistent runtime coordination semantics as new coordination-relevant contingencies are introduced incrementally across refinement rounds?

RQ3. Can `RaP` infer coordination conditions from an explicitly provided contingency case and generalize them to situations beyond the observed one?

4.1 Experimental Setup

Environment. To evaluate behavioral preservation after runtime compilation, we use BiCoord (Peng et al., 2026), a long-horizon bimanual manipulation benchmark with tight spatial-temporal coordination. For contingency adaptation, we construct Quadmanual, a four-arm Franka manipulation environment in Isaac Lab (Mittal et al., 2023) on NVIDIA Isaac Sim, with multiple observable realizations of coordination-relevant contingencies. These variations alter how a contingency is observed while preserving the coordination requirement induced by the task.

Baselines. We compare `RaP` against three baselines with different refinement targets within *compile–execute–refine* frameworks. `Agentic skill (AS)` updates agentic skills with meta-level repair knowledge (Lu et al., 2026), `Skill graph (SG)` revises computation graphs of modular skills (Chen et al., 2026b), and `Behavior tree (BT)` refines behavior trees compiled from the same task policy programs. All methods share the same source task programs and primitive control and perception interfaces within each comparison.

Metrics. We measure Success Rate (SR) over task or subtask episodes, depending on the evaluation. Simultaneous Movement Percentage (SMP) is the fraction of a trajectory with both arms active (Peng et al., 2026). Details of the environment setup are in Appendix B.

4.2 Effects of Runtime Compilation

For **RQ1**, we compare task success before and after runtime compilation of synchronized bimanual programs across 18 BiCoord tasks, varying the compiler model and reasoning effort. We also compile sequentialized variants to measure changes in simultaneous arm motion and task success.

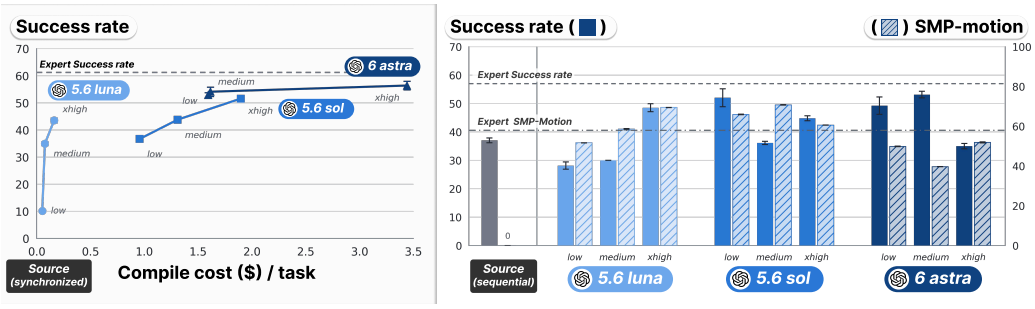

Figure 2: *(left)* Synchronized bimanual programs. Task success versus compilation cost per task across compiler models and reasoning efforts. *(right)* Sequentialized variants. Task success (left axis) and simultaneous arm motion (right axis), with sequential-source bars and synchronized-source reference lines. Compiled-program results are averaged over three execution repetitions, with error bars indicating standard deviations.

Compiler Model and Reasoning Effort. In Figure 2 *(left)*, task success after runtime compilation improves with more capable compiler models and higher reasoning effort. This dependence arises in part because the compiler must preserve task logic while satisfying the event and process semantics of the runtime, coupling task-level intent with runtime execution constraints in the generated code. Most execution failures and performance losses arise from generated code that violates runtime execution requirements. The resulting execution failures reduce the success rates of affected tasks to zero, lowering the aggregate success rate.

Concurrency recovery. In Figure 2 *(right)*, runtime compilation recovers simultaneous arm motion from sequentialized bimanual task programs, allowing skill executions to overlap beyond the ordering imposed by the source control flow. The compiled coordination semantics govern this overlap by specifying when each skill may proceed and which dependencies must hold among concurrent skills. However, task success does not consistently improve over the sequential source and remains below the synchronized source reference, suggesting that the benefit of execution flexibility depends on how well the compiled coordination semantics capture the dependencies required for task completion. Detailed results and evaluation procedures for both compilation settings are in Appendix D.1.

4.3 Coordination Refinement under Contingencies

For **RQ2**, we evaluate the refinement of runtime coordination semantics over 3 *compile–execute–refine* rounds in Quadmanual. We compare two forms of event information for the same contingencies within each round. One setting provides *continuous measurements* from which the program must infer the event condition. The other provides *descriptive event tags* that describe the observed condition without specifying the required coordination response. In both settings, each round introduces a new class of contingencies while retaining coordination revisions from previous rounds. After each round, we report the success rate over the same 120 held-out cases evaluated in separate subtask episodes.

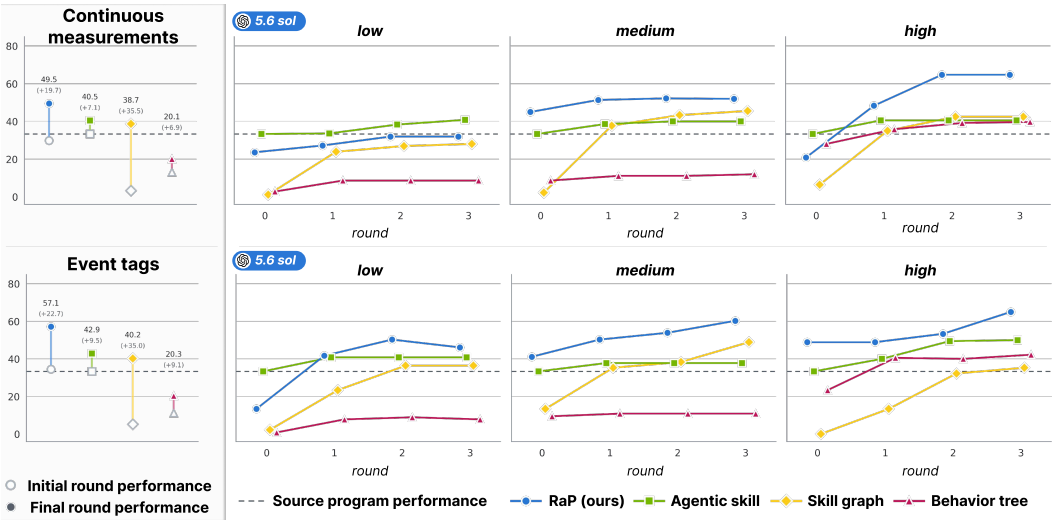

Figure 3: Refinement performance with *GPT-5.6 Sol* in Quadmanual. Event information is provided as *continuous measurements (top)* or *descriptive event tags (bottom)*. *(left)* Initial and final performance averaged equally across 3 seeds and the *low*, *medium*, and *high* reasoning efforts. Labels report final success rates, and parenthesized values indicate gains in percentage points over the initial performance of each approach. *(right)* Performance over refinement rounds, averaged over three seeds for each method–effort configuration.

Adaptation performance. In Figure 3, `RaP` achieves the highest mean final held-out success across the three reasoning efforts under both forms of event information.

With *continuous measurements (top)*, `AS` starts from the source program and gradually improves its success, with gains concentrated in cases similar to those encountered during refinement. In contrast, `SG` and `BT` begin below the source performance after translating the task program into their respective representations. `SG` recovers part of the performance lost during compilation through local subgraph updates. However, feedback from earlier rounds is retained mainly through accepted graph changes, which can limit the accumulation of reusable coordination semantics from unresolved failures. For `BT`, expressing responses to diverse contingencies can require increasingly complex branching structures, making coordination revisions harder to incorporate through local tree updates.

RaP retains explicit coordination semantics as a persistent refinement target, allowing successive rounds to build on compiled rules and accumulate revisions from execution feedback, with the highest final held-out success achieved at *medium* and *high* effort. With *descriptive event tags (bottom)*, RaP follows a similar refinement trend, suggesting that its ability to infer and refine coordination semantics from execution feedback extends across different forms of event information.



Figure 4: *(left)* A contingency observed during refinement, in which the cart traverses the shared workspace from west to east, and a held-out case in which the traversal direction is reversed from east to west, yielding a different realization of the same workspace-occupancy condition. *(right)* Failure cases of the baselines and the execution trace of RaP under the held-out case.

Runtime Reuse of Coordination Semantics. The runtime reuses retained coordination semantics to adjust transfer timing while preserving the task's skill sequence. Figure 4 examines a bottle handover and delivery subtask in which the cart traverses the shared workspace in the direction opposite to that encountered during refinement. In this cart contingency, reversing the traversal direction changes the timing of workspace occupancy while preserving the task objective and primitive skills.

Although AS checks whether the shared workspace is available before transfer, a subsequent motion interruption terminates the program without recovery. SG reaches the handover region but terminates with the bottle left in the cart's path, where the cart subsequently contacts it. Checking workspace availability only before transfer does not account for changes in occupancy during execution.

In RaP, the retained rule conditions transfer on the availability of the shared workspace based on current and projected cart occupancy. The runtime enforces this condition throughout transfer by suspending and resuming motion while preserving the grasp. RaP completes bottle delivery in this replay, but entry into the transfer region occurs before the cart has fully passed, violating the cart-entry criterion.

4.4 Real-World Coordination Transfer

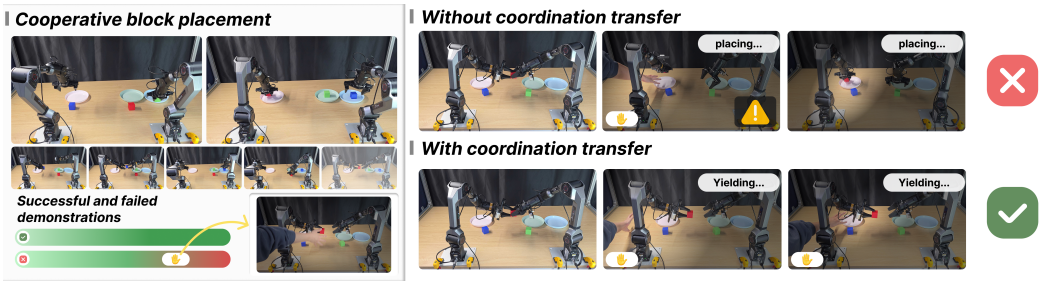

Figure 5: *(left)* Cooperative Block Placement setup and successful and failed handover demonstrations used for refinement. *(right)* Reuse of the handover occupancy condition during placement. Without coordination transfer (top), placement continues while a human hand occupies the target workspace. With transfer (bottom), the arm yields while the workspace is occupied.

For **RQ3**, two Piper manipulators perform Cooperative Block Placement, using handovers when a target plate is beyond the reach of the arm holding the block. During handover, a human hand enters the workspace as a coordination-relevant contingency. We test whether coordination conditions inferred by RaP from successful and failed handovers under this interference can be reused when the same contingency arises during picking and placement, without further refinement.

Real-world evaluation. We consider a contingency in which an external object transiently occupies the workspace of an upcoming skill. Each method receives a single case comprising two handover demonstrations. In the successful demonstration, the arm pauses while the sensor reports `hand_resting` and resumes when it reports `hand_passed`. In the failed demonstration, the arm moves while the sensor still reports `hand_resting`, and the program terminates. From this case, each method performs a single refinement step without further execution. Task success is then measured on held-out situations that place the same occupancy on the workspaces of other skills, a cube before pick and a plate before place, and on the handover workspace under a different goal, layout, and timing. The evaluation therefore fixes the contingency and varies the skill it must govern, whereas RQ2 varies how a contingency is observed under a fixed coordination requirement. At *high* reasoning effort, `RaP` transfers the handover occupancy condition to placement as illustrated in Figure 5, succeeding in 7 of 10 held-out cases in the logical evaluator, compared with 2 of 10 for `AS` and 0 of 10 for `SG`, whose compiled graph terminated at the first branch.

4.5 Ablation

We ablate each component of `RaP` using the RQ2 setup to evaluate the contributions of the runtime kernel, explicit coordination semantics, and hypotheses.

Table 1: Ablations of explicit coordination semantics and hypotheses with *GPT-5.6 Sol*. Checkmarks indicate enabled components. K denotes the runtime kernel and C denotes explicit coordination semantics while H denotes hypotheses. SR is averaged across the two event information settings and reported as mean and standard deviation over three seeds. `RaP` uses the same runs as the RQ2 evaluation. The Avg. column additionally averages across the three reasoning efforts.

Variant	Components			SR (%) ↑			
	K	C	H	*low*	*medium*	*high*	Avg.
`AS`	✗	✗	✗	40.8±1.4	38.9±2.4	45.3±7.4	41.7±1.9
`AS + Coord. Guidance`	✗	✓	✗	59.0±10.2	49.2±9.1	43.3±2.6	50.5±2.1
`AS + Event Kernel`	✓	✗	✗	32.8±14.0	48.2±6.7	53.9±3.8	45.0±5.1
`RaP w/o hypothesis`	✓	✓	✗	37.7±15.4	46.5±5.2	56.7±16.0	46.9±11.9
`RaP`	✓	✓	✓	39.0±19.8	56.1±13.0	64.9±9.3	53.3±7.6

In Table 1, `AS + Coord. Guidance` improves mean SR over `AS` from 41.7% to 50.5%. This supports explicit runtime coordination semantics as guidance for program refinement. `AS + Event Kernel` outperforms `AS` at *medium* and *high* effort but performs worse at *low* effort. Within the kernel variants explicit coordination semantics increase mean SR from 45.0% to 46.9% although the gain is not consistent across reasoning efforts. `RaP w/o hypothesis` reduces mean SR from 53.3% for `RaP` to 46.9% and performs worse at every reasoning effort. This consistent decrease supports using hypotheses to assess and refine coordination semantics.

5 Conclusion

We present `RaP`, a *compile–execute–refine* framework that turns skill invocations into runtime-managed processes and externalizes their coordination semantics as a persistent refinement target. By separating these semantics from task-specific control flow, `RaP` uses execution feedback to revise when skills may proceed and retains the revised conditions for subsequent adaptation. Our results support explicit coordination refinement as a means of improving task success under contingencies and provide initial evidence that the resulting conditions can be reused when the same contingency arises in new execution contexts.

AI USE STATEMENT

Generative AI tools were used to assist with implementing the simulation environment and refining the writing of this paper. They were also used as experimental agents to generate programmatic policy code in the evaluated systems.

REPRODUCIBILITY STATEMENT

To support reproducibility, Appendix A presents pseudocode and an expanded description of the implemented method. Appendix C.2 documents the model configurations and hardware resources used in the experiments.

REFERENCES

Jicong Ao, Fan Wu, Yansong Wu, Abdalla Swiki, and Sami Haddadin. Llm-as-bt-planner: Leveraging llms for behavior tree generation in robot task planning. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1233–1239. IEEE, 2025.

Kaylee Burns, Ajinkya Jain, Keegan Go, Fei Xia, Michael Stark, Stefan Schaal, and Karol Hausman. GenCHiP: Generating robot policy code for high-precision and contact-rich manipulation tasks. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 9596–9603, 2024. doi: 10.1109/IROS58592.2024.10801525. URL https://doi.org/10.1109/IROS58592.2024.10801525.

Yishuai Cai, Xinglin Chen, Zhongxuan Cai, Yunxin Mao, Minglong Li, Wenjing Yang, and Ji Wang. Mrbtp: Efficient multi-robot behavior tree planning and collaboration. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 14548–14557, 2025a.

Yishuai Cai, Xinglin Chen, Yunxin Mao, Minglong Li, Shaowu Yang, Wenjing Yang, and Ji Wang. HBTP: Heuristic behavior tree planning with large language model reasoning. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 13706–13713. IEEE, 2025b. doi: 10.1109/ICRA55743.2025.11127999. URL https://doi.org/10.1109/ICRA55743.2025.11127999.

Yishuai Cai, Xinglin Chen, Yunxin Mao, Kun Hu, and Minglong Li. Cabto: Context-aware behavior tree grounding for robot manipulation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 36180–36188, 2026.

Nicolas Carion et al. SAM 3: Segment anything with concepts. *arXiv preprint arXiv:2511.16719*, 2025.

Guojun Chen, Alex Schott, and Lin Zhong. Typego: An os runtime for embodied agents. *arXiv preprint arXiv:2607.05482*, 2026a.

Kaiyuan Chen, Shuangyu Xie, Letian Fu, Justin Yu, William Pacini, Sandeep Bajamahal, Hudson Kim, Jaimyn Drake, Daehwa Kim, Haoru Xue, et al. Gap: A graph-as-policy multi-agent self-learning harness for variational automation tasks. *arXiv preprint arXiv:2607.05369*, 2026b.

Siyi Chen, Hugo Hadfield, Alex Zook, Mikaela Angelina Uy, Chan Hee Song, Erwin Coumans, Xuning Yang, Faisal Ladhak, Qing Qu, Stan Birchfield, et al. 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, Zixuan Li, Qiwei Liang, Xianliang Lin, Yiheng Ge, Zhenyu Gu, et al. Robotwin 2.0: A scalable data generator and benchmark with strong domain randomization for robust bimanual robotic manipulation. *arXiv preprint arXiv:2506.18088*, 2025.

David Coleman, Ioan Sucan, Sachin Chitta, and Nikolaus Correll. Reducing the barrier to entry of complex robotic software: a MoveIt! case study. *arXiv preprint arXiv:1404.3785*, 2014.

Michele Colledanchise and Lorenzo Natale. Handling concurrency in behavior trees. *IEEE Transactions on Robotics*, 38(4):2557–2576, 2022. doi: 10.1109/TRO.2021.3125863. URL https://doi.org/10.1109/TRO.2021.3125863.

Christian Dreher and Tamim Asfour. Learning symbolic and subsymbolic temporal task constraints from bimanual human demonstrations. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5160–5167. IEEE, 2024.

Christian Dreher, Patrick Dormanns, Andre Meixner, and Tamim Asfour. Unified learning of temporal task structure and action timing for bimanual robot manipulation. *arXiv preprint arXiv:2603.06538*, 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.

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

Caelan Garrett and Fabio Ramos. Schedulestream: Temporal planning with samplers for gpu-accelerated multi-arm task and motion planning & scheduling. *arXiv preprint arXiv:2511.04758*, 2025.

Peter W. Glynn. A GSMP formalism for discrete event systems. *Proceedings of the IEEE*, 77(1): 14–23, 1989. doi: 10.1109/5.21067.

Philip Huang, Ruixuan Liu, Shobhit Aggarwal, Changliu Liu, and Jiaoyang Li. Apex-mr: Multi-robot asynchronous planning and execution for cooperative assembly. *arXiv preprint arXiv:2503.15836*, 2025.

Matteo Iovino, Julian Förster, Pietro Falco, Jen Jen Chung, Roland Siegwart, and Christian Smith. On the programming effort required to generate behavior trees and finite state machines for robotic applications. In *2023 IEEE international conference on robotics and automation (ICRA)*, pp. 5807–5813. IEEE, 2023.

Matteo Iovino, Julian Förster, Pietro Falco, Jen Jen Chung, Roland Siegwart, and Christian Smith. Comparison between behavior trees and finite state machines. *IEEE Transactions on Automation Science and Engineering*, 2025.

Riccardo Andrea Izzo, Gianluca Bardaro, and Matteo Matteucci. BTGenBot: Behavior tree generation for robotic tasks with lightweight LLMs. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 9684–9690. IEEE, 2024. doi: 10.1109/IROS58592.2024.10802304. URL https://doi.org/10.1109/IROS58592.2024.10802304.

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 *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9493–9500, 2023. doi: 10.1109/ICRA48891.2023.10160591. URL https://doi.org/10.1109/ICRA48891.2023.10160591.

Yichao Liang, Dat Nguyen, Cambridge Yang, Tianyang Li, Joshua B Tenenbaum, Carl Edward Rasmussen, Adrian Weller, Zenna Tavares, Tom Silver, and Kevin Ellis. Exopredicator: Learning abstract models of dynamic worlds for robot planning. In *International Conference on Learning Representations*, volume 2026, pp. 49174–49214, 2026.

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 Fan, and Guanzhi Wang. ASPIRE: Agentic /skills discovery for robotics. *arXiv preprint arXiv:2607.00272*, 2026. URL https://arxiv.org/abs/2607.00272.

Mayank Mittal, Calvin Yu, Qinxi Yu, Jingzhou Liu, Nikita Rudin, David Hoeller, Jia Lin Yuan, Ritvik Singh, Yunrong Guo, Hammad Mazhar, Ajay Mandlekar, Buck Babich, Gavriel State, Marco Hutter, and Animesh Garg. Orbit: A unified simulation framework for interactive robot learning environments. *IEEE Robotics and Automation Letters*, 8(6):3740–3747, 2023. doi: 10.1109/LRA.2023.3270034.

Yudha Pane, Vahid Mokhtari, Erwin Aertbeliën, Joris De Schutter, and Wilm Decré. Autonomous runtime composition of sensor-based skills using concurrent task planning. *IEEE Robotics and Automation Letters*, 6(4):6481–6488, 2021. doi: 10.1109/LRA.2021.3094498.

Xingyu Peng, Chen Gao, Liankai Jin, Annan Li, and Si Liu. Bicoord: A bimanual manipulation benchmark towards long-horizon spatial-temporal coordination. *arXiv preprint arXiv:2604.05831*, 2026.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. ProgPrompt: Generating situated robot task plans using large language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 11523–11530, 2023. doi: 10.1109/ICRA48891.2023.10161317. URL https://doi.org/10.1109/ICRA48891.2023.10161317.

Jonathan Styrud, Matteo Iovino, Mikael Norrlöf, Mårten Björkman, and Christian Smith. Automatic behavior tree expansion with llms for robotic manipulation. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1225–1232. IEEE, 2025.

Guido van Rossum and Phillip J. Eby. PEP 342: Coroutines via enhanced generators. Technical Report 342, Python Software Foundation, 2005. URL https://peps.python.org/pep-0342/.

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

Jianwei Yang, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, and Jianfeng Gao. Set-of-mark prompting unleashes extraordinary visual grounding in GPT-4V. *arXiv preprint arXiv:2310.11441*, 2023.

Håkan L. S. Younes and Reid G. Simmons. Solving generalized semi-markov decision processes using continuous phase-type distributions. In Deborah L. McGuinness and George Ferguson (eds.), *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pp. 742–748. AAAI Press / The MIT Press, 2004. URL http://www.aaai.org/Library/AAAI/2004/aaai04-117.php.

Duo Zhang, Junshan Huang, and Jingjin Yu. High-performance dual-arm task and motion planning for tabletop rearrangement. *arXiv preprint arXiv:2512.08206*, 2025.

A METHOD DETAILS

A.1 PROCEDURE OVERVIEW

RaP makes skill behavior and runtime coordination explicit targets for program refinement. A coding agent compiles a source task program into an executable that runs skill invocations as processes. It also defines the skills and coordination relations used by the program and writes hypothesis tests for them. Execution applies these relations and collects evidence without agent calls. Refinement uses the evidence to revise the task program, its definitions, and its tests.

Figure 6 shows the procedure with development rollouts. One refinement agent works on each task, while automatic procedures, which we call the host, check and evaluate its proposals. An admitted proposal becomes the program for the next round. Repairs can also produce advisory knowledge that later agents consult across tasks. Held-out evaluation measures the resulting programs independently of refinement.

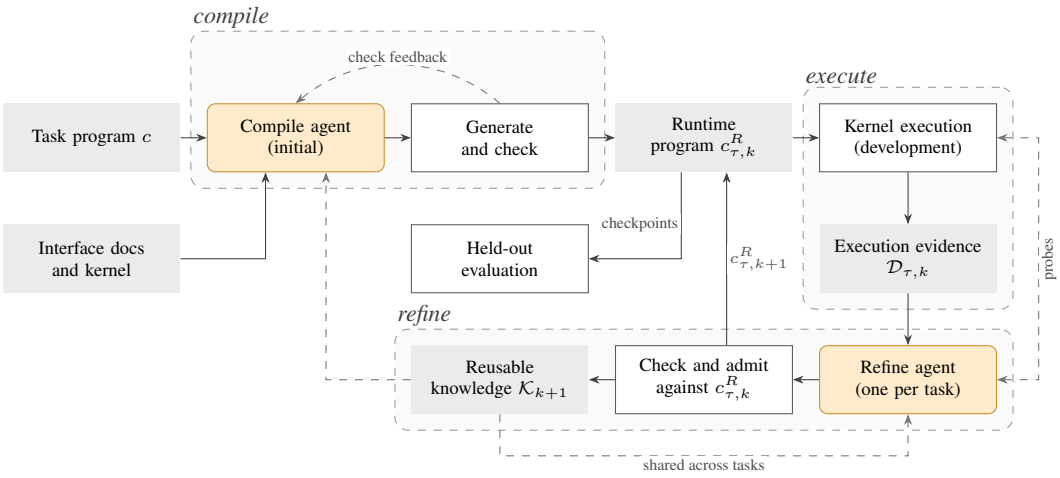

Figure 6: Lifecycle with development rollouts. Orange nodes denote agent steps, white nodes host procedures, and gray nodes their inputs and outputs. Dashed regions mark compile, execute, and refine. Probes evaluate intermediate candidates on development cases. Advisory knowledge passes across tasks, while held-out evaluation remains outside refinement.

A.2 REPRESENTATIONS AND REPAIR TARGETS

The representation separates the task's strategy from the conditions governing its execution. Table 2 connects the objects of this representation to the notation in Sections 3.2–3.3. Together, the task program, skills, runtime coordination semantics, and hypotheses form the compiled policy $\theta_{\tau,k}$ of task τ at round k. Its executable is denoted $c^R_{\tau,k}$, while $\hat{\kappa}_\tau$ and $\mathcal{H}_\tau$ omit the round index for brevity. For a fixed task and round, π_Ω denotes the event-selection policy realized by this executable, with $\Omega = \Omega_k$.

Table 2: Method objects and their roles in refinement.

Object	Representation	Role in refinement
Task program c	Task strategy expressed through control flow and skill invocations.	Revise the strategy and its implementation on the runtime.
Skills Ω	Functional units with stated inputs, assumptions, effects, and failures.	Revise a skill's behavior or the conditions under which it is expected to work.
Runtime coordination $\hat{\kappa}_\tau$	Definitions of relations among processes, implemented through rules, synchronization calls, and resource declarations.	Revise which processes may proceed and how they respond to changing conditions.
Hypotheses $\mathcal{H}_\tau$	Claims linked to skill or coordination definitions and to tests in a test policy.	Relate observations to the definition that may need revision.
Evidence $\mathcal{D}_{\tau,k}$	Task outcomes, runtime records, and hypothesis observations.	Assess the current claims and compare candidate repairs.
Knowledge $\mathcal{K}_k$	Repair advice with explicit applicability conditions.	Inform revisions in other tasks, subject to their own admission.

Task and skill behavior. The source task program c expresses the task strategy. The agent lowers it to a *nominal program* that runs on the kernel and remains executable without hypothesis tests. Each process advances through steps that issue primitive commands or synchronized command groups, with waits between them. The nominal program and test policy compose into the runtime program $c^R_{\tau,k}$, which is executed. Source and nominal programs may coincide when the source already uses the kernel.

Each skill definition states what the skill should accomplish and the assumptions under which it applies. Its implementation may occupy a module or a span of the task program and may compose several primitive commands. The implementation marks each invocation as a *skill event*, an interval with start and end records. This interval may contain several command events of the event model in Section 3.1. Skill events associate observations with invocations and expose their current activities, such as motion or perception, to coordination rules.

Runtime coordination definitions and enforcement. A coordination definition states when and how processes may proceed together. The definition records where the relation is implemented. Synchronization calls, such as waits and barriers, govern the processes that make them. Kernel rules govern all processes of a run that match their scope, while resource declarations express mutual exclusion.

These definitions remain explicit repair targets even when their implementation lies in the nominal program. A relation implemented there requires a corresponding code edit when its behavior changes. A relation implemented as a kernel rule can change without revising the task's skill sequence. Prose describes the relation's meaning and applicability, while its executable implementation determines behavior.

A kernel rule represents a relation by

$$(\text{when},\ \text{scope},\ \text{transition},\ \text{recover}),$$

together with a set of unaffected processes. The condition *when* reads public observations, inferred states, or skill progress. The *scope* selects processes by name or current activity, and the unaffected set removes exceptions such as perception processes that should continue during a motion hold. The *transition* determines the response in Table 3. For a suspended step, recovery requires a declared condition to hold for a specified number of control steps. By default, this condition is the negation of *when*.

Table 3: Kernel rule transitions and their effects on selected processes.

Transition	Effect while *when* holds	Continuation
Hold	Starts no new step. A running step continues.	Builds and starts the pending step when the hold clears.
Yield	Cancels the running step and preserves task progress. A process without a running step is held.	After recovery, reissues the interrupted step from the current physical state, or fails the attempt if declared.
Cancel	Cancels the running step once per rising edge of *when*.	The step policy judges the canceled attempt.
Admit	Starts a process from a declared factory once per rising edge of *when*.	The new process joins the current run.

Interpreting observations. An inferred state space separates the interpretation of observations from the coordination response. It defines named states and a classifier that maps public observations to one of them. Rules and waits use these states through `rt.state`. The classifier may use trial memory and context such as the arm or object being observed. An unreadable or unclassified observation yields a declared default state and a diagnostic record. The default makes the response to missing information explicit, for example holding motion in an unknown state.

This separation provides two repair targets. A change in how a condition appears can require a new classifier or a finer state space. A change in the required response can require a new decision over those states. Both changes can remain within the coordination semantics when the decisions are implemented as kernel rules.

Hypotheses and tests. A hypothesis $h \in \mathcal{H}_\tau$ states an applicability or behavior claim about one skill or runtime coordination definition. Skill and runtime coordination hypotheses belong to $\mathcal{H}_\tau^\Omega$ and $\mathcal{H}_\tau^\kappa$, respectively. It references that definition and names a test, or explains why no public observation can check it. The test policy specifies when to gather information and evaluate the claim. At each test, m_h extracts the evaluation context $z_{h,t} = m_h(s_t) \in \mathcal{Z}_h$, and φ_h evaluates it as $h(s_t) = \varphi_h(z_{h,t}) \in \{0, 1, \bot\}$, as in equation 7. A skill test might observe whether a grasp retains its target. A coordination test might observe whether transfer motion overlaps workspace occupancy. These links let refinement relate a contradicting observation to a particular definition. Assessing the definition's adequacy also requires the task outcomes and runtime context.

A.3 Illustrative Runtime Coordination Repair

Consider a bottle transfer with public observations of grasp retention and shared-workspace occupancy. This example illustrates how the representation guides a repair. Suppose the initial program checks clearance before starting the transfer. Its tests observe two claims during transfer motion: the grasp retains the bottle, and the shared workspace remains clear.

Now suppose occupancy begins after the initial check. The trace shows continuing transfer motion, a true grasp observation, and a false clearance observation. This evidence directs attention to when the coordination condition is enforced. It motivates testing a rule that responds throughout transfer, while retaining the grasp behavior.

Pseudocode 1 sketches this candidate. The inferred state `zone` has the values `free`, `occupied`, and `unknown`, with `unknown` as its default. The rule suspends motion while the zone is not free and lets perception continue. The test policy samples grasp retention and clearance while transfer motion is active. The read functions below summarize public observations, and `recovery_steps` is the declared stability interval.

The refinement agent can probe this candidate on the affected development cases. The host then compares the complete development and validation outcomes before admitting a changed executable. Thus, a contradicting observation motivates a repair, while the task comparison determines whether to retain it. If the classifier itself misinterprets occupancy, the agent must also reconsider the mapping from observations to states. Tests that use the same classifier cannot independently establish that this mapping is correct.

Pseudocode 1: A candidate workspace coordination repair

```
# Execute this rule throughout the run.
rt.rule(when=lambda: rt.state("zone") != "free",
        scope=Effects("motion"),
        unaffected=Effects("perception"),
        transition="yield",
        recover=Recover(until=lambda: rt.state("zone") == "free",
                        stable_for=recovery_steps, then="reissue"))

# Called by the test policy while transfer motion is active.
def observe_transfer(rt):
    rt.observe("grasp_retained", read_grasp_retention)
    rt.observe("transfer_clear", read_workspace_clearance)
```

A.4 Compilation

Compilation turns the source program and its interface documentation into the executable representation above. The agent receives the kernel and available knowledge and authors the task's skills, coordination definitions, hypotheses, and test policy. The skill and relation vocabularies are open, allowing the agent to introduce units suited to the task. Only initial compilation calls a separate compilation agent. Later refinement sessions revise these same objects and use the same generation and checking procedure.

The host checks whether the authored objects form a program compatible with the public interfaces. Checks cover hypothesis declarations, module loading, interface use, and regeneration of the runtime program from its nominal program and test policy. The host also executes the program against simplified interface responses, without simulator interaction. These checks establish program compatibility. Whether the implementation realizes a relation's intended meaning and achieves the task is assessed through execution evidence.

Pseudocode 2 shows the authoring loop. Calls through `agent` invoke the coding agent, and other calls denote host procedures. A `program` bundles the authored objects with their derived executable. A `task` supplies its source, interface documentation, and development and validation cases.

Pseudocode 2: Compilation

```
def compile(task, knowledge):
    feedback = None
    for _ in range(budget.compile_attempts):
        objects = agent.author(task, knowledge, feedback)
        program, feedback = generate_and_check(objects, task)
        if feedback.ok:
            return program
    return None  # no executable for evaluation

def generate_and_check(objects, task):
    executable = compose(objects.test_policy, objects.nominal)
    program = bundle(objects, executable)
    feedback = check(program, task.interface, KERNEL)
    return program, feedback
```

A.5 Execution Evidence

Execution runs the current runtime program under fixed definitions. The kernel enforces coordination and records what each process did or waited for. The test policy supplements this trace with observations linked to hypotheses. Appendix A.7 specifies the kernel procedure.

Each hypothesis observation records the outcome $h(s_t)$ as true (1), false (0), or unknown ($\perp$), together with its claim, test, control step, and enclosing skill invocation when one is active. An exception or a non-Boolean reading yields unknown. A claim whose test never runs remains unobserved. Recording an observation through `rt.observe` has no automatic control effect. The test policy may gather information or intervene through explicit code, whose effects are part of the evaluated runtime program.

Step and wait records describe outcomes, durations, and blocking conditions. Rule records identify the coordination transitions applied to each process. State records provide the initial inferred state and subsequent transitions, while error records identify failed reads and classification fallbacks. Together, these records distinguish a failed observation from an observed contradiction and connect failures to the conditions under which they occurred. Task outcomes complete the evidence $\mathcal{D}_{\tau,k}$, since a locally successful recovery may still fail to complete the task.

A.6 Refinement and Retention

Refinement uses this evidence to choose what to revise and tests whether the revision improves the task. One agent session receives the current task objects, development traces, earlier admission feedback, and applicable knowledge. It may revise the task strategy, skills, coordination definitions, hypotheses, and tests together. Validation contributes outcomes and metrics to admission feedback. The agent can read the fixed kernel but cannot modify it or inspect other tasks' programs and traces. The isolation procedure is described in Appendix C.4.

Diagnosis and development probes. The agent assesses hypothesis observations alongside the runtime records and retains counterexamples and limitations in its claims. A false observation identifies a candidate repair target. An unknown observation instead calls for examining the read or the information available to the test. The agent may test competing explanations by revising a candidate and requesting rollouts on selected development cases. Each request uses the checked candidate and returns outcomes and trace views within the declared probe budget.

Admission. Pseudocode 3 gives the host's decision procedure. Changes to claims or prose that leave the executable unchanged are retained directly. A changed executable must improve on the complete development set. Success count is the primary criterion. At equal success counts, the candidate must improve at least one declared efficiency metric beyond its margin without worsening another beyond its margin. By default, these metrics are trajectory length over successful rollouts and simultaneous movement percentage over all rollouts. Their margins are 5% of the current trajectory length and 5 percentage points of simultaneous movement percentage. All comparisons require complete outcomes, and efficiency comparisons also require complete metric measurements.

Validation, when configured, requires at least as many successes as the current program. If the development gain was in efficiency and validation success counts are equal, the same efficiency margins bound regressions. These criteria preserve aggregate success counts, so individual cases may change outcome. Every task with development cases can enter refinement, including one whose cases already succeed.

An admitted candidate becomes the next round's program. A failed check, comparison, or exhausted budget retains the current program. Refinement has already generated and checked the accepted executable, so the next round needs no additional compilation agent. For the one-shot real-world study (RQ3), the agent instead receives supplied case evidence and makes one offline revision. That setting uses program compatibility checks without development probes or rollout-based admission.

What is retained and reused. Within a task, accepted coordination definitions and their implementations remain available in later rounds. A kernel rule applies wherever a process matches its scope, including when the same contingency arrives at another point in the task. Retained claims carry the counterexamples and limitations that inform subsequent revisions.

Across tasks, learned information passes through advisory knowledge. The refinement agent may propose a procedure with explicit applicability conditions, supported by cases that failed before an admitted repair and succeeded afterward. The repair must increase the total number of successful development cases, and the host verifies the cited evidence. Hypothesis observations, validation results, and unadmitted trials alone do not qualify. A receiving agent uses the advice to author its

Pseudocode 3: Refinement and knowledge admission

```
def refine(program, trials, task, knowledge):
    # The agent may probe checked candidates on development cases.
    objects, proposals = agent.repair(program, trials, knowledge)
    candidate, feedback = generate_and_check(objects, task)
    if not feedback.ok:
        return program, []
    if candidate.executable == program.executable:
        return candidate, []
    dev = run_all(candidate, task.development)
    gain = compare(outcomes(trials), dev)
    if gain is None:
        return program, []
    val = task.validation
    if val and not preserves(program, candidate, val, gain):
        return program, []
    return candidate, proposals if gain == "success" else []

def compare(base, cand):
    if cand.successes > base.successes:
        return "success"
    if cand.successes < base.successes:
        return None
    if no_worse(cand, base) and some_better(cand, base):
        return "efficiency"
    return None

def preserves(program, candidate, cases, gain):
    base, cand = run_all(program, cases), run_all(candidate, cases)
    if cand.successes != base.successes:
        return cand.successes > base.successes
    return gain == "success" or no_worse(cand, base)

def promote(knowledge, repairs):
    return knowledge + [p for r in repairs for p in r.proposals
                        if cites_admitted_dev_repair(p)]
```

own definitions and behavior, which must pass that task's admission. This separates reuse of a rule within its scope from adoption of repair knowledge by another task.

A.7 Kernel

The kernel implements the event-driven runtime of Section 3.1. Each process is a generator that yields a step or a condition under cooperative scheduling (van Rossum & Eby, 2005). A step issues a primitive command or a paced group of commands. A condition suspends progress until its predicate holds. These yields are execution boundaries at which the kernel decides whether the process may proceed. Recording a skill's start or end alone does not create such a boundary.

The commands that the kernel starts or continues form the selected event set $\pi_\Omega(s)$. A suspension removes the interrupted command event. Recovery reissues the command from the current physical state, enabling a new event with a duration determined from that state. The state s therefore includes task progress and the history needed by coordination, such as how long a recovery condition has held.

Pseudocode 4 specifies scheduling and command execution. The environment supplies the cadence in control steps between clock advances, which the program may override. Rule conditions are checked at each scheduling pass. Inferred states are classified at most once per control step for the same context, so decisions within that step read the same value.

Pseudocode 4: Kernel execution

```
def execute(program, case):
    env = reset(case)
    program.run(env.api)
    return env.trace, env.success()

def kernel_run(rt, processes):
    while processes.live():
        while True:
            rt.apply_rules(processes)
            ready = processes.ready()
            if not ready:
                break
            for p in ready:
                p.advance()  # advance to the next step or condition
        if not processes.live():
            break
        if not rt.releasable(processes):
            raise Deadlock(processes)
        rt.api.tick(rt.cadence)

def step_policy(rt, p, step):
    for attempt in step.retries_then_alternatives():
        while True:
            yield rt.until_free(p)  # no hold and an idle arm
            command = attempt.build(rt.api)
            result = yield rt.start(command)
            if not result.suspended:
                break
            yield rt.until_recovered(result.rule)
            if result.rule.then == "fail":
                break
        if result.ok and step.check(result):
            return result
    return step.on_fail(result)
```

Overlapping rules and resources. The kernel applies rules in registration order. The rule that suspends a step supplies its recovery condition. Before reissuing the step, the process must also clear any currently applicable hold or yield rule. Suspension requires cancellation to take effect, and a command that finishes first is judged normally. An unreadable rule condition evaluates to false and produces an error record. Conditions over inferred states can instead use the declared default to hold motion under an uncertain reading.

The kernel treats each arm as exclusive. Skill events may additionally declare objects or spatial regions as resources, delaying acquisition while a conflicting event is active. Events acting on the same object are exempt from region conflicts to permit joint operations. These declarations and the coordination rules determine which steps may start.

Failure and termination. A command attempt fails if it does not execute, skips planned actions, or violates its declared result check. The step policy then uses the declared retries, alternatives, and final failure response. The kernel reports a deadlock when no command is executing and no remaining wait permits further control steps. Observation waits, holds, suspensions, and event acquisitions blocked by an active event of the same run permit time to advance. A recovery condition that never holds can therefore block progress until the trial limit.

B ENVIRONMENTS

This section describes the two simulated environments and the real-world environment used in the experiments.

B.1 BICOORD

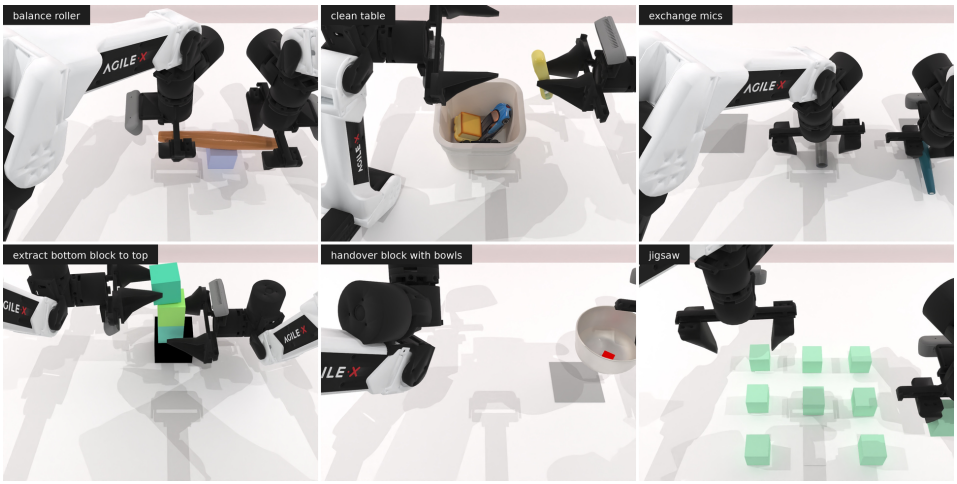

Figure 7: The BiCoord environment. Each panel shows a late frame from a successful rollout of the expert program of one task of the six-task subset. The top row shows `Balance Roller`, `Clean Table`, and `Exchange Mics`. The bottom row shows `Extract Bottom Block to Top`, `Handover Block with Bowls`, and `Jigsaw`.

BiCoord (Peng et al., 2026) is an existing benchmark of long-horizon bimanual tasks built on RoboTwin 2.0 (Chen et al., 2025). It uses the SAPIEN physics engine, the `aloha-agilex` dual-arm embodiment of RoboTwin 2.0, and the mplib motion planner. Each task ships an expert program written over shared primitives for arm motion, gripper control, synchronized two-arm motion, and delay. We keep the tasks, the primitives, the expert programs, and the success checks of the benchmark unchanged and use the 18 tasks of its official task set. Programs read object poses and annotated grasp points through the object interface of the benchmark, as the expert programs do. The shared asset repairs are described in Appendix C.4.

Control steps. We execute programs through a control-step loop that reproduces the execution of the benchmark primitives. Each arm owns a queue of actions, and each action is planned when it starts. One control step applies the next waypoint or gripper value of every active arm and then advances the physics scene by one step at 250 Hz. The synchronized two-arm primitive reproduces the benchmark execution round by round. It paces two motions by fractional progress so that both finish together, and a delay holds the current targets for a given number of steps. The interruptible calls of our runtime drive the same loop one step at a time. Polling and scheduling between control steps do not themselves advance simulated time.

Measurement. Every control step records for each arm whether a waypoint or gripper value was applied and whether the gripper is closed. The closed-gripper predicate follows the benchmark definition, an opening below 20% of the fully open width. Trajectory length counts the control steps of a successful rollout. Simultaneous movement percentage is the share of control steps in which both arms are active, where an arm is active when it moves or holds a closed gripper. We also record a motion-only variant that counts steps in which both arms apply a waypoint. This variant measures overlapping arm motion without counting stationary holds. Section C.3 gives the formulas.

Relation to the benchmark step limit. The step limit of the benchmark counts learned-policy actions, each of which spans many control steps, and applies only to its learned-policy evaluation

path. Our trajectory length counts control steps and is therefore not comparable with the values reported in the BiCoord paper. All trajectory-length comparisons in this work are between programs executed under the same control-step loop. Each trial is bounded at 20 control steps per unit of the native step limit of the task, and the expert programs remain within this bound.

B.2 QUADMANUAL

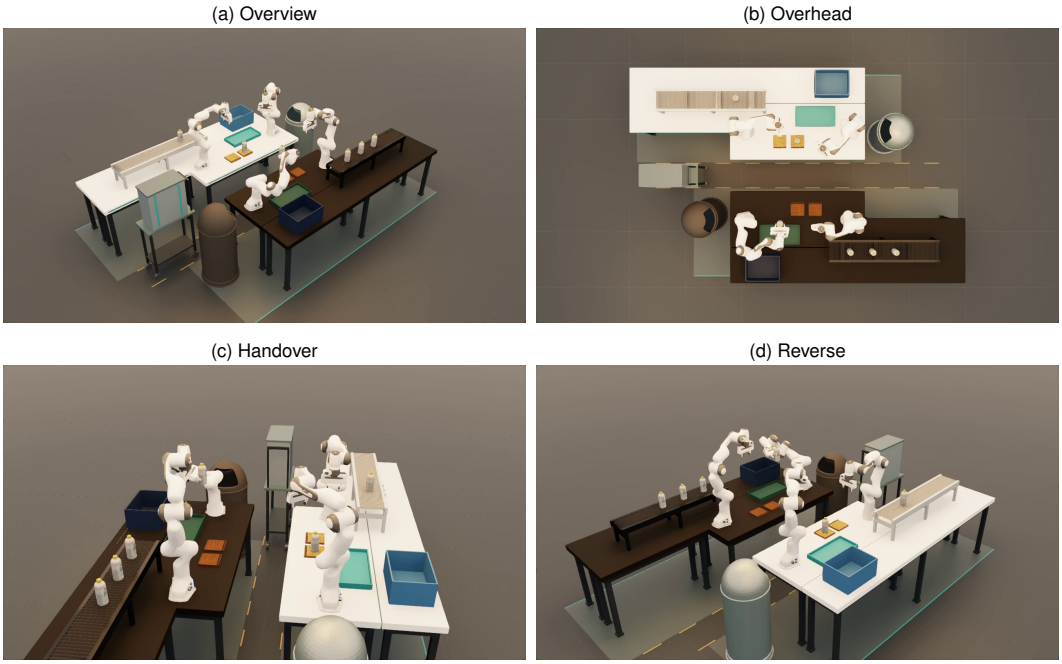

Figure 8: The Quadmanual environment. The four panels show one moment of an episode from the overview, overhead, handover, and reverse cameras. Each of the two facing workstations holds a supply arm, a receiver arm, a conveyor feed, two holding pads, and three destinations. A rail-mounted cart passes through the aisle between the workstations.

B.2.1 OVERVIEW

Quadmanual is a four-arm conveyor sorting environment with handover and contingency events built on Isaac Sim. Four Franka Emika Panda arms with Panda hands are mounted on two facing workstations (Figure 8). The north workstation holds arms A and B, and the south workstation holds arms C and D. Each workstation has a bottle feed with an infeed conveyor and a pickup conveyor, two holding pads for temporary storage, and three destinations. The destinations are a lidded trash bin, a recovery bin, and a sort tray. An aisle separates the two workstations, and a rail-mounted cart can pass through the aisle and occupy the handover zone between them.

Supply arms A and C pick bottles from their conveyors and hand them to receiver arms B and D. A handover is local when it stays within a workstation and cross when it reaches the opposite workstation. A handover is direct when the receiver takes the bottle from the supply arm. It is indirect when the supply arm places the bottle on a holding pad and releases it before the receiver picks it up. Receiver arms deliver bottles to their assigned destinations and return to their start poses.

All four arms, objects, and devices advance on a shared clock with a 40 ms tick composed of eight 5 ms physics steps. A policy submits motion requests per arm, polls their status, and advances the shared clock. Section B.2.5 specifies this interface, and Table 4 summarizes the environment.

Table 4: Specification of the Quadmanual environment.

Item	Specification
Robots	Four Franka Emika Panda arms, two per workstation
Gripper	Panda Hand with an opening width of 0 to 0.08 m and a grasp effort of at most 40 N
Workstations	Two facing workstations, north with arms A and B and south with arms C and D
Supply	One bottle feed per workstation, with separate infeed and pickup conveyors
Temporary storage	Two holding pads per workstation
Destinations	Lidded trash bin, recovery bin, and sort tray per workstation, six in total
Handover routes	Local A→B and C→D, cross A→D and C→B, each direct or through a holding pad
Mobile obstacle	Rail-mounted cart that crosses the central aisle on a 3 m rail in 37.5 s and can occupy the cross handover zone
Workload	One to four bottles per workstation with local, cross, or mixed routing
Representative task	Two north bottles and four south bottles with mixed routing
Policy input	Numeric readings of the current state including object poses from the simulator
Control tick	Shared 40 ms tick composed of eight 5 ms physics steps
Recording views	Overview, overhead, handover, and reverse cameras at 1280×720 and 25 fps

B.2.2 Objective

The task `Four-Arm Sorting` requires delivering every required bottle to its assigned destination and returning all four arms to their start poses. Each workstation receives one to four bottles. The routing mode assigns local routes, cross routes, or a mix of both. Each bottle carries a route that names its supply arm, its receiver arm, and its destination. The receiver arm of a route is fixed before the handover. A handover counts only when the receiver arm alone lifts the bottle away from the transfer point. A bottle counts as delivered when it rests inside its destination for two seconds after release without support from any arm. Motion completion alone does not count as delivery. An episode succeeds when every required bottle is delivered and all four arms have returned to their start poses. Delivering a subset of the bottles does not count as success. Completion time is measured in simulation seconds at physical time and excludes waiting for motion-planner computation. Contingency events change destinations, priorities, and the set of required bottles during execution. Other events block a destination, the handover zone, or a conveyor. Section B.2.6 lists them.

B.2.3 Task Structure

The representative task feeds two bottles on the north conveyor and four bottles on the south conveyor. Its routing is mixed. Odd-numbered bottles on each conveyor take local routes, and even-numbered bottles take cross routes. Each bottle defines one subtask that moves the bottle from its conveyor to its destination. A subtask consists of a pick, a handover, and a place. Table 5 gives the completion condition of each stage.

Table 5: Stages of one subtask in `Four-Arm Sorting`.

Stage	Arms	Completion condition
Pick	Supply	The supply arm grasps the bottle at the pickup point and lifts it by 8 cm.
Storage	Supply	The bottle rests on a holding pad without a grasp. The same arm then regrasps it and lifts it 8 cm above the pad.
Handover	Supply and receiver	Both arms hold the bottle for at least 80 ms within 8 cm of the transfer point without other contact. The receiver arm then holds it alone for at least 160 ms.
Place	Receiver	The bottle rests inside its destination for two seconds after release without support.

Storage is an optional stage between the pick and the handover. A handover first reserves a transfer volume. The supply arm then aligns the bottle at the transfer point, and the receiver arm grasps it. The supply arm releases the bottle and withdraws before the receiver arm lifts it.

Subtasks overlap in time. A supply arm starts its next pick once it has withdrawn from a handover. A receiver arm holds one bottle at a time. The local handovers AB and CD each use a transfer volume inside their workstation. The cross handovers AD and CB share one transfer volume in the central aisle. The difficulty of this long-horizon task lies in scheduling six subtasks over four arms and the shared transfer volumes.

Quadmanual includes a reference program for this task written over the public API of Section B.2.5. Table 6 lists the six subtasks with the order in which the reference program delivers them. Each conveyor feeds its bottles in index order. Receiver arm B delivers T2, T3, and T6. Receiver arm D delivers T1, T4, and T5.

Table 6: Subtasks of the representative task and their delivery order in the reference program.

Subtask	Bottle	Arms	Handover	Destination	Delivery order
T1	South 1	C→D	Local	South trash bin	1
T2	North 1	A→B	Local after storage	North trash bin	4
T3	South 2	C→B	Cross	North recovery bin	2
T4	North 2	A→D	Cross	South recovery bin	3
T5	South 3	C→D	Local	South sort tray	5
T6	South 4	C→B	Cross	North sort tray	6

The south workstation delivers T1 through its local handover first. Meanwhile arm A stores T2 on a north holding pad after the pick. This storage keeps receiver arm B free for the cross handover of T3. T4 enters the central transfer volume after T3 has left it. Arm A then regrasps T2 and hands it to arm B. T5 and T6 complete the south supply. The north feed order stays T2 before T4, while the north handover order becomes T4 before T2. The reference program fixes no start times. Each stage starts when the readings of the shared tick satisfy its conditions.

B.2.4 Simulation and Software Versions

Table 7 lists the simulator and library versions of the Quadmanual implementation described here.

Table 7: Software versions of the Quadmanual environment.

Component	Version
Simulator	Isaac Sim 5.1.0.0
Simulation framework	Isaac Lab v2.3.2 (commit `37ddf62`)
Motion planner	cuRobo v0.7.8 (commit `d64c4b0`)
GPU kernel library	Warp 1.8.2 (bundled `omni.warp.core`)
Deep learning runtime	PyTorch 2.7.0 with CUDA 12.6
Python	3.11
Tested host	Ubuntu 22.04, GeForce RTX 4090 (24 GiB), NVIDIA driver 580.178.04

B.2.5 POLICY API

Policies act on Quadmanual only through the functions of a public API book. The API book is the only environment document given to the policy. Table 8 lists the perception functions, the state sensors, and the control functions. The perception functions return measurements of the current state together with task geometry and issued requests. Object poses come from the simulator state, and no camera image reaches the policy. Readings do not expose the event schedule or host evaluation results. Short-horizon workspace projections are derived from the current state. Poses are TCP poses in world coordinates with positions in meters and unit quaternions.

Motion follows an asynchronous request model. A motion call returns a token without advancing time. The policy polls the token and advances the shared clock with `tick`. Each arm holds at most one active token. The other arms keep their own tokens. Motion planning runs one request at a time and pauses the clock for the whole scene. Before each tick a shared check predicts the swept volumes of every moving body over the next 80 ms at 10 ms intervals. A failed shared check stops every active request. A failure of a single request stops only that request. The policy replans after a stop. A token ends as `completed` or with one of nine failure statuses such as `collision` or `infeasible`. The status `completed` states only that the motion finished. A finished close command likewise does not establish a grasp. Table 9 lists the options of `move_to_pose`. The Cartesian options require the arm to be stationary at the start.

Table 9: Options of `move_to_pose` in the Quadmanual environment.

Option	Effect
Default	Motion planned by cuRobo with collision checks
`cartesian_translation`	Straight line of 1 mm to 40 cm with a fixed orientation
`cartesian_interpolation`	Straight line with an orientation change of at most 0.35 rad
`cartesian_waypoints`	Continuous spline through control poses at most 50 cm and 0.35 rad apart
`continuation_pose`	Prechecks a short Cartesian segment after the target without executing it
`allowed_contacts`	Permits contact between named hand links and named bodies during the motion
`carried_objects`	Requires contact of both fingers with a bottle and adds the bottle to the predicted collision geometry. The bottle stays a free body
`avoid_cart_rail`	Reserves the full rail corridor of the cart as an obstacle for this plan

Each state sensor returns a reading for one contingency target on an explicit call, without advancing time or including it in `observe` or `tick`. The observation profile of the episode sets the form of this reading. Under *descriptive event tags*, the reading is one state name that says what is observed, not what to do. Under *continuous measurements*, the reading gives the underlying measurements without a name, and the program infers the condition from them. Section B.2.6 describes the state semantics.

Table 8: Policy API of the Quadmanual environment.

Function	Description
Perception	
`observe`	Current snapshot that combines the readings below
`get_arm_state`	Measured joint state and finger opening of one arm
`get_arm_pose`	Measured TCP pose of one arm
`list_objects`	Poses and velocities of all observed bodies. A bottle that appears during the episode carries its route
`get_object_pose`	Pose of one observed body
`get_contacts`	Finger normal forces and peak contact forces between bodies
`get_devices`	Measured joint positions and rates of every device
`get_workspaces`	Whether the cart occupies each central transfer volume now and within a one-second projection
`get_requests`	Issued destination and priority requests with their history
`get_effective_requests`	Revision of each request in effect at the current time
`get_goals`	Declared destination geometry with the open range of each lid
`get_buffers`	Declared holding-pad geometry and capacity
`get_task`	Declared routes and the routes admitted so far with the required bottles
State sensors	
`observe_transfer_space`	State of the transfer volume of a route
`observe_goal_gate`	State of the lid of a lidded destination
`observe_feeder`	State of the conveyor feed of one workstation
`observe_arrivals`	State of new bottles on the feed of one workstation
`observe_destination`	State of the destination of one required bottle
`observe_order`	State of the delivery order of one supply-receiver flow
Control	
`move_to_pose`	Plans a motion of one arm and returns a token. Table 9 lists its options
`move_by_displacement`	Moves one arm from its measured TCP pose along world axes
`back_to_origin`	Moves one arm back to its start pose along a checked path
`set_gripper`	Sets the gripper width and effort from the next tick. `open_gripper` and `close_gripper` supply default values
`set_conveyor`	Drives one of the four conveyors at up to 5 rad/s
`assign_receiver`	Binds a prepared bottle to a receiver arm in dispatch workloads
`poll`	Returns the status of a motion
`cancel`	Stops a motion and holds the measured pose of the arm
`tick`	Advances the shared clock by 40 ms and returns a new snapshot
Helpers	
`geometry`	Checks caller-supplied poses against declared destination and pad geometry in one snapshot
`TickScheduler`	Runs policy branches that own disjoint arms and merges their ticks into one shared tick

B.2.6 CONTINGENCIES

Quadmanual defines six contingency events. Codes starting with R denote requests that change the task specification during execution. Codes starting with E denote physical events that move a device or add a bottle. Scenarios prescribe when events occur, either at fixed times or relative to task stages such as grasping or approaching the goal. The policy observes an event only through public readings and never reads the event schedule. Table 10 lists the events with the stages they affect in the representative task.

Table 10: Contingency events of the Quadmanual environment and the stages they affect in the representative task. Sensor names refer to Table 8.

Code	Event	Affected stage	State sensor
R1	The destination of a bottle is reassigned	Place of T1 to T6	`observe_destination`
R2	The delivery order of one flow is revised	Order of all subtasks	`observe_order`
E1	An extra bottle appears on a conveyor	New subtask T7	`observe_arrivals`
E2	The lid of a trash bin closes	Place of T1 and T2	`observe_goal_gate`
E3	The cart passes through the central aisle	Handover of T3, T4, and T6	`observe_transfer_space`
E4	Roller brakes jam a conveyor	Pick on that conveyor	`observe_feeder`

Each state sensor assigns a state to its contingency target. The environment design groups these states into response families. States in a family differ in how an event appears and share a prescribed response. Table 11 groups the states by family. The family can depend on the stage of the affected subtask. For example, the design prescribes waiting for a lid closure before the bottle enters the bin and classifies closure after entry as a failure. One event can also affect several subtasks at once. A single cart pass can block the handovers of T3, T4, and T6.

The family labels and prescribed responses belong to the environment design and are omitted from the public API book. The book lists the state names that each sensor can return, without their families or responses. A policy infers the response associated with a state from its name and execution outcomes.

B.3 Real-World Environment

Overview. The real-world environment consists of two AgileX Piper arms mounted side by side on a table (Figure 5). The world frame is the base of the first arm, and the base of the second arm lies 0.663 m away along the table. Each arm owns a zone of the table, and the two zones overlap in a shared band 0.14 m wide between the arms. A handover point above this band lets one arm pass an object to the other in the air. A skill call is assigned to an arm by the zone of its target, and a handover is inserted when a place target lies outside the zone or reach of the arm holding the object.

Hardware. Table 12 summarizes the hardware and control configuration. A single workstation runs the arm containers, camera acquisition, and perception service.

Table 12: Hardware and control configuration of the real-world environment.

Component	Specification
Robots and control	
Manipulators	Two AgileX Piper arms, six joints per arm
Grippers	Integrated parallel-jaw grippers, maximum opening 9.9 cm
Control software	ROS 2 Humble, AgileX `piper_ros`, and MoveIt (Coleman et al., 2014); separate container and ROS domain per arm
Communication	1 Mbps CAN link per arm
Motion settings	10% of velocity and acceleration limits; straight-line motions at 4 cm/s
Contact detection	Joint-torque-based detection at a 3 N threshold during placement; no wrist force-torque sensor
Cameras and calibration	
Top camera	Intel RealSense D435i; color and aligned depth at 1280×720, 30 Hz
Wrist cameras	One Intel RealSense D405 per arm; color and depth at 848×480, 30 Hz
Calibration	ChArUco hand-eye calibration; translation residuals of 1.3 mm for each wrist camera and 3.5 mm for the top camera. The top camera also registers the second arm's base in the world frame
Workstation	
CPU	AMD Ryzen 5 9600
Memory	32 GB
GPU	NVIDIA GeForce RTX 3060, 12 GiB

Perception. SAM 3 (Carion et al., 2025) segments objects in the top-camera color image. A vision-language model (GPT-5.4 mini) assigns names to the segments through Set-of-Mark prompting (Yang et al., 2023). Aligned depth gives each object's position in the world frame. The skills and scene reads use these object names and positions.

Task. The task `Cooperative Block Placement` requires placing three cubes on three plates and returning both arms to their home poses. Depending on the layout, zero to two cubes must be handed over between the arms. The instruction names each cube and its plate with the labels that perception produces for the episode. An episode succeeds when every cube rests on its plate, both grippers are empty, and both arms are home within 240 s. A collision, an exception, or a failure report ends the episode as a failure.

Policy API. Policies interact with the environment through the functions documented in a public API book, as in Quadmanual. Table 13 summarizes the available functions.

Table 13: Public policy interface of the real-world environment.

Function	Description
Perception and scene information	
`observe`	Refresh object detections and positions from the camera
`get_objects`	Return the currently observed object names
`get_task`	Return the instruction and object-to-target assignments
`get_scene`	Return object positions and the arms that can reach them
`get_arms`	Return each arm's workspace zone and held object
Manipulation and synchronization	
`pick`	Grasp a named object from above
`pick_side`	Grasp a named object from the side
`place_on`	Place the held object on a named target
`place_side`	Place a side-grasped object on a named target using a side approach
`place_at`	Place the held object at specified table coordinates
`home`	Return the selected arm or arms to their home poses
`handover`	Transfer an object to the other arm
`sync`	Wait for all queued arm motions to finish
Occupancy sensing	
`observe_occupancy(target)`	Return the occupancy state of the handover workspace or the region around a named object

We implement `observe_occupancy` using a hand-tracking monitor running at 10 Hz on images from the top camera. Each call returns the latest monitored state for the requested target.

Occupancy Contingency. The contingency is a human hand that temporarily or persistently occupies the handover workspace, a cube before its pick, or a plate before its place. Each scenario fixes the occupied target and the timing of the hand in advance, and the policy observes the hand only through the occupancy sensor. Table 14 lists the eight sensor states grouped by the response the environment design prescribes. A skill breaches the contingency when its gripper is at the target while the reading forbids entry. A breach stops the skill as a collision and fails the episode. The state `hand_stuck` requires a failure report within 30 s of its first reading. The provided demonstrations and evaluation protocol for RQ3 are described in Section D.3.

Table 14: Occupancy states of the real-world environment, grouped by the response prescribed in the environment design. The family labels are omitted from the public API book.

State	Observed condition
Proceed. The affected skill continues.	
`clear`	Nothing occupies the target
`hand_passed`	A previously detected hand has remained undetected in the camera image for at least 1 s
Wait. The affected skill stays out of the target until a proceed reading. The arm may keep or set down what it holds.	
`hand_nearby`	The hand moves near the target without reaching it
`hand_approaching`	The hand moves toward the target and reaches it within about one second
`hand_passing`	A previously detected hand has remained undetected in the camera image for less than 1 s
`hand_resting`	The hand rests on the target and leaves later
`unknown`	An error occurs in occupancy monitoring
Fail. The program stops and reports failure.	
`hand_stuck`	The hand rests on the target and does not leave

Table 11: Contingency states of the Quadmanual state sensors, grouped by the response prescribed in the environment design.

Event	State	Observed condition
Proceed. The affected subtask continues.		
R1	`destination_initial`	No destination request is in effect
R2	`order_initial`	No priority request is in effect
R2	`order_next_changed`	The new order moves only bottles that no arm holds yet
R2	`order_cancelled`	The priority request is canceled
E1	`no_new_item`	No new route appears
E2	`gate_open_stable`	The lid stays inside its open range for 0.2 s
E3	`cart_clear_quiet`	The volume is clear now and in the projection, and the cart moves slower than 5 mm/s
E3	`local_route`	The route uses a local handover that the cart cannot block
E4	`at_pickup`	The bottle reaches the pickup point
Wait. The affected stage pauses, and an arm keeps any bottle it holds.		
E1	`item_falling`	A new bottle is observed before it lands
E2	`gate_open_settling`	The lid has been inside its open range for less than 0.2 s
E2	`gate_closed`	The lid is outside its open range
E2	`gate_moving`	The lid is inside its open range but moves too fast to stay open over the next 80 ms
E4	`feeding`	The rollers turn without brake contact before the bottle reaches the pickup point
E4	`jammed`	The rollers stall under brake contact for 0.08 s
E4	`recovering`	The rollers have turned again without brake contact for less than 0.2 s
E4	`recovered`	The rollers have turned freely for 0.2 s before the bottle reaches the pickup point
Store. The supply arm places the held bottle on a free holding pad and regrasps it when the subtask may continue.		
R2	`order_held_demoted`	The bottle that the supply arm holds moves back in the order
E3	`cart_clear_moving`	The volume is clear now and in the projection, and the cart moves at 5 mm/s or faster
E3	`cart_approaching`	The volume is clear now but not in the projection
E3	`cart_passing`	The cart occupies the volume and moves
E3	`cart_holding`	The cart occupies the volume and stands still
E3	`cart_unknown`	The volume has no reading
Replan. The receiver arm redirects the place stage to the destination in effect. A canceled request restores the initial destination.		
R1	`destination_changed`	A new destination takes effect before release while the grasp holds
R1	`destination_cancelled`	The destination request is canceled
Add. The new bottle joins the required bottles as a new subtask at the end of its flow.		
E1	`item_landed`	A new bottle rests on the rollers
E1	`item_vanished`	An admitted bottle leaves the readings, and its delivery obligation remains
Fail. The environment design treats these conditions as terminal failures.		
R1	`destination_after_release`	The destination changes after release or after the grasp is lost
R2	`order_prefix_changed`	The new order changes bottles that a receiver has already taken
E2	`gate_lost_after_entry`	The lid leaves its open range after the bottle has entered the bin
E4	`feed_unobserved`	The roller readings are missing

C Experiment Details

C.1 Environments

RQ1 uses BiCoord (Peng et al., 2026), whose long-horizon bimanual tasks require the runtime to coordinate concurrent skills that interact through shared resources (Section B.1). RQ2 and the ablation use Quadmanual, where contingency events change the coordination requirements of a four-arm task during execution (Section B.2). RQ3 uses the real-world environment, where a human hand occupies the workspace of an upcoming skill (Section B.3).

C.2 Evaluation Details

Input programs. Each research question defines its own experimental conditions. Expert programs provide a reference for evaluating changes introduced by compilation. Sequentialized variants retain their skill calls while constraining compatible invocations to execute in sequence. The source programs and available interfaces are shared across methods within a comparison unless they form the experimental variable.

Agent configurations. Table 15 lists the model identifiers and agent settings. The agent configuration remains fixed within a method comparison unless it is the experimental variable. The availability of environment feedback and the limits on agent calls are declared for each stage.

Hardware configurations. Table 16 lists the GPU and CPU configurations tested for simulation rollouts. Each rollout uses an independent simulator process with concurrency limited by measured host capacity.

Table 15: Agent configurations.

Setting	Value
GPT-5.6 Luna	`gpt-5.6-luna`
GPT-5.6 Sol	`gpt-5.6-sol`
GPT-6 Astra	`gpt-6-astra`
Agent interface	Codex CLI 0.155.0
Effort comparisons (RQ1)	*low* / *medium* / *xhigh*
Effort comparisons (RQ2)	*low* / *medium* / *high*

Table 16: Tested hardware configurations.

NVIDIA GPU	CPU
GeForce RTX 4090 (24 GiB)	AMD Ryzen 9 7950X3D
GeForce RTX 5090 (32 GiB)	AMD Ryzen Threadripper PRO 9975WX
GeForce RTX 5090 (32 GiB)	AMD EPYC 7K62
GeForce RTX 5090 (32 GiB)	AMD EPYC 7453
RTX A6000 (48 GiB)	Intel Core i9-10980XE

The tested configurations use the following software versions.

- **CUDA runtime:** 12.1 on the GeForce RTX 4090 configuration and 12.8 on the other configurations.
- **NVIDIA driver:** 580.95.05 / 580.173.02 / 580.178.04 / 595.84 across the tested hosts.
- **Simulator:** RoboTwin 2.0 with SAPIEN 3.0.0b1.

Evaluation. Each RQ declares its task set and stopping criterion before execution. Development and validation use separate seeds. Evaluation seeds are excluded from the automated refinement loop. RQ1 uses evaluation instances previously encountered during method development, as disclosed in Appendix D.1. Compilation is compared with its source program on matched task instances. Refinement is evaluated at the checkpoints declared for the experiment. Evaluation fixes the selected program and resets episode state between trials. Its outcomes do not enter automated refinement or automated candidate selection.

Reporting. Task success is reported as a percentage with the number of evaluated trials stated as its denominator. Trajectory length is reported over successful trials and simultaneous movement percentage is reported where concurrent activity is defined by the environment. Section C.3 specifies

these metrics. We report task-level results and aggregate performance with the number of repetitions stated alongside the mean and standard deviation. RQ1 compares compiled programs with their source references. Results under contingencies are reported separately for each condition.

RQ1 reports the compilation cost of each task with the accounting described in Appendix D.1.

C.3 METRICS

Following BiCoord (Peng et al., 2026) we measure task completion through success rate. Trajectory length and simultaneous movement percentage characterize execution efficiency and concurrent activity.

Success Rate (SR). Success rate measures the fraction of evaluation rollouts that satisfy the task success criterion and is reported as a percentage. For N evaluation rollouts, let $y_i \in \{0, 1\}$ indicate whether rollout i succeeds. The success rate is

$$\mathrm{SR} = \frac{100}{N} \sum_{i=1}^{N} y_i. \tag{10}$$

Trajectory Length (TL). Trajectory length measures the number of timesteps required to complete a task. Let L_i denote the trajectory length of rollout i. TL is averaged over successful rollouts,

$$\mathrm{TL} = \frac{\sum_{i=1}^{N} y_i L_i}{\sum_{i=1}^{N} y_i}. \tag{11}$$

When no rollout succeeds, TL is undefined and the task is omitted from the TL average.

Simultaneous Movement Percentage (SMP). Simultaneous movement percentage measures the fraction of a trajectory during which both arms are active and is reported as a percentage. An arm is active when its end effector moves or its gripper remains closed. SMP therefore measures concurrent activity, including stationary holding. The motion-only variant counts only steps in which both arms apply a motion waypoint. Let $m_t^a \in \{0, 1\}$ denote the activity indicator of arm $a \in \{\mathrm{left}, \mathrm{right}\}$ at timestep t. For a trajectory of length L, SMP is

$$\mathrm{SMP} = \frac{100}{L} \sum_{t=1}^{L} m_t^{\mathrm{left}} m_t^{\mathrm{right}}. \tag{12}$$

C.4 AGENT ISOLATION AND REPRODUCIBILITY

All methods run their coding agents under the same isolation, so each agent receives only the information that its method provides. Each agent call runs inside a bubblewrap sandbox over a copy of the method repository. The sandbox exposes only this copy and the software that the agent needs to run. The agent therefore cannot access the simulator, the benchmark success checks, held-out outcomes, or the records of other runs. The repository copy is also the only location that the agent can modify. Of the agent's changes, only those to the part that the method declares editable carry over to the next checkpoint.

Each run also keeps the records needed to reproduce it. The run stores the experiment definition, the code version, and the command that started it. Each agent call keeps its prompt and the full record of its interaction. Every evaluated candidate from these calls, including rejected ones, is committed to a Git repository outside the sandbox. This repository also holds every checkpoint, so each evaluated program can be restored from it. Table 17 summarizes these measures together with the conditions of the evaluation population and the benchmark assets.

Table 17: Agent isolation and reproducibility measures shared by all methods.

Measure	Effect
Agent isolation	
Sandbox	The agent sees only its copy of the method repository and the software that it needs to run. It can modify only this copy.
Independent calls	Each call starts without the conversation history of earlier calls or other sessions.
Fixed agent setting	Every call specifies the model and the reasoning effort explicitly.
Evidence scope	The repository copy contains development traces and admission feedback. Validation contributes outcomes and metrics only. Held-out outcomes never enter it.
Edit scope	Only changes to the part that the method declares editable reach the next checkpoint.
Provenance and reproduction	
Run record	Each run stores its experiment definition, code version, and start command.
Agent record	Each agent call stores its prompt and the full record of its interaction.
Checkpoint history	Checkpoints are never modified once created. A Git repository stores every checkpoint and every evaluated candidate.
Evaluation machines	Evaluation machines receive the same code before each run. They run simulation only and never run the agent.
Evaluation population and benchmark	
Rejected seeds	Seeds that the environment rejects before any policy runs are excluded for every method.
Infrastructure failures	Rollouts interrupted by simulator or machine failures are rerun with the same program. They do not count as task failures.
Benchmark assets	All methods use the same asset repairs. For `Place Plate and Cup`, we restore missing plate contact annotations. For `Sweep Block`, we add the canonical grasp annotation and correct its position and orientation from the released geometry and frame metadata. Task code, physical meshes, and success checks remain unchanged.

D EXPERIMENT

D.1 EFFECTS OF RUNTIME COMPILATION

RQ1 evaluates preservation of source-program behavior during compilation and recovery of concurrent execution from sequential input. We compare compiler models across reasoning efforts using both the original BiCoord programs and their sequentialized variants.

Evaluation setting. We use the task programs provided with BiCoord (Peng et al., 2026) as the source reference. Both comparisons evaluate the same 18 tasks with 30 seeds per task. Compilation proceeds without environment feedback or subsequent refinement. Each compiled program is evaluated in three repetitions using the same seeds. These repetitions measure execution variability of fixed programs, not variability across independent compilations. Initializations rejected by the simulator are excluded from task-level SR.

Public API documentation. The compiler receives the task program and instruction together with documentation of the available observations and actions. Motion commands move either end effector to a target pose or by a relative displacement. The API also provides grasping and placement primitives with separate commands for gripper opening and closing. Execution functions support synchronized bimanual motion or independent arm commands with progress monitoring and cancellation. Additional commands hold the current configuration or return an arm to its initial pose. State queries provide robot observations and oracle object poses together with annotated grasp and placement points.

Table 18: Success rates (%) after compilation of the original BiCoord programs across reasoning efforts. Entries report the mean and sample standard deviation over three execution repetitions. Each repetition averages task-level SR over all 18 tasks with compilation failures reported as 0% execution success. The source reference achieves 61.3%.

Compiler	*low*	*medium*	*xhigh*
GPT-5.6 Luna	10.2±0.5	35.0±0.7	43.6±0.9
GPT-5.6 Sol	36.7±0.7	43.8±1.2	51.6±0.7
GPT-6 Astra	52.9±0.9	54.1±1.7	56.4±1.6

Compiler Model and Reasoning Effort. SR increases with reasoning effort across the three evaluated models (Table 18). GPT-6 Astra at *xhigh* effort achieves the highest SR among compiled programs but does not attain the SR of the source programs.

Sequential-source compilation. Sequentialized programs execute paired arm motions successively while preserving the primitive calls. The compiler receives no annotations identifying the original motion pairs. All compiled conditions have lower success-conditioned TL and higher SMP than the sequential source (Table 19). Because the successful task instances can differ across conditions, these aggregate differences do not isolate the effect of concurrent execution. SR exceeds the source only for GPT-6 Astra at *low* and *medium* effort. GPT-6 Astra at *medium* effort achieves the highest SR at 50.0% compared with 45.7% for the sequential source.

Table 19: Compilation of 18 sequentialized BiCoord programs across reasoning efforts. Entries report the mean and sample standard deviation of task averages over three execution repetitions. Compilation failures are reported as 0% execution success. TL and SMP under *Successful rollouts* exclude failures and omit tasks without successes.

Program	Effort	All rollouts		Successful rollouts	
		SR (%)	SMP (%)	TL (steps)	SMP (%)
Sequential source		45.7±1.1	58.0±0.1	7161.5±7.4	57.2±0.2
GPT-5.6 Luna	*low*	22.9±0.3	71.7±0.0	4094.8±10.4	76.1±0.5
	medium	31.0±0.5	72.5±0.1	4890.9±11.8	72.9±0.2
	xhigh	41.6±1.5	70.7±0.2	4912.4±84.9	76.1±0.5
GPT-5.6 Sol	*low*	41.7±1.1	78.3±0.1	4731.5±111.2	80.6±0.4
	medium	37.7±0.5	86.5±0.1	4642.1±170.2	85.0±0.3
	xhigh	45.3±0.4	82.9±0.0	5129.6±8.2	84.5±0.2
GPT-6 Astra	*low*	46.8±1.4	71.9±0.2	5563.1±15.5	75.2±0.2
	medium	50.0±0.7	74.4±0.0	5566.8±10.6	75.2±0.2
	xhigh	44.9±0.5	76.9±0.5	5716.8±174.6	77.8±1.5

Task-level results. GPT-6 Astra at *medium* effort increases SR on `Jigsaw` from 54.4% to 70.0%. In the same condition SR on `Build Tower with Blocks` decreases from 68.9% to 23.3%.

Excluding stationary holding yields a simultaneous-motion percentage of 40.0% for GPT-6 Astra at *medium* effort compared with 0.0% for the sequential source. Changes to program branches and to the set of successful executions also affect aggregate TL. The TL comparison therefore does not isolate the contribution of concurrency on matched successful executions.

Reported conditions. Several conditions required repeated compilation attempts before a program was admitted for evaluation. The reported results characterize the resulting fixed programs. They do not estimate the probability that a single compilation attempt succeeds. Evaluation instances were also reused during method development, including diagnosis of execution-interface defects. These instances support a comparison of the evaluated programs, with generalization to previously unused instances left untested.

RQ1 compiler-cost summaries count calls that completed normally and have measured token usage. They exclude interrupted calls, earlier restarted runs, manual development work, and simulator execution. These summaries therefore cover recorded, completed compiler calls rather than total development cost.

D.2 Coordination Refinement under Contingencies

Tables 22 and 23 provide the numerical results for Figure 3. They report *continuous measurements* and *descriptive event tags* separately. Both settings use *GPT-5.6 Sol* in Quadmanual. At each checkpoint, SR is computed over the same 120 held-out cases: 111 contingency cases and nine nominal cases. Each run receives a single task program covering subtasks T1, T3, and T4 of the representative long-horizon task. Execution feedback from contingencies arising in these different subtasks is used to refine the shared task policy. The policy is evaluated in separate episodes for each subtask, with 40 held-out cases per subtask. `RaP` and all baselines follow the same setup. Round 0 evaluates the initial program before refinement. Rounds 1–3 evaluate the program after each refinement round.

Development, validation, and held-out cases. Development episodes are divided into three disjoint sets of 11, 10, and six episodes, introduced over successive refinement rounds. Validation uses three separate nominal episodes, one per subtask, to check whether proposed revisions preserve task performance. The held-out episodes are disjoint from both development and validation, and their outcomes never enter automated refinement or candidate selection.

Table 20 summarizes the conditions targeted in each refinement round and the composition of the held-out set. The evaluation set stays fixed across all methods, runs, and refinement rounds, and includes conditions introduced in later rounds.

Table 20: RQ2 case composition by subtask. Parentheses in the development columns give the number of episodes introduced in that round. The held-out column lists disjoint groups totaling 40 episodes per subtask: 37 contingency cases and three nominal cases. Conditions name the primary state targeted by a case; other states may also occur during execution.

Subtask	Round 1	Round 2	Round 3	Held-out composition
T1	Jammed feed; new bottle landed; destination changed after release (3)	Moving lid; falling bottle; new bottle on a holding pad (3)	Closed lid; lid closure after bottle entry (2)	Two variants of each of the eight development conditions; three variants of each of seven additional conditions listed below; three nominal cases
T3	Feed jam, recovery in progress, recovered feed, missing feed reading; moving cart outside the transfer volume, approaching cart (6)	Cart passing, passed, or unobserved; stalled bottle (4)	Cart holding in the transfer volume; stuck cart (2)	Four jam variants; three variants of each of the other eleven development conditions; three nominal cases
T4	Moving cart outside the transfer volume; approaching cart (2)	Cart passing, passed, or unobserved (3)	Cart holding in the transfer volume; stuck cart (2)	Six variants each of cart passing and cart passed; five variants of each of the other five development conditions; three nominal cases
Total	11	10	6	120

For T1, the seven additional primary conditions are a recovered feed, multiple landed bottles, a bottle arriving while the feed is idle, a reopened lid, a lid settling within its open range, a canceled destination request, and an added bottle disappearing from observation. These conditions are not primary targets of the T1 development cases; this does not imply that every such state is absent from

their trajectories. Held-out variants change event onset and duration, lid opening, bottle position, arrival count and location, or destination. Cart variants additionally change travel direction, speed, and distance, and the timing or duration of missing workspace readings. Nominal variants change the initial bottle position without introducing a contingency.

Observation examples. Table 21 illustrates the two observation formats using recorded calls to `observe_goal_gate` for the T1 bin and `observe_transfer_space` for the T3 handover volume. The numeric and tag examples come from separate executions of the same episode recipes and illustrate the information available to a policy. The numeric profile returns a `values` dictionary without a state name; the tag profile returns a `state` name without a prescribed response. Response families and success criteria remain part of the evaluator and are not included in either observation or supplied to the refinement agent.

Table 21: Recorded observation examples for the same types of conditions. Numeric entries show selected fields from the returned `values` dictionary; tags show the returned `state` field. Angles are in radians, angular rates in radians per second, distances in meters, speeds in meters per second, and durations in seconds.

Condition	Continuous measurements	Descriptive event tag

Lid closure (T1)	`lid_angle_rad: 1.0263`
	`lid_rate_rad_s: -0.5236`
	`open_margin_rad: -0.0209`
		`state: gate_closed`

Cart passage (T3)	`rail_gap_m: 0.0`
	`cart_speed_m_s: 0.1342`
	`occupied_s: 0.04`
		`state: cart_passing`

The accompanying `appendix/rq2_case_protocol.json` provides the complete development, validation, and held-out case lists, event triggers and parameters, evaluation criteria, and full observation payloads with their source records.

Runs and aggregation. Each method–effort configuration uses the same three seeds as the main figure. `AS` starts each run from the unchanged source program. The other methods compile or translate the source anew. Effort-specific rows report the mean and sample standard deviation across the three runs. Overall rows report the mean and sample standard deviation across all nine runs, giving equal weight to each effort and compilation seed. This standard deviation reflects variation across both efforts and seeds.

Table 22: Refinement SR (%) under *continuous measurements* with *GPT-5.6 Sol*. Effort-specific rows report the mean and sample standard deviation over the three seeds used in Figure 3. Overall reports the mean and sample standard deviation across all nine runs from the three reasoning efforts. Round 0 precedes refinement. Round k follows k refinement rounds.

Method	Effort	SR (%) ↑ by refinement round			
		0	1	2	3
RaP	*low*	23.6±28.3	27.2±34.4	31.9±34.9	31.9±34.9
	medium	45.0±2.5	51.4±3.4	52.2±6.7	51.9±6.7
	high	20.8±25.5	48.3±4.4	64.7±17.0	64.7±17.0
	Overall	29.8±22.3	42.3±20.8	49.6±24.4	49.5±24.3
AS	*low*	33.3±0.0	33.6±1.3	38.3±3.8	40.8±2.9
	medium	33.3±0.0	38.6±3.8	40.0±1.4	40.0±1.4
	high	33.3±0.0	40.6±2.1	40.6±2.1	40.6±2.1
	Overall	33.3±0.0	37.6±3.9	39.6±2.5	40.5±2.0
SG	*low*	1.1±1.9	23.9±20.8	26.9±23.4	28.1±21.5
	medium	2.2±1.9	37.8±6.8	43.3±7.6	45.6±10.8
	high	6.4±5.3	35.0±2.9	42.5±9.2	42.5±9.2
	Overall	3.2±3.8	32.2±12.8	37.6±15.4	38.7±15.2
BT	*low*	2.8±1.3	8.6±9.2	8.6±9.2	8.6±9.2
	medium	8.6±12.8	11.1±17.1	11.1±17.1	11.9±16.5
	high	28.1±13.5	35.8±6.7	39.2±8.8	39.7±7.9
	Overall	13.1±14.8	18.5±16.6	19.6±18.2	20.1±18.0

Table 23: Refinement SR (%) under *descriptive event tags* with *GPT-5.6 Sol*. Effort-specific rows report the mean and sample standard deviation over the three seeds used in Figure 3. Overall reports the mean and sample standard deviation across all nine runs from the three reasoning efforts. Round 0 precedes refinement. Round k follows k refinement rounds.

Method	Effort	SR (%) ↑ by refinement round			
		0	1	2	3
RaP	*low*	13.3±17.3	41.7±8.8	50.3±8.3	46.1±5.7
	medium	41.1±13.5	50.3±16.7	53.9±11.6	60.3±20.0
	high	48.9±18.4	48.9±18.4	53.3±12.6	65.0±14.5
	Overall	34.4±21.6	46.9±13.8	52.5±9.7	57.1±15.3
AS	*low*	33.3±0.0	40.8±0.0	40.8±0.0	40.8±0.0
	medium	33.3±0.0	37.8±3.4	37.8±3.4	37.8±3.4
	high	33.3±0.0	40.0±1.4	49.4±14.9	50.0±14.5
	Overall	33.3±0.0	39.5±2.3	42.7±9.3	42.9±9.2
SG	*low*	2.2±1.9	23.3±17.3	36.4±5.3	36.4±5.3
	medium	13.3±17.3	35.3±2.9	38.3±1.7	48.9±16.9
	high	0.0±0.0	13.3±17.3	32.2±1.9	35.3±6.5
	Overall	5.2±10.7	24.0±15.6	35.6±4.0	40.2±11.5
BT	*low*	0.8±0.8	7.8±12.8	8.9±14.7	7.8±12.8
	medium	9.4±12.0	10.8±14.4	10.8±14.4	10.8±14.4
	high	23.3±10.8	40.6±1.0	40.0±1.7	42.2±1.0
	Overall	11.2±12.7	19.7±18.4	19.9±18.3	20.3±19.1

Failure analysis. Under *continuous measurements*, an AS run repairs an observed cart-approach failure by retreating the receiving arm and rechecking clearance before transfer. The same repair fails on a held-out timing variation in which the cart approaches after transfer begins, because clearance is not maintained during motion. This example illustrates a limitation of the repair's generalization, rather than an absence of improvement on all held-out cases.

Initial compilation can also introduce execution errors independently of contingency handling. A run of RaP at *low* effort completes delivery but terminates because of an invalid runtime call, which subsequent refinement corrects. An SG run cannot begin execution because a required input is not connected. These failures account for part of the initial performance loss relative to the source program.

Under *descriptive event tags*, a run of RaP at *medium* effort learns to recheck the bin lid immediately before release and retain the bottle until the lid reopens. This repair resolves an observed closure and generalizes to a held-out variation with a later onset and longer duration. The same held-out case remains unresolved in the runs at *low* effort.

Cart-direction visualization. Figure 4 shows physical re-executions of policies from the study checkpoints under reversed cart motion. The workspace-clearance rule in RaP was already present in its initial compiled program, so this example illustrates the execution of retained coordination semantics rather than their acquisition through refinement. In the physical re-execution, RaP completes the manipulation task but enters the transfer region before the cart-passed condition of the environment, and therefore does not satisfy the contingency criterion. For SG, the cart contacts the held bottle during continued simulation after the program has terminated. No further policy commands are issued in this interval. These frames illustrate the resulting physical behavior and are separate from the held-out measurements reported in Figure 3.

D.3 Real-World Coordination Transfer

Setup and provided case. RQ3 uses the `Cooperative Block Placement` task in the real-world environment described in Section B.3. The source task program does not query the occupancy sensor. Each method receives a single case comprising two handover demonstrations. In the successful demonstration, the arm pauses while the sensor reports `hand_resting` and resumes when it reports `hand_passed`. In the failed demonstration, the arm moves while the sensor still reports `hand_resting`, and the program terminates. From this case, each method performs a single refinement step without further execution.

Evaluation. The refined programs are held fixed during evaluation. The held-out cases apply the occupancy states in Table 14 to the handover workspace under a different goal, layout, and timing, and to cubes before pick and plates before place, with five cases per state. Three nominal episodes without occupancy check that a refined program preserves the task.

D.4 Ablation

Tables 24 and 25 report the results underlying Table 1 separately for *continuous measurements* and *descriptive event tags*. Both settings use *GPT-5.6 Sol* at *low*, *medium*, and *high* reasoning efforts. Each run evaluates the final checkpoint after three refinement rounds on 120 held-out cases, comprising 111 contingency cases and nine nominal cases. SR is the percentage of these 120 cases completed successfully.

Ablation setting. AS executes the unchanged source program and repairs its code directly, without the runtime kernel, coordination semantics, or hypotheses. `AS + Coord. Guidance` additionally maintains coordination semantics definitions in each round and provides them to the code repair. These definitions are neither executed nor assessed as hypotheses. `AS + Event Kernel` starts from the compiled program of each seed with its state representations moved into the task code, and its refinement repairs only that code. Coordination rules that the compiled program registers remain unchanged. `RaP w/o hypothesis` retains the runtime kernel and state representations without hypotheses. Its refinement agent can inspect these representations and author coordination rules. The reported values compare final performance across these refinement procedures.

Aggregation. These tables use seeds 1–3 to align with the three-seed RQ2 evaluation. For `RaP`, we use the same final-checkpoint results as Figure 3, with one run per seed and effort. For variants without the runtime kernel, the seeds are independent runs of the unchanged source program. For variants with the kernel, they are the three compiled programs. Repeated refinement runs for the same seed are averaged before computing statistics across seeds. Under *continuous measurements*, the first seed contains four refinement runs at *low* effort and five at *medium* effort for `RaP w/o hypothesis`. All remaining entries contain one refinement run per seed. The last column of each table gives the mean and sample standard deviation over the three seeds. Each Avg. row first averages the three efforts within a seed. Table 1 first averages the two event information settings within each seed and effort, then summarizes the three seeds. Its Avg. column also averages the three efforts within each seed before computing the mean and sample standard deviation.

Table 24: Detailed SR (%) under *continuous measurements* for the measured variants in Table 1. Columns 1–3 report seeds 1–3 after three refinement rounds. `RaP` uses the same runs as the RQ2 evaluation in Figure 3. The last column reports the mean and sample standard deviation across seeds. Avg. rows average the three reasoning efforts within each seed.

Variant	Effort	Seed			Avg. ± SD
		1	2	3	
`AS`	*low*	42.5	37.5	42.5	40.8±2.9
	medium	38.3	40.8	40.8	40.0±1.4
	high	40.8	42.5	38.3	40.6±2.1
	Avg.	40.6	40.3	40.6	40.5±0.2
`AS + Coord. Guidance`	*low*	39.2	65.8	52.5	52.5±13.3
	medium	55.0	44.2	59.2	52.8±7.7
	high	45.0	55.0	57.5	52.5±6.6
	Avg.	46.4	55.0	56.4	52.6±5.4
`AS + Event Kernel`	*low*	55.0	15.8	0.0	23.6±28.3
	medium	43.3	42.5	48.3	44.7±3.2
	high	60.8	50.0	48.3	53.1±6.8
	Avg.	53.1	36.1	32.2	40.5±11.1
`RaP w/o hypothesis`	*low*	58.8	28.3	0.0	29.0±29.4
	medium	49.7	42.5	47.5	46.6±3.7
	high	63.3	50.0	43.3	52.2±10.2
	Avg.	57.2	40.3	30.3	42.6±13.6
`RaP`	*low*	69.2	26.7	0.0	31.9±34.9
	medium	52.5	58.3	45.0	51.9±6.7
	high	60.8	50.0	83.3	64.7±17.0
	Avg.	60.8	45.0	42.8	49.5±9.8

Table 25: Detailed SR (%) under *descriptive event tags* for the measured variants in Table 1. Columns 1–3 report seeds 1–3 after three refinement rounds. `RaP` uses the same runs as the RQ2 evaluation in Figure 3. The last column reports the mean and sample standard deviation across seeds. Avg. rows average the three reasoning efforts within each seed.

Variant	Effort	Seed			Avg. ± SD
		1	2	3	
`AS`	*low*	40.8	40.8	40.8	40.8±0.0
	medium	34.2	40.8	38.3	37.8±3.4
	high	66.7	40.8	42.5	50.0±14.5
	Avg.	47.2	40.8	40.6	42.9±3.8
`AS + Coord. Guidance`	*low*	58.3	72.5	65.8	65.6±7.1
	medium	56.7	33.3	46.7	45.6±11.7
	high	35.8	33.3	33.3	34.2±1.4
	Avg.	50.3	46.4	48.6	48.4±2.0
`AS + Event Kernel`	*low*	40.8	44.2	40.8	41.9±1.9
	medium	40.8	68.3	45.8	51.7±14.6
	high	55.0	56.7	52.5	54.7±2.1
	Avg.	45.6	56.4	46.4	49.4±6.0
`RaP w/o hypothesis`	*low*	49.2	43.3	46.7	46.4±2.9
	medium	55.0	42.5	41.7	46.4±7.5
	high	86.7	48.3	48.3	61.1±22.1
	Avg.	63.6	44.7	45.6	51.3±10.7
`RaP`	*low*	52.5	41.7	44.2	46.1±5.7
	medium	50.0	83.3	47.5	60.3±20.0
	high	81.7	58.3	55.0	65.0±14.5
	Avg.	61.4	61.1	48.9	57.1±7.1