

RAPID: Robot Agentic Programming from Demonstrations

Yuyao Liu^{1,2}, Jiayuan Mao^{3,†}, David Hsu^{2,4,†}, Leslie Pack Kaelbling^{1,†}, and Tomás Lozano-Pérez^{1,†}

¹Massachusetts Institute of Technology ²National University of Singapore

³University of Pennsylvania ⁴NVIDIA

Abstract—Coding agents have demonstrated enormous success in solving complex programming problems. To leverage their potential for robot systems, this work introduces Robot Agentic Programming from Demonstrations (RAPID), which automatically generates, verifies, and refines robot programs, given a single visual human demonstration. The iterative agentic loop of code refinement requires several key ingredients: (i) a testable task specification, (ii) action primitives for robot execution, and (iii) an interactive environment for program execution and verification. RAPID infers all three from the demonstration automatically. To make the resulting program reusable beyond the demonstration setting, RAPID uses an *object-centric relational program representation* that focuses on the underlying structure of the demonstrated strategy rather than the specific motion *per se*: it expresses the action primitives as trajectory-optimization programs that realize object-level motion effects, while composing them through relational constraints that capture scene-specific geometry at run time. We evaluated RAPID in simulation on eight challenging contact-rich nonprehensile manipulation tasks as well as general prehensile manipulation tasks in the LIBERO-Pro benchmark. We also successfully deployed it on a real Franka arm and evaluated on all eight nonprehensile tasks. In all experiments, RAPID demonstrated strong performance, with generalization over object pose, shape, material, and environment. Website: <https://yuyao.liu.me/projects/rapid>.

I. INTRODUCTION

Recently, coding agents [1], [2] have shown remarkable capabilities in developing complex software systems, sparking growing interest in solving robotic manipulation tasks through programming [3]–[9]. Consider how a human engineer would develop a robot program for the task in Fig. 1 before the advent of coding agents: given a *task* (pick up the cereal box), they would implement a program *representing* the solution strategy (press move→push→flip→grasp), and *verify* the program on test cases and iteratively debug it. This workflow mirrors the core strength of coding agents: not one-shot code generation, but the iterative cycle of generating, executing, verifying, and revising a program, commonly referred to as the agentic coding loop [10], [11].

However, bringing this agentic coding loop to robotic manipulation requires several ingredients that are not readily available: (a) a testable task specification that defines the intended outcome, (b) manipulation primitives that bridge program-level reasoning and low-level execution, and (c) interactive environments for verification. Existing robotic coding-agent systems typically assume these ingredients

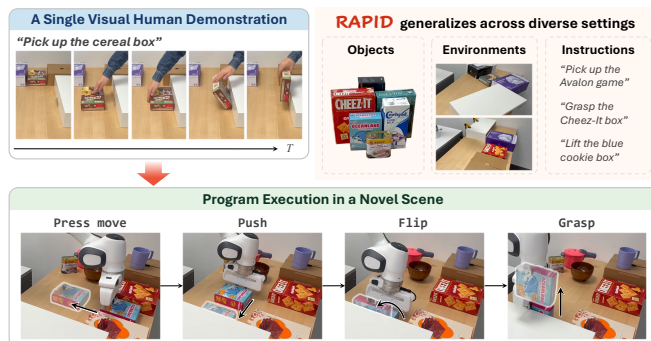


Fig. 1: RAPID (Robot Agentic Programming from Demonstrations) generates a reusable manipulation program from a single visual human demonstration; the program generalizes across variations in object appearance, pose, geometry, physical properties, and scene configuration.

are provided *a priori*, through ground-truth rewards and success signals, human-engineered primitives, and resettable environments. This raises a fundamental question: **how can we close the agentic coding loop without assuming that these ingredients are provided *a priori*?**

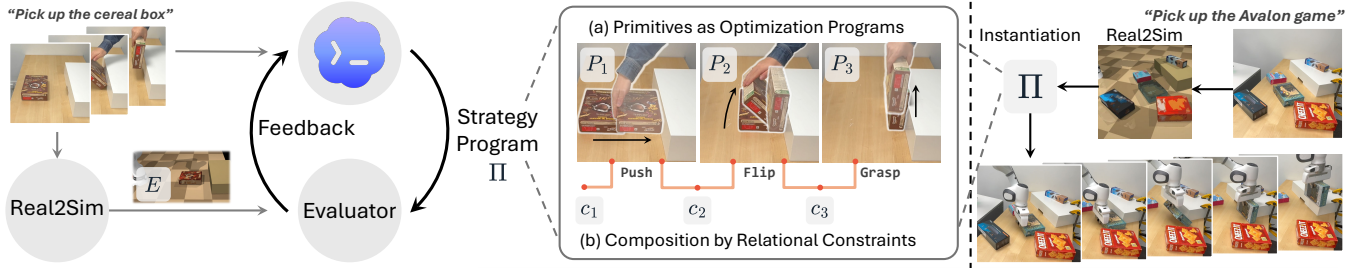
Our key idea is to use *demonstrations* as the programming interface for supplying the task- and motion-level information needed by the agentic coding loop. A demonstration directly shows a feasible way to accomplish a task and, together with a language description, conveys the intended outcome. Motivated by this perspective, we introduce **Robot Agentic Programming from Demonstrations (RAPID)**, a framework for generating and iteratively refining robot programs from *a single visual human demonstration* with a coding agent (Fig. 2). From the demonstration, RAPID infers the task specification, constructs manipulation primitives and composes them into a strategy. It further reconstructs an interactive simulation environment from the demonstration for program verification and refinement, and generates scene variants to improve generalization. In this way, a single demonstration bootstraps the task specification, manipulation primitives, and verification environment for closing the agentic coding loop.

Rather than merely reproducing the demonstration, we aim to extract a reusable and generalizable manipulation program. Programs that directly encode poses, distances, or waypoints may succeed in the demonstrated scene, but can fail when the scene changes. In this work, we propose an *object-centric relational program representation*, where the solution strategy is represented by object-centric relations and effects. Specifically, RAPID writes primitives as local trajectory-optimization programs that realize object-level motion effects, such as pushing an object by a displacement or flipping it by an angle, while resolving scene-dependent

[†] Equally advising.

Corresponding author: Yuyao Liu (yuyao.liu@mit.edu).

Work done while Yuyao Liu was visiting NUS.



Closing the Agentic Loop from a Demonstration **Object-Centric Relational Program Representation** **Strategy Instantiation in Novel Scenes**

Fig. 2: Overview of RAPID. E is a simulation environment reconstructed from the demonstration, P_i 's are manipulation primitives, c_i 's are relational constraints connecting two consecutive primitives, and Π is the strategy program generated by the coding agent.

geometry at execution time. At the task level, primitives are composed into a strategy through relational constraints that determine how the outcome of one phase specifies the next. For example, in Fig. 3, rather than replaying a fixed motion, the program first computes the goal position from the relation between the brown obstacle and the white fixture, and then determines how far to move the target object during the current primitive (press move) based on the state produced by the preceding primitive (flip). By expressing primitives and their composition relationally, RAPID preserves the invariant structure of the demonstrated strategy while adapting to changes in object appearance, pose, geometry, physical properties, and scene configuration.

The remaining challenge is *verification*: the coding agent needs an interactive environment in which candidate programs can be executed, evaluated, and revised. RAPID reconstructs such an environment in simulation from the demonstration and augments it with scene variants generated by perturbing the demonstrated scene. The coding agent then generates task-specific filters that discard variants that are physically infeasible or inconsistent with the task. Afterwards, the coding agent executes candidate programs across this verification set and iteratively revises them based on observed failures.

In this work, we focus on *nonprehensile manipulation*, a particularly challenging domain in robotic manipulation. Many prehensile tasks can be composed from a compact and well-established set of reusable primitives [12], such as *pick*, *move in free space*, and *place*. Nonprehensile manipulation, by contrast, involves a much broader range of geometry- and contact-dependent interactions, including pushing [13], pivoting [14], toppling [15], flipping [16], and sliding [17]. Capturing this diversity with a fixed primitive library is difficult, making manual primitive design a substantial engineering burden. Nonprehensile manipulation therefore provides a natural testbed for studying whether a coding agent can construct reusable manipulation primitives from a single demonstration rather than relying on human engineering.

We evaluate RAPID on eight contact-rich, multi-step nonprehensile manipulation tasks in simulation, asking whether programs constructed from a single demonstrated instance can generalize to substantial variations in object appearance, pose, geometry, physical properties, and scene configuration. RAPID outperforms baselines without relying on pre-specified success signals, human-engineered manipulation primitives or provided resettable environments. To evaluate the generality of RAPID beyond nonprehensile manipulation, we additionally

test it on LIBERO-Pro [18], a simulation benchmark featuring prehensile manipulation tasks. Finally, we demonstrate that RAPID can be deployed effectively on a real robotic system.

II. RELATED WORK

Programming for Robotics. Code as Policies [9] uses robot programs to connect language reasoning with control. Related work generates programs [19], [20], geometric constraints [21], [22], rewards [23], [24], and simulation environments [25]. Agentic programming adds iterative execution and refinement: CaP-X [6] and related systems [3], [4], [8], [26] improve programs through feedback, while Voyager [27] and skill-discovery approaches [5], [28] grow reusable libraries. However, these approaches typically rely on task success signals, human-engineered primitives, or verification environments supplied *a priori*. In contrast, RAPID derives these ingredients from a single visual human demonstration to close the agentic coding loop.

Learning from Few Demonstrations. Learning from few demonstrations benefits from representations that preserve task-relevant structure across objects and scenes. Spatial abstractions support transfer through visual alignment [29], object geometry and correspondence [30]–[33], and contact or interaction structure [34]–[37]. Demonstrations can also be decomposed into subgoals and primitives [38], [39]. Structured skill representations enable long-horizon planning and composition through contact retargeting, symbolic models, or geometric reasoning [40]–[44]. However, these representations are generally embedded in fixed learning or planning pipelines. In contrast, RAPID combines an object-centric relational program representation of manipulation effects and composition constraints with agentic programming to iteratively construct, verify, and refine transferable manipulation strategies.

Nonprehensile Manipulation. Nonprehensile manipulation exploits environmental contact to reposition objects and enable otherwise difficult grasps [45]. Prior work develops mechanics-based models [14], [46], contact-aware planners [15], [17], [47], and optimization methods for trajectories and primitives [48]–[51]. Learning-based approaches acquire contact-rich policies [52]–[54] or compose predefined primitives [55]. However, these methods typically rely on explicit task specifications or task-specific policy training. In contrast, RAPID approaches nonprehensile manipulation through agentic programming from a single visual human demonstration, automatically constructing and iteratively refining the required primitives and their composition.

III. OVERVIEW OF RAPID

Problem Statement. We consider programming a reusable strategy for quasi-static rigid-body manipulation from a *single visual human demonstration* D and a language description l . The program should solve a family of task instances $\mathcal{E}$, each comprising a scene-instruction pair (E, l) , with similar objectives but varying object appearance, geometry, pose, physical properties, and scene configuration. No manipulation primitives, explicit success or reward signals, or instantiated simulation environments are provided *a priori*. Given only (D, l) , the agent aims to construct executable primitives $\mathcal{P} = \{P_1, \dots, P_N\}$, which produce low-level control signals, and compose them into a strategy Π that captures the demonstration’s underlying structure and generalizes to unseen task instances $(E^*, l^*) \in \mathcal{E}$:

$$(D, l) \mapsto (\mathcal{P}, \Pi), \quad \Pi(E^*, l^*; \mathcal{P}) \text{ succeeds for } (E^*, l^*) \in \mathcal{E}.$$

Program Representation. We introduce an *object-centric relational program representation* that enables strategies constructed by the agent to generalize across different scenes. RAPID represents each primitive as a geometry- and physics-aware local trajectory-optimization program that achieves an object-level effect. A strategy then composes these primitives through relational constraints that specify the desired effect of each primitive. We detail this representation in Sec. IV.

Strategy Construction. As shown in Fig. 2, RAPID closes the agentic coding loop by deriving task specifications, manipulation primitives, and verification environments from (D, l) . It segments the demonstration into several phases, infers a subgoal and success predicate for each phase, and performs real-to-sim reconstruction to obtain interactive simulation environments. RAPID further synthesizes feasible scene variants for additional verification. For each phase, the coding agent generates a primitive and iteratively executes, verifies, and revises it across the reconstructed environment and its variants. RAPID then infers the overall task goal and success predicate, composes the primitives into a strategy, and refines the composition through the same loop. We describe this construction and verification process in Sec. V.

Strategy Deployment. At deployment, given an RGB-D observation of a novel scene and a language instruction, RAPID reconstructs the scene in simulation, instantiates the strategy in the reconstructed scene, and executes the resulting trajectory on the robot. Lightweight semantic binding assigns scene objects to the semantic roles in the strategy, while the strategy and primitives themselves remain unchanged.

IV. OBJECT-CENTRIC RELATIONAL PROGRAMS

We now define our object-centric relational program representation: primitives (Sec. IV-A), their composition into task-level strategies (Sec. IV-B), and strategy instantiation in novel scenes (Sec. IV-C).

A. Primitives as Local Trajectory-Optimization Programs

A primitive is a reusable behavior for realizing an object-level motion effect, such as pushing an object by a displacement or flipping an object by an angle. We represent a

TABLE I: Local trajectory-optimization program for the primitive *flip*.

Component	Content
Semantic goal G_i	Flip the target object against a fixture by a specific angle.
Relational interface $C_i = (\mathbf{r}_i, \Gamma_i, \mathbf{q}_i)$	Semantic roles $\mathbf{r}_i$: (target object, support plane, fixture, manipulator). Relations Γ_i : the object rests on the support plane, and the fixture exposes a vertical face oriented toward the target object and reachable by the manipulator. Effect parameter $\mathbf{q}_i$: the requested flipping angle.
Success predicate Ψ_i	Check that the achieved rotation matches the flipping angle and that the target object remains supported, near the fixture, and settled.
Geometry resolver ρ_i	Resolve the support plane, the vertical face of the fixture, the flipping axis, and the dimensions of the target object at each step.
Final cost J_i^f	Penalize terminal rotation error relative to the flipping angle, the gap between the fixture and the target object, and terminal velocity.
Progress cost J_i^p	Encourage rotation and manipulator proximity while discouraging backtracking and irregular motion.

primitive as a local relational trajectory-optimization program

$$P_i = (G_i, C_i, \Psi_i, \rho_i, J_i) \in \mathcal{P}.$$

Here, G_i is a semantic, object-centric description of the intended effect; C_i is the relational interface; Ψ_i is the verifiable success condition; ρ_i is the geometry resolver; and J_i is the optimization cost. Together, these components specify how the primitive is instantiated, optimized, and verified in a specific scene. A primitive is therefore neither a fixed trajectory nor a black-box policy. Table I illustrates each component for flipping an object against a fixture.

Relational Interface. We write the primitive interface as

$$C_i = (\mathbf{r}_i, \Gamma_i, \mathbf{q}_i).$$

Here, $\mathbf{r}_i$ declares the semantic roles involved in the primitive; for example, for the primitive *flip* in Table I, these roles are (*target object*, *support plane*, *fixture*, *manipulator*). Γ_i specifies the relations among these roles; for *flip*, it encodes that the target object rests on the support plane and that the fixture exposes a vertical face oriented toward the target object and reachable by the manipulator. Finally, $\mathbf{q}_i$ declares the effect parameter; for *flip*, it is the requested flipping angle.

To apply a primitive in a scene, an invocation supplies runtime arguments (b_i, g_i) satisfying

$$b_i \in \text{Bind}_E(\mathbf{r}_i, \Gamma_i), \quad g_i \in \text{Val}(\mathbf{q}_i),$$

where $\text{Bind}_E(\mathbf{r}_i, \Gamma_i)$ denotes the type-compatible assignments of the objects in the scene E to $\mathbf{r}_i$ that satisfy Γ_i , and $\text{Val}(\mathbf{q}_i)$ denotes the values matching the types declared by $\mathbf{q}_i$. Scene-dependent geometry remains internal to the primitive and is resolved by ρ_i at execution time.

Geometry Resolver. Let $\mathcal{X}(E)$ be the state space of environment E . For the state $x_t \in \mathcal{X}(E)$ at time step t , the geometry resolver ρ_i computes the geometric features $\phi_{i,t}$:

$$\phi_{i,t} = \rho_i(E, x_t, b_i), \quad t = 0, \dots, T.$$

For example, for the primitive *flip* in Table I, the agent would write detectors for the support plane, the vertical face of the fixture, the flipping axis, etc.

Success Predicate. For a terminal state $x_T \in \mathcal{X}(E)$, the predicate $\Psi_i(E, x_T; g_i, \phi_{i,T}) \in \{0, 1\}$ provides an executable version of G_i . It checks the achieved object relations and requested effect, yielding an unambiguous signal for verification.

Optimization Cost. For a simulated trajectory $\tau = (x_0, \dots, x_T)$

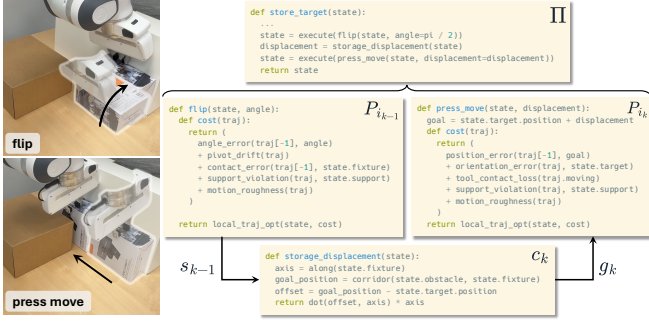


Fig. 3: Composing primitives by relational constraints.

in environment E , where $x_t \in \mathcal{X}(E)$, the primitive cost is

$$J_i(E, \tau; g_i, \phi_{i,0:T}) = J_i^f(E, x_T; g_i, \phi_{i,T}) + \lambda_i J_i^p(E, \tau; g_i, \phi_{i,0:T}).$$

Here, J_i^f is the final cost, which measures how far x_T is from the effect encoded by G_i ; J_i^p is the progress cost, which may depend on the full trajectory and guides the optimizer through contact-rich interaction heuristics; and $\lambda_i \geq 0$ is a constant. The coding agent proposes the heuristics by reasoning about the visual human demonstration.

During primitive construction, the agent also proposes an initialization heuristic based on the resolved geometry and requested effect. The heuristic provides a plausible interaction trajectory from which to begin the search. Let z parameterize a candidate robot motion, let $\tau(z) = \text{Rollout}_E(x_0, z)$ denote its simulated state trajectory from x_0 in environment E , and let $\phi_{i,0:T}(z)$ be the feature sequence resolved along that trajectory. The executable trajectory is obtained as

$$z_i^* = \arg \min_z J_i(E, \tau(z); g_i, \phi_{i,0:T}(z)).$$

We solve this trajectory-optimization problem with the cross-entropy method (CEM) [56], a gradient-free sampling-based optimizer that allows us to optimize robot motions without differentiating through the contact-rich simulation. The primitive thereby adapts its motion to the object pose, size, geometry, physical properties, and requested effect.

B. Strategies as Primitive Composition by Constraints

After the primitives $\mathcal{P}$ are constructed, where each $P_i \in \mathcal{P}$ realizes an object-level motion effect, the next step is to compose them into a strategy Π , a reusable task-level program that accomplishes an overall task. It specifies the sequence of primitives to invoke, how task roles map to each primitive, and how the requested effect of each invocation is determined:

$$\Pi = \left(G_\Pi, C_\Pi, \Psi_\Pi, \langle (P_{i_k}, m_k, c_k) \rangle_{k=1}^K \right),$$

where G_Π describes the overall task goal and Ψ_Π evaluates its final outcome.

Strategy Interface. A strategy exposes semantic roles and their required relations,

$$C_\Pi = (\mathbf{r}_\Pi, \Gamma_\Pi).$$

Invoking the strategy requires a semantic role binding $b \in \text{Bind}_E(\mathbf{r}_\Pi, \Gamma_\Pi)$ that assigns an entity to each strategy role while satisfying Γ_Π . Primitive effect parameters are not exposed through C_Π ; instead, they are derived inside the

strategy during execution.

Primitive Composition by Constraints. A strategy consists of K phases, each represented by a tuple (P_{i_k}, m_k, c_k) , where $P_{i_k} \in \mathcal{P}$ is a primitive, m_k is a role map, and c_k is a connecting constraint. For the invoked primitive P_{i_k} , m_k produces a role binding b_k that fills the role declarations $\mathbf{r}_{i_k}$, while c_k produces an effect argument g_k that fills the parameter declarations $\mathbf{q}_{i_k}$. The connecting constraint declaratively specifies how phase k depends on the outcome of phase $k-1$ and is evaluated just in time at the beginning of phase k :

$$b_k = m_k(b) \in \text{Bind}_E(\mathbf{r}_{i_k}, \Gamma_{i_k}),$$

$$g_k = c_k(E, s_{k-1}, b) \in \text{Val}(\mathbf{q}_{i_k}),$$

where $s_{k-1} \in \mathcal{X}(E)$ denotes the environment state after the execution of phase $k-1$. The phase is then executed as

$$s_k = \text{Exec}(P_{i_k}; E, s_{k-1}, g_k, b_k).$$

Here, Exec solves the local trajectory-optimization problem of P_{i_k} and executes the optimized trajectory.

For example, Fig. 3 shows a *press move* following a *flip*. The role map m_k maps the strategy's target object, supporting surface, and tool roles to the corresponding roles of the *press move* primitive, producing b_k . The connecting constraint c_k determines the goal position from the relation between the brown obstacle and the white fixture. Using the target object's current position in the post-flip state s_{k-1} , it computes the displacement toward this goal along the fixture axis, producing the effect argument g_k for the *press move* primitive.

C. Strategy Instantiation in Novel Scenes

To deploy a strategy in a novel scene, we only need to provide a semantic binding. Given a novel scene E^* and instruction l^* , RAPID uses a coding agent to rapidly generate a semantic binding $b^* \in \text{Bind}_{E^*}(\mathbf{r}_\Pi, \Gamma_\Pi)$ that assigns scene entities to the semantic roles exposed by the strategy interface. Since this step only associates the exposed semantic roles with scene entities, it is lightweight and incurs little deployment overhead (a few seconds). At phase k , the strategy derives the primitive binding and effect argument automatically.

Throughout this process, the strategy program Π and the referenced primitives remain frozen. Generalization therefore arises from the object-centric relational program representation, together with the adaptation of trajectory optimization to the geometry and physical properties of the new scene.

V. AGENTIC PROGRAMMING FROM A DEMONSTRATION

We now describe how RAPID constructs and verifies the programs defined in Sec. IV, covering environment reconstruction, primitive construction, and strategy composition.

A. Reconstructing Interactive Environments

Interactive Environment. To verify a manipulation program, RAPID reconstructs an interactive simulation environment from the initial RGB-D frame of the demonstration. We first use a vision-language model (VLM) to identify the objects in the scene, assign semantic labels to them, and estimate their physical properties. Then, for each semantic name, we utilize

Algorithm 1: RAPID

Input: Human demonstration D , instruction l
Output: Verified primitives $\mathcal{P}$ and strategy Π

```
1  $\{(D_i, l_i)\}_{i=1}^K \leftarrow \text{Segment}(D, l)$ 
2  $\mathcal{P} \leftarrow \emptyset$ 
3 // Primitive construction ▷ Sec. V-B
4 for  $i = 1, \dots, K$  do
5    $\hat{E}_i \leftarrow \text{Reconstruct}(D_i)$  ▷ Sec. V-A
6    $(G_i, \Psi_i) \leftarrow \text{InferObjective}(D_i, l_i)$ 
7    $g_i \leftarrow \text{InferEffect}(D_i, G_i)$ 
8    $P_i \leftarrow \text{ProposePrimitive}(D_i, G_i, \Psi_i, \hat{E}_i)$ 
9    $\mathcal{V}_i \leftarrow \{(g_i, \hat{E}_i)\} \cup \text{Variants}(g_i, \hat{E}_i, P_i)$ 
10   $P_i \leftarrow \text{VerifyRevise}(P_i, \mathcal{V}_i, l_i)$ 
11   $\mathcal{P} \leftarrow \mathcal{P} \cup \{\text{Freeze}(P_i)\}$ 
12 // Strategy composition ▷ Sec. V-C
13  $\hat{E}_0 \leftarrow \text{Reconstruct}(D)$ 
14  $\Pi \leftarrow \text{ComposeStrategy}(D, l, \mathcal{P}, \hat{E}_0)$ 
15  $\mathcal{V}_\Pi \leftarrow \{\hat{E}_0\} \cup \text{Variants}(\hat{E}_0, \Pi)$ 
16  $\Pi \leftarrow \text{VerifyRevise}(\Pi, \mathcal{V}_\Pi, l)$ 
17 return  $\mathcal{P}, \text{Freeze}(\Pi)$ 
18 // Shared verification and revision
19 function  $\text{VerifyRevise}(Q, \mathcal{V}, a)$ 
20   repeat
21      $\mathcal{R} \leftarrow \text{ExecuteEvaluate}(Q, \mathcal{V}, a)$ 
22     if  $\neg \text{AllSuccessful}(\mathcal{R})$  then
23        $Q \leftarrow \text{DiagnoseRevise}(Q, \mathcal{R})$ 
24   until  $\text{AllSuccessful}(\mathcal{R})$ 
25   return  $Q$ 
```

SAM 3 [57] to get a segmentation mask. The RGB-D frame and the mask are then fed into SAM 3D [58] to reconstruct a complete mesh. Finally, we use FoundationPose [59] to register each mesh to the RGB-D frame and assemble the reconstructed objects and robot model in MuJoCo [60]. The resulting environment preserves the scene entities, geometry, and spatial relations needed to execute the candidate program and evaluate its success predicates.

Feasible Scene Variants. To construct variants of the demonstrated scene, RAPID uses a task-agnostic variation sampler to randomize object appearances, sizes, poses, and physical parameters such as mass, friction, and elasticity. However, the randomly sampled scenes might not be feasible for a certain primitive or strategy. Therefore, we ask the coding agent to write a filter for each primitive or strategy that accepts only variants preserving the declared roles and relations, task meaning, and physical feasibility.

B. Primitive Construction

Given the full demonstration D and its natural language description l , RAPID calls a subagent to partition D into an ordered sequence of interaction segments $(D_1, \dots, D_K)$ and assign a local description l_i to each D_i . For each (D_i, l_i) , if the coding agent recognizes that the behavior is covered by an existing primitive in the primitive library $\mathcal{P}$, we can simply retrieve this primitive. Otherwise, the coding agent will infer a primitive specification consisting of a semantic goal G_i and a success predicate Ψ_i from (D_i, l_i) . It further reasons about the contact interactions that produce the demonstrated effect, the relational structure that should remain invariant across executions, and the quantities that might vary between invocations. For example, in Table I, the demonstrated rotation

supplies the flipping-angle argument, while relations among the target object, support plane, fixture, and manipulator define the reusable interaction structure.

Given this specification and interaction analysis, the coding agent generates the relational interface C_i , geometry resolver ρ_i , and optimization cost J_i introduced in Sec. IV-A. As shown in Algorithm 1, RAPID verifies and revises P_i across a set $\mathcal{V}_i$ containing the demonstrated effect–environment pair $(g_i, \hat{E}_i)$ and feasible variants of both the environment and effect argument generated by a task-agnostic sampler. Verification continues until Ψ_i is satisfied for every pair using its associated effect argument. The verified primitive is then added to the library $\mathcal{P}$.

C. Strategy Composition

After constructing the required primitives, RAPID composes them into a task-level strategy. As shown in Algorithm 1, given the full demonstration D and instruction l , the coding agent infers a strategy specification consisting of the overall task goal G_Π and final success predicate Ψ_Π . The sequence of the interaction segments $(D_1, \dots, D_K)$ determines the ordering of the primitive composition. The agent then constructs the strategy interface C_Π and, for each phase k , selects a verified primitive P_k , defines the role map m_k , and specifies the connecting constraint c_k . Together, these components form an initial version of the strategy Π .

RAPID verifies the strategy in the reconstructed environment $\hat{E}_0$ and its feasible scene variants using the execution procedure in Sec. IV-B. Failures guide the coding agent to revise the composition while keeping the verified primitives fixed, until Ψ_Π is satisfied across this verification set $\mathcal{V}_\Pi$.

VI. EXPERIMENTS

In this section, we conduct experiments showing that (1) RAPID successfully closes the agentic coding loop from a single visual human demonstration to construct reusable programs for nonprehensile manipulation (Table II); (2) our object-centric relational program representation improves the generalization of the constructed manipulation programs (Tables II and III); and (3) RAPID can be effectively deployed on a real-world robotic system (Fig. 5).

A. Experimental Setup for Nonprehensile Manipulation

Tasks. In this section, we evaluate RAPID on eight nonprehensile manipulation tasks in simulation. For each task, we provide a single visual human demonstration together with a language description. In all tasks, the objects are initially not graspable by the parallel gripper, so the robot must execute a sequence of nonprehensile manipulation primitives, such as pushing, flipping, pivoting, and toppling, to reach the target configuration. Importantly, the primitives are *not* provided to the coding agent *a priori*. We provide the descriptions of the 8 tasks below, and Fig. 4 illustrates them in the real world: **T1.** *Push the object toward the corridor, switch the contact face, and push it again to store it inside the corridor.* **T2.** *Push the object toward the fixture, flip it to expose a graspable edge, and grasp it.*

TABLE II: Success rates on nonprehensile manipulation tasks in simulation. We report mean $\pm$ std across three trials; the best results are **bolded**.

Method	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6	Task 7	Task 8	Average
CaP	0.053 $\pm$ 0.009	0.000 $\pm$ 0.000	0.000 $\pm$ 0.000	0.000 $\pm$ 0.000	0.000 $\pm$ 0.000	0.000 $\pm$ 0.000	0.013 $\pm$ 0.009	0.040 $\pm$ 0.016	0.013 $\pm$ 0.003
CaP + OReP	0.040 $\pm$ 0.000	0.000 $\pm$ 0.000	0.000 $\pm$ 0.000	0.000 $\pm$ 0.000	0.000 $\pm$ 0.000	0.000 $\pm$ 0.000	0.013 $\pm$ 0.009	0.033 $\pm$ 0.025	0.011 $\pm$ 0.002
CaP-Agent0	0.353 $\pm$ 0.115	0.040 $\pm$ 0.057	0.000 $\pm$ 0.000	0.013 $\pm$ 0.019	0.173 $\pm$ 0.176	0.053 $\pm$ 0.041	0.240 $\pm$ 0.128	0.293 $\pm$ 0.242	0.146 $\pm$ 0.025
CaP-Agent0 + SV	0.440 $\pm$ 0.141	0.053 $\pm$ 0.009	0.020 $\pm$ 0.028	0.047 $\pm$ 0.066	0.113 $\pm$ 0.034	0.080 $\pm$ 0.059	0.167 $\pm$ 0.066	0.087 $\pm$ 0.096	0.126 $\pm$ 0.043
RAPID w/o SV	0.933 $\pm$ 0.009	0.833 $\pm$ 0.125	0.400 $\pm$ 0.161	0.440 $\pm$ 0.315	0.447 $\pm$ 0.256	0.340 $\pm$ 0.236	0.567 $\pm$ 0.229	0.293 $\pm$ 0.415	0.532 $\pm$ 0.071
RAPID	0.933 $\pm$ 0.057	0.847 $\pm$ 0.025	0.700 $\pm$ 0.033	0.787 $\pm$ 0.082	0.793 $\pm$ 0.009	0.587 $\pm$ 0.050	0.807 $\pm$ 0.203	0.620 $\pm$ 0.028	0.759 $\pm$ 0.024

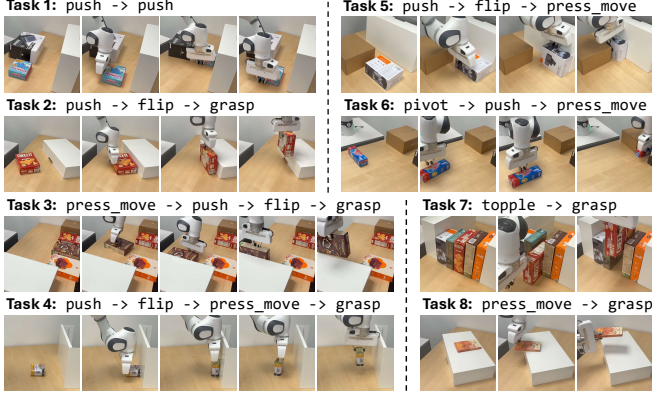


Fig. 4: Real-world execution of the solution trajectories found by RAPID.

T3. Move the object away from the obstacles, push it toward the fixture, flip it to expose a graspable edge, and grasp it.

T4. Push the object toward the fixture, flip it, reposition it to expose a graspable edge, and grasp it.

T5. Push the object toward the fixture, flip it to align with the opening, and move it into the opening for storage.

T6. Pivot the box to align it with the corridor, push it toward the corridor, and then move it to store it inside the corridor.

T7. Topple the object to expose an edge and grasp it.

T8. Move the object to the side of the platform to expose a graspable edge, and grasp it.

Benchmark. We construct a simulation benchmark in MuJoCo [60] using a Franka Research 3 robot arm equipped with a compliant operational-space controller [61]. Our primary evaluation metric is *generalization*. For each of the 8 tasks, we construct 50 novel test scenes that differ from the demonstration scene in object pose, appearance, size, shape, and material. We also introduce irrelevant objects to make the test scenes visually distracting. We evaluate each method by measuring its success rate on each task using a private success criterion hidden from RAPID. We run three trials of each experiment and report the mean and standard deviation.

Baselines. We compare six approaches along three principal axes: the closed agentic loop (*Agent*), the *object-centric relational program representation* (*OReP*), and synthesized scene variants (*SV*). 1) *Open-loop + Python* (*CaP*): For each novel test scene, we use Code as Policies (CaP) [9] to generate a Python program conditioned on the demonstration, language description, and observation, without simulation feedback during program generation. 2) *Open-loop + OReP* (*CaP + OReP*): We augment CaP with our OReP and cross-entropy method (CEM) optimizer, enabling motions to be expressed relative to scene entities. 3) *Agent + Python* (*CaP-Agent0*): We use CaP-Agent0 from CaP-X [6] to assess the benefit of our program representation. We give CaP-Agent0 a more favorable agentic-programming setting than

that of RAPID: during program construction, it receives rollout feedback consisting of ground-truth success signals, environment states, and visual observations; at evaluation time, it may additionally execute rollouts in each test scene and revise its program. 4) *Agent + Python + SV* (*CaP-Agent0 + SV*): We further augment CaP-Agent0 with our scene-variant generation module, allowing it to refine its program on synthesized scene variants. 5) *Agent + OReP* (*RAPID w/o SV*): We remove the scene-variant generation module from RAPID. 6) *Agent + OReP + SV* (*RAPID*): Our full method. Comparisons of 4) vs. 3) and 6) vs. 5) isolate the benefit of variant-based verification. We do not compare with ASPIRE [5] on nonprehensile tasks because it requires motion primitives to be provided. We use Codex [1] powered by GPT-5.6 Sol with high reasoning effort as the coding agent and provide the same robot control APIs for all methods.

B. Simulation Experiments for Nonprehensile Manipulation

RAPID successfully constructs reusable programs by closing the agentic loop from a single demonstration. As shown in Table II, vanilla CaP performs poorly despite being allowed to generate a program tailored to each test scene. Without an interactive environment for verifying its programs, the agent cannot check whether the proposed motions produce the intended object interactions. Augmenting CaP with our object-centric relational program representation does not improve its performance either, suggesting that the representation alone is insufficient: without closed-loop verification, the coding agent struggles to reason about object relations and therefore fails to construct meaningful relational programs. In contrast, RAPID reconstructs a simulation environment from the demonstration, infers a success predicate from the demonstration and language instruction, and uses this predicate to verify and iteratively refine candidate programs in simulation. In this way, RAPID constructs a reusable program for each task that generalizes across diverse test scenes. In our experiments, this construction takes on the order of tens of minutes per task. This cost is incurred once per task: the resulting strategy and primitives remain fixed across the 50 novel scenes, with semantic role binding taking only a few seconds per scene.

The object-centric relational program representation enhances the generalization of the strategy program.

The comparison with the two CaP-Agent0 baselines in Table II supports the benefit of our program representation: even with stronger verification access, including ground-truth feedback and evaluation-time refinement, CaP-Agent0 substantially underperforms RAPID. Although CaP-Agent0 incorporates state feedback, its generated policies often use

TABLE III: Experimental Results on LIBERO-Pro.

Method	libero-object		libero-goal		libero-spatial	
	Pos. (Avg.)	Task. (Avg.)	Pos. (Avg.)	Task. (Avg.)	Pos. (Avg.)	Task. (Avg.)
OpenVLA	0	0	0	0	0	0
π_0	0	0	0	0	0	0
$\pi_{0.5}$	0.17	0.01	0.38	0	0.20	0.01
CaP-Agent0	0.22	0.18	0.26	0.17	0.12	0.14
ASPIRE	0.98	0.95	0.81	0.45	0.51	0.60
RAPID	1	0.95	0.67	0.53	0.83	0.81

fixed world-frame offsets and predetermined timing, limiting spatial generalization and robustness to state deviations during execution. In contrast, RAPID computes geometry-dependent subgoals from the state reached after each preceding phase. The gap is even larger for contact-rich, forceful behaviors such as flipping and toppling, which are naturally expressed by our relational, optimization-based motion primitives.

Ablation study of scene variant generation. To evaluate the effectiveness of scene variant generation, we remove it from RAPID and report the results in Table II. Except for Tasks 1 and 2, the performance of RAPID drops substantially, while the standard deviation of the success rate increases, indicating reduced stability and robustness. Interestingly, augmenting CaP-Agent0 with scene variant generation does not improve its performance. This suggests that, without our object-centric relational program representation, CaP-Agent0 struggles to construct a generalizable program that can solve diverse generated variants without overfitting to individual scenes.

C. Simulation Experiments on LIBERO-Pro

Experimental Setup. To demonstrate the general applicability of RAPID beyond nonprehensile manipulation, we evaluate it on LIBERO-Pro [18], an extension of LIBERO [62] with stronger position (Pos.) and instruction (Task.) perturbations. LIBERO-Pro predominantly features prehensile tasks, including pick-and-place and grasp-based articulated-object manipulation. We compare against two groups of baselines: (1) vision-language-action (VLA) models, including OpenVLA [63], π_0 [64], and $\pi_{0.5}$ [65], whose results are directly taken from the LIBERO-Pro paper; and (2) code-as-policy (CaP) methods, including CaP-Agent0 [6] and ASPIRE [5]. For a fair comparison with CaP methods, which do not use demonstrations, RAPID also receives no demonstrations and is given only the task description and a single debug scene. This setting is more restrictive than the setting used by ASPIRE, which uses 15 debug scenes per task. Moreover, while CaP baselines are provided with predefined motion primitives such as grasping, placing, and transporting, RAPID must discover and implement these primitives solely through exploration and iterative refinement in the debug scene.

Results. Table III reports the results on LIBERO-Pro. RAPID achieves the highest success rate on five of the six task settings, demonstrating the effectiveness of our object-centric relational program representation. On LIBERO-Goal (Pos.), RAPID ranks second behind ASPIRE. This gap stems from articulated-object manipulation: primitives discovered by RAPID from only a single debug scene, such as drawer opening, are less robust to spatial variation, whereas ASPIRE is provided with an *interpolate segment* motion primitive that simplifies such interactions. Overall, these results demonstrate the generality

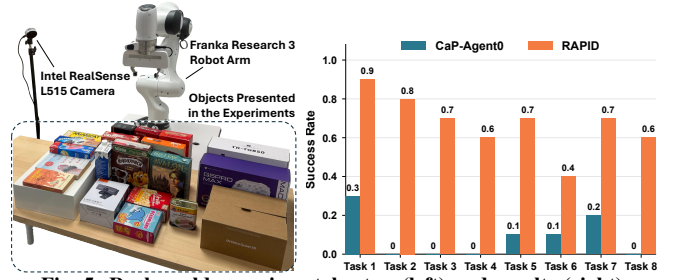


Fig. 5: Real-world experimental setup (left) and results (right).

of our framework beyond nonprehensile manipulation.

D. Real-World Experiments for Nonprehensile Manipulation

To demonstrate that RAPID can operate effectively on a real robotic system, we conduct real-world experiments on the eight nonprehensile manipulation tasks shown in Fig. 4. As illustrated in Fig. 5, our setup consists of a Franka Research 3 robot arm equipped with a parallel gripper and an Intel RealSense L515 camera for RGB-D observations. We compare RAPID against CaP-Agent0 [6]. For each task, we evaluate 10 test scenes and report the success rate.

Results. As shown in Fig. 5, with our object-centric relational program representation and compliant operational-space controller, contact-rich nonprehensile behaviors optimized in simulation can be transferred directly to the real robot and executed effectively. In contrast, CaP-Agent0 struggles to generate reliable trajectories using procedural programs. Moreover, leveraging the VLM’s prior knowledge to estimate physical parameters such as mass, friction, and restitution, together with SAM 3D’s strong 3D reconstruction capability, helps keep the real-to-sim and sim-to-real gaps manageable.

VII. CONCLUSION

We presented RAPID, a framework that closes the agentic coding loop for robotic manipulation from a single visual human demonstration and a language description. By automatically deriving task specifications, manipulation primitives, and interactive verification environments, RAPID enables a coding agent to construct, execute, and iteratively refine reusable manipulation strategies. Our object-centric relational program representation represents manipulation primitives as trajectory-optimization programs and composes them with relational constraints, enabling generalization to novel scenes. Experiments on eight contact-rich nonprehensile manipulation tasks demonstrate strong generalization across object and scene variations, while evaluations on LIBERO-Pro and a real robotic system support RAPID’s broader applicability and practical deployment.

In future work, RAPID will incorporate other tools for learning primitives, such as reinforcement learning for manipulation with dexterous hands. Further, RAPID will automatically benefit from the relentless progress in coding agents, robot foundation models, and simulation engines (e.g., for deformable objects [66]), and scale up.

REFERENCES

- [1] OpenAI, “Codex,” 2025, website: <https://openai.com/codex/>.
- [2] Anthropic, “Claude Code,” 2025, website: <https://claude.com/product/claude-code>.

- [3] K. Chen *et al.*, “GaP: A Graph-as-Policy Multi-Agent Self-Learning Harness For Variational Automation Tasks,” *arXiv:2607.05369*, 2026.
- [4] W. Xiao *et al.*, “ENPIRE: Agentic Robot Policy Self-Improvement in the Real World,” *arXiv:2606.19980*, 2026.
- [5] R. Lu *et al.*, “ASPIRE: Agentic /Skills Discovery for Robotics,” *arXiv:2607.00272*, 2026.
- [6] L. Fu *et al.*, “CaP-X: A framework for benchmarking and improving coding agents for robot manipulation,” in *ICML*, 2026.
- [7] S. Berman, M. Ilie, J. Deng, and D. Freeman, “Claude Plays Robotics,” 2026, research Blog: <https://www.anthropic.com/research/claude-plays-robotics>.
- [8] Noematrix Team, “RoboRSI: Stable, Efficient, and Reusable Robot Self-Evolution in Complex Real-World Environments,” 2026, research Blog: <https://lab.noematrix.ai/blog/2-roborsi/>.
- [9] J. Liang *et al.*, “Code as Policies: Language Model Programs for Embodied Control,” in *ICRA*, 2023.
- [10] A. Ng, “Three Key Loops for Building Great Software,” 2026, research Blog: <https://www.deeplearning.ai/the-batch/three-key-loops-for-building-great-software>.
- [11] D. Huang *et al.*, “AgentCoder: Multi-Agent-Based Code Generation with Iterative Testing and Optimisation,” *arXiv:2312.13010*, 2023.
- [12] C. R. Garrett *et al.*, “Integrated Task and Motion Planning,” *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 4, pp. 265–293, 2021.
- [13] M. T. Mason, *Mechanics of Robotic Manipulation*. MIT Press, 2001.
- [14] A. Holladay, R. Paolini, and M. T. Mason, “A General Framework for Open-Loop Pivoting,” in *ICRA*, 2015.
- [15] X. Cheng, E. Huang, Y. Hou, and M. T. Mason, “Contact Mode Guided Sampling-Based Planning for Quasistatic Dexterous Manipulation in 2D,” in *ICRA*, 2021.
- [16] L. U. Odhner, R. R. Ma, and A. M. Dollar, “Open-Loop Precision Grasping with Underactuated Hands Inspired by a Human Manipulation Strategy,” *T-ASE*, vol. 10, no. 3, pp. 625–633, 2013.
- [17] X. Cheng, E. Huang, Y. Hou, and M. T. Mason, “Contact Mode Guided Motion Planning for Quasidynamic Dexterous Manipulation in 3D,” in *ICRA*, 2022.
- [18] X. Zhou *et al.*, “LIBERO-PRO: Towards Robust and Fair Evaluation of Vision-Language-Action Models beyond Memorization,” *arXiv:2510.03827*, 2025.
- [19] I. Singh *et al.*, “ProgPrompt: Generating Situated Robot Task Plans using Large Language Models,” in *ICRA*, 2023.
- [20] C. Wang *et al.*, “Chain-of-Modality: Learning Manipulation Programs from Multimodal Human Videos with Vision-Language-Models,” in *ICRA*, 2025.
- [21] W. Huang *et al.*, “VoxPoser: Composable 3D Value Maps for Robotic Manipulation with Language Models,” in *CoRL*, 2023.
- [22] W. Huang, C. Wang, Y. Li, R. Zhang, and F.-F. Li, “ReKep: Spatio-Temporal Reasoning of Relational Keypoint Constraints for Robotic Manipulation,” in *CoRL*, 2024.
- [23] Y. J. Ma *et al.*, “Eureka: Human-Level Reward Design via Coding Large Language Models,” in *ICLR*, 2024.
- [24] T. Xie *et al.*, “Text2Reward: Reward Shaping with Language Models for Reinforcement Learning,” in *ICLR*, 2024.
- [25] Y. Wang *et al.*, “RoboGen: Towards Unleashing Infinite Data for Automated Robot Learning via Generative Simulation,” in *ICML*, 2024.
- [26] R. Li *et al.*, “RoboClaw: An Agentic Framework for Scalable Long-Horizon Robotic Tasks,” *arXiv:2603.11558*, 2026.
- [27] G. Wang *et al.*, “Voyager: An Open-Ended Embodied Agent with Large Language Models,” *TMLR*, 2024.
- [28] J. Zhang *et al.*, “Playful Agentic Robot Learning,” *arXiv:2606.19419*, 2026.
- [29] E. Johns, “Coarse-to-Fine Imitation Learning: Robot Manipulation from a Single Demonstration,” in *ICRA*, 2021.
- [30] B. Wen, W. Lian, K. Bekris, and S. Schaal, “You Only Demonstrate Once: Category-Level Manipulation from Single Visual Demonstration,” in *RSS*, 2022.
- [31] A. Simeonov *et al.*, “Neural Descriptor Fields: SE(3)-Equivariant Object Representations for Manipulation,” in *ICRA*, 2022.
- [32] W. Shen *et al.*, “Distilled Feature Fields Enable Few-Shot Language-Guided Manipulation,” in *CoRL*, 2023.
- [33] J. Zhu *et al.*, “Densematcher: Learning 3d semantic correspondence for category-level manipulation from a single demo,” in *ICLR*, 2025.
- [34] O. Biza *et al.*, “One-shot Imitation Learning via Interaction Warping,” in *CoRL*, 2023.
- [35] Y. Liu, J. Mao, J. B. Tenenbaum, T. Lozano-Pérez, and L. P. Kaelbling, “One-Shot Manipulation Strategy Learning by Making Contact Analogies,” in *ICRA*, 2025.
- [36] M. Sieb, Z. Xian, A. Huang, O. Kroemer, and K. Fragkiadaki, “Graph-Structured Visual Imitation,” in *CoRL*, 2019.
- [37] Y. Zhu, A. Lim, P. Stone, and Y. Zhu, “Vision-Based Manipulation from Single Human Video with Open-World Object Graphs,” *Autonomous Robots*, vol. 50, no. 2, p. 27, 2026.
- [38] Z. Zhang *et al.*, “Universal Visual Decomposer: Long-Horizon Manipulation Made Easy,” in *ICRA*, 2024.
- [39] T. Gao, S. Nasiriany, H. Liu, Q. Yang, and Y. Zhu, “PRIME: Scaffolding Manipulation Tasks with Behavior Primitives for Data-Efficient Imitation Learning,” *RA-L*, vol. 9, no. 10, pp. 8322–8329, 2024.
- [40] A. Wu, R. Wang, S. Chen, C. Eppner, and C. K. Liu, “One-shot transfer of long-horizon extrinsic manipulation through contact retargeting,” in *IROS*, 2024.
- [41] W. Liu, N. Nie, R. Zhang, J. Mao, and J. Wu, “Learning Compositional Behaviors from Demonstration and Language,” in *CoRL*, 2024.
- [42] S. Cheng, C. R. Garrett, A. Mandlekar, and D. Xu, “NOD-TAMP: Generalizable Long-Horizon Planning with Neural Object Descriptors,” in *CoRL*, 2024.
- [43] N. Nie *et al.*, “Learning Composable Skills by Discovering Spatial and Temporal Structure with Foundation Models,” in *ICRA*, 2026.
- [44] B. Zandonati, T. Lozano-Pérez, and L. P. Kaelbling, “Rational Inverse Reasoning: Few-Shot Imitation by Inferring Intent through Planning,” *arXiv:2508.08983*, 2025.
- [45] N. Chavan-Dafle *et al.*, “Extrinsic Dexterity: In-Hand Manipulation with External Forces,” in *ICRA*, 2014.
- [46] K. M. Lynch and M. T. Mason, “Stable Pushing: Mechanics, Controllability, and Planning,” *IJRR*, vol. 15, no. 6, pp. 533–556, 1996.
- [47] G. Lee, T. Lozano-Pérez, and L. P. Kaelbling, “Hierarchical Planning for Multi-Contact Non-Prehensile Manipulation,” in *IROS*, 2015.
- [48] J.-P. Sleiman, J. Carius, R. Grandia, M. Wermelinger, and M. Hutter, “Contact-Implicit Trajectory Optimization for Dynamic Object Manipulation,” in *IROS*, 2019.
- [49] B. Aceituno-Cabezas and A. Rodriguez, “A Global Quasi-Dynamic Model for Contact-Trajectory Optimization in Manipulation,” in *RSS*, 2020.
- [50] T. Pang, H. J. T. Suh, L. Yang, and R. Tedrake, “Global Planning for Contact-Rich Manipulation via Local Smoothing of Quasi-Dynamic Contact Models,” *T-RO*, 2023.
- [51] E. Huang, X. Cheng, Y. Mao, A. Gupta, and M. T. Mason, “Autogenerated Manipulation Primitives,” *IJRR*, 2023.
- [52] W. Zhou and D. Held, “Learning to Grasp the Ungraspable with Emergent Extrinsic Dexterity,” in *CoRL*, 2022.
- [53] W. Zhou, B. Jiang, F. Yang, C. Paxton, and D. Held, “HACMan: Learning Hybrid Actor-Critic Maps for 6D Non-Prehensile Manipulation,” in *CoRL*, 2023, pp. 241–265.
- [54] Y. Cho, J. Han, Y. Cho, and B. Kim, “CORN: Contact-based Object Representation for Nonprehensile Manipulation of General Unseen Objects,” in *ICLR*, 2024.
- [55] S. Nasiriany, H. Liu, and Y. Zhu, “Augmenting Reinforcement Learning with Behavior Primitives for Diverse Manipulation Tasks,” in *ICRA*, 2022.
- [56] P.-T. de Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein, “A Tutorial on the Cross-Entropy Method,” *Annals of Operations Research*, vol. 134, no. 1, pp. 19–67, 2005.
- [57] N. Carion *et al.*, “SAM 3: Segment Anything with Concepts,” in *ICLR*, 2026.
- [58] X. Chen *et al.*, “SAM 3D: 3Dfy Anything in Images,” in *CVPR*, 2026.
- [59] B. Wen, W. Yang, J. Kautz, and S. Birchfield, “FoundationPose: Unified 6D Pose Estimation and Tracking of Novel Objects,” in *CVPR*, 2024.
- [60] E. Todorov, T. Erez, and Y. Tassa, “MuJoCo: A Physics Engine for Model-Based Control,” in *IROS*, 2012.
- [61] O. Khatib, “A Unified Approach for Motion and Force Control of Robot Manipulators: The Operational Space Formulation,” *IEEE Journal on Robotics and Automation*, vol. 3, no. 1, pp. 43–53, 1987.
- [62] B. Liu *et al.*, “LIBERO: Benchmarking Knowledge Transfer for Lifelong Robot Learning,” in *NeurIPS*, 2023.
- [63] M. J. Kim *et al.*, “OpenVLA: An Open-Source Vision-Language-Action Model,” in *CoRL*, 2024.
- [64] K. Black *et al.*, “ π_0 : A Vision-Language-Action Flow Model for General Robot Control,” in *RSS*, 2025.

- [65] Physical Intelligence *et al.*, “ $\pi_{0.5}$: A Vision-Language-Action Model with Open-World Generalization,” *arXiv:2504.16054*, 2025.
- [66] NVIDIA, “Newton Physics Engine,” 2025, website: <https://developer.nvidia.com/newton-physics>.