PO-PDDL: Learning Symbolic POMDPs from Visual Demonstrations for Robot Planning Under Uncertainty

Wenjing Tang[1,2§], **Xuanjin Jin**[1,2], **Yuan Liu**[1,2], **Renming Huang**[1,2], **Cewu Lu**[1,2], **Panpan Cai**[1,2†]

[1]Shanghai Jiao Tong University [2]Shanghai Innovation Institute

Abstract

Real-world robot task planning must operate under both stochastic action execution and partial observability, yet constructing Partially Observable Markov Decision Process (POMDP) models for real robotics domains remains difficult and labor-intensive. We introduce PO-PDDL, a symbolic formulation of POMDPs that preserves the relational structure and LLM-friendly syntax of the Planning Domain Definition Language (PDDL), while explicitly modeling partial observability, stochasticity, and beliefs. Building on this formulation, we propose a demonstration-driven pipeline for learning PO-PDDL models. The proposed method reconstructs latent symbolic state trajectories from real-robot execution videos, identifies partial observability via inconsistencies between inferred states and visual observations, and learns stochastic transition and observation models accordingly. The resulting PO-PDDL domains are reusable across tasks and enable online belief-space planning under both perception and execution uncertainty. Experiments on real-world long-horizon manipulation tasks show that our method consistently outperforms existing PDDL and POMDP model-learning approaches, achieving robust task planning under uncertainty with significantly lower planning cost.[1]

1 Introduction

Real-world robot task planning must operate under substantial uncertainty arising from both perception and action execution. In practical settings, robots face perceptual uncertainty caused by occlusions, limited sensing, and perception errors, as well as stochastic action outcomes from low-level execution modules such as motion planners and vision-language-action (VLA) [1, 2] policies. These uncertainties naturally call for modeling robot decision-making as a Partially Observable Markov Decision Process (POMDP). However, constructing POMDP models for real-world tasks remains challenging: specifying transition, observation, and reward functions requires significant expertise, and learning such models directly from experience—e.g., from visual demonstrations of policy executions—is highly non-trivial.

Existing approaches fall short of addressing this challenge. On the one hand, recent work that learns symbolic planning domains in the Planning Domain Definition Language (PDDL) [3, 4], such as UniDomain [5], supports structured reasoning but assumes fully observable and deterministic environments. On the other hand, POMDP formulations can be specified in languages such as POMDPX [6] or implemented in Python, and recent attempts have explored learning such models from demonstrations using Large Language Models (LLMs) [7]. In practice, however, these approaches remain difficult to scale to visual robot domains. They are typically evaluated in simplified low-dimensional settings and struggle with the state abstraction, stochasticity, and partial observability present in real robot demonstrations.

[§]Wenjing Tang is the visiting students at Shanghai Innovation Institute.

[†]Corresponding author: `cai_panpan@sjtu.edu.cn`

[1]Project page: `https://roboticsjtu.github.io/PO-PDDL`.

In this work, we propose a new approach to bridge this gap. We first introduce **PO-PDDL**, an LLM-friendly symbolic POMDP formulation that extends PDDL-style representations with explicit observation modeling and belief specifications. PO-PDDL preserves the structured, relational syntax that enables LLM-based generation, while supporting stochastic transitions and partial observability. Building on this formulation, we develop a learning pipeline that constructs a PO-PDDL domain directly from visual demonstrations of robot policy executions. Our method infers predicate schemas, stochastic operator models, and observation functions by reconstructing hidden symbolic trajectories and identifying inconsistencies between inferred states and visual observations. The learned domain supports compositional generalization to novel tasks within the target environment, accommodating changing scene configurations while assuming existing predicates and skills remain applicable.

We evaluate our method on real-world robot tasks involving long horizons, inter-action constraints, stochastic skill execution, and partial observability. Results show that our method effectively tackles these uncertainties, achieving a 100% success rate and the highest cumulative reward across all task levels, and substantially outperforming both other model learning methods and VLM planning.

In summary, we make the following contributions: (i) We propose PO-PDDL, a symbolic formulation that unifies PDDL-style representations with POMDP modeling of uncertainty; (ii) We develop a learning pipeline that constructs PO-PDDL domains from visual demonstrations, including stochastic transitions and observation models; (iii) We introduce a mismatch-based mechanism for identifying partial observability from data; (iv) We demonstrate that the learned domains enable belief-space planning for long-horizon robot tasks under both perception and execution uncertainty.

2 Related Work

2.1 Domain Definition Languages for Uncertainty

POMDP planning requires explicit transition, observation, reward, and initial-belief models. POMDPX [6] provides an XML-based representation, but is poorly aligned with LLM generation and often requires manual model engineering. RDDL [8] offers a Dynamic Bayesian Network (DBN)-based relational formalism, but its compact dependency syntax is difficult for current LLMs to synthesize reliably. These limitations motivate PO-PDDL, which retains PDDL-style relational structure while adding observation and belief specifications for stochastic partially observable planning.

2.2 LLM-based POMDP Planning

LLM-based POMDP planning can be broadly categorized into implicit and explicit approaches. Implicit methods use LLMs or VLMs directly for planning through prompting, uncertainty estimation, or reinforcement learning [9, 10, 11, 12, 13]. However, their beliefs and dynamics remain implicit, limiting solver-level guarantees and requiring repeated model inference online. Explicit methods combine LLMs with symbolic or probabilistic planners. LLM-MCTS [14] uses LLM-generated beliefs for Monte Carlo tree search, but handles limited hidden factors and incurs high online LLM cost. Tru-POMDP [15] uses hypothesis trees but relies on manually specified models. VLM-guided POMDPs for object search [16] are specialized to navigation. In contrast, our method learns reusable transition, observation, reward, and initial-belief components from visual demonstrations, enabling solver-based online planning without task-specific manual engineering.

2.3 Learning Planning Models from Demonstrations

Learning planning models from demonstrations reduces manual modeling effort. Prior work learns PDDL domains or problem instances from demonstrations [5, 17, 18] and repairs symbolic models using environmental feedback [19, 20, 21, 22]. Representative approaches include extracting PDDL operator preconditions and effects from demonstrations [23], learning predicates for bilevel planning with a known transition model [24], learning compact abstract planning models [25], inferring planning domains for task-and-motion planning [26], and jointly learning symbolic abstractions and continuous policies from skill demonstrations [27]. These approaches primarily focus on deterministic or fully observable settings, or learn only selected components of a planning model, rather than jointly modeling stochastic transitions and probabilistic observations.

Recent methods address partial observability via iterative deterministic replanning [28] or symbolic belief-state planning with coarse three-valued uncertainty [29], but do not learn full probabilistic transition and observation models for POMDP belief-space planning. Beyond deterministic symbolic learning, PlanU [30] learns stochastic action outcomes, while prior work [31] estimates POMDP transition and observation models. However, these methods are typically evaluated in structured domains with simplified state abstractions. In contrast, our method learns symbolic POMDP models directly from robot execution videos, including stochastic transitions and observation models, and uses them for closed-loop belief-space planning under both execution and perception uncertainty.

3 Problem Formulation: Partially Observable PDDL

3.1 Existing Formulations

PDDL. The Planning Domain Definition Language (PDDL) represents planning problems in a symbolic and factored form. A classical PDDL problem is defined by $\langle \mathcal{P}, \mathcal{A}, s_0, \mathcal{G} \rangle$, where $\mathcal{P}$ denotes the predicates, $\mathcal{A}$ operators, s_0 the initial state, and $\mathcal{G}$ the goal condition. Each operator $a \in \mathcal{A}$ is defined by: $a = \langle \mathrm{pre}(a), \mathrm{eff}(a), R(a) \rangle$, where $\mathrm{pre}(a)$ specifies applicability conditions, $\mathrm{eff}(a)$ defines deterministic or stochastic state transitions and $R(a)$ represents the reward (in PPDDL [4]). In practice, $\mathcal{P}$ and $\mathcal{A}$ are specified in a PDDL *domain file*, while s_0 and $\mathcal{G}$ are defined in a PDDL *problem file*. The former is shared across tasks within the same domain.

POMDP. A Partially Observable Markov Decision Process (POMDP) models decision-making under uncertainty as $\langle S, A, O, T, Z, R, b_0 \rangle$, where S is the state space, A the action space, O the observation space, T the transition function, Z the observation function, R the reward, and b_0 the initial belief. Both T and Z are specified as conditional probability distributions. POMDPs handle stochastic dynamics and partial observability through belief updates and belief-space planning.

PDDL and POMDPs therefore offer complementary strengths: PDDL is symbolic, interpretable, and LLM-friendly, while POMDPs provide principled uncertainty modeling. This motivates a unified formulation that preserves the relational structure of PDDL while incorporating stochastic transitions, observations, and belief-state planning.

3.2 PO-PDDL

We propose **Partially Observable PDDL (PO-PDDL)**, a symbolic formulation of POMDPs that preserves the structured syntax of PDDL while explicitly modeling partial observability. A PO-PDDL model is defined as $\mathcal{M} = \langle \mathcal{P}, \mathcal{O}, \mathcal{A}, \mathcal{Z}, b_0, \mathcal{G} \rangle$, where $\mathcal{P}$ is a set of predicate schemas, $\mathcal{O}$ is a set of observable schemas, $\mathcal{A}$ is a set of stochastic operators, $\mathcal{Z}$ is a symbolic observation model, b_0 is an initial belief, and $\mathcal{G}$ is a goal condition.

A predicate schema $p \in \mathcal{P}$ induces grounded predicates $g = p(\bar{x})$ by assigning objects $\bar{x}$ to its typed arguments; let $\mathcal{G}_\mathcal{P}$ be the set of all such predicates. A fully observed symbolic state is a complete assignment $s : \mathcal{G}_\mathcal{P} \to \{0, 1\}$, with $g \in s$ denoting $s(g) = 1$, while a partially observed state is a partial assignment $\hat{s} : \mathcal{G}_\mathcal{P} \to \{0, 1, ?\}$, where "?" denotes an unknown value. An observable schema $o_p \in \mathcal{O}$ is associated with a predicate schema p, representing perception outcomes. Once grounded, $\mathcal{P}$ and $\mathcal{O}$ define the POMDP state space S and observation space O, respectively.

Each operator $a \in \mathcal{A}$ has a precondition $\mathrm{pre}(a)$, stochastic effects $\mathrm{eff}(a)$ over grounded predicates, and a reward $R(a)$. Its stochastic effects induce a transition distribution $T(s'|s, a)$ over complete predicate assignments. The observation model $\mathcal{Z}$ is specified as conditional symbolic rules: each rule maps a logical condition over grounded predicates to a probability distribution over observable values. Once grounded, $\mathcal{A}$ defines the POMDP action space A, transition function T and reward function R, while $\mathcal{Z}$ represents the POMDP observation function Z.

The initial belief b_0 is a probability distribution over complete symbolic states s, rather than over direct visual observations. The goal $\mathcal{G}$ is a logical condition over grounded predicates, or equivalently a set of satisfying complete assignments, and induces a terminal reward.

Similarly to PDDL, $\mathcal{P}$, $\mathcal{O}$, $\mathcal{A}$, and $\mathcal{Z}$ are specified in a *domain file*, while b_0 and $\mathcal{G}$ are defined in a *problem file*. See Appendix A.1 for an example of PO-PDDL files.

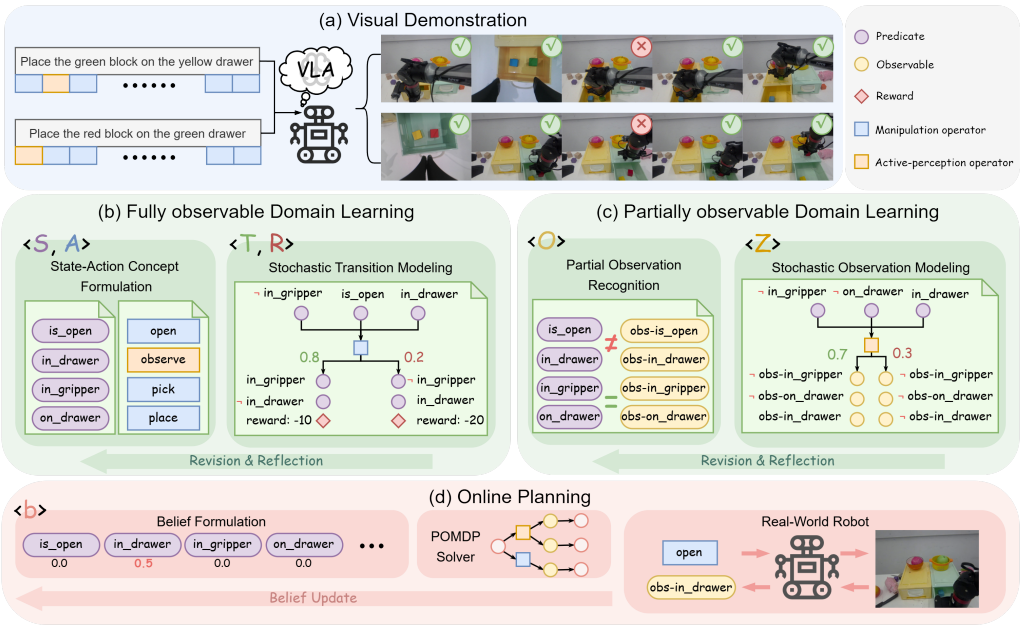


Figure 1: Overview of our method. See detailed descriptions in Section 4 and Section 5.

4 Learning a PO-PDDL Domain from Demonstrations

A PO-PDDL domain is tied to a specific scene and a fixed low-level execution stack, e.g., the low-level execution policy and the perception module, but is shared across tasks. To learn such a domain, we assume a visual-language demonstration dataset $\mathcal{D} = \{D_i\}_{i=1}^{K}$, where each episode D_i rolls out a multi-step high-level task (Fig. 1a). The high-level task is given as an ordered sequence of natural-language subtasks, each executed by the low-level policy. During the rollout, if a subtask fails, a recovery subtask is inserted, until the overall task is completed.

The resulting episode of task i is represented as $D_i = (\mathcal{V}_i, \mathcal{L}_i)$. This includes visual observations $\mathcal{V}_i = \{(V_{i,t}^{\mathrm{h}}, V_{i,t}^{\mathrm{w}})\}_{t=0}^{T_i}$, comprising synchronized RGB streams from head and wrist cameras. Language annotations $\mathcal{L}_i = (L_i, \{(L_{i,t}, X_{i,t}, Y_{i,t})\}_{t=1}^{T_i})$ contain a task instruction L_i and step-level annotations—$L_{i,t}$ is the subtask instruction at step t, $X_{i,t}$ the VLM-generated scene description, and $Y_{i,t} \in \{0, 1\}$ the execution outcome. Videos are segmented into subtask-aligned clips. Our goal is to learn a PO-PDDL domain $\mathcal{M} = \langle \mathcal{P}, \mathcal{O}, \mathcal{A}, \mathcal{Z} \rangle$ from $\mathcal{D} = \{D_i\}_{i=1}^{K}$, where each problem instance in this domain induces a POMDP model $\langle S, A, O, T, Z, R \rangle$.

The learning pipeline proceeds in two stages. We first learn a fully observable symbolic model $(\mathcal{P}, \mathcal{A})$ by reconstructing hidden state trajectories (Fig. 1b). We then identify partially observable components $(\mathcal{O}, \mathcal{Z})$ by analyzing contradictions between reconstructed states and visual observations (Fig. 1c), yielding a complete PO-PDDL domain.

4.1 Fully Observable Domain Learning

The following pipeline learns predicate schemas and operator interfaces of a fully observable domain.

4.1.1 State-Action Concept Formulation

We begin by inducing the symbolic vocabulary from visual-language demonstrations $\mathcal{D}$.

(a) *State concepts*. An LLM takes as input the task instruction L_i, subtask instructions $\{L_{i,t}\}$, and VLM-generated scene descriptions $\{X_{i,t}\}$. It generates a set of *types* that capture static attributes of entities (e.g., object classes), and a set of *predicates* $\mathcal{P}$ that represent variable attributes, including object properties and spatial relations that evolve across time and are necessary for explaining task progression (e.g., "object_on_table", "container_is_open").

(b) *Action concepts*. The LLM abstracts subtask instructions $\{L_{i,t}\}$ into a set of operator interfaces $\tilde{\mathcal{A}}$, where each interface specifies an operator name and parameter signature, but without preconditions or effects, analogous to function interfaces without implementation. Operators are further classified as either *manipulation operators*, which intend to change the environment state, or *active-perception operators*, which aim at gathering information about unknown predicate values.

4.1.2 Stochastic Transition Modeling

Given predicate schemas $\mathcal{P}$ and operator interfaces $\tilde{\mathcal{A}}$, we now construct stochastic operator models.

(a) *Goal and action grounding.* For each episode D_i, an LLM formulates a symbolic goal $\mathcal{G}_i$ according to the task instruction L_i, using $\mathcal{P}$ as the vocabulary. Subtask instructions $\{L_{i,t}\}_{t=1}^{T_i}$ are then mapped to an action sequence $\mathbf{a}_i = (a_{i,1}, \ldots, a_{i,T_i})$ by selecting operator interfaces from $\tilde{\mathcal{A}}$ and assigning parameters (e.g., typed objects). Next, we derive state trajectories.

(b) *State trajectory reconstruction.* Due to partial observability, scene descriptions $\{X_{i,t}\}$ may not reflect ground-truth states. We propose forward-backward inference. In the forward pass, for each action $a_{i,t}$, an LLM infers its symbolic effects $\epsilon_{i,t} = (\epsilon_{i,t}^+, \epsilon_{i,t}^-)$ from visual observations before and after execution, where $\epsilon_{i,t}^+$ and $\epsilon_{i,t}^-$ denote added and removed grounded predicates. Next, given $\{\mathbf{a}_i\}$ and $\{\epsilon_{i,t}\}$, together with scene descriptions $\{X_{i,t}\}_{t=0}^{T_i}$ and the goal $\mathcal{G}_i$, the LLM performs backward inference to derive a ground-truth initial state $s_{i,0}$. Then, starting from $s_{i,0}$, sequentially applying effects $\{\epsilon_{i,t}\}$ yields the reconstructed trajectory $\{s_{i,t}\}_{t=0}^{T_i}$. Finally, we enforce goal consistency by requiring $s_{i,T_i} \models \mathcal{G}_i$; if violated, the LLM revises $s_{i,0}$, $\{\epsilon_{i,t}\}$, or $\mathcal{G}_i$ until consistency is achieved.

(c) *Stochastic operator modeling.* Now, we can model preconditions and stochastic effects of operators. For each operator $\alpha \in \tilde{\mathcal{A}}$, its precondition is obtained as predicate assignments that are consistently seen in all pre-states $\{s_{i,t}\}$ where α is applied. Operator effects are obtained by clustering all seen effects $\{\epsilon_{i,t}\}$ into distinct modes and assigning probabilities to each mode based on empirical frequencies. The reward of each operator is defined as the negative execution cost, which is proportional to the average duration of its corresponding subtask. The above procedure transforms abstract operator interfaces $\tilde{\mathcal{A}}$ into fully-specified operators $\mathcal{A}$.

4.1.3 Domain Review

To ensure quality, we apply a VLM-based verification loop to refine the learned operator set $\mathcal{A}$. For each effect mode of an operator $\alpha \in \mathcal{A}$, we sample a representative subtask instance (i, t) and prompt the VLM to check whether its inferred predicates, preconditions, and effects are consistent with the visual observations $(V_{i,t}^{\mathrm{h}}, V_{i,t}^{\mathrm{w}})$. If any mismatch is found, the VLM revises the corresponding symbolic descriptions of that step. The affected state trajectory and operator models are updated accordingly. This loop repeats until no inconsistencies remain. After convergence, we obtain a refined fully observable domain $(\mathcal{P}, \mathcal{A})$. When grounded, it provides POMDP components S, A, T, and R.

4.2 Partially Observable Domain Learning

Next, we derive the partially observable components $(\mathcal{O}, \mathcal{Z})$ by identifying state-observation contradictions.

4.2.1 Partial Observability Recognition

We recognize partial observability by identifying mismatches between the reconstructed states and visual states directly parsed from observations.

(a) *Visual state parsing.* For each step $D_{i,t}$, we parse a symbolic visual state $\hat{s}_{i,t}$, by prompting an LLM to assign predicates in $\mathcal{P}$ according to the scene description $X_{i,t}$, with values in $\{0, 1, ?\}$.

(b) *Contradiction discovery.* For each grounded predicate $g = p(\bar{x})$, we compare its value $\hat{s}_{i,t}(g)$ in the visual state with its value $s_{i,t}(g)$ in the reconstructed state. If $\hat{s}_{i,t}(g) \neq ?$ and $\hat{s}_{i,t}(g) \neq s_{i,t}(g)$, we discover a *contradiction* for the predicate schema p. To avoid mis-detection of contradictions due to erroneous visual grounding or trajectory reconstruction, we use a VLM to check the raw visual observations. If any mistake is detected, we return to the fully observable learning stage for revision.

(c) *Observable concept formation.* An LLM proposes an observable schema o_p for each predicate schema p with contradictions discovered; all such schemas form the observable set $\mathcal{O}$. Further, we

refer to a predicate schema without an associated observable schema as a *fully observable* predicate, since its observation outcomes are always consistent with the hidden state; a predicate schema with an associated observable schema is referred to as a *partially observable* predicate.

4.2.2 Stochastic Observation Modeling

Given predicate schemas $\mathcal{P}$ and observable schemas $\mathcal{O}$, we estimate a symbolic observation model $\mathcal{Z}$. For fully observable predicates, the observation model is always an identical mapping. For a partially observable predicate p, the observation model specifies $\mathcal{Z}(o \mid p, a)$ and $\mathcal{Z}(o \mid \neg p, a)$, where $o \in \Omega_p = \{o_p^+, o_p^-\}$ denotes a positive or negative observation outcome.

Moreover, observation models are defined differently for manipulation operators and active-perception operators. We refer to them as *passive observation* and *active observation* functions, respectively.

(a) *Passive observation.* For a step (i, t) corresponding to a manipulation action $a_{i,t}$, we parse the observable values $o_{i,t}$ from the visual state $\hat{s}_{i,t}$, by directly mapping predicate names to the associated observable names. For each partially observable grounded predicate $g = p(\bar{x})$ that exists in $\hat{s}_{i,t}$, we construct a data tuple $(o_{i,t}(g), \hat{s}_{i,t}(g), a_{i,t})$.
(b) *Active observation.* The observation model of an active-perception action $a_{i,t}$ highly concentrates on its observation intent, i.e., the predicate for which it aims to gather information. A VLM watches the raw video clip $(V_{i,t}^h, V_{i,t}^w)$ to infer which grounded predicate $g = p(\bar{x})$ is of interest, and determines its observation outcome $o_{i,t}(g) \in \Omega_p$. This gives a data tuple $(o_{i,t}(g), \hat{s}_{i,t}(g), a_{i,t})$.
(c) *Observation likelihood estimation.* We estimate observation likelihoods by empirical frequency. For each action a and partially observable predicate p, samples satisfying p yield the empirical observation distribution $\mathcal{Z}(o \mid p, a)$ for all $o \in \Omega_p$; samples with $\neg p$ yield $\mathcal{Z}(o \mid \neg p, a)$. These conditional distributions are referred to as *observation rules*; the collection of them defines the observation model $\mathcal{Z}$.

The above procedure produces the partially observable components $(\mathcal{O}, \mathcal{Z})$ of the PO-PDDL domain. Once grounded, they induce the POMDP observation space O and observation model Z.

5 Planning with a Learned PO-PDDL Domain

Given a learned PO-PDDL domain $\langle \mathcal{P}, \mathcal{O}, \mathcal{A}, \mathcal{Z} \rangle$, we construct a task-specific problem instance. Together with the learned domain, it induces a POMDP over predicate assignments, which we solve via belief-space planning (Fig. 1d).

5.1 PO-PDDL Problem Generation

We construct the PO-PDDL problem instance from an initial scene observation and a task instruction L, which are used to generate the initial belief b_0 and goal condition $\mathcal{G}$, respectively.

(a) *Initial belief generation.* We construct an initial belief b_0 over complete symbolic states. Fully observable predicates are grounded directly from the initial scene observation $(I_0^{\mathrm{h}}, I_0^{\mathrm{w}})$ using a VLM, yielding deterministic assignments. For partially observable predicates, we identify structural dependencies, such as mutual exclusiveness among object locations, to define a feasible space of consistent assignments. Over this space, we combine an empirical prior b^{emp} estimated from demonstration initial states $\{s_{i,0}\}$ with a uniform prior b^{uni} over feasible assignments: $b_0 = \lambda b^{\mathrm{emp}} + (1 - \lambda) b^{\mathrm{uni}}$, where $\lambda \in [0, 1]$.
(b) *Goal specification.* The goal set $\mathcal{G}$ is generated from the task instruction L using an LLM. The LLM selects relevant predicate schemas, groups mutually-exclusive predicates, and enumerates feasible assignments within each group. These group-wise assignments are composed into fully-assigned states and checked against L; all semantically-matching states are included in $\mathcal{G}$.

5.2 Belief-Space Planning

Planning is performed in the belief space over symbolic states induced by the learned PO-PDDL domain. Starting from the initial belief b_0, the objective is to select actions that maximize expected cumulative reward under uncertainty, with terminal conditions defined by the goal $\mathcal{G}$.

At each time step t, the agent maintains a belief $b_t(s)$ over complete predicate assignments s. After executing action a_t and receiving observation o_{t+1}, the belief is updated via Bayesian filtering: $b_{t+1}(s') \propto Z(o_{t+1} \mid s') \sum_s T(s' \mid s, a_t)\, b_t(s)$, where T and Z are induced by the stochastic operator effects and symbolic observation model, respectively.

We solve the resulting POMDP in an online manner using DESPOT [32]. At each time step, DESPOT samples state hypotheses from b_t, performs sparse belief-tree search by simulating action-observation trajectories under (T, Z), and returns the root action with the highest estimated value. The agent then executes this action, incorporates the new observation, and replans in a receding-horizon loop until sufficient belief mass satisfies $\mathcal{G}$ or termination is reached.

6 Experiments

We evaluate our method on real-world robotic planning tasks, to assess whether PO-PDDL can learn reusable symbolic POMDP models under action constraints and stochastic action and perception outcomes. Results show that PO-PDDL consistently achieves the strongest planning performance across all tasks, substantially outperforming existing model-learning approaches and VLM planning.

6.1 Experimental Setup

We conduct experiments in a real-world tabletop scene, shown in Figure 2. The environment contains drawers, cups, blocks, and a movable black box. We construct two task sets: (1) 5 short-horizon demonstration tasks (average horizon 3.4 steps), each rolled out multiple times to collect visual demonstrations for domain learning; and (2) 5 longer-horizon evaluation tasks, requiring compositional reasoning and online planning under uncertainty.

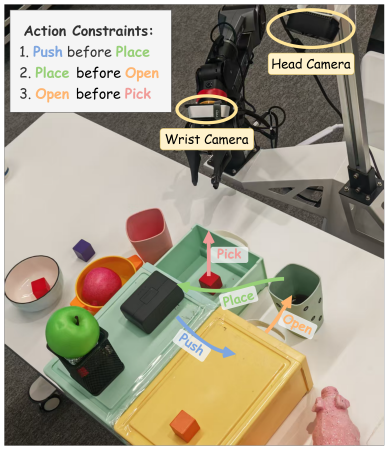


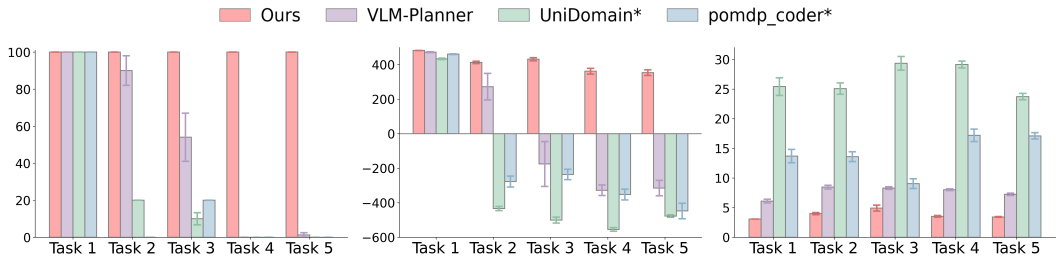
Figure 2: Experimental setup.

For the low-level stack, manipulation is performed by local VLA skills chained with motion planning, and perception is provided by a VLM operating on RGB observations. The VLA policy is based on the public release of $\pi_{0.5}$ [2], post-trained on in-domain robot data. We train 41 manipulation skills for opening drawers, moving cups and blocks, and pushing the black box, together with 2 active-perception skills for looking into a cup and looking into a drawer. The resulting domain contains both execution uncertainty from imperfect VLA skills and observation uncertainty caused by limited viewpoints, occlusions, hidden interior spaces, and VLM grounding errors.

The five evaluation tasks are designed with increasing difficulty[2]:

- *push_box*: "push the black box to the top of the yellow drawer." A 1-step task without partial observability or action constraints, evaluating basic grounding and low-level execution capabilities.
- *block_in_drawer*: "put one block on top of one drawer, and close all the drawers." The drawer interiors are initially unobservable, while grasping blocks inside drawers exhibits non-negligible failures.
- *cup_with_water*: "place all cups with dark liquid on the top of the green drawer, and keep all cups without dark liquid on the table." Cup contents are initially hidden, requiring active perception. A black box occupies the target placement area of cups, requiring constraint resolution.
- *cups_then_blocks*: "put all blocks in the same drawer, and close all drawers." A long-horizon task with an average horizon of 7.6 actions under perfect execution. Opening drawers requires first removing cups that block the front, introducing sequential action constraints.
- *box_cups_blocks*: same task as *cups_then_blocks*, but additionally requiring cups to be placed on top of the green drawer, while the black box initially occupies the target placement area, introducing an additional constraint-resolution step. The average horizon is 8.7 actions under perfect execution.

[2]For each evaluation task, we assess 10 trials.



(a) Task Progress (%) ↑ (b) Cumulative Reward ↑ (c) Planning Time per Step (s) ↓

Figure 3: Performance comparison of our method and baselines. Task 1–5 refer to *push_box*, *block_in_drawer*, *cup_with_water*, *cups_then_blocks* and *box_cups_blocks* respectively. Each bar represents the average value with standard error (SE).

We report three evaluation metrics: *task progress*, defined as the fraction of key task elements accomplished; *cumulative reward*, defined by the PO-PDDL reward function; and *planning time*, measured in seconds. The reward function assigns a $+500$ reward for successful task completion, a -100 penalty for falsely declaring task completion, a -300 penalty for irreversible failures such as knocking over a cup containing liquid, and a time cost that accounts for both online planning latency and action-execution time. All evaluated methods consistently use GPT-5.4 [33] as the LLM/VLM backend. See Appendix A.2 for additional details.

6.2 Baselines

We compare against both explicit model-learning approaches and direct VLM-based planning with history conditioning and in-context learning from demonstrations.

- *VLM-Planner*: a VLM directly performs online planning under uncertainty by selecting the next action based on the current visual observation, conditioned on the execution history and language demonstrations (the same demonstrations used for our domain learning).
- *UniDomain** [5]: learns a deterministic PDDL domain from the same visual demonstrations. We adapt its planning module to online planning, by regenerating the PDDL problem at each time step from the current visual observation and solving it with Monte Carlo tree search.
- *pomdp_coder** [31]: learns a POMDP model represented as Python code, from language-based demonstrations without visual inputs. We adapt it to our setting by applying our visual grounding pipeline to construct symbolic demonstrations and using our belief initialization and belief update modules for online execution. See Appendix A.3 for implementation details.

6.3 Comparison Results

Domain Learning Results. During domain learning, our method learns a correct and complete POMDP model with the lowest learning cost among all explicit model-learning approaches. PO-PDDL takes 291 seconds to learn from 29 demonstrations containing 103 execution steps, while *UniDomain** and *pomdp_coder** require 5111 and 328 seconds, respectively. See Appendix A.4 for detailed learning-time and token-cost statistics. Appendix A.5 further compares the correctness and completeness of the learned models.

Online Planning Results. we first evaluate whether our learned domain supports effective planning on the demonstrated tasks. The generated plans achieve 100% task completion, cover all essential actions in the demonstrations, and require 0.2 fewer execution steps on average, indicating that the learned model captures the symbolic structure needed for planning.

As shown in Figure 3, PO-PDDL consistently outperforms all baselines across all task settings.

VLM-Planner performs reasonably well on *push_box* (Task 1) and *block_in_drawer* (Task 2), where constraint reasoning is limited. However, it struggles on *cup_with_water* (Task 3), which introduces action constraints and hidden states, and completely fails on the long-horizon tasks *cups_then_blocks* (Task 4) and *box_cups_blocks* (Task 5). The VLM often fails to infer the underlying action constraints from visual observations and demonstrations, leading to infeasible actions.

UniDomain* performs well on *push_box* (Task 1), where uncertainty is minimal and deterministic symbolic planning is sufficient. It achieves only marginal success on *block_in_drawer* (Task 2) and *cup_with_water* (Task 3), where partial observability and stochastic execution become important. It completely fails on *cups_then_blocks* (Task 4) and *box_cups_blocks* (Task 5), because stochastic execution outcomes and varying pre-states produce inconsistent symbolic representations for the same action, resulting in unsolvable PDDL domains.

pomdp_coder* is generally incapable of modeling complex visual robot domains, and fails on all tasks except the single-step *push_box* (Task 1). Although its learned transition and observation programs achieve high coverage of the demonstrated reality, combining them into a globally consistent and solvable POMDP model remains difficult.

In contrast, PO-PDDL successfully solves all tasks, achieving 100% task success. The strong performance is attributed to: (1) explicitly modeling action constraints through symbolic operator preconditions; (2) explicitly modeling partial observability and stochastic execution; (3) using an LLM-friendly symbolic formulation for POMDPs; and (4) applying extensive closed-loop verification and revision during domain learning to ensure model correctness and consistency.

Notably, PO-PDDL also achieves the shortest planning time, with an average improvement of 50.4% over *VLM-Planner*, 85.7% over *UniDomain**, and 73.2% over *pomdp_coder**. This efficiency comes from the fact that most steps only require Bayesian belief updates using the learned transition models. Only actions with observation models require reformulation of the belief with VLMs.

Prompts of PO-PDDL and all baselines, as well as qualitative results, can be found on the project website: `https://roboticsjtu.github.io/PO-PDDL`.

6.4 Ablation Studies

We assess the contribution of our key design choices using the following variants:

- *Open-Pass*: instead of separating *fully observable domain learning* from *partially observable domain learning*, this variant performs joint learning of both components in a single stage during symbolic grounding and trajectory reconstruction.
- *No-ActiveObs*: removes the distinction between manipulation operators and active-perception operators during observation modeling. All observations are modeled as passive observations.

As shown in Figure 4, both variants exhibit substantially degraded performance.

Open-Pass fails to reliably distinguish fully observable predicates from partially observable predicates, often introducing redundant observables for predicates that provide useless or even destructive information. This produces inefficient belief updates. The issue is particularly severe in *cup_with_water* (Task 3), where unfocused observation modeling corrupts the belief over hidden cup contents, causing planning failures.

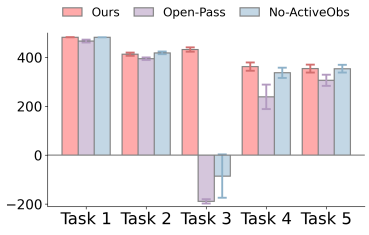


Figure 4: Ablation study results. Values denote cumulative reward.

No-ActiveObs under-models active-perception observations. Since passive observations are often consistent with inferred hidden states, the learned model misses the information-gathering role of active-perception operators. In *cup_with_water* (Task 3), this prevents newly observed cup contents from updating the belief, leading to poor task performance.

Beyond these ablations, we further analyze the benefit of online adaptation of operator transition probabilities. See results in Appendix A.6.

7 Conclusion and Limitations

We presented PO-PDDL, a symbolic POMDP formulation that extends PDDL with stochastic transitions, partial observability, and belief representations, while preserving its LLM-friendly nature. Building on this formulation, we developed a demonstration-driven pipeline that learns reusable PO-PDDL domains directly from robot execution videos and supports online belief-space planning. Experiments on real-world manipulation tasks show that our method consistently outperforms

existing PDDL and POMDP model-learning approaches, achieving substantially stronger planning performance with much lower learning and planning time.

Despite these results, several limitations remain. First, our framework assumes a closed domain without post-demonstration domain shift. Extending PO-PDDL to open-world environments requires online revision of the symbolic POMDP model from new visual observations and interactions. Second, learning cost may grow with large-scale demonstration datasets. A promising direction is to build reusable knowledge bases of modular PO-PDDL domains that can be adapted across scenes and tasks, similar to UniDomain. Finally, system performance remains bounded by the capability and biases of the underlying VLM. Future systems may explicitly identify such bottlenecks and request human assistance when needed.

Acknowledgment

This work was supported in part by the National Key R&D Program of China (Grant No. 2024YFB4707600) and NSFC under grant No. 62303304.

References

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

[2] Physical Intelligence, Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, et al. $\pi_{0.5}$: A vision-language-action model with open-world generalization. *arXiv preprint arXiv:2504.16054*, 2025.

[3] Constructions Aeronautiques, Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins Sri, Anthony Barrett, Dave Christianson, et al. Pddl—the planning domain definition language. *Technical Report, Tech. Rep.*, 1998.

[4] Håkan LS Younes and Michael L Littman. Ppddl1. 0: An extension to pddl for expressing planning domains with probabilistic effects. *Techn. Rep. CMU-CS-04-162*, 2(99):12, 2004.

[5] Haoming Ye, Yunxiao Xiao, Cewu Lu, and Panpan Cai. Unidomain: Pretraining a unified pddl domain from real-world demonstrations for generalizable robot task planning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

[6] POMDPX File Format (Version 1.0). `https://bigbird.comp.nus.edu.sg/pmwiki/farm/appl/index.php?n=Main.PomdpXDocumentation`, 2009. National University of Singapore.

[7] Yoonwoo Kim, Raghav Arora, Roberto Martín-Martín, Peter Stone, Ben Abbatematteo, and Yoonchang Sung. Large-language-model-guided state estimation for partially observable task and motion planning. *arXiv preprint arXiv:2603.03704*, 2026.

[8] Scott Sanner et al. Relational dynamic influence diagram language (rddl): Language description. *Unpublished ms. Australian National University*, 32(27):12, 2010.

[9] Lingfeng Sun, Devesh K Jha, Chiori Hori, Siddarth Jain, Radu Corcodel, Xinghao Zhu, Masayoshi Tomizuka, and Diego Romeres. Interactive planning using large language models for partially observable robotic tasks. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 14054–14061. IEEE, 2024.

[10] Allen Z Ren, Anushri Dixit, Alexandra Bodrova, Sumeet Singh, Stephen Tu, Noah Brown, Peng Xu, Leila Takayama, Fei Xia, Jake Varley, et al. Robots that ask for help: Uncertainty alignment for large language model planners. *arXiv preprint arXiv:2307.01928*, 2023.

[11] James F Mullen Jr and Dinesh Manocha. Towards robots that know when they need help: Affordance-based uncertainty for large language model planners. *arXiv preprint arXiv:2403.13198*, 3, 2024.

[12] Connie Jiang, Yiqing Xu, and David Hsu. Llms for robotic object disambiguation. *arXiv preprint arXiv:2401.03388*, 2024.

[13] Kangrui Wang, Pingyue Zhang, Zihan Wang, Yaning Gao, Linjie Li, Qineng Wang, Hanyang Chen, Chi Wan, Yiping Lu, Zhengyuan Yang, et al. Vagen: Reinforcing world model reasoning for multi-turn vlm agents. *arXiv preprint arXiv:2510.16907*, 2025.

[14] Zirui Zhao, Wee Sun Lee, and David Hsu. Large language models as commonsense knowledge for large-scale task planning. *Advances in neural information processing systems*, 36:31967–31987, 2023.

[15] Wenjing Tang, Xinyu He, Yongxi Huang, Yunxiao Xiao, Cewu Lu, and Panpan Cai. Tru-pomdp: Task planning under uncertainty via tree of hypotheses and open-ended pomdps. *arXiv preprint arXiv:2506.02860*, 2025.

[16] Qianwei Wang, Yifan Xu, Vineet Kamat, and Carol Menassa. A robust framework for open-vocabulary multi-object search with visual-language understanding. In *ICRA 2025 Workshop on Foundation Models and Neuro-Symbolic AI for Robotics*.

[17] Emanuele La Malfa, Ping Zhu, Samuele Marro, Sara Bernardini, and Michael Wooldridge. An end-to-end planning framework with agentic llms and pddl. *arXiv preprint arXiv:2512.09629*, 2025.

[18] Qianwei Wang, Bowen Li, Zhanpeng Luo, Yifan Xu, Alexander Gray, Tom Silver, Sebastian Scherer, Katia Sycara, and Yaqi Xie. Unifying deep predicate invention with pre-trained foundation models. *arXiv preprint arXiv:2512.17992*, 2025.

[19] Walker Byrnes, Miroslav Bogdanovic, Avi Balakirsky, Stephen Balakirsky, and Animesh Garg. Climb: Language-guided continual learning for task planning with iterative model building. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8. IEEE, 2025.

[20] Jinbang Huang, Yixin Xiao, Zhanguang Zhang, Mark Coates, Jianye Hao, and Yingxue Zhang. One demo is all it takes: Planning domain derivation with llms from a single demonstration. *arXiv preprint arXiv:2505.18382*, 2025.

[21] Sadegh Mahdavi, Raquel Aoki, Keyi Tang, and Yanshuai Cao. Leveraging environment interaction for automated pddl translation and planning with large language models. *Advances in Neural Information Processing Systems*, 37:38960–39008, 2024.

[22] Zhouliang Yu, Yuhuan Yuan, Tim Z Xiao, Fuxiang Frank Xia, Jie Fu, Ge Zhang, Ge Lin, and Weiyang Liu. Generating symbolic world models via test-time scaling of large language models. *arXiv preprint arXiv:2502.04728*, 2025.

[23] Maximilian Diehl, Chris Paxton, and Karinne Ramírez-Amaro. Automated generation of robotic planning domains from observations. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6732–6738. IEEE, 2021.

[24] Tom Silver, Rohan Chitnis, Nishanth Kumar, Willie McClinton, Tomás Lozano-Pérez, Leslie Pack Kaelbling, and Joshua B. Tenenbaum. Predicate invention for bilevel planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 12120–12129, 2023.

[25] Nishanth Kumar, Willie McClinton, Rohan Chitnis, Tom Silver, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Learning efficient abstract planning models that choose what to predict. In *Proceedings of the 7th Conference on Robot Learning*, volume 229 of *Proceedings of Machine Learning Research*, pages 2070–2095. PMLR, 2023.

[26] Jinbang Huang, Allen Tao, Rozilyn Marco, Miroslav Bogdanovic, Jonathan Kelly, and Florian Shkurti. Automated planning domain inference for task and motion planning. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pages 12534–12540. IEEE, 2025.

[27] Pierrick Lorang, Hong Lu, Johannes Huemer, Patrik Zips, and Matthias Scheutz. Few-shot neuro-symbolic imitation learning for long-horizon planning and acting. In *Proceedings of the 9th Conference on Robot Learning*, volume 305 of *Proceedings of Machine Learning Research*, pages 2501–2518. PMLR, 2025.

[28] Liancheng Gong, Wang Zhu, Jesse Thomason, and Li Zhang. Zero-shot iterative formalization and planning in partially observable environments. *arXiv preprint arXiv:2505.13126*, 2025.

[29] Linfeng Zhao, Willie McClinton, Aidan Curtis, Nishanth Kumar, Tom Silver, Leslie Pack Kaelbling, and Lawson LS Wong. Seeing is believing: Belief-space planning with foundation models as uncertainty estimators. *arXiv preprint arXiv:2504.03245*, 2025.

[30] Ziwei Deng, Mian Deng, Chenjing Liang, Zeming Gao, Chennan Ma, Chenxing Lin, Haipeng Zhang, Songzhu Mei, Siqi Shen, and Cheng Wang. Planu: Large language model reasoning through planning under uncertainty. *arXiv preprint arXiv:2510.18442*, 2025.

[31] Aidan Curtis, Hao Tang, Thiago Veloso, Kevin Ellis, Joshua B Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Llm-guided probabilistic program induction for pomdp model estimation. In *Conference on Robot Learning*, pages 3137–3184. PMLR, 2025.

[32] Nan Ye, Adhiraj Somani, David Hsu, and Wee Sun Lee. DESPOT: Online POMDP planning with regularization. *Journal of Artificial Intelligence Research*, 58:231–266, 2017.

[33] OpenAI. Gpt-5.4 thinking deployment safety report. Technical report, OpenAI, April 2026. Accessed: May 1, 2026.

[34] OpenAI. OpenAI API Pricing. `https://developers.openai.com/api/docs/pricing`, 2026. Accessed: May 4, 2026.

A Technical appendices

A.1 Example PO-PDDL Formulation: Tiger Problem

We provide a complete PO-PDDL formulation based on the classic Tiger problem, consisting of a domain file and a problem file. In this problem, an agent faces two doors, with a tiger hidden behind one door and a reward behind the other. The agent can either listen, which provides a noisy observation about the tiger's location, or open one of the doors. Opening the door with the tiger incurs a large penalty, while opening the other door gives a positive reward. After opening a door, the tiger location is reset randomly.

The domain file defines predicates, observable schemas, stochastic operators, rewards, and symbolic observation rules. The problem file specifies the objects, initial belief over hidden symbolic states, goal condition, and optimization metric. Together, the two files instantiate a complete PO-PDDL planning model.

```
(define (domain tiger-doors-pomdp)
  (:requirements :strips :typing :probabilistic-effects
                 :conditional-effects :fluents :pomdp)

  (:types
    door
    action-tag
  )

  (:constants
    ;; Spatial convention: two doors arranged left-to-right.
    ;; Names: d_left = left door, d_right = right door.
    d_left d_right - door

    act_listen act_open - action-tag
  )

  (:predicates
    ;; Hidden state: where the tiger is.
    (tiger-at ?d - door)

    ;; Minimal history tag to condition observations on whether we listened last.
    ;; "no-history-yet" is the explicit initial case (no previous action recorded).
    (last-action ?a - action-tag)
    (no-history-yet)
  )

  (:observables
    (obs-hear ?d - door)
    (obs-nothing)
  )

  (:functions
    (total-reward)
  )

  (:action listen
    :parameters ()
    :precondition ()
    :effect (and
      (decrease (total-reward) 1)

      ;; record last action
      (not (no-history-yet))
      (last-action act_listen)
      (not (last-action act_open))
    )
  )

  (:action open
    :parameters (?d - door)
    :precondition ()
    :effect (and
      ;; reward depends on whether tiger is behind the opened door
      (when (tiger-at ?d)
        (decrease (total-reward) 100))
      (when (not (tiger-at ?d))
        (increase (total-reward) 10))

      ;; after opening, reset tiger location uniformly (classic Tiger POMDP reset)
```

```
        (not (tiger-at d_left))
        (not (tiger-at d_right))
        (probabilistic
          0.5 (and (tiger-at d_left) (not (tiger-at d_right)))
          0.5 (and (tiger-at d_right) (not (tiger-at d_left)))
        )

        ;; record last action
        (not (no-history-yet))
        (not (last-action act_listen))
        (last-action act_open)
      )
    )

    ;; Observation model: only meaningful after listening; 85% accurate, 15% flipped.
    (:observation hear-when-tiger-left
      :parameters ()
      :condition (and (last-action act_listen) (tiger-at d_left))
      :distribution (probabilistic
        0.85 (obs-hear d_left)
        0.15 (obs-hear d_right)
      )
    )

    (:observation hear-when-tiger-right
      :parameters ()
      :condition (and (last-action act_listen) (tiger-at d_right))
      :distribution (probabilistic
        0.85 (obs-hear d_right)
        0.15 (obs-hear d_left)
      )
    )
  )
```

Listing 1: PO-PDDL domain file for the Tiger problem.

```
  (define (problem tiger-doors-instance)
    (:domain tiger-doors-pomdp)

    (:objects
      ;; Uses domain constants d_left, d_right for the two doors.
    )

    (:init-belief
      ;; Group: tiger-location
      (joint 0.5 (and (tiger-at d_left) (not (tiger-at d_right))))
      (joint 0.5 (and (not (tiger-at d_left)) (tiger-at d_right)))
    )


    (:goal
      ;; Any open action terminates the episode in many formulations; here we model it
      ;; as achieving "last action was open".
      (last-action act_open)
    )

    (:metric maximize (total-reward))
  )
```

Listing 2: PO-PDDL problem file for the Tiger problem.

A.2 Experimental Setup Details

Demonstration Dataset. We use 29 short-horizon demonstrations in total, covering five subtask types:

- Black-box pushing followed by cup placement: “push the black box to the yellow drawer on the left, and place the {color} cup on the top of the green drawer on the right.” Each demonstration has 3 high-level steps; 4 demonstrations.
- Cup placement by liquid content: “place the cup with/without dark liquid on the top of the green drawer on the right.” Each demonstration has 3–4 high-level steps; 8 demonstrations.
- Cup placement followed by drawer opening: “place the {color} cup on the top of the green drawer on the right, and open the {color} drawer.” Each demonstration has 3 high-level steps; 4 demonstrations.

- Block transfer: “pick the {color} block from the {color} drawer and place it into/on top of the {color} drawer on the left.” Each demonstration has 3–6 high-level steps; 11 demonstrations. This group includes pick-block failures.

- Drawer closing: “close the {color} drawer.” 2 demonstrations.

VLA policy and subtask instructions. We post-trained a $\pi_{0.5}$ model as the last-mile VLA policy for executing manipulation and active-perception skills. All high-level subtask instructions used by the policy are listed below.

```
TASK_LIST: list[str] = [
    "open the yellow drawer",
    "open the green drawer",
    "close the yellow drawer",
    "close the green drawer",
    "pick up the pink cup on the left of the table",
    "pick up the pink cup on the right of the table",
    "pick up the green cup on the left of the table",
    "pick up the green cup on the right of the table",
    "pick up the pink cup on the top of the green drawer",
    "pick up the green cup on the top of the green drawer",
    "place the pink cup on the top of the green drawer",
    "place the green cup on the top of the green drawer",
    "place the pink cup on the left of the table",
    "place the pink cup on the right of the table",
    "place the green cup on the left of the table",
    "place the green cup on the right of the table",
    "pick up the blue block in the yellow drawer",
    "pick up the blue block in the green drawer",
    "pick up the yellow block in the yellow drawer",
    "pick up the yellow block in the green drawer",
    "pick up the green block in the yellow drawer",
    "pick up the green block in the green drawer",
    "pick up the red block in the yellow drawer",
    "pick up the red block in the green drawer",
    "place the blue block into the yellow drawer",
    "place the blue block into the green drawer",
    "place the yellow block into the yellow drawer",
    "place the yellow block into the green drawer",
    "place the green block into the yellow drawer",
    "place the green block into the green drawer",
    "place the red block into the yellow drawer",
    "place the red block into the green drawer",
    "place the blue block on the top of the yellow drawer",
    "place the blue block on the top of the green drawer",
    "place the yellow block on the top of the yellow drawer",
    "place the yellow block on the top of the green drawer",
    "place the green block on the top of the yellow drawer",
    "place the green block on the top of the green drawer",
    "place the red block on the top of the yellow drawer",
    "place the red block on the top of the green drawer",
    "push the black box on the green drawer to the yellow drawer",
    "look into the cup on the left",
    "look into the cup on the right",
    "look into the yellow drawer on the left",
    "look into the green drawer on the right",
]
```

Evaluation Tasks and Progress Metrics. For each evaluation task, we set a maximum number of execution steps and define a sequence of key progress subtasks for computing task progress. Task progress is measured as the fraction of key progress subtasks achieved during an episode. Unless otherwise specified, these subtasks are checked in the listed order: a later subtask is counted only if all preceding subtasks have been achieved. Subtasks enclosed in braces can be achieved in any order.

- *push_box*. Max steps: 5. Key progress subtasks: push the black box to the yellow drawer; report goal completion.

- *block_in_drawer*. Max steps: 10. Key progress subtasks: open one of the drawers; pick one of the blocks; place the same block on top of one of the drawers; close one of the drawers; report goal completion.

- *cup_with_water*. Max steps: 10. Key progress subtasks: {look into one of the cups; push the black box}; pick one of the cups; place the same cup on the green drawer; report goal completion.

- *cups_then_blocks*. Max steps: 15. Key progress subtasks: pick one of the cups; place the same cup on one of the drawers; open one of the drawers; pick one of the blocks; place the same block in one of the drawers; close one of the drawers; report goal completion.
- *box_cups_blocks*. Max steps: 15. Key progress subtasks: push the black box to the yellow drawer; pick one of the cups; place the same cup on one of the drawers; open one of the drawers; pick one of the blocks; place the same block in one of the drawers; close one of the drawers; report goal completion.

Initial scene setup. Figure 5 shows the initial scene configurations for the five evaluation tasks. For each task, we evaluate every method over 10 real-world trials. Across trials, the positions of task-irrelevant distractor objects are randomized. This protocol tests whether each method can robustly solve the same high-level task under small variations in the physical scene, rather than overfitting to a fixed object arrangement.



(a) *push_box* (b) *block_in_drawer* (c) *cup_with_water* (d) *cups_then_blocks* (e) *box_cups_blocks*

Figure 5: Real-world experimental setup.

The task-relevant objects consist of one black box, two blocks (blue and red), two drawers (green and yellow), and two cups (pink and green). For each task:

- *push_box*: Both drawers are initially closed, and the black box is placed on top of the green drawer.
- *block_in_drawer*: Both drawers are initially closed, and the drawer containing each block is randomized across trials.
- *cup_with_water*: Both drawers are initially closed and the black box is placed on top of the green drawer, while the positions of the two cups are swapped across trials.
- *cups_then_blocks*: Both drawers are initially closed, the drawer containing each block is randomized, and the cup positions are swapped across trials.
- *box_cups_blocks*: Both drawers are initially closed and the black box is placed on top of the green drawer, while both the block locations and cup positions are varied across trials.

A.3 Modifications for *pomdp_coder** Baseline

A.3.1 Domain-specific Python interface.

The original *pomdp_coder* framework learns POMDP components as short probabilistic programs over a domain-specific Python API that defines the state, action, and observation spaces. Therefore, to adapt *pomdp_coder* to our real-world tabletop manipulation domain, we implement a new Python interface rather than using the APIs from its original benchmark domains. Our interface defines task-native symbolic states, grounded action payloads, symbolic observation payloads, and runtime functions for sampling initial states, transitions, observations, and rewards. This allows *pomdp_coder* to generate candidate probabilistic programs for our learned manipulation domain while keeping the same high-level model-learning and belief-space planning procedure as the original method.

```
"""Formal Python interface for learned ‘pomdp_coder‘ POMDP models."""

from pathlib import Path
from typing import Any, TypedDict


OBJECT_TYPES_FORMAT = {
    "<type_name>": ["<object_name>", "..."],
}


ACTION_SCHEMA_FORMAT = [
```

```
    {
        "name": "<action_name>",
        "parameters": [
            {
                "name": "<parameter_name>",
                "type": "<type_name>",
            }
        ],
    }
]


SPECIAL_ACTION_SCHEMA_FORMAT = [
    {
        "name": "<special_action_name>",
        "parameters": [],
    }
]


STATE_FIELDS_FORMAT = (
    "<state_field_name>",
    "...",
)


OBSERVATION_FIELDS_FORMAT = (
    "<observation_field_name>",
    "...",
)


class ActionPayload(TypedDict):
    """Grounded symbolic action payload."""

    name: str
    args: dict[str, str]


class ObservationPayload(TypedDict):
    """Symbolic observation payload."""

    positive: list[str]
    negative: list[str]
    metadata: dict[str, Any]


class SymbolicState(TypedDict):
    """Task-native symbolic latent state."""

    # Replace these keys with the task-specific fields induced by the task
    # interface. The actual state schema is not fixed globally.
    ...


GOAL_INTERFACE_FORMAT = {
    "type": "<goal_representation_type>",
    "predicate_templates": [
        "<predicate_template>",
        "...",
    ],
}


def sample_initial_state(rng) -> SymbolicState:
    """Sample an initial latent symbolic state."""
    raise NotImplementedError


def sample_next_state(
    state: SymbolicState,
    action: ActionPayload,
    rng,
) -> SymbolicState:
    """Sample the next latent symbolic state."""
    raise NotImplementedError


def sample_observation(
    next_state: SymbolicState,
    action: ActionPayload,
```

```
        rng,
    ) -> ObservationPayload:
        """Sample a symbolic observation emitted after the transition."""
        raise NotImplementedError


    def reward(
        state: SymbolicState,
        action: ActionPayload,
        next_state: SymbolicState,
    ) -> float:
        """Return the scalar reward for the transition."""
        raise NotImplementedError
```

A.3.2 Adaptation Pipeline.

We adapt *pomdp_coder* to our manipulation benchmark by preserving its core model-learning principle, while replacing the original benchmark-specific inputs with structured artifacts produced from our visual demonstrations. The original *pomdp_coder* framework learns the initial-state, transition, observation, and reward components as probabilistic programs, evaluates candidate programs by empirical coverage, and repairs uncovered cases through feedback. In our setting, the key adaptation is therefore to convert our robot demonstrations into the structured symbolic form required by *pomdp_coder*.

(a) **Scene-described demonstrations.** We start from the same demonstrations used by our method. Each episode is normalized into an annotated trajectory, where every step is associated with a high-level subtask instruction, execution success or failure, and a textual scene description generated from the robot videos. These scene descriptions bridge raw visual inputs and the structured symbolic supervision required by *pomdp_coder*, mirroring their role in our own pipeline for hidden-state reconstruction and visual-state comparison. Prompting designs are available in the prompt library at `https://roboticsjtu.github.io/PO-PDDL`.

(b) **Structured data generation.** We convert the scene-described demonstrations into a symbolic interface and grounded transition records. The interface defines object types, state fields, action schemas, observation fields, goal representation, reward interface, and special actions, serving as the domain-specific API over which *pomdp_coder* synthesizes programs. Each annotated episode is then grounded into records containing symbolic pre-state, grounded action, action outcome, symbolic observation, post-state, reward, terminal flag, and metadata. Finally, we construct component-specific specifications for the initial-state, transition, observation, and reward programs, which define the input-output contracts for candidate programs.

(c) **Candidate program proposal.** Given the structured interface and grounded records, we follow *pomdp_coder*'s proposal stage. The LLM synthesizes multiple candidate Python programs for each POMDP component: the initial-state model, transition model, observation model, and reward model. Each proposal is conditioned on the shared symbolic interface, the component-specific specification, execution rules, a starter template, and a representative subset of grounded training examples. As in our method, the reward component is computed from the empirical average execution time of each action. Using subsets keeps the synthesis problem tractable and encourages compact probabilistic programs rather than memorization of individual demonstrations.

(d) **Coverage-based evaluation and repair.** We retain *pomdp_coder*'s coverage-based evaluation and repair loop. Candidate programs are executed on grounded symbolic records and evaluated by whether they can reproduce empirical outcomes. For stochastic components, coverage is estimated by repeated sampling: a record is covered if the program can generate the observed symbolic outcome within a fixed Monte Carlo budget. Uncovered examples are converted into mismatch feedback containing the symbolic inputs, expected empirical output, and sampled or predicted output, which is then used by the LLM to repair the full program. The final program is selected by development-set coverage, with mismatch count as a secondary criterion.

(e) **Runtime integration.** After learning the four component programs, we assemble them into a runtime POMDP model and use it in the same online execution setting as our method. The planner maintains a particle-based belief state, propagates particles through the learned transition program, and reweights them using the learned observation likelihood after receiving observations. For a fair comparison, the adapted *pomdp_coder* baseline uses the same Bayesian

belief update and VLM-based observation module as our method. The observation module is invoked only when the learned observation model indicates that the current action may produce informative observations under the current belief. We also retain the Belief-Space Planner used in the original *pomdp_coder* framework as the POMDP solver.

A.4 Domain-Learning Cost Comparison

Domain Learning Cost for Our Method and Explicit Baselines. Table 1 reports the offline domain-learning cost for our method and explicit model-learning baselines on the same 29-demonstration dataset with 103 total high-level steps. Our method achieves the lowest learning time while maintaining a comparable token cost.

Table 1: Domain-learning Cost Comparison.

Method	Total Time (s)	LLM/VLM Calls	Cost-Equiv. Tokens	Cost-Equiv. Tokens / Step
Ours	291.00	541	363,247	2,752
pomdp_coder*	328.46	201	237,042	1,796
UniDomain*	5111.19	932	355,253	2,691

Cost-equivalent tokens are output-token-equivalent counts computed using the GPT-5.4 API input/output price ratio [34]: $N_{\mathrm{eq}} = N_{\mathrm{out}} + \frac{1}{6} N_{\mathrm{in}}$. In Tokens / Step, steps include step 0, i.e., the initial observation step.

Table 1 shows that PO-PDDL has the lowest domain learning time among the explicit model-learning methods. *UniDomain** is substantially slower because its original pipeline extracts atomic domains from demonstrations and then repeatedly fuses them into a unified domain through hierarchical domain merging. This pairwise merging process requires many LLM/VLM calls and is difficult to parallelize, making it costly in our setting.

pomdp_coder* has a simpler learning structure and fewer total LLM/VLM calls. For fairness, our adaptation also parallelizes many preprocessing and candidate-generation steps, resulting in a slightly higher learning time than ours but lower token usage. However, its weakly constrained Python representation and coverage-based repair do not produce a reliable model for our long-horizon visual manipulation tasks.

PO-PDDL maintains a reasonable token cost while substantially improving model quality through its structured representation, two-stage domain learning, and explicit observation modeling.

A.5 Learned Model Analysis

We analyze the learned models through the six induced POMDP components $\langle S, A, O, T, Z, R \rangle$. The reference is a set of human-verified symbolic requirements derived from the task specification, physical constraints, demonstration annotations, and held-out execution traces. We use this analysis to relate component-level modeling differences to the observed planning behaviors.

A.5.1 Learned PO-PDDL Model from Our Method

The learned PO-PDDL model from our method satisfies the required components for the evaluation tasks: it contains task-relevant state and action abstractions, encodes key inter-action constraints (i.e., causal action-ordering constraints), represents stochastic effects, and defines compact action-conditioned observation rules. These components support long-horizon composition under action constraints, stochastic execution, and partial observability, beyond simply covering the short-horizon demonstrations.

State and action spaces. The learned state space contains the task-relevant predicates needed to describe object locations, drawer states, gripper states, hidden cup contents, blocking relations, and left/right spatial structure, without introducing redundant overlapping representations. For example, instead of using generic location fields, the learned PO-PDDL model uses typed and side-specific predicates such as `in_front_of_left`, `in_front_of_right`, `on_left`, `on_right`, and side-specific drawer-top predicates. The action space is also side-specific, with separate operators for left-side and right-side manipulation, preserving transition differences caused by side-dependent VLA execution.

```
(:predicates
  (contains_dark_liquid ?x0 - cup)
  (gripper_empty)
  (gripper_holding_block ?x0 - block)
  (gripper_holding_cup ?x0 - cup)

  (in ?x0 - drawer ?x1 - block)
  (in_front_of_left ?x0 - cup ?x1 - drawer)
  (in_front_of_right ?x0 - cup ?x1 - drawer)

  (on_left ?x0 - fixed_item)
  (on_right ?x0 - fixed_item)

  (on_top_of_left_block_drawer ?x0 - block ?x1 - drawer)
  (on_top_of_left_box_drawer ?x0 - box ?x1 - drawer)
  (on_top_of_right_box_drawer ?x0 - box ?x1 - drawer)
  (on_top_of_right_cup_drawer ?x0 - cup ?x1 - drawer)

  (open ?x0 - drawer)
)

(:action open_object_on_left ...)
(:action open_object_on_right ...)
(:action pick_up_object_in_object_on_left ...)
(:action pick_up_object_in_object_on_right ...)
(:action place_object_on_top_of_object_on_left ...)
(:action place_object_on_top_of_object_on_right ...)
```

Listing 3: Structured state and action space in the learned PO-PDDL model.

This state-action design satisfies the required side-specific spatial distinctions and avoids redundant encodings of the same physical fact. The resulting state space is compact, interpretable, and sufficient for grounding the POMDP state space S and action space A.

Preconditions. The learned preconditions cover both local executability and verified inter-action constraints. These include constraints such as "a drawer cannot be opened if a cup blocks its front" and "a cup cannot be placed on a drawer top if the black box already occupies that region." For example, the learned `open_object_on_left` operator requires the drawer to be on the left, closed, gripper-empty, and free from blocking cups or objects on relevant drawer-top regions.

```
(:action open_object_on_left
  :parameters (?arg0 - drawer)
  :precondition
  (and
    (gripper_empty)
    (not (on_right ?arg0))
    (not (open ?arg0))
    (on_left ?arg0)

    ;; no object held
    (forall (?b - block)
      (not (gripper_holding_block ?b)))
    (forall (?c - cup)
      (not (gripper_holding_cup ?c)))

    ;; no cup blocks the drawer front
    (forall (?c - cup)
      (not (in_front_of_left ?c ?arg0)))
    (forall (?c - cup)
      (not (in_front_of_right ?c ?arg0)))

    ;; no object blocks the relevant drawer-top region
    (forall (?b - block)
      (not (on_top_of_left_block_drawer ?b ?arg0)))
    (forall (?c - cup)
      (not (on_top_of_right_cup_drawer ?c ?arg0)))
  )
)
```

Listing 4: Precondition encoding drawer-opening constraints.

This example shows that the learned preconditions encode causal ordering constraints required by long-horizon tasks, rather than only local action applicability. Similarly, the learned

`place_object_on_top_of_object_on_right` operator excludes boxes on the target drawer top, capturing the need to push the black box away before cup placement.

Effects and stochastic effects. The learned effects cover the relevant action outcomes observed in the demonstrations and required by the evaluation tasks. The model supports probabilistic effects by separating different outcomes of the same action into explicit effect modes. For example, picking a block from the left drawer has both a success mode and a failure mode. In the success mode, the robot holds the block and the block is removed from the drawer; in the failure mode, the state does not incorrectly assert that the robot holds the block.

```
(:action pick_up_object_in_object_on_left
  :parameters (?arg0 - block ?arg1 - drawer)
  :precondition
  (and
    (gripper_empty)
    (in ?arg1 ?arg0)
    (not (gripper_holding_block ?arg0))
    (not (on_right ?arg1))
    (on_left ?arg1)
    (open ?arg1)
  )
  :effect
  (and
    (decrease (total-reward) 23.536364)
    (probabilistic
      ;; success mode
      0.636364
      (and
        (gripper_holding_block ?arg0)
        (not (gripper_empty))
        (not (in ?arg1 ?arg0)))

      ;; failure mode
      0.363636
      (and
        ;; the block remains in place and the gripper remains empty
      )
    )
  )
)
```

Listing 5: Stochastic effect modes for block picking.

The learned effects also maintain state consistency through explicit add/delete updates. For example, the learned box-pushing operator removes the box from the right drawer top and adds it to the left drawer top, rather than updating only one redundant location field.

```
(:action push_object_on_object_on_right_to_object_on_left
  :parameters (?arg0 - box ?arg1 - drawer ?arg2 - drawer)
  :effect
  (and
    (decrease (total-reward) 21.925)
    (probabilistic
      1.000000
      (and
        (on_top_of_left_box_drawer ?arg0 ?arg2)
        (not (on_top_of_right_box_drawer ?arg0 ?arg1)))
    )
  )
)
```

Listing 6: Consistent location update for pushing the box.

These examples show that the learned transition model T is stochastic and state-consistent. It represents VLA execution uncertainty through effect probabilities and preserves mutually exclusive symbolic locations through explicit add/delete effects.

Observation space. The learned observation space is compact and aligned with the verified partially observable predicates. Instead of treating all visible facts as observations, the learned model only introduces observable schemas for predicates that are genuinely uncertain or visually unreliable. In the learned domain, the observation space contains only `obs_in` for hidden drawer contents and `obs_contains_dark_liquid` for cup liquid content, plus `obs-nothing`. It does not include fully

observable facts such as gripper state, drawer open/closed state, or ordinary object locations as redundant observations.

```
(:observables
  (obs-nothing)
  (obs_in ?arg0 - drawer ?arg1 - block)
  (obs_contains_dark_liquid ?arg0 - cup)
)
```

Listing 7: Compact learned observation space.

This observation space is small, task-relevant, and aligned with the partially observable predicates needed for belief update.

Observation rules. The learned observation rules are constructed only for observables and actions that can produce informative or uncertain observations. Passive observation rules are generated for action contexts where visual grounding may be unreliable, while active-perception rules are generated for actions whose purpose is to inspect hidden states. For example, the learned model includes rules for the hidden predicate `contains_dark_liquid`: an initial/passive observation rule captures uncertainty from ordinary visual grounding, while an active-perception rule captures the more informative observation produced by `look_into_cup`.

```
(:observation initial_contains_dark_liquid_true
  :parameters (?arg0 - cup)
  :condition
  (and
    (contains_dark_liquid ?arg0))
  :distribution
  (probabilistic
    0.666667 (obs_contains_dark_liquid ?arg0)
    0.333333 (not (obs_contains_dark_liquid ?arg0))
  )
)

(:observation initial_contains_dark_liquid_false
  :parameters (?arg0 - cup)
  :condition
  (and
    (not (contains_dark_liquid ?arg0)))
  :distribution
  (probabilistic
    0.000000 (obs_contains_dark_liquid ?arg0)
    1.000000 (not (obs_contains_dark_liquid ?arg0)))
)
```

Listing 8: Initial observation rule for cup contents.

```
(:observation active_look_into_object_on_left_success_0_contains_dark_liquid_true
  :parameters (?arg0 - cup)
  :condition
  (and
    (last_action_1_param
      look_into_object_on_left_success_0
      ?arg0)
    (contains_dark_liquid ?arg0))
  :distribution
  (probabilistic
    1.000000 (obs_contains_dark_liquid ?arg0)
    0.000000 (not (obs_contains_dark_liquid ?arg0)))
)

(:observation active_look_into_object_on_left_success_0_contains_dark_liquid_false
  :parameters (?arg0 - cup)
  :condition
  (and
    (last_action_1_param
      look_into_object_on_left_success_0
      ?arg0)
    (not (contains_dark_liquid ?arg0)))
  :distribution
  (probabilistic
    0.000000 (obs_contains_dark_liquid ?arg0)
    1.000000 (not (obs_contains_dark_liquid ?arg0)))
)
```

Listing 9: Active-perception observation rule for cup contents.

This example shows that the learned model does not emit generic visual facts as observations. Instead, it defines action-conditioned observation rules for task-relevant hidden predicates. Ordinary visual grounding provides uncertain evidence about `contains_dark_liquid`, while `look_into_cup` produces a more informative observation for the same hidden state. Thus, the learned observation model Z is tied to specific hidden predicates, observable schemas, and actions that can reveal them.

A.5.2 Learned PDDL Domain from *UniDomain**

UniDomain* recovers many manipulation actions and basic predicates, but several induced POMDP components remain incomplete for the required partially observable and stochastic planning problem. It mainly learns a deterministic PDDL model over local action effects, rather than a model that captures partial observability, action uncertainty, and long-horizon inter-action constraints.

State and action spaces. *UniDomain** learns many basic manipulation predicates, such as object locations, drawer open/closed states, gripper state, and cup liquid content. It also recovers the main manipulation actions for opening and closing drawers, picking and placing objects, and pushing the black box. However, the learned state abstraction does not fully match the required symbolic structure: it misses some task-relevant spatial distinctions and introduces redundant location predicates.

UniDomain* lacks explicit left/right localization predicates. Some predicates that should distinguish the left and right sides of the workspace are merged into generic predicates. This also appears in the action schemas, where left-side and right-side operations are often represented by the same action schema. Such merging may be acceptable under deterministic effects, but it loses an important source of transition uncertainty in our setting, because VLA skills can have different success rates on the left and right sides.

```
;; UniDomain-style generic location predicates
(on ?o ?x)
(on_drawer ?o ?d)
(in_drawer ?o ?d)

;; Missing side-specific predicates such as
;; (on_left ?d)
;; (on_right ?d)
;; (on_top_of_left_* ...)
;; (on_top_of_right_* ...)
```

The learned state space also contains redundant location predicates whose effects are not always kept consistent. For example, *UniDomain** may represent drawer-top placement using both `on` and `on_drawer`. However, some actions update only one of them. In the following representative effect, pushing an object updates `on` but not `on_drawer`, so the two predicates can become inconsistent.

```
(:action push_to_drawer
  :parameters (?o ?from ?to)
  :precondition (and
    (drawer ?from)
    (drawer ?to)
    (on ?o ?from)
    (hand_free))
  :effect (and
    (not (on ?o ?from))
    (on ?o ?to))
)

;; If (on_drawer ?o ?from) was also true before the action,
;; it is not removed here, producing inconsistent locations.
```

UniDomain* does not model active perception meaningfully. It learns predicates such as `inspected` and `inspected_inside` for active-perception actions, but these are deterministic action markers. They do not define observable outcomes, observation likelihoods, or belief updates. Therefore, these actions do not provide useful information for planning under partial observability.

```
(:action look_into_cup
  :parameters (?c)
```

```
  :precondition (and (cup ?c))
  :effect (and
    (inspected ?c)
    (inspected_inside ?c))
)

;; This records that inspection happened, but it does not define
;; observations such as obs_contains_dark_liquid or Z(o | s, a).
```

Transition model. The learned transition model from *UniDomain** is incomplete for constrained and stochastic manipulation dynamics. It learns local preconditions and deterministic effects for individual actions, but misses three key properties required by the task dynamics: inter-action constraints, location exclusivity, and stochastic failure modes.

UniDomain* does not recover important inter-action constraints between actions. For example, placing a cup on top of a drawer should require the target drawer-top region to be unblocked. In the physical domain, the black box may occupy the drawer top, so it must be pushed away before cup placement. However, the learned placement action only checks local conditions such as whether the robot is holding the object, and does not encode this blocking constraint.

```
(:action place_on_drawer
  :parameters (?o ?d)
  :precondition (and
    (drawer ?d)
    (holding ?o)
    (not (contains_dark_liquid ?o)))
  :effect (and
    (on ?o ?d)
    (on_drawer ?o ?d)
    (hand_free)
    (not (holding ?o)))
)

;; Missing inter-action constraint:
;; the drawer top should be unblocked before placement.
```

Similarly, opening a drawer should require the space in front of the drawer to be clear, since a cup in front of the drawer can prevent opening. Picking a block from a drawer should also require the drawer to be open. These constraints are necessary for tasks such as *cups_then_blocks* and *box_cups_blocks*, where cups must be moved before drawer opening and drawers must be opened before retrieving blocks. Since *UniDomain** does not recover these causal ordering constraints, its planner may generate action sequences that are valid in the learned PDDL model but infeasible in the physical scene.

The learned effects do not consistently maintain location exclusivity. Object locations are mutually exclusive in the physical domain: after an object is placed into a new location, its previous location should be removed. In contrast, several *UniDomain** effects only add the new location predicate without deleting all potentially conflicting old locations. For example, placing an object on a drawer adds `on` and `on_drawer`, but does not remove possible previous location predicates such as `on_table`, `in_drawer`, or `at_drawer_area`. This can make the same object appear in multiple locations simultaneously in the symbolic state.

```
(:action place_on_drawer
  :parameters (?o ?d)
  :precondition (and
    (drawer ?d)
    (holding ?o)
    (not (contains_dark_liquid ?o)))
  :effect (and
    (on ?o ?d)
    (on_drawer ?o ?d)
    (hand_free)
    (not (holding ?o)))
)

;; Missing location-exclusivity effects, e.g.,
;; (not (on_table ?o ?t))
;; (not (in_drawer ?o ?d2))
;; (not (at_drawer_area ?o ?d2))
```

This problem is amplified by redundant location predicates in the learned state space. For instance, *UniDomain** may represent drawer-top placement using both `on` and `on_drawer`. Some effects update only one of these predicates, which can make the two location representations inconsistent. A representative example is pushing an object from one drawer to another: the effect updates `on`, but does not update `on_drawer`.

```
(:action push_to_drawer
  :parameters (?o ?from ?to)
  :precondition (and
    (drawer ?from)
    (drawer ?to)
    (on ?o ?from)
    (hand_free))
  :effect (and
    (not (on ?o ?from))
    (on ?o ?to))
)

;; If (on_drawer ?o ?from) was true before pushing,
;; it is not removed, and (on_drawer ?o ?to) is not added.
```

UniDomain* uses deterministic PDDL effects and therefore cannot represent VLA execution failures. This is problematic because the demonstrations include failed block-picking attempts, and the real robot may fail even when an action is applicable. In a stochastic model, a pick action should have at least two effect modes: a success mode in which the gripper holds the block, and a failure mode in which the block remains in place and the gripper remains empty. The learned deterministic effect can only encode the success branch.

```
(:action pick_from_drawer_area
  :parameters (?o ?d)
  :precondition (and
    (drawer ?d)
    (at_drawer_area ?o ?d)
    (hand_free))
  :effect (and
    (holding ?o)
    (not (at_drawer_area ?o ?d))
    (not (hand_free)))
)

;; Missing stochastic effect modes:
;; success: holding the object
;; failure: object remains in place and gripper remains empty
```

Thus, *UniDomain** learns a deterministic local transition model rather than the stochastic, constraint-aware, and state-consistent transition model required for the tasks. It can describe some successful action effects in isolation, but it does not preserve mutually exclusive object locations, encode long-horizon action-ordering constraints, or model execution failures.

Reward and planning objective. Because *UniDomain** is based on standard PDDL, it does not learn a reward function or action-cost model comparable to the R component in the induced POMDP. In our online adaptation, we use Monte Carlo tree search with a uniform action prior at each step. The search is performed over deterministic symbolic effects from the learned PDDL model. As a result, *UniDomain** can solve simple fully observable tasks, but it fails to produce robust and cost-effective behavior in long-horizon tasks with action uncertainty and partial observability.

A.5.3 Learned POMDP Model from *pomdp_coder**

Compared with *UniDomain**, *pomdp_coder** explicitly learns Python programs for the initial-state model, transition model, observation model, and reward model. However, the learned model is weakly structured: state variables, action semantics, transition effects, and observations are implemented through dictionaries and procedural rules rather than explicit relational predicates, stochastic operator modes, and action-conditioned observation rules.

State and action spaces. The learned state space is loose and redundant relative to the required symbolic state structure. Instead of complete assignments over grounded predicates, *pomdp_coder** represents states using a small set of dictionaries, such as `object_location`, `drawer_state`, `object_properties`, and `support_occupancy`. This representation can encode many relevant

variables, but it does not enforce relational structure, typing, or mutual-exclusion constraints as explicit symbolic requirements. It also lacks explicit left/right localization predicates. For example, side information is mostly encoded through object names such as `green_drawer` and `yellow_drawer`, rather than predicates such as `on_left`, `on_right`, `in_front_of_left`, and `in_front_of_right`.

```
state = {
    "holding": None,
    "object_location": {},
    "drawer_state": {},
    "object_properties": {},
    "support_occupancy": {
        "table_top": "unknown",
        "green_drawer_top": None,
        "yellow_drawer_top": None,
    },
    "goal_spec": [],
}
```

Listing 10: Learned state representation.

This representation also introduces redundant state variables. For instance, an object on a drawer top can be represented both by `object_location[obj] = "on_top_of(green_drawer)"` and by `support_occupancy["green_drawer_top"] = obj`. Such redundancy requires every transition effect to update multiple fields consistently. A structured symbolic model should maintain such consistency explicitly through predicate-level effects or equivalent constraints. The action space has a similar limitation: as in *UniDomain**, left-side and right-side manipulation are not explicitly separated, which loses transition differences caused by side-dependent VLA execution success rates.

Transition model. The learned transition model partially improves over *UniDomain** in local precondition modeling. For example, it checks whether an object is reachable before picking it, and requires the target drawer to be open before placing an object inside. However, these preconditions are implemented as scattered Python `if` statements. Since the program structure is loose, the LLM must enumerate all necessary spatial, temporal, and causal preconditions procedurally. This is difficult to do completely from short-horizon demonstrations, leading to missing constraints in long-horizon planning.

```
if name == "place_in":
    obj = args.get("obj")
    dst = args.get("dst")
    if obj is None or dst is None:
        return next_state
    if holding != obj:
        return next_state
    if drawer_state.get(dst) != "open":
        return next_state
    _place_in_container(next_state, obj, dst)
    return next_state

if name == "open_container":
    dst = args.get("dst")
    if dst is None:
        return next_state
    drawer_state[dst] = "open"
    _maybe_reveal_object_on_open(next_state, dst)
    return next_state
```

Listing 11: Learned transition preconditions.

The first branch shows that *pomdp_coder** can learn useful local preconditions, such as requiring an open drawer for `place_in`. However, the `open_container` branch directly opens the drawer without checking whether a cup blocks the drawer front. This misses an important inter-action constraint required by *cups_then_blocks* and *box_cups_blocks*: cups must be moved away before the corresponding drawer can be opened.

The learned effects show mixed quality. Compared with *UniDomain**, *pomdp_coder** attempts to maintain location consistency through helper functions such as `_clear_previous_support`. However, this consistency is maintained procedurally and remains incomplete. More importantly, the model lacks a clear effect-mode structure that would guide the learner to separate different outcomes

of the same action, such as success and failure. As a result, action failures are not modeled as explicit stochastic effect modes, even though VLA execution is stochastic.

```
    def _clear_previous_support(next_state, obj):
        support_occupancy = next_state.setdefault("support_occupancy", {})
        object_location = next_state.setdefault("object_location", {})
        old_loc = object_location.get(obj)
        if isinstance(old_loc, str) and old_loc.startswith("on_top_of("):
            container = old_loc[len("on_top_of("):-1]
            top_key = _container_top_key(container)
            if support_occupancy.get(top_key) == obj:
                support_occupancy[top_key] = None

    def _place_on_top(next_state, obj, container):
        _clear_previous_support(next_state, obj)
        next_state["holding"] = None
        next_state.setdefault("object_location", {})[obj] = \
            f"on_top_of({container})"
        next_state.setdefault("support_occupancy", {})[
            _container_top_key(container)] = obj
```

Listing 12: Learned transition effects.

This helper-based implementation is more careful than the deterministic PDDL effects learned by *UniDomain**, but it is still ad hoc. The consistency of `object_location` and `support_occupancy` depends on whether every action calls the right helper and updates all related fields. In addition, some effects mix state transition with information revelation. For example, opening a drawer calls `_maybe_reveal_object_on_open`, which assigns every object with unknown location to the opened drawer. This turns an observation event into a physical state transition, corrupting the latent state rather than updating the belief.

```
    def _maybe_reveal_object_on_open(next_state, container):
        object_location = next_state.setdefault("object_location", {})
        for obj, loc in list(object_location.items()):
            if loc == "unknown":
                object_location[obj] = f"in_container({container})"
```

Listing 13: Information reveal encoded as transition.

Although *pomdp_coder** can express stochastic programs, the learned transition model remains weakly structured. Its preconditions are incomplete because they must be enumerated as Python conditions, and its effects lack an explicit stochastic effect-mode structure for modeling action failures.

Observation space and observation model. *pomdp_coder** has a more complete observation interface than *UniDomain**: it explicitly returns symbolic observation payloads with positive observations, negative observations, and metadata. However, its observation space is overly broad and redundant. It mixes fully observable facts with partially observable facts, including gripper holding state, drawer open/closed state, visible object locations, object properties, and support-clear observations. The required observation space should instead focus on task-relevant partially observable predicates, with each observable tied to a specific hidden predicate.

```
    return {
        "positive": sorted(positive),
        "negative": sorted(negative),
        "metadata": metadata,
    }
```

Listing 14: Learned observation payload.

The learned observation function further illustrates this redundancy. It emits observations for many ordinary visible facts, such as holding state and drawer state, even though these are not necessarily the task-relevant hidden predicates that require belief updates.

```
    # Holding is almost always visible.
    if holding is None:
        positive.add("holding(null)")
        positive.add("holding_null")
        for obj in objects:
            negative.add("holding(%s)" % obj)
    else:
```

```
        positive.add("holding(%s)" % holding)
        for obj in objects:
            if obj != holding:
                negative.add("holding(%s)" % obj)

    # Drawer states are generally observable when known.
    for container in containers:
        ds = drawer_state.get(container, "unknown")
        if _drawer_observable(ds, action_name):
            if ds == "open":
                positive.add("drawer_open(%s)" % container)
                negative.add("drawer_closed(%s)" % container)
            elif ds == "closed":
                positive.add("drawer_closed(%s)" % container)
                negative.add("drawer_open(%s)" % container)
```

Listing 15: Learned observation model.

This observation space lacks an explicit binding between an observable outcome and the hidden predicate it is meant to inform. As a result, the observation model behaves more like a procedural visibility heuristic than a compact action-conditioned likelihood model. It also does not clearly model which observable instances an active-perception action is intended to query. Instead, active perception is mixed with generic visibility rules over object locations, drawer states, and object properties.

```
    # Visible object locations.
    for obj in objects:
        if _visible_object(next_state, obj, action_name):
            visible_objects.append(obj)
            loc = object_location.get(obj, "unknown")
            for atom in _location_atoms(obj, loc):
                positive.add(atom)
            _add_negatives_for_object(negative, obj, loc, containers)

    # Support occupancy sometimes yields clear(top) observations.
    for support_name, occ in support_occupancy.items():
        if isinstance(support_name, str) and support_name.endswith("_top"):
            container = support_name[:-4]
            if occ is None:
                if action_name in ("push_to", "place_on_top", "pick") \
                        or rng.random() < 0.4:
                    positive.add("clear(%s)" % support_name)
                    for obj in objects:
                        negative.add("on_top_of(%s, %s)" % (obj, container))
```

Listing 16: Generic visibility-based observations.

These issues arise from the same source as the transition-model errors: the model is learned as a loose Python program and does not use a two-stage learning pipeline that reconstructs a reliable fully observable domain before identifying truly partially observable predicates. Consequently, *pomdp_coder** tends to produce redundant observations and may invoke VLM-based observation more frequently during online execution.

Reward and planning implications. For the reward component, *pomdp_coder** uses the same empirical average action duration as our method, rather than autonomously inducing a distinct reward model. Thus, the main differences are not in R, but in the learned S, A, T, O, and Z. Overall, *pomdp_coder** learns a valid POMDP-style Python interface and can cover short demonstration records, but its loose state representation, incomplete procedural preconditions, weak effect-mode structure, and redundant observation model make it unreliable for long-horizon belief-space planning. This explains why it can solve the simplest task but generalizes poorly to longer-horizon tasks with action constraints and partial observability.

A.6 Qualitative Experiment: Online Adaptation of Operator Effect Probabilities

Our structured PO-PDDL representation allows operator effect probabilities to be updated during online execution. Since each outcome can be mapped to a symbolic effect mode, the planner can recalibrate transition probabilities from real feedback without relearning the domain structure.

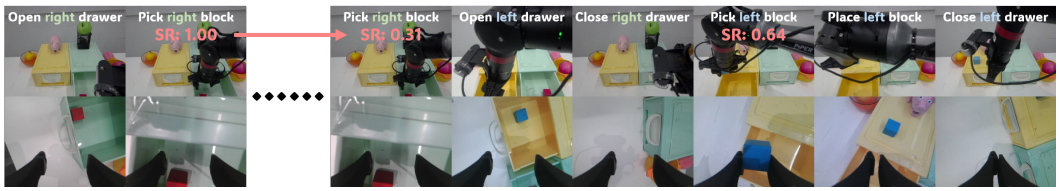


Figure 6: An example of online adaptation on *block_in_drawer*. SR: success rate for the action.

Figure 6 shows a qualitative example on *block_in_drawer*. To simulate test-time skill degradation from domain shift, we make right-drawer grasping unreliable during evaluation. The learned model initially prefers the right drawer, since it learned a lower success probability for left-drawer grasps from demonstrations. After repeated failures, the estimated success probability of right-drawer grasping decreases, and the planner switches to opening the left drawer and grasping the block from there.

The adapted probabilities also provide a diagnostic signal for low-level skill weaknesses. For example, when transferring to a new domain, a skill whose estimated success probability drops repeatedly, such as right-drawer grasping in Figure 6, indicates a mismatch between the learned model and the current execution environment. These low-probability or rapidly degrading skills can be prioritized for targeted data collection, additional demonstrations, or VLA fine-tuning, instead of uniformly collecting more data for all skills.

A.7 Hyperparameters in Online POMDP Planning

Parameter	Description	Value
λ	Empirical–uniform prior mixing weight for initial belief generation.	0.5
T	Temperature for all LLMs and VLMs	0.0
k	Number of scenarios in Belief Tree Search	500
d_s	Maximum search depth in Belief Tree Search	50
d_r	Rollout policy execution depth	20

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification:The abstract and Introduction (Section 1) clearly state the main contributions, which are consistently supported by the methods and experiments described in Sections 3 – 6.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: A dedicated Limitations section (Sec. 7) is provided.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper focuses on a new symbolic formulation, an algorithmic framework and empirical evaluation; it does not contain formal theorems or proofs.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer:[Yes]

Justification: The paper describes the algorithm design (Sec. 4 – 5), experimental setup (Sec. 6.1), baselines and evaluation protocol (Sec. 6.2 and 6.3) in sufficient detail to support reproduction of key results.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer:[Yes]

   We have publicly released the PO-PDDL implementation and documentation, including instructions for reproducing the learning and planning pipelines. The code is available at `https://github.com/RoboticSJTU/PO_PDDL`, and additional examples and qualitative results are provided at `https://roboticsjtu.github.io/PO-PDDL`.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: The main paper (Sec. 6) describes datasets, task setups, and evaluation protocols; additional hyperparameter details are included in the appendix.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: We report standard error in all bar plots (Figures 3 and 4).

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Hardware and runtime details are provided in the experimental setup (Sec. 6.1).

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We have reviewed the NeurIPS Code of Ethics, and our research complies with its guidelines; no ethical concerns are identified in this work.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [No]

    Justification: The paper does not explicitly discuss broader societal impacts; its focus is on new formulation and algorithmic design, and empirical evaluation for robot planning under uncertainty.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification:The paper does not release any pretrained models or scraped datasets that pose a high risk for misuse.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer:[Yes]

    Justification: We use $\pi_{0.5}$ as the VLA policy in experiments, and is properly cited in Section 6.1, with licenses and usage terms respected as per their official releases.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.

- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: The paper does not release new assets such as datasets or models; only the formulation and algorithm code will be open-sourced.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve any crowdsourcing or research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve research with human subjects and therefore does not require IRB approval.

    Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs play a central role in our approach, powering both the offline domain learning (Section 4) and online planning modules (Section 5). Further usage details are provided in the Appendix.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.