Cross-Domain Demo-to-Code via Neurosymbolic Counterfactual Reasoning

Supplementary Material

Contents

A. Further Related Works **2**

B. Environment Settings **2**
- B.1. Genesis . . . 2
- B.2. Real-world . . . 5
- B.3. Metrics . . . 5
- B.4. **NeSyCR** Implementation . . . 7
- B.5. Baselines . . . 12

C. Additional Experiments **17**
- C.1. Experiments on VLM State Translation . . . 17
- C.2. Ablations on VLM choice . . . 17
- C.3. Robustness to Scene Graph Perturbation . . . 17
- C.4. Running Time Analysis . . . 17

D. Additional Visualizations **19**
- D.1. Figure 5 in Main Paper Visualizations . . . 19
- D.2. Real-world Experiment Visualizations . . . 21

E. Prompts **22**
- E.1. Demo2Code . . . 22
- E.2. GPT4V-Robotics . . . 23
- E.3. Critic-V . . . 24
- E.4. MoReVQA . . . 26
- E.5. Statler . . . 30
- E.6. LLM-DM . . . 31
- E.7. NeSyCR . . . 35
- E.8. Common . . . 37

A. Further Related Works

Foundation models for embodied control. Foundation models have emerged as a promising alternative to conventional neural policies for embodied control, offering broad commonsense knowledge and reasoning capabilities that enable generalized task planning without extensive task-specific data [7, 15, 21, 27, 45, 55]. By leveraging pretrained language and vision representations, these models interpret human instructions and generate goal-directed behaviors for embodied agents. Recent studies have utilized LLMs and VLMs with code-writing capabilities to synthesize executable control code for embodied agents. LLM-based approaches map language instructions to code policies [4, 8, 20, 22, 29, 30, 47, 50], while VLM-based methods extend this to video demonstrations [49, 51, 53, 57]. In this work, we enhance the VLM-based paradigm with robust cross-domain transfer to bridge mismatches between demonstrations and deployment.

Cross-domain adaptation from demonstrations. A well-established approach for training embodied agents is to learn from demonstrations, which enables policy acquisition without explicit rewards or manual supervision [5, 25, 34, 41]. However, policies learned through behavioral cloning or inverse reinforcement learning often struggle to generalize under significant domain shifts, such as variations in perceptual and physical factors, between expert demonstrations and the agent's deployment [11, 16, 39, 40, 42]. Prior works have explored adaptation strategies based on feature alignment, policy fine-tuning, and state-transition matching, as well as video-based imitation methods that align visual representations across domains (e.g., human-to-robot transfer) [10, 38, 59, 60]. Despite these, achieving robust generalization under perceptual and physical variations remains challenging, especially with limited demonstrations and procedurally complex tasks. In this work, we focus on domain gaps that induce procedural discrepancies and address them through counterfactual reasoning.

Neurosymbolic approaches for embodied agents. Neurosymbolic methods integrate the adaptability of neural networks with the formal reasoning capabilities of symbolic tools, improving the correctness and interpretability of task-level planning and reasoning. Recent studies have leveraged foundation models to produce symbolic formulations that are subsequently verified by symbolic tools [24, 36, 37, 56]. In embodied control, symbolic formalisms such as PDDL [1] and ASP [18] were employed, using LLMs to encode domain knowledge and translate task descriptions into problem instances [2, 3, 9, 12–14, 28, 32, 33, 44]. Furthermore, recent works have incorporated VLMs to enable perceptually grounded symbolic reasoning, leveraging the models' vision–language pretraining to extract predicates, infer object relations, and construct symbolic world models that support embodied planning [6, 31, 46, 63]. Our work integrates neurosymbolic reasoning with counterfactual inference, where the VLM proposes alternative actions while a symbolic tool validates and adjusts the resulting procedures to ensure reliable task execution across domains.

B. Environment Settings

B.1. Genesis

Genesis [62] is a GPU-accelerated physics simulation platform designed for general-purpose robotic learning and evaluation. The platform provides a Python-based API that enables flexible implementation of custom environments with configurable scene layouts, object properties, and task specifications. We leverage Genesis to construct the simulated environment for evaluating our models on the cross-domain demo-to-code tasks, as shown in Table 1 of the main paper. Our evaluation framework encompasses two key dimensions: (1) cross-domain settings (rows of Table 1) that systematically vary environmental and embodiment factors, and (2) tasks with graduated complexity levels (columns of Table 1) ranging from simple operations to complex multi-step procedures. We provide detailed descriptions of each dimension below.

B.1.1. Cross-domain settings

To implement the cross-domain settings, we define five cross-domain factors grouped into three evaluation categories: (1) Obstruction and Object Affordance, which assess performance under environmental shifts; (2) Kinematic Configuration and Gripper Type, which assess performance under embodiment shifts; and (3) Combination, which evaluates robustness under both types of shifts simultaneously. These cross-domain settings can be applied to each task, allowing to introduce controlled domain gaps between the demonstration and deployment domains. We describe each cross-domain setting in detail below.

Obstruction and Object affordance. To implement this environmental factor, we introduce mechanisms to control obstruction and object-affordance levels for each subtask. The obstruction level ranges from 0 to 2, with higher levels indicating

increased domain gap and task complexity. As the obstruction level increases, target objects become more occluded, requiring the agent to resolve the obstruction before initiating the task. The scenes for the cross-domain settings for each subtask type are depicted in Figure 1, with their explanations provided in Table 1.

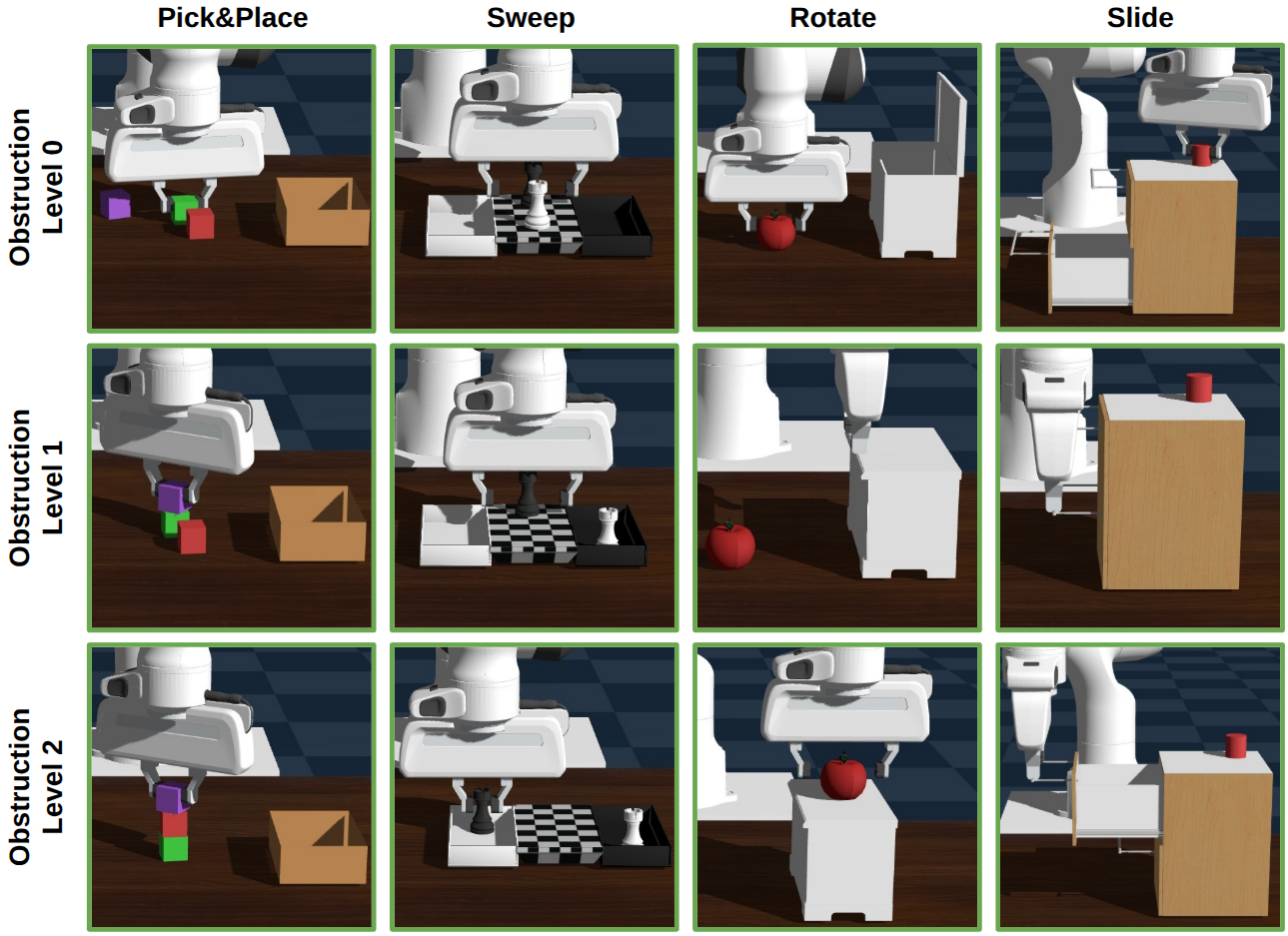

Figure 1. Example scene of Obstruction and Object affordance

Task	Level 1	Level 2
Pick & Place	A cube block stacked on preceding block.	Multiple cube blocks stacked on preceding block.
Sweep	A chess piece starts in the wrong box.	Multiple chess pieces start in the wrong box.
Rotate	Hinge lid starts closed.	Objects start stacked on top of the closed lid.
Slide	Target drawer starts closed.	Target drawer starts closed, other obstructs from above.

Table 1. Description of task variations across obstruction levels

Kinematic configuration and Gripper type. To implement this embodiment factor, we configure the robot with a 7-DoF vacuum suction gripper to compare it against a 9-DoF finger gripper. The finger gripper grasps objects by opening and closing its fingers, whereas the vacuum gripper secures objects by activating and deactivating its suction mechanism. As each embodiment provides a distinct Action API set, the code policy must reorganize how actions are invoked and composed to conform to the target APIs. When adaptation is incomplete, this restructuring of API usage does not occur. The example scenes for the cross-domain setting are depicted in Figure 2.

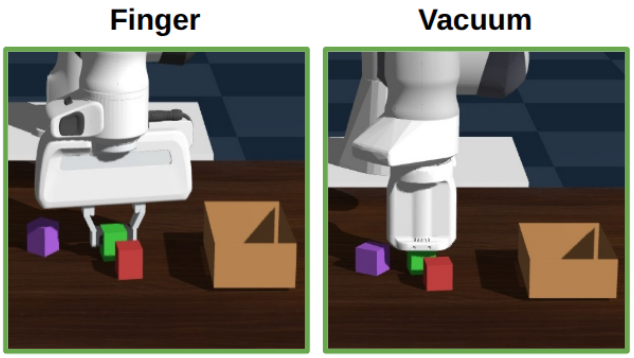

Figure 2. Example scene of Kinematic configuration and Gripper type

Combination. The Combined setting implements all cross-domain factor variations simultaneously, including obstruction and object-affordance levels, kinematic configuration, and gripper type. This setup creates the most challenging evaluation condition by introducing domain gaps across both environmental and embodiment dimensions, requiring comprehensive adaptation from the demonstration to the deployment domain.

B.1.2. Benchmark tasks

To assess the effectiveness of the generated code policy in the embodied domain, we design and implement four representative subtasks widely used in the area of robotic manipulation research [26, 52] in Genesis. These subtasks are composed to generate a single long-horizon task, with the difficulty of each subtask controlled by both domain factors and the complexity level, allowing for systematic evaluation. The example scenes for each subtasks are depicted in Figure 3.

- **Pick&Place**. The robot picks up cube blocks from the table and places them into a box.
- **Sweep**. The robot sweeps chess pieces across the board, pushing each piece into its corresponding box. The task succeeds only if all chess pieces are pushed inside their correct boxes.
- **Rotate**. The robot picks up fruits, places them into a hinged container, and closes the container by rotating the lid around its axis. The task is considered successful only if all fruits are placed inside the container and the lid is fully closed.
- **Slide**. The robot picks up cylinder blocks, places them into a drawer, and closes the drawer by sliding it shut. The task is considered successful only if all cylinder blocks are placed inside the drawer and the drawer is fully closed.

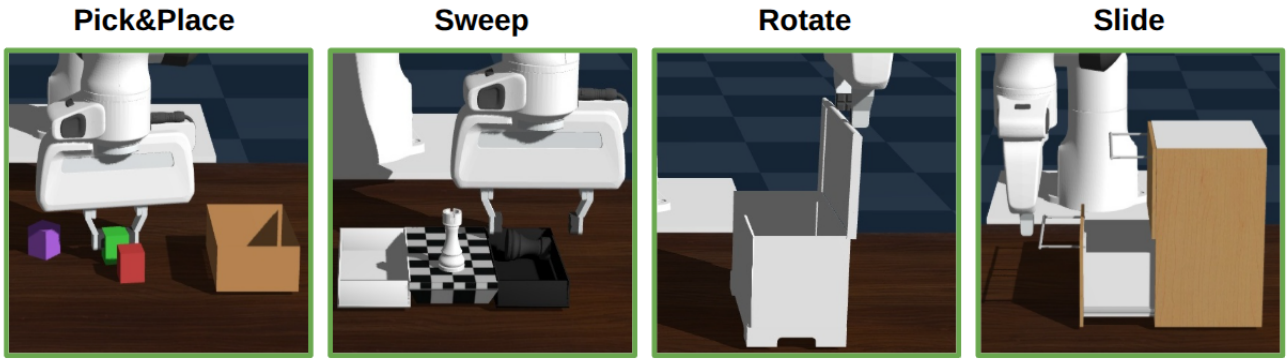

Figure 3. Example scenes for the subtasks

Task complexity level. By controlling the individual complexity level of each subtask and the total number of subtasks used to compose a single task, we control the task complexity used for evaluation. We divide the complexity level into three categories—low, medium, and high—where the low level consists of two subtasks, and the medium and high levels consist

of three and four subtasks, respectively. As complexity increases, not only does the number of subtasks grow, but the number of task-objects in each subtask also increases, making the task more complex and long-horizon. Example scenes for each task complexity level are depicted in Figure 4.

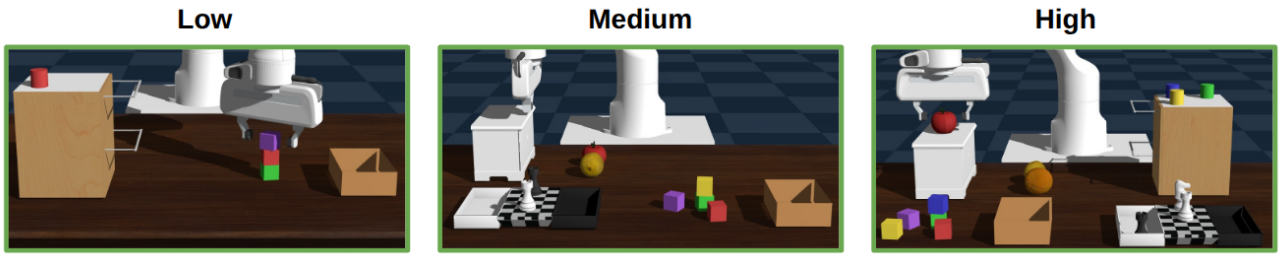

Figure 4. Example scenes for each task complexity level

B.1.3. Evaluation

We evaluate scenarios constructed by combining domain settings with task complexity levels. In total, 220 scenario configurations are generated and categorized into three complexity levels: 80 Low, 80 Medium, and 60 High. Each configuration is evaluated on scenes initialized with two different random seeds, yielding a total of 440 evaluation scenarios as depicted in Table 2.

Domain Factors \ Task Complexity	**Low**	**Medium**	**High**	**Total**
Obstruction & Object Affordance	60	60	40	160
Kinematic Config & Gripper	60	60	40	160
Combination	40	40	40	120
Total Scenarios	**160**	**160**	**120**	**440**

Table 2. Experimental scenarios for model evaluation

B.1.4. Primitive APIs

To connect the generated code policy with robot control, we expose a set of primitive APIs that the VLM-generated code policy can invoke during execution. Specifically, we provide two primitive interfaces: a Perception API and an Action API. The complete specifications of these APIs are detailed in Table 3 and Table 4.

B.2. Real-world

Environment setup. We conducted the real-world experiments using a 7-DoF Franka Emika Research 3 robotic arm with a two-finger gripper on a tabletop workspace. An Intel RealSense D435 RGB-D camera was mounted above the table to provide top-down RGB and depth observations. The captured images were then processed by an object detection and segmentation module to extract the categories and bounding boxes of task-relevant objects.

Object configuration. Across all experiments including the main scenario in Figure 1 of the main paper, we use an object pool consisting of three drawers, a magnetic hook, various types of screws, and other auxiliary objects for intermediate manipulations. The initial positions of all objects are randomized for each trial. Figure 5 shows a representative example of the environment used in the experiments.

B.3. Metrics

We employ three metrics that assess different aspects of task completion and procedure adherence.

Perception API

Primitive API	Description
`is_obj_visible(obj_name)`	Returns a boolean indicating whether the object is present in the scene's object list.
`get_obj_names()`	Returns a list of all object names currently present in the scene.
`get_obj_pos(obj_name)`	Returns the (x, y, z) position of the corresponding object.
`get_obj_bbox(obj_name)`	Returns its axis-aligned bounding box `[min, max]` in world coordinates.
`get_obj_size(obj_name)`	Returns its size vector computed as the difference between the max and min corners of its bounding box.
`gripper_is_open()`	Returns a Boolean indicating whether the gripper is currently open.
`obj_in_gripper(obj_name)`	Returns a Boolean indicating whether the object is currently within the gripper's grasp or suction region.
`get_empty_floor_xy(obj_name)`	Returns a collision-free (x, y) position on the floor where an object can be placed without overlapping existing objects.

Table 3. Perception API primitives

Action API

Primitive API	Description
`move_gripper_to(obj_name, depth)`	Moves the end-effector toward the object.
`move_to_position(pos)`	Moves the end-effector to a target position.
`move_parallel(move_dir, offset)`	Moves the end-effector parallel to the workspace plane in the specified direction by a given offset.
`grasp_handle(handle_name)`	Grasps the handle when the end-effector is sufficiently close.
`release_handle()`	Releases the currently grasped handle and opens the gripper.
`open_gripper()`	Open the finger gripper.
`close_gripper()`	Close the finger gripper.
`attach_vacuum_handle(handle_name)`	Vacuum suction tool counterpart to `grasp_handle`.
`detach_vacuum_handle()`	Vacuum suction tool counterpart to `release_handle`.
`deactivate_vacuum()`	Vacuum suction tool counterpart to `open_gripper`.
`activate_vacuum()`	Vacuum suction tool counterpart to `close_gripper`.

Table 4. Action API primitives

Success Rate (SR). The Success Rate measures the percentage of tasks that are completed in full. A task is counted as successful only when all subtasks are achieved:

$$\text{SR} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}[\text{all subtasks completed in task } i]$$

where N is the total number of tasks and $\mathbb{1}[\cdot]$ is the indicator function.

Goal Condition (GC). The Goal Condition metric measures the proportion of success conditions achieved, reflecting the degree of subtask completion [43]:

$$\text{GC} = \frac{1}{N} \sum_{i=1}^{N} \frac{\text{number of achieved subtasks in task } i}{\text{total number of subtasks in task } i}$$

This metric provides a more granular view of partial task completion compared to SR.

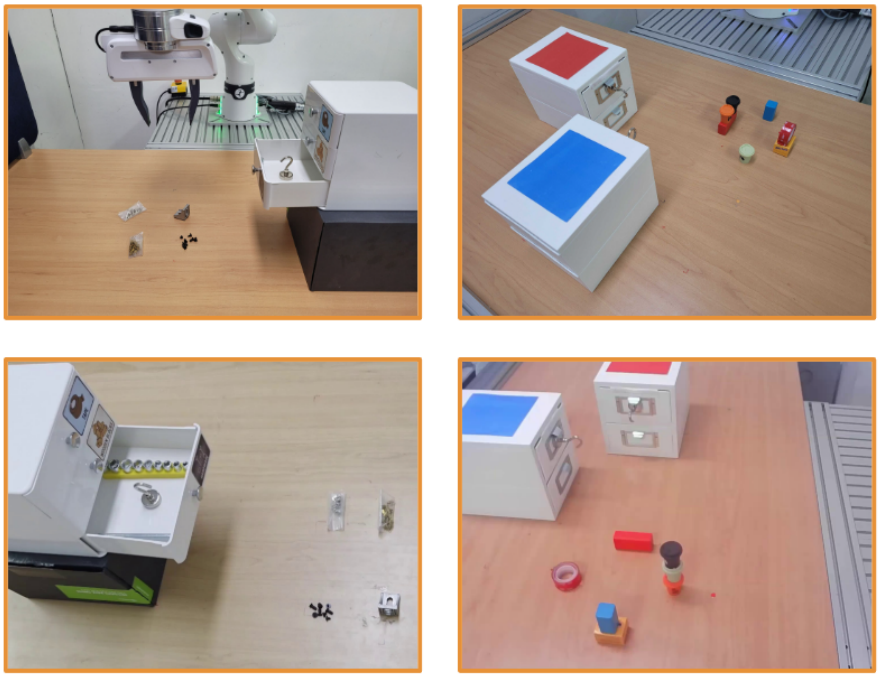

Figure 5. Example scenes from the real-world environment used in our experiments

Procedure Deviation (PD). The Procedure Deviation quantifies the alignment between the adapted procedure and the demonstrated procedure using a length-normalized edit distance over their subtask-achievement sequences of succeeded tasks [17, 23].

$$\text{PD} = \frac{1}{N_{\text{success}}} \sum_{i \in \mathcal{S}} \frac{\text{EditDistance}(S_i^{\text{adapt}}, S_i^{\text{demo}})}{\max(|S_i^{\text{adapt}}|, |S_i^{\text{demo}}|)}$$

where $\mathcal{S}$ is the set of successfully completed tasks, $N_{\text{success}} = |\mathcal{S}|$ is the number of successful tasks, S_i^{adapt} and S_i^{demo} are the subtask-achievement sequences for the adapted and demonstrated procedures in task i, respectively, and $\text{EditDistance}(\cdot, \cdot)$ [23] computes the edit distance between two sequences.

B.4. NeSyCR Implementation

In this section, we describe the implementation details of NeSyCR, which is built upon two complementary reasoning engines: a vision language model (VLM) and a symbolic tool. For the VLM component, we employ the general-purpose GPT-5 model and specialize its behavior via stage-specific prompting to satisfy the distinct functional requirements of each module. For symbolic tool, we use the open-source PDDLGym [1] library to handle PDDL parsing and state grounding. Because NeSyCR only requires forward state progression—rather than full planning—we implement a symbolic execution logic in Python that performs precondition verification and effect application. We provide the pseudo-code for the symbolic tool in Algorithm 1.

To use the symbolic representation, a predicate set that determines the scope of symbolic abstraction must be supplied. We design a predicate set aimed at capturing generalizable relations common across embodied domains, including both physical relations and embodiment-specific predicates. Although richer or more domain-specific relations could be incorporated, we restrict ourselves to general predicates, as predicate invention lies outside the focus of this work. The predicate set used in main experiment is provided in Figure 6, note that this same predicate set is also shared among baselines that use symbolic representations.

[1] https://github.com/tomsilver/pddlgym

Algorithm 1 Symbolic Tool Φ

/ Forward execution with precondition check */*
function SYMBOLICEXECUTE(s, a)
  if pre(a) $\not\subseteq$ s **then**
    return $\bot$, pre(a) $\setminus$ s — *Precondition Verification*
  end if
  $s' \leftarrow (s \setminus \text{del}(a)) \cup \text{add}(a)$ — *Effect Application*
  return s', $\emptyset$

/ Verification process */*
function SYMBOLICVERIFY(s, a)
  $s', V \leftarrow$ SYMBOLICEXECUTE(s, a)
  if $V \neq \emptyset$ **then**
    return FAIL, V — *Inconsistent*
  end if
  return PASS, s' — *Consistent*

Predicate Set

Predicate	Definition
`# Physics-related`	
`(OverOf ?a ?b)`	?a is vertically above ?b without contact.
`(OnTopOf ?a ?b)`	?a is resting on and supported by ?b.
`(InsideOf ?a ?b)`	?a is contained within ?b.
`(Open ?x)`	Container ?x is fully open.
`(Closed ?x)`	Container ?x is fully closed.
`# Embodiment-specific`	
`(FingerGripper)`	Robot uses a two-finger gripper.
`(VacuumSuction)`	Robot uses a vacuum suction tool.
`(GripperSurrounding ?x)`	Gripper encloses ?x without closing.
`(GripperHolding ?x)`	Gripper is closed and holding ?x.
`(GripperOpen)`	Gripper is open.
`(GripperClosed)`	Gripper is closed.
`(VacuumAligned ?x)`	Vacuum is aligned with ?x but inactive.
`(VacuumAttached ?x)`	Vacuum is active and attached to ?x.
`(VacuumActive)`	Vacuum suction is on.
`(VacuumInactive)`	Vacuum suction is off.

Figure 6. Predicate set used for symbolic state representation

B.4.1. Symbolic World Model Construction

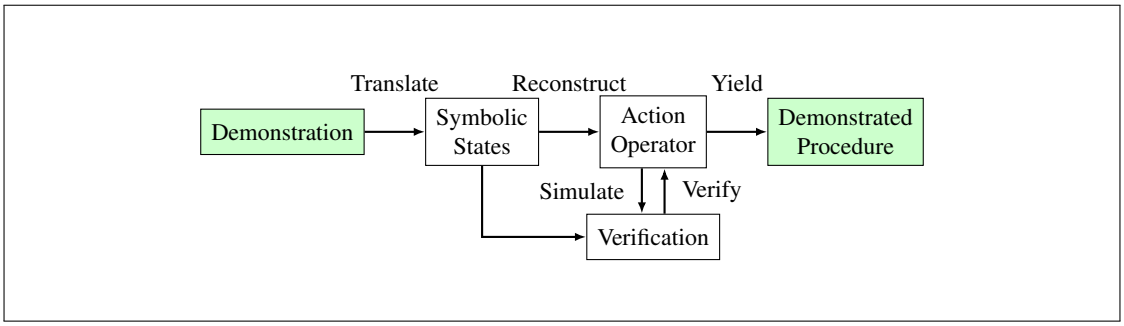

Figure 7. NESYCR symbolic world model construction high-level flow

VLM. In symbolic world model construction, VLM performs: (1) *symbolic state translation*, where we process demonstration using a VLM with sequence of multi-view images and objects to extract grounded scene graphs representing symbolic states at each timestep, forming an ordered sequence of symbolic states from the first timestep to the final timestep that captures object entities and spatial relations; and (2) *symbolic dynamics reconstruction*, where the VLM analyzes consecutive state pairs at each timestep to predict action operators. For each transition between timestep t and $t+1$, the VLM is provided with the previous state, current state, and their state difference (additions and deletions of predicates). The VLM then predicts action operators specifying: (i) action semantic description, (ii) preconditions required for execution, and (iii) effects produced after execution. Examples of the symbolic states and the action operators are provided in Figure 8.

Symbolic Tool. In symbolic world model construction, symbolic tool performs: (1) *action verification* through the following process: for each predicted action operator at timestep t, the tool (i) verifies that the current symbolic state fulfills all preconditions specified in the operator, (ii) applies the action's effects to the current state to produce the resulting next state, and (iii) validates that this resulting state matches the expected symbolic state at timestep $t+1$. If verification fails, the VLM is triggered to repredict the action operator until the entire sequence is verified.

Symbolic State Example

```
- (Closed top_drawer)
- (GripperClosed)
- (GripperHolding green_cylinder)
- (OnTopOf apple floor)

  ⋮

- (OnTopOf toy_box floor)
- (OnTopOf yellow_cube floor)
- (Open bottom_drawer)
- (Open hinge_lid)
```

Action Operator Example

Name:
`MoveGripperToSurroundOrange`

Preconditions:

```
- (GripperSurrounding orange)
- (GripperOpen)
- (forall (?y - thing)
    (not (GripperHolding ?y))
  )
```

Effects:

```
- (GripperClosed)
- (GripperHolding orange)
- (not (GripperOpen))
- (not (GripperSurrounding orange))
```

Figure 8. Example of symbolic state and action operator extracted from the demonstration

B.4.2. Neurosymbolic Counterfactual Adaptation

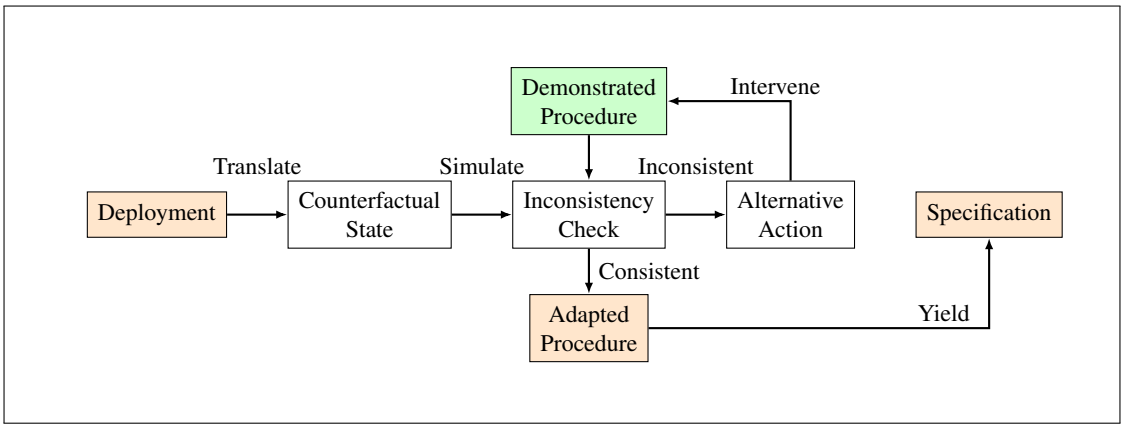

Figure 9. NESYCR neurosymbolic counterfactual adaptation high-level flow

VLM. In neurosymbolic counterfactual adaptation, VLM is used for: (1) *counterfactual state translation*, where the VLM observes the deployment scene to generate a counterfactual initial state that reflects target-domain conditions while maintaining compatibility with the symbolic predicates from the world model; and (2) *counterfactual exploration*, where for each action identified as inconsistent during symbolic forward simulation, the VLM proposes alternative action operators by analyzing the current counterfactual state, the incompatible original action, and the violated preconditions, generating interventions whose effects restore the violated preconditions or achieve equivalent outcomes under the deployment domain constraints.

Symbolic Tool. In neurosymbolic counterfactual adaptation, the symbolic tool performs two functions: (1) *counterfactual identification*, where it simulates the demonstrated procedure in the counterfactual setting by iteratively computing the next counterfactual state by applying each demonstrated action to the current counterfactual state, checking whether the action's preconditions are satisfied in the current state, flagging actions as inconsistent when some preconditions are violated; and (2) *action verification*, where it validates VLM-proposed alternative action by verifying that its preconditions are satisfied in the counterfactual state.
The identification-exploration loop continues iteratively until either the adapted procedure successfully reaches the goal condition while maintaining causal consistency throughout, or the maximum number of iterations is reached. An example of counterfactual identification and exploration is provided in Figure 10. Main hyperparameters of NESYCR are listed in Table 5.

Hyperparameters	Value
Max explorations	10 (low/medium), 20 (high) (Obstruction/Kinematic)
	15 (low/medium), 30 (high) (Combination)

Table 5. NESYCR hyperparameters

Identification Example

Current Counterfactual State:

```
- (Closed bottom_drawer)
- (Closed hinge_lid)
- ...
```

Incompatible Action:

```
OpenGripperToDropOrangeIntoHingeBody
- Preconditions:
  - (not (Closed hinge_lid))
  - ...
- Effects:
  - (InsideOf orange hinge_body)
  - ...
```

Violated Precondition:

```
- (not (Closed hinge_lid))
```

Exploration Example

```
<<<<<<< SEARCH
OpenGripperToDropOrangeIntoHingeBody
- Preconditions:
  - (not (Closed hinge_lid))
  - ...
- Effects:
  - (InsideOf orange hinge_body)
  - ...
.
.
.

=======
MoveHeldOrangeOverFloor
- Preconditions:
  - (GripperHolding orange)
  - ...
- Effects:
  - (OverOf orange floor)
  - ...
.
.
.

>>>>>>> REPLACE
```

Figure 10. Example of counterfactual identification and exploration for the demonstration

B.5. Baselines

All baselines receive a single demonstration as input—comprising a sequence of multiview images, instruction, and object set, supplemented with task context—and produce a target task specification in the form of high-level procedure. Once the task specification is generated, the code policy synthesis process remains consistent across all baseline models. An example of task specification is provided in Figure 11.

Task Specification Example

```
1. Move gripper to surround green cylinder
2. Close gripper on green cylinder
3. Move held green cylinder over bottom drawer
4. Release green cylinder into bottom drawer

  ⋮

46. Move gripper to surround red cube
47. Close gripper on red cube
48. Move held red cube over toy box
49. Open gripper to place red cube into toy box
```

Figure 11. Example of task specification

Table 6 summarizes the characteristics of each baseline method across five dimensions: **Natively Code** indicates whether the method generates code policies in its original work, or has been adapted for our evaluation. **Explicit Adapt** denotes explicit mechanisms for adapting demonstrations to the deployment domain. **Replanning** indicates generating compatible procedures by replanning from scratch in the deployment domain, rather than adapting demonstrations. **World Model** refers to use of structured world representations. **Symbolic Tool** denotes the use of external symbolic tool.

Baseline	Natively Code	Explicit Adapt	Replanning	World Model	Symbolic Tool
Demo2Code	✓	✗	✗	✗	✗
GPT4V-Robotics	✓	✓	✓	✗	✗
Critic-V	✗	✓	✗	✗	✗
MoReVQA	✗	✓	✗	✗	✗
Statler	✗	✓	✓	✓	✗
LLM-DM	✗	✓	✓	✓	✓

Table 6. Comparison of baseline methods

- **VLM-based code policy synthesis.** uses VLMs to generate task specifications from visual demonstrations and synthesizes code policies, without incorporating an explicit adaptation mechanism.
  - Demo2Code [51] generates task specifications by recursively summarizing the demonstration, with deployment domain information provided at intermediate steps.
- **VLM-based reasoning.** utilize the reasoning capabilities of VLMs to perform adaptation, generating task specifications tailored to the target domain.
  - GPT4V-Robotics [48] generates task information from demonstrations and uses a VLM to generate a grounded target specifications from visually observing the target scene.
  - Critic-V [61] uses a VLM-based critic to iteratively refine the initial task specification through generated critiques, ensuring compatibility with the deployment domain.
  - MoReVQA [35] follows a multi-stage modular reasoning process. It generates task information from demonstrations and uses VQA-style VLM querying to generate grounded target specifications.

- **World-model-based approaches.** leverage LLM-based or neurosymbolic world models to support target task specification generation through high-level replanning mechanisms.
  - Statler [58] equips LLMs with explicit world state representations that serve as memory throughout the replanning process, facilitating consistent reasoning over extended time horizons.
  - LLM-DM [19] constructs explicit PDDL world models from demonstrations using LLMs, then employs domain-independent symbolic planners to generate target task specifications by searching for high-level plans in the problem file for deployment domain.

Demo2Code. This method generates a code policy from demonstrations by recursively summarizing the demonstration through extended chain-of-thought to yield a task specification, which acts as a seed for generating the code policy. While the original work does not assume a cross-domain setting, we modify the implementation by injecting deployment domain information in the final stage of summarization.

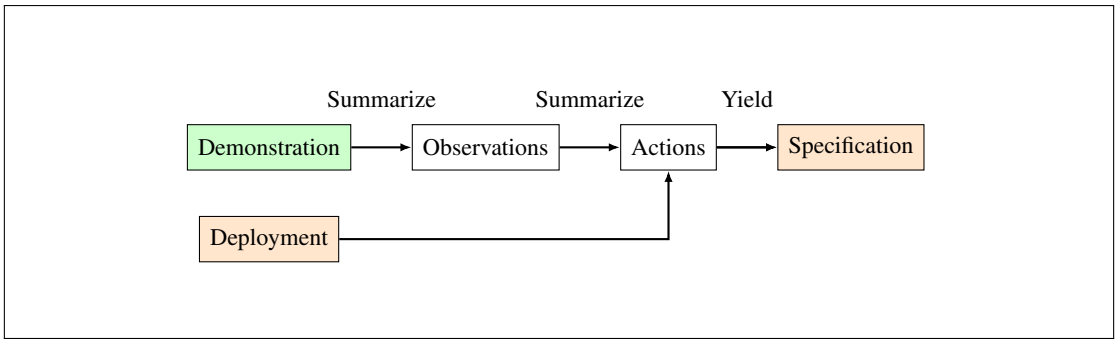

Figure 12. Demo2Code high-level flow

We implement Demo2Code referring to the official repository [2]. Implementation of adapted Demo2Code compose of: (1) *observation prediction*, where we batch-process source demonstration timesteps using a VLM with multi-view images (top, front, back) to extract an ordered sequence of high-level observations; and (2) *action prediction*, where the VLM predicts high-level actions from the observation sequence while being provided with deployment domain information to generate domain-adapted actions that serve as the task specification.

GPT4V-Robotics. This method generates task specifications from visual demonstrations by first extracting domain-agnostic task descriptions from source demonstrations, then grounding them to deployment environments through visual scene understanding. While the original work does not assume a cross-domain setting, we adapt the model by providing target scene information during the action planning stage. Note that we retain the name GPT4V-Robotics from the original work, though we use GPT-5 as the base VLM for consistency.

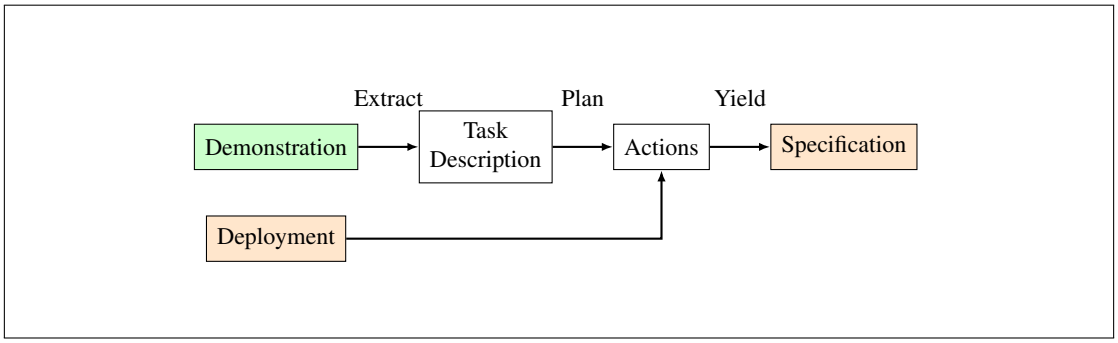

Figure 13. GPT4V-Robotics high-level flow

[2] https://github.com/portal-cornell/demo2code

We implement GPT4V-Robotics referring to the official repository [3]. Implementation of adapted GPT4V-Robotics compose of: (1) *task description generation*, where a VLM analyzes the source demonstration to extract high-level task understanding and domain knowledge; and (2) *action planning*, where the VLM generates an ordered sequence of grounded actions by observing the deployment scene

Critic-V. This method enhances VLM multimodal reasoning through iterative refinement with natural language critiques from visual observations. We adapt this framework for cross-domain demo-to-code by using it to iteratively refine action plans generated from source demonstrations until they align with the visual analysis of the deployment scene, and use the action plans to yield specifications for generating code policies.

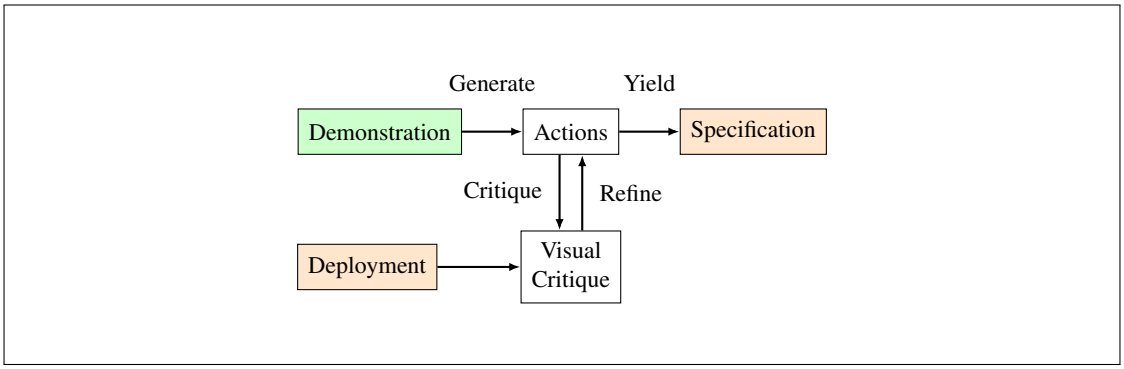

Figure 14. Critic-V high-level flow

We implement Critic-V referring to the official repository [4]. Implementation of the adapted Critic-V compose of: (1) *initial action generation*, where a VLM generates an initial action sequence from source demonstrations; (2) *critique generation*, where a VLM critic observes the deployment scene to identify incompatibilities in the actions and provides natural language feedback; and (3) *action refinement*, where the VLM refines the actions based on the feedback. Steps (2) and (3) repeat until the critic determines no issues exist or a maximum number of iterations is reached.

Hyperparameters	Value
Max critique refinement	10 (low/medium), 20 (high) (Obstruction/Kinematic)
	15 (low/medium), 30 (high) (Combination)

Table 7. Critic-V hyperparameters

MoReVQA. This method operated through a three-stage modular pipeline consisting of event parsing, grounding, and reasoning. We adapt it for cross-domain demo-to-code by using these stages to construct subgoals and query the deployment scene for achieve subgoal, ultimately generating a grounded specification that is used to produce the final code policy.
We implement MoReVQA based on the official supplementary material provided [5]. The implementation of the adapted MoReVQA compose of: (1) *M1 Event Parsing* processes the input instruction from the demonstration, converts it into a parsed event, and stores it in shared memory. (2) *M2 Grounding* uses both the demonstration's task description and the parsed event stored in shared memory to generate subgoals that are adapted to the deployment scene. (3) *M3 Reasoning* combines the parsed event in memory with the VQA results derived from the deployment scene and generates the actions required to achieve each subgoal.

[3] https://github.com/microsoft/GPT4Vision-Robot-Manipulation-Prompts
[4] https://github.com/kyrieLei/Critic-V
[5] https://juhongm999.github.io/morevqa

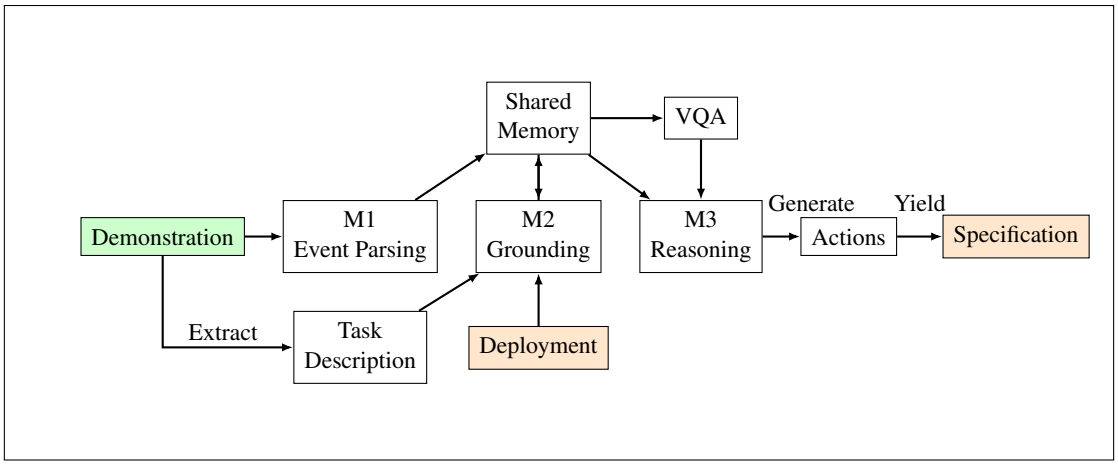

Figure 15. MoReVQA high-level flow

Statler. This method ensures consistent state tracking across planning steps by maintaining an explicit world state representation as a memory. We adapt this framework for cross-domain demo-to-code scenarios by extracting task descriptions from demonstrations to re-generate action plans on the deployment domain, and use the action plans to yield specifications for generating code policies.

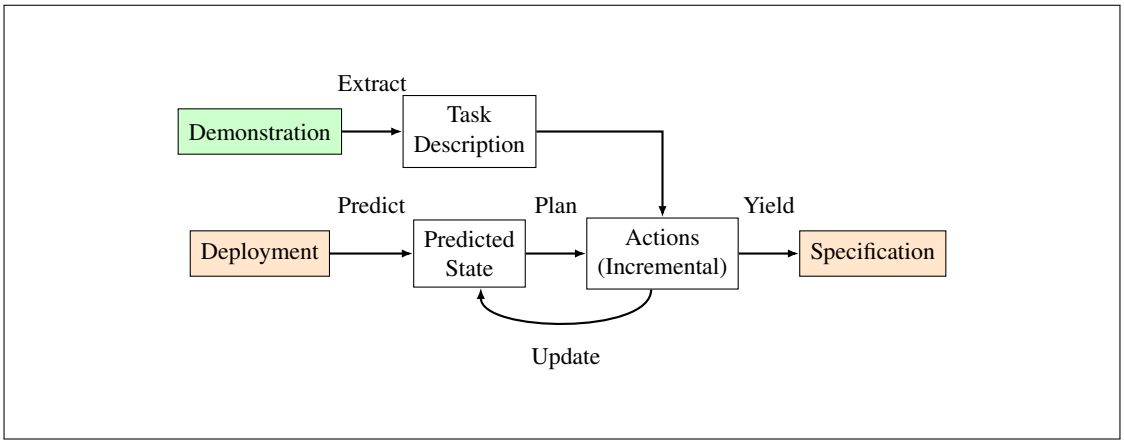

Figure 16. Statler high-level flow

We implement Statler referring to the official repository [6]. Implementation of the adapted Statler compose of: (1) *task description generation*, where a VLM analyzes the source demonstration to extract high-level task understanding and domain knowledge; (2) *initial state prediction*, where the VLM observes the deployment scene to predict the initial state using the same predefined predicates as NESYCR; and (3) *incremental action planning*, where at each step the VLM generates next several actions conditioned on the current state, task description, and previous actions, then updates the state representation accordingly. This state-action cycle repeats iteratively until the goal is reached or a maximum number of iterations is reached.

LLM-DM. This method employs a neurosymbolic planning approach by generating PDDL (Planning Domain Definition Language) files via VLM and leverage symbolic solver to derive action plan. We adapt this framework for cross-domain demo-to-code by using VLMs to predict domain file from source demonstration and problem file from deployment information, then iteratively refine the domain file until a valid plan is found through PDDL solving and using the plan as specification to generate code policy.

[6] https://github.com/ripl/statler

Hyperparameters	Value
Max planning iteration	10 (low/medium), 20 (high) (Obstruction/Kinematic) 15 (low/medium), 30 (high) (Combination)

Table 8. Statler hyperparameters

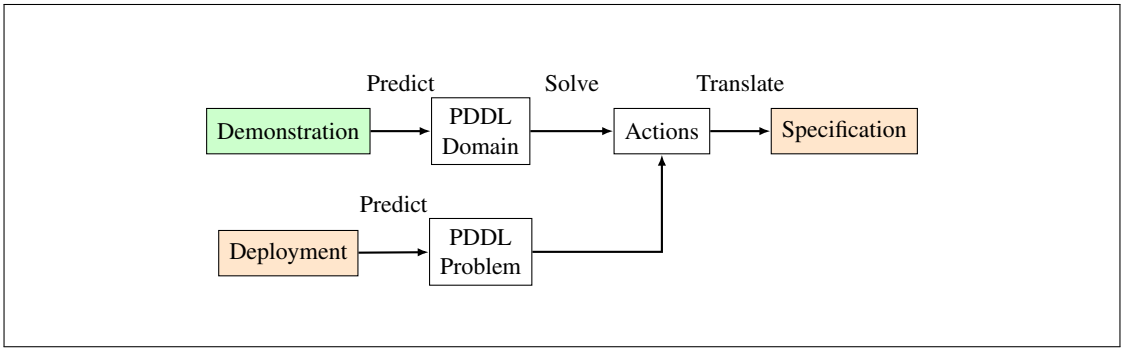

Figure 17. LLM-DM high-level flow

We implement LLM-DM referring to the official repository [7], with our own implementation for the problem file prediction. Implementation of the adapted LLM-DM consists of: (1) *domain prediction*, where VLMs analyze source demonstrations to propose additional predicates beyond those in NESYCR, along with actions, for constructing a PDDL domain file; (2) *problem prediction*, where VLMs observe the deployment scene to formulate a PDDL problem file using the proposed predicates; (3) *PDDL-based solving*, where a symbolic solver attempts to synthesize an action sequence satisfying the given files; and (4) *domain refinement*, where upon solving failure, VLMs analyze the failure and refine the domain model. This plan-refine cycle repeats until a valid plan is found or maximum attempts are reached. For the symbolic solver, we use Fast Downward as provided in the official PDDLGym Planners repository [8].

Hyperparameters	Value
Max domain refinement	10 (low/medium), 20 (high) (Obstruction/Kinematic) 15 (low/medium), 30 (high) (Combination)

Table 9. LLM-DM hyperparameters

[7] https://github.com/GuanSuns/LLMs-World-Models-for-Planning
[8] https://github.com/ronuchit/pddlgym_planners

C. Additional Experiments

C.1. Experiments on VLM State Translation

To evaluate the performance of symbolic state translation, we conduct experiments following [54], with results presented in Table 10. The results show that the VLM reliably translates raw frames into scene graphs, achieving an F1-score above 0.82 against the ground-truth symbolic states of real-world scenes using GPT-5.

Method	*Real-world*		
	Precision	Recall	F1
Method: NESYCR			
GPT-5	0.97	0.71	0.82
GPT-5-mini	0.88	0.75	0.80

Table 10. Performance on VLM symbolic state translation

C.2. Ablations on VLM choice

Table 11 reports the ablation study on VLM choice for NESYCR. The results show that while GPT-5 achieves the best performance (with 80.00% SR, 88.33% GC), smaller models like GPT-5-mini and GPT-4 series still maintain competitive results with SR ranging from 63-70% and GC above 75%. This shows that NESYCR remains effective across varying model scales.

Method	*Combined*		
	SR	GC	PD
Method: NESYCR			
GPT-5	80.00 ±7.43	88.33 ±4.60	3.75 ±2.61
GPT-5-mini	70.00 ±8.51	76.67 ±7.08	11.43 ±4.04
GPT-4.1	66.67 ±8.75	80.00 ±5.67	9.00 ±4.16
GPT-4o	63.33 ±8.95	75.00 ±6.67	11.05 ±4.39

Table 11. Ablation on VLM choice

C.3. Robustness to Scene Graph Perturbation

To evaluate the robustness of NESYCR to imperfect symbolic state translations, we conduct experiments under two types of scene graph perturbations: (1) randomly dropping 10% of relations, and (2) injecting 10% noisy (incorrect) relations. As shown in Table 12, NESYCR maintains stable performance even when perturbations are applied to all scene graphs along the demonstration. This robustness stems from the fact that perturbation-induced precondition violations are handled identically to violations arising from true cross-domain mismatches, through the same verification and refinement loops employed in both the symbolic world modeling and adaptation phases.

	None	10% Drop	10% Noise
NESYCR	65.00±7.64	55.00±7.97	60.00±7.84

Table 12. Robustness to scene graph perturbation

C.4. Running Time Analysis

Table 13 reports the running time comparison between iterative methods, decomposed into VLM inference time and symbolic tool execution time. NESYCR incurs minimal overhead compared to other iterative baselines, as symbolic tool

operates as a forward executor that performs sequential state transitions with complexity linear in the demonstration length, rather than exponential symbolic search. The primary computational bottleneck is VLM inference, a cost shared across all iterative baselines; the symbolic tool itself contributes marginal latency.

Method	VLM	Symbolic	Total
Critic-V	196.23s	–	196.23s
Statler	83.96s	–	83.96s
LLM-DM	91.56s	0.40s	91.96s
NESYCR	118.22s	0.01s	118.23s

Table 13. Running time comparison between iterative methods.

D. Additional Visualizations

D.1. Figure 5 in Main Paper Visualizations

In this section, we illustrate the adaptation process of NESYCR in Figure 18, and the execution of the generated code policy for our main scenario in Figure 19.

Left. The demonstration-derived procedure includes the redundant action "Release Magnetic Hook Into Bottom Drawer" since the action requires the target object to not be in the target position whereas magnetic hook already occupies its target position in the deployment domain, violating the precondition `(not (InsideOf magnetic_hook bottom_drawer))` identified by the symbolic tool. Based on the violated precondition and the current state showing that the magnetic hook is already in place, the VLM generates an exploration to remove the redundant action chunk.

Middle. A domain gap exists in the deployment domain where the fine-grained object, black screws are scattered rather than gathered, violating the precondition for action "Grasp Black Screws" `(not (and (Finegrained black_screws) (not (Gathered black_screws))))`. Based on the violated precondition and current state showing that magnetic hook is present in the scene, the VLM generates an exploration that repurposes the magnetic hook as an aggregation tool, inserting new actions to collect the scattered screws before attempting to grasp them.

Right. "Grasp Magnetic Hook" in the newly added hook retrieval action causes a precondition violation `(not (and (OnTopOf silver_screws magnetic_hook) (not (OnTopOf gold_screws bottom_drawer)) (not (OnTopOf bracket bottom_drawer))))` indicating that multiple objects are stacked on top of the hook. VLM resolves by reordering the procedure: the hook retrieval and screw aggregation steps are moved earlier in the sequence, before organizing intermediate objects. This resolves the violation, producing a final procedure compatible with the deployment domain.

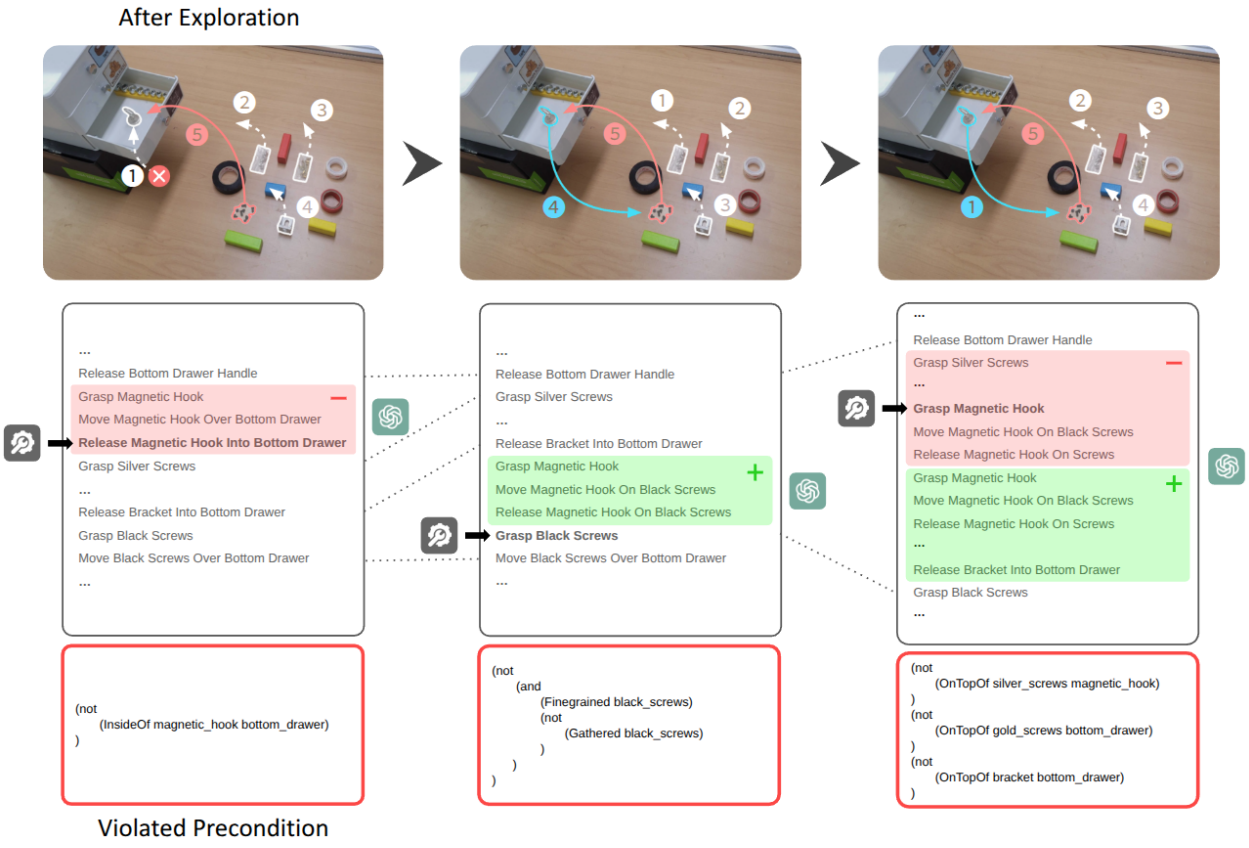

Figure 18. An expanded visualization of the adaptation process illustrated in Figure 1

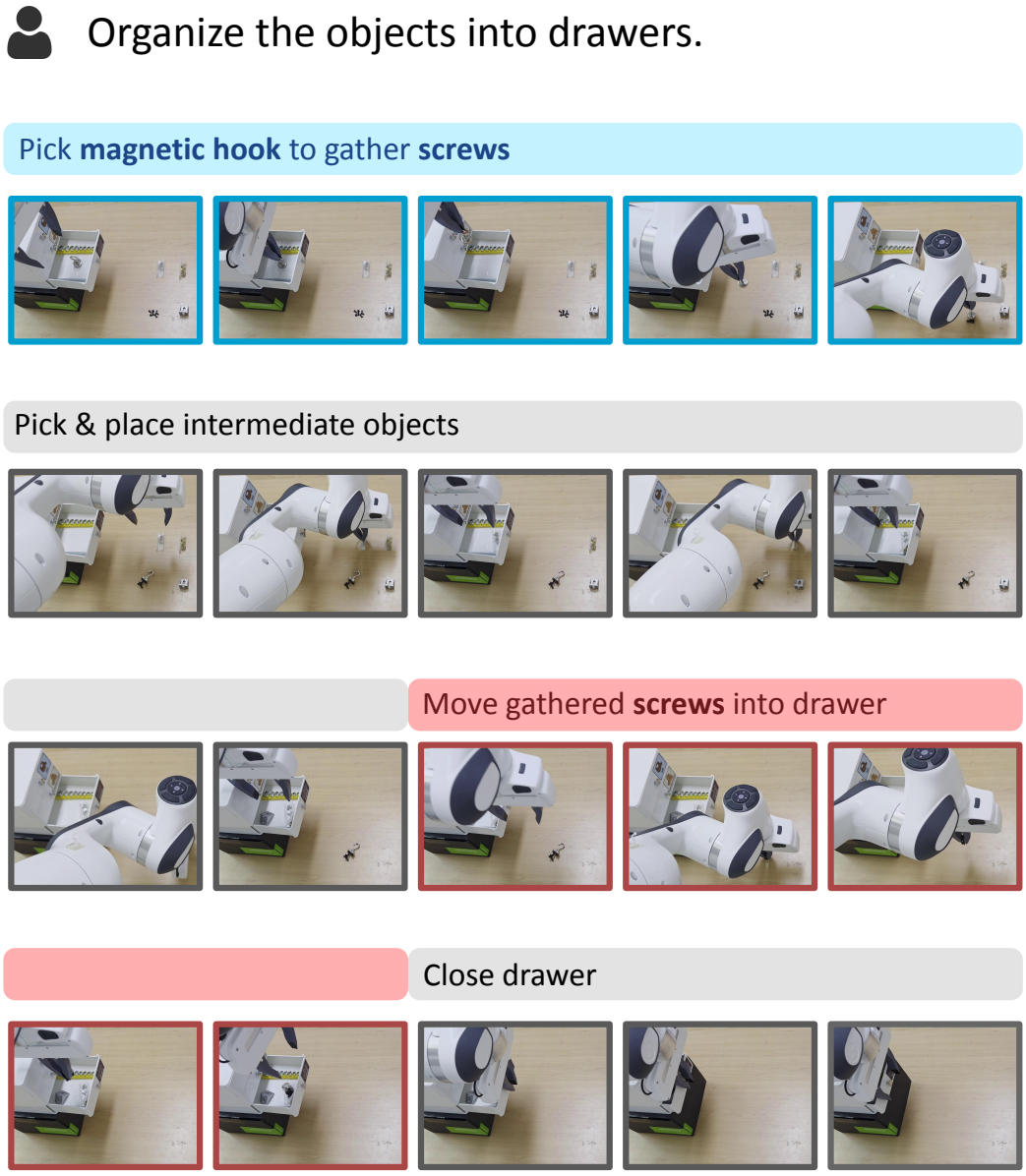

Figure 19. An expanded visualization of the task execution illustrated in Figure 1

D.2. Real-world Experiment Visualizations

In this section, we illustrate the execution of the generated code policy for our real-world experiment in Figure 20.

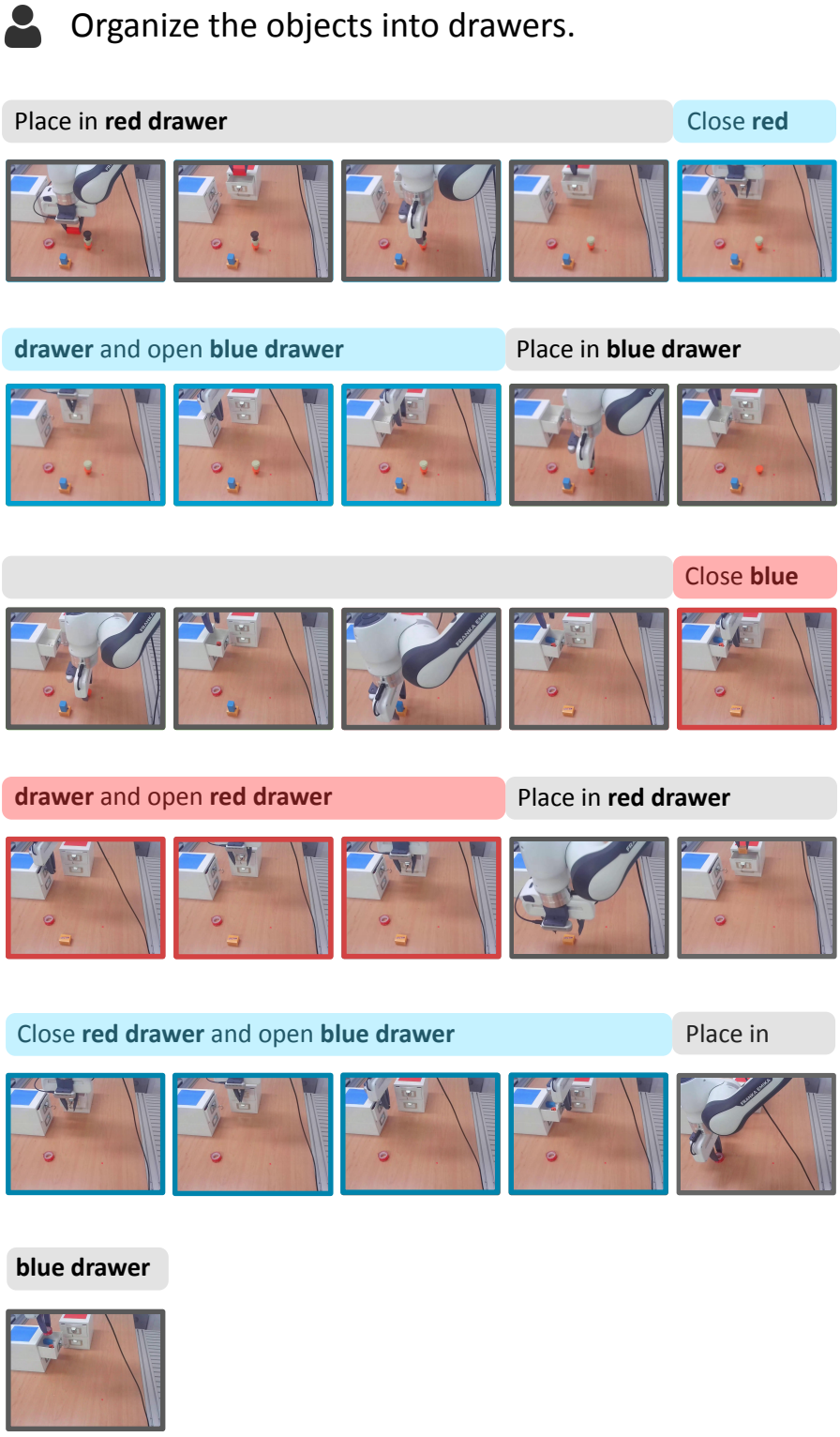

Figure 20. Visualization of real-world experiment in Table 2

E. Prompts

E.1. Demo2Code

Demo2Code — Observation Prediction

[System]
You are a scene descriptor for robotics. The user will provide three synchronized images of the scene (top/front/back views). Your task is to describe what is happening in the scene and how are objects positioned and oriented based on provided scene and additional information.

Output requirements:

- Provide one clear and concise scene description including the position and orientation of the objects.
- Start the scene description with an unordered dash (-).

[User]
Your task is to describe what is happening in the scene and how are objects positioned and oriented based on provided scene and additional information.

Below is an example.
`[Example]`

Now provide the scene descriptions for the following scene and information.

Information

- Instruction:
  {`instruction`}
- Object in scene:
  {`objects`}
- Object state in scene:
  {`object_state`}

Observation Description

Demo2Code — Action Prediction

[System]
You are an action predictor for robotics. The user will provide observation sequence consisting of scene descriptions for each timestep and additional scene information. Your task is to infer the high-level action that occurs between each pair of consecutive timesteps based on provided observation sequence and scene information.

Output requirements:

- Provide one clear and concise action description, including the semantic and movement details of objects, for each transition between timesteps.
- Start the action description with ordered numbering (1., 2., ...).
- Use scene information and frames to identify the possible inconsistencies between observations and the scene, if exist infer the adapted actions.

[User]
Your task is to infer the high-level action that occurs between each pair of consecutive timesteps based on provided observation sequence and scene information.

Below are the definitions of the predicates.
{predicates}

Below is an example.
`[Example]`

Now provide the action descriptions for the following observation sequence and scene information.

Information

- Instruction:
  {`instruction`}
- Object in scene:
  {`objects`}
- Object state in scene:
  {`object_state`}
- Observation Descriptions:
  {`observations`}

Action Descriptions

E.2. GPT4V-Robotics

GPT4V-Robotics — Domain Description

`[System]`

You are a domain descriptor for robotics. The user will provide three synchronized image sequences of the demonstration (top/front/back views). Your task is to describe the domain and goal of the task being demonstrated in single-line natural language based on provided demonstration and information.

Output requirements:

- Produce one clear and specific domain description which can help future task planning for same domain in new scenes, including the specific goal information.
- Start and end the domain description with triple backticks (```).

`[User]`

Your task is to describe the domain and goal of the task being demonstrated in single-line natural language based on provided demonstration and information.

Below is an example.
`[Example]`

Now provide the domain descriptions for the following demonstration and information.

Information

- Instruction:
  {`instruction`}
- Object in scene:
  {`objects`}
- Object state for each timestep:
  {`object_state`}

Domain Description

GPT4V-Robotics — Action Planning

[System]
You are an task planner for robotics. The user will provide three synchronized images of the scene (top/front/back views). Your goal is to generate task plan for scene based on provided task information and deployment scene.

Output requirements:

- Provide a step-by-step action plan to accomplish the task with each clear and concise single-line actions.
- Start the action plan with ordered numbering (1., 2., ...).

[User]
Your goal is to generate task plan for scene based on provided task information and deployment scene.

Below are the definitions of the predicates.
{predicates}

Below is an example.
`[Example]`

Now provide the action plan for the following information.

Information

- Domain Description:
  {`domain_description`}
- Instruction:
  {`instruction`}
- Objects in scene:
  {`objects`}
- Gripper state in scene:
  {`gripper_state`}

Action Plan

E.3. Critic-V

Critic-V — Feedback Generation

[System]
You are a feedback generator for robotics. The user will provide source action plan with task information and three synchronized images of the deployment scene (top/front/back views). Your task is to analyze the source action plan and give a feedback on how to refine it to succeed on the task in the deployment scene.

Output requirements:

- Provide clear and concise feedback on how to correct the action plan for deployment scene, if no problem exists, state `''No issues''`.
- Only provide feedback for the most major critical issue that would lead to task failure in the deployment scene.
- Start the feedback with an unordered dash (-).

[User]
Your task is to analyze the source action plan and give a feedback on how to refine it to succeed on the task in the deployment scene.

Below is an example.
`[Example]`

Now provide the feedback for the following initial action plan and information.

Information

- Instruction:
  {instruction}
- Objects in scene:
  {objects}
- Object state in scene:
  {object_state}
- Initial Action Plan:
  {demo_summary}

Correction Feedback

Critic-V — Correction Proposal

[System]

You are an action plan corrector for robotics. The user will provide an action plan with a feedback for better task success. Your task is to generate a SEARCH/REPLACE patch that applies the feedback to the action plan.

Output requirements:

- Use commonsense reasoning to propose a patch that applies the feedback, ensuring the action plan is executable in the deployment scene.
- First provide a reasoning about the root cause of the feedback, how to realize the feedback as an actual action patch, and why your proposed patch works.
- Propose one SEARCH block and one REPLACE block which can be applied to the original action plan to apply the feedback.
- Format your response in the following way:

Information

- (your reasoning here starting with a dash)

Correction patch:

```
<<<<<<< SEARCH
ActionToRemove_1
ActionToRemove_2
...
=======
ActionToAdd_1
ActionToAdd_2
...
>>>>>>> REPLACE
```

[User]

Your task is to generate a SEARCH/REPLACE patch that applies the feedback to the action plan.

Below is an example.
[Example]

Now provide the correction patch for the following action plan and feedbacks.

Information

- Instruction:
  {instruction}
- Objects in scene:
  {objects}
- Action Plan:
  {demo_summary}
- Feedback:
  {feedback}

Correction rationale

E.4. MoReVQA

MoReVQA — Event Parsing

[System]
You are a parsed event maker to answer the plans of the question for event parsing. The user will provide a question.

Output requirements:

1. question: Change the instruction into a question.
2. conjunction: One of (And / Or / None).
3. parse_event: A list of sub-events split by conjunction. If none exists, include one event.
4. event_object: Specify the main object(s) for each sub-event.
5. classify: One of (which / where / why / how).

[User]
You are a parsed event maker to answer the plans of the question for event parsing.

Below is an example.
[Example]

Now make the Parsed Event based on the provided information.

Information

- Instruction:
  {instruction}

Answer

MoReVQA — Event Grounding

[System]
You are a grounding module that aligns parsed events with the visual frames of a target scene. The user will provide feedback (optional), domain description, target scene, target objects, parsed events, and event objects. Your task is to create specific parsed events and objects grounded in the given scene and target objects, ensuring physical feasibility.

Output requirements: JSON format.

- parse_event: scene-grounded, physically feasible.
- event_object: specify subject and target for each grounded event.

[User]
You are a grounding module to align parsed events with the visual frames of a target scene.

Below is an example.
`[Example]`

Now make specific parsed events and event objects.

Information

- Domain Description:
  {`domain_description`}
- Target Objects:
  {`target_objects`}
- Event Queue:
  {`event_queue`}
- Event Object:
  {`event_object`}

Specific parsed events and event objects

MoReVQA — M2 Verified API Generator

[System]
You are an API generation module that verifies whether a given question is successfully executed. Your task is to generate a verify API that determines whether the question can succeed.

[User]
You are an API generation module that verifies whether a given question is successfully executed.

Below is an example.
`[Example]`

Now generate the M2 verify API.

Information

- Question:
  {`question`}

M2 verify API

MoReVQA — Verify API Executor

[System]
You are a verify-API executor that determines whether the question can succeed if events are successfully executed. The user will provide target scene, event queue, and verify API. Your task is to verify whether the question itself can succeed assuming all events execute successfully.

Output requirements:
JSON format.

- `event_queue`: executed event queue.
- `verified`: true if question can succeed; plural or `''all''` still counts as true with one representative.
- `reason`: explanation if false; otherwise `''None''`.

- `verify_action`: returns true if the question can succeed in target scene.

[User]
You are a verify-API executor that evaluates whether the given question can succeed.

Below is an example.
`[Example]`

Now generate the verified API.

Information

- verify_api:
  {`verify_api`}
- question:
  {`question`}
- event_queue:
  {`event_queue`}

Result

MoReVQA — M3 VQA API Generation Module

[System]
You are an M3 VQA API generation module. Your task is to generate exactly three procedural VQA sub-questions for each event.

Assumptions:

- All required objects exist in the scene, and the action is feasible.
- Do NOT ask existence questions.
- Focus on procedural HOW questions.

Input:

- question: event_queue

Output:

- For each event_queue, output:
  - One top-level VQA prompt phrased as ‘‘How to <event>?’’
  - Exactly three sub-questions specifying steps or constraints
- Prefer imperative, scene-grounded, concise wording.

Output format:

```
vqa("How to <event>?")
vqa(["<sub-q1>", "<sub-q2>", "<sub-q3>"])
```

[User]
You are an M3 VQA API generation module that generates exactly three procedural VQA sub-questions.

Below is an example.
`[Example]`

Now generate the M3 VQA API.

Information

- Event Queue:
  {event_queue}

M3 VQA API

MoReVQA — VQA API Executor

[System]
You are a VQA API executor that answers questions about the target scene to evaluate whether demonstrated actions can be successfully reproduced.

The user will provide:
- target_scene
- event queue and event objects

Output: Answer the VQA.

[User]
You are a VQA API executor that evaluates whether the demonstrated actions can be successfully reproduced in the given target scene.

Below is an example:
[Example]

Now make the answer for the VQA.

Information
- VQA:
  {vqa}

Result

MoReVQA — Action Planning

[System]
You are a task planner for robotics. The user will provide three synchronized images of the scene. Your goal is to generate task plans based on task information and the deployment scene.

Output requirements:
- Provide a step-by-step action plan.
- Each action must be a clear and concise single-line instruction.
- Start with ordered numbering (1., 2., ...).

[User]
Your goal is to generate an action plan for the scene based on the provided task info and deployment scene.

Below are the definitions of the predicates:
{predicates}

Below is an example:
[Example]

Now provide the action plan for the following information.

Information
- Event Queue:

{event_queue}

- Objects in scene:
  {target_objects}
- Object state in scene:
  {target_object_state}
- VQA Answer:
  {vqa_answer}

Action Plan

E.5. Statler

Statler — Action Planning

[System]
You are a task planner for robotics. The user will provide three synchronized images of the scene (top/front/back views). Your goal is to continue the task plan with a few actions and predict the state after the plan based on provided task information and the deployment scene.

Output requirements:

- Provide a step-by-step partial action plan (at most 10 steps) to head toward the goal; if the goal is reached, output ‘‘Goal reached’’.
- Start the action plan with ordered numbering (1., 2., ...).
- After the action plan, output the predicted state as a list of grounded atoms holding true for the scene after executing the action plan.
- Start each grounded atom of the state with an unordered dash (-).

[User]
Your goal is to continue the task plan with a few actions for the scene based on the provided task information and deployment scene.

Below are the definitions of the predicates.
{predicates}

Below is an example.
[Example]

Now use the same format for the following information.

Information

- Domain Description:
  {domain_decription}
- Instruction:
  {instruction}
- Object in scene:
  {objects}

Previous Actions and Current State

- Previous Actions:
  {demo_summary}

- Current State:
  {current_state}

Action Plan (at most 10 steps)

Predicted State

E.6. LLM-DM

LLM-DM — Action Recommendation

[System]
You are a PDDL action recommender for robotics. The user will provide a description of the domain and task instruction, along with the objects in the scene. Your task is to recommend the useful PDDL actions to solve the task in natural language.

Output requirements:

- The actions should have a name and a short description of what the action does in natural language.
- Start the actions with ordered numbering (1., 2., ...).
- Make the actions general enough to be reusable in different tasks within the same domain.

[User]
Your task is to recommend the useful PDDL actions to solve the task in natural language.

Below is an example.
[Example]

Now recommend the actions based on the provided demonstration information.

Demonstration Information

- Domain Description:
  {domain_description}
- Instruction:
  {instruction}
- Objects in scene:
  {objects}

Action Recommendations

LLM-DM — Predicate Proposal

[System]
You are a predicate proposer for robotics. The user will provide target initial state, base predicates, and domain/action descriptions in natural language. Your task is to propose a set of untyped predicates with associated descriptions that is useful to define the actions in PDDL format.

Output requirements:

- Provide one clear and specific list of untyped predicates with descriptions that is useful for defining the actions in PDDL format.
- Start the predicates with unordered dash (-).

[User]

Your task is to propose a set of untyped predicates with associated descriptions that is useful to define the actions in PDDL format.

Below are the definitions of the existing predicates. Do not invent predicates with duplicated meanings.
{predicates}

Below is an example.
[Example]

Now propose predicates based on the provided domain description and action descriptions.

Domain Description

- {domain_description}

Action Descriptions

- {action_description}

Predicate Proposal

LLM-DM — Action Construction

[System]
You are an action constructor for robotics. The user will provide action description in natural language and predicates. Your task is to convert the action description into a PDDL-style action definition based on the provided domain description and predicate set.

Output requirements:

- Produce one clear PDDL-style action definition for the action in the action description.
- The action definition should include the action name, parameters, preconditions, and effects.
- Start the name with ‘‘-’’ (dash followed by a space) under the **Name:** section; the action name should not overlap with predicate names.
- Start the parameters with ordered numbering (1., 2., ...) under the **Parameters:** section.
- Start and end the preconditions and effects with triple backticks (```) under the **Preconditions:** and **Effects:** sections respectively.
- All predicates used in the action definition must be from the provided predicates.
- Use predicate names exactly as given in the provided predicate list; do not invent new predicates or rename/alias any predicate.
- You can use `(not ...)`, `(and ...)` in the preconditions and effects; do not use `(or ...)`, `(when ...)`.

[User]
Your task is to convert the action description into a PDDL-style action definition based on the provided domain description and predicate set.

Below is an example.
[Example]

Now convert the action description into PDDL-style action definition.

Action Description

- {action_description}

Predicates

- {predicates}

Action Definition

LLM-DM — Problem Prediction

[System]
You are a PDDL problem predictor for robotics. The user will provide a domain description, a list of objects, a predicate set, and three synchronized images of the deployment scene (top/front/back views). Your task is to predict the PDDL-style problem definition for the target initial scene.

Output requirements:

- Produce one clear and specific PDDL-style problem definition for the target initial scene.
- The objects, initial state, and goal state should be in PDDL format separately wrapped in triple backticks (```) under **‘‘Objects:’’**, **‘‘Initial state:’’**, and **‘‘Goal state:’’** sections respectively.
- Start the content of objects, initial state, and goal state with ‘‘(:objects’’, ‘‘(:init’’, and ‘‘(:goal’’ respectively.
- All predicates in the initial state and goal state must be from the provided predicate set.

[User]
Your task is to predict the PDDL-style problem definition for the target initial scene.

Below is an example.
[Example]

Now predict the PDDL-style problem definition based on the provided information.

Domain Information

- Domain description:
  {domain_description}
- Predicates:
  {predicates}

Problem Information

- Instruction:
  {instruction}
- Objects in scene:
  {objects}
- Object state in scene:
  {object_state}

Predicted Problem Definition

LLM-DM — Domain Refinement

[System]
You are a PDDL domain refiner for robotics. The user will provide a problematic domain PDDL which failed to generate a plan for the given problem PDDL, and optionally some context about the failure if available. Your task is to diagnose the domain PDDL based on the problem PDDL and apply all necessary fixes to make the problem solvable.

Output requirements:

- First provide a reasoning about what is wrong with the original domain and how you fixed it.
- Start the refinement rationale with ‘‘-’’ (dash followed by a space) under the **‘‘Refinement Rationale:’’** section.
- Produce one refined domain PDDL that can solve the given problem PDDL; keep the name of the domain the

same as the original.
- Start and end the refined domain PDDL with triple backticks (```) under the **‘‘Refined Domain PDDL:’’** section.

[User]
Your task is to diagnose the domain PDDL based on the problem PDDL and apply all necessary fixes to make the problem solvable.

Below are the definitions of the predicates.
{predicates}

Now return the refined domain in a whole.

Domain PDDL
- {domain_pddl}

Problem PDDL
- {problem_pddl}

Context about failure (if any):
- {failure_context}

Refinement Rationale:

Refined Domain PDDL:

LLM-DM — Plan Translation

[System]
You are a plan translator for robotics. The user will provide a target plan in PDDL format. Your task is to translate the plan into clear and concise natural language steps, preserving the order and intention of each action.

Output requirements:
- Translate each action into natural language that describes what the robot is doing.
- Start the actions with ordered numbering (1., 2., ...).

[User]
Your task is to translate the plan into clear, concise, human-readable task steps, preserving the order and intention of each action.

Below is an example.
[Example]

Now translate the PDDL-style plan into natural language based on the provided information.

Target plan:
- {target_plan}

Natural-language Plan:

E.7. NeSyCR

NeSyCR — Action Prediction

[System]

You are an action predictor for robotics. The user will provide state transition information consisting of previous state, current state and state difference and scenes. Your task is to predict the executed action's semantic, preconditions and effects based on provided state transition information.

Output requirements:

- First provide a reasoning about what action was executed, why these preconditions were necessary, and why these effects occurred.
- Then produce one clear and specific action semantic inferred from the state transition.
- Start the action semantic with ``-'' (dash followed by a space).
- If preconditions is empty leave it as ``None''. The effects can never be empty.
- Use negation in preconditions and effects when relevant (e.g., `(not (GripperHolding block))`).
- Use `forall` quantifiers when the preconditions or effects should apply to all possible objects (e.g., `(forall (?x - thing) (not (GripperHolding ?x)))`).
- Except the `forall` quantifier, all atoms must be fully grounded with specific object names.

[User]

Your task is to predict the executed action's semantic, preconditions and effects based on provided state transition information.

Below are the definitions of the predicates.
{predicates}

Below are examples.
[Example]

Now predict the action based on the following information.

State Description:

- Instruction:
  {instruction}
- Objects in scene:
  {objects}
- Previous state (before action):
  {prev_state}
- Current state (after action):
  {curr_state}
- State difference:
  {state_diff}

Prediction Rationale:

Predicted Action:

Action Semantic:

Precondition:

Effect:

NeSyCR — Refinement Proposal

[System]
You are an action plan refiner for robotics. Given the task information, action information and error details, propose a refinement patch so that the target state now meets the previously unmet preconditions of the erroneous action.

Output requirements:

- Use commonsense reasoning to propose a patch that resolves the failure, ensuring the erroneous action's preconditions are satisfied.
- First provide a reasoning about the root cause of the error, how to make the erroneous action's preconditions satisfied, and why your proposed patch works.
- Propose one `SEARCH` block and one `REPLACE` block which can make preconditions of the erroneous action satisfied.
- The `SEARCH` block must contain a continuous, consecutive sequence of actions from the action plan in their exact form.
- Format your response in the following way:

Refinement rationale:
- (your reasoning here starting with a dash)

Refinement patch:

```
<<<<<<< SEARCH
ActionToRemove_1
- Preconditions:
        ...
- Effects:
        ...

ActionToRemove_2
- Preconditions:
        ...
- Effects:
        ...
...
=======
ActionToAdd_1
- Preconditions:
        ...
- Effects:
        ...

ActionToAdd_2
- Preconditions:
        ...
- Effects:
        ...
...
>>>>>>> REPLACE
```

[User]
Revise the action sequence with a SEARCH/REPLACE patch so unmet preconditions of the erroneous action are satisfied.

Below are the predicate definitions:
{predicates}

Now produce your refinement patch for the information below.

Task Information

- Instruction:
  {instruction}
- Objects in scene:
  {objects}

Action Information

- Previously executed actions:
  {executed_actions}
- Erroneous action:
  {erroneous_action}
- Remaining actions (excluding erroneous action):
  {remaining_actions}

Error Detail

- State when error occurred:
  {error_state}
- Unfulfilled Preconditions:
  {unfulfilled_preconditions}

Refinement rationale:

Refinement patch:

E.8. Common

Common — State Prediction

[System]
You are a state predictor for robotics. The user will provide you with three synchronized images of the scene (top/front/back views) and partial ground truth object states. Your task is to output state as list of grounded atoms which additionally holds true for the provided scene and information.

Output requirements:

- Start the grounded atoms with unordered dashes (-).
- Only include atoms that are strongly supported by the images, output "None" if no additional atoms can be inferred.
- Do not repeat information already provided in object state.

[User]
Your task is to output state as list of grounded atoms which additionally holds true for the provided scene and information.

Below are the definitions of the predicates.
{predicates}

Below are examples.
[Example]

Now provide the scene descriptions for the following sequences.

Information:

- Instruction:
  {instruction}
- Objects in scene:
  {objects}
- Object state in scene:
  {object_state}

State Predictions:

Common — Code Generation

[System]
You are a python code generator for robotics. The user will provide two things:

1) A set of imported Python modules and docstrings describing the available functions.
2) A specification containing:
   - Instruction: A natural language command for the robot
   - Objects in scene: List of available objects
   - Demonstration summary: A sequence of high-level action steps that serve as a high-level plan.

Output requirements:

- Use only the provided Python libraries and functions. Do not import new libraries, create new APIs, or change function signatures.
- Adhere strictly to the given docstrings. Call functions exactly as defined, with the allowed parameters.
- Return only the Python code, enclosed in triple backticks (“‘python ... ”’).
- Treat the demonstration summary as a high-level plans; follow its sequence based on the Instruction and available objects.
- Do not add extra steps unless they are implied by the demonstration summary or the instruction.

[User]
Your task is to write robot control scripts in Python code. The Python code should be general and applicable to different robotics environments.

Below are the imported Python libraries and functions that you can use, you can not import new libraries.
[Libraries and Functions]

Below shows the docstrings for these imported library functions that you must follow. You can not add additional parameters to these functions.
[API Docstring]

Below are examples.
[Example]

Now generate Python code that follows the given specification.

Specification:

- Instruction:
  {instruction}
- Objects in scene:
  {objects}
- Demonstration summary:
  {demo_summary}

Generated Code:

References

[1] 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. 2

[2] Sudhir Agarwal, Anu Sreepathy, David H Alonso, and Prarit Lamba. Llm+ reasoning+ planning for supporting incomplete user queries in presence of apis. *arXiv preprint arXiv:2405.12433*, 2024. 2

[3] Sanghyun Ahn, Wonje Choi, Junyong Lee, Jinwoo Park, and Honguk Woo. Towards reliable code-as-policies: A neuro-symbolic framework for embodied task planning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. 2

[4] Montserrat Gonzalez Arenas, Ted Xiao, Sumeet Singh, Vidhi Jain, Allen Ren, Quan Vuong, Jake Varley, Alexander Herzog, Isabel Leal, Sean Kirmani, et al. How to prompt your robot: A promptbook for manipulation skills with code as policies. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 2024. 2

[5] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 2009. 2

[6] Ashay Athalye, Nishanth Kumar, Tom Silver, Yichao Liang, Jiuguang Wang, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. From pixels to predicates: Learning symbolic world models via pretrained vision-language models. *arXiv preprint arXiv:2501.00296*, 2024. 2

[7] Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, et al. Do as i can, not as i say: Grounding language in robotic affordances. In *Proceedings of the 6th Conference on Robot Learning*, 2023. 2

[8] Kaylee Burns, Ajinkya Jain, Keegan Go, Fei Xia, Michael Stark, Stefan Schaal, and Karol Hausman. Genchip: generating robot policy code for high-precision and contact-rich manipulation tasks. In *Proceedings of the 37th IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2024. 2

[9] Yongchao Chen, Yilun Hao, Yang Zhang, and Chuchu Fan. Code-as-symbolic-planner: Foundation model-based robot planning via symbolic code generation. In *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2025. 2

[10] Sungho Choi, Seungyul Han, Woojun Kim, Jongseong Chae, Whiyoung Jung, and Youngchul Sung. Domain adaptive imitation learning with visual observation. *Advances in Neural Information Processing Systems*, 2023. 2

[11] Wonje Choi, Woo Kyung Kim, SeungHyun Kim, and Honguk Woo. Efficient policy adaptation with contrastive prompt ensemble for embodied agents. *arXiv preprint arXiv:2412.11484*, 2024. 2

[12] Wonje Choi, Jooyoung Kim, and Honguk Woo. Nesypr: Neurosymbolic proceduralization for efficient embodied reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. 2

[13] Wonje Choi, Jinwoo Park, Sanghyun Ahn, Daehee Lee, and Honguk Woo. Nesyc: A neuro-symbolic continual learner for complex embodied tasks in open domains. *arXiv preprint arXiv:2503.00870*, 2025.

[14] Cristina Cornelio and Mohammed Diab. Recover: A neuro-symbolic framework for failure detection and recovery. *arXiv preprint arXiv:2404.00756*, 2024. 2

[15] Danny Driess, Fei Xia, Mehdi S. M. Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, Wenlong Huang, Yevgen Chebotar, Pierre Sermanet, Daniel Duckworth, Sergey Levine, Vincent Vanhoucke, Karol Hausman, Marc Toussaint, Klaus Greff, Andy Zeng, Igor Mordatch, and Pete Florence. Palm-e: An embodied multimodal language model. In *arXiv preprint arXiv:2303.03378*, 2023. 2

[16] Arnaud Fickinger, Samuel Cohen, Stuart Russell, and Brandon Amos. Cross-domain imitation learning via optimal transport. *arXiv preprint arXiv:2110.03684*, 2021. 2

[17] Maria Fox, Alfonso Gerevini, Derek Long, Ivan Serina, et al. Plan stability: Replanning versus plan repair. In *ICAPS*, 2006. 7

[18] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot asp solving with clingo. *Theory and Practice of Logic Programming*, 2019. 2

[19] Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. *Advances in Neural Information Processing Systems*, 2023. 13

[20] Siyuan Huang, Zhengkai Jiang, Hao Dong, Yu Qiao, Peng Gao, and Hongsheng Li. Instruct2act: Mapping multi-modality instructions to robotic actions with large language model. *arXiv preprint arXiv:2305.11176*, 2023. 2

[21] Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022. 2

[22] Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. Voxposer: Composable 3d value maps for robotic manipulation with language models. *arXiv preprint arXiv:2307.05973*, 2023. 2

[23] Mert Inan, Aishwarya Padmakumar, Spandana Gella, Patrick Lange, and Dilek Hakkani-Tur. Multimodal contextualized plan prediction for embodied task completion. *arXiv preprint arXiv:2305.06485*, 2023. 7

[24] Adam Ishay, Zhun Yang, and Joohyung Lee. Leveraging large language models to generate answer set programs. *arXiv preprint arXiv:2307.07699*, 2023. 2

[25] Eric Jang, Alex Irpan, Mohi Khansari, Daniel Kappler, Frederik Ebert, Corey Lynch, Sergey Levine, and Chelsea Finn. Bc-z: Zero-shot task generalization with robotic imitation learning. In *Conference on Robot Learning*, 2022. 2

[26] Yunfan Jiang, Agrim Gupta, Zichen Zhang, Guanzhi Wang, Yongqiang Dou, Yanjun Chen, Li Fei-Fei, Anima Anandkumar, Yuke Zhu, and Linxi Fan. Vima: General robot manipulation with multimodal prompts. In *Fortieth International Conference on Machine Learning*, 2023. 4

[27] Byeonghwi Kim, Jinyeon Kim, Yuyeong Kim, Cheolhong Min, and Jonghyun Choi. Context-aware planning and environment-aware memory for instruction following embodied agents. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023. 2

[28] Bowen Li, Tom Silver, Sebastian Scherer, and Alexander Gray. Bilevel learning for bilevel planning. *arXiv preprint arXiv:2502.08697*, 2025. 2

[29] Yin Li, Liangwei Wang, Shiyuan Piao, Boo-Ho Yang, Ziyue Li, Wei Zeng, and Fugee Tsung. Mccoder: streamlining motion control with llm-assisted code generation and rigorous verification. *arXiv preprint arXiv:2410.15154*, 2024. 2

[30] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. *arXiv preprint arXiv:2209.07753*, 2022. 2

[31] Yichao Liang, Nishanth Kumar, Hao Tang, Adrian Weller, Joshua B Tenenbaum, Tom Silver, João F Henriques, and Kevin Ellis. Visualpredicator: Learning abstract world models with neuro-symbolic predicates for robot planning. *arXiv preprint arXiv:2410.23156*, 2024. 2

[32] Xinrui Lin, Yangfan Wu, Huanyu Yang, Yu Zhang, Yanyong Zhang, and Jianmin Ji. Clmasp: Coupling large language models with answer set programming for robotic task planning. *arXiv preprint arXiv:2406.03367*, 2024. 2

[33] Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. Llm+ p: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*, 2023. 2

[34] YuXuan Liu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Imitation from observation: Learning to imitate behaviors from raw video via context translation. In *2018 IEEE international conference on robotics and automation (ICRA)*, 2018. 2

[35] Juhong Min, Shyamal Buch, Arsha Nagrani, Minsu Cho, and Cordelia Schmid. Morevqa: Exploring modular reasoning models for video question answering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024. 12

[36] Theo X Olausson, Alex Gu, Benjamin Lipkin, Cedegao E Zhang, Armando Solar-Lezama, Joshua B Tenenbaum, and Roger Levy. Linc: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers. *arXiv preprint arXiv:2310.15164*, 2023. 2

[37] Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. *arXiv preprint arXiv:2305.12295*, 2023. 2

[38] Yuzhe Qin, Yueh-Hua Wu, Shaowei Liu, Hanwen Jiang, Ruihan Yang, Yang Fu, and Xiaolong Wang. Dexmv: Imitation learning for dexterous manipulation from human videos. In *European Conference on Computer Vision*, 2022. 2

[39] Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, Giulia Vezzani, John Schulman, Emanuel Todorov, and Sergey Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. *arXiv preprint arXiv:1709.10087*, 2017. 2

[40] Dripta S Raychaudhuri, Sujoy Paul, Jeroen Vanbaar, and Amit K Roy-Chowdhury. Cross-domain imitation from observations. In *International conference on machine learning*, 2021. 2

[41] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011. 2

[42] Sangwoo Shin, Daehee Lee, Minjong Yoo, Woo Kyung Kim, and Honguk Woo. One-shot imitation in a non-stationary environment via multi-modal skill. In *International Conference on Machine Learning*, 2023. 2

[43] Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. Alfred: A benchmark for interpreting grounded instructions for everyday tasks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020. 6

[44] Tom Silver, Soham Dan, Kavitha Srinivas, Joshua B Tenenbaum, Leslie Kaelbling, and Michael Katz. Generalized planning in pddl domains with pretrained large language models. In *Proceedings of the AAAI conference on artificial intelligence*, 2024. 2

[45] Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the 19th IEEE/CVF International Conference on Computer Vision*, 2023. 2

[46] Hao Tang, Darren Key, and Kevin Ellis. Worldcoder, a model-based llm agent: Building world models by writing code and interacting with the environment. In *Advances in Neural Information Processing Systems*, 2024. 2

[47] Sai Vemprala, Rogerio Bonatti, Arthur Bucker, and Ashish Kapoor. Chatgpt for robotics: Design principles and model abilities. 2023. *Published by Microsoft*, 2023. 2

[48] Naoki Wake, Atsushi Kanehira, Kazuhiro Sasabuchi, Jun Takamatsu, and Katsushi Ikeuchi. Gpt-4v (ision) for robotics: Multimodal task planning from human demonstration. *IEEE Robotics and Automation Letters*, 2024. 12

[49] Beichen Wang, Juexiao Zhang, Shuwen Dong, Irving Fang, and Chen Feng. Vlm see, robot do: Human demo video to robot action plan via vision language model. *arXiv preprint arXiv:2410.08792*, 2024. 2

[50] Lirui Wang, Yiyang Ling, Zhecheng Yuan, Mohit Shridhar, Chen Bao, Yuzhe Qin, Bailin Wang, Huazhe Xu, and Xiaolong Wang. Gensim: Generating robotic simulation tasks via large language models. *arXiv preprint arXiv:2310.01361*, 2023. 2

[51] Yuki Wang, Gonzalo Gonzalez-Pumariega, Yash Sharma, and Sanjiban Choudhury. Demo2code: From summarizing demonstrations to synthesizing code via extended chain-of-thought. *Advances in Neural Information Processing Systems*, 2023. 2, 12

[52] Yufei Wang, Zhou Xian, Feng Chen, Tsun-Hsuan Wang, Yian Wang, Katerina Fragkiadaki, Zackory Erickson, David Held, and Chuang Gan. Robogen: Towards unleashing infinite data for automated robot learning via generative simulation. *arXiv preprint arXiv:2311.01455*, 2023. 4

[53] Senwei Xie, Hongyu Wang, Zhanqi Xiao, Ruiping Wang, and Xilin Chen. Robotic programmer: Video instructed policy code generation for robotic manipulation. *arXiv preprint arXiv:2501.04268*, 2025. 2

[54] Dongil Yang, Minjin Kim, Sunghwan Mac Kim, Beong-woo Kwak, Minjun Park, Jinseok Hong, Woontack Woo, and Jinyoung Yeo. Llm meets scene graph: Can large language models understand and generate scene graphs? a benchmark and empirical study. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2025. 17

[55] Yijun Yang, Tianyi Zhou, Kanxue Li, Dapeng Tao, Lusong Li, Li Shen, Xiaodong He, Jing Jiang, and Yuhui Shi. Embodied multi-modal agent trained by an llm from a parallel textworld. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2024. 2

[56] Zhun Yang, Adam Ishay, and Joohyung Lee. Coupling large language models with logic programming for robust and general reasoning from text. *arXiv preprint arXiv:2307.07696*, 2023. 2

[57] Weirui Ye, Fangchen Liu, Zheng Ding, Yang Gao, Oleh Rybkin, and Pieter Abbeel. Video2policy: Scaling up manipulation tasks in simulation through internet videos. *arXiv preprint arXiv:2502.09886*, 2025. 2

[58] Takuma Yoneda, Jiading Fang, Peng Li, Huanyu Zhang, Tianchong Jiang, Shengjie Lin, Ben Picker, David Yunis, Hongyuan Mei, and Matthew R Walter. Statler: State-maintaining language models for embodied reasoning. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 2024. 13

[59] Sarah Young, Dhiraj Gandhi, Shubham Tulsiani, Abhinav Gupta, Pieter Abbeel, and Lerrel Pinto. Visual imitation made easy. In *Conference on Robot learning*, 2021. 2

[60] Tianhe Yu, Chelsea Finn, Annie Xie, Sudeep Dasari, Tianhao Zhang, Pieter Abbeel, and Sergey Levine. One-shot imitation from observing humans via domain-adaptive meta-learning. *arXiv preprint arXiv:1802.01557*, 2018. 2

[61] Di Zhang, Jingdi Lei, Junxian Li, Xunzhi Wang, Yujie Liu, Zonglin Yang, Jiatong Li, Weida Wang, Suorong Yang, Jianbo Wu, et al. Critic-v: Vlm critics help catch vlm errors in multimodal reasoning. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, 2025. 12

[62] Xian Zhou, Yiling Qiao, Zhenjia Xu, TH Wang, Z Chen, J Zheng, Z Xiong, Y Wang, M Zhang, P Ma, et al. Genesis: A generative and universal physics engine for robotics and beyond. *arXiv preprint arXiv:2401.01454*, 2024. 2

[63] Wang Bill Zhu, Miaosen Chai, Ishika Singh, Robin Jia, and Jesse Thomason. Psalm-v: Automating symbolic planning in interactive visual environments with large language models. *arXiv preprint arXiv:2506.20097*, 2025. 2