

Rational Inverse Reasoning: Few-Shot Imitation by Inferring Programmatic Intent through Planning

Anonymous Authors

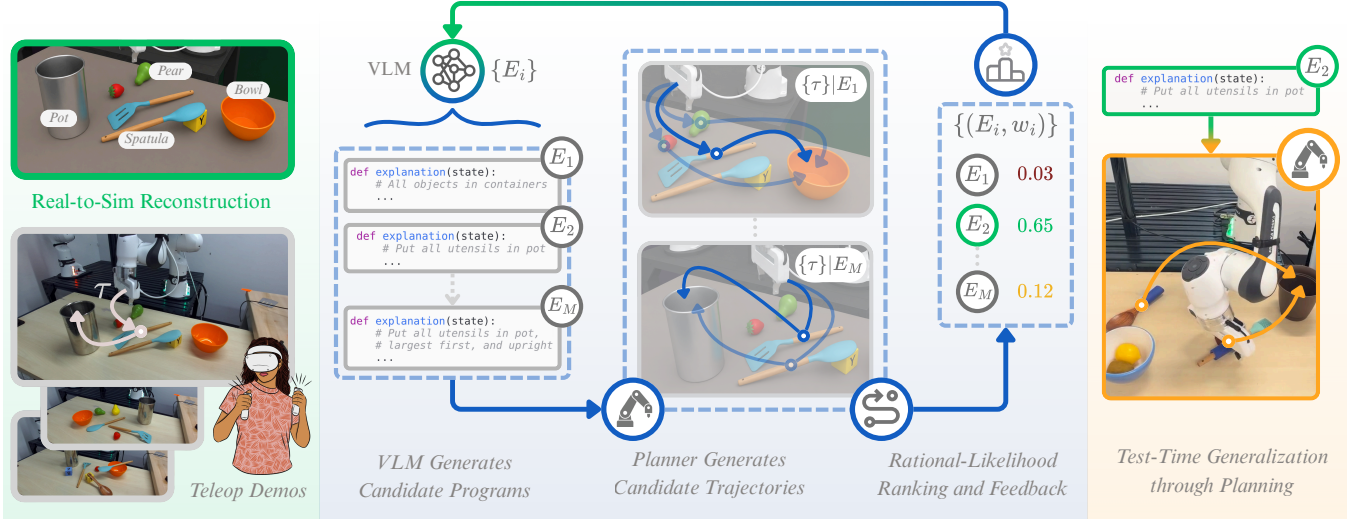


Fig. 1: Rational Inverse Reasoning. Demonstrations and object-centric scene states condition VLM proposals of explanation programs. Each program compiles to a planning specification; TAMP supplies demonstration-consistent and counterfactual skeletons and refinement evidence for the rational likelihood score and feedback. Scores and planner diagnostics guide subsequent agentic proposals in a finite program pool. At deployment, the selected program re-grounds the inferred intent in a new scene, and TAMP generates a new robot trajectory.

Abstract—Humans learn a manipulation task from one or two demonstrations, and perform it in a new room, with new objects, under new constraints. This requires identifying which object roles, subgoal orderings, and constraints should persist when the scene changes, as opposed to cloning object motions. We present Rational Inverse Reasoning (RIR), which infers this task structure as an explanation program: a compact, executable description of intent that maps an object-centric scene to a task-and-motion planning (TAMP) specification of goals, subgoals and constraints. This allows the robot to generate new actions for new objects and layouts, while preserving the demonstrator’s intent. A vision-language model proposes candidate programs, a planner-derived score evaluates whether each candidate makes the demonstrated behavior rational, relative to alternative strategies. These scores guide iterative program refinement. We evaluate RIR on 30 simulated, and four real-world tabletop tasks (using a Franka FR3), and 35 two-dimensional reasoning tasks. In all cases, RIR recovers transferable task structure from only 1-3 demonstrations. Generalizing to new layouts and object sets, it outperforms VLM, planning and few-shot learning baselines that lack explicit rationality and planning-grounded inference. See our interactive supplementary material and videos: <https://rir-project-f3a281.pages.dev>

I. INTRODUCTION

Few-shot imitation learning, where a robot must learn a manipulation task from a small number (1-3) of demonstrations, remains a central challenge [1], [2]. In this setting,

approaches which memorize or clone the observed trajectory distribution are most brittle, degrading under minor variations in layout, a different object set, added clutter, or changed geometric constraints [3]. By comparison, humans constantly learn and generalize from a handful of imperfect, under-specified and context-dependent demonstrations [4]. We argue that this gap arises due to a lack of abstraction.

A robot learning a new manipulation task from very few demonstrations must instead reason about which aspects of behavior should be carried over to a new scene. An agent may observe the actions taken in a particular scene, but not directly the *intent* that drives them. The same task can give rise to many different action sequences depending on the objects, layout, and constraints of the environment. To learn from only a handful of demonstrations therefore, a robot must instead infer this underlying intent: which objects were relevant, what intermediate objectives were being pursued, what ordering relationships mattered, and which constraints were essential. Achieving real generalization doesn’t come from replaying past behavior, but from recovering this abstracted representation and using it to synthesize behavior for a new scene. This shifts the problem away from learning a mapping from states to actions, toward the inference of a structured and transferable representation of task intent.

To this end, we introduce Rational Inverse Reasoning (RIR), which casts few-shot imitation as the inference over

explanation programs: compact, executable functions that map an object-centric scene to a task and motion planning (TAMP) specification of goals, ordered subgoals, and continuous constraints that describe the task intent (Fig. 1). Programmatic representations have proven effective for connecting language instructions to robot execution [5], [6], [7], [8]; we infer them from demonstrations instead of language. Each program compiles to a planning problem: in a new scene, objects and constraints are re-grounded. Then, a TAMP system plans and executes behaviors that may differ substantially from the demonstrations, while preserving their intent. Following inverse reinforcement learning and inverse planning [9], [10], [11], which infer the objective that best explains an agent’s actions under the assumption of rationality, RIR seeks a program which *rationalizes* the demonstrator’s behavior.

Three obstacles make this inference hard. First, the program space is far larger than the symbolic goals or reward functions of prior work [12], [13], [14]. Second, many programs explain the same demonstration, as shown in Fig. 1, and could comprise trajectories, object placements or objectives, so a prior is needed for disambiguation. Third, computing the exact likelihood is intractable, since scoring a candidate program requires solving many long-horizon, continuous, geometrically constrained planning problems. RIR addresses these problems by incorporating a hierarchical TAMP system within the loop of a vision-language model (VLM). The VLM proposes a pool of candidate programs that could produce the demonstrations. Each program is then scored with a GPU-accelerated surrogate for the inverse-planning likelihood, which measures whether a program makes the demonstration rational, consistent, and specific enough to be likely relative to plausible alternatives. These scores then condition the VLM’s next proposals, defining an iterative heuristic learning approach which improves the set of programs without exhaustive search. Through this agentic loop, RIR infers explanation programs describing transferable intent.

We evaluate RIR on a wide array of challenging long-horizon task settings, spanning simulated and real tabletop manipulation tasks using a Franka FR3 robot arm, comparing against few-shot imitation approaches, as well as on 2D reasoning tasks, comparing against human sample efficiency. In all these domains, RIR achieves few-shot generalization to novel objects, layouts, and constraints.

Contributions: (1) We formulate few-shot learning from demonstration as inference over executable explanation programs: scene-conditioned representations of task intent that compile to planning specifications and support interpretation, re-grounding, and re-planning. (2) We introduce a GPU-accelerated implementation of an efficient planner-grounded algorithm for computing the rationality score in challenging continuous robotics problems. (3) We present a VLM-planner inference loop in which the VLM proposes open-ended program hypotheses and the planner-based likelihood guides their refinement from only 1–3 demonstrations.

II. RELATED WORK

Few-Shot Imitation and Program Induction. In the robotics community, approaches to learning from a small number of examples focus on learning policies in a representation that has a very strong structural bias. Object-centric representations, such as ORION’s manipulation plans [15], support transfer by capturing object relationships and interaction structure. Program induction instead represents behavior through executable abstractions learned from examples [16]. PROLEX synthesizes robot programs in a constrained symbolic language [6], while Demo2Code uses hierarchical summarization to derive specifications and code from demonstrations and task descriptions [7]. RIR makes a stronger commitment, representing a policy indirectly as a program that produces a *planning* objective, run with a TAMP planning solver [17].

Inferring Objectives From Behavior. Inverse reinforcement learning recovers reward functions under which demonstrated behavior is approximately optimal [9], [10]. Bayesian inverse planning models observations as the output of a rational or boundedly rational agent, then inverts this model to infer latent objectives [4], [11]. Plan recognition as planning evaluates candidate goals through planning costs [12], [13], while open-ended approaches extend inference beyond a fixed hypothesis set [14]. These methods share our use of model-based behavioral evidence. RIR applies inverse planning to programs that generate structured manipulation objectives, evaluating both discrete strategy choices and continuous refinement evidence.

Foundation Models for Planning and Control. Foundation models generate code policies, skill sequences, and symbolic plans that are grounded through affordances, value functions, or execution feedback [5], [18], [19]. PRoC3S integrates language-model proposals with constraint satisfaction [8], and APRICOT combines preference inference, active clarification, and constraint-aware planning [20]. Vision-language-action models provide a complementary execution interface, mapping language and visual observations to robot actions [21]. In RIR, the VLM proposes explanation programs that compute planning objectives rather than actions. The planning system then generates the actions, thus exploiting the VLM’s strength in common-sense understanding while relying on model-based planning for geometric reasoning.

III. PROBLEM SETTING

In this section, we outline the problem setting, defining our inference target, *explanation programs*, and the robot’s model-based perception and planning capacity, TAMP [17].

At train time, the system is given a (small) set of demonstrations $D = \{(\tau^{(k)}, x_0^{(k)})\}_{k=1}^K$, where each demonstration $\tau^{(k)} = (x_0^{(k)}, \dots, x_T^{(k)})$ consists of raw observations x_t : For teleop demonstrations, this is RGBD camera views and robot joint configurations, and for VR demonstrations in simulation, the hand positions and object states are provided. Demonstrations may have distinct initial scenes (layouts, objects, clutter), but all share an unknown intent that can be

represented as a single explanation program E^* . At test time, the robot is presented with a novel scene observation x'_0 , and uses the inferred intent $\hat{E}$ to produce an action sequence whose induced trajectory $\hat{\tau}$ satisfies the demonstrator's intent.

We assume access to a perception system ϕ which produces an object-centric representation $s \in \mathcal{S}$ of demonstration states $\phi : \mathcal{X} \rightarrow \mathcal{S}$ where $s = (O, R)$. Here, $O = \{o_i\}_{i=1}^n$ are the objects/entities with attributes (e.g. type, semantic features, keypoints, 6D pose, mesh geometry), and R is a set of spatial properties and relations on objects.

An explanation program $E : \mathcal{S} \rightarrow \mathcal{G}$ is a Python function over the scene representation s which constructs a task specification $g \in \mathcal{G}$ from

$$g ::= \text{Ach}(\Phi) \mid \text{Seq}(g_1, \dots, g_m) \mid \text{Constrain}(c, g).$$

$\text{Ach}(\Phi)$ requires a conjunction of grounded predicates. $\text{Seq}(g_1, \dots, g_m)$ requires consecutive trajectory segments satisfying subgoals in order. $\text{Constrain}(c, g)$ additionally maintains c throughout the segment. The explanation program uses the scene description s to construct task specifications, flexibly grounding and combining spatial and semantic goals, filtering objects by geometric features, or branching on scene properties. For example, the explanation program might include placing utensils upright, by defining a more general 'upright' constraint based on keypoint relations.

At deployment, a generalized Task-and-Motion-Planning (TAMP) algorithm is used to produce action sequences from a task specification g' grounded on the initial scene representation $g' = \hat{E}(s'_0)$. The system finds a symbolic plan skeleton (the high-level *strategy*) π : an ordered sequence of high-level operations (picking, moving, placing) grounded in the objects in the scene. This plan skeleton π is then refined into a continuous environment trajectory $\hat{\tau}$ satisfying g' (see Appendices B and C). During inference, TAMP is treated as a hierarchical conditional generative model: $(\hat{\pi}, \tau) \sim \text{TAMP}(g, s_0)$ that we approximately invert in Sec. IV; it provides the inverse planning score that drives program refinement.

IV. RATIONAL INVERSE REASONING

Rational Inverse Reasoning (RIR) casts few-shot learning as the inference of explanation programs E from demonstration data D , by inverting a near-optimal embodied reasoning procedure for inferring and executing robot motions. The ideal inference objective is

$$p(E \mid D) \propto p(E) \prod_{k=1}^K p(\tau^{(k)} \mid x_0^{(k)}, E). \quad (1)$$

RIR approximates this objective by devising a tractable likelihood score $p(D \mid E)$, measuring how well a candidate E rationalizes D . We achieve this by extending the notion of *bounded rationality* [10] to our hierarchical generative model of physical reasoning. This informs iterative refinement of a pool of VLM program proposals. The following sections develop a tractable approximation to the likelihood term in

Eq. (1), and Sec. IV-C describes the adaptive program search used to approximate the resulting posterior over explanation programs.

A. Boltzmann All The Way Down:

The Hierarchical Generative Model of Behavior

For a given grounded specification g and initial scene state s_0 , the TAMP system selects a symbolic plan skeleton $\hat{\pi} \in \Pi(g, s_0)$, where $\Pi(g, s_0)$ is the (typically infinite) set of feasible plans for g and initial state s_0 . Then, the planner samples continuous refinement parameters such as grasps, placements, configurations and motion plans to generate a trajectory $\tilde{\tau}$. The behavioral likelihood marginalizes out the unknown plan $\hat{\pi}$:

$$p(\tau \mid x_0, E) = \sum_{\hat{\pi} \in \Pi(g, s_0)} p_{\text{traj}}(\tau \mid \hat{\pi}, g, s_0) p_{\text{plan}}(\hat{\pi} \mid g, s_0).$$

Here, p_{traj} captures execution rationality: the likelihood of τ as a continuous refinement of a skeleton $\hat{\pi}$. The factor p_{plan} captures the strategy rationality: the probability of selecting the plan $\hat{\pi}$ for a specification g and initial state s_0 .

Execution Rationality: Let $q(\tau \mid \hat{\pi}, g, s_0)$ be the planner's normalized proposal distribution over complete refinements of a skeleton into trajectory τ . Let $\mathbf{1}(\tau; \hat{\pi}, g) \in \{0, 1\}$ indicate that a trajectory is feasible and valid for $\hat{\pi}$, satisfying g , and let $C(\tau)$ be its domain-specific effort cost. The execution model is Boltzmann on skeleton refinement,

$$p_{\text{traj}}(\tau \mid \hat{\pi}, g, s_0) = \frac{q(\tau \mid \hat{\pi}, g, s_0) e^{-\beta_t C(\tau)} \mathbf{1}(\tau; \hat{\pi}, g)}{Z_\pi(\hat{\pi}, g, s_0)}, \quad (2)$$

where $Z_\pi = \mathbb{E}_q[e^{-\beta_t C(\tilde{\tau})} \mathbf{1}(\tilde{\tau}; \pi, g)]$. Its free energy $F_\pi = -\beta_t^{-1} \log Z_\pi$ summarizes both cost and the available probability mass of feasible refinements.

Strategy Rationality: We model the demonstrator's choice of plan skeleton $\hat{\pi}$ (*strategy*) according to,

$$p_{\text{plan}}(\pi \mid g, s) = \frac{e^{-\beta_p F_\pi}}{\sum_{\pi' \in \Pi(g, s)} e^{-\beta_p F_{\pi'}}}. \quad (3)$$

The two temperatures have distinct roles: β_t (traj) controls trajectory variability within a plan, while β_p (plan) controls how optimally (given a goal) strategies are selected. To evaluate a skeleton, we use free energy F_π , rather than the cost of its cheapest refinement $C_{\min}$, modeling a human demonstrator's tendency to prefer strategies that are robustly executable, rather than those with one fortuitous refinement.

B. A Tractable Planner-Grounded Likelihood

Our goal is to evaluate $p(\tau \mid x_0, E)$ (Sec. IV-A) for demonstration trajectories across a pool of candidate explanation programs E . However, the likelihoods defined in Sec. IV-A are not directly computable: the planner cannot enumerate all skeletons or integrate over all continuous refinements. RIR replaces it with a finite surrogate that preserves three sources of evidence: (i) compatibility of the task and plan with the demonstration, (ii) rationality of the demonstrated (observed) plan relative to counterfactuals, and (iii) the cost

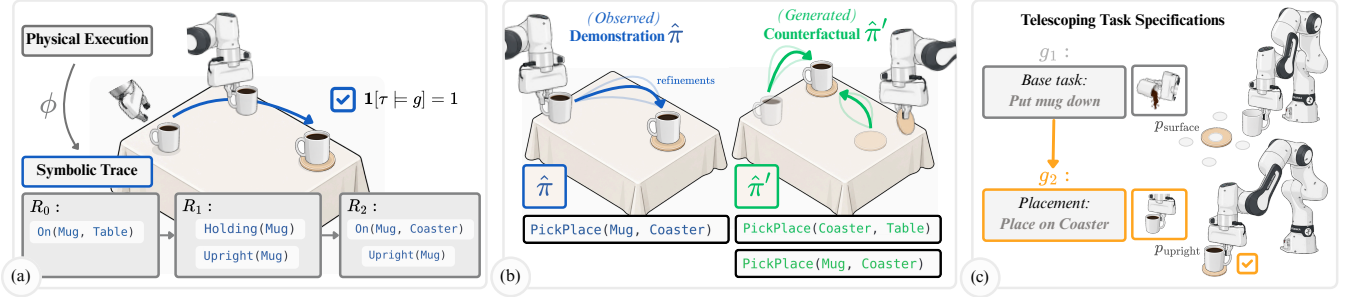


Fig. 3: Illustrations of aspects of RIR. (a) Observed demonstration is segmented according to semantic mode changes. (b) The demonstrated skeleton is evaluated relative to a planner-generated counterfactual. (c) For a fixed skeleton, specificity is estimated from the cost-weighted fraction of refinements satisfying each stronger specification.

and specificity of the observed execution. Each of these aspects are illustrated in Fig. 3.

Demonstrated and Counterfactual Strategies: For an observed trajectory, the indicator $\mathbf{1}(\tau; \hat{\pi}, g)$ tests whether the demonstration τ satisfies g , and can be segmented as $\hat{\pi}$. This defines the support for any plan-conditional likelihood contribution. Applying our perception ϕ to the demo trajectory lifts it to symbols R_t (predicate relations at time t). This allows us to compute $\mathbf{1}(\tau; \hat{\pi}, g)$ efficiently, via the planning domain theory and task specification, using dynamic programming, akin to LTL segmentation approaches [22], [23]. The result is a set of *bottom-up* plan proposals $\hat{\Pi}_{\text{bu}}(E)$ which have non-zero support in the generative model associated with the task specification g and plan skeleton $\hat{\pi}$. An explanation program which fails this symbolic check receives score $-\infty$.

Separately, for a candidate explanation program E (and corresponding task specification), we can use the TAMP system to sample plan skeletons using task-planning search. This supplies an alternative *top-down* set of counterfactual strategies $\hat{\Pi}_{\text{td}}$. These *counterfactual strategies* are essential: A demonstrated plan is informative only relative to what else the demonstrator could rationally have done under the same objective. Concretely, under the assumption of a rational agent, a detour is evidence for an intended subgoal only when a more direct plan would otherwise have been optimal [4], [13]. The union: $\hat{\Pi}(E) = \hat{\Pi}_{\text{bu}}(E) \cup \hat{\Pi}_{\text{td}}(E)$ approximates the strategy set in Eq. (3).

For each skeleton, a set of N proposal refinements gives

$$\hat{Z}_{\pi} = \frac{1}{N} \sum_{j=1}^N e^{-\beta_t C(\tilde{\tau}_j)} \mathbf{1}(\tilde{\tau}_j; \pi, g), \quad \hat{F}_{\pi} = -\frac{\log \hat{Z}_{\pi}}{\beta_t}, \quad (4)$$

a Monte-Carlo estimate relying on the TAMP proposal distribution. Invalid samples have zero weight and computation uses stable log-sum-exp, safely handling finite sample support. The use of free energy rather than the minimum sampled refinement cost $C_{\min} = \inf_j C(\tilde{\tau}_j)$ also makes the estimator more robust to stochasticity introduced by resource-bounded planning.

Relative Specificity from Nested TAMP Proposals Feasibility alone also favors weak explanations. *Place the mug on the coaster* and *place the mug [on the coaster] upright* can both match a demonstration, yet the latter explains an

orientation that the former leaves unconstrained. A planner conditioned on a constraint preferentially generates executions that satisfy it. Its samples alone therefore do not show how restrictive that constraint is.

We correct for this bias by comparing refinement distributions *across* task specifications from the pool, where the reference proposal g_0 is chosen adaptively. Specifically, we construct a chain of task specifications $g_0 \preceq g_1 \preceq \dots \preceq g_L = g$ where each backward step weakens ordering relations in Seq, or drops Constrain or Achieve conditions. The desired correction can then be decomposed into local comparisons, where refinements drawn under g_l are used to estimate the cost-weighted fraction satisfying g_{l+1} :

$$\tilde{r}_{\ell, \ell+1}(\pi) = \frac{\sum_j e^{-\beta_t C(\tilde{\tau}_{\ell j})} \mathbf{1}(\tilde{\tau}_{\ell j}; \pi, g_{\ell+1})}{\sum_j e^{-\beta_t C(\tilde{\tau}_{\ell j})} \mathbf{1}(\tilde{\tau}_{\ell j}; \pi, g_{\ell})}. \quad (5)$$

The specificity term is

$$\tilde{\Delta}_{\text{spec}}(\pi, g) = -\sum_{\ell=0}^{L-1} \log \tilde{r}_{\ell, \ell+1}(\pi). \quad (6)$$

An added condition contributes strongly when it excludes most plausible executions, provided the observed demonstration satisfies it, and weakly when it adds little behavioral information. Because the planning proposals depend on the specification and have unavailable densities, these ratios in Eq. (5) provide a planner-dependent approximation, rather than an exact bridge-sampling estimator [24].

Efficient Search and Planning: We implement a GPU-accelerated planner for the generalized TAMP problems defined by our task specifications (Sec. III). Inference requires solving hundreds of TAMP problems, making sequential evaluation expensive. We parallelize grasp sampling, inverse kinematics, collision checks, and constraint evaluation on the GPU [25], [26], enabling batched evaluation of candidate refinements. This particle-based approach [25] supports the Monte Carlo estimates required for inference.

Score Assembly For $g = E(s_0)$, compute $\hat{p}_{\text{plan}}$ from Eq. (3) using the free energies in Eq. (4) and the candidate set $\hat{\Pi}(E)$. The demonstration score is

$$\begin{aligned} \hat{\ell}_{\text{traj}, \pi} &= -\beta_t C(\tau) - \log \hat{Z}_{0, \pi}^{(0)} + \tilde{\Delta}_{\text{spec}}(\pi, g), \\ \hat{\ell}(E) &= \text{LSE}_{\pi \in \hat{\Pi}_{\text{bu}}(E)} [\log \hat{p}_{\text{plan}}(\pi | g, s_0) + \hat{\ell}_{\text{traj}, \pi}]. \end{aligned} \quad (7)$$

Algorithm 1: Planner-grounded demonstration score

Input: τ, s_0, E ; common reference g_0 ; sample budget N
 $g \leftarrow E(s_0)$; $\hat{\Pi}_{bu} \leftarrow \text{BOTTOMUP}(\tau, g)$
if $\hat{\Pi}_{bu} = \emptyset$ **then**
 return $-\infty$
end
 $\hat{\Pi} \leftarrow \hat{\Pi}_{bu} \cup \text{TAMPPLANS}(g, s_0)$
 $(g_0, \dots, g_L = g) \leftarrow \text{BRIDGESPECS}(g_0, g)$
foreach $\pi \in \hat{\Pi}$ **do**
 $B_{\ell, \pi} \leftarrow \text{REFINE}(s_0, \pi, g_\ell, N)$, $\ell = 0, \dots, L$
 Estimate $\hat{F}_\pi$ at g and $\hat{Z}_{0, \pi}^{(0)}$ at g_0 using Eq. (4)
 Compute $\hat{\Delta}_{\text{spec}}(\pi, g)$ using Eqs. (5)–(6)
end
 $\hat{p}_{\text{plan}} \leftarrow \text{softmax}_{\pi \in \hat{\Pi}}(-\beta_p \hat{F}_\pi)$
return $\hat{\ell}(E)$ from Eq. (7)

The reference estimate $\hat{Z}_{0, \pi}^{(0)}$ is computed at g_0 using Eq. (4). It remains inside the sum because it can depend on the skeleton. We omit the unknown refinement density $q(\tau \mid \pi, g, s_0)$, assuming that motion detail beyond the symbolic trace and effort cost (i.e. what we can explain given the program-planning domain) adds no evidence about intent. The result is a planner-dependent surrogate rather than an exact likelihood of the raw trajectory. We sum log scores over demonstrations. Alg. 1 summarizes the computation (see Appendix A).

C. Iterative Program Heuristic Reasoning

The surrogate score in Alg. 1 makes it possible to score a candidate explanation, but the hypothesis space $\mathcal{E}$ consists of open-ended Python programs and cannot be enumerated. Instead, RIR uses a VLM as an adaptive proposal mechanism and maintains a finite pool of weighted candidate programs $\mathcal{H} = \{(E_i, w_i)\}_{i=1}^M$. The VLM contributes a strong semantic prior over plausible human objectives and useful program structure; the planner-grounded score determines which of those proposals are supported by the demonstrations.

RIR then runs an iterative agentic particle search. The VLM is used to propose the initial M candidates. This first round ($t = 1$) is primarily top-down; the VLM proposes broad explanations conditioned on context containing the demonstrations, scene reconstructions, task API and in-context examples $E \sim \text{VLM}(E \mid \text{context}_t)$. Each candidate is scored by the surrogate likelihood, and the weights w_i are normalized over the candidate set $\mathcal{H}$. In the following rounds we add the scored program set $\mathcal{H}_t$ to the VLM context, along with planner diagnostics (which constraints or subgoals failed, whether its plan strategy score is low, whether its specificity ratio is near 1). The VLM is then tasked with re-proposing a refined candidate set $\mathcal{H}_{t+1}$, with the aim of preserving supported structure, while repairing specific sources of surprise: erroneous programs, low-weight hypotheses and sub-goal or constraint failures. High-weight particles are retained across rounds, and the remaining slots are filled with revisions and diverse new proposals. After the final round, we take $p(E)$ uniform over the pool, so the reported program is the max-likelihood (or the top-K) candidate in $\mathcal{H}_t$. To capture the prior, we introduce an occam-

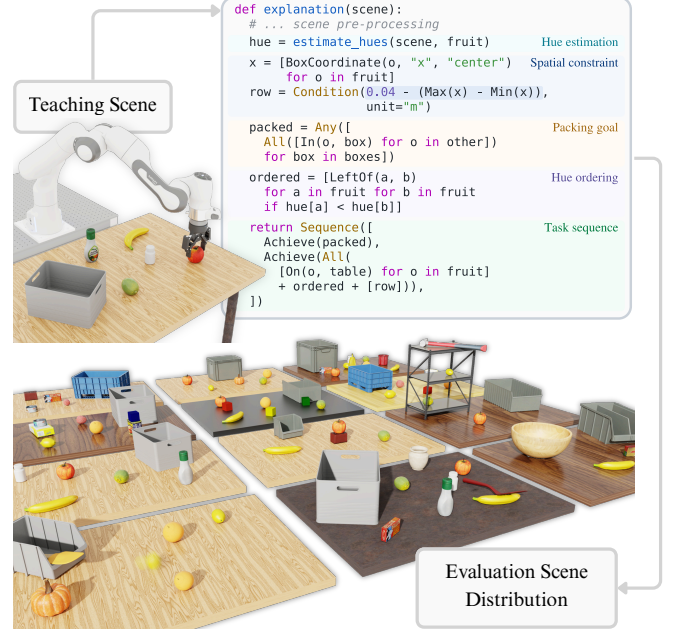


Fig. 4: Example of the teaching and evaluation scenes, as well as a simplified explanation program for the task: *place all the other clutter in the box, then line up the fruits by hue*. The learner must infer this complex task intent, and execute it in unseen scenes.

style bias *in-context*. We found doing this, coupled with the finite vocabulary and task api, provided better outcomes than a simple MDL program prior [16].

V. EXPERIMENTS

We organize the evaluation around four questions: **RQ1:** Does planner-grounded rational feedback improve intent inference and downstream success? **RQ2:** Does the inferred intent transfer across changes in objects, layouts, clutter, and geometry, and how does this depend on the number and diversity of demonstrations? **RQ3:** Where do failures arise during intent inference, planning, and physical execution? **RQ4:** How sample-efficient is RIR relative to human intent inference? We study RQ1–RQ2 at scale on 30 simulated tabletop tasks, RQ3 in simulation and on a real Franka FR3, and RQ4 on a 35-task two-dimensional reasoning corpus with human responses.

A. Evaluation metrics

We evaluate RIR at three levels. **Comprehension** (human estimated program equivalence) measures whether the inferred explanation program matches the demonstrator’s intent ($\hat{E} \equiv E^*$); it measures inference quality independent of any downstream planning. **Trajectory Success** measures whether the trajectory generated in a held-out scene satisfies the intended task under the planning/evaluation model. **Execution Success** measures task completion when the planned behavior is executed on the robot or rolled out with simulated dynamics. We report these outcomes separately because a program can be correct yet planned behavior wrong (an imperfect re-grounding). Similarly, the planned behavior can

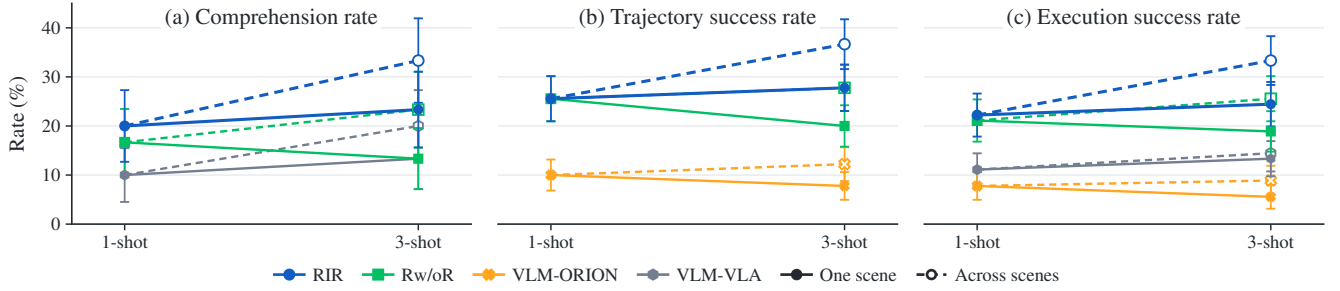


Fig. 5: Comprehension rate (a), trajectory success rate (b) and execution success rate (c) with one or three demonstrations from a single scene (low-diversity demo data) or across scenes (high-diversity demo data). Error bars indicate ± 1 standard error.

be correct, yet fail physically (a slipped grasp). Each *macro-task* (*place the mug on the shelf*) is evaluated over a large pool of [algorithm seed $\times$ scene variant] combinations.

B. Simulated Tabletop Manipulation

Task Distribution: We construct 30 tabletop manipulation tasks in IsaacSim for a Franka FR3 robot arm. Taking inspiration from RoboLab’s task format [27], we construct a more challenging task set requiring richer goal-subgoal-constraint reasoning. For example, the tasks include transporting liquid containers safely while avoiding the space above a laptop, sorting condiments before stacking the dishes and arranging objects on the table to spell the letter ‘A’. The task set spans more than 300 individual objects. Each macro-task has three training scenes, with three demonstrations per scene. Demonstrations are collected with VR controls and direct object manipulation. 3 held-out scenes per task provide 90 test scenes, spanning a generalization taxonomy of variation in objects, number, layout, and clutter, as illustrated in Fig. 4. We then evaluate the 1-shot and 3-shot settings for single-scene (low diversity demonstrations with basic layout variation across demos) and multi-scene inputs (full variation across demo scenes).

Comparisons: We compare RIR with baselines that probe the contributions of rational feedback, program induction, and the representation used for transfer. No method receives a natural-language description of the target task. Reasoning without Rationality (Rw/oR) uses the same VLM, inputs, and candidate and round budgets as RIR, but replaces planner-derived rational feedback with self-refinement, similar to Demo2Code-NoLang [7]. ORION transfers object-centric manipulation plans [15]. Our VLM-ORION variant uses the VLM to ground the learned ORION representation, broadening its generalization capabilities and enabling learning from multiple demonstrations. VLM-VLA instead uses natural-language intent as an expressive intermediate representation: given demonstrations in context, the VLM infers the intent, which a pretrained VLA (π -0.5 [28]) then executes. Rw/oR isolates the contribution of rational feedback, while VLM-ORION and VLM-VLA evaluate alternative inference procedures and representations for transfer. (see Appendix F)

Sim Results: As shown in Fig. 5, RIR achieves the highest intent comprehension and execution success across one- and three-shot setting. Specifically, RIR improves comprehension

by $1.4\times$ and overall success by $1.3\times$ as compared to the nearest baseline (Rw/oR) in the one-/three-shot settings, isolating the benefit of reasoning about why the demonstrated behavior was chosen relative to alternatives. Additional demonstrations reduce the ambiguity in the inferred intent, resulting in an increase in comprehension rate and subsequent execution success. In particular, we see that RIR is best able to make use of the higher-diversity demonstration data compared to VLM and object-centric approaches, as the planning system (and therefore the score computation) is able to generalize across scenes. Performance remains strong across novel objects, layouts, counts, and distractors, showing that the inferred program transfers task structure rather than demonstrated motion. Notably, we see that for low-diversity demonstration scenes, Rw/oR’s comprehension drops (shown by the solid green line in Fig. 5(a)). Self-refinement with 3 low-diversity demos tends to overfit the generated programs to the objects in those demos. RIR is able to counteract this, and with 3 high-diversity demonstrations (dashed line) its harder to formulate a program which overfits to high diversity demo data.

The average RIR inference time was 2m 57s and the average downstream planning time was 32s. However, these values are highly problem dependent making the range broad; simple pick-place tasks (*stack the blocks in any order*) solve in 4s and highly constrained problems (*Arrange the tools in an aligned row, with their ends facing away from you*) took up to 1m 27s.

C. Real-World Tabletop Manipulation

We evaluate RIR on a Franka FR3 in tabletop scenes.

Implementation: In real environments, we do not have access to the privileged simulation state, and must contend with partial observability and noise. As a result, we develop a real-to-sim pipeline that reconstructs the required sequence of object-centric states $\{s_t\}_t^T$ defined in Sec. III from raw observations in the demonstrations (RGBD frames and robot joint states; see Appendix D). Demonstrations of manipulation tasks (each 2-4 minutes long) are collected via a teleoperation system, where a human operator controls the end-effector pose in real-time using the Meta Quest VR headset. The scene reconstruction is used by the TAMP system, both at inference-time, to obtain an explanation

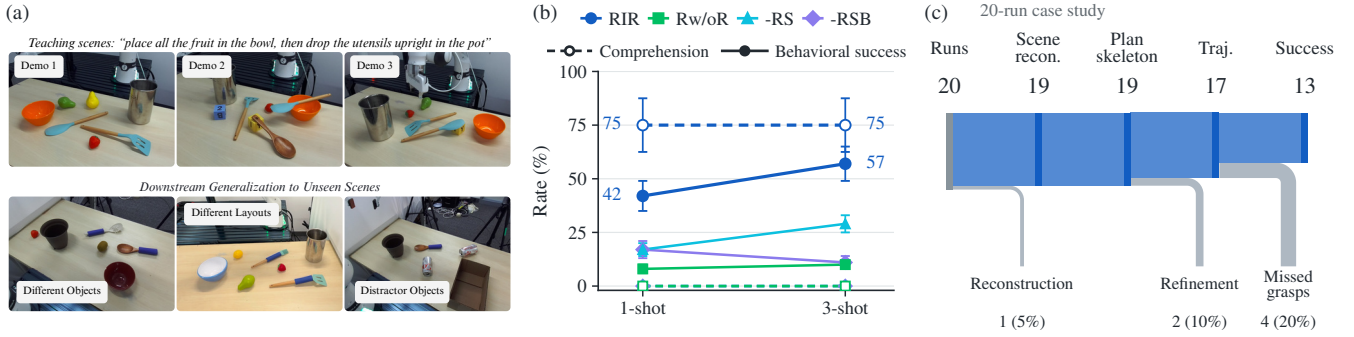


Fig. 6: Real robot evaluation. (a) Teaching scenes and transfer to new objects, layouts, and distractors. (b) Comprehension and trajectory (behavioral) success with one or three demonstrations; dashed and solid curves denote the two metrics, respectively. (c) A separate 20-run execution case study: 13 successes, one reconstruction failure, two refinement failures, and four missed grasps. Error bars indicate ± 1 standard error.

program E , and at execution-time, to execute E in a novel scene.

Task Distribution: We evaluate RIR system on a set of 4 long-horizon macro-tasks that have a large set of potentially plausible explanations. They involve complex sorting and tidying tasks, as illustrated in Fig. 6. Evaluating comprehension occurs across 3 trials with different demonstrations for each trial. Across trials, the system is presented with novel objects in unseen layouts, with additional random clutter/distractor objects (see Appendix G).

Real Robot Results: Fig. 6 demonstrates transfer from teleoperated demonstrations to physical scenes with new objects, layouts, and distractors. We compare RIR with cumulative ablations: Rw/oR (as before), -RS additionally removes scene-state inputs from the VLM context, and -RSB further removes bottom-up plan proposals. In Fig. 6(b), RIR achieves 75% comprehension with both one and three demonstrations, whereas the ablations recover no equivalent programs. With three demonstrations, RIR achieves 57% behavioral success on held-out scenes, compared with 10% for Rw/oR. Behavioral success increases from 42% to 57% despite unchanged aggregate comprehension, illustrating why the metrics are complementary: additional demonstrations can improve task-satisfying behavior without increasing the aggregate rate of exact intent recovery.

A separate 20-run execution case study involved tracing the execution of a learned explanation program on a novel scene. This resulted in 13 successes, one reconstruction failure, two trajectory-refinement failures, and four missed grasps (Fig. 6(c)). Missed grasps account for four of the seven failures, making execution reliability an important limitation. The current system executes plans open-loop; detecting and recovering from grasp errors is a concrete direction for improving physical task completion, especially under significant perceptual error.

D. 2D Reasoning Corpus

We introduce a suite of challenging 2D rearrangement tasks requiring fine-grained geometric and algorithmic reasoning, and inference through compositional task structure.

Task Distribution: The corpus contains 35 tasks: 25 spatial reasoning tasks and 10 additional tasks involving algorithmic structure such as filtering, counting, and conditionals. An illustration of one of the 2D tasks is shown in Fig. 7, requiring the agent to infer, from only 3 demonstrations, that the task is to *place the largest triangle block at the top-right, then place all other triangle blocks at the bottom-left*. Here, even with this simple demonstration, we must disambiguate between many possible hypotheses, and produce a rich Python explanation program that compiles into the correct sorting behavior. For each inferred program, evaluation uses three object sets with five layout initializations each. The human study collected responses from 24 participants. Each task elicited an explanation after one demonstration, and an update after three (see Appendix H).

2D Corpus Results In Fig. 7(right), we see that even human comprehension is imperfect, illustrating the ambiguity of these tasks. Top- k comprehension further shows that RIR places the correct intent among a small set of hypotheses. Here, we see the clear benefit to comprehension rate as we include our rational likelihood to guide program generation and refinement. This is especially stark in the spatial+algorithmic tasks, where in the Top-10 regime, we almost triple performance. Beyond comprehension, RIR improves Top-1 execution success over Rw/oR from 22.4% to 58.9% on spatial tasks and from 14.5% to 24.0% on spatial-plus-algorithmic tasks, each with three demonstrations. Although RIR’s Top-5 comprehension approaches human Top-1 performance, its lower Top-1 comprehension shows that proposing the intended explanation and ranking it first remain distinct challenges.

VI. CONCLUSION AND FUTURE WORK

We introduced Rational Inverse Reasoning, which treats demonstrations as evidence from a rational planner and recovers intent as a *compact, compositional explanation program*. Combining foundation-model proposals with planner-grounded scoring in TAMP, RIR produces interpretable, executable, and transferable explanations that support one- and few-shot generalization in continuous manipulation. We believe this lays groundwork for more generalizable and

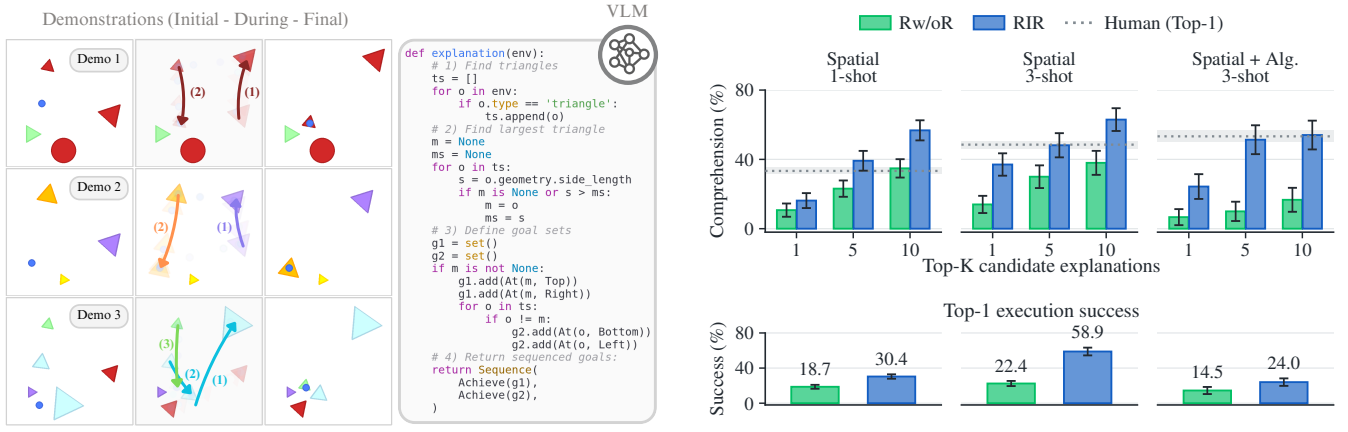


Fig. 7: Task structure and comprehension in the 2D reasoning tasks. (Left) The task of moving the largest triangle to the top-right before moving the remaining triangles to the bottom-left; the program API is abbreviated. (Right) Top- k comprehension for RIR and Rw/oR, with human top-1 comprehension as a reference (upper), and execution success of the top-ranked program (lower). Error bars indicate ± 1 standard error.

explainable imitation learning, where robots have the ability to learn from just a few examples by reasoning about the *why* behind observed behavior.

Future work will extend RIR to lifelong learning, building a library of manipulation programs and compressing their shared structure into the domain theory.

ACKNOWLEDGMENT

AI disclosure: Claude Code and Codex were used for code completion and debugging; all generated code was reviewed by the authors.

REFERENCES

- [1] T. Osa, J. Pajarinen, G. Neumann, J. A. Bagnell, P. Abbeel, and J. Peters, “An algorithmic perspective on imitation learning,” *Found. Trends Robot.*, vol. 7, no. 1–2, pp. 1–179, 2018.
- [2] C. Finn, T. Yu, T. Zhang, P. Abbeel, and S. Levine, “One-shot visual imitation learning via meta-learning,” in *Proc. Conf. Robot Learn. (CoRL)*, ser. PMLR, vol. 78, 2017, pp. 357–368.
- [3] P. de Haan, D. Jayaraman, and S. Levine, “Causal confusion in imitation learning,” in *Adv. Neural Inf. Process. Syst.*, vol. 32, 2019.
- [4] C. L. Baker, R. Saxe, and J. B. Tenenbaum, “Action understanding as inverse planning,” *Cognition*, vol. 113, no. 3, pp. 329–349, 2009.
- [5] J. Liang *et al.*, “Code as Policies: Language model programs for embodied control,” in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2023, pp. 9493–9500.
- [6] N. Patton, K. Rahmani, M. Missula, J. Biswas, and I. Dillig, “Programming-by-demonstration for long-horizon robot tasks,” *Proc. ACM Program. Lang.*, vol. 8, no. POPL, pp. 18:1–18:34, 2024.
- [7] Y. Wang, G. Gonzalez-Pumariega, Y. Sharma, and S. Choudhury, “Demo2Code: From summarizing demonstrations to synthesizing code via extended chain-of-thought,” in *Adv. Neural Inf. Process. Syst.*, vol. 36, 2023, pp. 14 848–14 956.
- [8] A. Curtis, N. Kumar, J. Cao, T. Lozano-Pérez, and L. P. Kaelbling, “Trust the PROCS: Solving long-horizon robotics problems with LLMs and constraint satisfaction,” in *Proc. Conf. Robot Learn. (CoRL)*, ser. PMLR, vol. 270, 2025, pp. 1362–1383.
- [9] A. Y. Ng and S. J. Russell, “Algorithms for inverse reinforcement learning,” in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2000, pp. 663–670.
- [10] B. D. Ziebart, A. Maas, J. A. Bagnell, and A. K. Dey, “Maximum entropy inverse reinforcement learning,” in *Proc. AAAI Conf. Artif. Intell.*, 2008, pp. 1433–1438.
- [11] T. Zhi-Xuan, J. L. Mann, T. Silver, J. B. Tenenbaum, and V. K. Mansinghka, “Online Bayesian goal inference for boundedly rational planning agents,” in *Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 19 238–19 250.
- [12] M. Ramírez and H. Geffner, “Plan recognition as planning,” in *Proc. Int. Joint Conf. Artif. Intell. (IJCAI)*, 2009, pp. 1778–1783.
- [13] —, “Probabilistic plan recognition using off-the-shelf classical planners,” in *Proc. AAAI Conf. Artif. Intell.*, 2010, pp. 1121–1126.
- [14] T. Zhi-Xuan, G. Kang, V. K. Mansinghka, and J. B. Tenenbaum, “Infinite ends from finite samples: Open-ended goal inference as top-down Bayesian filtering of bottom-up proposals,” in *Proc. Annu. Meeting Cogn. Sci. Soc.*, vol. 46, 2024.
- [15] Y. Zhu, A. Lim, P. Stone, and Y. Zhu, “Vision-based manipulation from single human video with open-world object graphs,” *Auton. Robots*, vol. 50, 2026, art. no. 27.
- [16] K. Ellis *et al.*, “DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning,” *Philos. Trans. R. Soc. A*, vol. 381, no. 2251, 2023, art. no. 20220050.
- [17] C. R. Garrett *et al.*, “Integrated task and motion planning,” *Annu. Rev. Control Robot. Auton. Syst.*, vol. 4, pp. 265–293, 2021.
- [18] B. Ichter *et al.*, “Do as I can, not as I say: Grounding language in robotic affordances,” in *Proc. Conf. Robot Learn. (CoRL)*, ser. PMLR, vol. 205, 2023, pp. 287–318.
- [19] W. Huang *et al.*, “Inner Monologue: Embodied reasoning through planning with language models,” in *Proc. Conf. Robot Learn. (CoRL)*, ser. PMLR, vol. 205, 2023, pp. 1769–1782.
- [20] H. Wang *et al.*, “APRICOT: Active preference learning and constraint-aware task planning with LLMs,” in *Proc. Conf. Robot Learn. (CoRL)*, ser. PMLR, vol. 270, 2025, pp. 1590–1642.
- [21] M. J. Kim *et al.*, “OpenVLA: An open-source vision-language-action model,” in *Proc. Conf. Robot Learn. (CoRL)*, ser. PMLR, vol. 270, 2025, pp. 2679–2713.
- [22] Y. Wang, N. Figueroa, S. Li, A. Shah, and J. Shah, “Temporal logic imitation: Learning plan-satisficing motion policies from demonstrations,” in *Proc. Conf. Robot Learn. (CoRL)*, ser. PMLR, vol. 205, 2023, pp. 94–105.
- [23] A. G. Puranic, J. V. Deshmukh, and S. Nikolaidis, “Learning from demonstrations using signal temporal logic in stochastic and continuous domains,” *IEEE Robot. Autom. Lett.*, vol. 6, no. 4, pp. 6250–6257, 2021.
- [24] A. Gelman and X.-L. Meng, “Simulating normalizing constants: From importance sampling to bridge sampling to path sampling,” *Statist. Sci.*, vol. 13, no. 2, pp. 163–185, 1998.
- [25] W. Shen *et al.*, “Differentiable GPU-parallelized task and motion planning,” in *Proc. Robot. Sci. Syst. (RSS)*, 2025.
- [26] B. Sundaralingam *et al.*, “cuRobo: Parallelized collision-free robot motion generation,” in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2023, pp. 8112–8119.
- [27] X. Yang *et al.*, “RoboLab: A high-fidelity simulation benchmark for analysis of task generalist policies,” in *Proc. Robot. Sci. Syst. (RSS)*, 2026.
- [28] K. Black *et al.*, “ $\pi_{0.5}$: A vision-language-action model with open-world generalization,” in *Proc. Conf. Robot Learn. (CoRL)*, ser. PMLR, vol. 305, 2025, pp. 17–40.

APPENDIX CONTENTS

Appendix A develops the inference score, Appendices B–C describe planning, and Appendix D documents perception. Evaluation and empirical details follow in Appendices E–H.

Appendix A: Rational inverse reasoning (RIR)	10
A-A Bottom-up demonstration structure and candidate skeleton sets	10
A-A.1 Symbolic trace and segmentation	10
A-A.2 Segmentation DAG and bottom-up skeleton proposals	10
A-A.3 Bottom-up skeletons in the score	10
A-B Execution likelihood and the symbolic abstraction	11
A-C Hierarchical normalization and relative specificity	11
A-C.1 Combining execution and strategy rationality	11
A-C.2 An exact specificity identity under one reference proposal	11
A-C.3 Implemented score	12
A-C.4 Finite support and computation	12
Appendix B: Generalized task and motion planning (TAMP)	12
B-A Explanation programs and grounded specifications	13
B-B Plan search	13
B-C Plan refinement	14
Appendix C: Robot task and motion planning implementation	15
C-A System overview	15
C-B Task specification	15
C-B.1 Goal types	16
C-B.2 Supported predicates	16
C-B.3 External constraints	16
C-C Planning domain	16
C-C.1 Types	16
C-C.2 Skills (operators)	16
C-C.3 Continuous parameter generators	16
C-C.4 Initial state construction	16
C-D GPU parallelized continuous CCSP solver	16
C-D.1 Algorithm overview	16
C-D.2 Subgraph-conditional sampler dependency resolution	16
C-D.3 Particle state	16
C-D.4 Effective sample size and resampling	17
C-E Continuous parameter samplers	17
C-E.1 Grasp sampling	17
C-E.2 Placement sampling	17
C-E.3 Drop/insertion pose sampling	17
C-E.4 Inverse kinematics	17

C-F Motion planning and trajectory stitching	18
C-F.1 Waypoint recording	18
C-F.2 Trajectory stitching	18
C-G Implementation details for real-robot rationality scoring	18
Appendix D: Real-to-sim reconstruction	18
D-A System overview	18
D-B VLM scene parsing	19
D-B.1 Model and API	19
D-B.2 Two-step detection with semantic parts	19
D-C SAM2 segmentation	19
D-D Neural stereo depth enhancement	19
D-E Entity generation	20
D-E.1 Entity point reconstruction	20
D-E.2 World building	20
D-F Mesh generation	21
D-G State extraction	21
D-G.1 Per-entity state	21
D-G.2 Predicate extraction	21
D-G.3 Cross-validation	21
D-H Robot-driven tracking	22
D-I Typical timing	22
Appendix E: Evaluation definitions and reporting	22
Appendix F: Simulated tabletop manipulation	22
F-A Task distribution and generalization	22
F-B Evaluation protocol and comparisons	22
Appendix G: Real-world tabletop manipulation	23
G-A Demonstrations and evaluation protocol	23
G-B Separate physical execution case study	23
G-C Task distribution and inferred explanation programs	24
G-D Case-study example of feedback diagnostics	24
Appendix H: 2D reasoning corpus and human study	24
H-A TERC simulation environment summary	24
H-B Dataset breakdown	25
H-C Examples of generated code blocks for various tasks	25
H-D Comprehension and success evaluation pipeline	27
H-D.1 LLM+human evaluation pipeline	28
H-E Human visual reasoning survey	29
H-F TERC-specific likelihood instantiation and cost function	29
H-G Additional results: iteration budget and stopping criteria	29
H-H Reproducibility notes for TERC experiments	30

APPENDIX A
RATIONAL INVERSE REASONING (RIR)

This section provides additional details on demonstration segmentation and the planner-grounded score used by Rational Inverse Reasoning (RIR). It complements Secs. IV-A–IV-C of the main paper. Domain-specific implementations and experimental settings are described in the corresponding experiment sections of this appendix.

a) Notation: We use E for an explanation program, $g = E(s_0)$ for its grounded task, π for a plan skeleton, and τ for a trajectory. A hat on a skeleton *set*, such as $\hat{\Pi}$, denotes finite support; hats on Z , F , and ℓ denote estimates. The main paper’s inverse temperatures are β_t (refinement cost) and β_p (strategy choice). The proposal density $q(\tau \mid \pi, g, s_0)$ is distinct from a robot joint configuration $\mathbf{q}$. Domain costs below instantiate the common effort functional $C(\tau)$.

A. Bottom-up demonstration structure and candidate skeleton sets

A central algorithmic idea in RIR is to combine *top-down* hypothesis proposal by a VLM with *bottom-up* structure extracted from the demonstration. Bottom-up extraction serves two roles: it identifies skeletons that could have generated the observed symbolic behavior, and it filters explanation programs whose grounded specifications cannot admit a skeleton consistent with that behavior. Top-down planning supplies counterfactual strategies against which the demonstrated strategy is evaluated.

1) Symbolic trace and segmentation: Given a demonstration $\tau = (x_0, \dots, x_T)$, perception produces object-centric states $s_t = \phi(x_t) = (O_t, R_t)$. We use the corresponding symbolic trace $(R_0, \dots, R_T)$, where R_t contains the predicates true at time t . Each operator instance ω has symbolic *preconditions* Pre_ω , *effects* Eff_ω , and *maintain* conditions Mnt_ω that must remain true throughout its execution.

A *segment* is an interval $[i, k]$ associated with an operator ω such that (i) $R_i \models \text{Pre}_\omega$; (ii) $R_t \models \text{Mnt}_\omega$ for every $t \in \{i, \dots, k-1\}$; and (iii) R_k is consistent with the operator effects and associated postconditions. Predicate content and tolerances are domain-specific, while the segmentation structure is shared across domains. These symbolic checks describe compatibility at the resolution of the reconstructed trace; they do not remove the possibility of perception or abstraction errors.

2) Segmentation DAG and bottom-up skeleton proposals: We build a directed acyclic graph (DAG) whose nodes correspond to time indices and whose edges correspond to compatible segments. A path from node 0 to node T defines an operator sequence that can symbolically explain the demonstration. Algorithms 2 and 3 describe graph construction and finite path extraction. We denote these trace-compatible proposals by $\hat{\Pi}_{\text{obs}}(\tau)$. Checking their compatibility with the grounded specification $g = E(s_0)$ gives the bottom-up set $\hat{\Pi}_{\text{bu}}(g, \tau)$ used in the main paper. The specification check accounts for the goal, ordered subgoals, and conditions maintained over their associated segments.

Algorithm 2: Build segmentation DAG from a symbolic trace

Input: Symbolic trace $(R_0, \dots, R_T)$, operator library $\mathcal{W}$
Output: Segmentation DAG $(\{0, \dots, T\}, \mathcal{E})$
Initialize the DAG with nodes $\{0, \dots, T\}$ and $\mathcal{E} \leftarrow \emptyset$;
for $i \leftarrow 0$ **to** $T-1$ **do**
 for $k \leftarrow i+1$ **to** T **do**
 foreach operator instance $\omega \in \mathcal{W}$ compatible with the
 objects in $(R_i, \dots, R_k)$ **do**
 if $R_i \models \text{Pre}_\omega$ **and** $(\forall t \in \{i, \dots, k-1\}) R_t \models \text{Mnt}_\omega$
 and $R_k \models \text{Post}_\omega$ **then**
 $\mathcal{E} \leftarrow \mathcal{E} \cup \{(i, k, \omega)\}$;
 end
 end
 end
end
return $(\{0, \dots, T\}, \mathcal{E})$;

Algorithm 3: Extract trace-compatible skeleton proposals

Input: Segmentation DAG $(\mathcal{V}, \mathcal{E})$, path budget K_{obs} ,
edge cost c_{edge}
Output: Trace-compatible skeleton proposals
 $\hat{\Pi}_{\text{obs}}(\tau)$
 $\hat{\Pi}_{\text{obs}}(\tau) \leftarrow \emptyset$;
Compute up to K_{obs} best paths from 0 to T under
 c_{edge} , using a k -shortest-path algorithm for DAGs;
foreach computed path $p = (0 \rightarrow \dots \rightarrow T)$ **do**
 Convert p into a skeleton π by reading its
 operator labels;
 Add π to $\hat{\Pi}_{\text{obs}}(\tau)$;
end
return $\hat{\Pi}_{\text{obs}}(\tau)$;

3) Bottom-up skeletons in the score: The bottom-up set enters scoring in two distinct ways. First, the indicator $1(\tau; \pi, g)$ checks whether the demonstration satisfies the specification and can be segmented as the proposed skeleton. Consequently, only demonstration-compatible skeletons contribute to the demonstration-score mixture. A specification that fails this symbolic compatibility check receives score $-\infty$.

Second, bottom-up proposals provide a finite support approximation aligned with the observations. For each candidate explanation and demonstration we form

$$\hat{\Pi}^{(k)}(E) = \hat{\Pi}_{\text{bu}}(g^{(k)}, \tau^{(k)}) \cup \hat{\Pi}_{\text{td}}(g^{(k)}, s_0^{(k)}), \quad (8)$$

where $g^{(k)} = E(s_0^{(k)})$ and $\hat{\Pi}_{\text{td}}$ contains additional skeletons generated by planning under that specification. This union is used to normalize strategy probabilities. Counterfactual skeletons need not reproduce the demonstration: their role is to represent alternative behavior that would also achieve the candidate objective. For example, a demonstrated detour provides evidence for an intended intermediate goal only relative to a more direct strategy available without that goal. The finite proposal budgets limit which alternatives are considered; the computation is not an exhaustive marginalization

over all possible skeletons.

B. Execution likelihood and the symbolic abstraction

The hierarchical model in Sec. IV-A of the main paper represents execution rationality by Boltzmann reweighting of the planner’s refinement proposal distribution $q(\tau \mid \pi, g, s_0)$. It also models strategy rationality through a distribution over skeletons. Directly evaluating a dense action-level inverse-planning model would require reasoning about many trajectory prefixes, continuous actions, and candidate task specifications. In continuous manipulation, this can entail repeated planning and refinement over a much larger space than the observed changes in task-level structure.

RIR therefore uses an abstraction assumption: the symbolic trace captures the task-relevant structure of the demonstration, while motion detail beyond that trace and effort cost is treated as nuisance variability. Two checks make the compatibility requirement explicit:

$$\begin{aligned} \text{ValidDemo}(\tau, g) &\in \{0, 1\}, \\ \text{Consistent}(\tau, \pi) &\in \{0, 1\}. \end{aligned}$$

The first checks whether the demonstrated behavior satisfies the grounded goals and constraints; the second checks whether it can be segmented as an execution of π . Together they define the support of the plan-conditional demonstration contribution.

The implementation omits the unavailable density $q(\tau \mid \pi, g, s_0)$ when scoring the observed trajectory. This omission does not establish a uniform distribution over continuous trajectories. Instead, the resulting score combines symbolic compatibility, effort, counterfactual strategy comparisons, and relative specificity. As stated in Sec. IV-B of the main paper, it is a planner-dependent surrogate rather than an exact density of the raw demonstration.

Hard compatibility is a substantive modeling assumption. A demonstration that approaches a goal but never achieves it does not support that goal under the current model. For example, hovering a mug over a coaster and then moving it elsewhere does not support an explanation requiring placement on the coaster. A demonstration that initially places the mug elsewhere and then places it on the coaster can remain compatible with that goal, provided it also satisfies any ordering and maintained conditions. Bounded strategy rationality can assign weight to such a nonoptimal strategy when it lies in the represented support. Symbolic verification is therefore efficient, but its conclusions inherit the predicates, tolerances, and errors of perception.

C. Hierarchical normalization and relative specificity

1) *Combining execution and strategy rationality:* For one demonstration, let $s_0 = \phi(x_0)$ and $g = E(s_0)$. Suppress the fixed (g, s_0) arguments and write $q_\pi(\tau) = q(\tau \mid \pi, g, s_0)$ and $I_g(\tau) = \mathbf{1}(\tau; \pi, g)$, with the skeleton dependence of I_g

understood. For $\beta_t, \beta_p > 0$, define

$$\begin{aligned} Z_\pi &= \mathbb{E}_{q_\pi}[e^{-\beta_t C(\tilde{\tau})} I_g(\tilde{\tau})], \\ F_\pi &= -\beta_t^{-1} \log Z_\pi, \quad \eta = \beta_p / \beta_t, \\ A(g, s_0) &= \sum_{\pi' \in \Pi(g, s_0)} Z_{\pi'}^\eta. \end{aligned} \quad (9)$$

The following identities apply to skeletons with $Z_\pi > 0$ and require $0 < A(g, s_0) < \infty$. Substituting the free energy into the strategy model gives

$$\begin{aligned} p_{\text{plan}}(\pi \mid g, s_0) &= \frac{Z_\pi^\eta}{A(g, s_0)}, \\ p(\tau, \pi \mid g, s_0) &= q_\pi(\tau) e^{-\beta_t C(\tau)} I_g(\tau) \\ &\quad \times \frac{Z_\pi^{\eta-1}}{A(g, s_0)}. \end{aligned} \quad (10)$$

Summing the joint density over skeletons recovers the behavioral likelihood in the main paper. This expansion shows why the execution and strategy normalizers cannot generally be treated independently. When $\beta_p = \beta_t$, the per-skeleton factor $Z_\pi^{\eta-1}$ cancels, but the global normalizer A and proposal density q_π remain. More generally, β_t affects both the conditional refinement distribution and the free energies used to compare strategies; β_p controls the latter comparison at fixed free energies. These are properties of the ideal hierarchical model, not guarantees that finite planner samples realize that model exactly.

2) *An exact specificity identity under one reference proposal:* Fix a skeleton and initial scene, and consider nested specifications $g_0 \preceq \dots \preceq g_L = g$. Let $I_\ell(\tau) = \mathbf{1}(\tau; \pi, g_\ell)$, so that $I_{\ell+1}(\tau) \leq I_\ell(\tau)$. To isolate the meaning of specificity, use the *same* normalized reference proposal $q_0(\tau \mid \pi, s_0)$, cost function, and inverse temperature for every specification, and define

$$Z_{\ell, \pi}^{(0)} = \mathbb{E}_{q_0}[e^{-\beta_t C(\tilde{\tau})} I_\ell(\tilde{\tau})]. \quad (11)$$

For a positive denominator, define the normalized cost-weighted reference distribution restricted to g_ℓ by

$$\begin{aligned} \rho_{\ell, \pi}^{(0)}(\tau) &= \frac{q_0(\tau \mid \pi, s_0)}{Z_{\ell, \pi}^{(0)}} \\ &\quad \times e^{-\beta_t C(\tau)} I_\ell(\tau). \end{aligned} \quad (12)$$

Nestedness then gives the exact identity

$$\begin{aligned} r_{\ell, \ell+1}^{(0)}(\pi) &= \frac{Z_{\ell+1, \pi}^{(0)}}{Z_{\ell, \pi}^{(0)}} \\ &= \mathbb{E}_{\rho_{\ell, \pi}^{(0)}}[I_{\ell+1}(\tilde{\tau})]. \end{aligned} \quad (13)$$

Thus the ratio is the fraction of cost-weighted execution mass remaining after adding the next condition. If all masses in the chain are positive, the ratios telescope:

$$\begin{aligned} \Delta_{\text{spec}}^{(0)}(\pi, g) &= - \sum_{\ell=0}^{L-1} \log r_{\ell, \ell+1}^{(0)}(\pi) \\ &= \log Z_{0, \pi}^{(0)} - \log Z_{L, \pi}^{(0)}, \\ &= - \log Z_{L, \pi}^{(0)} + \log Z_{0, \pi}^{(0)} \\ &\quad + \Delta_{\text{spec}}^{(0)}(\pi, g). \end{aligned} \quad (14)$$

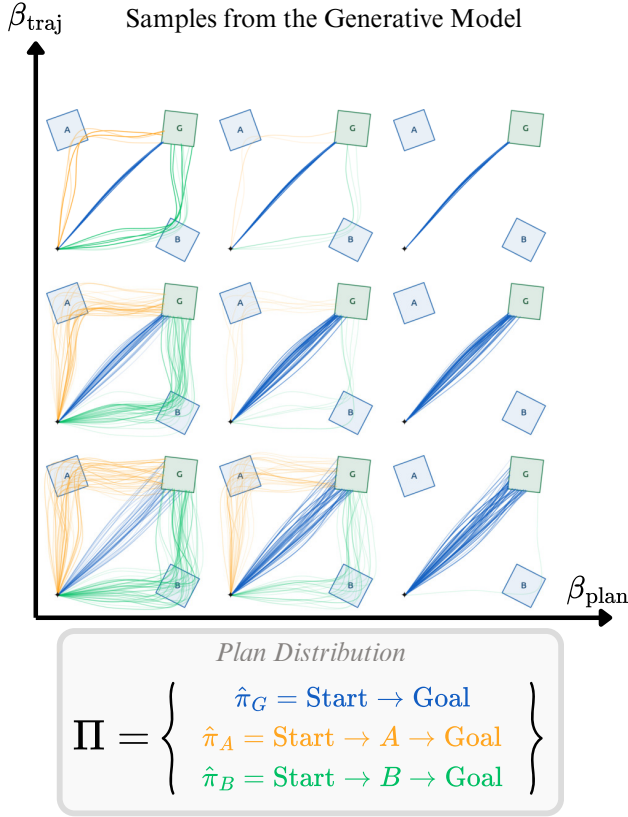


Fig. 8: Illustration of the hierarchical Boltzmann model, retained from the earlier paper. The figure’s β_{plan} and β_{traj} denote β_p and β_t here. Increasing β_p favors lower-free-energy strategies at fixed free energies; increasing β_t concentrates conditional refinements toward lower cost and also affects strategy weights through F_π . Colors distinguish direct motion and routes through intermediate regions A or B.

A condition may have high specificity while adding little motion cost: it can remove many otherwise plausible refinements. This interpretation applies only if the demonstration also satisfies the added condition.

3) *Implemented score:* The planner generates each refinement bank under its own specification g_ℓ , rather than drawing every bank from the fixed reference q_0 . The numerator and denominator of an adjacent empirical ratio therefore use samples from $q(\cdot \mid \pi, g_\ell, s_0)$. Without a suitable relation between these proposal distributions, or a corresponding density correction, that ratio need not equal Eq. (13). The unavailable proposal densities prevent such a correction here. The implemented $\tilde{\Delta}_{\text{spec}}$ is consequently a planner-dependent approximation to the common-reference construction, not an exact bridge-sampling estimator.

The score also omits the observed-trajectory density $q_\pi(\tau)$. Writing out its two levels makes the remaining normalization

explicit:

$$\begin{aligned} \log \hat{p}_{\text{plan}}(\pi) &= \eta \log \hat{Z}_\pi \\ &\quad - \text{LSE}_{\pi' \in \hat{\Pi}(E)} (\eta \log \hat{Z}_{\pi'}), \\ \hat{\ell}(E) &= \text{LSE}_{\pi \in \hat{\Pi}_{\text{bu}}(E)} \left[\log \hat{p}_{\text{plan}}(\pi) \right. \\ &\quad \left. - \beta_t C(\tau) - \log \hat{Z}_{0,\pi}^{(0)} \right. \\ &\quad \left. + \tilde{\Delta}_{\text{spec}}(\pi, g) \right]. \end{aligned} \quad (15)$$

The strategy distribution is normalized over demonstrated and counterfactual skeletons, while the final mixture includes only demonstration-compatible skeletons. A common reference specification does not make $\hat{Z}_{0,\pi}^{(0)}$ independent of π ; its term must remain inside the mixture. Finite support and zero-weight cases depend on the implementation’s support policy. The construction provides a scoring rule for the finite candidate pool, not a normalized likelihood over all raw trajectories.

4) *Finite support and computation:* Invalid refinements have zero weight, and nonzero weights are accumulated in log space. Finite sampling can nevertheless return no valid refinement or no sample satisfying the next stronger specification. Such an outcome reflects the available sample support and does not, by itself, establish physical infeasibility. The treatment of these cases is part of the score implementation rather than a consequence of log-sum-exp arithmetic.

Refinement sampling is the main computational cost and admits parallelism across particles, skeletons, and demonstrations. The GPU implementation batches grasp sampling, inverse kinematics, collision checks, and constraint evaluation. When refinement samples are cached, checks of additional conditions can reuse trajectories from the relevant weaker specification; their originating scene, skeleton, and proposal specification remain relevant to interpreting the estimate. Program-level search likewise maintains a finite set of candidates rather than enumerating the space of Python programs. Scores are summed across demonstrations, and the final pool is ranked with a uniform explicit program prior, while the VLM and its in-context simplicity preference influence which programs are proposed.

APPENDIX B

GENERALIZED TASK AND MOTION PLANNING (TAMP)

As noted in the main paper, we fulfill the long-horizon embodied reasoning tasks g with Task and Motion Planning (TAMP). In the following sections, we follow the description of TAMP components in Shen et al. [25]. A TAMP instance is defined as a tuple $\mathcal{P} = (\mathcal{W}, s_0, g)$ consisting of a set of parameterized operators $\mathcal{W}$, the initial state s_0 , and the grounded task specification g . States are represented symbolically using predicates $v \in V$; for example, the predicate $\text{At}(\text{obj}, \text{location})$ holds if object obj is at symbolic location location .

Each parameterised operator $\omega \in \mathcal{W}$ is specified by:

- **Parameters:** A tuple $\theta_\omega = (\theta_{\text{disc}}, \theta_{\text{cont}})$ of discrete (objects, symbolic locations), and continuous (trajectories, configurations) parameters.
- **Constraints:** Constraints $\text{Con}(\theta_{\text{cont}})$ on continuous parameters which must be satisfied for action feasibility. For example, kinematic constraints connecting grasp poses and feasible inverse kinematics solutions.
- **Preconditions:** A logical formula describing conditions on the symbolic state $\text{Pre}(\theta_{\text{disc}})$ which must be valid for the action to be applicable.
- **Effects:** A logical formula describing the atoms $\text{Eff}(\theta_{\text{disc}})$ made true or false upon successful execution of the action.
- **Maintain:** A logical formula describing any conditions on the symbolic state $\text{Mnt}(\theta_{\text{disc}})$ which must *remain* true during action execution, enabling runtime monitoring. For example, maintaining a hold on an object while moving it.
- **Cost:** An optional scalar cost $c_{\text{search}}(\omega)$ for optimization - usually set to 1 for symbolic search.

For example, consider a `Place` action that moves object o to support b at pose p , from configuration $\mathbf{q}_0$ to $\mathbf{q}_1$ via motion ξ , with grasp γ . Its preconditions include `Holding( $o$ )` and `AtConf( $\mathbf{q}_0$ )`; effects include `HandEmpty` and `AtConf( $\mathbf{q}_1$ )`. Kinematic and collision constraints connect the continuous parameters. The grasp is maintained during transport, up to release.

These operator conditions differ from the task-level constraints compiled from an explanation program into g . For example, retaining a grasp during transport is an operator condition; avoiding a particular region can be a constraint imposed by the inferred task. We use γ for grasps and bold $\mathbf{q}$ for configurations, reserving g for specifications and scalar $q(\tau \mid \pi, g, s_0)$ for the refinement proposal density.

A plan skeleton (high-level strategy) $\pi = (\omega_1, \dots, \omega_n)$ is a sequence of grounded abstract operators, where each grounded operator ω binds discrete parameters of an action template. For example, the grounding of the abstract operator `Place(obj, location, ...)` to the grounded operator `Place(box, shelf, ...)`.

This grounding procedure applied to the abstract actions over the space of predicates and symbolic entities in the world induces an abstract transition model. This abstract transition model $\mathcal{F} : \mathcal{R}_\phi \times \mathcal{W} \rightarrow \mathcal{R}_\phi$, where $\mathcal{R}_\phi$ is the symbolic state space induced by perception function ϕ , checks logical feasibility and updates states symbolically.

The planner searches iteratively. In the taxonomy of TAMP solvers, we consider a *search-then-sample* procedure. The planner first generates candidate symbolic skeletons using abstract reasoning (via the abstract transition model $\mathcal{F}$), then *refines* each candidate by sampling continuous parameters. If a refinement fails due to unsatisfied constraints, the symbolic planner proposes a new skeleton. This iterative search continues until a valid solution is found or a computational budget is exhausted.

The symbolic layer checks logical and temporal conditions, while continuous refinement solves geometric con-

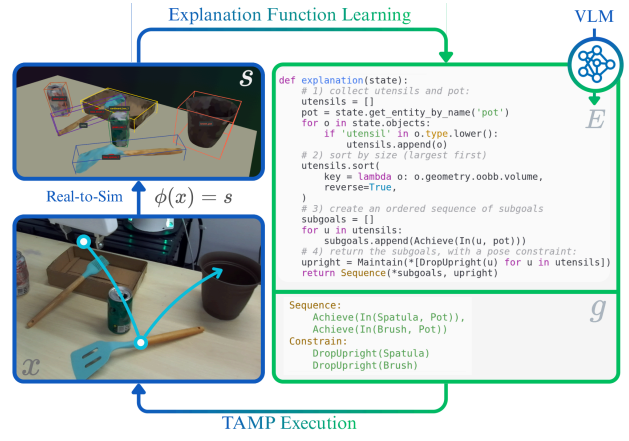


Fig. 9: An explanation program grounded in an object-centric scene. The program selects utensils and constructs ordered subgoals for upright placement in a pot. This schematic, retained from the earlier paper, uses legacy API names: `Achieve` and `Sequence` correspond to `Ach` and `Seq`, while its maintenance notation denotes the constraint structure formalized by `Constrain` in the main paper. The current formal specification is authoritative.

straints to produce an executable robot trajectory.

A. Explanation programs and grounded specifications

An explanation program $E : \mathcal{S} \rightarrow \mathcal{G}$ maps an object-centric scene $s = (O, R)$ to a grounded specification $g = E(s)$. The program can filter objects by their attributes, compute derived features, and branch on scene properties before constructing goals and constraints. The program is reused across scenes; its grounded object arguments and the resulting plan can change. Figure 9 illustrates this distinction between a reusable program and its scene-specific output.

B. Plan search

Constructing plan skeletons π involves symbolic search through the transition model induced by the set of operators $\mathcal{W}$. In this case, we must consider an entire grounded task specification g to solve for. This includes sub-goals (discrete constraints) and goals, which alters the search procedure.

Specifically, we instantiate an A* search procedure with a staged Hamming heuristic to guide the A* search towards task-relevant solutions. Although more efficient search heuristics exist, the staged Hamming heuristic is very simple to compute, and as such, results in a sufficiently fast search, even if the number of expanded nodes in the tree is high. The idea behind the staged Hamming heuristic is that we treat each remaining goal from the current stage onward as way-points. The heuristic is the sum of Hamming distances (symmetric set difference sizes) between consecutive way-points:

$$h(R, k) = |R \triangle G_k| + \sum_{j=k}^{m-1} |G_j \triangle G_{j+1}|. \quad (16)$$

Here R is the current symbolic state, $G_k, \dots, G_m$ are the remaining goal predicate sets, and $\triangle$ denotes symmetric difference. The sum is empty at the final stage. This heuristic

need not be admissible because one operator may change several predicates.

We then construct a plan generator which allows us to sample plans from the tree by cost order. Instantiating one generator allows us to re-use computation across iterations. For a fixed compute budget, we sample up to a maximum number of plan skeletons based on their symbolic cost - this is large, as the costly step in this process is plan refinement, rather than symbolic search. Across runs, we perform de-duplication and prefix matching/blocking to ensure that plans for which refinement has failed within the allocated budget are blocked from repeated refinement attempts. Budget exhaustion does not prove infeasibility.

C. Plan refinement

Refining a plan skeleton $\pi = (\omega_1, \dots, \omega_P)$ involves producing a set of parameters $\theta = \{\theta_{\omega_1}, \dots, \theta_{\omega_P}\}$ that satisfies the spatio-temporal constraint graph induced by π . That is, each plan skeleton π corresponds to a constraint satisfaction problem (CSP).

In addition to these abstract operators, we attach sampling sub-graphs to expedite the solution procedure. Specifically, we can design sampling mechanisms (motion planning, grasp sampling) on the constraint manifold, verifying constraints, and solving the local constraint subgraphs at the same time. For example, consider Fig. 10. This figure is a diagram of

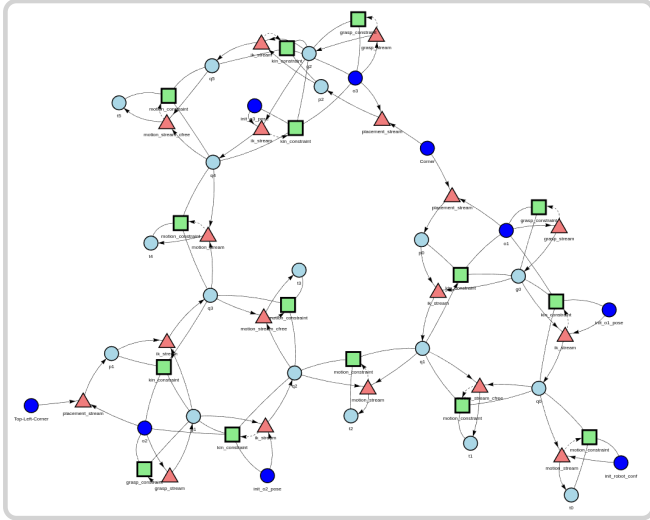


Fig. 10: An example constraint graph for the task of moving two objects into the corners of the environment.

the semi-directed constraint graph involved in solving for the continuous parameters of the task of moving both objects in the scene to the corners. The blue circles represent the parameters θ that we have to solve for. The green squares represent the constraints that we have to satisfy for the solution to be valid. The red triangles here are the samplers that we use to produce candidates for each of the parameters. The dark-blue circles denote the known parameters - those which are imposed by the plan skeleton, or are already present given the initial state of the environment.

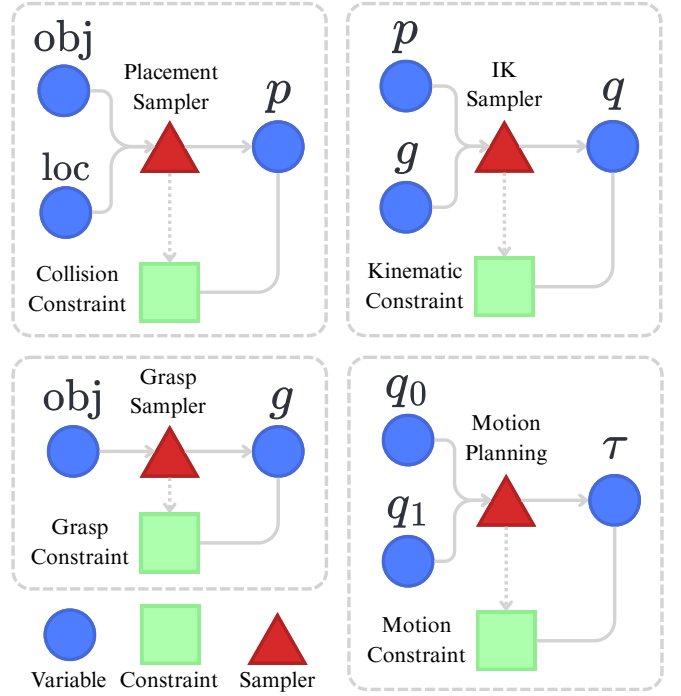


Fig. 11: Local sampling procedures which we compose for CSP subgraph solving.

Dedicated samplers exploit the local structure of this spatio-temporal constraint graph. From Fig. 10 we can see that the graph is highly structured, with many local subgraphs with similar constraint structure. This allows us to devise an informed CSP solver.

a) *Informed CSP Solver:* The constraint graphs which result from embodied reasoning tasks are not without structure. This allows us to define a hierarchical sampling scheme whose samplers satisfy local subgraphs in the CSP. These local sampling steps are shown in Fig. 11. In this way, we can prioritize: Pose $\rightarrow$ Grasp $\rightarrow$ IK $\rightarrow$ Motion Planning to solve each *Pick* and *Place* and any other similar skill subgraph sequentially.

This affords us two modes of backtracking when we cannot solve for certain parameters (given the previous parameters). First, we can backtrack within the local sub-graph solve. In other words, if we cannot sample a valid trajectory, we can backtrack and re-sample a new placement for that sub-graph. Beyond this local backtracking, we can also perform a global backtrack by discarding the current solution and resetting the solver back to the original state.

By fixing the number of each backtracks and samples we use during the plan refinement step, we fix the amount of computation to apply per-problem, when deciding whether a plan skeleton is downward-refinable - i.e., whether the plan-skeleton CSP can be solved (and therefore executed) in the environment.

Notably, in Sec. C, we detail our real-robot solver, which searches over candidate continuous assignments in parallel.

An example of the csp for a robot *Pick* operator is shown in Fig. 12. Here the unknown variables are γ, q_1, ξ (the

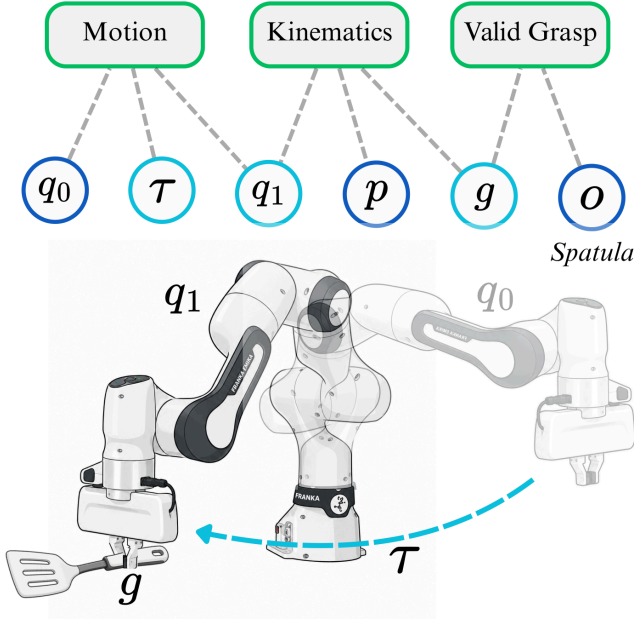


Fig. 12: Example small local CSP for the problem of `Pick-ing` an object up.

grasp, final robot configuration, and the trajectory to get there), while the known variables are o, p, q_0 (the object, its current pose, and the initial robot configuration). The path to solving it therefore calls into use the grasp, IK and motion planning sub-graph solvers, and the constraint graph structure induces the solver flow illustrated in Fig. 13, where parameters are sampled in dependency order and checked against their constraints. The retained diagrams use g, q , and τ for grasp, configuration, and local motion; these correspond to $\gamma, \mathbf{q}$, and ξ here.

APPENDIX C

ROBOT TASK AND MOTION PLANNING IMPLEMENTATION

For our real robot experiments, we use a particle-based sequential TAMP solver that combines symbolic planning with GPU-accelerated continuous constraint satisfaction. Given a reconstructed 3D scene (produced by the Real2Sim pipeline described in Appendix D) and a task specification, TAMP generates collision-free, kinematically feasible manipulation trajectories. The following sections describe the planning domain, samplers, and continuous solver.

Note that this implementation differs from the generalized TAMP implementation for the TERC experiments, as scaling to real robot experiments with more complicated tasks requires a more sophisticated and faster solver.

A. System overview

The TAMP system operates as a three-layer pipeline:

- 1) **Task Specification** (Sec. C-B): An explanation program queries the reconstructed scene and produces a grounded task specification g (e.g., “place all red blocks in the bowl”), along with optional geometric

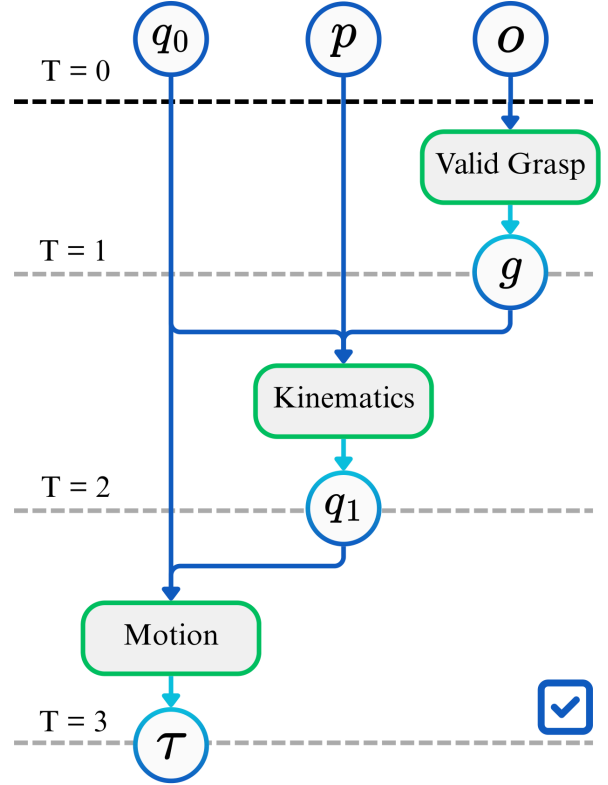


Fig. 13: Example CSP sub-graph iterative solver based on the local on-manifold/dedicated samplers outlined in Fig. 11

constraints (e.g., “fork tines should point upward when dropped”).

- 2) **Symbolic Planning** (Section C-C): A PDDL-style planner enumerates candidate plan skeletons (sequences of high-level skills such as `PICK`, `PLACE`, and `DROP`) that achieve the goal from the initial symbolic state.
- 3) **Continuous CCSP Solving** (Section C-D): A particle-based sequential solver instantiates each skeleton by sampling and filtering continuous parameters (grasps, placements, IK solutions, trajectories) using importance-weighted particles with systematic resampling.

For the real-robot experiments, the planner iterates over candidate skeletons (up to `max_skeletons=10`, with a timeout of 60s) and returns the first skeleton for which a feasible continuous solution is found at test time.

B. Task specification

Explanation programs are Python functions that receive a reconstructed environment and return a task specification in terms of goals, subgoals and constraints. The environment exposes methods for querying objects (for example, `get_pickable_objects()`, `get_containers()`, `filter_by_label()`), accessing metadata (VLM-derived labels, descriptions, semantic tags, keypoint parts), and constructing goals.

TABLE I: Planning domain types.

Type	Source	Description
object	Base	Movable rigid objects (blocks, tools, etc.)
conf	Base	Robot joint configurations $\mathbf{q} \in \mathbb{R}^{\text{DOF}}$
grasp	Base	Grasp transforms in object frame
pose	Base	Object placement/drop poses
traj	Base	Joint-space trajectories
container	Extended	Objects that can receive dropped items

TABLE II: Planning domain skills with preconditions and effects.

Skill	Parameters	Preconditions	Effects
PICK	$(o, \gamma, \mathbf{q}, \mathbf{q}', t)$	$\text{HandEmpty} \wedge \text{On}(o, _) \wedge \text{Clear}(o)$	$\text{Holding}(o) \wedge \neg \text{HandEmpty} \wedge \neg \text{On}(o, _) \wedge \text{Clear}(\text{.below})$
PLACE	$(o, \gamma, \mathbf{q}, \mathbf{q}', p, t, s)$	$\text{Holding}(o) \wedge \text{Clear}(s)$	$\text{On}(o, s) \wedge \neg \text{Holding}(o) \wedge \text{HandEmpty} \wedge \text{Clear}(o) \wedge \neg \text{Clear}(s)$
DROP	$(o, \gamma, \mathbf{q}, \mathbf{q}', p, t, c)$	$\text{Holding}(o) \wedge \text{Accessible}(c)$	$\text{InContainer}(o, c) \wedge \neg \text{Holding}(o) \wedge \text{HandEmpty}$

1) *Goal types:* The API constructors `Achieve` and `Sequence` correspond to `Ach` and `Seq` in the main paper. The system supports two goal structures:

- **Achieve**($\cdot$): Achieve a set of predicates $\mathcal{G}$ simultaneously (single-stage planning).
- **Sequence**($\cdot, \dots, \cdot$): Achieve subgoals in strict order (multi-stage planning). Each stage k must be fully satisfied before the planner proceeds to stage $k + 1$.

2) *Supported predicates:*

- $\text{On}(o, s)$: Object o is stably placed on surface s .
- $\text{InContainer}(o, c)$: Object o has been dropped into container c .
- $\text{Holding}(o)$: The robot is currently grasping object o .

3) *External constraints:* Task programs can register constraints that are enforced during continuous solving. We manually provide examples (in-context and in the API), however the system is free to write its own constraints based on the hierarchical object-centric state representation presented in Sec. D.

C. Planning domain

The planning domain follows the structure outlined in the formal description of generalized Task-and-Motion Planning (Sec. B). Here, we outline the specific types, predicates, constraints, samplers and skills used for planning.

1) *Types:* The domain defines the following types, split between the base domain and the extended domain:

2) *Skills (operators):* Table II lists all skills with their preconditions and effects:

3) *Continuous parameter generators:* Here γ denotes a grasp, $\mathbf{q}$ a robot configuration, and g the task specification; the main-paper $q(\tau \mid \pi, g, s_0)$ denotes the refinement proposal distribution. Each stream maps known continuous parameters to unknown ones. The streams are processed in dependency order during CCSP solving:

4) *Initial state construction:* The initial symbolic state is constructed from the reconstruction pipeline output - taken from Sec. D-G:

- HandEmpty : Always true initially.

TABLE III: Continuous parameter Samplers.

Stream	Inputs	Outputs	Method
sample-grasp	object o	grasp γ	Antipodal or cached grasp sampling
sample-placement	object o , surface s	pose p	Ray-based or AABB placement
sample-drop-pose	object o , container c	pose p	Drop pose sampling via opening extraction
solve-ik	object o , grasp γ , pose p	config $\mathbf{q}$	cuRobo GPU-accelerated IK
plan-motion	config $\mathbf{q}_s$, config $\mathbf{q}_g$	traj t	cuRobo trajectory optimization
plan-motion-holding	config $\mathbf{q}_s$, config $\mathbf{q}_g$, object o , grasp γ	traj t	cuRobo with attached object

- $\text{On}(o, s)$: Derived from reconstructed support relations. By default, all movable objects are on the table unless the reconstruction specifies otherwise (e.g., $\text{On}(\text{block}_1, \text{block}_2)$ for stacked objects, or $\text{InContainer}(\text{spoon}_1, \text{bowl}_1)$ for contained objects).
- $\text{Clear}(o)$: True for objects with nothing on top. Computed as the complement of $\{s : \exists o. \text{On}(o, s)\}$ restricted to movable objects.
- $\text{Accessible}(c)$: True for all non-articulated containers.

D. GPU parallelized continuous CCSP solver

The key implementation detail for the generalized TAMP planner used in the real system (and for inference) is the use of a GPU parallelized continuous constraint satisfaction problem solver. Specifically, we employ a particle-based solver that instantiates plan skeletons by sequentially sampling and filtering continuous parameters.

1) *Algorithm overview:* Given a compiled plan skeleton $\pi = (\text{skill}_1, \dots, \text{skill}_j)$, the solver maintains N_{part} particles, each representing a candidate continuous solution. The solver processes skills in order; within each skill, it processes subgraph-conditional samplers in dependency order, following the CCSP graph. At each stream, all N_{part} particles are updated in parallel (batched on GPU):

2) *Subgraph-conditional sampler dependency resolution:* Within each skill, subgraph-conditional samplers are processed in a topological order determined by fixed-point iteration over parameter dependencies:

- 1) Initialize the *known* set with parameters that have data (constants from previous skills or the initial state).
- 2) Find streams whose input parameters are all in the known set.
- 3) Add their outputs to the known set and record the stream in the execution order.
- 4) Repeat until all streams are processed or no progress can be made.

3) *Particle state:* Each particle p_i maintains:

- `robot_config`: Current joint configuration $\mathbf{q} \in \mathbb{R}^7$.
- `object_poses`: Dictionary mapping object names to current SE(3) poses.
- `held_object`: Name of currently grasped object (or None).
- `grasp`: Current grasp transform (if holding).
- `variables`: Dictionary of solved continuous parameters (grasps, poses, configs).
- `constants`: Dictionary of known constants (initial poses, fixed configs).
- `waypoints`: List of $(\mathbf{q}_{\text{start}}, \mathbf{q}_{\text{end}})$ pairs for motion stitching.

Algorithm 4: Particle Sequential CCSP Solver

Input: Compiled skills $(\text{skill}_1, \dots, \text{skill}_J)$, world state $\mathcal{W}_{\text{scene}}$, initial configuration $\mathbf{q}_0$
Output: A stitched trajectory or failure
 $\mathcal{P} \leftarrow \text{INITIALIZEPARTICLES}(N_{\text{part}}, \mathcal{W}_{\text{scene}}, \mathbf{q}_0)$
for $m \leftarrow 1$ **to** J **do**
 $\text{subgraph} \leftarrow \text{EXTRACTSUBGRAPHORDER}(\text{skill}_m)$;
 // Arrange streams in dependency order
 foreach $\text{stream } \sigma \in \text{subgraph}$ **do**
 $\text{SAMPLEANDWEIGHT}(\mathcal{P}, \sigma)$;
 // Batch sampling and feasibility filtering
 if $\sum_i w_i = 0$ **then**
 return FAILURE;
 // No surviving particle in this refinement pass
 end
 $\bar{w}_i \leftarrow w_i / \sum_j w_j$;
 $\text{ESS} \leftarrow \frac{1}{\sum_i \bar{w}_i^2}$;
 // Effective sample size from normalized weights
 if $\text{ESS} < N_{\text{part}} \rho_{\text{ESS}}$ **then**
 $\mathcal{P} \leftarrow \text{SYSTEMATICRESAMPLE}(\mathcal{P}, N_{\text{part}})$;
 // $\rho_{\text{ESS}} = 0.5$
 end
 end
 $\text{APPLYSKILLEFFECTS}(\mathcal{P}, \text{skill}_m)$;
 // Update the world state
end
if $|\{p \in \mathcal{P} : p.\text{feasible}\}| > 0$ **then**
 $p^* \leftarrow \arg \max_{p \in \mathcal{P} : p.\text{feasible}} w_p$;
 // Select the highest-weight surviving particle
 return $\text{STITCHMOTION}(p^*)$;
 // Plan trajectories
end
else
 return FAILURE;
end

- `is_feasible`: Boolean flag; infeasible particles are excluded from sampling.
- `log_weight`: Log importance weight (updated additively).

4) *Effective sample size and resampling*: Particle weights are maintained in log-space and normalized via log-sum-exp. The effective sample size (ESS) is:

$$\text{ESS} = \frac{1}{\sum_{i=1}^{N_{\text{part}}} \bar{w}_i^2}, \quad \bar{w}_i = \frac{\exp(\log w_i)}{\sum_{j=1}^{N_{\text{part}}} \exp(\log w_j)} \quad (17)$$

When $\text{ESS} < N_{\text{part}} \cdot \rho_{\text{ESS}}$ (with $\rho_{\text{ESS}} = 0.5$), systematic resampling is triggered. Systematic resampling draws N_{part} particles from the current set with probabilities proportional to their normalized weights, using a single uniform random offset $u \sim \text{Uniform}(0, 1/N_{\text{part}})$ to select indices. After resampling, all unnormalized particle weights are reset to equal values ($\log w_i = 0$, giving normalized weights $1/N_{\text{part}}$). Resampling is checked after each parameter rather than only after each skill.

E. Continuous parameter samplers

Each sub-graph solver in the CCSP is backed by a batched sampler that generates candidate values for all N_{part} particles simultaneously.

1) *Grasp sampling*: The primary grasp generation strategy is *antipodal sampling* on object meshes, generated from the `real2sim` pipeline detailed in Sec. D.

2) *Placement sampling*: Placement sampling generates candidate SE(3) poses for placing objects on support surfaces. The primary engine is GPU-accelerated ray-based placement.

a) *Ray-Based Placement*: The placement sampler uses NVIDIA Warp for GPU-accelerated mesh ray-casting. The procedure:

- 1) **Stable Pose Computation**. For each object, compute the set of physically stable resting orientations via convex hull analysis. Up to $P = 16$ stable poses are retained, each with a probability weight. For each stable pose, footprint probe points (4 corners + center on the bottom face) and ceiling probe points are pre-computed and stored on GPU.
- 2) **Scene Mesh Construction**. All entity meshes (movable objects and statics including the table) are loaded onto GPU. Entity transforms are updated dynamically to reflect current particle-tracked object poses.
- 3) **Grid Sampling**. A grid of $N_{\text{grid}} \times N_{\text{grid}}$ (default 32×32) candidate XY positions is generated over the support surface extent, centered on the support object's current position (which may have moved in multi-skill plans).
- 4) **Downward Ray-Casting**. For each grid point and each stable pose, downward rays are cast from above through the footprint probe points. A placement is valid if at least f_{support} fraction (default 0.5) of the footprint points hit the target support surface.

3) *Drop/insertion pose sampling*: Drop pose sampling generates candidate poses above/inside container openings where the robot should insert and release held objects. The procedure operates in the container's local frame, where we use the mesh to formulate a canonical 2D opening region, and sample poses and heights that are valid for the containment of the object.

4) *Inverse kinematics*: IK solving uses NVIDIA cuRobo [26] for GPU-accelerated collision-aware inverse kinematics.

a) *Batch Solving*: Target gripper poses are computed from the object pose and grasp transform as $T_{\text{grip}} = T_{\text{obj}} \circ T_{\text{grasp}}$. All N_{part} particle targets are batched into a single cuRobo IK call. To maintain CUDA graph efficiency, batches smaller than N_{part} are padded to N_{part} by repeating the last valid pose; padding results are discarded after solving.

b) *Collision Management*: Before IK solving, the collision world is updated to reflect the current particle state:

- For PICK skills, the target object's collision body is *disabled* in cuRobo to allow the gripper to reach into the object. It is re-enabled after IK solving.
- For PLACE/DROP skills, all object poses are updated in cuRobo's collision tensors to reflect their current positions (objects may have moved in earlier skills).
- After PLACE/DROP skills, the collision world is synced with the particle state.

F. Motion planning and trajectory stitching

Motion planning, the most expensive step, is deferred to the end and applied to particles that pass the preceding waypoint and constraint checks. Passing these checks does not yet establish that a collision-free connecting trajectory exists.

1) *Waypoint recording*: During CCSP solving with motion stitching enabled, motion planning samplers (plan-motion, plan-motion-holding) record ($\mathbf{q}_{\text{start}}, \mathbf{q}_{\text{end}}, \text{held_obj}, \text{grasp}$) waypoints instead of actually planning trajectories. This avoids GPU trajectory optimization during the sampling phase.

2) *Trajectory stitching*: After the CCSP solver finds a particle that passes these checks, the `stitch_motion()` method plans actual trajectories between recorded waypoints. Waypoints are grouped by attachment mode (free-space vs. holding) to minimize collision world reconfiguration. Consecutive waypoints with the same held object form a *motion group*. Within a group, the collision world (attachment configuration, disabled objects) is set up once. Between groups, attachments are reconfigured. When holding an object during motion planning, the object’s collision spheres are transformed from object frame to gripper frame via the grasp inverse transform. Spheres are down-sampled or zero-padded to match cuRobo’s fixed attachment sphere count. Then, all waypoints in a group are planned simultaneously using batched joint-space planning (implemented with cuRobo traj-opt).

G. Implementation details for real-robot rationality scoring

As detailed in Sec. IV of the main paper and Appendix A, we use TAMP to score candidate explanation programs (task specifications) against observed demonstrations using a Boltzmann rationality extended for the bi-level planning formulation. In practice, we use the parallelized sampling to perform efficient refinements over plan skeletons and task specifications, also allowing us to quickly estimate relative specificity corrections.

a) *Trajectory distance cost*: In our real-robot experiments, due to the cost of running full trajectory stitching to estimate true configuration space costs for every single particle, we instead compute approximate heuristic costs that act as estimates, relying only on the particle waypoints which can be sampled in parallel. To compute the heuristic, we densely interpolate between waypoints in joint space and compute the SE(3) path length along the resulting end-effector trajectory. For each waypoint pair ($\mathbf{q}_m^{\text{start}}, \mathbf{q}_m^{\text{end}}$):

- 1) Linearly interpolate L interior configurations in joint space: $\mathbf{q}_m^{(\ell)} = (1-t_\ell) \mathbf{q}_m^{\text{start}} + t_\ell \mathbf{q}_m^{\text{end}}$, where $t_\ell = \ell/(L+1)$ for $\ell = 0, 1, \dots, L+1$ (default $L = 10$ interior points, giving $L+2 = 12$ total).
- 2) Compute end-effector poses at all interpolated configurations via *batched GPU forward kinematics* through cuRobo. All configurations across all particles and all waypoints are concatenated into a single FK batch call for maximal GPU utilization.

- 3) Compute the dense SE(3) path length by summing consecutive segment distances:

$$C_{\text{robot}}(\theta) = \sum_{m=1}^{J_{\text{wp}}} \sum_{\ell=0}^L \left[\|\mathbf{p}_m^{(\ell+1)} - \mathbf{p}_m^{(\ell)}\|_2 + \alpha_{\text{rot}} d_{\text{SO}(3)}(\mathbf{R}_m^{(\ell)}, \mathbf{R}_m^{(\ell+1)}) \right], \quad (18)$$

Here J_{wp} is the number of waypoint pairs, θ is the sampled continuous assignment, and $\mathbf{R}_m^{(\ell)}$ is the end-effector rotation from forward kinematics. The distance $d_{\text{SO}(3)}$ measures angular separation. The waypoint cost C_{robot} approximates the trajectory effort $C(\tau)$ in the main paper; it does not measure the final stitched trajectory or certify collision freedom. This captures the nonlinearity of the kinematic mapping: a small joint-space displacement can produce a large Cartesian displacement (and vice versa), so the dense SE(3) path length is a more faithful estimate of the actual motion effort than the endpoint-only distance.

b) *VLM sampling*: For the VLM hypothesis generation, we use the Gemini-3-Pro model to generate 5 candidate explanation programs. These are candidate programs, distinct from the continuous refinement particles. This setting is specific to the robot experiments. The VLM prompts corresponding to API access for the real2sim and downstream TAMP planning system are provided in Listings ?? and ??.

APPENDIX D

REAL-TO-SIM RECONSTRUCTION

In this section we detail the Real2Sim reconstruction pipeline used to convert the raw scene observations x , consisting of two-view RGB-D inputs and robot joint states, into an object-centric scene representation used for downstream reasoning and planning. This implements the perception map $\phi : X \rightarrow S$ in Sec. III of the main paper, producing $s_t = \phi(x_t) = (O_t, R_t)$ with object attributes and spatial relations. The system transforms multi-view stereo RGB-D observations of tabletop manipulation scenes into complete simulation-ready 3D reconstructions, including per-entity watertight meshes, semantic annotations, spatial predicates, and temporal tracking of entity poses through robot demonstrations. We describe every component of the pipeline in full detail, including all algorithmic choices, hyperparameter settings, model configurations, and error handling strategies. We begin with an overview of the system architecture before providing detailed descriptions of each component in the order they are executed.

A. System overview

The Real2Sim pipeline is a modular, DAG-based system composed of seven core components, each implemented as a self-contained pipeline stage with typed inputs, outputs, caching, and profiling. The pipeline operates in two primary modes: (1) *initial frame reconstruction*, used at test time to produce a full 3D scene representation from a single multi-view stereo frame pair; and (2) *demonstration reconstruction and tracking*, used at training time to process an entire

manipulation demonstration, reconstructing the initial scene and then tracking all entity poses through the demonstration using robot gripper state. The component execution order follows a strict dependency DAG:

- 1) **VLM Scene Parsing** (Section D-B): Detects and labels all entities in multi-view images using a Vision-Language Model (VLM), producing canonical entity identifiers, bounding boxes, keypoints, semantic tags, support relations, and semantic part annotations.
- 2) **SAM2 Segmentation** (Section D-C): Segments each detected entity using the Segment Anything Model 2 (SAM2), prompted with VLM-provided bounding boxes and keypoints to produce per-entity binary masks in each camera view.
- 3) **Neural Stereo Depth** (Section D-D): Enhances raw sensor depth using FoundationStereo, a learned stereo matching network, to produce dense depth maps from calibrated stereo image pairs.
- 4) **Entity Generation** (Section D-E): Combines segmentation masks and enhanced depth to backproject masked regions into 3D point clouds, fuses multi-view observations, extracts table geometry, and populates a world manager with semantically-annotated entity objects including 3D semantic part keypoints.
- 5) **Mesh Generation** (Section D-F): Generates per-entity watertight collision meshes using GPU-accelerated TSDF fusion followed by CGAL alpha wrapping, with optional Taubin smoothing for improved grasp sampling coverage.
- 6) **State Extraction** (Section D-G): Extracts the complete symbolic world state, including geometric predicates (on, above), semantic predicates (in, is_container), and cross-validates geometric and VLM-derived relations.
- 7) **Robot-Driven Tracking** (Section D-H): Tracks entity poses through a demonstration using the robot gripper state, with mesh-based grasp detection, contact-based centering, a gripper state machine, and dynamic predicate maintenance.

B. VLM scene parsing

The VLM scene parsing component is the entry point of the pipeline and serves as the source of truth for entity identifiers throughout the system.

1) *Model and API*: The system uses Google’s Gemini family of VLMs. The default model is `gemini-3-flash-preview`, selected for its strong spatial reasoning capabilities and structured output support. The VLM is wrapped in a thin client (`GeminiVLM`) that supports both synchronous and asynchronous inference, structured JSON output with Pydantic schema validation, and batch processing with configurable concurrency.

2) *Two-step detection with semantic parts*: The VLM operates in a two-step mode that decouples entity detection from semantic part labeling:

a) *Step 1: Entity Detection*: The VLM receives multi-view images (typically a front and back pair) and is prompted to label all individual objects on the table. The prompt

TABLE IV: VLM Scene Parsing hyperparameters.

Parameter	Default	Description
<code>model</code>	<code>gemini-3-flash-preview</code>	VLM model identifier
<code>temperature</code>	0.1	Generation temperature
<code>max_output_tokens</code>	16384	Maximum response tokens
<code>thinking_budget</code>	0	Thinking tokens (0 = disabled)
<code>two_step</code>	True	Two-step part labeling
<code>max_retries</code>	3	Retries on API failure
<code>articulation_detection</code>	False	Enable articulation VLM

TABLE V: SAM2 Segmentation hyperparameters.

Parameter	Default	Description
<code>checkpoint</code>	<code>sam2_hiera_large</code>	SAM2 model checkpoint
<code>points_per_side</code>	32	Points for auto mask generation
<code>multimask_output</code>	True	Output multiple masks per prompt
<code>use_keypoint_only_for_tables</code>	True	Keypoint-only for table entities
<code>keep_server</code>	True	Keep SAM2 server alive

includes several critical instructions for multi-view correspondence: Multiple objects of the same type must be labeled separately with unique suffixes (e.g., `wooden_spoon_1`, `wooden_spoon_2`). The table itself should be labeled with a keypoint on its surface but no bounding box.

For each detected entity, the VLM returns: a normalized label, a longer description, a confidence score in $[0, 1]$, a 2D keypoint $[y, x]$ normalized to $[0, 1000]$, and a bounding box $[ymin, xmin, ymax, xmax]$ also normalized to $[0, 1000]$.

b) *Keypoint Placement Guidance*: The prompt includes critical guidance for keypoint placement that directly affects downstream segmentation quality. Keypoints must be placed on the actual visible surface of the object, not at the bounding box center (which may be empty space or another object). For objects with holes (mugs, bowls), the keypoint must be on solid material, not the opening. For stacked objects, the keypoint for the bottom object must be on an exposed portion not occluded by the top object.

c) *Step 2: Semantic Part Labeling*: A follow-up prompt requests fine-grained semantic parts for entities that require them. This step uses a part taxonomy that defines known part types (e.g., `handle`, `tool_tip`, `opening_rim`, `spout`, `blade`) and includes guidelines for minimalism (most objects need zero or one part) and projection safety (parts must be on visible surfaces). For each part, the VLM returns: a part name, a 2D keypoint, and a confidence score. Additionally, the VLM provides support relation annotations: for each entity, it reports the `support_surface` (what the entity rests on, e.g., “table” or “bowl.1”) and the `support_relation` (exactly one of ‘on’ or ‘in’), along with semantic tags such as ‘container’ or ‘graspable’. This is used later in the state extraction (Sec. D-G) to verify the geometric state extraction.

Table IV summarizes all hyperparameters for the VLM component.

C. SAM2 segmentation

The segmentation component uses the Segment Anything Model 2 (SAM2) to produce per-entity binary masks in each camera view, using the VLM-provided bounding boxes and keypoints as prompts. The SAM2 hyperparameters are outlined in Tab. V.

D. Neural stereo depth enhancement

The depth component replaces raw sensor depth with neural stereo depth for 3D reconstruction. The system uses

TABLE VI: Stereo Depth Enhancement hyperparameters.

Parameter	Default	Description
model	foundation_stereo	Depth model identifier
scale	1.0	Depth scale factor
hierarchical	False	Hierarchical inference
min_depth	0.01 m	Minimum valid depth
max_depth	2.0 m	Maximum valid depth
confidence_threshold	0.3	Confidence filter threshold
keep_server	False	Keep server alive

FoundationStereo, a learned stereo matching network, as the primary depth estimation model. FoundationStereo takes a calibrated stereo image pair (left RGB and right RGB with known baseline and intrinsics) and produces a dense disparity map, which is converted to metric depth using the camera baseline and focal length. The depth enhancement pipeline operates as follows:

- 1) For each camera in the frame bundle, check whether a stereo pair is available. If no stereo pair exists, fall back to the raw sensor depth (from raw ZED-2i).
- 2) Submit the stereo pair to the FoundationStereo server.
- 3) Critically, the system preserves the *raw sensor confidence maps* from the ZED neural depth before enhancement. The raw confidence is preferred over the neural network’s own confidence because the sensor confidence captures measurement-specific noise characteristics (e.g., reflective surfaces, edges) that are lost in the neural depth output. These confidence maps are used downstream for filtering during backprojection.

The hyperparameters for this component are outlined in Tab. VI.

E. Entity generation

The entity generation component is the core stage that transforms 2D perception outputs into a 3D world representation. It performs four sequential steps: depth application, masked backprojection, multi-view fusion, and world building.

1) *Entity point reconstruction*: The first step applies the neural stereo depth to the frame bundle. For each camera view, the enhanced depth map replaces the raw depth in the frame data structure, while the original sensor confidence map is retained for filtering. This produces an enhanced frame bundle that is used for all subsequent 3D operations.

The enhanced and masked RGBD frames are backprojected into the 3D scene, where observations from all camera views for each entity are fused into a single entity reconstruction:

- 1) All 3D points from all views are concatenated into a single array. Colors are similarly concatenated.
- 2) The merged point cloud is voxel-downsampled using Open3D with a voxel size of `voxel_size = 0.003 m` (3 mm). This both regularizes the point density and reduces memory requirements.
- 3) If `outlier_removal=True` (default), statistical outlier removal is applied with `nb_neighbors=20` and `std_ratio=2.0`. This removes isolated noisy points that may result from depth errors at object boundaries.

TABLE VII: Entity Generation hyperparameters.

Parameter	Default	Description
erosion_pixels	5	Mask erosion before backprojection
confidence_threshold	0.3	Depth confidence filter threshold
min_points	100	Min points per entity after fusion
voxel_size	0.003 m	Voxel size for downsampling
outlier_removal	True	Statistical outlier removal
compute_hull	True	Compute convex hull
compute_obb	True	Compute oriented bounding box
obb_z_up	False	Constrain OBB z-axis to world z
enable_table_extraction	True	Extract table geometry
table_xy_margin	-0.1 m	XY margin for table bounds

- 4) Entities with fewer than `min_points=100` points after fusion are discarded, as they lack sufficient geometric information for reliable reconstruction.

2) World building:

a) *Entity Creation*: A rigid body is created from the fused point cloud, which computes: the entity centroid, an axis-aligned bounding box (AABB), a convex hull (if configured), and an oriented bounding box (OBB). When configured and/or detected (default: `False`), the OBB z-axis is constrained to align with the world z-axis, which produces better oriented bounding boxes for objects resting on flat surfaces - guided by the VLM on-table extraction.

b) *Detection Info Attachment*: VLM-derived semantic metadata is attached to each entity containing: per-camera bounding boxes and keypoints, confidence scores, the VLM label and description, support surface and relation annotations, semantic tags (e.g., ‘container’, ‘graspable’), and per-view part detections. When the same entity is detected in multiple views, the system aggregates information: confidence is taken as the maximum across views, semantic tags are merged, and the first non-null support relation is preserved. State extraction subsequently consolidates the per-view support relations by majority vote (Sec. D-G).

c) *3D Semantic Part Keypoints*: For each entity with VLM-detected semantic parts (e.g., handle, spout, tool-tip), the component backprojects the 2D part keypoints to 3D using the enhanced depth maps. The process:

- 1) Groups part observations by name across all camera views.
- 2) For each part and each camera view with a valid 2D keypoint, backprojects to 3D using the same pinhole model as entity backprojection. If the depth at the exact keypoint pixel is invalid, the pipeline uses valid local depth within a search window of radius 5 pixels.
- 3) Averages the 3D keypoint positions from all valid views for robustness.
- 4) Computes the part’s relative pose as a pure translation offset from the entity centroid to the 3D keypoint position.

The resulting semantic parts are stored on the entity and are used downstream for tracking, constraint application in planning (see Sec. G), and scene prompt generation. Keypoints move rigidly with the entity.

Hyperparameters for this step are outlined in Tab. VII.

TABLE VIII: Mesh Generation hyperparameters

Parameter	Default	Description
voxel_size_m	0.002 m	TSDF voxel size
weight_threshold	0.5	TSDF weight threshold
depth_max_m	2.0 m	Max integration depth
skip_table	True	Skip table meshing
offset	0.001 m	Alpha wrap offset
alpha	0.035	Alpha wrap alpha
collision_target_triangles	8000	Target triangle count
smooth_mesh	False	Enable Taubin smoothing
smooth_max_edge	0.02 m	Max edge for subdivision
smooth_taubin_iterations	3	Smoothing iterations
smooth_taubin_lamb	0.5	Smoothing λ
smooth_taubin_nu	0.53	Anti-shrink ν

F. Mesh generation

The mesh generation component produces per-entity watertight collision meshes suitable for simulation, grasp generation and motion planning. It follows a two-stage geometry strategy: (1) observed geometry via masked TSDF fusion to capture surface details and concavities; (2) robust watertight envelope via CGAL alpha wrapping to enforce manifoldness and suppress artifacts.

For each entity, the component constructs a Truncated Signed Distance Function (TSDF) volume from multi-view depth observations and extracts a triangle mesh. After extraction, small disconnected components are removed.

The TSDF mesh may contain holes, self-intersections, and non-manifold edges due to incomplete observations and depth noise. Alpha wrapping produces a watertight envelope around the observed geometry. The watertight mesh is decimated to a target triangle count of `collision_target_triangles` (default: 8000) using quadric edge collapse. This reduces the computational cost of collision checking in downstream motion planning by reducing the mesh size.

The mesh generation hyperparameters are outlined in Tab. VIII.

G. State extraction

The state extraction component produces the complete symbolic world state by combining geometric and semantic reasoning. It extracts per-entity states, spatial predicates, and cross-validates geometric and VLM-derived relations.

1) *Per-entity state*: For each entity in the scene graph, the component extracts: the entity’s $SE(3)$ pose (from the entity’s centroid and/or OBB), detection confidence, point count, mesh availability, and semantic properties (container status, support surface, support relation, parts) derived from VLM annotations. Semantic information is consolidated across views via majority voting: the most frequently observed (surface, relation) pair is selected.

2) *Predicate extraction*: The perception system extracts four types of predicates:

a) *Exists Predicates*: A unary `exists(e)` predicate is created for every valid entity in the scene.

b) *On Predicates (Geometric)*: The geometric “on” detector uses a multi-phase approach to determine support relationships:

TABLE IX: State Extraction hyperparameters.

Parameter	Default	Description
predicates	[exists, on, in, is_container]	Predicates to extract
contact_threshold(gap_max)	0.01 m	On-contact gap tolerance
above_threshold	0.05 m	Above relation threshold
validate_geometric_vs_vlm	True	Cross-validate predicates
vlm_supersedes_geometric	True	VLM “in” overrides geometric “on”
extract_containers	True	Extract IsContainer from tags
On detector: pen_tol	0.005 m	Penetration tolerance
On detector: samples	256	Raycast sample count
On detector: min_support_ratio	0.1	Min support fraction

1) **Coarse Phase (AABB Band Test)**: For each pair of entities (A, B) , check whether A ’s AABB bottom is within a contact band of B ’s AABB top. The contact band is defined by `gap_max` (default: 0.01 m = 1 cm) for the maximum gap and `pen_tol` (default: 0.005 m = 5 mm) for allowed penetration. Both entities must also overlap in the xy plane within `tol_xy` (default: 0.005 m).

2) **Fine Phase (Convex Hull XY Intersection)**: If the coarse check passes, the convex hulls of both entities are projected onto the xy plane and their intersection area is computed. If the intersection area divided by A ’s hull area exceeds `min_overlap_ratio_a` (default: 0.0), the pair passes the fine check.

3) **Contact Refinement (Raycast Sampling)**: When `contact_refine=True` (default), the system samples `samples=256` random points within the xy overlap region and casts rays downward to determine what fraction of A ’s footprint is actually supported by B ’s surface. A pair is accepted if the support ratio exceeds `min_support_ratio=0.1`.

Optionally, when entity meshes are available, the “on” detector can use mesh geometry instead of point clouds (+convex hull) for more accurate contact computation.

c) *In Predicates (Semantic)*: Containment predicates are extracted from VLM annotations. The VLM reports a `support_relation` of “in” when an entity is inside a container. These are extracted by parsing the per-view detections and consolidating across views. This is used in combination with a similar verification check to the “On” predicate, verifying with AABB intersections.

d) *IsContainer Predicates*: Container predicates are derived from VLM semantic tags. When a detection includes “container” in its `semantic_tags` list, the corresponding entity receives an `is_container` predicate.

3) *Cross-validation*: Validating geometric predicates against VLM-derived support relations is critical. When there is a conflict e.g. the geometric detector says A is “on” B , but the VLM says A is “in” B , the “in” relation supersedes (when `vlm_supersedes_geometric=True`, which is the default). This is important because objects inside containers may geometrically appear to be “on” the container’s bottom surface, while containment and support have different meanings for planning.

The hyperparameters for state extraction are shown in Tab. IX.

TABLE X: Timing for reconstruction and tracking stages. Times are measured across representative tabletop scenes with 4–10 entities using two stereo camera pairs. Per-camera, per-entity, and tracking costs are identified separately. Server initialization is a one-time cost when servers remain active.

Stage	Time (seconds)
VLM Scene Parsing (Gemini API)	5–20
SAM2 Segmentation (per camera)	2–5
Neural Stereo Depth (per camera pair)	2–5
Entity Generation (backproject + fuse)	0.5–2
Mesh Generation (per entity, quality preset)	0.5–2
State Extraction	<1
Total (initial frame, uncached)	32.63 (29–35)
Total (initial frame, uncached, w/o warmup)	50.61 (46–56)
Total (cached, state extraction only)	<10
Tracking (per frame)	<0.01
Tracking (full segment, ~300 frames)	2–5
SAM2 server initialization (one-time)	5–10
FoundationStereo server initialization (one-time)	5–15

H. Robot-driven tracking

The tracking component processes entire manipulation demonstrations by tracking entity poses over time using the robot gripper state. This is a streaming component that processes frames sequentially, unlike the single-frame components described above. The system implements a *robot-only tracking* paradigm based on two key assumptions:

1) **Grasped objects move rigidly with the gripper.**

When the gripper grasps an entity stably, the entity’s pose is updated at each frame by composing the gripper’s world-frame pose with the fixed relative transform between the gripper and entity: $T_{\text{world} \leftarrow \text{entity}}^{(t)} = T_{\text{world} \leftarrow \text{grripper}} \cdot T_{\text{grripper} \leftarrow \text{entity}}$, where the relative transform $T_{\text{grripper} \leftarrow \text{entity}}$ is computed at the moment of grasp and remains constant until release.

2) **Ungrasped objects remain static.** Objects not currently held by the gripper stay at their last known pose. This assumes that objects rest stably on surfaces and do not move unless manipulated. That is, we maintain a quasi-static assumption on the environment.

This paradigm eliminates the need for frame-by-frame visual object tracking (which is fragile for small objects under significant occlusion) and instead leverages the precise robot state available from joint encoders, with occasional updates to verify our quasi-static assumption. Unobserved slip or passive object motion can violate these assumptions and produce errors in the reconstructed demonstration.

I. Typical timing

Table X provides typical timing for each pipeline stage on representative tabletop scenes (reconstructions during real-robot evaluation) with 4–10 entities, using a GPU-equipped workstation (RTX 5090), across 10 runs each (with and without server warm-up).

These reconstruction-stage timings are separate from the simulated inference and planning times in Appendix F.

APPENDIX E

EVALUATION DEFINITIONS AND REPORTING

Comprehension measures whether an inferred explanation matches the intended task semantics, including object selection, ordering, and constraints; it does not require identical source code. Top- k comprehension is successful when an equivalent explanation occurs among the k ranked candidates. Trajectory success checks whether planned behavior in a held-out scene satisfies the intended task. Execution success checks task completion after executing or rolling out that behavior. Evaluation is against the intended task, rather than merely the inferred specification. A non-equivalent program can still succeed on an individual scene, so these metrics are complementary.

APPENDIX F

SIMULATED TABLETOP MANIPULATION

A. Task distribution and generalization

The simulated benchmark contains 30 tabletop macro-tasks for a Franka FR3 in IsaacSim and spans more than 300 individual objects. Each macro-task has three training scenes, with three demonstrations collected per scene using VR controls and direct object manipulation. Three training scenes and three held-out scenes per task give 180 total scenes that comprise the dataset, varying object identity, object count, layout, and clutter. Each scene represents a unique layout, and uses a large object library. The tasks require combinations of goals, ordered subgoals, and constraints. Examples include transporting liquid containers while avoiding the space above a laptop, sorting condiments before stacking dishes, and arranging objects to form the letter A. The full set is shown in Tab. XI.

B. Evaluation protocol and comparisons

We evaluate inference from one or three demonstrations. In the single-scene condition, demonstrations contain basic layout variation within a scene; the across-scenes condition includes broader variation across training scenes. The inferred intent is evaluated in the held-out scenes using the comprehension, trajectory-success, and execution-success metrics defined in the main paper. Examples of the training and held-out evaluation scenes for 6 of the 30 tasks are shown in Fig. 14.

We compare RIR with Rw/oR, VLM-ORION, and VLM-VLA. Rw/oR uses the same VLM, inputs, candidate budget, and round budget as RIR, replacing planner-derived rational feedback with self-refinement. VLM-ORION uses VLM grounding of an object-centric manipulation-plan representation and supports multiple demonstrations. The original ORION algorithm doesn’t support multi-demonstration reasoning. As such, we use a retrieval-based system which uses the VLM and the grounded objects to identify the most-likely orion sub-segment to retrieve. This allows the system to use components or subsequences learned across different demonstrations at eval time. VLM-VLA infers a natural-language intent and passes it to a pretrained $\pi_{0.5}$ model for



Fig. 14: 6 of the 30 representative simulated task environments and held-out scene variants. Task descriptions are shown for the reader and are not supplied to the learner.

execution. No method receives a natural-language description of the target task. Notably, when evaluating comprehension and trajectory success rate, because ORION doesn’t produce a high-level understanding/representation of the task, we don’t evaluate its comprehension. Similarly, VLM-VLA doesn’t produce trajectories or plans, so we don’t evaluate its trajectory success.

APPENDIX G

REAL-WORLD TABLETOP MANIPULATION

This section describes the four real-world task families, their demonstration and evaluation protocols, and the separate physical execution case study. The setup uses a Franka FR3 and two ZED-2i cameras (Fig. 15). Reconstruction and planning are detailed in Appendices D and C.

A. Demonstrations and evaluation protocol

We evaluate four long-horizon macro-tasks, with three demonstration trials per task. Demonstrations last approximately 2–4 minutes and are collected by teleoperating the

end-effector with a Meta Quest headset. The test scenes contain novel objects, layouts, and distractors. The figures below retain the four task definitions and examples from the earlier experiments.

The ablations are cumulative: *Rw/oR* removes rational feedback, *−RS* additionally removes scene-state input from the VLM context, and *−RSB* further removes bottom-up proposals from that context. Removing a VLM input is distinct from removing geometry required by planning.

B. Separate physical execution case study

The 20-run case study executes a learned program on novel scenes: 19 runs passed reconstruction, 19 produced a skeleton, 17 produced a refined trajectory, and 13 completed successfully. The first-failure counts are one reconstruction failure, two refinement failures, and four missed grasps; all fractions use the 20-run denominator. The system executes the resulting plans open-loop. These runs are separate from the multi-task comprehension and behavioral evaluation in Fig. 6(b).

TABLE XI: Simulation Task List.

Idx	Task Description	Idx	Task Description
1	Place all non-fruit objects in the box, then line up the fruits by hue	16	Line up the objects by size, smallest on the left and largest on the right
2	Place objects matching the container's color into it, then discard everything else into the other container	17	Place the cube in the bowl, then place the banana in the bowl
3	Move the liquid containers to the crate, keeping them upright at all times	18	Place the hammers in the left bin and the other objects in the right bin
4	Sort the objects into bins by color: red first, then blue, then green	19	Put the animal toys in the bin
5	Arrange the tools in an aligned row, with their ends facing away from you	20	Put two mugs on the top shelf
6	Ensure there are three bananas in the grey bin	21	Stack all the mugs into a tower
7	Move all red objects into the box, moving them in order from smallest to largest	22	Separate the objects into two piles by color
8	Place the sauce condiments in the grey bin, then stack the cup in the bowl	23	Move all cups and mugs to the left, and everything else to the right
9	Stack objects of the same type together	24	Tidy all the objects on the table, then line up the food bags for collection
10	Ensure all jugs are kept upright, then discard the bottles	25	Move all objects of the most numerous type to the right, and tidy the rest into the bin
11	Stack the blocks in color order, from bottom to top: red, yellow, green, blue	26	Arrange the objects on the table to form the letter 'A'
12	Put all oranges on the plate	27	Discard all the cans into the crate or bin
13	Place all green utensils upright in the pot	28	Build a tower of one color on the right and a tower of another color on the left
14	Reorient all mugs upright, with their openings facing upwards	29	Move the liquid containers (mugs, jugs, buckets) to the bin or container without passing them above the laptop
15	Move red objects to the left and all other objects to the right	30	Line up all utensils from shortest to longest, with their handles pointing toward you

C. Task distribution and inferred explanation programs

Figures 17–20 show the four task definitions, teaching scenes, inferred programs, and held-out scenes. The programs specify object-selection rules, orderings, and constraints that can be re-grounded when the scene changes.

D. Case-study example of feedback diagnostics

To allow the VLM to improve its explanation hypotheses over time, we use the planner to provide rational feedback. This comes in the form of the rationality score, as well as additional surprise signals/diagnostic information that is useful to pinpoint exactly what parts of the program are incorrect/need adjustment. In particular, if a particular explanation program results in constraints, continuous or discrete, which are violated by the demonstrations, this is referred back to the high-level VLM, to improve guided edits. For example, consider Fig. 16, where we see an actual example of the planner feedback and subsequent improvements to the explanation programs across iterations. In this case, the VLM produces an initial hypothesis which produces the incorrect subgoals from the video (i.e. utensils first, then other items). The planner extracts this failure, and

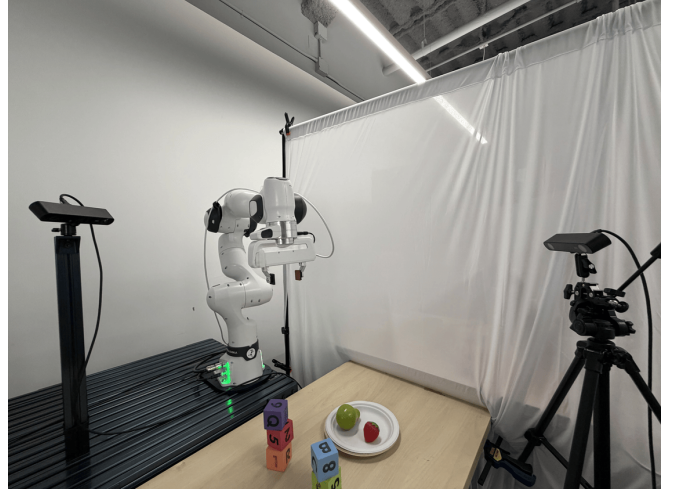


Fig. 15: Overview of the real-robot setup, where we use two ZED-2i cameras viewing the tabletop scene, and a Franka FR3 7-DoF robotic arm to perform manipulation

provides a natural language description, including the exact demonstration and stage where this failure occurs. Fig. 16 also contains an example of the extended reasoning behind a program candidate being added to the candidate pool. We can see that the VLM processes the feedback from all of the previous candidates, and updates/refines the hypotheses accordingly, increasing the rationality score across iterations.

APPENDIX H

2D REASONING CORPUS AND HUMAN STUDY

In this section, we outline details relating to the dataset presented in this paper, the Tiny Embodied Reasoning Corpus (TERC). First, we discuss the simulation and parameters. We then discuss how we gather human teleoperation data. Next, we provide a taxonomy of the dataset, providing example rollouts/images from each required task, as well as example explanation functions that the system has to extract from teleop data. We then discuss the domain-specific details regarding implementing TAMP and RIR within the TERC environments and task family. Finally, we highlight our generalization taxonomy, illustrating how we evaluate each method by changing object and agent poses, object shape, color, size and numbers of objects.

A. TERC simulation environment summary

The TERC simulation environment is built on top of the Pymunk physics simulator. We setup the agent and environment dynamics to allow for a physics-based pick-and-place simulation, rather than a contact-rich pushing simulation. The action space of the agent is $a \in \mathbb{R}^2 \times \{0, 1\}$, defining position waypoints as well as the gripper state. Similar to base implementations, a low-level PD controller is used to control the agent position towards the waypoints.

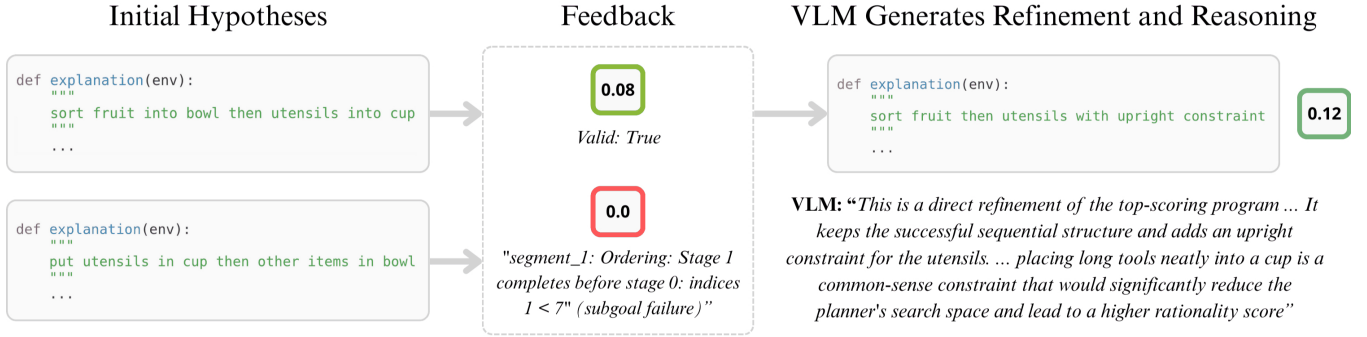


Fig. 16: Case study example of the additional feedback from the planner to provide guided edits on the explanation program hypotheses. Task shown: Tidying.

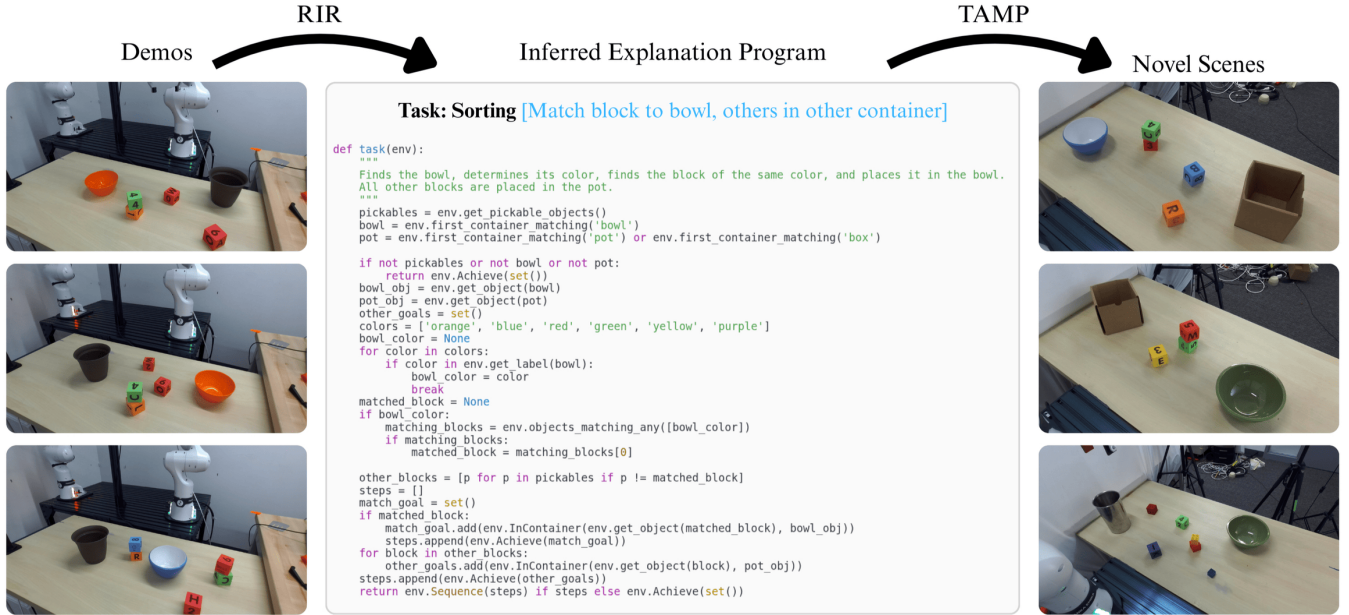


Fig. 17: Sorting: place the block matching the bowl color in that bowl, then place the remaining blocks in the other container. Teaching scenes, an inferred program, and three held-out examples are shown.

B. Dataset breakdown

The dataset used in this paper consists of 35 physical reasoning tasks, broken down into a 'spatial' subset (1-25) and a 'spatial+algorithmic' subset (26-35). Specifically, the 'spatial' subset focuses mainly on using the counterfactual reasoning afforded by the rationality computation to discern between fine-grained goals, with simple algorithmic reasoning components for filtering objects by (functions of) their properties. The 'spatial+algorithmic' subset extends this task archetype with more complicated algorithmic reasoning requirements. These tasks use programmatic operations such as counting, extrema, and conditional branching before constructing the grounded planning specification. A full breakdown of the tasks in natural language is provided in Tab. XII.

Across these tasks we observe varying degrees of difficulty associated with: discerning the specificity of a goal (top, top-corner, top-right-corner etc), using atomic parameters to

construct filters (square vs rectangle, biggest etc), relating objects in the environment to conditional behavior (if x do y etc), and whether or not the order of specific goals is significant (A then B or A and B). All of these tasks require fine-grained physical reasoning combined with algorithmic reasoning components at the abstract level.

C. Examples of generated code blocks for various tasks

To illustrate the range of hypotheses that the system must reason over, we outline some of the tasks in natural language, and the corresponding programmatic representation of E . For example, in Fig. 21, we outline three examples of simple goal definition and composition. Specifically, these code snippets define the following generalized tasks: $E_1 = \text{'move the circles to the corners'}$, $E_2 = \text{'Move the green objects to the left, and the blue objects to the right'}$ and $E_3 = \text{'Place the green box at the bottom-right-corner'}$.

Other tasks require more challenging goal definition and composition. For example, Fig. 22 shows two code snippets

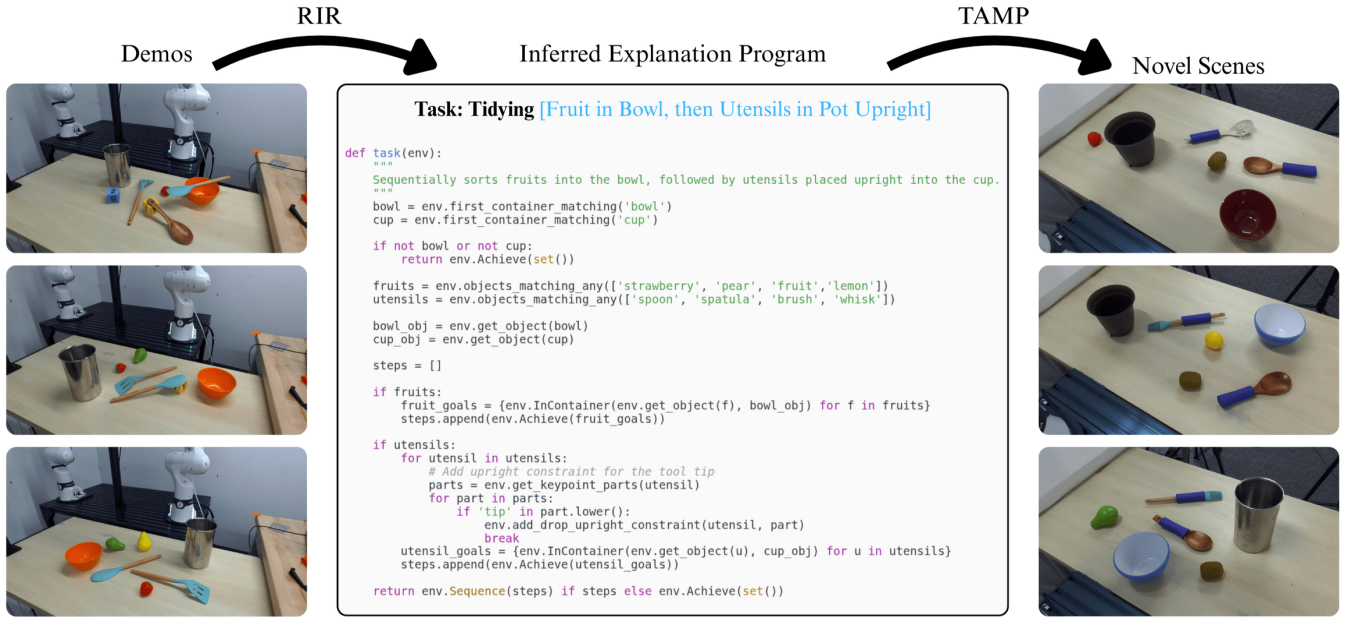


Fig. 18: Tidying: place fruit in the bowl, then place utensils upright in the pot or cup. The inferred program is re-grounded in three held-out scenes.

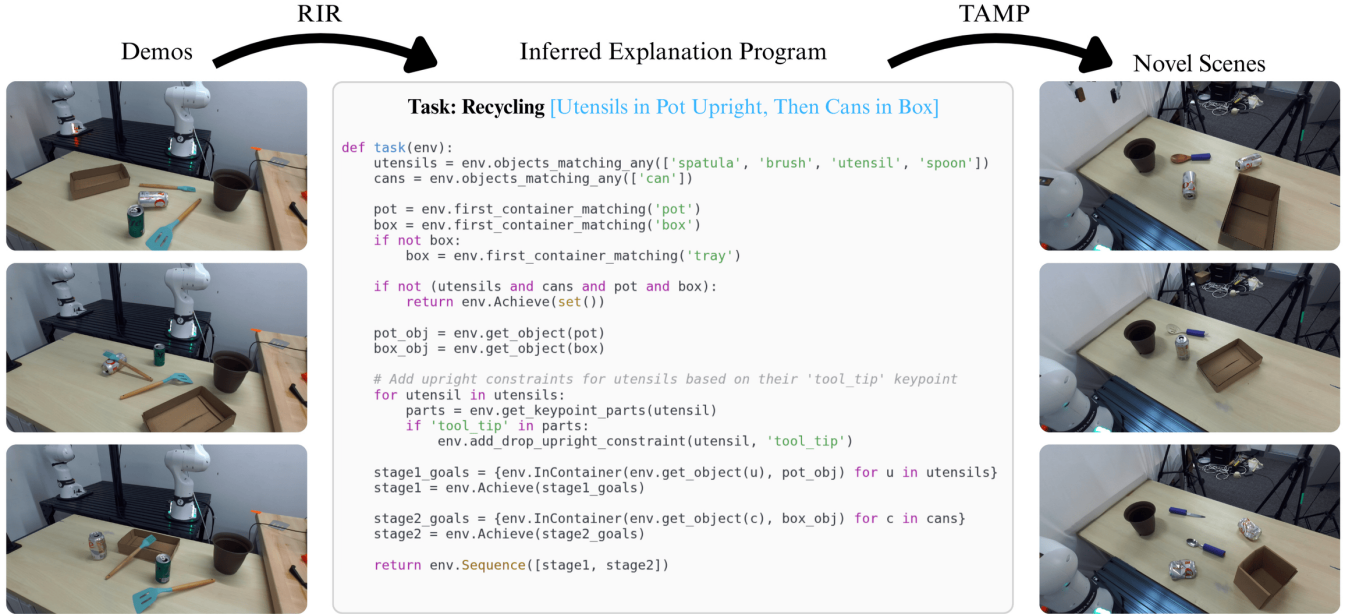


Fig. 19: Recycling: place utensils upright in the cup, then pack cans into the cardboard box. The inferred program is re-grounded in three held-out scenes.

which describe the following tasks: $E_1 = \text{'Put the most numerous objects by shape on the left'}$ and $E_2 = \text{'Put the largest circle at the bottom-right-corner'}$. Both of these cases require more complex filtering operations to be applied before task-spec construction; the first requires arithmetic and the max operation, whilst the second requires inequalities over numeric object properties.

More complicated still are tasks that require the agent to perform complex physical and algorithmic reasoning, with the added difficulty of temporal and goal compositions.

For example, consider Fig. 23, which outlines the tasks: $E_1 = \text{'Place the rectangles on the left, then the squares on the right'}$, $E_2 = \text{'In sequence, place the largest object at the top-left, the 2nd largest in the middle, and the 3rd largest at the bottom-right'}$ and $E_3 = \text{'place the largest triangle at the top-right, then place all the other triangles at the bottom-left'}$. These complex tasks require temporal and parallel compositions of the previous task archetypes, in addition to defining more advanced object/goal filtering mechanisms.

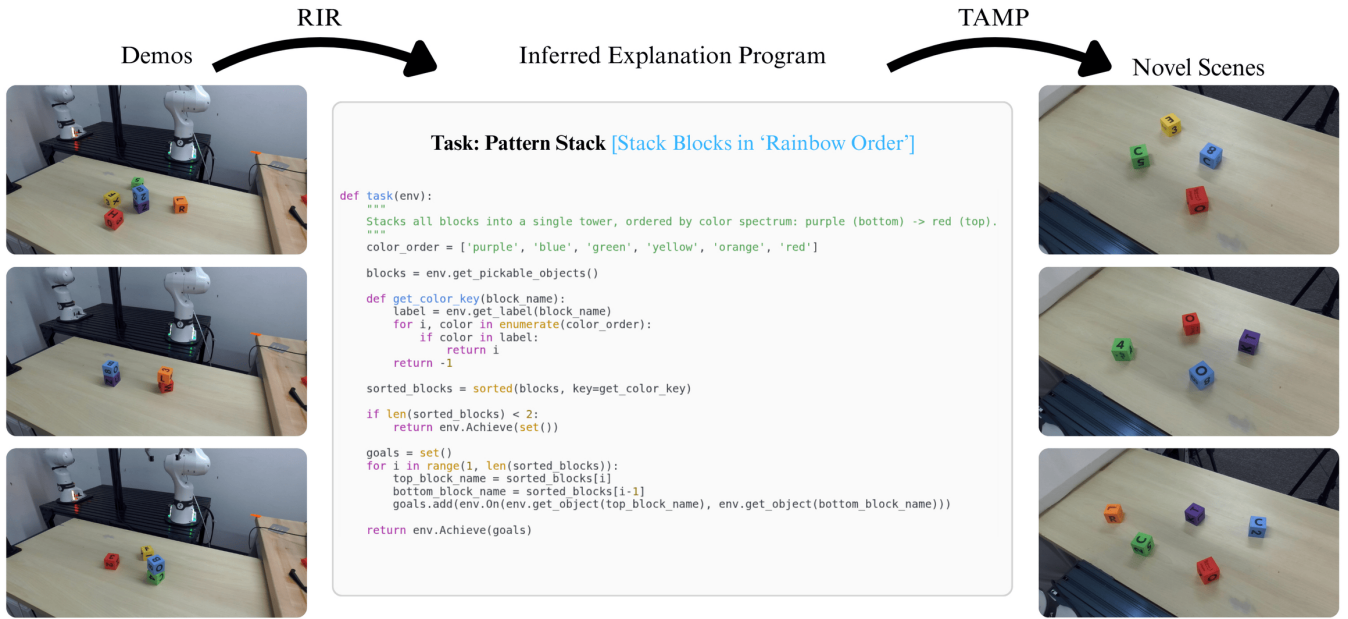


Fig. 20: Rainbow stacking: stack blocks in red, orange, yellow, green, blue, purple order from top to bottom. The inferred program is re-grounded in three held-out scenes.

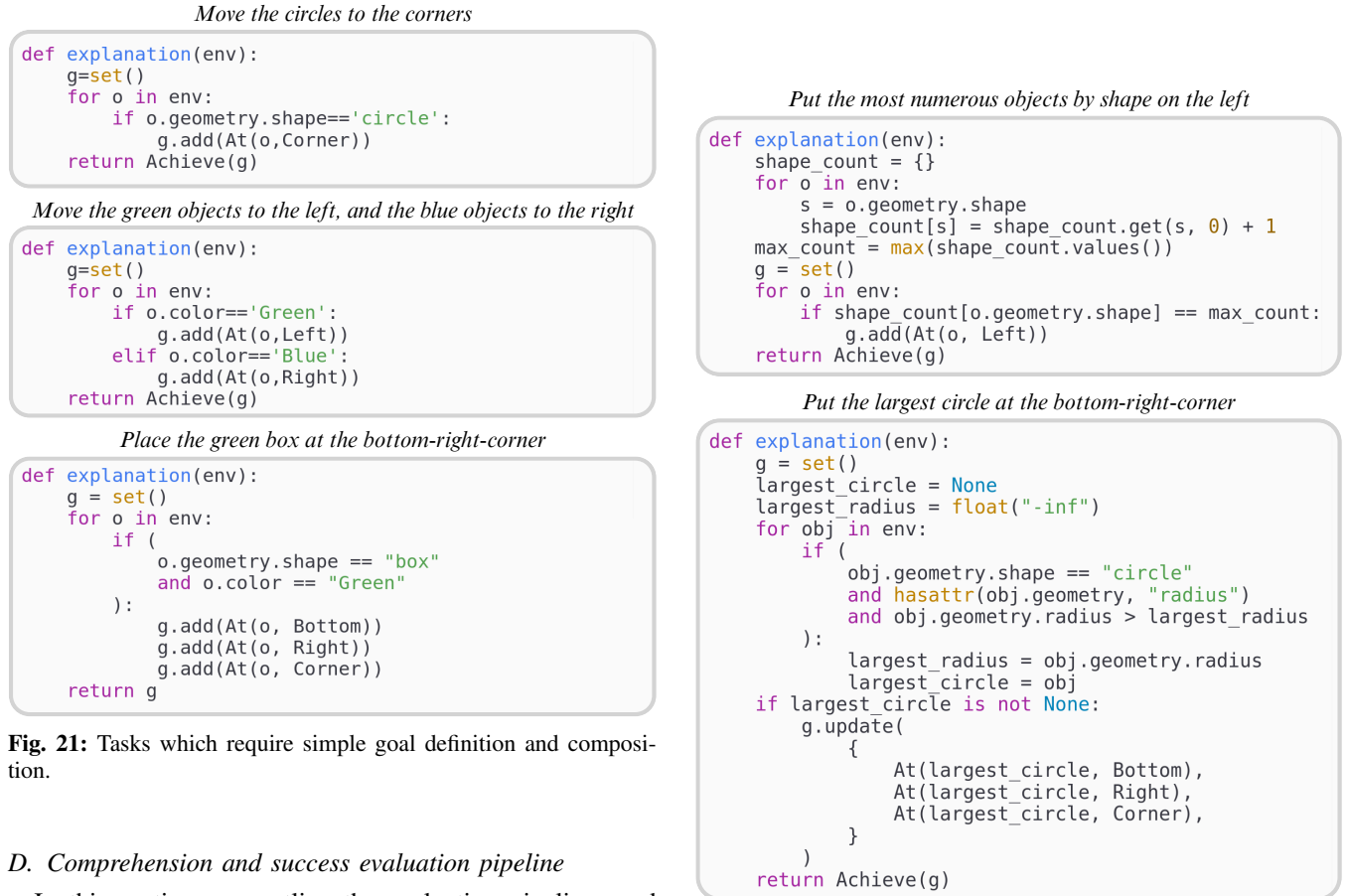


Fig. 21: Tasks which require simple goal definition and composition.

D. Comprehension and success evaluation pipeline

In this section, we outline the evaluation pipeline used for comprehension and success. Specifically, we use an LLM-based pipeline to evaluate the natural language or code-block explanations in comparison to ground-truth task descriptions (both natural language and code-blocks), and the

Fig. 22: Tasks which require challenging goal definition and composition.

TABLE XII: TERC Task List.

Idx	Task Description	Idx	Task Description
1	Move the red circle to the top-right-corner	19	Move the largest circle to the bottom-right corner
2	Put every circle in the middle	20	Move the smallest object to the centre
3	Place the green boxes at the bottom	21	Move the largest box to the bottom-left
4	Bring all the triangles to the left	22	Move the purple square to the top-left-corner
5	Put the circles in the corners	23	Move the largest blue or green circle to the left
6	Put every box on the left	24	Move the largest yellow or green box to the top-right
7	Place all the pink shapes in the middle	25	Move green objects left, and blue objects to the right
8	Place all green objects at the top	26	Put the boxes at the top-left, then put the circles at the bottom-right
9	Move all the triangles to the bottom-left	27	First move all the circles to the top, then put everything else on the bottom
10	Move all the red objects to the bottom-left corner	28	First move all the rectangles to the left, then move all the squares to the right
11	Move the green box to the bottom-right corner	29	Move the red objects to the right, then move the green objects to the left
12	Place the orange triangle in the bottom-right-corner	30	First move the largest triangle to the top-right, then move the other triangles to the bottom-left
13	Place the green objects on the left and the blue objects on the right	31	Put the objects for which there exists the most type of on the left
14	Move all the squares to the top-left	32	Put the odd-one-out by color at the top-right corner
15	Move the yellow triangle to the top-right	33	If there is a triangle, move the circles to the top-right; otherwise move the boxes there
16	Move all the purple objects to the top-left corner	34	If the pink triangle exists, move it to the top-left corner; otherwise move the pink circle there
17	Move every object except the yellow one to the bottom	35	Sort the three objects by size: largest top-left, medium centre, smallest bottom-right (in that order)
18	Move the orange box to the bottom-left corner		

TAMP solver with trajectory evaluation in the environment to measure success. Each inferred program is evaluated on three object sets with five layout initializations each, giving 15 held-out scenes. Success in this corpus refers to satisfying the intended task under a noiseless simulation rollout. The full pipeline for output generation and evaluation of these two metrics is shown in Fig. 24.

1) *LLM+human evaluation pipeline*: To extract comprehension rate, we use another LLM to compare the generated explanations in either natural language or in programmatic form to the ground-truth explanations (both natural language and example code snippets). The LLM is provided the same explanation function description (Listing ??) and in-context examples (Listing ?? and Listing ??) as the VLM hypothesis procedure. Specifically, we use the OpenAI GPT-4o model for this evaluation pipeline. We apply the same evaluation procedure to both the generated explanation functions (from RIR or Rw/oR), as well as the answers to the human survey. The LLM is encouraged to focus on evaluating overall functional equivalence, rather than exact wording/code formulation. We note that underlying semantics are anchored by an executable interface and can be audited with counterexample

Place the rectangles on the left, then the squares on the right

```
def explanation(env):
    rectangles_goal = set()
    squares_goal = set()
    for obj in env:
        if obj.geometry.shape == "box":
            height = obj.geometry.height
            width = obj.geometry.width
            if height == width: # square
                squares_goal.add(At(obj, Right))
            else: # rectangle
                rectangles_goal.add(At(obj, Left))
    return Sequence(
        Achieve(rectangles_goal),
        Achieve(squares_goal),
    )
```

In sequence, place the largest object at the top-left, the 2nd largest in the middle, and the 3rd largest at the bottom-right

```
def explanation(env):
    import math
    def area(o):
        if o.geometry.shape == 'box':
            return o.geometry.width * o.geometry.height
        if o.geometry.shape == 'circle':
            return math.pi * (o.geometry.radius ** 2)
        if o.geometry.shape == 'triangle':
            return (3 ** 0.5 / 4) * (o.geometry.side_length ** 2)
        return float('-inf')
    objs = sorted(list(env), key=area, reverse=True)
    largest, medium, smallest = objs
    g1 = {At(largest, Top), At(largest, Left)}
    g2 = {At(medium, Middle)}
    g3 = {At(smallest, Bottom), At(smallest, Right)}
    return Sequence(
        Achieve(g1),
        Achieve(g2),
        Achieve(g3),
    )
```

Place the largest triangle at the top-right, then place all the other triangles at the bottom-left

```
def explanation(env):
    largest_triangle = None
    largest_side = float("-inf")
    for obj in env:
        if obj.geometry.shape == "triangle":
            side = getattr(obj.geometry, "side_length", None)
            if side is not None and side > largest_side:
                largest_side = side
                largest_triangle = obj
    first_goal = set()
    second_goal = set()
    if largest_triangle is not None:
        first_goal.update({At(largest_triangle, Top), At(largest_triangle, Right)})
    for obj in env:
        if obj.geometry.shape == "triangle" and obj is not largest_triangle:
            second_goal.update({At(obj, Bottom), At(obj, Left)})
    return Sequence(
        Achieve(first_goal),
        Achieve(second_goal),
    )
```

Fig. 23: Tasks which require advanced goal definition, composition and temporal sequencing

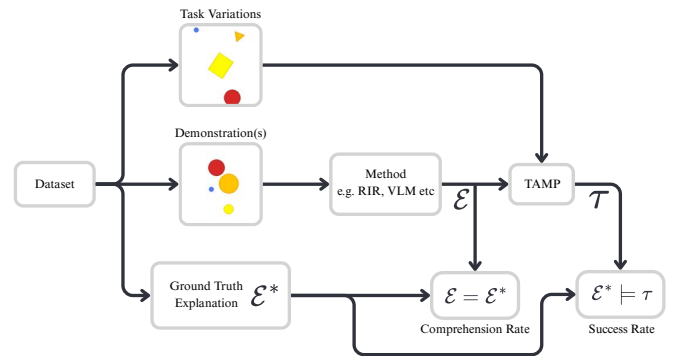


Fig. 24: Pipeline diagram for the end-to-end evaluation pipeline.

environments when eval uncertainty is high. The responses produced by the LLM are then checked by human evaluators, who correct identified errors.

E. Human visual reasoning survey

To contextualize VLM and RIR performance, we conducted a human survey measuring *comprehension* on TERC. The intent is not to optimize human performance, but to establish that (i) the tasks are challenging and (ii) comprehension is imperfect even for humans when restricted to few demonstrations and when forced to articulate explicit explanations.

a) *Survey structure*: Participants completed 35 task items using the same demonstrations provided to the VLM, with two responses per task (70 responses per participant). Each task proceeded in two phases. First, the participant watched a single demonstration video and provided a natural language explanation of what they believed the task to be. Second, the participant watched two additional videos of the *same* underlying task (3-shot total) and provided an updated explanation. The survey instructions emphasized precision and encouraged participants to state conditional and ordering structure explicitly when suspected.

b) *Quality control and exclusions*: We collected responses from 24 participants. One participant was removed due to survey misunderstanding, eliminating 70 responses out of 1680 total. The remaining 1610 responses from 23 participants were used to estimate human comprehension rates in the 1-shot and 3-shot settings.

c) *Human comprehension evaluation*: Human responses are in natural language, while ground truth is a Python explanation function. To evaluate human comprehension consistently with VLM comprehension, we used the same LLM-based equivalence evaluator, prompting the evaluator with the ground-truth program and the human’s natural language description and asking whether the description corresponds to the same task semantics. While imperfect, this evaluator is anchored by the formal DSL semantics and is used identically across methods, enabling consistent comparisons. Human annotators review uncertain outputs and correct identified evaluation errors.

d) *Survey instruction excerpt*: The survey instructions are provided in Listing ???. The full instruction block is included, including examples and clarifications about the meaning of regions such as `Corner` and about the strictness of Sequence ordering.

F. TERC-specific likelihood instantiation and cost function

The planner-dependent score used by RIR (Appendix A) requires domain-specific choices for (i) a trajectory cost functional, (ii) compute budgets for enumerating and refining alternatives, and (iii) the domain specific strategy for bilevel TAMP planning.

a) *Trajectory cost*: In TERC we evaluate the trajectory distance cost:

$$C_{2D}(\tau) = \sum_{t=1}^T \|a_t - a_{t-1}\|_2^2 + \lambda n_{\text{grasp}}(\tau), \quad (19)$$

Parameter	Value (TERC)
Demo video rate	10 Hz
Demo length	80–120 frames
Action	(x, y, grip)
Trajectory penalty coefficient λ (Eq. 19)	80
LCB confidence parameter δ (skeleton compute allocation)	0.05
Max plan candidates (rationality estimation)	5
Max solve iterations (rationality estimation)	20
Max plan candidates (full forward solve)	10
Max solve iterations (full forward solve)	100
Plan-choice inverse temperature β_p	0.5
Execution inverse temperature β_t	1.0
RIR iteration budget T_{RIR} (default)	3

TABLE XIII: TERC-specific implementation hyperparameters. These values are fixed across all reported TERC results unless otherwise noted.

where $a_t = (x_t, y_t, g_t)$ is the demonstrated action at time t , and $n_{\text{grasp}}(\tau)$ denotes the grasp-event count used by this domain cost. The second term penalizes gratuitous grasping and empirically stabilizes inference by discouraging overly complex skeletons that match the demonstration only through excessive pick-place segments. In all TERC experiments we set $\lambda = 80$.

b) *Budgets and rationality hyperparameters*: The plan-choice term in the score requires enumerating alternatives and estimating their Boltzmann-normalized probabilities. In TERC, we implement this using a fixed compute budget and a Boltzmann rationality parameter over refinement free energies. We cap (i) the number of candidate skeletons refined and (ii) the number of refinement iterations per planning call. For rationality estimation, we use a smaller budget than for full forward planning during evaluation, reflecting the fact that rationality estimation is performed many times per candidate explanation. We use $\beta_p = 0.5$ for the Boltzmann plan-choice distribution. The full set of TERC hyperparameters is summarized in Table XIII.

G. Additional results: iteration budget and stopping criteria

RIR performs an iterative rationalization-generation loop. In TERC, the number of iterations T_{RIR} is a central practical hyperparameter because each iteration requires multiple planner calls per program particle per demonstration, and allows the VLM to refine the hypothesis set. We therefore performed a targeted ablation to determine a robust default across task complexities.

a) *Fixed iteration choice* ($T_{\text{RIR}} = 3$): Empirically, $T_{\text{RIR}} = 3$ iterations worked well across a range of program complexities. For simple spatial programs, a single round of rationalization often produced large gains because feasibility and goal restrictiveness were quickly corrected. For algorithmic tasks (Tasks 26–35), additional iterations were essential to refine program structure (e.g., introducing Sequence, correctly selecting maxima, or handling conditional branches), with $T_{\text{RIR}} = 3$ providing a useful tradeoff between inference time and comprehension across all subsets. Fig. 25 illustrates the trend in top-1 and top-10 comprehension as a function of iteration count.

b) *Adaptive stopping attempts*: We also explored preliminary adaptive stopping heuristics based on stabilization

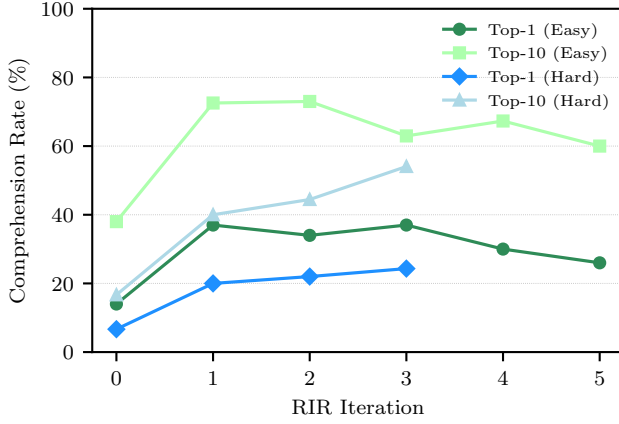


Fig. 25: Average comprehension rate (Top-1/10) for varying RIR iterations and subset difficulty: spatial only (easy) vs spatial + algorithmic (hard).

of the candidate set across iterations. However, the per-trial variation in how many iterations were required to reach a stable candidate set was large, making such heuristics unreliable.

H. Reproducibility notes for TERC experiments

a) VLM and baseline configuration: For TERC, our baseline explanation generator is a multimodal reasoning VLM equipped with structured prompting (Gemini-2.5-Pro). The first iteration of RIR uses the same model and prompt; improvements beyond the baseline are therefore attributable to the embodied rationalization loop rather than to differences in model class. The baseline “reasoning without rationality” variant (R w/o R) uses self-refinement only and does not incorporate embodied plan rationality signals, though it is provided the same scene reconstruction and optional bottom-up plan proposals for fairness.

b) Implementation and compute: All TERC experiments require repeated planning calls. To ensure tractability, we rely on caching of refinement samples and on fixed compute budgets summarized in Table XIII. We also provide top- k metrics to expose candidate-ranking quality and to better understand when the correct explanation appears among high-ranked candidates even if the top-ranked candidate is incorrect.