SCENE2DEMO: Self-Evolving Embodied Data Generation via Object-Action Graph

Anonymous submission

Paper ID Anonymous

Abstract

We present SCENE2DEMO*, a self-evolving framework for offline embodied data generation. Given a single real-world RGB image and a user query,* SCENE2DEMO *constructs an interactive simulated scene and generates executable task configurations, multi-view execution videos, and offline robot-learning datasets.* SCENE2DEMO *uses a structured multi-module workflow via an object-action graph, representing task generation through object-centric configurations and action transitions. Failed or incomplete executions are further refined by feedback agents that inspect visual rollouts and revise action flows through sequence modification or parameter adjustment. Across 102 automatically generated primitive scene-task pairs,* SCENE2DEMO *achieves a 71.6% execution success rate; on four representative long-horizon tasks, self-evolution improves both task success and subtask-level execution quality over primitive-only execution, and comparisons with RoboGen and GenSim2 show stronger task planning and execution performance under automated data-generation settings. Finally, behavior cloning policies achieve 96.0% and 92.0% success on two representative tasks, validating that the generated data can support downstream policy learning. Our project page is available at* `https://scene2demo-anon.github.io/`*. This paper has been under submission.*

1. Introduction

Embodied robot learning requires interaction data that is executable, physically valid, and structured enough for downstream training. In this paper, we study an offline embodied data generation problem: given a real-world scene image and a user task query, how can a system automatically produce executable robot task data? The desired outputs include an interactive simulated scene, task execution configurations, primitive action flows, multi-view execution videos, and observation-action datasets. Our goal is to build an automated data generation engine that provides reusable data for imitation learning, reinforcement learning from demonstrations, and world model training.

We propose SCENE2DEMO, a self-evolving framework for embodied data generation via an object-action graph. The input is a single RGB scene image and a user query, such as "pick up the glass". We use an existing scene construction pipeline, following Digital Cousins [5], to reconstruct interactive simulated scenes from real-world observations. On top of this scene substrate, SCENE2DEMO generates structured task definitions, BDDL-style configurations [14], object-size adjustments, primitive action flows, execution videos, and robot-learning datasets.

The first stage of SCENE2DEMO is graph-based task generation through five role-specialized modules. The object-action graph provides the abstraction behind this stage: nodes encode object-action-time configurations, while edges represent either primitive action transitions within a subtask or transfer links between consecutive subtasks. Module 1 expands the user query into a scene-grounded task description. Module 2 decomposes the task according to our symbolic task definitions. For each simple subtask, Module 3 generates the task configuration, Module 4 adjusts object size when needed to satisfy gripper constraints, and Module 5 plans an initial action flow from a primitive action library. This design is inspired by recent automated robot data generation systems such as RoboGen [31] and GenSim2 [10], but differs in two aspects: it separates the generation process into inspectable modules, and it constrains task decomposition using symbolic definitions rather than relying only on unconstrained LLM outputs.

The second stage is self-evolution for failed or incomplete subtasks. After the current action flow is executed in simulation, the Robotic Safety Inspector analyzes the current primitive action and recent multi-view frames to produce step-wise critiques. The Robotic Task Supervisor then aggregates these critiques with a longer-horizon video context, the current action flow, and previous iteration history. It revises the action flow through sequence modifi-

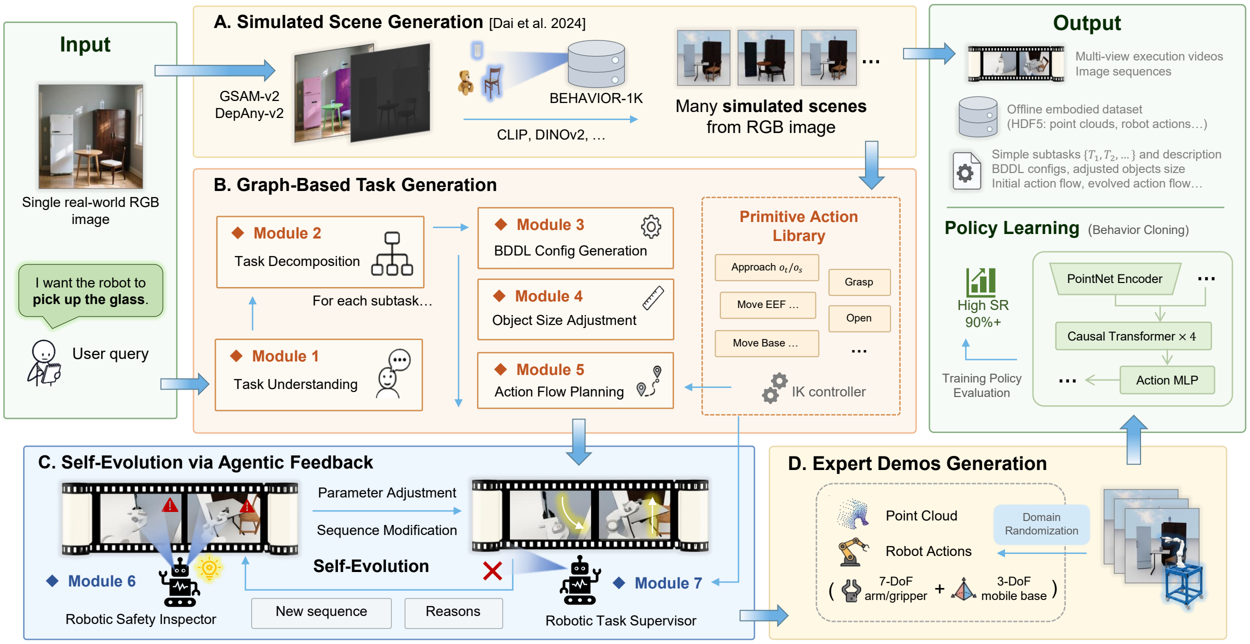


Figure 1. An introduction of our pipeline.

cation or parameter adjustment, while reusing the primitive action library as the executable action interface. This closed-loop design follows the broader idea of combining reasoning, acting, feedback, and memory in agentic workflows [27, 32, 37].

A key part of SCENE2DEMO is its graph-grounded task formulation. Instead of allowing an LLM to freely decompose a long-horizon instruction, we constrain the decomposition using symbolic definitions of simple tasks, complex tasks, action flows, and inter-task transfers. In the object-action graph, this means that a long-horizon instruction is represented as a compositional path: intra-task edges execute each simple task, while inter-task transfer edges ensure that the terminal state of one subtask can serve as the valid initialization of the next. This constraint is important in practice: repeated unconstrained LLM decomposition may produce inconsistent subtask structures for the same user query, while robot execution requires stable and parseable task units. In our repeated decomposition study, SCENE2DEMO consistently decomposes the same instruction into the same executable subtasks, while unconstrained baselines produce varying subtask counts.

We evaluate SCENE2DEMO as an automated embodied data generation engine. Across 102 primitive scene-task pairs, SCENE2DEMO achieves a 71.6% execution success rate. On four representative long-horizon tasks, self-evolution improves complete-task success and subtask-level execution quality over primitive-only execution. Direct comparisons with RoboGen and GenSim2 further show that SCENE2DEMO achieves stronger task planning and execution performance under automated data-generation settings. Finally, behavior cloning policies trained on 100 generated demonstrations achieve 96.0% success on opening a refrigerator and 92.0% success on picking up a glass, showing that the generated data can support downstream policy learning.

Our contributions are threefold. First, we introduce SCENE2DEMO, a self-evolving framework that converts a real-world image and a user query into executable embodied task data. Second, we formulate task generation through an object-action graph, improving the stability and executability of long-horizon decomposition. Third, we introduce a visual-feedback-driven self-evolution mechanism that uses step-wise inspection and task-level supervision to revise failed action flows.

2. Related Works

Generative video models and physics simulators. Generative video models have made rapid progress in visual prediction and interactive scene synthesis [3, 18, 33]. However, recent studies suggest that video generation models can still produce physically inconsistent results and may fail to generalize physical laws beyond training-like scenarios [12, 16]. For robot interaction data, physics simulators remain a more direct substrate because they provide explicit states, collision handling, and action-conditioned transitions. Simulators and benchmarks such as iGibson [13], ThreeDWorld [8], Genesis [41], ManiSkill [28], OmniGibson [14], and IsaacGym [21] support embodied interaction and robot learning in physically grounded environments. SCENE2DEMO follows this simulation-based direction, but focuses on automatically generating executable task data rather than proposing a new simulator.

Automated scene, task, and demonstration generation. Recent work has explored using foundation models to reduce the manual effort of robot scene, task, and data generation. RoboGen generates tasks, scenes, rewards, and supervision for automated robot learning [31], while GenSim2 scales robot data generation with multimodal and reasoning LLMs [10]. Other works generate or reconstruct embodied simulation environments from language, real-world images, or 3D scans, including Holodeck, ReGen, GRS, EmbodiedGen, MetaScenes, and Digital Cousins [5, 23, 30, 35, 38, 42]. These works provide important tools for creating realistic or interactive simulation substrates. SCENE2DEMO leverages interactive scene construction as the substrate for a multi-agent workflow that generates, executes, inspects, and revises embodied task data.

LLM/VLM agents and workflow-based reasoning. ReAct couples reasoning traces with environment actions [37], Reflexion improves agents through verbal feedback and episodic memory [27], and AutoGen organizes multiple customizable agents through conversation-based workflows [32]. In robotics, SayCan grounds language planning with feasible robot skills [1], Code-as-Policies represents robot policies as language-model-generated programs [17], and CaP-X benchmarks Code-as-Policy agents that compose perception and control primitives [7]. Recent robotic agent frameworks also explore using LLMs and VLMs for general robotic manipulation [36]. SCENE2DEMO follows this modular workflow perspective, but grounds it in offline embodied data generation, while the shared primitive action library serves as an executable interface.

Self-evolution, failure correction, and policy learning. Self-evolution in SCENE2DEMO is related to feedback-driven robot execution and VLM-guided failure correction. Inner Monologue shows that LLM planners can incorporate environment feedback for embodied planning [11], while recent VLM-based systems detect, explain, or correct manipulation failures through visual reasoning, constraint monitoring, or visual-symbol guidance [6, 39, 40]. LLM/VLMs have also been used for reward design, task supervision, and self-evolving embodied agents [19, 20, 29]. After self-evolution, SCENE2DEMO stores successful demonstrations for downstream learning. This connects to demonstration-generation systems such as MimicGen, SkillMimicGen, and MoMaGen [9, 15, 22], as well as behavior-cloning and visuomotor policy learning methods such as RT-1 and Diffusion Policy [2, 4].

3. Method

Given a real-world RGB image X_0 and a user query I, SCENE2DEMO first constructs an interactive simulated scene using an ACDC-based reconstruction pipeline [5], and then generates executable robot task data through two stages: graph-based task generation and self-evolution. The first stage contains five role-specialized modules: task understanding, task decomposition, BDDL configuration generation, object size adjustment, and action-flow planning. The second stage contains two feedback agents: a Robotic Safety Inspector and a Robotic Task Supervisor.

3.1. Scene-Conditioned Simulation Substrate

The simulation substrate reconstructs an interactive scene from a single real-world RGB observation. Let $\mathcal{S}$ denote the universal physical state space of the environment, where each element $s_\tau \in \mathcal{S}$ represents a possible relationship between objects at time τ. Given the real-world input image $X_0 \in \mathbb{R}^{H\times W\times 3}$, the scene construction pipeline outputs a simulated initial state $S_0 \subset \mathcal{S}$ as $S_0 \sim p(S \mid X_0; \epsilon_0)$, where ϵ_0 denotes the reconstruction error inherited from the scene construction pipeline. See Appendices A.1 and A.2.3 for scene generation details and state-definition examples.

3.2. Modules 2–3: Symbolic Task Interface

This interface directly connects Module 2, which decomposes a user query into simple subtasks, and Module 3, which turns each subtask into initial and goal conditions.

Object roles. For each simple task, we denote the target object as o_t and the two support objects as o_{s_1} and o_{s_2}. The target object o_t is the object directly manipulated or operated by the robot. The first support object o_{s_1} specifies the precondition of the target object, while the second support object o_{s_2} specifies the desired relation after execution. For tasks that only change the state of o_t itself, one or both support objects may be null.

The initialization and termination states are defined as functions of these object roles:

$$s_{\text{init}}(o_t, o_{s_1}),\; s_{\text{goal}}(o_t, o_{s_2}) \subseteq \mathcal{S}. \quad (1)$$

And we write $S \models s_{\text{init}}(T)$ or $S \models s_{\text{goal}}(T)$ when the global state S (see Appendix B.1) satisfies the corresponding BDDL-style predicate.

Simple task. A simple task is the basic unit produced by Module 2 and instantiated by Module 3.

[Simple Task] A simple task T is defined as $T = \langle o_t, o_{s_1}, o_{s_2}, s_{\text{init}}, s_{\text{goal}}\rangle$.

Module 2 is responsible for decomposing a user query into one or more simple tasks and assigning the object roles (o_t, o_{s_1}, o_{s_2}) for each subtask. Module 3 then instantiates the logical boundary of each subtask by generating BDDL-style predicates for s_{init} and s_{goal}.

Complex long-horizon task. A complex long-horizon task is represented as an ordered sequence of simple tasks. This definition constrains Module 2 to produce a parseable task sequence rather than an unconstrained natural-language plan.

[Complex Long-Horizon Task] A complex long-horizon task $\mathcal{T}$ is defined as $\mathcal{T} = \{T_1, T_2, \dots, T_K\}$, $K \geq 2$. For a single simple task, we also use $\mathcal{T} = \{T_1\}$.

3.3. Module 5: Action Flow and Action Transfer Planning

Module 5 maps symbolic task definitions into executable primitive-action programs. It contains two coupled responsibilities: intra-task action-flow planning, which executes each simple task, and inter-task action-transfer planning, which connects consecutive subtasks in a long-horizon task.

Action flow for simple tasks. A simple task T is executed by an action flow:

$$\pi(T) = \{a_1(\theta_1 \mid T), a_2(\theta_2 \mid T), \dots, a_n(\theta_n \mid T)\}, \quad (2)$$

where each a_i is an atomic primitive selected from the primitive action library, and θ_i denotes its task-conditioned parameters, such as the target object, support objects, navigation targets, or continuous control values. The primitive action library contains navigation, base movement, end-effector operations, and other task-conditioned primitives, serving as the executable interface shared by the initial planner and the self-evolution stage.

Since the action flow is generated by an LLM/VLM module, we model it as a sample from a conditional distribution $\pi \sim p_F(\pi \mid T; \epsilon_1)$, where ϵ_1 denotes generation error in action flow planning.

A simple task T is executable if there exists at least one action flow π such that $P(s_{\text{goal}} \mid s_{\text{init}}, \pi) > 0$. This condition means that the primitive sequence can move the subtask from its initial state to its goal state while preserving the irrelevant parts of the scene state.

Action transfer between subtasks. For a complex long-horizon task $\mathcal{T} = \{T_1, \dots, T_K\}$, Module 5 must also ensure that consecutive subtasks are logically connected under the global task context. Unlike intra-task action flows, the transition between two consecutive simple tasks T_k and T_{k+1} involves a semantic and physical context shift, which we call an action transfer:

$$\begin{aligned} e_k =& e(T_k, T_{k+1} \mid \mathcal{T}) \\ =& \left\{ a_{\text{end}}(\theta_{\text{end}}^{(k)} \mid \mathcal{T}), a_{\text{start}}(\theta_{\text{start}}^{(k+1)} \mid \mathcal{T}) \right\}. \end{aligned} \quad (3)$$

Here, a_{end} denotes the terminal action of subtask T_k, and a_{start} denotes the initial action of subtask T_{k+1}. Their parameters are conditioned not only on the local subtasks, but also on the global task $\mathcal{T}$, so that the transfer remains consistent with the overall long-horizon objective. The action transfer does not necessarily correspond to an additional physical operation; instead, it encodes whether the terminal state of one subtask can serve as the valid initialization of the next subtask.

We model the action-transfer sequence $E = \{e_1, \dots, e_{K-1}\}$ as

$$E \sim p_T(E \mid \mathcal{T}; \epsilon_2) = \prod_{k=1}^{K-1} p_T(e_k \mid \mathcal{T}, e_{<k}; \epsilon_2), \quad (4)$$

where $e_{<k} = \{e_1, \dots, e_{k-1}\}$ denotes previous transfer decisions. This formulation avoids treating each transfer as an isolated local decision and reflects that transfer planning should remain globally consistent across the full task sequence. Thus, the success of a long-horizon task depends on both intra-task execution feasibility and globally consistent inter-task transition planning.

3.4. Object-Action Task Graph and Compositional Reachability

The above symbolic definitions can be viewed as a path-planning problem over an object-action task graph. This graph formulation clarifies the sufficient conditions under which a decomposed long-horizon task can be composed from executable primitive-action segments.

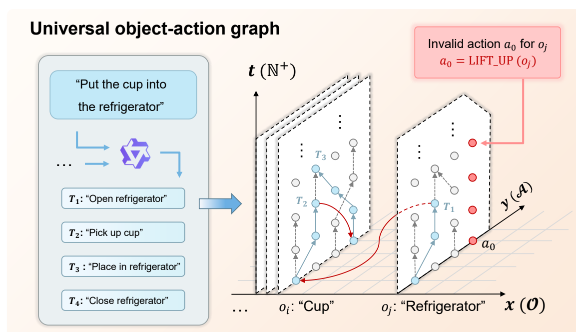


Figure 2. Universal object-action graph.

Universal object-action graph. Let $\mathcal{O}$ denote the set of all manipulable objects, $\mathcal{A}$ denote the universal set of atomic actions, and $\mathbb{N}^+$ denote the discrete temporal order within a simple task. We define a universal directed graph

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}), \qquad \mathcal{V} = \mathcal{O} \times \mathcal{A} \times \mathbb{N}^+, \quad (5)$$

where $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$ represents physically or semantically feasible transitions between object-action-time configurations. The temporal dimension denotes the relative order inside a subtask rather than a global timestamp.

Compositional reachability. Given a reconstructed scene state S_0, the universal graph $\mathcal{G}$ induces a scene-specific subgraph $\mathcal{G}_{S_0} \subset \mathcal{G}$ over the objects available in the scene and the primitive actions supported by our action library. On this subgraph, Proposition B.1 provides a sufficient condition for compositional reachability. Specifically, for a validly decomposed long-horizon task $\mathcal{T}$, if each simple task admits at least one executable primitive action flow and adjacent subtasks have compatible boundary conditions, then there exists a witness trajectory composed of intra-task action flows and inter-task transfers that reaches the final goal with non-zero probability.

This result completes the theoretical view of our graph-based task generation pipeline: Module 2 proposes symbolic subtasks with compatible boundaries, Module 3 instantiates checkable initial and goal predicates, and Module 5 searches for executable action flows and transfer links. The full scene-specific graph construction, proposition, and proof are provided in Appendix B.

3.5. Graph-Based Task Generation

Given the user query I, the simulated scene S_0, and the original RGB image X_0, the task-generation workflow produces a structured task sequence $\mathcal{T} \sim p_G(\mathcal{T} \mid I, S_0, X_0; \epsilon_3)$, where ϵ_3 denotes the uncertainty introduced by task understanding and decomposition.

The pipeline contains five role-specialized modules. Module 1 expands the user query into a scene-grounded task description. Module 2 decomposes the expanded task into simple tasks according to the symbolic definitions above. For each simple task, Module 3 generates a BDDL-style configuration that defines s_{init} and s_{goal}. Module 4 adjusts the scale of manipulable objects when the reconstructed asset size is incompatible with the robot gripper. Module 5 maps each simple task to an initial action flow and checks the inter-task transfers between consecutive subtasks.

3.6. Self-Evolution for Task Progression

The initial action flow may fail due to imperfect scene reconstruction, object-size mismatch, navigation offsets, or accumulated errors in long-horizon execution. To handle these failures as much as possible, SCENE2DEMO introduces a self-evolution mechanism.

Robotic Safety Inspector. At evolution iteration τ, for each subtask $T_k \in \mathcal{T}$, the current action flow $\hat{\pi}_\tau(T_k)$ is executed in simulation, and multi-view video frames are collected as visual feedback. Let $\mathcal{C} \subset \{\text{Global}, \text{Head}, \text{Wrist}\}$ denote the set of camera views, and let p_2 be the frame sampling interval. Let $x_{k,j}^{c,l} \in \mathbb{R}^{H\times W\times 3}$, if action a_j takes time t_j, the image sequence for the j-th action from view c is

$$X_{k,j}^{c} = \left(x_{k,j}^{c,1}, \ldots, x_{k,j}^{c,L_j}\right), \quad L_j = \min\left(\left\lceil \frac{t_j}{p_2} \right\rceil, 6\right). \tag{6}$$

The Robotic Safety Inspector analyzes execution locally. To provide temporal context, we use a look-back window p_1. For the j-th action, the inspector observes the multi-view frames corresponding to actions $\{a_{\max(j-p_1,1)}, \ldots, a_j\}$. Let

$$X_{k,j}^{\mathcal{C}} = \left\{X_{k,\max(j-p_1,1)}^{c}, \ldots, X_{k,j}^{c}\right\}_{c\in\mathcal{C}}, \tag{7}$$

and $X_k^{\mathcal{C}} = \bigcup_{j=1}^{|\hat{\pi}_\tau(T_k)|} X_{k,j}^{\mathcal{C}}$ denote the complete visual observation for the subtask. The inspector outputs a step-wise critique $m_{\tau,j} \sim p_E(m \mid X_{k,j}^{\mathcal{C}}; \epsilon_4)$, where ϵ_4 denotes the uncertainty of VLM-based visual diagnosis.

Robotic Task Supervisor. At evolution iteration τ, let $m_\tau = \{m_{\tau,j}\}_{j=1}^{|\hat{\pi}_\tau(T_k)|}$ denote the sequence of step-wise critiques for the current action flow $\hat{\pi}_\tau(T_k)$. We define the evolution history before iteration τ as $\mathcal{H}_{<\tau} = \{(\hat{\pi}_i(T_k), m_i, \hat{r}_{i+1})\}_{i=0}^{\tau-1}$, where each record stores a previously evaluated action flow, its step-wise critiques, and the supervisor's revision reason.

The Robotic Task Supervisor aggregates the current action flow, the current critiques, the visual rollout, the task context, and the previous evolution history to produce a revised action flow:

$$(\hat{\pi}_{\tau+1}(T_k), \hat{r}_{\tau+1}) \sim p_{\text{sup}}\left(\cdot \mid \hat{\pi}_\tau(T_k), m_\tau, X_k^{\mathcal{C}}, \mathcal{T}, \mathcal{H}_{<\tau}; \epsilon_4\right), \tag{8}$$

where $\hat{r}_{\tau+1}$ denotes the supervisor's global textual explanation for the revision. The revised action flow $\hat{\pi}_{\tau+1}(T_k)$ may change the primitive sequence, adjust action parameters, or modify navigation waypoints. After each failed attempt, we update the history as $\mathcal{H}_{<\tau+1} = \mathcal{H}_{<\tau} \cup \{(\hat{\pi}_\tau(T_k), m_\tau, \hat{r}_{\tau+1})\}$, which helps the supervisor avoid repeatedly trying previously failed configurations. The self-evolution loop repeats until the goal condition $s_{\text{goal}}(T_k)$ is satisfied or the maximum number of iterations is reached.

4. Experiments

We evaluate SCENE2DEMO as an automated embodied data generation engine. Our experiments answer: (1) whether the generated primitive scene-task pairs are executable, (2) whether self-evolution improves complex long-horizon task execution, (3) how SCENE2DEMO compares with existing automated data-generation pipelines, and (4) whether the generated demonstrations provide useful data for downstream policy learning.

4.1. Experimental Setup

We first evaluate primitive task generation on 102 automatically generated scene-task pairs reconstructed from 34 real-world scenes. These tasks include articulated-object operations ($\mathcal{T}_5$: open/close) and rigid-object manipulation ($\mathcal{T}_6$: pick/place). We then evaluate long-horizon execution on

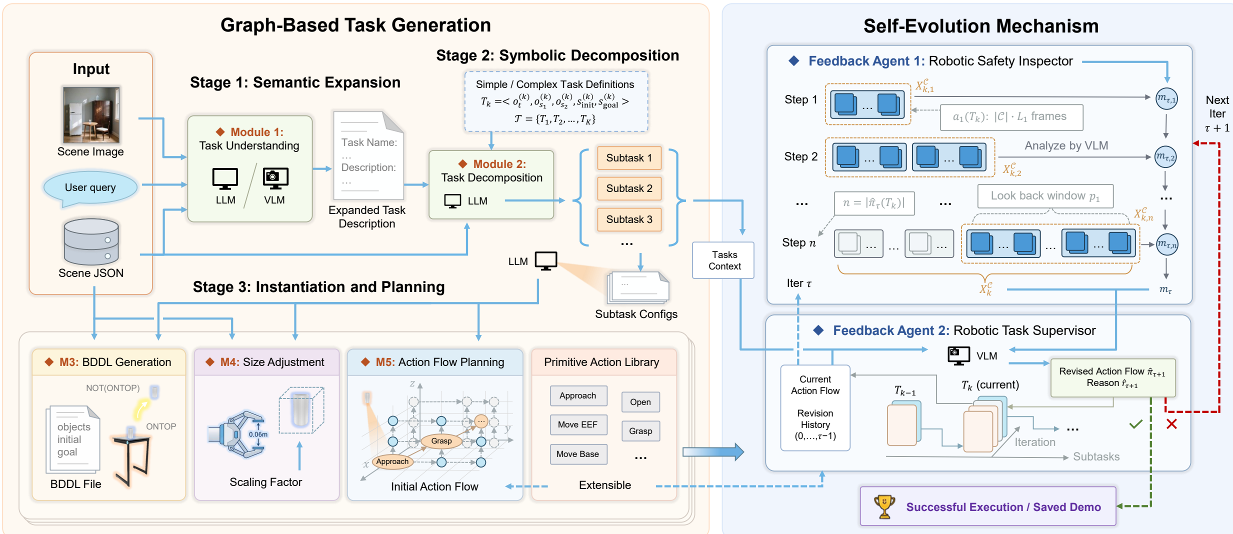


Figure 3. **Graph-Based Task Generation with Self-Evolution.** For details, see Appendix A.2,A.3.

four representative tasks: $\mathcal{T}_1$: "Put the glass into the refrigerator", $\mathcal{T}_2$: "Put the apple into the refrigerator", $\mathcal{T}_3$: "Put the apple into the bowl", $\mathcal{T}_4$: "Put the cup on the table".

Table 1. SR on 102 automatically generated scene-task pairs.

Category	Count	Success	SR (%)
$\mathcal{T}_5$: Open/Close	36	24	66.7
$\mathcal{T}_6$: Pick/Place	66	49	74.2
Total	102	73	71.6

As summarized in Table 1 and Figure 5(a), our method achieves a 71.6% overall success rate, with 66.7% success on articulated-object operations and 74.2% success on rigid-object manipulation. This result verifies that the generated task configurations and primitive action flows are executable across diverse reconstructed scenes. The scene generated examples and experiment details see Appendix C.3.

4.2. Task Execution and Baseline Comparison

We report the main metric as **SR / Subtask-Level Score**. SR measures complete-task success, while the subtask-level score provides a diagnostic measure of partial execution quality under each system's own subtask definition and evaluator. The exact number of evaluated demonstrations or episodes for each method is provided in Appendix C.4. Detailed long-horizon task breakdowns and generated task configurations are provided in Appendix C.5.

Table 2 shows three main findings. First, self-evolution consistently improves long-horizon execution over primitive-only planning. Second, compared with RoboGen and GenSim2 under automated data-generation settings, SCENE2DEMO achieves stronger complete-task success on the four representative long-horizon tasks, while also showing competitive subtask-level execution quality. Third, on simple task categories, SCENE2DEMO remains competitive: it achieves 66.7% on $\mathcal{T}_5$ and 74.2% on $\mathcal{T}_6$, while GenSim2 is stronger on Open/Close but weaker on Pick/Place.

4.3. Detailed Self-Evolution Performance

To further analyze the role of self-evolution, we retain the per-task detailed results on long-horizon tasks, including task success rate (SR), evolution success rate (ESR), and the average number of iterations for successful executions. For $\mathcal{T}_1$ and $\mathcal{T}_3$, we compare Gemini-3-Flash, GPT-5.2, and the open-source model Qwen3-VL (Specifically, all Qwen3-VL experiments use *qwen3-vl-235b-22a-thinking*); for others, we use Qwen3-VL. We set a strict failure condition: if any subtask exceeds 5 evolution iterations, the entire episode is marked as failed.

In addition to the main success-rate improvements, Table 3 reveals that self-evolution is most useful for later subtasks, where long-horizon execution accumulates navigation and placement errors. For example, later subtasks such as transporting and precise placing often require multiple correction iterations, while early subtasks such as opening or pickup are usually solved with initial planning. This observation is consistent with the qualitative evolution examples in Figure 4.

Detailed experimental results can be found in Appendix C.5,C.6. Failures in the later stages mainly stem from situations such as objects falling or getting stuck at the edges, and these situations are usually difficult to correct within a few steps. For detailed failure analysis, see Appendix C.7.

4.4. Repeated Decomposition and Ablation Studies

We also analyze the design choices and generation cost of SCENE2DEMO. First, we study repeated decomposi-

Table 2. Main task execution comparison. Each cell reports **SR / Subtask-Level Score** (%).

Method	$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}_3$	$\mathcal{T}_4$	$\mathcal{T}_5$	$\mathcal{T}_6$
M1: Ours (Primitive)	0 / 50	0 / 50	40 / 70	0 / 50	66.7 / 66.7	**74.2** / 74.2
M2: Ours (Prim+Evo)	**40 / 84**	**30 / 79**	**70 / 93**	**70 / 85**	–	–
M3: GenSim2 (Primitive)	0 / 73	5 / 78	55 / 91	0 / 68	**100 / 100**	38.8 / **82.1**
M4: RoboGen (Primitive)	0 / 50	0 / 46.5	0 / 50	0 / 50	0 / 50.2	0 / 42.8
M5: RoboGen (Prim+RL)	fail	fail	fail	fail	success	success

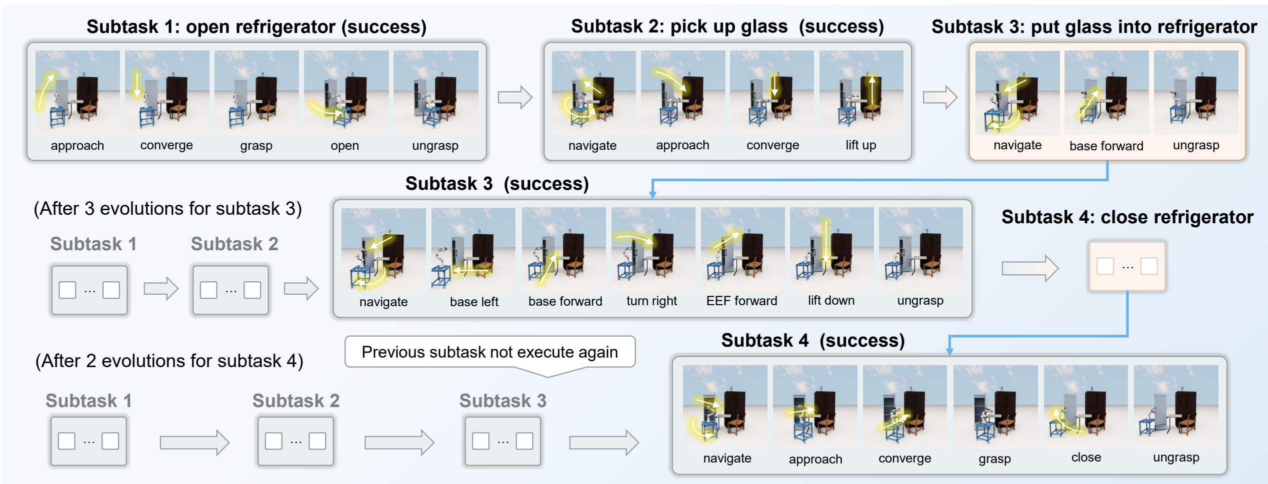


Figure 4. **Detailed Action Display of Self-Evolution.** We take the task $\mathcal{T}_1$ (Glass→Refrigerator) as an example. The task is always decomposed into four subtasks: "open refrigerator", "pick up glass", "put glass into refrigerator", and "close refrigerator".

tion consistency. As shown in Figure 5(b), for the same instruction ($\mathcal{T}_2$: "put the apple into the refrigerator"), SCENE2DEMO consistently decomposes the task into the same executable subtasks across repeated runs, while unconstrained baselines produce varying subtask counts. This result supports our claim that symbolic task definitions improve decomposition stability.

Second, we analyze the wall-clock time of SCENE2DEMO. As shown in Figure 5(c), a simple task without self-evolution takes about 2 minutes. For long-horizon tasks, initial action-flow generation takes 35s–1m20s, simulator execution takes 22s–1m30s per subtask, and a single evolution iteration takes 1m19s–2m57s using Qwen3-VL. A complete long-horizon demonstration typically takes about 15–30 minutes including all iterations. Because SCENE2DEMO is an offline data generation engine, the full self-evolution loop only needs to find a successful action configuration once, after which domain randomization can rapidly generate diverse demonstration.

Third, we conduct ablation studies on visual feedback and temporal context. As shown in Figure 5(d)(e), We investigate the impact of camera-view combinations and the look-back window p_1 used in the self-evolution mechanism. The camera-view ablation evaluates different subsets of $\{\text{Global}, \text{Head}, \text{Wrist}\}$, and the context-window ablation varies the amount of recent action history. Consistent with the original findings, the standard and full multi-view settings achieve the best overall performance, while using only a single view leads to more failures. We also observe that increasing the look-back window from $p_1 = 1$ to $p_1 = 2$ does not improve performance and may even degrade complete-task success, while also increasing VLM input length and inference latency. Therefore, we use $p_1 = 1$.

4.5. Downstream Policy Learning

Finally, we evaluate whether the generated demonstrations can support downstream policy learning. We train behavior cloning policies using 100 expert demonstrations autonomously generated by SCENE2DEMO, with randomization on target-object scale and rotation to improve data diversity. The policy uses a PointNet encoder for point clouds, fuses the point-cloud features with proprioceptive states, and applies a 4-layer Causal Transformer followed by an action MLP to predict 10-DoF actions (7-DoF arm/gripper and 3-DoF mobile base). For details, see Appendix C.8.

On two representative tasks, the learned policies achieve 96.0% success (24/25) on *Open Refrigerator* and 92.0% success (23/25) on *Pick Up Glass*. We emphasize that this experiment is not intended to claim a new policy-learning

Table 3. Self-Evolution Performance on Complex Long-Horizon Tasks.

Tasks	VLM	Demos	Subtask 1		Subtask 2			Subtask 3			Subtask 4			Complete Task	
			SR	Iter	SR	ESR	Iter	SR	ESR	Iter	SR	ESR	Iter	SR	Iter
$\mathcal{T}_1$	Gemini-3-Flash	10	**100**	0	**100**	–	0	30	30	1.7	**100**	**100**	0.7	30	2.3
	GPT-5.2	10	**100**	0	**100**	–	0	30	30	2	66.7	66.7	1.0	20	3.5
	Qwen3-VL	20	**100**	0	**100**	–	0	**45**	**45**	2.7	88.9	66.7	0.5	**40**	3.1
$\mathcal{T}_2$	Qwen3-VL	10	**100**	0	**100**	**100**	0.3	40	40	1.8	75	66.7	0.7	30	2.3
$\mathcal{T}_3$	**Gemini-3-Flash**	10	**100**	0	**70**	**62.5**	1.6	–	–	–	–	–	–	**70**	1.6
	GPT-5.2	10	**100**	0	50	16.7	0.2	–	–	–	–	–	–	50	0.2
	Qwen3-VL	10	**100**	0	50	16.7	0.4	–	–	–	–	–	–	50	0.4
$\mathcal{T}_4$	Qwen3-VL	10	**100**	0	70	60	1.0	–	–	–	–	–	–	70	1.0

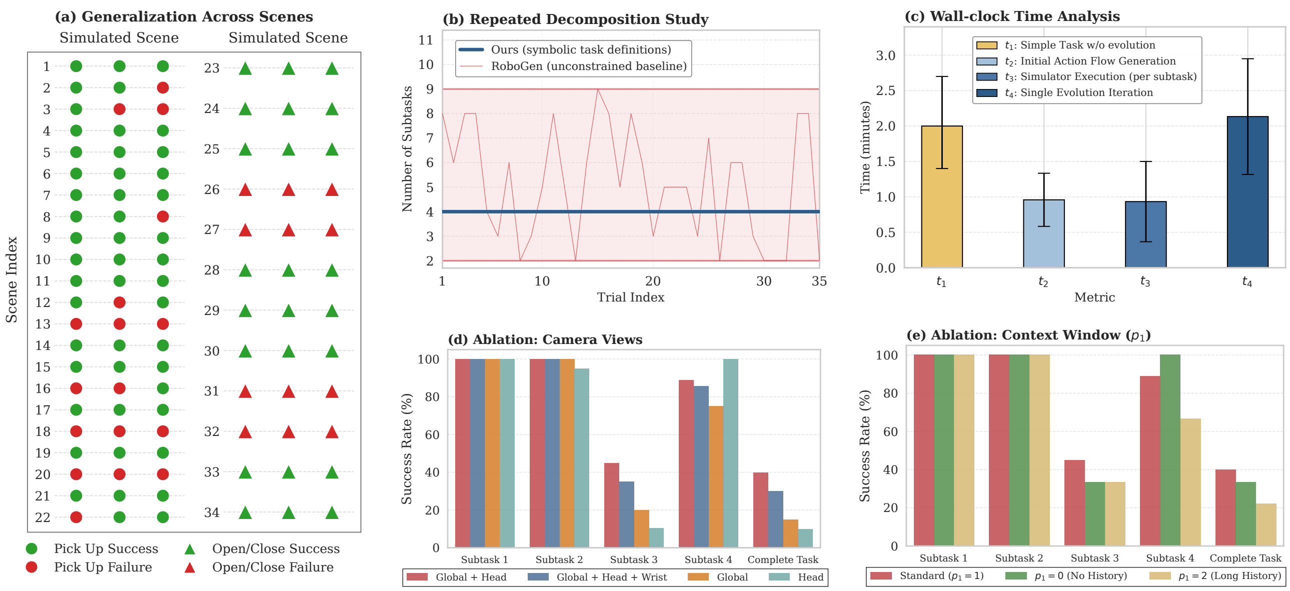


Figure 5. **Comprehensive Evaluation and Ablation Studies.** **(a)** Automatically generated 102 primitive scene-task pairs. **(b)** Repeated decomposition study on the same user query. **(c)** Wall-clock time analysis of SCENE2DEMO. Bars show average time with the observed range. **(d)** Ablation on visual input views for self-evolution. **(e)** Ablation on the context window p_1. Ablations are conducted on task $\mathcal{T}_1$.

method; rather, it verifies that the data is directly usable for downstream imitation learning.

5. Conclusion

We introduced SCENE2DEMO, a self-evolving framework for embodied data generation. Given a real-world RGB image and a user query, SCENE2DEMO constructs an interactive simulated scene and generates executable task configurations and offline robot-learning datasets. The core of SCENE2DEMO is an object-action graph formulation that connects object-centric task states through primitive action edges and inter-task transfer edges, together with a feedback-driven self-evolution mechanism for revising failed executions. The generated data can further support downstream imitation learning, showing the practical value of SCENE2DEMO as an offline data generation engine.

There are several limitations to the current framework. First, SCENE2DEMO is constrained by the capability of the primitive action library and the IK-based controller. At present, the executable action space mainly covers Open/-Close, Pick/Place, and mobile-base navigation, and does not yet support fine-grained manipulation such as folding clothes or other dexterous tasks. Future work may introduce stronger RL-based agents or learned low-level controllers to expand the action library and handle more complex manipulation skills. Second, our scene construction stage relies on the Digital Cousins pipeline [5], so failures may come from imperfect scene reconstruction or missing interactive assets, such as microwave assets that support both opening and button pressing. Future work can explore stronger scene reconstruction and asset-grounding methods to improve the robustness of the simulation substrate. Third, our self-evolution mechanism is still an early closed-loop implementation and depends heavily on the visual reasoning ability of the underlying VLM. A promising direction is to build a skill documentation library from successful tasks, where reusable correction strategies and action patterns are stored and retrieved to strengthen the feedback agents' ability to diagnose and repair future failures.

References

[1] Michael Ahn, Anthony Brohan, Noah Brown, et al. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on Robot Learning*, 2022. 3

[2] Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Joseph Dabis, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alexander Herzog, Jasmine Hsu, et al. Rt-1: Robotics transformer for real-world control at scale. In *Robotics: Science and Systems*, 2023. 3

[3] Jake Bruce, Michael D Dennis, Ashley Edwards, Jack Parker-Holder, Yuge Shi, Edward Hughes, Matthew Lai, Aditi Mavalankar, Richie Steigerwald, Chris Apps, et al. Genie: Generative interactive environments. In *Forty-first International Conference on Machine Learning*, 2024. 2

[4] Cheng Chi, Zhenjia Xu, Siyuan Feng, Eric Cousineau, Yilun Du, Benjamin Burchfiel, Russ Tedrake, and Shuran Song. Diffusion policy: Visuomotor policy learning via action diffusion. In *Robotics: Science and Systems*, 2023. 3

[5] Tianyuan Dai, Josiah Wong, Yunfan Jiang, Chen Wang, Cem Gokmen, Ruohan Zhang, Jiajun Wu, and Li Fei-Fei. Automated creation of digital cousins for robust policy learning. In *Conference on Robot Learning (CoRL)*, 2024. 1, 3, 8, 11, 17

[6] Jiafei Duan, Wilbert Pumacay, Nishanth Kumar, Yi Ru Wang, Shulin Tian, Wentao Yuan, Ranjay Krishna, Dieter Fox, Ajay Mandlekar, and Yijie Guo. Aha: A vision-language-model for detecting and reasoning over failures in robotic manipulation. In *International Conference on Learning Representations*, 2025. 3

[7] Max Fu, Justin Yu, Karim El-Refai, et al. CaP-X: A framework for benchmarking and improving coding agents for robot manipulation. *arXiv preprint arXiv:2603.22435*, 2026. 3

[8] Chuang Gan, Jeremy Schwartz, Seth Alter, Damian Mrowca, Martin Schrimpf, James Traer, Julian De Freitas, Jonas Kubilius, Abhishek Bhandwaldar, Nick Haber, et al. Threedworld: A platform for interactive multi-modal physical simulation. *arXiv preprint arXiv:2007.04954*, 2020. 2

[9] Caelan Reed Garrett, Ajay Mandlekar, Bowen Wen, and Dieter Fox. Skillmimicgen: Automated demonstration generation for efficient skill learning and deployment. In *8th Annual Conference on Robot Learning*, 2024. 3

[10] Pu Hua, Minghuan Liu, Annabella Macaluso, Yunfeng Lin, Weinan Zhang, Huazhe Xu, and Lirui Wang. Gensim2: Scaling robot data generation with multi-modal and reasoning llms. In *8th Annual Conference on Robot Learning*, 2024. 1, 3

[11] Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Noah Brown, Tomas Jackson, Linda Luu, Sergey Levine, Karol Hausman, and Brian Ichter. Inner monologue: Embodied reasoning through planning with language models. In *Conference on Robot Learning*, 2023. 3

[12] Bingyi Kang, Yang Yue, Rui Lu, Zhijie Lin, Yang Zhao, Kaixin Wang, Gao Huang, and Jiashi Feng. How far is video generation from world model? – a physical law perspective. *arXiv preprint arXiv:2411.02385*, 2024. 2

[13] Chengshu Li, Fei Xia, Roberto Martín-Martín, Michael Lingelbach, Sanjana Srivastava, Bokui Shen, Kent Elliott Vainio, Cem Gokmen, Gokul Dharan, Tanish Jain, Andrey Kurenkov, Karen Liu, Hyowon Gweon, Jiajun Wu, Li Fei-Fei, and Silvio Savarese. igibson 2.0: Object-centric simulation for robot learning of everyday household tasks. In *Proceedings of the 5th Conference on Robot Learning*, pages 455–465. PMLR, 2022. 2

[14] Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-Martín, Chen Wang, Gabrael Levine, Michael Lingelbach, Jiankai Sun, et al. Behavior-1k: A benchmark for embodied ai with 1,000 everyday activities and realistic simulation. In *Conference on Robot Learning*, pages 80–93. PMLR, 2023. 1, 2, 11, 12

[15] Chengshu Li, Mengdi Xu, Arpit Bahety, Hang Yin, Yunfan Jiang, Huang Huang, Josiah Wong, Sujay Garlanka, Cem Gokmen, Ruohan Zhang, et al. Momagen: Generating demonstrations under soft and hard constraints for multi-step bimanual mobile manipulation. *arXiv preprint arXiv:2510.18316*, 2025. 3

[16] Dacheng Li, Yunhao Fang, Yukang Chen, Shuo Yang, Shiyi Cao, Justin Wong, Michael Luo, Xiaolong Wang, Hongxu Yin, Joseph E Gonzalez, et al. Worldmodelbench: Judging video generation models as world models. *arXiv preprint arXiv:2502.20694*, 2025. 2

[17] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International conference on robotics and automation (ICRA)*, pages 9493–9500. IEEE, 2023. 3

[18] Yixin Liu, Kai Zhang, Yuan Li, Zhiling Yan, Chujie Gao, Ruoxi Chen, Zhengqing Yuan, Yue Huang, Hanchi Sun, Jianfeng Gao, et al. Sora: A review on background, technology, limitations, and opportunities of large vision models. *arXiv preprint arXiv:2402.17177*, 2024. 2

[19] Jason Ma, William Liang, Hung-Ju Wang, Yuke Zhu, Linxi Fan, Osbert Bastani, and Dinesh Jayaraman. Dreureka: Language model guided sim-to-real transfer. Robotics: Science and Systems (RSS), 2024. 3

[20] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *International Conference on Learning Representation (ICLR)*, 2024. 3

[21] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, et al. Isaac gym: High performance gpu-based physics simulation for robot learning. *arXiv preprint arXiv:2108.10470*, 2021. 2

[22] Ajay Mandlekar, Soroush Nasiriany, Bowen Wen, Iretiayo Akinola, Yashraj Narang, Linxi Fan, Yuke Zhu, and Dieter Fox. Mimicgen: A data generation system for scalable robot learning using human demonstrations. In *7th Annual Conference on Robot Learning*, 2023. 3

[23] Phat Tan Nguyen, Tsun-Hsuan Wang, Zhang-Wei Hong, Erfan Aasi, Andrew Silva, Guy Rosman, Sertac Karaman, and

Daniela Rus. Regen: Generative robot simulation via inverse design. In *The Thirteenth International Conference on Learning Representations*, 2025. 3

[24] Maxime Oquab, Timothée Darcet, Théo Moutakanni, Huy Vo, Marc Szafraniec, Vasil Khalidov, Pierre Fernandez, Daniel Haziza, Francisco Massa, Alaaeldin El-Nouby, et al. Dinov2: Learning robust visual features without supervision. *arXiv preprint arXiv:2304.07193*, 2023. 11

[25] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, pages 8748–8763, 2021. 11

[26] Nikhila Ravi, Valentin Gabeur, Yuan-Ting Hu, Ronghang Hu, Chaitanya Ryali, Tengyu Ma, Haitham Khedr, Roman Rädle, Chloe Rolland, Laura Gustafson, Eric Mintun, Junting Pan, Kalyan Vasudev Alwala, Nicolas Carion, Chao-Yuan Wu, Ross Girshick, Piotr Dollár, and Christoph Feichtenhofer. Sam 2: Segment anything in images and videos. *arXiv preprint arXiv:2408.00714*, 2024. 11

[27] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023. 2, 3

[28] Stone Tao, Fanbo Xiang, Arth Shukla, Yuzhe Qin, Xander Hinrichsen, Xiaodi Yuan, Chen Bao, Xinsong Lin, Yulin Liu, Tse kai Chan, Yuan Gao, Xuanlin Li, Tongzhou Mu, Nan Xiao, Arnav Gurha, Viswesh Nagaswamy Rajesh, Yong Woo Choi, Yen-Ru Chen, Zhiao Huang, Roberto Calandra, Rui Chen, Shan Luo, and Hao Su. Maniskill3: Gpu parallelized robotics simulation and rendering for generalizable embodied ai. *Robotics: Science and Systems*, 2025. 2

[29] Wanxin Tian, Shijie Zhang, Kevin Zhang, Xiaowei Chi, Chunkai Fan, Junyu Lu, Yulin Luo, Qiang Zhou, Yiming Zhao, Ning Liu, et al. Seea-r1: Tree-structured reinforcement fine-tuning for self-evolving embodied agents. *arXiv preprint arXiv:2506.21669*, 2025. 3

[30] Xinjie Wang, Liu Liu, Yu Cao, Ruiqi Wu, Wenkang Qin, Dehui Wang, Wei Sui, and Zhizhong Su. Embodiedgen: Towards a generative 3d world engine for embodied intelligence. *arXiv preprint arXiv:2506.10600*, 2025. 3

[31] 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. In *Forty-first International Conference on Machine Learning*, 2024. 1, 3

[32] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024. 2, 3

[33] Wilson Yan, Yunzhi Zhang, Pieter Abbeel, and Aravind Srinivas. Videogpt: Video generation using vq-vae and transformers. *arXiv preprint arXiv:2104.10157*, 2021. 2

[34] Lihe Yang, Bingyi Kang, Zilong Huang, Zhen Zhao, Xiaogang Xu, Jiashi Feng, and Hengshuang Zhao. Depth anything v2. *Advances in Neural Information Processing Systems*, 37:21875–21911, 2024. 11

[35] Yue Yang, Fan-Yun Sun, Luca Weihs, Eli VanderBilt, Alvaro Herrasti, Winson Han, Jiajun Wu, Nick Haber, Ranjay Krishna, Lingjie Liu, et al. Holodeck: Language guided generation of 3d embodied ai environments. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16227–16237, 2024. 3

[36] Yi Yang, Kefan Gu, Yuqing Wen, Hebei Li, Yucheng Zhao, Tiancai Wang, and Xudong Liu. Maniagent: An agentic framework for general robotic manipulation. *arXiv preprint arXiv:2510.11660*, 2025. 3, 11

[37] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. 2, 3

[38] Huangyue Yu, Baoxiong Jia, Yixin Chen, Yandan Yang, Puhao Li, Rongpeng Su, Jiaxin Li, Qing Li, Wei Liang, Zhu Song-Chun, Tengyu Liu, and Siyuan Huang. Metascenes: Towards automated replica creation for real-world 3d scans. In *Conference on Computer Vision and Pattern Recognition(CVPR)*, 2025. 3

[39] Xianchao Zeng, Xinyu Zhou, Youcheng Li, Jiayou Shi, Tianle Li, Liangming Chen, Lei Ren, and Yong-Lu Li. Diagnose, correct, and learn from manipulation failures via visual symbols. *arXiv preprint arXiv:2512.02787*, 2025. 3

[40] Enshen Zhou, Qi Su, Cheng Chi, Zhizheng Zhang, Zhongyuan Wang, Tiejun Huang, Lu Sheng, and He Wang. Code-as-monitor: Constraint-aware visual programming for reactive and proactive robotic failure detection. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 6919–6929, 2025. 3

[41] 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

[42] Alex Zook, Fan-Yun Sun, Josef Spjut, Valts Blukis, Stan Birchfield, and Jonathan Tremblay. Grs: Generating robotic simulation tasks from real-world images. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 594–603, 2025. 3

Impact Statement

This work aims to support scalable and reproducible embodied AI research by automatically generating executable robot interaction data in simulation. By producing task configurations, action traces, execution videos, and offline learning datasets, SCENE2DEMO may reduce the manual effort required to construct embodied training data and make robot-learning experiments easier to inspect and reproduce. Since all demonstrations are generated and refined in simulation, the framework can also help identify unsafe or failed behaviors before real-world deployment. However, the quality and diversity of the generated data depend on the underlying scene reconstruction tools, simulation assets, primitive action library, and VLM feedback quality. We therefore encourage careful validation before using such generated data in real-world robotic systems.

A. Implementation Details

A.1. Simulated Scene Generation Details

This section presents the complete process of simulated scene generation described in Section 3.1, which follows the reconstruction method of [5]. Given a single real-world RGB observation, the scene generation pipeline reconstructs an interactive simulated scene while preserving semantic affordances and object states. The pipeline is as follows:

1. **Real-world Extraction:** Obtain data and information from the real world, input the camera's internal parameter matrix K and a single RGB image X, and output a set of object representations including object labels, masks, point clouds, etc., for the subsequent creation of simulated scenes. The main methods include observing and extracting objects (using LLM, *GroundedSAM-v2* [26]), followed by depth estimation (*DepthAnything-v2* [34]) and extracting point clouds, etc.
2. **Assets Matching:** Hierarchical search is conducted on the asset dataset *BEHAVIOR-1K* [14]. Through the *CLIP* and *DINOv2* models [24, 25], the most suitable assets are matched for each object. This step outputs the virtual assets and snapshots corresponding to each cousin in the corresponding direction.
3. **Simulated Scene Generation:** Improve the details such as asset location determination and scene physical stability (for example, use LLM to determine whether to install objects on the wall). And based on the *OmniGibson* platform [14], simulation scenes are generated, providing physical simulation and rendering including object states, etc.

ManiAgent [36] directly estimates the 3D coordinates of real objects from RGB images using *Florence-2*, and then employs *AnyGrasp* to estimate the 6-DoF pose of the target position and plan the path. In contrast, our work estimates the 3D coordinates of each object from RGB images while reconstructing the entire simulated environment. It retains advanced scene attributes, such as spatial object layout S_0, key semantics, and physical visibility, which offers advantages such as maintaining global physical consistency and preserving the state of complex long-horizon tasks.

A.2. Implementation Details of Graph-Based Task Generation

In this section, we provide a comprehensive breakdown of the Graph-Based Task Generation pipeline, corresponding to the left component of Figure 3 in the main paper. This process utilizes Large Language Models (LLMs) or Vision-Language Models (VLMs) to autonomously translate vague user commands into executable, physically grounded task specifications. The pipeline consists of three sequential stages: Semantic Expansion, Subtask Decomposition, and Task Instantiation.

A.2.1. Semantic Expansion

The goal of this stage is to ground a potentially vague user query into a precise scene context.

Input:

- **User Keyword ($\mathcal{I}$):** This can be a specific object name (e.g., "refrigerator", "cup"), a specific action (e.g., "pick up the cup"), or a high-level abstract instruction requiring reasoning (e.g., "put the cup in the refrigerator", "clean the room").
- **Scene Configuration (*scene_json*):** A JSON file containing the ground-truth metadata of the reconstructed scene, including object IDs, 3D coordinates, and bounding boxes. This is essential for grounding semantic concepts to specific simulation assets (e.g., mapping "cup" to the unique ID *glass_0*).
- **Scene Observation (Optional):** An RGB image of the scene. If provided, a VLM is employed for visual reasoning; otherwise, an LLM processes the textual JSON data.

Output:

- **Task Name (*task_name*):** A strictly formatted string where all objects are replaced by their unique scene IDs (e.g., "put glass_0 into cabinet_0").
- **Detailed Task Description (*task_message_detail*):** A step-by-step narrative of the task logic generated by the model. For example: *"This is a kitchen scene. There is a cup (glass_0) on top of a cabinet (cabinet_0). The goal is to place the cup inside the cabinet. The robot must first open the cabinet door, approach the cup, grasp it, lift it up, navigate to the cabinet, and place it inside..."*

A.2.2. Subtask Decomposition

This stage breaks down the long-horizon task into a sequence of atomic simple tasks, bridging the definition in

Section 3.2 of the main paper.

Input: The *task_name* and *task_message_detail* from A.2.1, along with the *scene_json*.

Output: A sequential list of subtask configurations (JSON). For a complex task $\mathcal{T} = \{T_1, T_2, \ldots, T_K\}$, each subtask configuration contains:

- **Subtask Name:** The atomic activity (e.g., "Open Cabinet", "Pick up Cup").
- **Description:** Detailed instructions specific to this subtask.
- **Target Object ID (o_t):** The primary object being manipulated.
- **Support Object ID (o_{s_1}, o_{s_2}):** The two support objects are associated with the state definitions in the task: the first support object o_{s_1} specifies the initial state relative to the target object (e.g., the table supporting the cup), while the second support object o_{s_2} specifies the goal state relative to the target object (e.g., the container receiving the cup), they all extracted from the *scene_json*.
- **BDDL Category:** The specific object category mapped to the OmniGibson asset library, used for subsequent BDDL file generation.

A.2.3. Subtask Instantiation

For each decomposed subtask, we perform three parallel instantiation steps to ensure execution feasibility: BDDL Generation, Object Size Adjustment, and Action Flow Planning.

1. BDDL Generation (Initial State and Success Detection) We utilize the Behavior Domain Definition Language [14] (BDDL) to define the initial state (s_{init}) and success criteria (s_{goal}) for the simulation. The LLM generates a BDDL file based on task-specific templates.

This module takes the subtask configuration as input, including the target object and support-object categories, and outputs a standard `.bddl` file. Here are some example predicates:

- s_{init}: *inroom(robot, kitchen)*, *ontop(glass_0, table_0)*.
- s_{goal}: *open(fridge_0)*, *inside(apple_0, bowl_0)*, or *not(ontop(cup_0, table_0))*.

2. Object Size Adjustment (Physical Feasibility) To ensure objects are graspable by the robot's parallel gripper (maximum width $\approx$ 0.06m, depending on the robot size parameters), we dynamically adjust object scales based on their bounding box extents defined in *scene_json*. We classify objects into two types:

- **Type A (Fixed/Articulated Fixtures):** Objects such as refrigerators, cabinets, microwaves, ovens, drawers, dishwashers, and tables. These are operated via handles or are static supports.
- **Type B (Manipulable Items):** Objects such as cups, glasses, apples, bottles, boxes, bowls, and knives. These must fit inside the gripper.

For type A, no scaling is applied, and return scale factor $S = 1.0$. For type B, we calculate the minimum dimension on the horizontal plane: $d_{\text{min}} = \min(\text{bbox}_x, \text{bbox}_y)$. If $d_{\text{min}} > 0.06$m, we compute a scaling factor S such that $S \times d_{\text{min}} \approx 0.05$m (ideal grasping width). Otherwise, $S = 1.0$.

3. Action Flow Planning with Primitive Action Library Module 5 maps each simple subtask T to an executable action flow using a primitive action library. Formally, the output action flow is defined as

$$\pi(T) = \{a_i(\theta_i \mid T)\}_{i=1}^{n}, \tag{9}$$

where each a_i is an atomic primitive selected from the library, and θ_i denotes its task-conditioned parameters. These parameters may include the target object o_t, support objects o_{s_1}, o_{s_2}, navigation targets, joint limits, or continuous control values such as distances and angles. This notation reflects that primitive actions are not restricted to the target object alone, but may also involve support objects or scene-level navigation goals.

We define a library of **21 atomic primitives** in the *BasicSkill* class, covering navigation, base movement, end-effector (EEF) operations, articulated-object interaction, and other task-conditioned primitives. This library serves as the executable interface shared by the initial action-flow planner and the self-evolution stage. During self-evolution, the Robotic Task Supervisor can revise the primitive sequence, replace unsuitable primitives, or tune primitive parameters while remaining within the valid action library.

Each primitive action is represented by an integer primitive ID and a structured parameter dictionary. The ID denotes the primitive type rather than an object identifier. For example, an action can be represented as

$$(\text{ID}, \text{params}), \tag{10}$$

where ID indexes a primitive such as *NAVIGATE_TO_SUPPORT*, *MOVE_BASE_FORWARD*, *GRASP*, or *ARTICULATE_OPEN*, and params stores the required object references or continuous control values. Thus, a format such as {*"ID": value*} should be interpreted as a primitive-ID-to-parameter mapping, not as an object-ID mapping.

We categorize the 21 atomic primitives according to their parameter requirements and functional domains. This categorization distinguishes between context-aware primitives that rely on the underlying solver for pose determination and parameterized primitives that allow explicit tuning during self-evolution.

3.1. Parameter-Free Context-Aware Actions.

These actions do not require explicit numerical parameters. They rely on the underlying motion planner, object state, and task context to determine the target pose or interaction target.

- *APPROACH*: Move the end-effector to a pre-grasp or pre-interaction approach pose.
- *CONVERGE*: Finely align the end-effector with the target handle or object centroid.
- *GRASP*: Close the gripper to grasp the target object.
- *UNGRASP*: Open the gripper to release the currently held object.
- *RETREAT*: Move the end-effector away from the target to a safe retreat pose.
- *NAVIGATE_TO_TARGET*: Move the robot base near the target object o_t.
- *NAVIGATE_TO_SUPPORT*: Move the robot base near the support object, typically o_{s_2}.

3.2. End-Effector Translation.

These primitives move the end-effector along a specified direction. The parameter is a continuous distance value in meters, represented as {*"ID": distance_in_meters*}.

- *LIFT_EEF_UP*: Lift the end-effector vertically upward.
- *LIFT_EEF_DOWN*: Move the end-effector vertically downward.
- *MOVE_EEF_FORWARD*: Move the end-effector forward.
- *MOVE_EEF_BACKWARD*: Move the end-effector backward.
- *MOVE_EEF_LEFT*: Move the end-effector to the left.
- *MOVE_EEF_RIGHT*: Move the end-effector to the right.

3.3. Mobile Base Translation and Rotation.

These primitives control the mobile base relative to the robot's current heading. For translation primitives, the parameter is a distance in meters, represented as {*"ID": distance_in_meters*}.

- *MOVE_BASE_FORWARD*: Move the robot base forward.
- *MOVE_BASE_BACKWARD*: Move the robot base backward.
- *MOVE_BASE_LEFT*: Move the robot base left by holonomic sliding.
- *MOVE_BASE_RIGHT*: Move the robot base right by holonomic sliding.

For rotation primitives, the parameter is an angle in degrees, represented as {*"ID": angle_in_degrees*}.

- *TURN_BASE_LEFT*: Rotate the robot base counter-clockwise.
- *TURN_BASE_RIGHT*: Rotate the robot base clockwise.

3.4. Articulated-Object Interaction.

These primitives interact with articulated objects such as doors, drawers, and refrigerators. The parameter specifies normalized joint limits or physical joint limits, represented as {*"ID": [min_limit, max_limit]*}. We also allow *None* as the parameter value, in which case the default solver limits are used.

- *ARTICULATE_OPEN*: Execute an opening motion for a revolute or prismatic joint.
- *ARTICULATE_CLOSE*: Execute a closing motion for a revolute or prismatic joint.

A.3. Implementation Details of Self-Evolution Mechanism

This section provides implementation details of the **Self-Evolution Mechanism** in Figure 3. The main text describes the mathematical formulation of the Robotic Safety Inspector and the Robotic Task Supervisor. Here, we focus on the practical execution protocol, the VLM prompt logic, and the reset mechanism used during iterative refinement.

Resetting before each evolution attempt. A failed execution may irreversibly alter the scene, such as dropping an object, pushing it out of reach, or leaving the robot in an invalid configuration. Therefore, before evaluating each revised action flow, we reset the simulator to a saved subtask initial state $\bar{S}^{(k)}_{\text{init}}$, which satisfies the symbolic initial condition $s_{\text{init}}(T_k)$. This ensures that different candidate action flows are evaluated from the same subtask boundary state, making the self-evolution process reproducible and preventing later attempts from being affected by previous failed rollouts.

A.3.1. Visual Data Pre-processing

For each action step in the current action flow $\hat{\pi}_\tau(T_k)$, we capture RGB frames from multiple camera views $\mathcal{C} \subset \{\text{Global}, \text{Head}, \text{Wrist}\}$. The *Global View* provides a static scene overview, the *Head View* captures the robot-centered perspective, and the *Wrist View* focuses on fine-grained manipulation near the end-effector. As defined in the main text, each action-level visual sequence is uniformly downsampled to at most 6 frames, and the inspector receives a sliding-window visual context $X^{\mathcal{C}}_{k,j}$ containing the current action and the previous p_1 action steps. We use $p_1 = 1$ by default in the main experiments.

A.3.2. Step-wise Critique by the Robotic Safety Inspector

The Robotic Safety Inspector verifies each primitive action locally. For the j-th action in $\hat{\pi}_\tau(T_k)$, the inspector receives the action description and the local visual context $X^{\mathcal{C}}_{k,j}$, and outputs a step-wise critique $m_{\tau,j}$. The prompt asks the VLM to check three aspects:

- **Motion Verification:** whether the robot movement matches the intended primitive action, such as whether the base reaches the target region or the end-effector moves in the expected direction.
- **State Contrast:** whether the intended state transition occurs by comparing the beginning and ending frames, such as whether the object is actually grasped or placed.

- **Anomaly Detection:** whether unintended collisions, object displacement, failed grasps, or unsafe motions occur during execution.

The output $m_{\tau,j}$ is a concise textual diagnosis of the current action step. The full critique sequence for the current attempt is denoted as $m_\tau = \{m_{\tau,j}\}_{j=1}^{|\hat{\pi}_\tau(T_k)|}$, consistent with the notation in the main text.

A.3.3. Task-Level Revision by the Robotic Task Supervisor

The Robotic Task Supervisor performs global action-flow revision. It receives the current action flow $\hat{\pi}_\tau(T_k)$, the step-wise critique sequence m_τ, the complete visual rollout $X_k^{\mathcal{C}}$, the task context $\mathcal{T}$, and the evolution history $\mathcal{H}_{<\tau}$. The history follows the definition in the main text:

$$\mathcal{H}_{<\tau} = \{(\hat{\pi}_i(T_k), m_i, \hat{r}_{i+1})\}_{i=0}^{\tau-1},$$

where each record stores a previously evaluated action flow, its step-wise critiques, and the supervisor's revision reason.

The supervisor prompt enforces three reasoning steps:

1. **Root Cause Analysis:** identify the earliest failed or risky step, rather than only describing the final failure.
2. **Action-Flow Revision:** revise the primitive sequence, tune action parameters, or modify navigation waypoints based on the diagnosis.
3. **Library Constraint:** ensure that the revised action flow only uses valid primitives from the primitive action library in Appendix A.2.

The supervisor outputs a revised action flow $\hat{\pi}_{\tau+1}(T_k)$ and a global textual explanation $\hat{r}_{\tau+1}$. After each failed attempt, the history is updated with the evaluated action flow, its critiques, and the revision reason. This memory prevents the supervisor from repeatedly trying previously failed configurations.

A.3.4. Algorithm Summary

Algorithm A.3.4 summarizes the complete self-evolution procedure. The key implementation detail is that every candidate action flow is executed after resetting the simulator to the saved initial state $\bar{S}_{\text{init}}^{(k)}$ of the current subtask.

[h] [1] **Input:** Initial action flow $\hat{\pi}_0(T_k)$, subtask T_k, task context $\mathcal{T}$, camera views $\mathcal{C}$, saved subtask initial state $\bar{S}_{\text{init}}^{(k)}$, max iterations $\tau_{\max}$ Initialize evolution history $\mathcal{H}_{<0} \leftarrow \emptyset$ $\tau = 0$ to $\tau_{\max}$ **Reset** simulator to $\bar{S}_{\text{init}}^{(k)}$ **Execute** $\hat{\pi}_\tau(T_k)$ in simulation and capture visual rollout $X_k^{\mathcal{C}}$ success condition $s_{\text{goal}}(T_k)$ is satisfied **Output:** $\hat{\pi}_\tau(T_k)$ **Terminate** Initialize critique sequence $m_\tau \leftarrow \emptyset$ each action step j in $\hat{\pi}_\tau(T_k)$ Construct sliding-window visual context $X_{k,j}^{\mathcal{C}}$ $m_{\tau,j} \leftarrow \text{VLM}_{\text{Inspector}}\left(a_j, X_{k,j}^{\mathcal{C}}\right)$ $m_\tau.\text{append}(m_{\tau,j})$ $(\hat{\pi}_{\tau+1}(T_k), \hat{r}_{\tau+1}) \leftarrow \text{VLM}_{\text{Supervisor}}\left(\hat{\pi}_\tau(T_k), m_\tau, X_k^{\mathcal{C}}, \mathcal{T}, \mathcal{H}_{<\tau}\right)$ $\mathcal{H}_{<\tau+1} \leftarrow \mathcal{H}_{<\tau} \cup \{(\hat{\pi}_\tau(T_k), m_\tau, \hat{r}_{\tau+1})\}$ **Output:** Failure

B. Theoretical Supplement

This appendix provides additional theoretical details for the graph-based task formulation used in SCENE2DEMO. We first present the object-action task graph and the compositional reachability proposition. We then provide a probabilistic interpretation of action transfer as stochastic functional switching, followed by the proof of Proposition B.1.

B.1. Object-Action Task Graph and Compositional Reachability

Universal object-action graph. Let $\mathcal{O}$ denote the set of all theoretically manipulable objects, $\mathcal{A}$ denote the universal set of atomic actions, and $\mathbb{N}^+$ denote the discrete temporal order within a simple task. We define a universal directed graph

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}), \qquad \mathcal{V} = \mathcal{O} \times \mathcal{A} \times \mathbb{N}^+, \tag{11}$$

where $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$ represents physically or semantically feasible transitions between object-action-time configurations. The temporal dimension denotes the relative order inside a subtask rather than a global timestamp.

Scene-specific graph. Given a reconstructed scene state S_0, we instantiate a scene-specific object-action graph:

$$\mathcal{G}_{S_0} = (\mathcal{V}_{S_0}, \mathcal{E}_{S_0}) \subset \mathcal{G}, \qquad \mathcal{V}_{S_0} = \mathcal{O}_{S_0} \times \mathcal{A}_{\text{prim}} \times \mathbb{N}^+, \tag{12}$$

where $\mathcal{O}_{S_0}$ contains the objects present in the reconstructed scene, and $\mathcal{A}_{\text{prim}}$ is the primitive action library. The global state at time τ is the aggregate of all object-pair relations in the scene:

$$S_\tau = \{s_\tau(o_i, o_j)\}_{\{o_i, o_j\} \subset \mathcal{O}_{S_0}} \subset \mathcal{S}. \tag{13}$$

This formulation allows us to describe both local subtask transitions and global scene consistency.

Intra-task and inter-task edges. For a simple task T_k, the action flow corresponds to a sequence of intra-task edges in the task-specific object-action slice required by T_k. Following the notation in the main text, we write $\pi(T_k) = \{a_i(\theta_i \mid T_k)\}_{i=1}^{n_k}$, where each primitive action a_i may be parameterized by target objects, support objects, navigation goals, or continuous control values. These intra-task edges represent executable primitive transitions that move the scene toward the symbolic goal predicate of T_k.

For two consecutive subtasks T_k and T_{k+1}, the action transfer $e_k = e(T_k, T_{k+1} \mid \mathcal{T})$ corresponds to an inter-task edge that links the terminal context of T_k to the initial context of T_{k+1}. To describe this context switch, we introduce the augmented state

$$S_\tau^+ \triangleq S_{\tau+\Delta\tau} = \{s_{\tau+\Delta\tau}(o_i, o_j)\}_{\{o_i, o_j\} \subset \mathcal{O}_{S_0}}, \tag{14}$$

where $\Delta\tau > 0$ is a small temporal margin. We use $S \models s_{\text{init}}(T)$ and $S \models s_{\text{goal}}(T)$ to indicate that the global state S satisfies the symbolic initial or goal predicate of task T. For a valid task decomposition, the transfer from T_k to T_{k+1} should satisfy

$$P\big(S^+_{\tau_k} \models s_{\text{init}}(T_{k+1}) \mid S_{\tau_k} \models s_{\text{goal}}(T_k), e_k\big) > 0. \quad (15)$$

This probability does not mean that the physical scene must change discontinuously. Instead, it measures whether the terminal context of one subtask can be validly interpreted as the initialization context of the next subtask. A more detailed interpretation is provided in Appendix B.2.

Compositional reachability. The following proposition gives a sufficient condition for long-horizon feasibility. It applies to a validly decomposed task sequence whose adjacent subtasks satisfy the required boundary compatibility. We keep the notation $\mathcal{T} = \{T_1, \dots, T_K\}$ for consistency with the main text, where the braces denote the ordered subtask list generated by Module 2.

[Global Reachability via Hierarchical Composition] Let $\mathcal{T} = \{T_1, \dots, T_K\}$ be a validly decomposed long-horizon task in the scene-specific graph $\mathcal{G}_{S_0} \subset \mathcal{G}$. Assume the following conditions hold:

1. Primitive Completeness. For every simple task T_k, if the current global state satisfies its initial predicate,

$$S^+_{\tau_{k-1}} \models s_{\text{init}}(T_k), \quad (16)$$

then there exists an action flow

$$\pi_k = \pi(T_k) = \{a_i(\theta_i \mid T_k)\}_{i=1}^{n_k} \quad (17)$$

such that the subtask goal can be reached with positive probability:

$$P\Big(S_{\tau_k} \models s_{\text{goal}}(T_k) \mid S^+_{\tau_{k-1}}, \pi_k\Big) > 0. \quad (18)$$

2. Locality of Primitive Effects. For each action flow π_k, object-pair relations that are irrelevant to T_k remain unchanged unless explicitly affected by the primitive action.

3. Connectivity of Context. For every pair of consecutive subtasks T_k and T_{k+1}, their boundary predicates are compatible. That is, after completing T_k, there exists an action transfer

$$e_k = e(T_k, T_{k+1} \mid \mathcal{T}) \quad (19)$$

such that the next subtask can be initialized with positive probability:

$$P\big(S^+_{\tau_k} \models s_{\text{init}}(T_{k+1}) \mid S_{\tau_k} \models s_{\text{goal}}(T_k), e_k\big) > 0. \quad (20)$$

Then there exist an action-flow sequence

$$\Pi = \{\pi_k : \pi_k \sim p_F(\pi \mid \mathcal{T}; \epsilon_1)\}_{k=1}^{K} \quad (21)$$

and a transfer sequence

$$E = \{e_k : e_k \sim p_T(e \mid \mathcal{T}; \epsilon_2)\}_{k=1}^{K-1} \quad (22)$$

that induce at least one witness trajectory

$$\Gamma = (S_{\tau_1}, S^+_{\tau_1}, S_{\tau_2}, S^+_{\tau_2}, \dots, S_{\tau_K}) \quad (23)$$

with positive joint probability:

$$P(\Gamma \mid S_0, \Pi, E) = \prod_{k=1}^{K} P(S_{\tau_k} \mid S^+_{\tau_{k-1}}, \pi_k) \prod_{k=1}^{K-1} P(S^+_{\tau_k} \mid S_{\tau_k}, e_k) > 0, \quad (24)$$

where the base case is defined as $S^+_{\tau_0} \triangleq S_0$. Consequently, the marginal probability of reaching the final goal is lower bounded by this witness trajectory:

$$P(S_{\tau_K} \models s_{\text{goal}}(T_K) \mid S_0, \Pi, E) \geq P(\Gamma \mid S_0, \Pi, E) > 0. \quad (25)$$

This proposition is a structural guarantee rather than an empirical success guarantee. It states that if Module 2 produces a valid decomposition with compatible symbolic boundaries, Module 3 instantiates checkable initial and goal predicates, and Module 5 finds action flows and transfer links satisfying the above assumptions, then the resulting task admits a feasible witness trajectory on the scene-specific object-action graph. The self-evolution stage improves the empirical realization of these conditions by correcting failed primitive sequences or adjusting their parameters. The proof is given in Appendix B.3.

B.2. Probabilistic Interpretation of Action Transfer

This section clarifies the probability used for action transfer. The transfer edge $e_k = e(T_k, T_{k+1} \mid \mathcal{T})$ does not necessarily correspond to an additional physical action. Instead, it represents a semantic and physical compatibility check between the terminal state of T_k and the initial predicate of T_{k+1}.

Given a completed subtask T_k, the next subtask T_{k+1} is valid only if the post-subtask state can satisfy the initial predicate required by T_{k+1}. We therefore define the transfer condition as

$$P\big(S^+_{\tau_k} \models s_{\text{init}}(T_{k+1}) \mid S_{\tau_k} \models s_{\text{goal}}(T_k), e_k\big) > 0. \quad (26)$$

Here $S^+_{\tau_k}$ denotes the state after a small transition margin following the completion of T_k. This notation captures the fact that the same physical scene may be reinterpreted under a new task context.

When the object sets of T_k and T_{k+1} are disjoint, this compatibility often reduces to checking that the objects required by T_{k+1} remain in valid initial conditions. When the object sets overlap, for example when an object is picked in T_k and placed in T_{k+1}, we do not rely on background invariance. Instead, the compatibility is checked directly through the boundary predicates:

$$S_{\tau_k} \models s_{\text{goal}}(T_k), \qquad S^+_{\tau_k} \models s_{\text{init}}(T_{k+1}). \tag{27}$$

Thus, action transfer acts as a semantic gatekeeper that determines whether the output context of one subtask can serve as the valid input context of the next.

B.3. Proof of Proposition B.1

We prove the claim by constructing one feasible witness trajectory. By Primitive Completeness, for each subtask T_k whose initial predicate is satisfied by $S^+_{\tau_{k-1}}$, there exists an action flow π_k such that

$$P\Big(S_{\tau_k} \models s_{\text{goal}}(T_k) \mid S^+_{\tau_{k-1}}, \pi_k\Big) > 0. \tag{28}$$

By the Locality of Primitive Effects assumption, the execution of π_k does not invalidate unrelated object-pair relations required by subsequent subtasks unless such relations are explicitly modified by the task. Thus, the local transition of each subtask can be embedded into the global scene state with positive probability.

Next, by Connectivity of Context, for each adjacent pair (T_k, T_{k+1}), there exists a transfer edge e_k such that

$$P\big(S^+_{\tau_k} \models s_{\text{init}}(T_{k+1}) \mid S_{\tau_k} \models s_{\text{goal}}(T_k), e_k\big) > 0. \tag{29}$$

This ensures that the terminal context of T_k can serve as a valid initialization context for T_{k+1}.

Combining these local execution transitions and transfer transitions yields a witness trajectory

$$\Gamma = (S_{\tau_1}, S^+_{\tau_1}, \ldots, S_{\tau_K}). \tag{30}$$

By the chain rule along this specific trajectory,

$$\begin{aligned} &P(\Gamma \mid S_0, \Pi, E) = \\ &\prod_{k=1}^{K} P(S_{\tau_k} \mid S^+_{\tau_{k-1}}, \pi_k) \prod_{k=1}^{K-1} P(S^+_{\tau_k} \mid S_{\tau_k}, e_k). \end{aligned} \tag{31}$$

Each factor is strictly positive by Primitive Completeness and Connectivity of Context. Since the product contains finitely many strictly positive terms, we have

$$P(\Gamma \mid S_0, \Pi, E) > 0. \tag{32}$$

Because reaching the witness trajectory implies reaching the final goal predicate, the marginal probability of final-task success is at least the probability of this witness trajectory:

$$P(S_{\tau_K} \models s_{\text{goal}}(T_K) \mid S_0, \Pi, E) \geq P(\Gamma \mid S_0, \Pi, E) > 0. \tag{33}$$

Therefore, the validly decomposed long-horizon task is compositionally reachable under the stated assumptions.

C. More experimental results

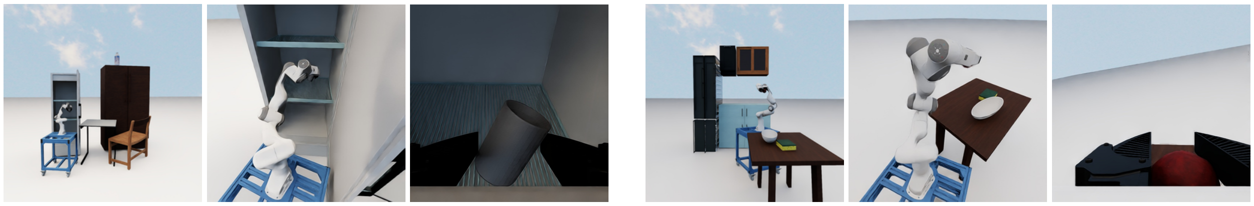
C.1. The Details of Three Camera View

In our Self-Evolution mechanism, the VLM receives visual feedback from a set of cameras $I \subset \{\text{Global, Head, Wrist}\}$. Each perspective offers distinct information density and spatial context, contributing differently to the planning and verification process.

1. **Global View (Static Scene Overview):** This view corresponds to the original perspective of the real-world input image. Since real-world photos lack intrinsic camera parameter matrices, we estimate these parameters during the simulated scene reconstruction process to align the simulation camera with the original photo's pose. This view provides the most comprehensive understanding of the task execution, capturing the relative positions of the robot, the target object, and the support surface simultaneously. It is the primary source for checking task completion logic.
2. **Head View (Robot-Following):** This camera is mounted virtually above the mobile base and moves synchronously with the robot. It serves as an approximation of a "third-person" view bound to the agent. This perspective allows for clear observation of the robotic arm's pose, joint configurations, and gripper status (open/closed). However, it lacks global environmental context. As shown in our ablation study Figure 5 (d), relying solely on this view degrades performance in tasks requiring affordance reasoning—for example, it is often impossible to determine the opening direction of a refrigerator door from this angle, leading to navigation failures.
3. **Wrist View (End-Effector Camera):** Fixed to the robotic arm's wrist, this camera looks directly at the gripper's target. Its field of view is extremely limited, capturing only the immediate interaction area. While theoretically useful for fine-grained verification (e.g., checking if a cup is level), it offers almost no utility for the coarse trajectory planning and semantic reasoning required in our graph-based generation. Consequently, as indicated in Figure 5 (d), adding this view provides negligible performance gains for our task, as the global and Head views provide sufficient information.

C.2. Robot System and Execution Details

Robot Configuration and Control. We employ a *FrankaMounted* robotic arm equipped with a mobile base for all experiments. The agent perceives the environment through on-board RGB-D sensors. Motion planning is governed by an **Inverse Kinematics (IK)** controller. For each primitive action defined in the graph, the system first calculates the target coordinate and pose for either the mobile base or the End-Effector (EEF). The IK controller then gen-



Global View | Head View | Wrist View | Global View | Head View | Wrist View

Figure 6. **Visual Comparison of Camera Perspectives.** The figure is divided into two task scenarios. **Left (Columns 1-3):** The task "Put the cup into the refrigerator". The *Global View* reveals the arm reaching inside the fridge, though the specific arm posture is occluded. The *Head View* clearly shows the robot's joint state and the open fridge door. The *Wrist View* captures the close-up of the cup within the gripper. **Right (Columns 4-6):** The task "Place the apple into the bowl". The *Global View* shows the overall layout, but depth ambiguity makes it hard to judge if the apple is perfectly over the bowl. The *Head View* compensates for this by showing the alignment relative to the base. However, since the gripper is in mid-air, the *Wrist View* captures mostly empty space, providing little actionable information.

erates a continuous motion path to reach these targets sequentially. To ensure safe interaction, we initialize the robot at a fixed offset distance relative to the target object before the start of any task execution.

Scale Adaptation. A challenge inherent to monocular 3D reconstruction is the ambiguity of absolute scale. Since the scene dimensions are estimated from a single image, the physical size of the reconstructed environment may vary from real-world norms. This discrepancy necessitates a rescaling of the robotic agent to match the simulation's affordances. Through empirical testing, we uniformly set the robot scale factor to $s = 0.7$. While this setting works for the majority of our generated scenes, we observed a few failure cases still stemming from scale mismatch.

Grasping Hyperparameters. The precision of the *CONVERGE* primitive, which moves the gripper to the final grasping position, is highly sensitive to the distance offset parameter. This offset dictates how far the gripper stops in front of the target coordinate.

If the offset is set too low, the gripper extends too far forward. For tasks like "Pick up object from table", this often causes the gripper to collide with the object or grasp it too high, leading to instability. If the offset is set too high, the gripper may stop prematurely, resulting in "grasping air" failure where the fingers close without contacting the object.

To address this, we define the convergence offset as a function of the robot's scale. In our experiments, we use a unified hyperparameter: $\delta_{\text{offset}} = 0.14 \times s$, where $s = 0.7$. This adaptive value ensures that the gripper positioning remains consistent relative to the scaled robot body.

C.3. Scalable Scene Generation Statistics

In this section, we provide detailed statistics on the dataset construction, visualize the generated environments, and evaluate the success rates of the autonomously generated atomic tasks.

To ensure the diversity and realism of our evaluation environment, we constructed a dataset of **102 unique interactive scenarios**. The generation process follows a hierarchical pipeline:

1. **Source Image Generation:** We utilized *Qwen3-Max* and *Seedream4.5*, a state-of-the-art generative model, to synthesize 34 high-quality real-world reference images. To cover the most common household manipulation settings, we prompted the model to generate diverse layouts. The distribution includes:
   - **16 Kitchens:** Focusing on complex articulated objects like refrigerators, ovens, microwaves, and cabinets.
   - **10 Bedrooms:** Focusing on wardrobes, drawers, and bedside tables.
   - **8 Others (Study Rooms, Workshops, etc.):** Focusing on tables, shelves, and miscellaneous clutter.
2. **Simulated Scene Reconstruction:** Using the reconstruction pipeline [5], we extracted semantic and geometric information from these 34 images.
3. **Variation Augmentation:** To further expand the dataset, we generated **3 unique variations** for each real-world reference by randomizing the top-3 matched assets from the BEHAVIOR-1K dataset (e.g., matching a "fridge" in the image to three different interactive fridge models in simulation). This resulted in a total of $34 \times 3 = 102$ simulation environments.

Automated Task Proposal.

Unlike manual task design, our framework leverages the **Graph-Based Task Generation** module (detailed in Appendix A.2) to autonomously populate these scenes with valid tasks.

For each of the 102 scenes, we employ a VLM to analyze the spatial layout and propose a context-aware keyword (e.g., "cup", "open fridge"). The system then expands this keyword into a full task definition tuple

$\langle$task_name, task_message$\rangle$. This automated pipeline ensures that the generated tasks are not only semantically aligned with the visual scene but also grounded in the available physical assets. Consequently, we obtained 102 distinct *Scene-Task Pairs*, serving as a reliable benchmark for evaluating the robustness of our method.

Visualization of Simulated Scene.

We visualize a subset of the generated pairs in Figure 7. The top row displays the reference real-world images generated by Qwen3-Max or Seedream-4.5, while the bottom row shows the reconstructed interactive simulated scenes in OmniGibson. Despite the domain gap, our framework successfully preserves the semantic layout (e.g., the relative position of the fridge and table) and functional affordances (e.g., the graspability of handles), providing a solid foundation for task execution.

Evaluation of Atomic Task Generation.

To validate the effectiveness of our Graph-Based Task Generation mechanism, we evaluated the execution success rate on the 102 generated Scene-Task pairs.

Since the primary goal here is to assess the feasibility of the generated tasks and the robustness of the graph-based planner, these tasks are primarily simple tasks (atomic primitives) such as "Open Refrigerator" or "Pick up Cup". For this experiment, we disabled the *Self-Evolution* mechanism and relied solely on the initial open-loop plan generated by the graph solver.

As shown in Table 1, our method achieves an overall success rate of **71.6%** across 102 tasks. By analyzing our experimental results, the following conclusions can be drawn:

- **Feasibility:** The high success rate confirms that our reconstruction method and affordance-graph planning effectively bridge the sim-to-real gap, producing physically feasible task definitions.
- **Efficiency:** This result demonstrates that SCENE2DEMO can serve as a low-cost, high-throughput engine for generating large-scale, high-quality demonstration data without human annotation.
- **Role of Self-Evolution:** The failure cases (approx. 28.4%) were mostly due to edge cases (e.g., kinematic singularities in very narrow corners). This justifies our framework's design: using the efficient Graph-Based generator for the vast majority of data production, while reserving the computationally more expensive *Self-Evolution* mechanism as a specialized agent to solve complex, long-horizon, or failure-prone edge cases.

C.4. Additional Details for Main Task Execution Comparison

This section provides additional details for the main task execution comparison in Table 2, including the number of evaluated demonstrations or episodes and the definitions of success rate (SR) and Subtask-Level Score. We use "episode" to denote a distinct evaluated task instance or scene-task configuration, and "demo" to denote one rollout under that episode.

Evaluation counts. Table 4 summarizes the exact number of evaluated demonstrations or episodes for each method and task. For $\mathcal{T}_1$–$\mathcal{T}_4$, we evaluate representative long-horizon tasks. For $\mathcal{T}_5$ and $\mathcal{T}_6$, we evaluate primitive task categories, where $\mathcal{T}_5$ corresponds to Open/Close tasks and $\mathcal{T}_6$ corresponds to Pick/Place tasks.

Table 4. Exact number of evaluated demos and episodes for each method and task. Each cell reports all demos as *demos/episode* × *episodes*.

Method	$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}_3$	$\mathcal{T}_4$	$\mathcal{T}_5$	$\mathcal{T}_6$
Ours (primitive)	10 (10×1)	10 (10×1)	10 (10×1)	10 (10×1)	36 (1×36)	66 (1×66)
Ours (prim+Evo)	20 (20×1)	10 (10×1)	10 (10×1)	10 (10×1)	–	–
GenSim2 (primitive)	20 (20×1)	20 (20×1)	20 (20×1)	20 (20×1)	80 (20×4)	80 (20×4)
RoboGen (primitive)	10 (10×1)	10 (10×1)	10 (10×1)	10 (10×1)	40 (10×4)	40 (10×4)
RoboGen (prim+RL)	1	1	1	1	1	1

For GenSim2 and RoboGen, all evaluations are executed using their native automated pipelines. For RoboGen (prim+RL), we evaluate one task-specific RL run for each task as a reference solver setting. The evaluated RoboGen+RL tasks are: $\mathcal{T}_1$ Put the bottle into the refrigerator, $\mathcal{T}_2$ Put the mouse into the refrigerator, $\mathcal{T}_3$ Put the mouse into the bowl, $\mathcal{T}_4$ Put the bottle on the table, $\mathcal{T}_5$ Open Microwave, and $\mathcal{T}_6$ Pick Bottle.

Success rate. For each task, the success rate is computed by averaging over all evaluated rollouts. Let E be the number of evaluated episodes and D_e be the number of demos evaluated in episode e. Then SR is computed as

$$\mathrm{SR} = \frac{1}{\sum_{e=1}^{E} D_e} \sum_{e=1}^{E} \sum_{i=1}^{D_e} \mathbf{1}[\mathrm{Success}_{e,i}] \times 100\%. \quad (34)$$

For our long-horizon tasks $\mathcal{T}_1$–$\mathcal{T}_4$, each task is evaluated in one fixed scene-task episode with multiple demos, e.g., 10×1 means 10 demos in one episode. For our primitive task categories $\mathcal{T}_5$ and $\mathcal{T}_6$, we evaluate many scene-task episodes with one demo each, e.g., 1×36 for Open/Close and 1×66 for Pick/Place. For complex long-horizon tasks, a demo is counted as successful only if all subtasks are successfully completed.

Subtask-Level Score. Besides complete-task SR, we report a Subtask-Level Score to measure partial execution quality. Since SCENE2DEMO, GenSim2, and RoboGen define and evaluate subtasks differently, this second metric should be interpreted as a diagnostic subtask-level measure rather than a strictly identical mathematical quantity across all systems.

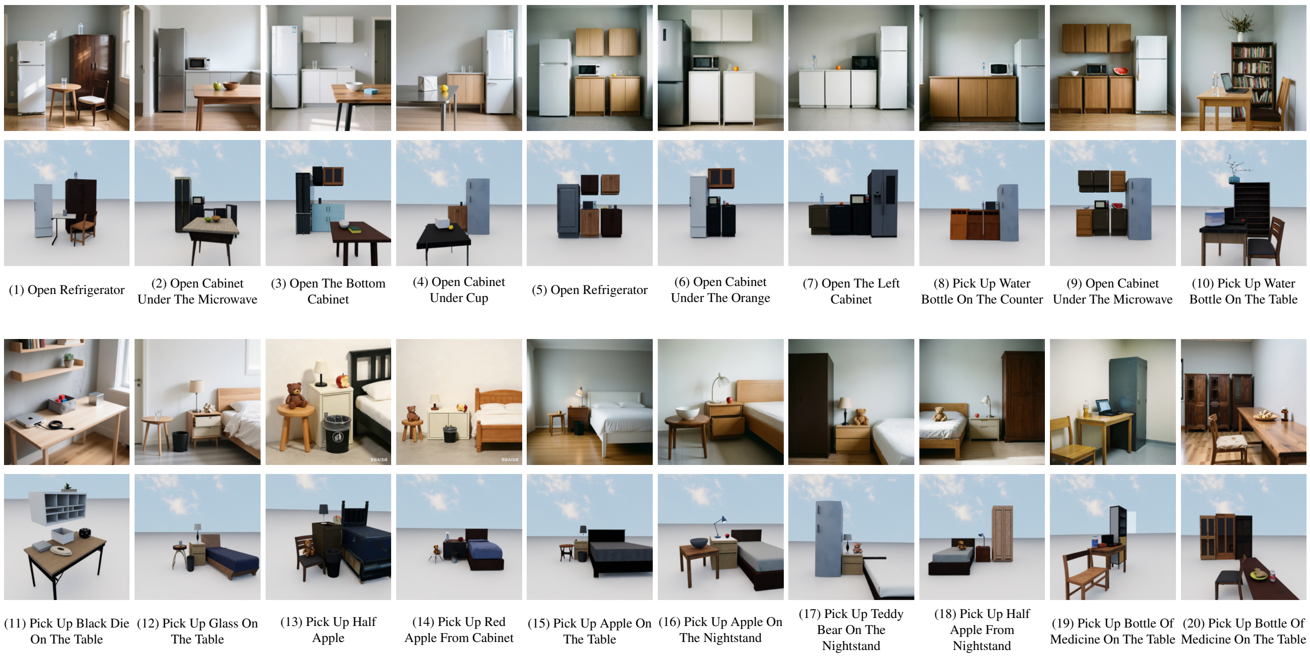


Figure 7. **Visualization of 20 Representative Scene-Task Pairs. Top Row:** Real-world reference images generated by Qwen3-Max or Seedream-4.5. **Bottom Row:** Reconstructed interactive simulation environments. Each column corresponds to a specific atomic task proposed by the VLM, demonstrating our framework's ability to generate diverse household scenarios.

For SCENE2DEMO, each long-horizon task is decomposed into a fixed sequence of symbolic subtasks. We compute the Subtask-Level Score as the average conditional success rate over these subtasks. Let K be the number of subtasks. For subtask T_k, we compute its success rate among the demos that reach T_k:

$$\mathrm{CondSR}(T_k) = \frac{\sum_i \mathbf{1}[\mathrm{Reach}_{i,k}]\mathbf{1}[\mathrm{Success}_{i,k}]}{\sum_i \mathbf{1}[\mathrm{Reach}_{i,k}]}. \tag{35}$$

The Subtask-Level Score of SCENE2DEMO is then

$$\mathrm{SubtaskScore}^{\textsc{Scene2Demo}} = \frac{1}{K}\sum_{k=1}^{K} \mathrm{CondSR}(T_k) \times 100\%. \tag{36}$$

This metric measures how reliably each local subtask can be executed once its precondition is reached. For our primitive task categories $\mathcal{T}_5$ and $\mathcal{T}_6$, each evaluated task contains one subtask, so the Subtask-Level Score is identical to SR.

For GenSim2, we use its native progress evaluator. GenSim2 defines task-specific success criteria and maintains a progress vector over these criteria; the reported score is the average completion value returned by its native evaluator. For RoboGen, because the generated task structure is not fixed and may contain both primitive and RL-type subtasks, we compute the ratio of completed generated subtasks to all generated subtasks for each episode and then average across episodes. In the primitive-only RoboGen setting, RL subtasks are disabled, so early primitive subtasks may be completed while later subtasks fail, leading to non-zero subtask-level scores even when complete-task SR is 0%.

It is worth noting that, although $\mathcal{T}_5$ and $\mathcal{T}_6$ are treated as one-subtask primitive categories in our evaluation, RoboGen may still decompose the corresponding tasks into multiple subtasks. Therefore, RoboGen's Subtask-Level Score and SR can differ even on $\mathcal{T}_5$ and $\mathcal{T}_6$.

C.5. Performance of Example Complex Long-Horizon Tasks

In this section, we provide a granular breakdown of the generation and instantiation process for the four representative complex long-horizon tasks ($\mathcal{T}_1$ to $\mathcal{T}_4$). We detail the intermediate outputs of our pipeline, from semantic expansion to action flow planning, to demonstrate the reliability and logical consistency of SCENE2DEMO.

C.5.1. Visual Overview of Examples Scenarios

To intuitively present the experimental setup, Figure 8 visualizes the four tasks using a structured grid. Each complex long-horizon task will display real-world picture, simulated reconstruction scene, main target object, main support object, and the user keyword.

C.5.2. Semantic Expansion Results

Upon receiving the user keyword, scene configuration, and observation, the VLM performs Semantic Expansion. While the exact phrasing may vary slightly across different demo runs due to the probabilistic nature of VLMs, the

glass_0 refrigerator_0 | apple_0 refrigerator_0 | apple_0 bowl_0 | glass_0 table_0

"Transport a glass into a refrigerator" | "Place an apple into a refrigerator" | "Transfer an apple into a bowl" | "Place a cup onto a table"

Figure 8. **Visual Overview of Complex Long-Horizon Tasks. Row 1:** Pairs of real-world reference images and reconstructed simulation environments. **Row 2:** Close-up views of the main target object and support object matched from the simulation environments. **Row 3:** The user-provided keywords that seed the autonomous task generation process.

semantic intent remains strictly consistent with the user's instructions. The following are representative outputs for a single demo of each complex long-horizon task:

$\mathcal{T}_1$: Put the glass into the refrigerator

```
"task_activity_name": "
    open_the_refrigerator_and_put_the_glass_ir
    ",
"task_detail_message": "This is a
    kitchen scene, and there is a
    glass_0 on the table_0, and there
    is a refrigerator_0 on the left
    side. The task we need to
    accomplish is to put the glass_0
    on the table_0 into the
    refrigerator_0. The robot needs to
     perform a sequence of actions.
    First, the robotic arm needs to
    approach the handle of the
    refrigerator_0 and rotate to open
    it. Second, pick up the glass_0
    from the table_0 and place it
    inside the refrigerator_0. Finally
    , close the refrigerator_0."
```

$\mathcal{T}_2$: Put the apple into the refrigerator

```
"task_activity_name": "
    open_the_refrigerator_and_put_the_apple_ir
    ",
"task_detail_message": "This is a
    modern kitchen scene. The initial
    condition is that there is a
    half_apple (apple_0) resting on
    top of a breakfast_table (table_0)
    , and a refrigerator (
    refrigerator_0) is positioned to
    the right side of the scene,
    mounted against the wall. The task
     we need to accomplish is to put
    the apple_0 into the
    refrigerator_0. The robot needs to
     perform a sequence of actions:
    First, approach the refrigerator_0
     and open its door by grasping and
     rotating the handle. Second, pick
     up the apple_0 from the table_0.
    Third, carefully place the apple_0
     inside the refrigerator_0.
    Finally, close the refrigerator_0
    door to complete the task."
```

$\mathcal{T}_3$: Put the apple into the bowl

```
"task_activity_name": "
    pick_up_apple_and_put_into_bowl",
"task_detail_message": "This is a
    modern kitchen scene. The initial
    condition is that there is an
    apple_0 resting on the countertop
    of cabinet_0, and a bowl_0 placed
    on the wooden table_0 in the
    foreground. The task we need to
    accomplish is to pick up the
    apple_0 from the countertop and
```

```
    place it into the bowl_0. The
    robotic arm must first approach
    and grasp the apple_0 while it is
    stationary on the cabinet_0
    surface. Then, without releasing
    the apple_0, the arm must move it
    over to the bowl_0 on the table_0
    and gently release it inside the
    bowl_0. No container needs to be
    opened or closed as both objects
    are accessible and unobstructed."
```

$\mathcal{T}_4$**: Put the cup on the table**

```
"task_activity_name": "
    put_the_glass_on_the_table",
"task_detail_message": "This is a
    modern kitchen scene. The initial
    condition is that there is a
    glass_0 (water_glass) currently
    located near the cabinet_0, not
    yet on the table_0. The task we
    need to accomplish is to place the
     glass_0 onto the table_0. The
    robotic arm needs to perform the
    following steps: First, approach
    and grasp the glass_0 from its
    current position. Second, lift and
     move the glass_0 toward the
    table_0. Third, carefully place
    the glass_0 onto the surface of
    the table_0, ensuring it is stable
     and properly positioned. No other
     objects need to be moved or
    manipulated during this task."
```

C.5.3. Subtask Decomposition

A defining characteristic of SCENE2DEMO is its structured, graph-based decomposition. By modeling the task space as transitions between simple sub-tasks, we ensure that the decomposition is deterministic across hundreds of demos. Regardless of visual variations in the scene, the logical steps required to satisfy the complex goal remain invariant.

Due to space limitations, this document only shows the names and descriptions of each sub-task within the complex long-horizon tasks. Information such as target objects, support objects, and BDDL names, which are included in the output configuration file, are not presented here. The following are representative outputs for a single demo of each complex long-horizon task:

$\mathcal{T}_1$**: Put the glass into the refrigerator**

```
"sub_task_1": {
   "task_activity_name": "
      open_refrigerator",
   "task_detail_message": "First, the
      robotic arm needs to approach
      the handle of the
      refrigerator_0 and rotate to
      open it.",
   ...
},
"sub_task_2": {
   "task_activity_name": "
      pick_up_glass",
   "task_detail_message": "Second,
      pick up the glass_0 from the
      table_0. The robotic arm needs
      to navigate to the glass_0,
      approach it, grasp it, and lift
       it from the table_0.",
   ...
},
"sub_task_3": {
   "task_activity_name": "
      put_glass_into_refrigerator",
   "task_detail_message": "Third,
      place the glass_0 inside the
      refrigerator_0. The robotic arm
       needs to navigate to the
      refrigerator_0, move forward
      while holding the glass_0, and
      ungrasp to place it inside the
      refrigerator_0.",
   ...
},
"sub_task_4": {
   "task_activity_name": "
      close_refrigerator",
   "task_detail_message": "Finally,
      close the refrigerator_0. The
      robotic arm needs to approach
      the handle of the
      refrigerator_0 and rotate to
      close it.",
   ...
}
```

$\mathcal{T}_2$**: Put the apple into the refrigerator**

```
"sub_task_1": {
   "task_activity_name": "
      open_refrigerator",
   "task_detail_message": "First,
      approach the refrigerator_0 and
       open its door by grasping and
```

```
      rotating the handle.",
    ...
  },
  "sub_task_2": {
    "task_activity_name": "
      pick_up_apple",
    "task_detail_message": "Second,
      pick up the apple_0 from the
      table_0. The robotic arm needs
      to navigate to the target
      apple_0, approach it, grasp it,
       and lift it off the table_0.",
    ...
  },
  "sub_task_3": {
    "task_activity_name": "
      put_apple_into_refrigerator",
    "task_detail_message": "Third,
      carefully place the apple_0
      inside the refrigerator_0. The
      robotic arm needs to navigate
      to the refrigerator_0, move
      forward, and release the
      apple_0 inside the
      refrigerator_0.",
    ...
  },
  "sub_task_4": {
    "task_activity_name": "
      close_refrigerator",
    "task_detail_message": "Finally,
      close the refrigerator_0 door
      to complete the task. The robot
       needs to navigate to the
      refrigerator_0, get close to
      the handle, and rotate to close
       the door.",
    ...
  }
```

$\mathcal{T}_3$: Put the apple into the bowl

```
  "sub_task_1": {
    "task_activity_name": "
      pick_up_apple",
    "task_detail_message": "The robotic
       arm must first approach and
      grasp the apple_0 while it is
      stationary on the countertop of
       cabinet_0. The target object
      is apple_0, and the support
      object is cabinet_0 as the
      success condition is that the
      apple_0 is no longer resting on
       the cabinet_0 surface.",
    ...
  },
  "sub_task_2": {
    "task_activity_name": "
      put_apple_into_bowl",
    "task_detail_message": "Then,
      without releasing the apple_0,
      the arm must move it over to
      the bowl_0 on the table_0 and
      gently release it inside the
      bowl_0. The target object
      remains apple_0, but now the
      support object is bowl_0, as
      the success condition is that
      the apple_0 is inside the
      bowl_0.",
    ...
  }
```

$\mathcal{T}_4$: Put the cup on the table

```
  "sub_task_1": {
    "task_activity_name": "
      pick_up_glass",
    "task_detail_message": "First, the
      robotic arm must locate and
      pick up the glass_0 from its
      current position (near the
      cabinet or floor level). The
      robot needs to approach the
      glass_0, grasp it securely, and
       lift it off its current
      surface.",
    ...
  },
  "sub_task_2": {
    "task_activity_name": "
      put_glass_on_table",
    "task_detail_message": "Second, the
       robotic arm must move the
      glass_0 toward the table_0 and
      gently place it on the tabletop
      , ensuring stability. The robot
       should navigate to the table_0
      , lower the glass_0 onto its
      surface, and release it
      carefully.",
    ...
  }
```

C.5.4. Object Size Adjustment

To bridge the gap between reconstructed visual assets and physical robot constraints, we implement a dynamic scaling module. For each subtask, the system identifies the target object and applies a scaling factor calculated from its bounding box extents relative to the robot's parallel gripper width ($\approx 0.06m$).

Detailed Scaling for $\mathcal{T}_1$ and $\mathcal{T}_2$. In complex long-horizon task 1, the target object sequence for each sub-task involves [refrigerator_0: 1.0, glass_0: 0.37, glass_0: 0.37, refrigerator_0: 1.0].

For fixed fixtures like the refrigerator, the scale is set to 1.0 to maintain the room's layout. For the manipulable glass_0 object, a scaling factor of 0.37 was determined in sub-tasks 2 and 3 to ensure the glass diameter was easy to grasp. Finally, we selected the first occurring scaling factors for the refrigerator and the glass, which are 1.0 and 0.37 respectively, and applied these scaling factors.

In complex long-horizon task 2, the target object sequence for each sub-task involves [refrigerator_0: 1.0, apple_0: 0.35, apple_0: 0.4, refrigerator_0: 1.0]. And we select 1.0 and 0.35 for the refrigerator and the apple respectively

Detailed Scaling for $\mathcal{T}_3$ and $\mathcal{T}_4$. In complex long-horizon task 3 and 4, the target object sequence for each sub-task is [apple_0: 0.42, apple_0: 0.42] and [glass_0: 0.4, glass_0: 0.4]. So we use the first occurring scaling factors 0.42 and 0.4 for $\mathcal{T}_3$ and $\mathcal{T}_4$ respectively.

C.5.5. BDDL Generation (Initial State and Success Detection)

The BDDL generation stage defines the logical boundaries of each subtask. The Table 5 shows the representative initial and target (goal) states of the BDDL definitions for these four complex long-horizon tasks, and we also show the representative outputs for a single demo.

Table 5. Initial and goal states for the sub-tasks of our complex long-horizon tasks

Tasks	Subtask Index	Initial State	Target State
$\mathcal{T}_1$: Glass→Fridge	1	(not (open refrigerator_0)	(open refrigerator_0)
	2	(ontop glass_0 table_0)	(not (ontop glass_0 table_0))
	3	(inside glass_0 gripper)	(inside glass_0 refrigerator_0)
	4	(open refrigerator_0)	(not (open refrigerator_0))
$\mathcal{T}_2$: Apple→Fridge	1	(not (open refrigerator_0)	(open refrigerator_0)
	2	(ontop apple_0 table_0)	(not (ontop apple_0 table_0))
	3	(inside apple_0 gripper)	(inside apple_0 refrigerator_0)
	4	(open refrigerator_0)	(not (open refrigerator_0))
$\mathcal{T}_3$: Apple→Bowl	1	(ontop apple_0 cabinet_0)	(not (ontop apple_0 cabinet_0))
	2	(inside apple_0 gripper)	(inside apple_0 bowl_0)
$\mathcal{T}_4$: Cup→Table	1	(ontop glass_0 cabinet_0)	(not (ontop glass_0 cabinet_0))
	2	(inside glass_0 gripper)	(ontop glass_0 table_0)

C.5.6. Initial Action Flow

The mapping of high-level subtask definitions to executable primitive sequences is a critical step in the SCENE2DEMO pipeline. While visual keyframes of these executions are omitted here for brevity, comprehensive video demonstrations of the initial and evolved policies for all tasks are available on our project page (as linked in the Abstract).

Detailed definitions and physical implementations of each atomic primitive (e.g., *APPROACH*, *CONVERGE*, *ARTICULATE*) are provided in Appendix A.2.3. Table 6 summarizes the deterministic initial action sequences generated by our graph-based planner for the four representative complex long-horizon tasks before any self-evolution iterations.

C.6. Performance of Self-Evolution

In this section, we present the step-by-step evolution trajectory for the four representative complex long-horizon tasks. The Self-Evolution mechanism plays a crucial role in correcting execution failures caused by geometric uncertainties or kinematic constraints that are difficult to resolve with open-loop planning.

We report the evolutionary process for one representative demonstration of each complex long-horizon task. Tables 7, 8, 9, and 10 detail the changes in the action flow across iterations. The "Iter 0" row represents the initial open-loop plan generated by the graph solver. Subsequent iterations show how the VLM refined the sequence by adjusting parameters or inserting new primitives.

Evolution of Task $\mathcal{T}_1$ (Put the glass into the refrigerator). The evolution process of Task $\mathcal{T}_1$ is shown in Table 7. Subtasks 1 and 2 were successfully completed on the first attempt. However, subtask 3 (place the glass) required two iterations of improvement. Initially, the robot collided with the open refrigerator door, causing the door to close and preventing the subsequent tasks from being executed. Therefore, the VLM adjusted the base movement path. After one iteration, the glass was released too high and too far back, so the VLM adjusted the robot arm's extension range to ensure safe placement in the correct position. Subtask 4 (close the refrigerator) required one iteration of improvement to ensure the refrigerator door was completely closed.

Evolution of Task $\mathcal{T}_2$ (Put the apple into the refrigerator). Table 8 show the evolution results. Subtask 3 failed initially because the robot could not reach the deep shelf. The evolution process involved significant adjustments to the base position and rotation (`TURN_BASE`) over 3 iterations to achieve a valid kinematic configuration for placement.

Evolution of Task $\mathcal{T}_3$ (Put the apple into the bowl). This evolution details is in Table 9, which required precise positioning to ensure the apple landed *inside* the bowl rather than hitting the rim. Subtask 2 evolved twice: first to adjust the approach angle via `TURN_BASE`, and second to fine-tune the release height.

Table 6. Initial action flows for the sub-tasks of our representative complex long-horizon tasks

Tasks	Subtask	Initial Action Flow (Pre-evolution)
$\mathcal{T}_1$	1	APPROACH → CONVERGE → GRASP → ARTICULATE_OPEN → UNGRASP
	2	NAVIGATE_TO_TARGET → APPROACH → CONVERGE → GRASP → LIFT_EEF_UP(0.2)
	3	NAVIGATE_TO_SUPPORT → MOVE_BASE_FORWARD(0.4) → MOVE_EEF_FORWARD(0.1) → UNGRASP
	4	NAVIGATE_TO_TARGET → APPROACH → CONVERGE → GRASP → ARTICULATE_CLOSE(0.0, 0.5) → UNGRASP
$\mathcal{T}_2$	1	APPROACH → CONVERGE → GRASP → ARTICULATE_OPEN(0.0, 0.7) → UNGRASP → RETREAT
	2	NAVIGATE_TO_TARGET → RETREAT → APPROACH → CONVERGE → GRASP → LIFT_EEF_UP(0.2)
	3	NAVIGATE_TO_SUPPORT → MOVE_BASE_FORWARD(0.4) → MOVE_EEF_FORWARD(0.1) → UNGRASP
	4	NAVIGATE_TO_TARGET → APPROACH → CONVERGE → GRASP → ARTICULATE_CLOSE(0.0, 0.7) → UNGRASP
$\mathcal{T}_3$	1	APPROACH → CONVERGE → GRASP → LIFT_EEF_UP(0.2)
	2	NAVIGATE_TO_SUPPORT → MOVE_BASE_FORWARD(0.4) → MOVE_EEF_FORWARD(0.1) → LIFT_EEF_DOWN(0.3) → UNGRASP
$\mathcal{T}_4$	1	APPROACH → CONVERGE → GRASP → LIFT_EEF_UP(0.2)
	2	NAVIGATE_TO_SUPPORT → MOVE_BASE_FORWARD(0.4) → MOVE_EEF_FORWARD(0.1) → LIFT_EEF_DOWN(0.3) → UNGRASP

Table 7. **Evolutionary Trajectory for Task $\mathcal{T}_1$: "Put the glass into the refrigerator".** Subtasks 1 and 2 succeeded initially. Subtask 3 required 2 iterations to correct placement height and depth. Subtask 4 required 1 iteration to fully close the door.

Subtask	Iter	Action Flow
1	0	APPROACH → CONVERGE → GRASP → ARTICULATE_OPEN → UNGRASP
2	0	NAVIGATE_TO_TARGET → APPROACH → CONVERGE → GRASP → LIFT_EEF_UP(0.2)
3	0	NAVIGATE_TO_SUPPORT → MOVE_BASE_FORWARD(0.4) → MOVE_EEF_FORWARD(0.1) → UNGRASP
	1	NAVIGATE_TO_SUPPORT → MOVE_BASE_FORWARD(0.3) → MOVE_EEF_FORWARD(0.45) → UNGRASP
	2	NAVIGATE_TO_SUPPORT → MOVE_BASE_FORWARD(0.3) → MOVE_EEF_FORWARD(0.45) → MOVE_EEF_FORWARD(0.2) → LIFT_EEF_DOWN(0.3) → UNGRASP
4	0	NAVIGATE_TO_TARGET → APPROACH → CONVERGE → GRASP → ARTICULATE_CLOSE(0.0, 0.5) → UNGRASP
	1	NAVIGATE_TO_TARGET → APPROACH → CONVERGE → GRASP → ARTICULATE_CLOSE(0.0, 0.6) → MOVE_EEF_FORWARD(0.1) → UNGRASP

Evolution of Task $\mathcal{T}_4$ (Put the cup on the table) Similar to $\mathcal{T}_3$, Table 10 shows that placing a cup on a specific spot on the table required adjusting the robot base's orientation (`TURN_BASE`) to reach the target zone.

VLM Reasoning Process. To illustrate the decision-making capability of the Self-Evolution mechanism, we provide the raw reasoning outputs for Task $\mathcal{T}_1$ in the box below. This includes the step-wise critiques m_τ and the final Global Supervisor reasoning.

First, subtask 3 underwent two iterations, and the output of all iterations are:

```
"iter_1": {
   "VLM-Observation": {
      "Step 0 observation (4 frames)":
           "The robot base navigates
          to the support (refrigerator
          ), positioning itself near
          the open refrigerator door.
          The refrigerator door
          remains open throughout, and
           no collision or
          environmental change is
          observed during this step.",
      "Step 1 observation (4 frames)":
           "The robot base moves
          forward 0.45 meters toward
          the refrigerator. During
          this motion, the robotic arm
           collides with the open
          refrigerator door, causing
          it to swing shut partially,
          which alters the environment
           and obstructs access to the
           refrigerator compartment.",
      "Step 2 observation (2 frames)":
           "The robot performs the
          UNGRASP action while its arm
           is positioned near the now
          partially closed
          refrigerator door. No
          further collision occurs,
          but the glass is not placed
```

Table 8. **Evolutionary Trajectory for Task $\mathcal{T}_2$: "Put the apple into the refrigerator".** Subtask 3 underwent 3 iterations, introducing base rotation and finer arm control to navigate the apple onto the fridge shelf.

Subtask	Iter	Action Flow
1	0	APPROACH → CONVERGE → GRASP → ARTICULATE_OPEN(0.0, 0.7) → UNGRASP → RETREAT
2	0	NAVIGATE_TO_TARGET → RETREAT → APPROACH → CONVERGE → GRASP → LIFT_EEF_UP(0.2)
3	0	NAVIGATE_TO_SUPPORT → MOVE_BASE_FORWARD(0.4) → MOVE_EEF_FORWARD(0.1) → UNGRASP
	1	NAVIGATE_TO_SUPPORT → MOVE_BASE_BACKWARD(0.3) → MOVE_EEF_FORWARD(0.4) → MOVE_EEF_FORWARD(0.2) → UNGRASP
	2	NAVIGATE_TO_SUPPORT → MOVE_BASE_BACKWARD(0.3) → MOVE_EEF_FORWARD(0.4) → TURN_BASE_LEFT(45) → MOVE_EEF_FORWARD(0.2) → UNGRASP
	3	NAVIGATE_TO_SUPPORT → MOVE_BASE_BACKWARD(0.3) → MOVE_EEF_FORWARD(0.4) → TURN_BASE_LEFT(45) → MOVE_EEF_FORWARD(0.3) → LIFT_EEF_DOWN(0.2) → UNGRASP
4	0	NAVIGATE_TO_TARGET → APPROACH → CONVERGE → GRASP → ARTICULATE_CLOSE(0.0, 0.7) → UNGRASP

Table 9. **Evolutionary Trajectory for Task $\mathcal{T}_3$: "Put the apple into the bowl".** The placement subtask (Subtask 2) required 2 iterations to prevent collision with the bowl's rim.

Subtask	Iter	Action Flow
1	0	APPROACH → CONVERGE → GRASP → LIFT_EEF_UP(0.2)
2	0	NAVIGATE_TO_SUPPORT → MOVE_BASE_FORWARD(0.4) → MOVE_EEF_FORWARD(0.1) → LIFT_EEF_DOWN(0.3) → UNGRASP
	1	NAVIGATE_TO_SUPPORT → MOVE_BASE_BACKWARD(0.15) → MOVE_EEF_FORWARD(0.4) → TURN_BASE_LEFT(15) → MOVE_EEF_FORWARD(0.1) → LIFT_EEF_DOWN(0.4) → UNGRASP
	2	NAVIGATE_TO_SUPPORT → MOVE_BASE_BACKWARD(0.3) → MOVE_EEF_FORWARD(0.2) → TURN_BASE_RIGHT(20) → MOVE_EEF_FORWARD(0.25) → LIFT_EEF_DOWN(0.4) → UNGRASP

```
            inside the refrigerator due
            to the obstructed access
            caused by the doors position
            . The environment remains
            unchanged during this step."
      },
      "reason": "During the execution of
         'put_glass_into_refrigerator',
         the robotic arm collided with
         the open refrigerator door
         while performing '
         MOVE_BASE_FORWARD', causing the
          door to close partially and
         blocking access to the
         compartment. Therefore, before
         'MOVE_BASE_FORWARD', the
         following step should be added:
          'MOVE_BASE_LEFT 0.3m' to clear
          the door swing path.",
      "new_sequence": [18, {"15": 0.3},
         {"13": 0.45}, 5]
   },
   "iter_2": {
      "VLM-Observation": {
         "Step 0 observation (4 frames)":
             "The robot base navigates
            toward the refrigerator,
```

```
            positioning itself so the
            arm can reach the open door.
             No collision occurs, and
            the refrigerator door
            remains open; the
            environment status is
            unchanged.",
         "Step 1 observation (3 frames)":
             "The robot base moves left
            by 0.3 meters, repositioning
             the arm slightly away from
            the refrigerator door. No
            collision occurs, and the
            refrigerator door remains
            open; no environmental
            changes are observed.",
         "Step 2 observation (4 frames)":
             "The robot base moves
            forward by 0.45 meters,
            bringing the arm closer to
            the refrigerator. No
            collision occurs, and the
            refrigerator door remains
            open; no environmental
            changes are observed.",
         "Step 3 observation (2 frames)":
             "The robot opens its
```

Table 10. **Evolutionary Trajectory for Task $\mathcal{T}_4$: "Put the cup on the table".** Subtask 2 required 2 iterations, heavily relying on base rotation to align with the table surface.

Subtask	Iter	Action Flow
1	0	APPROACH → CONVERGE → GRASP → LIFT_EEF_UP(0.2)
2	0	NAVIGATE_TO_SUPPORT → MOVE_BASE_FORWARD(0.4) → MOVE_EEF_FORWARD(0.1) → LIFT_EEF_DOWN(0.3) → UNGRASP
	1	NAVIGATE_TO_SUPPORT → TURN_BASE_RIGHT(60) → MOVE_EEF_FORWARD(0.4) → MOVE_EEF_FORWARD(0.1) → LIFT_EEF_DOWN(0.3) → UNGRASP
	2	NAVIGATE_TO_SUPPORT → TURN_BASE_RIGHT(60) → MOVE_EEF_FORWARD(0.4) → MOVE_EEF_FORWARD(0.2) → LIFT_EEF_DOWN(0.3) → UNGRASP


```
             gripper to release the glass
             , but the glass is not
             visible inside the
             refrigerator, indicating it
             was not placed successfully.
              No collision occurs, and
             the refrigerator door
             remains open; no
             environmental changes are
             observed."
      },
      "reason": "The task failed because
          after moving the base forward,
          the robotic arm did not extend
          far enough into the
          refrigerator to place the glass
           inside before releasing it.
          The UNGRASP action occurred
          while the gripper was still
          outside the refrigerator
          compartment. Therefore, after '
          MOVE_BASE_FORWARD', the
          following steps should be added
          : 1. 'MOVE_EEF_FORWARD 0.2m' to
           extend the arm into the
          refrigerator, and 2. '
          LIFT_EEF_DOWN 0.3m' to lower
          the glass to a safe placement
          height inside the refrigerator
          .",
      "new_sequence": [18, {"15": 0.3},
          {"13": 0.45}, {"9": 0.2}, {"8":
           0.3}, 5]
   }
```


Next, here are the output results for subtask 4:


```
   "iter_1": {
      "VLM-Observation": {
         "Step 0 observation (5 frames)":
              "The robot successfully
             navigates to the
             refrigerator, positioning
             itself beside the open door
             without hitting any
             obstacles. The environment
             remains unchanged during
             this step; the refrigerator
             door stays open, and no
             unintended movements or
             collisions occur.",
         "Step 1 observation (1 frames)":
              "The robot arm extends
             toward the refrigerator
             handle in a controlled
             manner, maintaining
             clearance from surrounding
             objects. No collision occurs
             , and the refrigerator door
             remains open without any
             unintended movement or
             environmental change.",
         "Step 2 observation (1 frames)":
              "The robot arm finely
             adjusts its position to
             align precisely with the
             refrigerator handle, showing
              no contact with surrounding
              objects. The refrigerator
             door remains open and
             stationary, with no
             environmental changes or
             collisions observed.",
         "Step 3 observation (1 frames)":
              "The robot gripper
             successfully closes around
             the refrigerator handle
             without colliding with any
             surrounding objects. The
             refrigerator door remains
             open and stationary, with no
              unintended environmental
             changes observed during the
             grasp.",
```

```
        "Step 4 observation (6 frames)":
             "The robot arm attempts to
            close the refrigerator door
            by pulling the handle, but
            the door does not fully shut
             and appears to be
            obstructed or misaligned. No
             collision with surrounding
            objects is observed, but the
             environment status changes
            partially  the door moves
            slightly but fails to reach
            a fully closed position,
            indicating an incomplete
            articulation.",
        "Step 5 observation (2 frames)":
             "The robot opens its
            gripper to release the
            refrigerator handle, with no
             contact or collision
            observed. The refrigerator
            door remains partially open
            and does not move further
            during ungrasping,
            indicating that the failure
            to fully close occurred
            prior to this step."
    },
    "reason": "The failure occurred
        during the 'ARTICULATE_CLOSE'
        step. Although the robot
        successfully grasped the handle
         and attempted to close the
        door, the door did not fully
        shut, likely due to
        insufficient pulling force or
        misalignment in the
        articulation range. To correct
        this, we should extend the
        articulation range slightly
        beyond 0.5 and add a small
        forward motion of the EEF
        before ungrasping to ensure
        full closure.",
    "new_sequence": [17, 1, 2, 3,
        {"19": [0.0, 0.6]}, {"9": 0.1},
         5]
}
```


C.7. Failure Details and Analysis

C.7.1. Performance Variance across VLM Architectures

We conducted a comparative study on the self-evolution performance of three VLMs: **Gemini-3-Flash**, **GPT-5.2**, and **Qwen3-VL** (specifically, the open-source model *qwen3-vl-235b-22a-thinking*). While all models demonstrated reasoning capabilities, their evolution strategies exhibited distinct characteristics:

- **Qwen3-VL (Conservative Strategy):** This model tends to be conservative in its modifications. Its proposed changes often involve minimal adjustments, such as tweaking a single parameter or adding just one or two atomic actions. While this ensures stability, Qwen3-VL frequently fails to identify "stuck" situations, where the robot is physically blocked but not in collision, resulting in minor updates that do not resolve the root cause.
- **Gemini-3-Flash (Aggressive Strategy):** Gemini-3-Flash exhibits a more aggressive and perceptive reasoning style. It is often capable of identifying subtle geometric constraints, such as the robot arm grazing the edge of a fridge door. Consequently, it proposes significant modifications, such as changing a movement parameter from around 0.3m to around 0.8m. However, given the offline nature of our evolution mechanism (where the VLM cannot interactively query state during reasoning), these large-scale parameter jumps sometimes degrade accuracy, causing the robot to deviate from the correct target position.
- **GPT-5.2 (Redundant Strategy):** GPT-5.2 shows a tendency to over-complicate solutions. It frequently stacks new actions onto the existing sequence rather than refining the current ones. This often leads to a "death spiral" where the action sequence grows to over 10 steps. As the sequence length increases, the probability of cumulative execution error rises, making success increasingly difficult despite valid high-level logic.

C.7.2. Case Studies on Persistent Failures

Qualitative inspection of failure cases reveals that errors are rarely due to high-level logic. Instead, they stem from low-level kinematic constraints, such as the robotic arm reaching a singularity when confined against the fridge side, or the hard-coded action primitives lacking the reflexivity to compensate for minor grasping errors.

Case 1: Kinematic Obstruction in Task $\mathcal{T}_1$ (Glass $\rightarrow$ Fridge). Figure 9 shows the first 4 iterations of subtask 3 (placing the glass into the refrigerator), which ultimately failed after 5 iterations. The visual record shows the robot attempting to extend its arm deep into the refrigerator.

As observed in the figures, the root cause is not the arm extension parameter, but the base positioning. The robotic arm link is physically blocked by the left side of the refrigerator frame, preventing the arm from extending fully. This type of collision is subtle and difficult for the VLM to perceive from the standard camera views.

Case 2: Precision Placing in Task $\mathcal{T}_3$ (Apple $\rightarrow$ Bowl). Figure 10 depicts a failure in Subtask 2 (Place apple into bowl). The challenge here is fine-grained spatial alignment.

As shown in the iteration history, the VLM struggles to calculate the exact spatial offset required to center the grip-

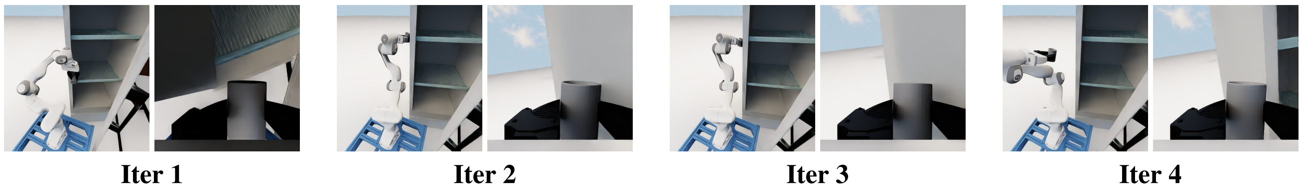


Figure 9. **Failure Analysis for $\mathcal{T}_1$ Subtask 3.** Despite 4 rounds of evolution (Iter 5 also failed), the robot consistently fails to insert the glass. The root cause, visible in the Head View, is that the lower link of the robotic arm collides with the left wall of the refrigerator, preventing the end-effector from advancing. This kinematic constraint is difficult for the VLM to diagnose from 2D images, as it requires a subtle backward adjustment of the mobile base to clear the singularity, which is a correction that iterative parameter tuning failed to find.

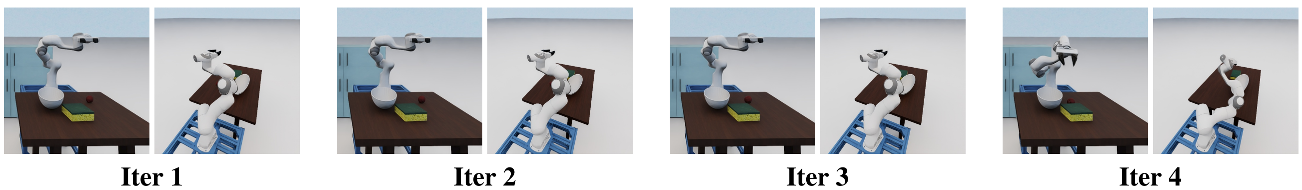


Figure 10. **Failure Analysis for $\mathcal{T}_3$ Subtask 2.** The robot struggles to align the apple directly above the bowl. Over 4 iterations, the VLM attempts to correct the drift, but lacks the precision to calculate the exact offset. The open-loop nature means the VLM oscillates between over-correcting (moving too far left) and under-correcting, failing to find the "sweet spot" within the limited iteration budget.

per over the bowl. Since our method generates the next plan offline based on static visual feedback, the VLM cannot rely on real-time continuous servoing. This results in an oscillation where the robot either adjusts too little (incremental trial-and-error) or modifies parameters too aggressively, missing the target window.

C.8. Downstream Policy Learning

We further evaluate whether the demonstrations generated by SCENE2DEMO can be directly used for downstream policy learning. We train behavior cloning (BC) policies on expert demonstrations autonomously generated by our pipeline. This experiment is not intended to introduce a new policy architecture; instead, it serves as a validation that the generated observation-action data is executable, diverse, and suitable for standard imitation learning.

Policy architecture. Figure 11 illustrates the policy architecture. The policy takes point clouds and proprioceptive states as input. The point cloud input contains 3D coordinates (x, y, z) and an end-effector mask. Before encoding, we apply point-cloud randomization, including downsampling to 1024 points, translation perturbation within ± 0.04m, and Gaussian noise with standard deviation $\sigma = 0.005$. The randomized point cloud is then encoded by a PointNet encoder composed of an MLP sequence and global max pooling.

The proprioceptive input contains the end-effector position, end-effector quaternion, gripper state, and mobile-base state. We normalize the proprioceptive vector and fuse it with the PointNet feature. The fused feature is linearly projected and combined with positional encoding. We use a causal Transformer with time window $T = 10$, 4 layers, hidden dimension 512, 8 attention heads, and dropout 0.1. The Transformer output is passed to an action MLP to predict a 10-DoF action, consisting of 7-DoF arm/gripper control and 3-DoF mobile-base control:

$$
\begin{aligned}
a_t = & [\Delta x, \Delta y, \Delta z, \Delta r_x, \Delta r_y, \Delta r_z, \\
& a_{\text{gripper}}, \Delta x_{\text{base}}, \Delta y_{\text{base}}, \Delta \theta_{\text{base}}].
\end{aligned} \tag{37}
$$

Training setup. Each policy is trained using 100 demonstrations generated by SCENE2DEMO. We use one NVIDIA A800 GPU, batch size 1024, and train for 200 epochs. Each epoch samples 102400 training samples. We use the AdamW optimizer with an initial learning rate of 4×10^{-4}. The learning rate follows a linear decay schedule. The BC objective minimizes the prediction error between the policy action and the expert action generated by our pipeline.

Domain randomization. We apply domain randomization during data generation to obtain more diverse and robust demonstrations. Specifically, for each generated demonstration, we perturb the target-object scale along the XYZ dimensions with a scale noise of ± 0.10. For tasks involving articulated-object interaction, such as opening a refrigerator, we additionally perturb the target-object orientation around the z axis within $\pm \pi/12$. These perturbations force the policy to learn from varied object geometry and

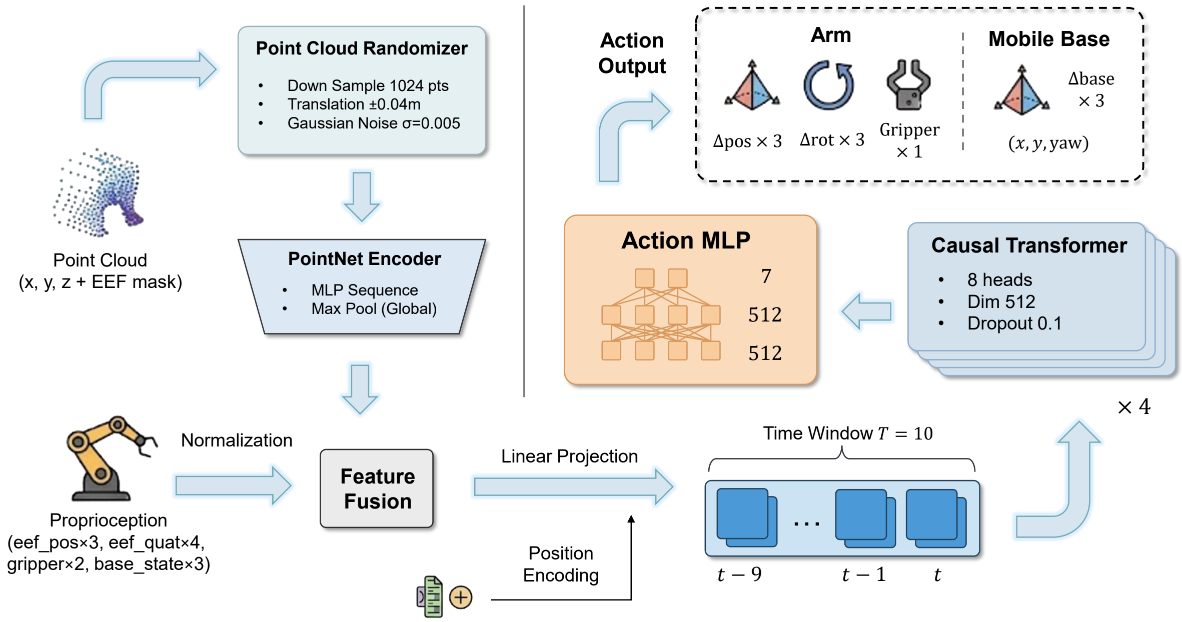


Figure 11. **Policy architecture for downstream behavior cloning.** The policy encodes randomized point clouds with a PointNet encoder, fuses the point-cloud feature with proprioceptive states, and predicts 10-DoF robot actions using a 4-layer causal Transformer and an action MLP.

pose configurations instead of overfitting to a single deterministic scene layout.

During policy evaluation, we apply the same domain-randomization setting as in data generation. In particular, the XYZ dimensions of the target objects are perturbed with scale noise ± 0.10, and the target-object rotation around the z axis is perturbed within $\pm\pi/12$ when applicable. This evaluation setting verifies that the learned policies retain robustness under object-scale and pose variations.

Results. The generated demonstrations can support effective downstream imitation learning. With 100 generated demonstrations, the BC policy achieves 96.0% success rate on *Open Refrigerator* with 24 successful trials out of 25, and 92.0% success rate on *Pick Up Glass* with 23 successful trials out of 25. These results show that SCENE2DEMO does not only produce visually plausible rollouts, but also generates action-observation trajectories that can be consumed by standard policy-learning pipelines.