CODE AS POLICIES VIA GEOMETRY-CONDITIONED SEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Synthesizing robot-control programs from language requires geometric information that object poses alone do not provide, including functional directions and interaction locations. We introduce LOGIC, a geometry-conditioned policy synthesis framework that maps task instructions to a search space structured around geometric hypotheses and task-relevant control parameters. For each hypothesis, an LLM constructs an execution plan, identifies the parameters needed to carry it out, and generates parameterized control programs. Environment rollouts guide both parameter optimization and selection among hypothesis-conditioned programs. Execution feedback also refines the search space itself: failures guide revisions to the geometric hypotheses and task-relevant parameterizations that define subsequent policy candidates. New programs are then generated and optimized within this revised space. This combination of search-space construction, execution-based optimization, and feedback-driven refinement enables the synthesis of policies that compute end-effector poses and gripper commands without predefined task-level skill APIs. Experiments across manipulation and tool-use tasks, using both simulator-provided geometry and geometry estimated from RGB-D observations, show substantial improvements over code-as-policies and evolutionary baselines.

1 INTRODUCTION

Robotic manipulation has advanced rapidly through vision-language-action (VLA) models such as OpenVLA and π_0 (Kim et al., 2024; Black et al., 2024), hierarchical systems such as Hi Robot (Shi et al., 2025), and modular perception–control approaches such as ReKep (Huang et al., 2024). Recent evaluations of GPT-6 Astra further demonstrate manipulation through language-model-generated end-effector commands (Menon et al., 2026). Yet successful executions alone provide limited assurance that a robot will behave reliably across objects, configurations, and repeated deployments. Failures may arise from incorrect geometric assumptions, poorly chosen action parameters, or even hazardous instructions (Robey et al., 2024; Sun et al., 2026). Dependable deployment therefore calls for policies whose assumptions can be inspected, failures diagnosed, and behavior systematically tested before reuse.

Code as policies offers a complementary route: an LLM synthesizes an executable controller that can be inspected and reused (Liang et al., 2023; Singh et al., 2022). Its branches, geometric computations, and parameters provide concrete targets for review, testing, and correction (Bosio & Mueller, 2025). A synthesized policy can react to new observations without generating a new action through the LLM at every step, amortizing synthesis cost across executions. Such programs can also use learned perception to extract object information from real-world images.

Producing such a controller becomes challenging when observations leave task-relevant geometry unspecified. Without task-level APIs such as `grasp(cup)` or `open(drawer)`, a program must compute end-effector poses and gripper commands itself. Yet object poses do not identify functional directions or interaction locations: using a hockey stick requires deciding which direction points toward its head and how far along the shaft to grasp; opening a door requires assumptions about its hinge and opening motion. These interpretations and numerical choices are coupled, so errors in either can invalidate the resulting actions.

Execution feedback provides a way to evaluate these choices and improve the resulting programs. Methods such as FunSearch (Romera-Paredes et al., 2024), Evolution of Heuristics (Liu et al., 2024),

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.

AlphaEvolve (Novikov et al., 2025), and MLES (Hu et al., 2025) guide program generation by measured performance, while numerical optimization further supports control-policy synthesis (Bosio et al., 2025). However, when task-relevant geometry remains implicit, semantic interpretations and parameterization are entangled in the code. Numerical tuning explores only the parameters already exposed; revising geometric assumptions or introducing missing parameters requires further program generation. This can make low-level policy search inefficient under limited rollout budgets.

Rather than focusing solely on generating or evolving individual programs, we introduce LOGIC to *construct, search, and refine a structured space of control programs*. The LLM *constructs* this space around geometric hypotheses, using anticipatory planning to identify the control parameters needed under each. The framework *searches* this space by generating hypothesis-conditioned programs and using rollouts to optimize their parameters and select candidates by task success. Failure feedback then guides revisions to the hypotheses and parameterizations, *refining* the space from which subsequent programs are generated.

We evaluate LOGIC on tasks from RLBench (James et al., 2020), Robosuite (Zhu et al., 2020), and LIBERO (Liu et al., 2023) using simulator-provided states or geometry estimated from RGB-D observations. Results show substantial gains over the evaluated baselines, with ablations supporting geometric branching, explicit parameter identification, and feedback-driven refinement.

2 MOTIVATION

Low-level control requires geometric information beyond an object's observed placement. Sparse state interfaces in manipulation environments commonly provide object positions and orientation quaternions (Gu et al., 2023; James et al., 2020). A 6D pose specifies a position $p \in \mathbb{R}^3$ and quaternion $q \in \mathbb{H}$ mapping an object-local frame to the world frame, but does not identify functional directions or interaction locations. An estimated center likewise leaves the placement of handles and hinges, and the resulting opening motion, ambiguous.

Generating a controller from these observations therefore involves two coupled choices: how to interpret the task-relevant geometry and which control parameters and values that interpretation requires. LOGIC makes these choices explicit through geometric hypotheses and associated parameterizations, defining a structured space of control programs that execution feedback can search and reshape.

The robotic hockey task in Fig. 1 illustrates how these choices interact. To grasp the stick and then strike the ball, the policy must locate the handle and head from the observed CoM position p and orientation quaternion q. This requires both an interpretation of which local direction points toward each component and estimates of their distances from the CoM; neither is specified by the pose alone.

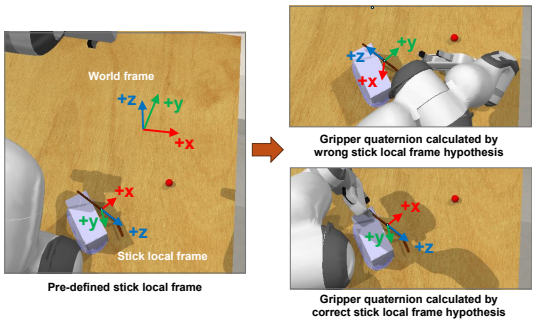


Figure 1: Local-frame ambiguity in hockey: an incorrect interpretation yields a failed grasp (upper right); a matching interpretation succeeds (lower right).

To act, the LLM must hypothesize a local direction $v_{\text{obj}} \in \mathbb{R}^3$ pointing toward the stick's head and identify task-relevant parameters. The direction is mapped to the world frame via $R(q)v_{\text{obj}}$, where $R(q)$ is the rotation matrix corresponding to q. The observed rotation and the hypothesized local alignment also determine the target gripper orientation. For grasping, the policy computes a target grasp position

$$p_{\text{grasp}} = p - d_{\text{handle}} \cdot (R(q)v_{\text{obj}}),$$

where d_{handle} determines the precise placement of the gripper along that direction. After grasping, the same abstraction yields the stick head position

$$p_{\text{head}} = p + d_{\text{head}} \cdot (R(q)v_{\text{obj}}),$$

which guides the striking motion toward the ball, where d_{head} determines the relative distance from the CoM to the stick's head.

Errors in either component cause cascading failures. As shown in Fig. 1, misinterpreting the semantic direction (e.g., using $v_{\text{obj}} = [0, 0, -1]^T$ instead of $v_{\text{obj}} = [0, 0, 1]^T$) inverts the geometry, leading to incorrect grasping and missed strikes. Similarly, inaccurate parameter estimates d_{handle} and d_{head} misalign execution even when the direction is correct.

Sparse observations often leave both task-relevant geometric relationships and the physical quantities needed for control unspecified. This motivates constructing a structured space of control programs around geometric hypotheses and associated parameterizations, then using execution feedback to search and refine that space.

3 Problem Setup

3.1 Programmatic Control without Skill APIs

We consider an embodied agent tasked with executing natural language instructions $\mathcal{I}$ in an unstructured environment. At each time step t, the agent receives an observation $o_t \in \mathcal{O}$ whose explicit object geometry is sparse. Depending on the observation interface, this may include object centers or full 6D poses, together with images, camera calibration, and robot state. A pose quaternion maps an object's local frame to the world frame, while an estimated center provides a positional reference even when the local orientation is unreliable. LOGIC uses the available information to formulate geometric hypotheses and construct policies. Task-relevant properties, such as functional directions, interaction locations, and articulation, remain only partially specified by these observations.

The action space $\mathcal{A}$ is continuous and operates at the low kinematic level, consisting of 6-DoF target end-effector poses and gripper states:

$$a_t = [x, y, z, q_x, q_y, q_z, q_w, g] \in SE(3) \times [-1, 1],$$

where (x, y, z) denotes the target end-effector position in the world frame, (q_x, q_y, q_z, q_w) is the corresponding orientation quaternion, and $g \in [-1, 1]$ is a normalized gripper opening/closing command.

To execute a_t, the robot's control stack converts the target pose into joint-level commands. For example, Cartesian-pose-to-joint inverse kinematics (IK) computes a target joint configuration, which a motion controller drives the robot toward. A gripper controller executes the opening or closing command g. The synthesized program uses updated observations to compute subsequent targets.

Our goal is to synthesize a programmatic policy $\Pi \in \mathcal{L}$ from a natural language instruction $\mathcal{I}$, where $\mathcal{L}$ denotes executable Python programs tailored for geometric reasoning and low-level control. Unlike API-driven Code-as-Policies (Liang et al., 2023), which map instructions to high-level skill primitives (e.g., `grasp(obj)`), we operate in a *skill-free* setting, where Π acts as a direct, closed-loop controller that computes geometric transformations and target poses directly from observations. Once synthesized, the policy operates solely on observations:

$$\Pi : \mathcal{O} \to \mathcal{A}.$$

While programmatic policies offer interpretability and enable symbolic reasoning (Verma et al., 2018), zero-shot synthesis of such policies for low-level control is fundamentally challenged by the gap between sparse observations $\mathcal{O}$ and the latent geometric structure required for task execution.

3.2 Latent Geometry and Environmental Ambiguity

Effective control requires geometric information about how to interact with objects that is not fully specified by the observation interface. This includes functional directions, the placement of interaction parts, and articulation, together with the numerical quantities needed to compute executable motions. We categorize the resulting ambiguity into two types:

1. **Geometric Semantic Ambiguity:** Ambiguity about task-relevant functional, spatial, or articulation relationships left unspecified by observations. Examples include which direction points toward a tool head, on which side of an object a handle or hinge lies, and how a door opens. Hypotheses express these relationships in coordinates supported by the available observations; interpreting object-local axes is one instance of this broader ambiguity.

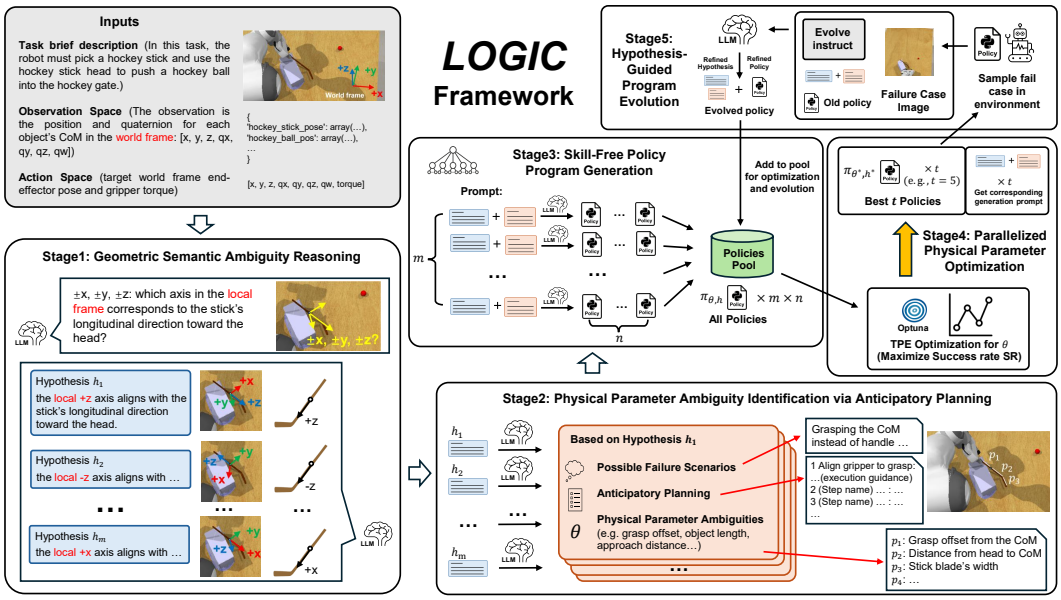


Figure 2: LOGIC constructs a program search space through geometric hypothesis branching and planning-based parameter identification, searches it through program generation and parameter optimization, and refines it using execution feedback.

2. **Physical Parameter Ambiguity:** Unknown numerical quantities needed to instantiate control under a geometric hypothesis. These include distances from the CoM to a handle or tool head, grasp offsets, hinge displacements, sweep radii, and motion clearances. For example, a semantic hypothesis specifies the side on which a hinge lies and the opening motion, while its associated parameters specify the hinge offset and the extent of that motion.

We organize candidate interpretations of this latent geometric structure and their associated parameters into a hypothesis space:

$$\mathcal{H} = \mathcal{H}_{\text{sem}} \times \mathcal{H}_{\text{param}}.$$

Under this formulation, low-level program synthesis becomes a joint search over the program Π and a geometric hypothesis with associated parameters $h \in \mathcal{H}$.

Objective. Formally, our objective is to select a program $\Pi^* \in \mathcal{L}$ and associated geometric hypothesis and parameters $h^* \in \mathcal{H}$ that maximize task success under sparse observations:

$$(\Pi^*, h^*) = \arg \max_{\Pi \in \mathcal{L},\, h \in \mathcal{H}} \; \mathbb{E}_{o_{1:T} \sim \mathcal{E}} \left[\mathbf{1}\big(\Pi(\mathcal{I}, o_{1:T}; h) \models \mathcal{G}\big) \right], \tag{1}$$

where $\mathcal{E}$ denotes the environment dynamics, T is the maximum execution steps and $\mathbf{1}(\cdot)$ indicates satisfaction of the task goal $\mathcal{G}$ specified by the instruction $\mathcal{I}$.

4 Method

We propose a framework that structures policy search around explicit geometric hypotheses. The LLM proposes alternative geometric interpretations and task-relevant parameterizations, which guide parallel program generation and optimization. Execution feedback ranks the resulting policies and guides their refinement. An overview of our framework is presented in Figure 2, and the complete pseudocode detailing the execution pipeline is provided in Algorithm 1 within Appendix B.

4.1 Geometric Semantic Ambiguity Reasoning

LOGIC constructs hypotheses about the task-relevant geometry left unspecified by the observation interface before generating policies. These hypotheses describe functional directions, interaction

locations, and articulation: for example, which direction points toward a tool head, where a handle lies relative to an object center, or how a door moves about its hinge. The LLM uses the task description and observation interface to determine which relationships require hypotheses and how to express them using the available geometric information.

Given the task description and observation interface, the LLM first identifies the geometric ambiguities relevant to execution (Appendix I.1). Let $U = \{u_1, \ldots, u_v\}$ denote these ambiguities. For each u_i, it proposes a finite set of plausible interpretations V_i using its prior knowledge (Huang et al., 2022a; Ahn et al., 2022) (Appendix I.2). We construct the hypothesis space as

$$\mathcal{H}_{\text{sem}} = V_1 \times \cdots \times V_v, \tag{2}$$

where each $\alpha \in \mathcal{H}_{\text{sem}}$ combines assumptions about the task-relevant geometry. The resulting m hypotheses define branches for subsequent planning, parameter identification, and policy generation (Sec. 4.3).

Hypotheses are expressed in a frame supported by the observations. When reliable 6D object poses are available, the LLM can associate functional directions with local axes and use the observed quaternion to compute world-frame directions and gripper orientations. For example, a hockey-stick hypothesis may associate the head-facing direction with local $+x$, $-x$, or $+y$ (Fig. 1). When an oriented bounding box (OBB) provides a useful center estimate but an unreliable orientation, LOGIC uses the center as a positional reference and formulates hypotheses about hinge sides, handle-offset directions, and opening motions directly in the world frame (Appendix E.1). In either case, the hypotheses guide the same planning, parameter optimization, and execution-based refinement procedure.

4.2 Physical Parameter Ambiguity Identification via Anticipatory Planning

Physical parameter ambiguity arises from task-critical quantities that are not specified by the observations, such as approach distances, grasp offsets, and clearance margins. Under each hypothesized geometric interpretation $\alpha \in \mathcal{H}_{\text{sem}}$, we prompt the LLM to refine the task description into an execution plan conditioned on α (Appendix I.3). Anticipating how each action could fail helps identify the quantities that must be exposed for optimization. The LLM produces a structured plan

$$\mathcal{P}_\alpha = \big[(s_1, e_1), (s_2, e_2), \ldots, (s_k, e_k)\big], \tag{3}$$

where each s_i denotes a subtask (e.g., *"Align gripper to grasp"*) and e_i provides execution guidance under the given hypothesis. For example, if the stick's local $+z$ axis is hypothesized to point toward its head, e_i may instruct the robot to align the gripper with that axis and approach at an offset along local $-z$ from the center of mass, toward the handle, to avoid premature contact with the head.

The explanations e_i reference spatial quantities whose values must be chosen for successful execution. We therefore prompt the LLM to extract these quantities as explicit variables $\theta \in \mathbb{R}^d$, rather than committing to numerical values during planning. In the example, the displacement along local $-z$ becomes an `approach_offset` parameter. The identified variables define the parameter space $\mathcal{H}_{\text{param}}$ associated with the hypothesis; different interpretations can require different quantities and parameterizations.

This process turns implicit numerical assumptions into optimization variables. The geometric hypothesis determines how the plan interprets the object, while anticipatory planning identifies which quantities the resulting program needs. Their numerical values are selected through execution in the subsequent optimization stage (Sec. 4.4).

4.3 Skill-Free Policy Program Generation

For each geometric interpretation $\alpha \in \mathcal{H}_{\text{sem}}$ and its identified parameters, the LLM generates n policy programs $\pi_{\theta,\alpha}$ guided by the anticipatory plan $\mathcal{P}_\alpha$ (Appendix I.4). Each parameter is accompanied by an inferred range $[\min, \max]$, defining its search bounds. These ranges reflect the LLM's estimates of feasible distances, clearances, and object scales; execution-based optimization then searches for effective values within them.

The generated programs obey a *skill-free* constraint: they compute coordinate transformations, end-effector targets, and gripper commands directly from observations using the action interface in Section 3. Consequently, the geometric interpretation must be implemented in the program's

computations, rather than delegated to a predefined task-level skill API. The identified quantities appear as tunable parameters instead of fixed numerical choices embedded in the control logic.

Generation is independent across the m hypothesis branches, so all $N = mn$ initial candidates can be generated in parallel before evaluation. Appendix I.6 illustrates the Hockey policy structure (Fig. 6) and provides the complete example code.

4.4 Parallelized Physical Parameter Optimization

Having generated $m \times n$ parameterized policies $\{\pi_{\theta,\alpha}\}$ across the hypothesis space $\mathcal{H}$, we deploy them simultaneously in a parallelized black-box optimization pipeline. Each policy is anchored to a specific geometric interpretation $\alpha \in \mathcal{H}_{\text{sem}}$, while its associated task-relevant parameters $\theta \in \mathcal{H}_{\text{param}}$ are bounded within sensible reference ranges.

For each program with a fixed hypothesis α, we optimize the corresponding parameters θ using the Tree-structured Parzen Estimator (TPE) algorithm implemented in Optuna (Akiba et al., 2019). The objective is to maximize the expected success rate of the policy:

$$\theta^*(\alpha) = \arg\max_{\theta \in \mathcal{H}_{\text{param}}} \mathbb{E}\left[S(\pi_{\theta,\alpha})\right].$$

Across all hypotheses, this yields a set of optimized candidates $\{(\alpha, \theta^*(\alpha))\}$. The final solution is selected by evaluating their performance and choosing the best-performing pair:

$$(\alpha^*, \theta^*) = \arg\max_{\alpha \in \mathcal{H}_{\text{sem}}} \mathbb{E}\left[S(\pi_{\theta^*(\alpha),\alpha})\right]. \tag{4}$$

This parallel evaluation ranks policies generated under different geometric interpretations. A branch's score reflects the combined effect of its geometric hypothesis, generated program, and optimized parameters. We select the top t policies across all branches by task success, retaining the combinations that provide the strongest starting points for refinement in Sec. 4.5.

4.5 Hypothesis-Guided Program Evolution

Even after parameter optimization, candidates may exhibit failures caused by their geometric assumptions, parameterization, or control logic. We use execution-guided program evolution (Novikov et al., 2025; Liu et al., 2024; Hu et al., 2025) to refine the top t candidates whose success rates satisfy $S(\pi) \geq \tau_{\text{min}}$. Each candidate retains its geometric hypothesis, anticipatory plan, and parameterization, providing an explicit context for diagnosing failures. Refinement therefore starts from hypothesis-conditioned programs that have already shown some task success.

For each selected candidate, we collect a failure trace and build a record containing the current policy code, success rate, and terminal observation image (Hu et al., 2025; Shinn et al., 2023). The LLM receives this record together with the original generation prompt, which preserves the assumptions behind the candidate (Appendix I.5). It can revise geometric assumptions, modify control logic, introduce or adjust parameters, and refine their search bounds. These revisions reshape the space from which subsequent programs are generated and optimized; the search is not restricted to tuning the original parameter vector. Revised candidates undergo parameter optimization and selection again.

For example, failure feedback in Hockey leads to two changes (Fig. 2). *Collision-aware trajectory generation* introduces a lateral offset that shifts the gripper sideways to avoid the stick's head, and a back offset that retracts the approach point along the handle. Together, these parameters define a multi-stage "descend-and-slide" trajectory with greater pre-grasp clearance. *Feedback-driven execution* updates the stick's observed position after grasping to compensate for slippage and adds a goal-based stopping condition to prevent overshoot. The example illustrates how a failed execution can reveal missing parameters and motivate a different execution structure. Appendix I.6 provides the resulting policy illustration and full code.

5 EXPERIMENTS

5.1 EXPERIMENT SETTING

Environments. We evaluate our framework under two observation settings: simulator-provided object states and RGB-D perception. The first setting comprises 18 RLBench tasks (James et al., 2020) and three Robosuite tasks (Zhu et al., 2020). The perception setting emulates real-world conditions in which ground-truth object poses are unavailable: policies obtain geometric information through an RGB-D perception pipeline, without access to simulator-provided object poses. This evaluation comprises 20 tasks from RLBench, Robosuite, and LIBERO; the perception pipeline is described in Appendix E.

The 18 RLBench tasks are grouped into single-object manipulation, long-horizon sequential manipulation, and precise tool use and indirect manipulation. Table 1 reports results for all 18 tasks; Appendix D.1 describes the categories.

Baselines. For the RLBench and Robosuite evaluations with simulator-provided object states, we compare our approach against four state-of-the-art frameworks: **Code as Policies (CaP)** (Liang et al., 2023), the foundational paradigm that directly maps language instructions to executable Python scripts for robotic control; **Evolution of Heuristics (EOH)** (Liu et al., 2024) and **AlphaEvolve** (Novikov et al., 2025), which are advanced evolutionary search methods that iteratively mutate and recombine code logic to discover optimal heuristic programs; and **MLES** (Hu et al., 2025), a recent evolutionary framework that incorporates multimodal visual feedback to guide the mutation of programmatic policies for control tasks. All baselines are strictly constrained to operate under the identical skill-free setting as our method, directly outputting low-level controller commands (6-DoF end-effector poses and gripper states) without the aid of predefined skill APIs. In their conventional implementations, ambiguous continuous variables (e.g., approach distances, grasp offsets, and clearance heights) are not explicitly parameterized. Instead, the LLM hardcodes these values as numerical constants within the generated policy body (Liang et al., 2023; Bosio et al., 2025). To ensure a fair comparison, we apply a post-generation regularization step to all baselines. We use regular expressions to extract hardcoded numerical constants from the generated programs, lift them into tunable parameters (e.g., `param0`, `param1`), and optimize them using the same parallel black-box routine (Optuna TPE) as in our method.

For the perception setting, we compare with CaP and CaP-Agent0, the reference agent provided by CaP-X (Fu et al., 2026), using the same perception tools and control interface. CaP receives TPE optimization of its numerical parameters, whereas CaP-Agent0 revises code through execution feedback within each episode. The evaluation procedures are detailed in Appendix E.3.

Implementation Details. For code generation and reflective reasoning in the comparisons using simulator-provided object states, we utilize the `gpt-5-mini` model across both our framework and the four baselines. For parameter optimization, we employ the Tree-structured Parzen Estimator (TPE) algorithm provided by the Optuna framework (Akiba et al., 2019), which explores the continuous parameter landscape over multiple episodic rollouts. The number of trials used in TPE, as well as the evaluator settings and other hyperparameters, can be found in Appendix H.

Evaluation Metrics. Our primary metric is the final Success Rate (SR). For the RLBench and Robosuite experiments with simulator-provided object states, we conduct evaluations across 5 independent random seeds, computing the SR over 50 randomized held-out environment initializations per seed. Mean SR and standard deviation across seeds are reported in Table 1 for all 18 RLBench tasks and Appendix F.3 for Robosuite. The perception evaluation uses 50 held-out episodes per task, as described in Appendix E.3.

5.2 RESULTS WITH SIMULATOR-PROVIDED STATES

Table 1 reports results for all 18 RLBench tasks. Our method consistently outperforms all competitors across all three categories. Even without the final visual-feedback evolution stage, LOGIC (w/o Evol.) reaches an average success rate of 65%, substantially exceeding every fully evolved baseline. This indicates that much of the advantage is already present in geometric hypothesis branching and parameterized policy generation and optimization. Evolution further improves the average success rate to 71%. The evolutionary methods AlphaEvolve and EOH yield only modest gains over CaP

Table 1: RLBench success rates (mean ± std) on all 18 tasks over 5 seeds, with 50 held-out episodes per seed. The final row averages task means.

Task Name	LOGIC (Full)	LOGIC (w/o Evol.)	CaP	AlphaEvolve	EOH	MLES
Task Type: Single Object Manipulation						
OpenBox	$\mathbf{0.72} \pm 0.21$	0.66 ± 0.20	0.47 ± 0.14	0.48 ± 0.19	0.50 ± 0.24	0.21 ± 0.11
OpenDrawer	$\mathbf{0.90} \pm 0.03$	0.88 ± 0.04	0.24 ± 0.21	0.18 ± 0.19	0.46 ± 0.13	0.39 ± 0.11
OpenOven	$\mathbf{0.58} \pm 0.16$	0.53 ± 0.14	0.14 ± 0.08	0.39 ± 0.17	0.23 ± 0.10	0.20 ± 0.10
OpenWashingMachine	$\mathbf{0.48} \pm 0.13$	0.46 ± 0.12	0.27 ± 0.14	0.26 ± 0.05	0.21 ± 0.08	0.23 ± 0.15
SlideCabinetOpen	$\mathbf{0.64} \pm 0.12$	0.59 ± 0.14	0.27 ± 0.18	0.34 ± 0.22	0.31 ± 0.26	0.31 ± 0.19
TakeMoneyOutSafe	$\mathbf{0.67} \pm 0.35$	0.62 ± 0.34	0.08 ± 0.09	0.18 ± 0.12	0.07 ± 0.06	0.07 ± 0.05
TakeToiletRollOffStand	$\mathbf{0.94} \pm 0.03$	0.90 ± 0.08	0.72 ± 0.17	0.82 ± 0.08	0.76 ± 0.18	0.78 ± 0.08
TakeFrameOffHanger	$\mathbf{0.90} \pm 0.03$	0.85 ± 0.05	0.74 ± 0.13	0.78 ± 0.08	0.83 ± 0.05	0.75 ± 0.05
Task Type: Long-Horizon Sequential Manipulation						
BlockPyramid	$\mathbf{0.74} \pm 0.06$	0.62 ± 0.12	0.17 ± 0.08	0.15 ± 0.09	0.08 ± 0.03	0.18 ± 0.06
PutItemInDrawer	$\mathbf{0.65} \pm 0.23$	0.57 ± 0.29	0.02 ± 0.03	0.18 ± 0.13	0.12 ± 0.15	0.07 ± 0.09
PutShoesInBox	$\mathbf{0.71} \pm 0.18$	0.61 ± 0.14	0.08 ± 0.05	0.14 ± 0.12	0.07 ± 0.08	0.13 ± 0.09
PutMoneyInSafe	$\mathbf{0.74} \pm 0.13$	0.71 ± 0.09	0.17 ± 0.05	0.28 ± 0.12	0.17 ± 0.14	0.33 ± 0.20
StackCups	$\mathbf{0.81} \pm 0.10$	0.70 ± 0.14	0.45 ± 0.17	0.68 ± 0.10	0.56 ± 0.16	0.52 ± 0.07
TakeCupOutFromCabinet	$\mathbf{0.60} \pm 0.12$	0.58 ± 0.11	0.23 ± 0.11	0.24 ± 0.13	0.33 ± 0.16	0.41 ± 0.14
Task Type: Precise Tool Use & Indirect Manipulation						
HitBallWithQueue	$\mathbf{0.82} \pm 0.16$	0.77 ± 0.17	0.03 ± 0.04	0.14 ± 0.11	0.06 ± 0.06	0.06 ± 0.07
Hockey	$\mathbf{0.76} \pm 0.19$	0.62 ± 0.23	0.28 ± 0.15	0.24 ± 0.08	0.20 ± 0.02	0.24 ± 0.13
PutUmbrellaInUmbrellaStand	$\mathbf{0.38} \pm 0.12$	0.36 ± 0.14	0.26 ± 0.08	0.20 ± 0.13	0.12 ± 0.07	0.12 ± 0.07
ScoopWithSpatula	$\mathbf{0.83} \pm 0.09$	0.73 ± 0.10	0.28 ± 0.10	0.43 ± 0.14	0.46 ± 0.20	0.41 ± 0.14
Average (all 18 tasks)	**0.71**	0.65	0.27	0.34	0.31	0.30

through code mutation, failing to address the "curse of dimensionality" inherent in low-level 6-DoF control; MLES, despite using visual feedback similar to our final stage, performs worse than both, since sparse failure signals produce unstable or misleading updates that drift toward suboptimal solutions. These results support geometry-conditioned policy search as an effective way to generate candidates for subsequent refinement. Appendix D.2 analyzes sample and computational efficiency. Appendix F details the held-out protocol, the VLM-as-judge control, and per-task compute budgets; Appendix E.4 reports robustness to pose noise.

Ablation. We evaluate the contribution of each component by removing key stages of the pipeline. Removing either geometric semantic reasoning or physical parameter ambiguity identification substantially degrades performance, supporting the contribution of both components to effective policy synthesis. Appendix D.3 gives full ablation and per-task results, and Appendix D.4 studies how the framework scales across different LLM backbones.

Robosuite. LOGIC achieves the highest mean success rate on `Door`, `Lift`, and `NutAssemblyRound` against the same four baselines, using Robosuite's continuous Operational Space Controller (Appendix F.3).

5.3 Results under RGB-D Perception

Under the perception setting, policies rely on geometric information estimated from RGB-D observations rather than ground-truth object poses, emulating the observation constraints of real-world deployment. We evaluate 7 RLBench, 2 Robosuite, and 11 LIBERO tasks using the perception and control interface adopted from CaP-X (Fu et al., 2026). SAM3 segmentation, depth back-projection, and oriented bounding-box fitting provide an estimated object center. LOGIC expresses hypotheses about interaction locations, hinges, and opening motions directly in the world frame in such environments. In our experience, the absence of reliable object orientation limits fine-grained tool alignment. Appendix E discusses this limitation, describes the perception pipeline, and illustrates how failure feedback can revise an opening-motion hypothesis.

All methods receive the same task descriptions, API documentation, and perception tools. Both LOGIC and CaP optimize policy parameters with TPE and select a policy using optimization-set success before held-out evaluation. CaP-Agent0 instead generates and executes code within each episode, using execution and visual feedback. Table 7 reports the LLM and environment-interaction budgets of these search procedures.

Table 2: Success rates on eight selected tasks under RGB-D perception, measured on 50 held-out episodes per task. The final row reports the mean success rate across all 20 tasks. All methods share the perception tools and control interface. Table 6 in Appendix E.3 reports the complete results.

Benchmark	Task	LOGIC	CaP	CaP-Agent0
RLBench	open the microwave	**0.86**	0.00	0.04
RLBench	open the drawer	**0.86**	0.66	0.14
RLBench	slide the cabinet door open	**0.54**	0.44	0.22
Robosuite	stack the cube	**1.00**	0.96	0.90
Robosuite	lift the cube	**0.98**	**0.98**	0.78
LIBERO	turn on the stove	**0.94**	0.58	0.10
LIBERO	open the top drawer of the cabinet	**0.98**	0.02	0.02
LIBERO	put the tomato sauce in the basket	**1.00**	0.94	0.44
Average (all 20 tasks)		**0.86**	0.58	0.44

Table 2 presents eight tasks spanning the three benchmarks and several interaction types. LOGIC achieves the highest mean success rate across the 20-task evaluation. Its gains on microwave and drawer opening are consistent with the need to infer contact locations and articulation that an estimated center does not specify. These results support the effectiveness of searching over interaction hypotheses and their associated control parameters under incomplete perceptual observations. Appendix E.3 includes the complete results, budgets, and additional Agent0 feedback and TPE controls; Appendix E.4 separately examines sensitivity to pose noise.

6 Related Work and Conclusion

Code-as-Policies and Programmatic Control. Code-as-policies methods synthesize executable robot controllers (Liang et al., 2023; Singh et al., 2022), with extensions to situated program generation (Singh et al., 2022), task-to-code mapping (Wang et al., 2023), and reward generation for reinforcement learning (Ma et al., 2023; Yu et al., 2023). Ahn et al. (2025) use neuro-symbolic verification to improve the reliability of generated code. These approaches typically compose predefined primitives or specify rewards for learned policies. Our focus is on synthesizing the geometric computations and control parameters needed to produce end-effector targets without task-level skill APIs.

Appendix A expands the related work in a broader context.

Conclusion. LOGIC constructs, searches, and refines a space of control programs organized by geometric hypotheses and task-relevant parameters. Evaluations across RLBench, Robosuite, and LIBERO support this formulation for policy synthesis from sparse object states and RGB-D-derived geometry, producing reusable controllers without task-level skill APIs.

7 Limitations

Our evaluation is limited to the manipulation of rigid objects in simulation, and transfer to physical robots remains untested. In addition, unreliable object orientation estimates limit performance on tasks requiring precise tool alignment. Further limitations are discussed in Appendix C.

Reproducibility Statement

We provide the source code for LOGIC in the supplementary materials. Detailed hyperparameter settings are provided in Appendix H.

AI Use Statement

Large language models are a component of the proposed method, as described in Section 4. AI tools were utilized in the drafting of this manuscript strictly to improve language fluency, word choice, and overall logical flow. The authors maintain complete oversight and responsibility for the final scientific content.

References

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

Sanghyun Ahn, Wonje Choi, Junyong Lee, Jinwoo Park, and Honguk Woo. Towards reliable code-as-policies: A neuro-symbolic framework for embodied task planning. *arXiv preprint arXiv:2510.21302*, 2025.

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 2623–2631, 2019.

OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, et al. π_0: A vision-language-action flow model for general robot control. *arXiv preprint arXiv:2410.24164*, 2024.

Serena Booth, W Bradley Knox, Julie Shah, Scott Niekum, Peter Stone, and Alessandro Allievi. The perils of trial-and-error reward design: misdesign through overfitting and invalid task specifications. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 5920–5929, 2023.

Carlo Bosio and Mark W. Mueller. Synthesizing interpretable control policies through large language model guided search. In *American Control Conference (ACC)*, 2025. URL `https://arxiv.org/abs/2410.05406`.

Carlo Bosio, Matteo Guarrera, Alberto Sangiovanni-Vincentelli, and Mark W Mueller. Combining large language models and gradient-free optimization for automatic control policy synthesis. *arXiv preprint arXiv:2510.00373*, 2025.

Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. *arXiv preprint arXiv:2307.15818*, 2023.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Pete Florence, Corey Lynch, Andy Zeng, Oscar A Ramirez, Ayzaan Wahid, Laura Downs, Adrian Wong, Johnny Lee, Igor Mordatch, and Jonathan Tompson. Implicit behavioral cloning. In *Conference on robot learning*, pp. 158–168. PMLR, 2022.

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.

Jiayuan Gu, Fanbo Xiang, Xuanlin Li, Zhan Ling, Xiqiang Liu, Tongzhou Mu, Yihe Tang, Stone Tao, Xinyue Wei, Yunchao Yao, et al. Maniskill2: A unified benchmark for generalizable manipulation skills. *arXiv preprint arXiv:2302.04659*, 2023.

Dylan Hadfield-Menell, Smitha Milli, Pieter Abbeel, Stuart J Russell, and Anca Dragan. Inverse reward design. *Advances in neural information processing systems*, 30, 2017.

Ankur Handa, Arthur Allshire, Viktor Makoviychuk, Aleksei Petrenko, Ritvik Singh, Jingzhou Liu, Denys Makoviichuk, Karl Van Wyk, Alexander Zhurkevich, Balakumar Sundaralingam, et al. Dextreme: Transfer of agile in-hand manipulation from simulation to reality. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5977–5984. IEEE, 2023.

Erik Hemberg, Stephen Moskal, and Una-May O'Reilly. Evolving code with a large language model. *Genetic Programming and Evolvable Machines*, 25(2):21, 2024.

Qinglong Hu, Xialiang Tong, Mingxuan Yuan, Fei Liu, Zhichao Lu, and Qingfu Zhang. Multimodal llm-assisted evolutionary search for programmatic control policies. *arXiv preprint arXiv:2508.05433*, 2025.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International conference on machine learning*, pp. 9118–9147. PMLR, 2022a.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022b.

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

Stephen James, Zicong Ma, David Rovick Arrojo, and Andrew J Davison. Rlbench: The robot learning benchmark & learning environment. *IEEE Robotics and Automation Letters*, 5(2):3019–3026, 2020.

Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, et al. Scalable deep reinforcement learning for vision-based robotic manipulation. In *Conference on robot learning*, pp. 651–673. PMLR, 2018.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, et al. Openvla: An open-source vision-language-action model. *arXiv preprint arXiv:2406.09246*, 2024.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O Stanley. Evolution through large models. In *Handbook of evolutionary machine learning*, pp. 331–366. Springer, 2023.

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. IEEE, 2023.

Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. LIBERO: Benchmarking knowledge transfer for lifelong robot learning. *arXiv preprint arXiv:2306.03310*, 2023.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. *arXiv preprint arXiv:2401.02051*, 2024.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023.

Achu Menon, Sravanthi Machcha, Sabrina Zou, Tzu Kit Chan, and Jay Chooi. GPT-6 Astra on robotic manipulation. Robocurve evaluation report, September 2026. URL https://openai.robocurve.org/gpt-6-astra/. Published September 4, 2026.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Alexander Robey, Zachary Ravichandran, Vijay Kumar, Hamed Hassani, and George J. Pappas. Jailbreaking LLM-controlled robots. *arXiv preprint arXiv:2410.13691*, 2024. URL https://arxiv.org/abs/2410.13691.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Pratyusha Sharma, Balakumar Sundaralingam, Valts Blukis, Chris Paxton, Tucker Hermans, Antonio Torralba, Jacob Andreas, and Dieter Fox. Correcting robot plans with natural language feedback. *arXiv preprint arXiv:2204.05186*, 2022.

Lucy Xiaoyang Shi, Brian Ichter, Michael Equi, Liyiming Ke, Karl Pertsch, Quan Vuong, James Tanner, Anna Walling, Haohuan Wang, Niccolo Fusai, et al. Hi robot: Open-ended instruction following with hierarchical vision-language-action models. *arXiv preprint arXiv:2502.19417*, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

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. *arXiv preprint arXiv:2209.11302*, 2022.

Edward Sun, Sravanthi Machcha, Sabrina Zou, Tzu Kit Chan, and Jay Chooi. RoboHarm: Do frontier robot policies refuse unsafe instructions? Robocurve evaluation report, September 2026. URL https://robocurve.org/roboharm/. Published September 18, 2026.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press Cambridge, 1998.

Stefanie Tellex, Nakul Gopalan, Hadas Kress-Gazit, and Cynthia Matuszek. Robots that use language. *Annual Review of Control, Robotics, and Autonomous Systems*, 3(1):25–55, 2020.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033. IEEE, 2012.

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International conference on machine learning*, pp. 5045–5054. PMLR, 2018.

Yuki Wang, Gonzalo Gonzalez-Pumariega, Yash Sharma, and Sanjiban Choudhury. Demo2code: From summarizing demonstrations to synthesizing code via extended chain-of-thought. *Advances in Neural Information Processing Systems*, 36:14848–14956, 2023.

Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, et al. Language to rewards for robotic skill synthesis. *arXiv preprint arXiv:2306.08647*, 2023.

Andy Zeng, Maria Attarian, Brian Ichter, Krzysztof Choromanski, Adrian Wong, Stefan Welker, Federico Tombari, Aveek Purohit, Michael Ryoo, Vikas Sindhwani, et al. Socratic models: Composing zero-shot multimodal reasoning with language. *arXiv preprint arXiv:2204.00598*, 2022.

Enshen Zhou, Qi Su, Cheng Chi, Zhizheng Zhang, Zhongyuan Wang, Tiejun Huang, Lu Sheng, and He Wang. Code-as-monitor: Constraint-aware visual programming for reactive and proactive robotic failure detection. *arXiv preprint arXiv:2412.04455*, 2024.

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

A Related Work

Large Language Models for Robotic Planning. The emergence of Large Language Models (LLMs) has revolutionized high-level robotic task planning by enabling agents to reason over complex, natural language instructions (Brown et al., 2020; Tellex et al., 2020). Early efforts like SayCan (Ahn et al., 2022) and Inner Monologue (Huang et al., 2022b) leveraged LLMs to decompose long-horizon tasks into sequences of pre-defined skills, grounding language in robotic affordances. Subsequent work has expanded this paradigm to include multimodal reasoning (Zeng et al., 2022), hierarchical planning (Huang et al., 2022a), and self-correction through natural language feedback (Sharma et al., 2022). While these methods excel at semantic reasoning, they often rely on fixed skill libraries, which limits their ability to adapt to tasks that require precise, low-level coordination. Our work builds upon these planning capabilities but extends them to direct low-level control, bypassing the need for pre-existing skill primitives.

Code-as-Policies and Programmatic Control. The "code-as-policies" paradigm, where LLMs generate executable programs to control robots, has gained significant traction (Liang et al., 2023; Singh et al., 2022). Models like Code as Policies (CaP) use LLMs to write Python scripts that orchestrate robotic behaviors by calling API functions. This approach also been extended to include situated program generation (Singh et al., 2022), task-to-code mapping (Wang et al., 2023), and automated reward engineering for reinforcement learning, as demonstrated by Eureka (Ma et al., 2023; Yu et al., 2023). To mitigate the limited environmental grounding and lack of robustness in dynamic or partially observable settings, Ahn et al. (2025) introduced a neuro-symbolic framework that employs symbolic verification to ensure the reliability of LLM-generated code. However, these frameworks typically assume a "thick" API layer or provide rewards for training black-box neural policies. In contrast, programmatic policies in our work are "skill-free", requiring the LLM to perform direct geometric and physical reasoning to output low-level 6-DoF poses (Verma et al., 2018).

LLM-Guided Evolutionary Search and Discovery. Automated discovery of algorithms through LLM-guided evolution has emerged as a powerful technique for scientific and mathematical discovery (Romera-Paredes et al., 2024; Novikov et al., 2025). Systems like FunSearch (Romera-Paredes et al., 2024) use an iterative process to discover mathematical witnesses, while AlphaEvolve (Novikov et al., 2025) and Evolution of Heuristics (EOH) (Liu et al., 2024) automate the discovery of optimization heuristics. In the robotics domain, MLES (Hu et al., 2025) has explored evolving programmatic policies. Our work advances this line of research by introducing a systematic ambiguity resolution mechanism, which prevents the evolutionary search from being derailed by ambiguous environmental information—a common failure mode in prior methods (Lehman et al., 2023; Hemberg et al., 2024).

Reinforcement Learning for Dexterous Manipulation. Deep reinforcement learning remains the dominant approach for training policies in high-dimensional, continuous action spaces (Sutton & Barto, 1998; Schulman et al., 2017). RL has achieved human-level performance in dexterous in-hand manipulation (Andrychowicz et al., 2020; Handa et al., 2023) and vision-based control (Kalashnikov et al., 2018; Florence et al., 2022). However, RL's reliance on extensive reward shaping (Hadfield-Menell et al., 2017; Booth et al., 2023) and its "black-box" nature pose challenges for transparency and safety. Our framework aims to achieve the performance levels of RL without dense reward shaping while maintaining the interpretability of programmatic policies. By using LLMs to perform structured search over environmental hypotheses, we bridge the gap between high-level reasoning and low-level dexterity (James et al., 2020).

A.1 Relation to Learning-Based and VLM-Centric Approaches

Table 3: Conceptual comparison of LOGIC with representative learning-based and VLM-centric approaches to robotic manipulation.

Method	Observation	Robot data	Task-level skills	Policy representation
LOGIC (ours)	object centers/poses	execution rollouts	none	control program
End-to-end VLA (OpenVLA, π_0)	raw images	demonstrations	none	neural policy
Hierarchical (Hi Robot)	raw images	demonstrations	learned low-level policy	language subgoals + VLA
ReKep	RGB-D	no task-specific data	none	keypoint constraints
Code-as-Policies	perception APIs	no task-specific training	interface-dependent	control program

VLA models such as RT-2 (Brohan et al., 2023), OpenVLA (Kim et al., 2024), and π_0 (Black et al., 2024) learn mappings from images and language to actions using robot demonstrations. Hi Robot (Shi et al., 2025) adds a high-level VLM that interprets instructions and feedback, issuing language subgoals to a learned low-level VLA. Modular approaches expose other useful intermediate representations: ReKep (Huang et al., 2024) optimizes motions subject to visually grounded keypoint constraints, and Code-as-Monitor (Zhou et al., 2024) generates constraints for execution monitoring. Recent evaluations of GPT-6 Astra also demonstrate agents that repeatedly interpret camera observations and issue end-effector commands (Menon et al., 2026). These approaches illustrate different allocations of reasoning, perception, and control. Table 3 summarizes representative interfaces and policy representations; its robot-data column concerns robot training or task-specific synthesis, and all methods may rely on pretrained foundation models.

LOGIC produces a reusable control program through execution-guided synthesis, without task-specific demonstrations or a pretrained motor policy. Its distinctive mechanism is the construction and refinement of a search space organized by geometric hypotheses and their associated parameters. The program computes 6-DoF end-effector targets and gripper commands from available observations, and a standard controller realizes them. This representation exposes assumptions, branches, and parameters for inspection and testing, and avoids repeated LLM calls for action generation once synthesis is complete. Learned perception can remain part of the observation interface. Relative to ReKep, LOGIC explores geometric assumptions through policy rollouts, using sparse object centers or poses instead of requiring detected relational keypoints. Its synthesis therefore requires an executable environment for repeated evaluation. Our experiments compare code-generation and program-search methods under shared interfaces; the VLA and hierarchical comparison here is conceptual, since we do not evaluate matched versions of their training data, embodiment, and action interfaces. The results establish improved program synthesis within the evaluated settings, while the explicit program provides a basis for inspection and validation.

Skill-free action space. The synthesized policy outputs a target end-effector pose in $SE(3)$ together with a gripper command; the environment's existing control stack executes these targets through inverse kinematics and motion planning or operational-space control. The LLM generates the policy code, while standard controllers handle motion execution. This mirrors real robot stacks, in which a mature component (e.g., MoveIt) handles Cartesian-pose-to-joint IK and the policy only specifies *which* pose to command. Reaching a specified pose is essentially a solved capability; the hard problem, and the one LOGIC addresses, is deciding the correct pose from sparse observations. "Skill-free" accordingly means that the policy does not call task-level primitives such as `grasp(obj)` or `open(drawer)` that already encode task semantics and hide the geometric reasoning; a generic `move(pose)` command is not a skill in this sense, because it carries no task-relevant semantics. This is precisely why our setting is strictly harder than API-driven Code-as-Policies.

B LOGIC ALGORITHM

In Algorithm 1, we detail the execution pipeline of the LOGIC framework. We denote the initial natural language task brief description as $\mathcal{I}$, the sparse observation space as $\mathcal{O}$, the environment as $\mathcal{E}$, and the used Large Language Model as $\mathcal{M}$. The ultimate objective is to synthesize an optimized, skill-free programmatic policy Π^*. During the reasoning phase, the LLM extracts geometric ambiguities U and candidate interpretations V to construct a discrete semantic hypothesis space $\mathcal{H}_{\text{sem}}$. For each specific hypothesis $\alpha \in \mathcal{H}_{\text{sem}}$, the framework formulates an α-conditioned improved task description $\mathcal{I}_\alpha$ and an anticipatory execution plan $\mathcal{P}_\alpha$. This plan yields explicitly parameterized continuous variables θ and their search bounds, guiding the generation of an initial parameterized policy $\pi_{\theta,\alpha}$ stored within a candidate pool $\mathcal{P}_{\text{pool}}$. Following black-box parameter optimization, the framework extracts the optimally tuned parameters θ^* and filters the candidates into an elite set $\mathcal{P}_{\text{elite}}$. Finally, over K evolutionary rounds (r), each policy π in the $\mathcal{P}_{\text{elite}}$ is visually debugged using a failure execution trace's terminal frame, generating a structurally refined policy π_{refined}. These refined policies form a new pool $\mathcal{P}_{\text{new}}$ that undergoes subsequent parameter tuning to dynamically update the elite set, culminating in the selection of the most robust policy. Please note that this algorithm provides a condensed high-level overview of the pipeline; for a comprehensive discussion of the specific methodological details, please refer to Section 4.

Algorithm 1 LOGIC Execution Pipeline

Require: Task description $\mathcal{I}$, Observation space $\mathcal{O}$, Environment $\mathcal{E}$, LLM $\mathcal{M}$
Ensure: Optimized programmatic policy Π^*

// Phase 1: Geometric Semantic Ambiguity Reasoning
$U \leftarrow \mathcal{M}$.identify_geometric_ambiguities$(\mathcal{I}, \mathcal{O})$
$V \leftarrow \mathcal{M}$.generate_candidate_interpretations(U)
$\mathcal{H}_{\text{sem}} \leftarrow$ Cartesian_Product(V) ▷ Construct discrete hypothesis branches
// Phase 2 & 3: Anticipatory Planning & Skill-Free Policy Program Generation
$\mathcal{P}_{\text{pool}} \leftarrow \emptyset$ ▷ Policies Pool
for each hypothesis $\alpha \in \mathcal{H}_{\text{sem}}$ **in parallel do**
    $\mathcal{I}_\alpha \leftarrow \mathcal{M}$.improve_task_description$(\mathcal{I}, \mathcal{O}, \alpha)$
    $\mathcal{P}_\alpha \leftarrow \mathcal{M}$.generate_anticipatory_plan$(\mathcal{I}_\alpha, \mathcal{O}, \alpha)$
    θ, bounds $\leftarrow \mathcal{M}$.extract_continuous_parameters$(\mathcal{P}_\alpha)$
    $\pi_{\theta,\alpha} \leftarrow \mathcal{M}$.generate_skill_free_policy_code$(\mathcal{P}_\alpha, \theta, \mathcal{I}_\alpha, \mathcal{O})$
    $\mathcal{P}_{\text{pool}} \leftarrow \mathcal{P}_{\text{pool}} \cup \{\pi_{\theta,\alpha}\}$
// Phase 4: Parallelized Physical Parameter Optimization
$\mathcal{P}_{\text{elite}} \leftarrow \emptyset$
Fine tune θ for each policy in $\mathcal{P}_{\text{pool}}$ using TPE (Optuna)
$\mathcal{P}_{\text{elite}} \leftarrow$ Top t policies with tuned θ^* and their corresponding α
// Phase 5: Hypothesis-Guided Program Evolution
for round $r = 1 \dots K$ **do**
    $\mathcal{P}_{\text{new}} \leftarrow \emptyset$
    for each policy $\pi_{\alpha,\theta^*} \in \mathcal{P}_{\text{elite}}$ **do**
        SR, Failure trace's terminal_frame $\leftarrow$ Execute$(\pi_{\alpha,\theta^*}, \mathcal{E})$
        if SR < 1.0 **then**
            // visual feedback debugging
            $\pi_{\text{refined}} \leftarrow \mathcal{M}$.evolve_policy$(\pi_{\alpha,\theta^*}$, SR, Failure trace's terminal_frame, $\alpha)$
            $\mathcal{P}_{\text{new}} \leftarrow \mathcal{P}_{\text{new}} \cup \{\pi_{\text{refined}}\}$
    Fine tune θ for each policy in $\mathcal{P}_{\text{new}}$ via Optuna TPE and update $\mathcal{P}_{\text{elite}}$ with top t performers
return $\Pi^* \leftarrow \arg\max_{\pi \in \mathcal{P}_{\text{elite}}} SR(\pi)$

C Limitations and Future Work

While LOGIC demonstrates the benefits of decoupling geometric reasoning from parameter tuning, several limitations remain.

Dependence on geometric observations. LOGIC uses object centers or reliable poses obtained from the observation interface. Our RGB-D evaluation estimates centers but discards unreliable object orientations, limiting fine-grained tool alignment (Appendix E). Severe occlusion and clutter remain untested. The controlled pose-noise study in Appendix E.4 shows greater sensitivity on tool-use tasks, where pose errors propagate through the tool geometry.

No real-robot experiments. All results are obtained in simulation (RLBench, Robosuite, and LIBERO). Deploying the synthesized programmatic policies on physical hardware, where contact dynamics, latency, and sensor noise differ from simulation, is left to future work.

Failure modes of hypothesis proposal. The framework relies on the LLM to propose candidate geometric interpretations and task-relevant parameters. If the proposals omit useful interpretations or crucial parameters, the evolution stage must find compensating policy changes or extend the candidate space using execution feedback (Appendix F.5); this recovery is driven by sparse success signals and is not guaranteed.

Rigid-body scope. The geometric hypotheses are currently instantiated through rigid-body transformations of object frames. Extending the formulation to deformable objects, cluttered scenes with multi-object ambiguity, and tasks that require closed-loop force or contact reasoning would require richer hypothesis families and feedback beyond binary task success, and is an important direction for future work.

D EXPERIMENTAL SUPPLEMENT

D.1 RLBENCH

Our experimental framework leverages RLBench to simulate complex robotic interactions for a wide range of tasks. RLBench utilizes the 7-degree-of-freedom Franka Emika Panda arm and it supports directly control of the end-effector pose to achieve more dexterous and complex control. At the beginning of each episode, the positions and orientations of the objects within the environment are randomly reset to ensure the robustness and generalization of the policy programs. We show the initial environment images of all 18 tasks we used for experiments in Figure 3.

The tasks are organized into the following three categories:

- *Single Object Manipulation*: These tasks involve interacting with a single object whose motion is restricted by physical joints (e.g., hinges) or specific geometric contact (e.g., a roll inserted through the stand).
- *Long-Horizon Sequential Manipulation*: These tasks involve multiple objects and multi-step execution, where success depends on consistently interpreting object-centric geometry across the sequence. Errors early in the process can propagate across steps and objects, leading to cascading failures and making zero-shot low-level policies highly brittle. In particular, `BlockPyramid` and `StackCups` require **loops** to enable iterative stacking behaviors.
- *Precise Tool Use & Indirect Manipulation:* This category represents a particularly challenging setting for low-level programmatic control, where the robot must use a held object as an extension of its kinematic chain to interact with a secondary target. In this indirect manipulation regime, the system is highly sensitive to geometric errors: small misalignments in the inferred tool frame or parameter estimates (both unobservable from sparse inputs) can be amplified through the tool's geometry, leading to complete task failure.

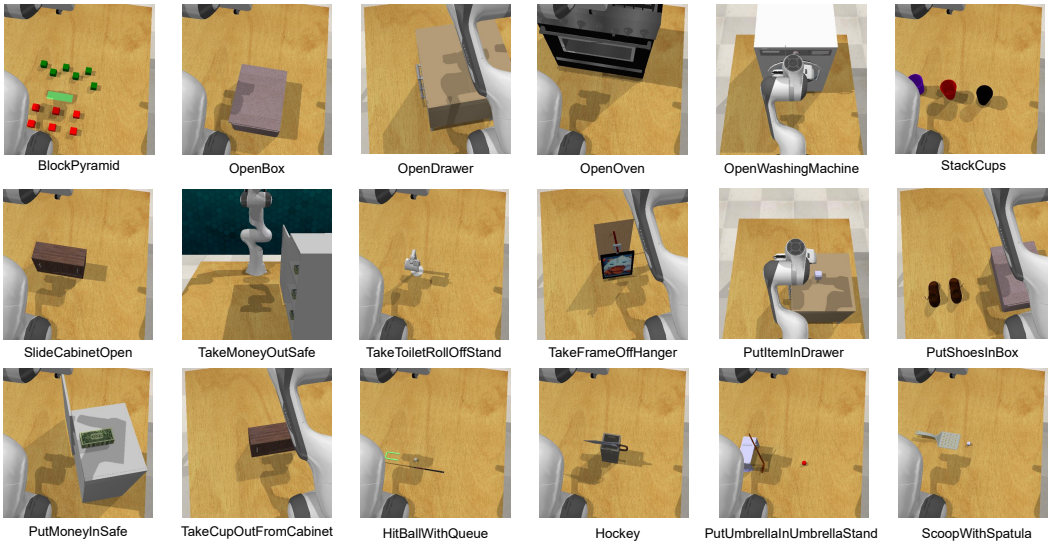
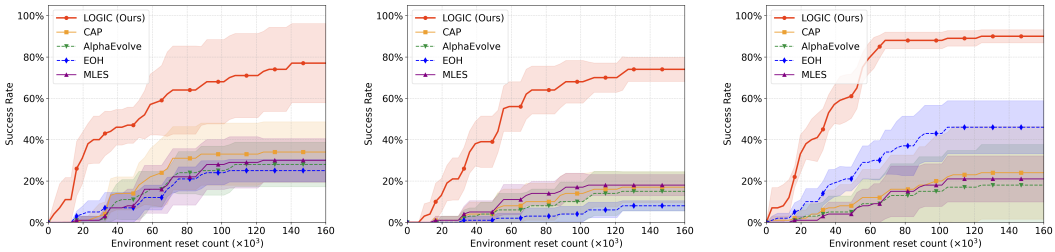
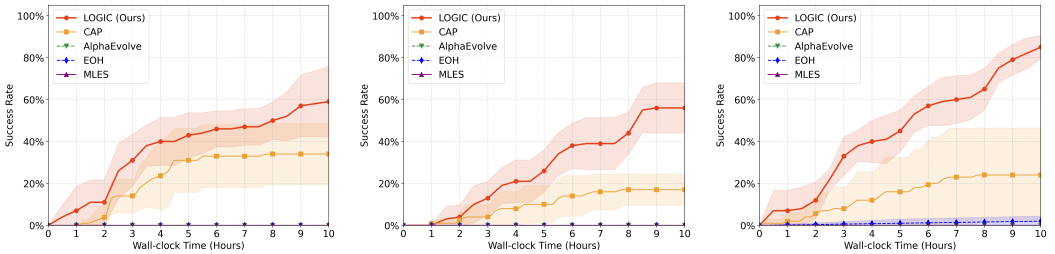


Figure 3: All 18 tasks we used for experiments: **Single Object Manipulation**: `OpenBox`, `OpenDrawer`, `OpenOven`, `OpenWashingMachine`, `SlideCabinetOpen`, `TakeMoneyOutSafe`, `TakeToiletRollOffStand`, `TakeFrameOffHanger`. **Long-Horizon Sequential Manipulation**: `BlockPyramid`, `PutItemInDrawer`, `PutShoesInBox`, `PutMoneyInSafe`, `StackCups`, `TakeCupOutFromCabinet`. **Precise Tool Use & Indirect Manipulation**: `HitBallWithQueue`, `Hockey`, `PutUmbrellaInUmbrellaStand`, `ScoopWithSpatula`.

Table 1 reports mean success rates and standard deviations across seeds for all 18 RLBench tasks.



(a) Hockey (Resets) (b) BlockPyramid (Resets) (c) OpenDrawer (Resets)



(d) Hockey (Time) (e) BlockPyramid (Time) (f) OpenDrawer (Time)

Figure 4: Learning efficiency comparisons across three distinct tasks. The top row (a, b, c) illustrates sample efficiency measured by success rate against environment reset counts ($\times 10^3$). The bottom row (d, e, f) demonstrates computational efficiency measured by success rate over 10 hours of wall-clock time. LOGIC achieves significantly faster convergence and higher final performance across all metrics.

D.2 Sample Efficiency Comparison

Figure 4 illustrates the learning progression with respect to both environment reset counts and wall-clock computation time across three representative tasks: Hockey, BlockPyramid, and OpenDrawer. As shown in the top row (Figure 4.a-c), LOGIC exhibits vastly superior sample efficiency. These results support the use of explicit geometric branches and task-relevant parameterizations to guide search toward successful policies across these tasks.

Furthermore, the bottom row (Figure 4.d-f) highlights the practical wall-clock efficiency within a constrained 10-hour window. The advanced evolutionary baselines (e.g., AlphaEvolve, EOH, MLES) exhibit minimal progress primarily because their evolutionary paradigms necessitate strictly serial optimization. Since each generation's programmatic mutations intrinsically depend on the sequential policy optimization, their wall-clock efficiency is severely bottlenecked, restricting them to very few policies population within the time limit. While CaP can circumvent this serial bottleneck by enabling parallel policy evaluation, its reliance on rigid, hardcoded constants without structural branching quickly traps it in a suboptimal policy. In contrast, LOGIC combines explicit geometric hypothesis branching with parallelized parameter tuning. This allows our framework to simultaneously evaluate policies with different hypothesis branches to achieve near-optimal performance at a fraction of the wall-clock cost.

D.3 Ablation Studies

To quantify the contribution of each core component, we conduct ablation studies by disabling key stages of the pipeline:

- **w/o Geometric Semantic Ambiguity Reasoning:** We remove explicit geometric hypothesis branching in Stage 1 and repeatedly sample policies without assigning them to geometric branches. At each iteration, the LLM performs Physical Parameter Ambiguity Identification via Anticipatory Planning and samples a policy using the resulting parameterization; TPE then optimizes the policy parameters. Parameter identification is performed anew for each sampled policy, allowing both the policy logic and its parameterization to vary across iterations. Sampling proceeds under the same overall search budget as the full method, with the same per-policy TPE settings and

visual-feedback evolution stage. This comparison evaluates the contribution of organizing policy search around explicit geometric hypotheses beyond diversity from repeated planning and policy sampling.

- **w/o Physical Parameter Ambiguity Identification:** We retain the geometric hypothesis branches from Stage 1 and generate policies conditioned on each branch's geometric interpretation. Within each branch, anticipatory planning is retained, but the LLM is not prompted to explicitly identify ambiguous physical quantities or expose them as tunable parameters in Stage 2. The resulting policies contain hardcoded numerical constants, which are extracted via regex after generation and lifted into tunable parameters for TPE optimization. Geometric branching, parameter optimization, and visual-feedback evolution are thus retained; this ablation evaluates the contribution of identifying task-relevant physical parameters during planning.
- **w/o Evolution:** We remove the visual feedback-driven refinement in Stage 5.

Table 4: Ablation study analyzing framework components on average task Success Rate (SR %) over 5 seeds..

Task	*w/o* Geometric Branching	*w/o* Physical Params Identification	*w/o* Evolution	LOGIC(Full)
OpenDrawer	0.54	0.76	0.88	**0.90**
BlockPyramid	0.32	0.52	0.62	**0.74**
Hockey	0.36	0.26	0.62	**0.76**

As shown in Table 4, replacing explicit geometric branching with diverse policy sampling reduces success from 0.90 to 0.54 on `OpenDrawer`, 0.74 to 0.32 on `BlockPyramid`, and 0.76 to 0.36 on `Hockey`. With anticipatory planning, parameter generation, optimization, evolution, and the total search budget held fixed, these results support the benefit of explicitly structuring policy diversity around geometric interpretations.

Removing physical parameter ambiguity identification also degrades performance. This shows that explicitly reasoning about environmental ambiguity by identifying which quantities are unknown (e.g., distances to functional parts or safe approach offsets) and exposing them as tunable parameters yields a well-structured optimization space, whereas post hoc extraction of hardcoded constants leaves the search poorly aligned with the task.

Finally, the *w/o Evolution* ablation shows that visual feedback-driven refinement provides additional gains. Execution feedback exposes concrete failure modes (e.g., misalignment, premature contact, or overshoot), enabling the LLM to make targeted code updates that correct residual errors and improve robustness.

D.4 Impact of different LLM models

We evaluate the scalability of LOGIC across various LLM backbones on three representative tasks. The results in Table 5 demonstrate that LOGIC scales effectively with model capability, as the more powerful GPT-5 and GPT-5.4 consistently achieves the highest success rates across all evaluated tasks. At the same time, the narrow performance gap between the standard and "mini" variants indicates that our structural hypothesis-driven reasoning provides a robust foundation that ensures high performance even with smaller models. This balance confirms that while our framework can leverage the superior reasoning of advanced backbones to reach peak performance, it simultaneously lowers the dependency on raw model scale for synthesizing reliable low-level policies.

E LOGIC under the Perception Setting

E.1 Perception Module and Policy Generation

We evaluate the same synthesis procedure using geometric information obtained from RGB-D images. We adopt the perception module and low-level control interface of CaP-X (Fu et al., 2026), but remove its grasp-pose network so that the generated policy must compute the gripper targets itself.

Table 5: Performance comparison between different LLM models on average task Success Rate (SR %) over 5 seeds.

Model	OpenDrawer	BlockPyramid	Hockey	Average
gpt-5-mini	0.90	0.74	0.76	0.80
gpt-5	0.92	**0.76**	0.80	**0.83**
gpt-5.4-mini	0.92	0.72	0.78	0.81
gpt-5.4	**0.94**	0.74	**0.80**	**0.83**

The policy has access to images, camera calibration, and robot state, but not ground-truth object poses or articulation states. Figure 5 shows how LOGIC uses this interface to open a microwave.

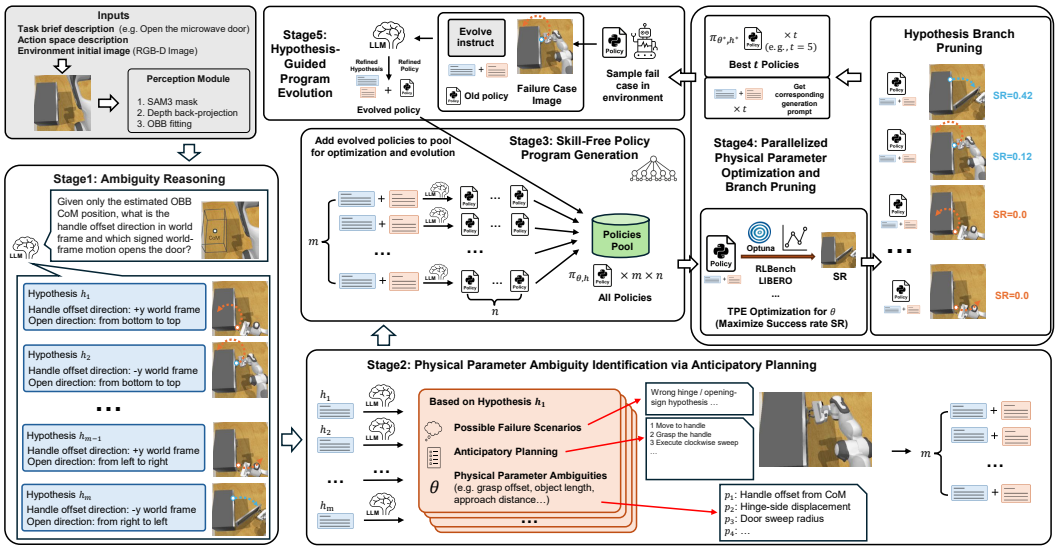


Figure 5: LOGIC under the perception setting. The perception module estimates an object center from RGB-D images. LOGIC proposes handle-offset and opening-motion hypotheses in the world frame, generates policies, and optimizes their parameters. The best t policies then enter evolution, where failure feedback can revise both their hypotheses and code. The revised policies return to the pool for further optimization and selection.

The perception module has three main steps. First, SAM3 takes a text prompt naming the object and produces a mask in the RGB image. Second, the masked depth pixels are back-projected into 3D using the camera intrinsics and transformed into the robot's world frame using the camera extrinsics. Third, the resulting point cloud is filtered and subsampled, and Open3D fits an oriented bounding box (OBB). Its center serves as an estimate of the object's CoM position, although a box fitted to visible surfaces need not coincide with the physical center of mass. The world frame is centered at the robot base, with $+x$ forward, $+y$ to the robot's left, and $+z$ upward.

The OBB also provides an orientation, but its axes can change order or sign when the observed surface changes, especially for nearly symmetric objects. A hypothesis such as "the handle lies along the local $+y$ axis" would then refer to different physical directions across observations. LOGIC therefore uses only the estimated center from the OBB and discards its orientation, expressing hypotheses directly in the world frame. This follows the observation-dependent hypothesis formulation described in Section 4.1.

Consider opening the microwave in Figure 5. Knowing its center does not specify where to grasp the handle or how to move it around the hinge. The image may suggest that the door should be pulled outward, but the program still needs a contact point and a signed opening trajectory in world coordinates. Stage 1 identifies these ambiguities in interaction geometry and proposes hypotheses, such as a handle offset along world $+y$ or $-y$, paired with different opening motions. Each hypothesis

guides Stage 2 in planning the approach, grasp, and door motion, and in identifying parameters such as the handle offset, hinge displacement, and sweep radius. Stage 3 generates policies from these plans, and Stage 4 uses TPE to tune their parameters and rank them by execution success. The search procedure of Figure 2 therefore applies to hypotheses about interaction locations and articulation as well as to interpretations of local axes.

The available geometry also limits the tasks this interface can support. Fine-grained tool alignment in tasks such as Hockey requires accurate object orientation so that a hypothesized local alignment can track the object when computing the gripper target. Our current perception module does not provide that accuracy, and world-frame hypotheses based on an estimated center do not supply the missing orientation information. CaP-Agent0 shares this interface and is subject to the same observation limitation.

E.2 Hypothesis Refinement

An incorrect hypothesis need not fail on every trial. In Figure 5, a policy assumes "handle offset direction: $-y$ world frame" and "open direction: from bottom to top." Although the door does not open by lifting, the gripper can exert an outward pull while attempting this motion and occasionally drag the door open. These incidental successes give the policy a nonzero success rate, which can be enough to place it among the best t candidates retained for evolution. Its inclusion in the pool therefore reflects some useful behavior, such as reaching and engaging the handle, without establishing that the assumed opening motion is correct.

Stage 5 uses the failed executions to reconsider that motion. The LLM receives the policy code, the prompt containing its original hypothesis, its success rate, and the final image of a failed trial. Reading the image together with the code allows it to compare the intended trajectory with the resulting door and gripper configuration, and to consider whether the failure arose from missing the handle or from moving it in the wrong way. In this example, the upward trajectory does not follow the door's rotation about its hinge. The correction is to replace the bottom-to-top assumption with an outward sweep around the hinge, rather than simply increase the lifting distance. The policy must change with the hypothesis: its waypoints should maintain contact with the handle while following the door's arc, with the hinge displacement and sweep radius exposed for optimization.

The revised policy returns to the pool for parameter optimization and evaluation. This process preserves useful parts of the old program while changing the assumption responsible for its failures. If the revised motion was not represented in the original hypotheses, the resulting policy introduces a new branch. Evolution can thus extend the initial search as well as improve policies within an existing branch; subsequent rollouts determine whether the proposed correction succeeds.

E.3 Experiments

We evaluate this setting on 20 tasks: 7 from RLBench, 2 from Robosuite, and 11 from LIBERO. The 13 Robosuite and LIBERO environments are adopted from CaP-X, and the RLBench tasks are ported to the same interface. All three methods receive the same task descriptions and API documentation, use the same perception tools, and are evaluated on 50 held-out episodes per task. These episodes are disjoint from the initializations used for policy optimization and selection.

LOGIC identifies task-relevant information left ambiguous by the observations, proposes hypotheses about it, and generates and optimizes policies under those hypotheses. These ambiguities can concern interaction locations, articulation, spatial relationships, or other assumptions needed to execute the task; directions are one example. CaP generates code through the same interface; we lift its hard-coded numerical constants into parameters and optimize them with the same TPE routine. Both methods use a fixed trial budget, stop optimization early when a policy solves all optimization initializations, and abandon policies that fail persistently. The selected policy is the one with the highest optimization-set success rate. This gives CaP parameter search beyond the single-shot execution reported in CaP-X.

CaP-Agent0 is the reference agent supplied with CaP-X. It generates a code block, executes it, and receives the execution output together with a textual description of the visual change, produced by a vision-language module comparing the observations before and after execution. It then revises the code or declares completion, for at most ten feedback rounds within an episode and without resets between rounds. We use the original trial loop and run the agent zero-shot, without its parallel

ensemble. In this 10-round baseline, CaP-Agent0 does not perform TPE search; its adaptation occurs through these feedback rounds.

Table 6 shows that LOGIC achieves the highest average success rate, with particularly strong gains on tasks such as opening microwaves and drawers. In these tasks, the estimated object center alone does not specify where to make contact or how to move the object. LOGIC explores alternative assumptions about this missing information, generates policies under each assumption, and uses execution feedback to select and refine them. The results suggest that this search helps turn incomplete perceptual observations into effective manipulation policies.

Table 6: Average success rates under the perception pipeline on 20 tasks, measured on 50 held-out episodes over 3 seeds per task. The final row reports the average across tasks.

Benchmark	**Task**	**LOGIC**	**CaP**	**CaP-Agent0**
RLBench	open the microwave	**0.86**	0.00	0.04
RLBench	open the drawer	**0.86**	0.66	0.14
RLBench	slide the cabinet door open	**0.54**	0.44	0.22
RLBench	close the fridge	**0.86**	**0.86**	0.72
RLBench	push the button	**0.98**	0.02	0.88
RLBench	close the drawer	0.84	**1.00**	0.86
RLBench	close the laptop lid	0.56	0.56	**0.72**
Robosuite	stack the cube	**1.00**	0.96	0.90
Robosuite	lift the cube	**0.98**	**0.98**	0.78
LIBERO	turn on the stove	**0.94**	0.58	0.10
LIBERO	open the top drawer of the cabinet	**0.98**	0.02	0.02
LIBERO	turn off the stove	**0.44**	0.24	0.16
LIBERO	put the bowl on top of the cabinet	**1.00**	0.00	0.94
LIBERO	put the bowl on the plate	0.94	**1.00**	0.30
LIBERO	put the tomato sauce in the basket	**1.00**	0.94	0.44
LIBERO	put the orange juice in the basket	**1.00**	0.96	0.52
LIBERO	put the wine bottle on top of the cabinet	**1.00**	0.98	0.20
LIBERO	put the red mug on the plate	**0.92**	0.88	0.36
LIBERO	open the microwave	**0.96**	0.00	0.02
LIBERO	put the frying pan on the stove	**0.54**	0.52	0.48
Average		**0.86**	0.58	0.44

The methods also spend computation differently. Table 7 compares their per-task budgets. With 10 feedback rounds, CaP-Agent0 uses about $19\times$ as many LLM requests and $31\times$ as many tokens as LOGIC, whereas LOGIC uses more environment interaction to optimize candidate policies. Extending feedback increases both the LLM and environment budgets, while adding TPE increases the environment budget without additional LLM calls.

Table 7: Approximate per-task budgets for the perception evaluation. The original LOGIC and 10-round CaP-Agent0 budgets are averaged over 20 tasks with 50 evaluation seeds per task. Environment steps count high-level environment interactions rather than low-level simulator physics steps.

Budget	**LOGIC**	**Agent0 / 10 feedback rounds**	**Agent0 / 100 feedback rounds**	**Agent0 / 10 feedback rounds + TPE**
LLM requests	∼29	∼560	∼5.6K	∼560
LLM tokens (input + output)	∼0.22M	∼6.9M	∼69M	∼6.9M
Environment steps	∼2.7M	∼0.25M	∼2.5M	∼2.7M

The larger interaction budget of LOGIC relative to the 10-round baseline raises the question of whether additional environment steps can close the success-rate gap. We examine two controls on five tasks. Increasing Agent0's feedback limit from 10 to 100 rounds gives it more opportunities to inspect failures and revise its code. Applying TPE to the policy produced after 10 feedback rounds instead searches its numerical parameters without additional LLM calls or LOGIC's hypothesis

search. Table 7 compares the controls at environment-step budgets similar to LOGIC, with a larger LLM budget for the 100-round variant. Table 8 compares their success rates. With 100 feedback rounds, Agent0 improves over its original setting but remains below LOGIC on all five tasks. Adding TPE gives stronger results than extending feedback to 100 rounds, yet LOGIC retains a higher average success rate. Together, the budget comparison and success rates support the benefit of LOGIC's hypothesis search: it achieves the highest average success rate while requiring fewer LLM calls and tokens than either control.

Table 8: Success rates with additional environment interaction for CaP-Agent0 (Agent0). Table 7 provides the corresponding budget comparison. The final row averages the five tasks. Bold marks the best result in each row.

Benchmark	Task	LOGIC	Agent0 / 10 feedback rounds	Agent0 / 100 feedback rounds	Agent0 / 10 feedback rounds + TPE
RLBench	open the microwave	**0.86**	0.04	0.52	0.60
Robosuite	stack the cube	**1.00**	0.90	0.96	1.00
LIBERO	open the top drawer of the cabinet	**0.98**	0.02	0.04	0.24
LIBERO	turn on the stove	**0.94**	0.10	0.48	0.72
LIBERO	put the tomato sauce in the basket	**1.00**	0.44	0.82	0.96
Average		**0.96**	0.30	0.56	0.70

E.4 Robustness to Pose Noise

The RGB-D evaluation above includes errors from segmentation and OBB fitting. To examine sensitivity to observation error separately, we also perturb the simulator-provided poses used in the main setting. This experiment retains an object orientation and does not use the perception module. We add zero-mean Gaussian noise with $\sigma = 1\,\text{cm}$ to positions and Gaussian angular noise with $\sigma = 5^\circ$ to orientations, and independently drop each object from the observation with probability 0.05. Only the policy observations are changed; simulator physics is unchanged. We rerun the full LOGIC pipeline and CaP through the same observation wrapper.

Table 9: Success rate under perception noise (position noise $\sigma = 1\,\text{cm}$, orientation noise $\sigma = 5^\circ$, detection dropout $p = 0.05$), applied only to the poses observed by the policy. Mean $\pm$ std over 5 seeds (50 held-out episodes per seed).

Task	LOGIC (clean)	LOGIC (+noise)	CaP (clean)	CaP (+noise)
OpenDrawer	0.90 ± 0.03	$\mathbf{0.78} \pm 0.08$	0.24 ± 0.21	0.14 ± 0.10
Hockey	0.76 ± 0.19	$\mathbf{0.52} \pm 0.18$	0.28 ± 0.15	0.04 ± 0.02
BlockPyramid	0.74 ± 0.06	$\mathbf{0.60} \pm 0.10$	0.17 ± 0.08	0.10 ± 0.08

As Table 9 shows, LOGIC remains above CaP on all three tasks under noise. Its largest drop is on `Hockey`, from 0.76 to 0.52, where small pose errors affect both the grasp and the projected position of the stick's head. This agrees with the geometric dependence described in Section 2: searching over hypotheses helps resolve missing directional information, but the resulting actions still depend on the accuracy of the available observations.

F Additional Experiments and Discussion

The main experiments compare the success of the final policies. To better understand these results, we examine how branch selection and evolution affect policy search, test transfer to another simulator, and measure the computation required. These studies assess the search procedure as well as the policies it produces.

F.1 HELD-OUT EVALUATION PROTOCOL

The best hypothesis–parameter pair (α^*, θ^*) in Eq. (4) is selected on an *optimization set* of environment initializations: the TPE trials and the final selection across hypotheses only use rollouts drawn from this set (20 rollouts per trial; Appendix H). For the simulator-state RLBench and Robosuite studies, success rates are measured on a *disjoint held-out set* of 50 randomized initializations per seed, over 5 seeds, that is never used for parameter tuning or hypothesis selection. Baselines in these studies use the identical split and the identical tune-and-select routine (their hard-coded constants are lifted into tunable parameters as described in Section 5). Table 1 reports held-out performance under this protocol. The separate perception protocol is described in Appendix E.3.

F.2 CAN VLM-BASED BRANCH SELECTION REPLACE HYPOTHESIS BRANCHING?

LOGIC keeps every geometric hypothesis as a separate branch, executes all of them, and prunes the branches whose policies fail (Section 4.4). A natural alternative is to skip the branching and let a strong VLM inspect the candidate policies and commit to the single interpretation that looks right. We test this alternative directly, giving the VLM every advantage we could provide. For each task we generate one policy per geometric hypothesis, exactly as in Stage 3 of LOGIC. The VLM (GPT-5.4) is then shown, for every hypothesis, the policy code, its measured success rate, and renderings from three cameras captured at the end of each policy sub-stage, and it selects the best hypothesis. The selected hypothesis then receives the same optimize-and-evolve budget as LOGIC: over five rounds, six candidate policies are generated and TPE-optimized, the best one is kept, and the VLM inspects that policy's success rate and three-camera renderings to refine the hypothesis, policy structure, and parameters that seed the next round. The only difference from LOGIC is therefore that a single interpretation chosen by the VLM is pursued, instead of branching over all hypotheses and pruning them by their execution results.

Table 10: Final success rate when a VLM (GPT-5.4) selects a single geometric interpretation (select-then-optimize) versus hypothesis branching with execution-based pruning (LOGIC). Both pipelines use the same optimization and evolution budget. Mean $\pm$ std over 5 seeds.

Task	**VLM-judge (select-then-optimize)**	**LOGIC**
OpenDrawer	0.36 ± 0.11	$\mathbf{0.90} \pm 0.03$
Hockey	0.18 ± 0.09	$\mathbf{0.76} \pm 0.19$
BlockPyramid	0.10 ± 0.06	$\mathbf{0.74} \pm 0.06$

Table 10 shows a large gap in task success. Across tasks, the VLM selects the correct interpretation in roughly one third of the cases and is frequently *confidently wrong*: its stated confidence is essentially uncorrelated with correctness. Many candidates appear plausible in the renderings while differing in object-centric axis semantics, grasp offsets, approach distances, or collision clearances. Under this protocol, exploring multiple hypothesis-conditioned branches yields more successful policies than committing to the VLM-selected branch. The task-success comparison evaluates the utility of these search strategies; it does not establish that the highest-scoring policy recovers the true geometric interpretation.

F.3 GENERALIZATION TO A SECOND SIMULATOR: ROBOSUITE

We additionally evaluate LOGIC on Robosuite (Zhu et al., 2020), a MuJoCo-based (Todorov et al., 2012) manipulation benchmark that provides a different set of tasks from RLBench. Low-level control uses Robosuite's continuous Operational Space Controller (OSC): the policy outputs 6-DoF end-effector targets and gripper commands, and no skill primitives are available. The full LOGIC pipeline (hypothesis generation, parameterized policy generation, TPE optimization, and evolution) runs end-to-end. We compare against CaP, AlphaEvolve, EOH, and MLES under the identical interface, using 5 seeds per task (Table 11). The three tasks involve the following geometric ambiguities and control parameters:

- `Door`: Only the positions of the door and handle are observed, without orientation. The hinge side, and hence the arc along which the handle must be pulled, is latent; pulling in

the wrong direction fails to open the door. LOGIC generates policies under alternative hinge-side and opening-direction hypotheses and selects them by execution success.

- `Lift`: The goal is to grasp a cube and lift it above a height threshold. A top-down grasp usually suffices, but the cube pose does not specify which local axis the gripper yaw should align with so that the fingers meet the cube faces, nor the hover and contact offsets. LOGIC explores candidate alignments and optimizes the offsets through execution.
- `NutAssemblyRound`: Only the nut's coordinate frame is observed. Which local axis points toward the handle is latent; a top-down grasp at the nut center closes on the empty through-hole. The gripper must instead grasp the handle at an offset along the handle axis and then seat the hole over the peg.

Table 11: Success rate on Robosuite (MuJoCo, OSC low-level control), mean ± std over 5 seeds. Bold marks the highest mean success rate in each row.

Task	LOGIC	CaP	AlphaEvolve	EOH	MLES
Door	$\mathbf{0.95} \pm 0.04$	0.20 ± 0.14	0.62 ± 0.15	0.82 ± 0.09	0.78 ± 0.18
Lift	$\mathbf{0.95} \pm 0.04$	0.90 ± 0.08	0.88 ± 0.14	0.90 ± 0.12	0.66 ± 0.09
NutAssemblyRound	$\mathbf{0.70} \pm 0.12$	0.15 ± 0.10	0.22 ± 0.17	0.38 ± 0.22	0.26 ± 0.24

Table 11 shows that LOGIC achieves the highest mean success rate on all three tasks. Its advantage over the strongest baseline is larger on `Door` and `NutAssemblyRound`, where choosing the hinge motion or handle axis is critical, than on `Lift`, where a default top-down grasp usually succeeds. These results support the benefit of resolving geometric ambiguity through execution in a second simulator.

F.4 Per-Task Compute Budget

Table 12 consolidates the per-task budget of LOGIC, derived from the hyperparameters in Table 14 and the runtimes in Table 13 and averaged over tasks and seeds. Stages 1–2 (ambiguity identification, hypothesis branching, and anticipatory planning) require only a handful of LLM calls and no rollouts. Stages 3–4 generate $n = 6$ policies per hypothesis (36 programs for a typical task), each optimized with 100 TPE trials of 20 rollouts. Stage 5 runs $K = 5$ evolution rounds, each refining the top $t = 5$ policies into $l = 2$ variants, i.e., 50 additional programs optimized with the same per-program budget.

Table 12: Average per-task budget of LOGIC (LLM calls, generated programs, TPE trials, environment rollouts, and wall-clock time), broken down by stage.

Stage (per task)	LLM calls	Programs	Trials	Rollouts	Wall-clock
Hypothesis enumeration + anticipatory planning	∼7	—	—	—	—
Initial generation + parallel TPE	∼36	36	3,600	72,000	∼10 h
Hypothesis-guided evolution (K=5, t=5, l=2)	∼50	50	5,000	100,000	∼48 h
Total	**∼93**	**86**	**8,600**	**172,000**	**∼58 h**

The main comparison in Table 1 is budget-matched: every baseline (CaP, AlphaEvolve, EOH, and MLES) is given the same total number of generated programs and environment rollouts as the full LOGIC pipeline. Under this equal budget, LOGIC without the evolution stage, which consumes less than half of the program and rollout budget (36 of 86 programs and 72,000 of 172,000 rollouts), still outperforms every fully evolved baseline by a large margin. These results support the effectiveness of geometric hypothesis branching and parameterized policy generation under a matched search budget. Wall-clock times for all methods, measured under the same program budget, are reported in Appendix G.

F.5 Hypothesis-Space Coverage and the Role of Evolution

The hypothesis space is proposed and refined, not hand-specified. The hypotheses are not written by hand. Given the task instruction and the sparse observation interface, the LLM identifies the

directional ambiguities of the task and proposes candidate object-centric interpretations (Section 4), so the size of $\mathcal{H}_{\text{sem}}$ adapts to the task. During evolution, execution feedback can motivate changes to geometric assumptions, control logic, and parameterization (for example, adding lateral- and back-offset parameters on `Hockey`). Such revisions can improve policies whose initial assumptions were unhelpful, but recovery from missing hypotheses is not guaranteed. The same prompt templates, pipeline, and hyperparameters are used unchanged across all 18 RLBench tasks; each task's ambiguities, hypotheses, parameters, and parameter ranges are generated automatically.

Coverage and excess hypotheses. We manually inspected the initial hypothesis space proposed for every evaluated task and found that it contained the correct interpretation in all of them. This observation concerns proposal coverage; it does not establish that the selected policy uses that interpretation or recovers physical parameter values. The execution-based filtering of Eq. (4) ranks candidate policies by task success. Additional hypotheses increase exploration but also consume search resources, so their value depends on the search budget and the quality of the resulting programs.

Evolution is complementary, not the source of the gain. The component ablation in Table 4 isolates what drives the improvement: removing semantic hypothesis branching drops success from 0.90 to 0.54 (`OpenDrawer`), 0.74 to 0.32 (`BlockPyramid`), and 0.76 to 0.36 (`Hockey`), and removing physical-parameter identification drops it to 0.76, 0.52, and 0.26, respectively. These ablations support the contribution of both explicit geometric branching and planning-informed parameterization to policy search. The evolution stage plays a complementary, secondary role: removing it lowers the average success rate in Table 1 from 0.71 to 0.65, which still leaves LOGIC far above every fully evolved baseline (whose averages range from 0.27 to 0.34). Its benefit is largest on the harder long-horizon and tool-use tasks (`BlockPyramid` $0.62 \rightarrow 0.74$, `Hockey` $0.62 \rightarrow 0.76$) and negligible where the task is already nearly solved (`OpenDrawer` $0.88 \rightarrow 0.90$). These results indicate that evolutionary refinement complements the candidates produced by geometry-conditioned search.

G RUNTIME

All experiments and policy generation pipelines were executed on a $8\times$ A100 server equipped with dual AMD EPYC 7513 32-Core Processors (128 logical threads, up to 3.68 GHz).

Table 13 details the computational time required for our LOGIC framework compared to the baselines. For our method, we decompose the total runtime into the time consumed by the initial policy generation alongside the Parallelized Black-Box Parameter Optimization phase, and the time consumed by the subsequent Hypothesis-Guided Program Evolution phase. For all evolve baselines (AlphaEvolve, EOH, MLES), we utilize three parallel evaluators. To ensure a fair comparison, the runtime for all baseline methods is measured under the exact same total policy sample budget as our full framework (including the evolution steps).

Table 13: Average runtime comparison across different task categories. For LOGIC, the total time is broken down into the Policy Generation + Parameter Optimization phase, and the Evolution phase. All baseline times are evaluated under the same total policy sample budget. All values are in hours.

Task Category	**LOGIC (Ours)**			**Baselines Total Time**			
	Gen. & Opt. Phase	**Evol. Phase**	**Total Time**	**CaP**	**EOH**	**AlphaEvolve**	**MLES**
Single Fixed-Object	8	40	48	8	229	189	190
Long-Horizon Placement	12	60	72	12	344	306	316
Precise Manipulation	10	45	54	10	257	220	232
Average Overall	**10**	**48.33**	**58**	**10**	**276.67**	**238.34**	**246**

H HYPERPARAMETERS

The hyperparameters utilized within our framework are consistent across all 18 RLBench tasks to ensure standardized evaluation. Table 14 lists the specific settings used for the LLM API, the Policy Generation, the Parallelized Physical Parameter Optimization (via Optuna), and the Hypothesis-Guided Program Evolution loop.

We utilized the `gpt-5-mini` model for all generation and reasoning steps, maintaining the default generation temperature to encourage creative hypothesis branching while relying on downstream optimization to filter out invalid logic.

Table 14: Hyperparameter configurations for LOGIC.

Hyperparameter	**Value**
LLM Settings	
Model	`gpt-5-mini`
Temperature	1.0 (API Default)
Generation & Optimization (Optuna TPE)	
Candidate policies generated per discrete hypothesis n	6
Optuna TPE search trials	100
Evaluation rollouts per trial (Optimization Metric)	20
Hypothesis-Guided Program Evolution	
Total evolution rounds K	5
Top policies selected per round t	5
Refined policies generated from per selected policy l	2

I PROMPTS

LOGIC uses five interconnected prompts to identify task-relevant geometric ambiguities, construct hypothesis branches, plan execution and identify parameters, generate policies, and guide refinement from execution feedback. The task description and observation interface guide which relationships require hypotheses and how they are represented. The templates below illustrate local-frame reasoning when 6D object poses are available; Appendix E.1 illustrates world-frame hypotheses when OBB orientation is unreliable. These are applications of the same synthesis procedure: hypotheses and parameterizations structure the program search space, and execution feedback guides its refinement.

I.1 IDENTIFY IMPORTANT GEOMETRIC AMBIGUITIES

Prompt - Identify Important Geometric Ambiguities

Role: You are a robotic manipulation expert.

Given:

1. Robotic task brief description

2. Observation and action space information of this task

Your task:

Since all quaternions in this environment are given in (qx, qy, qz, qw) format representing the rotation of the object's local frame relative to the world frame, and all provided poses are expressed in the world frame, how to get the correct local frame is essential when computing the correct gripper orientation or calculating points along the object's specific direction in its local coordinate frame.

First, you need to identify which specific directional information within an object's local frame is most critical for successfully completing the task. For instance, if the objective is to slide the front left door of a cabinet, it is absolutely vital to determine which axis within the cabinet's local frame corresponds to the front direction to help calculating the approach

direction and quaternion of the gripper.

- Identify the most critical directional property (or properties) like "the object_A front direction".
- The proposed directional property should be important for calculating the target end-effector pose or relative position from the object center.
- Extract ONLY the core descriptive phrase of the direction. Do not assign it to any specific axis (like x, y, z), as we will generate all possible axis candidates later.

Example Output:

["the cabinet front direction to approach", "the cabinet sliding right direction"]

I.2 Hypothesis Branching

Prompt - Hypothesis Branching

Role: You are a robotic manipulation expert.

Given:

1. Robotic task brief description

2. Observation and action space information of this task

3. Geometric ambiguities

Your task:

Please first carefully analyze the task description, observation and action space information. The geometric ambiguities is the most important information needed to finish the task. Please generate candidate interpretations that cover all possible combinations of these geometric ambiguities.

- For each geometric ambiguity, infer the set of possible candidate assignments implied by the text.
- Compute the Cartesian product of all candidate assignments.
- For each combination, write a clear paragraph describing that full interpretation.
- Each paragraph must explicitly resolve all geometric ambiguities.
- Return the results as a Python list of strings.

Example Output:

["Local frame axis +x represents the cabinet front direction, and local frame axis +y represents the cabinet right sliding direction", "Local frame axis +x represents the cabinet front direction, and local frame axis -y represents the cabinet right sliding direction", ...]

I.3 Anticipatory Planning and Identify Parameter Ambiguities

Prompt - Anticipatory Planning and Identify Parameter Ambiguities

Role: You are a robotic manipulation expert.

Given:

1. Robotic task brief description

2. Observation and action space information of this task

3. Candidate interpretation of geometric ambiguities

Your task:

Please first carefully analyze task description, observation and action space information and the candidate interpretation of geometric ambiguities, thinking about scenarios that could lead to task failure. Make the robotic task description details more concise to assure task success. Then, generate the following two items to help policy function generation: 1. an improved task description to avoid potential failures and help defining the policy parameters. 2. a physically executable plan that could successfully complete the task.

1. The "improved_task_description" must be a dictionary containing two key-value pairs:

- "task_description": A lightly modified version of the original task_description. keep the overall structure and most wording of the original description. But add the candidate interpretation of geometric ambiguities to the task description. Besides, please include practical guidance on how to perform the task successfully and how to avoid task failure based on the analysis of the new task description which is enhanced with the interpretation of geometric ambiguities.

- "policy_parameters_instructions": A description of parameters representing ambiguous environmental quantities needed for task execution and policy generation. These parameters should describe important continuous properties in the environment for the task execution (e.g., object length, distance between key points, relative offsets, object height, container opening width, etc.). DO NOT include discrete variables or categorical states. Do not include meaningless parameters that are not critical to the success of the task. The value should be a short natural language description explaining what continuous parameters should be needed to successfully execute the task.

2. The plan should include several substeps to finish the task. Return the plan as a python list and each element of the python list must be a dictionary with exactly one key-value pair:

- Key: substep name
- Value: one-sentence description of the guidance for execution
Requirements for the plan substeps:

- Every substep must be physically executable and only use the action space to be executed.
- Avoid meaningless substeps. Only include necessary steps to complete the task.
- Do not include "Begin of task", "End of task", or observation/verification-only substeps.
- Each substep should clearly describe the guidance for execution.

Example Output:

improved_task_description = {"task_description": "In this task, ...",
"policy_parameters_instructions": "The length between partA of Object1 and partB of Object1 should be added as a continuous parameter for policy generation because ..."
}

plan=[{"Rotate gripper": "The target quaternion should calculated by ... to ensure the gripper approaches ObjectA from the correct direction."},
{"Move gripper to ObjectA": "The target position and orientation should be set to position the gripper in front of ObjectA with a certain distance offset and the gripper is toward ObjectA."},
{"Grasp ObjectA": "Close the gripper to grasp ObjectA"}, ...]

I.4 POLICY GENERATION

Prompt - Policy Generation

Role: You are a robotic manipulation expert.

Given:

1. Improved robotic task description and policy_parameters_instructions

2. Observation and action space information of this task

3. Anticipatory plan guide for task execution

Your task:

Please first carefully analyze the task description and observation and action space information. You need to design a Policy function to complete this task. Please return the contents of a Python file. I will extract and save it as an actual Python file. In your output, you must use the markers “python file begin:” and “python file end” to delimit the Python file content.

The Python file must include:
1.All required imports.
2.A Python function called Policy that implements the task.
3.A dictionary called policy_info which contains: Default parameter values (all floats) and recommended parameter ranges.

Note:

- The input of the policy function should be the env, and all parameters you defined.
- You must complete the whole task within 50 steps(env.step(action)), you must ensure that all stages in your function are completed within the finite number of steps. So please limit the maximum number of steps for each stage.
- You can should use "_, _, done, _ = env.step(action)" for each step. You must use "positions = env.get_object_positions()" to access the current pose or position information of objects.
- The output of the policy function should be ’done’. If any ’done’ is true at any intermediate step, task execution terminates immediately.
- You should carefully read the policy_parameters_instructions to define the parameters of the policy function.

Example output:

python file begin:

```
import numpy as np

def Policy(env, a=0.08, b=0.02, . . . ):

    positions = env.get_object_positions()

   ==========Your Execution==========
   ...

   return

policy_info = {"parameters":{’a’: 0.08, ’b’: 0.02, ...},
```

"parameters_range":{'a': (0, 0.1), 'b': (-0.1, 0.1), ...} }

python file end

I.5 Evolution Guidance

Prompt - Evolution Guidance

Role: You are a robotic manipulation expert.

Given:

1. Old policy function code

2. Prompt used for generating old policy function

3. Evaluation success rate

Your task:

You are given a policy function for a specific robotic task. There is a failed case in the evaluation, and the final frame image is attached. The attached image shows the final state of the environment after the policy execution.

- Carefully analyze the old policy code and the execution result (especially the final frame image).
- Identify why the policy failed and how it can be improved(by modifying the policy logic or parameters representing environmental ambiguities).
- Design an improved policy function to complete the task, strictly following the original requirements and format in the prompt used for generating old policy function.

I.6 Example Policy Function

Figure 6 illustrates the policy structure generated by LOGIC for the Hockey task. Helper functions such as `quat_rotate` and `build_orientation` are also synthesized by the LLM and are omitted from the simplified illustration. Listing 1 provides the full example policy code.

```python
import numpy as np

def quat_rotate():
    ...

def build_orientation():
    ...

def Policy(env, head_offset, grasp_offset, lift_clearance, push_distance, ...):
    # 1. Extract sparse state observations
    positions = env.get_object_positions()
    stick_center = positions['hockey_stick_pose'][:3]
    stick_quat = positions['hockey_stick_pose'][3:7]
    # 2. Instantiate Semantic Hypothesis: Local +z points to the stick head
    local_plus_z = np.array([0.0, 0.0, 1.0])
    stick_head_dir = quat_rotate(stick_quat, local_plus_z)
    handle_dir = -stick_head_dir
    # 3. Compute precise low-level target pose using physical parameters
    grasp_center = stick_center + (grasp_offset * handle_dir)
    target_quat = build_orientation(handle_dir)
    # 4. Execute primitive-free 6-DoF control: [x, y, z, qx, qy, qz, qw, grip]
    action = np.concatenate([grasp_center, target_quat, [1.0]]) # 1.0 = open gripper
    _, _, done, _ = env.step(action)
    # ... [Subsequent execution steps: Lift, Position behind ball, Strike ball] ...
    return done
# 5. Policy metadata extracted for parallelized Optuna parameter optimization
policy_info = {
    'parameters': {'head_offset': 0.064, 'grasp_offset': 0.071, ...},
    'parameters_range': {'head_offset': (0.05, 0.3), 'grasp_offset': (0.05, 0.4), ...}
}
```

Figure 6: A simplified illustration of the policy structure generated by LOGIC for the Hockey task. The policy explicitly defines the semantic geometric hypothesis and utilizes tunable ambiguous parameters for precise 6-DoF control.

```python
import numpy as np
import math

# ----------------------------
# Quaternion / geometry utils
# ----------------------------
def normalize(v, eps=1e-8):
    v = np.array(v, dtype=float)
    n = np.linalg.norm(v)
    if n < eps:
        return v * 0.0
    return v / n

def quat_mul(q1, q2):
    x1, y1, z1, w1 = q1
    x2, y2, z2, w2 = q2
    x = w1*x2 + x1*w2 + y1*z2 - z1*y2
    y = w1*y2 - x1*z2 + y1*w2 + z1*x2
    z = w1*z2 + x1*y2 - y1*x2 + z1*w2
    w = w1*w2 - x1*x2 - y1*y2 - z1*z2
    return np.array([x, y, z, w], dtype=float)

def quat_conjugate(q):
    q = np.array(q, dtype=float)
    return np.array([-q[0], -q[1], -q[2], q[3]], dtype=float)

def quat_rotate(q, v):
    # rotate vector v (3,) by quaternion q (qx,qy,qz,qw)
    qv = np.array([v[0], v[1], v[2], 0.0], dtype=float)
    return quat_mul(quat_mul(q, qv), quat_conjugate(q))[:3]

def matrix_to_quat(m):
    # m is 3x3 rotation matrix, convert to quaternion (qx,qy,qz,qw)
    trace = m[0,0] + m[1,1] + m[2,2]
```

```
    if trace > 0.0:
        s = 0.5 / math.sqrt(trace + 1.0)
        w = 0.25 / s
        x = (m[2,1] - m[1,2]) * s
        y = (m[0,2] - m[2,0]) * s
        z = (m[1,0] - m[0,1]) * s
    else:
        if (m[0,0] > m[1,1]) and (m[0,0] > m[2,2]):
            s = 2.0 * math.sqrt(1.0 + m[0,0] - m[1,1] - m[2,2])
            w = (m[2,1] - m[1,2]) / s
            x = 0.25 * s
            y = (m[0,1] + m[1,0]) / s
            z = (m[0,2] + m[2,0]) / s
        elif m[1,1] > m[2,2]:
            s = 2.0 * math.sqrt(1.0 + m[1,1] - m[0,0] - m[2,2])
            w = (m[0,2] - m[2,0]) / s
            x = (m[0,1] + m[1,0]) / s
            y = 0.25 * s
            z = (m[1,2] + m[2,1]) / s
        else:
            s = 2.0 * math.sqrt(1.0 + m[2,2] - m[0,0] - m[1,1])
            w = (m[1,0] - m[0,1]) / s
            x = (m[0,2] + m[2,0]) / s
            y = (m[1,2] + m[2,1]) / s
            z = 0.25 * s
    q = np.array([x, y, z, w], dtype=float)
    q /= (np.linalg.norm(q) + 1e-12)
    return q

def build_quat_from_axes(x_axis, y_axis, z_axis):
    # Columns are x,y,z axes of the target frame expressed in world
    frame.
    x = normalize(x_axis)
    z = normalize(z_axis)
    # Make x orthogonal to z
    x = normalize(x - np.dot(x, z) * z)
    y = normalize(np.cross(z, x))
    R = np.zeros((3,3), dtype=float)
    R[:,0] = x
    R[:,1] = y
    R[:,2] = z
    return matrix_to_quat(R)

# ----------------------------
# Policy
# ----------------------------
def Policy(env,
           stick_head_offset=0.25,
           stick_shaft_diameter=0.03,
           ball_radius=0.03,
           pregrasp_height=0.18,
           grasp_offset_along_handle=0.06,
           lateral_offset=0.02,
           lift_height=0.12,
           head_clearance=0.02,
           back_offset=0.03,
           push_distance=0.6):
    """
    Robust policy to pick the hockey stick by the handle (away from
    head), orient the stick
    so the head points horizontally toward the ball, and push the
    ball into the goal.
    """
    # sanitize parameters
    stick_head_offset = float(max(0.01, stick_head_offset))
```

```
    stick_shaft_diameter = float(max(0.005, stick_shaft_diameter))
    ball_radius = float(max(0.005, ball_radius))
    pregrasp_height = float(max(0.02, pregrasp_height))
    grasp_offset_along_handle = float(max(0.0,
grasp_offset_along_handle))
    lateral_offset = float(max(0.0, lateral_offset))
    lift_height = float(max(0.01, lift_height))
    head_clearance = float(max(0.001, head_clearance))
    back_offset = float(max(0.0, back_offset))
    push_distance = float(max(0.02, push_distance))

    def do_step(action):
        _, _, done, _ = env.step(action)
        return done

    # Read initial positions
    positions = env.get_object_positions()
    stick_pose = np.array(positions['hockey_stick_pose'], dtype=float
)
    stick_pos = stick_pose[0:3].copy()
    stick_q = stick_pose[3:7].copy()
    ball_pos = np.array(positions['hockey_ball_pos'], dtype=float)
    goal_pos = np.array(positions['hockey_goal_pos'], dtype=float)

    # compute stick long axis in world (local +z rotated by stick
quaternion)
    stick_dir = normalize(quat_rotate(stick_q, np.array([0.0, 0.0,
1.0])))

    # compute head world position and handle end position
    head_world = stick_pos + stick_dir * stick_head_offset
    handle_end = stick_pos - stick_dir * stick_head_offset  #
approximate other end

    # choose grasp point a little in from handle end toward center
    grasp_point = handle_end + stick_dir * grasp_offset_along_handle

    # Build initial gripper orientation: gripper +z points down (
approach),
    # set gripper +x to be horizontal and roughly along stick's
horizontal projection,
    # so jaws close around shaft (perpendicular to stick axis).
    desired_z = np.array([0.0, 0.0, -1.0])
    stick_horiz = stick_dir.copy()
    stick_horiz[2] = 0.0
    if np.linalg.norm(stick_horiz) < 1e-3:
        desired_x = np.array([1.0, 0.0, 0.0])
    else:
        desired_x = normalize(stick_horiz)
    desired_y = normalize(np.cross(desired_z, desired_x))
    grip_quat = build_quat_from_axes(desired_x, desired_y, desired_z)

    # compute gripper axes
    grip_x = quat_rotate(grip_quat, np.array([1.0, 0.0, 0.0]))
    grip_y = quat_rotate(grip_quat, np.array([0.0, 1.0, 0.0]))
    grip_z = quat_rotate(grip_quat, np.array([0.0, 0.0, 1.0]))

    # choose lateral offset direction that moves AWAY from the head
to avoid collisions
    head_vec = head_world - grasp_point
    # project head_vec onto grip_y to determine sign
    sign = 1.0 if np.dot(head_vec, grip_y) >= 0.0 else -1.0
    lateral_dir = -sign * grip_y  # move opposite direction of head
projection

```

```
    # Pre-grasp pose: above the grasp point, slightly back along negative gripper x to avoid head collision
    pregrasp_pos = grasp_point - grip_z * pregrasp_height + lateral_dir * lateral_offset - grip_x * back_offset
    action = np.concatenate([pregrasp_pos, grip_quat, np.array([1.0])])
    done = do_step(action)
    if done: return done

    # Approach: move closer along gripper +z while keeping lateral/back offset
    shaft_radius = stick_shaft_diameter / 2.0
    approach_clearance = max(0.004, shaft_radius * 0.6)
    approach_pos = grasp_point - grip_z * approach_clearance + lateral_dir * lateral_offset - grip_x * back_offset
    approach_pos[2] = max(approach_pos[2], grasp_point[2] + 0.002)
    action = np.concatenate([approach_pos, grip_quat, np.array([1.0])])
    done = do_step(action)
    if done: return done

    # Fine descend to grasp position (align z to shaft center)
    grasp_pos = grasp_point + lateral_dir * lateral_offset - grip_x * back_offset
    grasp_pos[2] = max(grasp_point[2], approach_pos[2] - 0.03)
    action = np.concatenate([grasp_pos, grip_quat, np.array([1.0])])
    done = do_step(action)
    if done: return done

    # Close gripper to grasp shaft
    action = np.concatenate([grasp_pos, grip_quat, np.array([-1.0])])
    done = do_step(action)
    if done: return done

    # Lift to clear the table
    gripper_pose = env.obs.gripper_pose
    gripper_pos = np.array(gripper_pose[0:3], dtype=float)
    lift_pos = gripper_pos + np.array([0.0, 0.0, lift_height])
    action = np.concatenate([lift_pos, grip_quat, np.array([-1.0])])
    done = do_step(action)
    if done: return done

    # Refresh object poses after grasp/lift
    positions = env.get_object_positions()
    stick_pose = np.array(positions['hockey_stick_pose'], dtype=float)
    stick_pos = stick_pose[0:3].copy()
    stick_q = stick_pose[3:7].copy()
    stick_dir = normalize(quat_rotate(stick_q, np.array([0.0, 0.0, 1.0])))

    # Determine which gripper axis corresponds to the stick's local +z
    gripper_pose = env.obs.gripper_pose
    cur_grip_quat = np.array(gripper_pose[3:7], dtype=float)
    g_x = quat_rotate(cur_grip_quat, np.array([1.0, 0.0, 0.0]))
    g_y = quat_rotate(cur_grip_quat, np.array([0.0, 1.0, 0.0]))
    g_z = quat_rotate(cur_grip_quat, np.array([0.0, 0.0, 1.0]))
    dots = [abs(np.dot(stick_dir, g_x)), abs(np.dot(stick_dir, g_y)), abs(np.dot(stick_dir, g_z))]
    axis_idx = int(np.argmax(dots))  # 0->x,1->y,2->z
    axis_signs = [np.dot(stick_dir, g_x), np.dot(stick_dir, g_y), np.dot(stick_dir, g_z)]
    axis_sign = np.sign(axis_signs[axis_idx]) if abs(axis_signs[axis_idx]) > 1e-6 else 1.0
```

```
    # Compute push direction (ball -> goal) projected to horizontal
    positions = env.get_object_positions()
    ball_pos = np.array(positions['hockey_ball_pos'], dtype=float)
    goal_pos = np.array(positions['hockey_goal_pos'], dtype=float)
    push_dir = goal_pos - ball_pos
    push_dir[2] = 0.0
    if np.linalg.norm(push_dir) < 1e-6:
        tmp = ball_pos - stick_pos
        tmp[2] = 0.0
        if np.linalg.norm(tmp) < 1e-6:
            push_dir = np.array([1.0, 0.0, 0.0])
        else:
            push_dir = normalize(tmp)
    else:
        push_dir = normalize(push_dir)

    desired_z2 = np.array([0.0, 0.0, -1.0])
    if axis_idx == 0:
        desired_x2 = axis_sign * push_dir
        desired_y2 = normalize(np.cross(desired_z2, desired_x2))
        grip_quat2 = build_quat_from_axes(desired_x2, desired_y2,
    desired_z2)
    elif axis_idx == 1:
        desired_y2 = axis_sign * push_dir
        desired_x2 = normalize(np.cross(desired_y2, desired_z2))
        grip_quat2 = build_quat_from_axes(desired_x2, desired_y2,
    desired_z2)
    else:
        desired_x2 = push_dir
        desired_y2 = normalize(np.cross(desired_z2, desired_x2))
        grip_quat2 = build_quat_from_axes(desired_x2, desired_y2,
    desired_z2)

    # Rotate gripper to new orientation in place
    gripper_pose = env.obs.gripper_pose
    gripper_pos = np.array(gripper_pose[0:3], dtype=float)
    action = np.concatenate([gripper_pos, grip_quat2, np.array
    ([-1.0])])
    done = do_step(action)
    if done: return done

    # Small settle step
    positions = env.get_object_positions()
    stick_pose = np.array(positions['hockey_stick_pose'], dtype=float
    )
    stick_pos = stick_pose[0:3].copy()
    stick_q = stick_pose[3:7].copy()
    stick_dir = normalize(quat_rotate(stick_q, np.array([0.0, 0.0,
    1.0])))

    grip_x2 = quat_rotate(grip_quat2, np.array([1.0, 0.0, 0.0]))
    grip_y2 = quat_rotate(grip_quat2, np.array([0.0, 1.0, 0.0]))
    grip_z2 = quat_rotate(grip_quat2, np.array([0.0, 0.0, 1.0]))

    head_world = stick_pos + stick_dir * stick_head_offset
    gripper_pose = env.obs.gripper_pose
    gripper_pos = np.array(gripper_pose[0:3], dtype=float)
    head_vec = head_world - gripper_pos

    if axis_idx == 0: grip_axis_vec = grip_x2
    elif axis_idx == 1: grip_axis_vec = grip_y2
    else: grip_axis_vec = grip_x2

    measured_head_offset = float(np.dot(head_vec, grip_axis_vec))
```

```
    measured_head_offset = float(max(0.01, min(abs(measured_head_offset), 1.0)))

    # Position gripper so stick head is just behind the ball
    positions = env.get_object_positions()
    ball_pos = np.array(positions['hockey_ball_pos'], dtype=float)
    goal_pos = np.array(positions['hockey_goal_pos'], dtype=float)
    push_dir = goal_pos - ball_pos
    push_dir[2] = 0.0
    if np.linalg.norm(push_dir) < 1e-6:
        push_dir = normalize(np.array([1.0, 0.0, 0.0]))
    else:
        push_dir = normalize(push_dir)

    head_target_pos = ball_pos - push_dir * (ball_radius + 0.01)
    desired_head_z = ball_pos[2] + ball_radius + head_clearance
    gripper_target_pos = head_target_pos - grip_axis_vec * measured_head_offset
    gripper_target_pos[2] = desired_head_z

    action = np.concatenate([gripper_target_pos, grip_quat2, np.array([-1.0])])
    done = do_step(action)
    if done: return done

    # Pushing
    total_push = push_distance
    step_len = 0.04
    n_steps = int(max(1, math.ceil(total_push / step_len)))
    n_steps = min(n_steps, 15)
    per_step = (total_push / float(n_steps)) * push_dir

    cur_pos = gripper_target_pos.copy()
    for _ in range(n_steps):
        cur_pos = cur_pos + per_step
        cur_pos[2] = desired_head_z
        action = np.concatenate([cur_pos, grip_quat2, np.array([-1.0])])
        done = do_step(action)
        if done: return done
        positions = env.get_object_positions()
        ball_pos = np.array(positions['hockey_ball_pos'], dtype=float)
        goal_pos = np.array(positions['hockey_goal_pos'], dtype=float)
        if np.linalg.norm((goal_pos - ball_pos)[:2]) < 0.06:
            break

    # Retract and open
    gripper_pose = env.obs.gripper_pose
    gripper_pos = np.array(gripper_pose[0:3], dtype=float)
    retract_pos = gripper_pos + np.array([0.0, 0.0, lift_height])
    action = np.concatenate([retract_pos, grip_quat2, np.array([-1.0])])
    done = do_step(action)
    if done: return done

    action = np.concatenate([retract_pos, grip_quat2, np.array([1.0])])
    done = do_step(action)
    return done

# ----------------------------
# Policy Parameter Configurations
# ----------------------------
```

```

policy_info = {
    'parameters': {
        # Optimized parameters discovered after Optuna TPE search
        'back_offset': -0.00844181895408886,
        'ball_radius': -0.016330623709603705,
        'grasp_offset_along_handle': 0.06487959972631457,
        'head_clearance': -0.05114117627708561,
        'lateral_offset': 0.00640082184267126,
        'lift_height': 0.013446534590941334,
        'pregrasp_height': 0.2345345857512359,
        'push_distance': 1.227929796720189,
        'stick_head_offset': 0.18698448793380976,
        'stick_shaft_diameter': 0.027516680202429505
    }
}
```

Listing 1: A full example policy code for Hockey task generated by LOGIC