

Code-as-Monitor: Constraint-aware Visual Programming for Reactive and Proactive Robotic Failure Detection

Supplementary Material

The supplementary document is organized as follows:

- Sec. A: More Discussions like Technical Details, Limitations, and Future Work.
- Sec. B: Implementation Details of Painter, including data collection, training, and element pipeline details.
- Sec. C: Environment Configuration, including environmental setups, control policies, and baseline details.
- Sec. D: Evaluation Details, including detailed task definition, disturbances, and experimental results.
- Sec. E: More Ablation Studies.
- Sec. F: More Demonstrations of CaM.

A. More discussions

A.1. Methodology discussions

What contributes to generalization. VLM’s intrinsic knowledge and reasoning enable task-level generalization, while geometric abstraction ensures generalization across scenes and entities (*e.g.*, objects, robots). The constraint-based formulation seamlessly integrates both aspects, achieving overall framework generalization.

Proactive detection for long-term dynamics. Our framework can potentially handle long-term dynamics by leveraging the Constraint Generator to decompose long-horizon tasks into subgoals and enables proactive detection within each subgoal to *constrain the future state space* rather than directly predict future states. Moreover, failed subgoals are re-planned with updated proactive constraints, ensuring adaptability to long-term dynamic changes.

Key methodology compared to ReKep [7]. Our key method differs from ReKep in two key aspects: **(1) Element extraction:** ReKep simply relies on DINOv2 for keypoint detection via 2D feature clustering, which offers a limited representation of constraints and geometric relationships crucial for failure detection. In contrast, our method achieves higher-order geometric abstraction by leveraging *ConSeg* to extract constraint-aware objects/parts in 2D space, then combining these with point clouds to integrate 3D geometric constraints. **(2) Code Generation:** Unlike ReKep, which lacks fine-grained guidance, our method leverages these *geometry-rich elements* as visual prompts

on *multi-view* images to generate code more accurately *at each subgoal’s beginning*, enabling seamless integration with various policies to form closed-loop systems.

Part segmentation for ambiguous or incomplete instructions. In our framework, we mainly leverage *ConSeg*’s abilities to handle cases where instructions with ambiguous or missing explicit part information. **(1)** We do not impose whether the textual instructions contain part information in our dataset collection pipeline. This allows the *ConSeg* to possess a certain ability to handle ambiguous instructions. **(2)** *ConSeg* architecture adopts a VLM-based design, leveraging VLM’s world knowledge to infer constraint-related parts for unseen objects. Based on our experiments (see Supp. F.4), *ConSeg* does demonstrate some generalization ability, but its performance on unseen objects is weaker than on objects within the training distribution.

A.2. Technical detail discussions

Executability and reliability of code. For executability, we validate the code using the path coverage of White-box Testing at the subgoal’s beginning, where the current state is used as input to verify the accuracy of each logic branch (*i.e.*, every if-else block). If any error occurs during testing, the code is regenerated. For reliability, we prompt VLM to generate code that closely adheres to the given specifications and requirements. As frequent VLM calls increase hallucinations, we only invoke VLM once at each subgoal’s beginning, making it less affected by hallucination and more reliable.

Failure threshold Selection. We use two ways: **(1)** manually initializing thresholds for common tasks in an external knowledge base; **(2)** generating thresholds for unseen tasks using VLM’s internal knowledge. We find that the system remains robust to threshold selection in common settings, but the impact of threshold variations under diverse conditions remains unexplored and is left for future work.

Computational cost. The segmentation model is only invoked at the subgoal’s beginning to generate elements and *excluded during execution*, ensuring real-time code-based monitoring by tracking only the elements without high computational cost. We also perform parallel inference of segmentation models to accelerate at the subgoal’s beginning.

A.3. Limitations and Future Work

Despite promising results, our approach has several limitations. First, using Visual Language Models (VLMs)—such

as off-the-shelf GPT for code generation and constraint-aware segmentation models—inevitably leads to hallucination issues. Even with minimized VLM usage, inaccuracies in code generation and biases in segmentation may persist. Integrating more relevant knowledge via methods like Retrieval-Augmented Generation (RAG) [10] or reducing multimodal large language model (MLLM) hallucinations [15] could alleviate these problems. Second, while we unify reactive and proactive failure detection as spatio-temporal constraint satisfaction problems and propose constraint elements for simplified real-time, high-precision monitoring, the constraint element representation has limitations: (1) It primarily focuses on failures detectable through explicit displacement and rotation, rendering it less effective for force-direction-related failures without noticeable displacement—for example, a robotic gripper failing to open a drawer due to incorrect force application. (2) The simplified representation abstracts objects and minimizes irrelevant visual details for efficient failure detection but may overlook critical visual cues and multimodal inputs, such as flowing water or audible sounds from a partially closed faucet, which are ignored in our current framework. Thus, exploring more robust representations that balance real-time precision with minimal information loss or integrate richer multimodal inputs is a promising direction for future research.

B. Constraint Painter

In this section, we first describe the data collection and annotation process of the Constraint-aware Segmentation Dataset. Next, we present the training details of the proposed *ConSeg* model. Finally, it is followed by a detailed explanation of the Element Pipeline.

B.1. ConSeg Data Collection

To ensure broader coverage of scenarios and objects in our dataset, along with text-based instructions, we utilized the BridgeData V2 dataset [20]. This large and diverse dataset of robotic manipulation behaviors comprises 60,096 trajectories collected across 24 environments, encompassing 13 distinct skills.

The dataset collection pipeline is shown in Fig. 1. The entire process is divided into three stages, *i.e.*, trajectory decomposition, assigning textual information, and dataset collection. First, we decompose each trajectory’s instruction and initial observation from BridgeData V2 into subgoals for each stage, along with the constraints upon completion, constraints during execution, the corresponding object-part associations and element type for each constraint. Subfigure 1 in Fig. 1 illustrates a specific example, demonstrating the decomposition of the task “Take cup off plate”. Notably, the third subgoal, “Place the cup on the stove”, indicates that the initial observation ensures the decomposition process

fully understands the task’s contextual environment. During this process, we also obtain the constraint element type, which serves as the ground-truth text response for part-level constraint-aware segmentation. This decomposition is performed using the off-the-shelf GPT-4o API.

After obtaining the subgoals, constraints, and object-part associations for each stage, we need to assign each frame to its corresponding stage. Since BridgeData V2 does not provide per-frame annotations, we addressed this issue by sampling pick-and-place data and leveraging the additional information (*e.g.*, gripper open/close states) provided by BridgeData V2 for assignment. The pick-and-place task is typically divided into three stages: Approach, Grasp and Transfer, and Place, corresponding to the gripper states of open, closed, and open, respectively. Leveraging the characteristics of the pick-and-place task, we complete the frame-level assignment. Subfigure 2 in Fig. 1 illustrates a specific example of the assignment process.

Using the obtained frame-level constraint-aware object and part information, Instance-level and part-level segmentations are performed using Grounded SAM [17] and Semantic SAM [12], respectively. We conducted a sampled manual inspection of the final annotations to filter out errors and low-quality labels. Our final dataset is composed of 10,181 trajectories with 219,356 images.

B.2. ConSeg Training Details

We adopt LISA’s [9] loss function, including the next-token prediction loss for text output, and the combination of per-pixel BCE loss with DICE loss for mask output. Our *ConSeg*-13B model is trained on an 8 NVIDIA 80G H800 GPU for two days with a batch size of 4. Our training data comprises multiple components: SemanticSeg, ReferSeg, VQA, ReasonSeg, and ConstraintSeg, to ensure our model retains dialogue and reasoning segmentation capabilities while achieving constraint-aware segmentation. LISA inspires this training data setting. SemanticSeg includes ADE20K, COCO-Stuff, PACO-LVIS, and PASCAL-Part. ReferSeg includes refCLEF, refCOCO, refCOCO+, and refCOCog. VQA includes LLaVAInstruct-150k. ConstraintSeg includes instance and part-level data.

The training setting described above is for the *ConSeg*-base model. Since the training data consists entirely of real-world scenarios, there is a significant gap between the simulation and real-world environments. To address this, the *ConSeg* model used in the simulation experiments is a fine-tuned version, called *ConSeg*-ft, finetuned on a small amount of data collected from the simulator. Specifically, we collect 100 trajectories from each simulator, sampled frames at 1 Hz, and utilize either ground truth masks from the simulator or annotations generated using Grounded SAM and Semantic SAM. **Notably, we use the *ConSeg*-base model in real-world experiments, demon-**

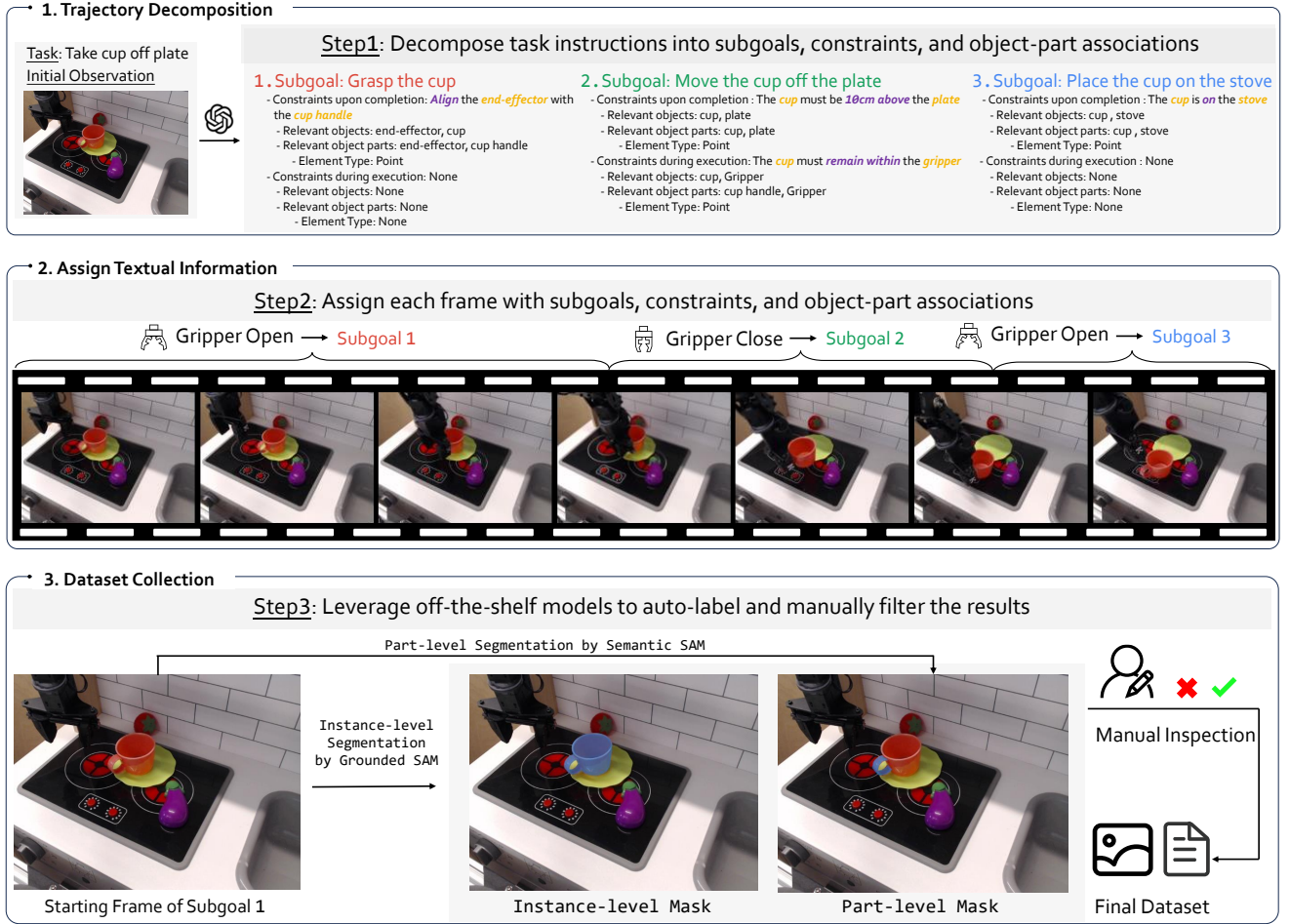


Figure 1. Dataset Collection Pipeline. Our data is sourced from BridgeData V2 [11]. The data collection process consists of three steps: (1) Using GPT-4o [1] to decompose the task instruction based on the initial observation from the first frame of the trajectory, generating subgoals along with two types of constraints for each subgoal (*i.e.*, constraints during execution and upon completion) and object-part associations. (2) Utilizing external references (*e.g.*, gripper open/close states) to assign subgoals, constraints, and object-part associations to each frame. (3) Leveraging off-the-shelf models (*e.g.*, Grounded SAM [17], Semantic SAM [12]) to generate instance- and part-level masks (blue mask in this figure) automatically, followed by manual filtering to curate the final dataset.

strating the model’s generalization capability across different scenarios.

B.3. Element Pipeline Details

Here, we provide additional details about the Constraint Element Pipeline. We first filter out outliers after obtaining the constraint-aware object/part point clouds. Next, we calculate the 3D spatial bounds occupied by the remaining point cloud and determine the voxel size for voxelization based on the element type. We then perform point cloud clustering using the DBSCAN algorithm, which has advantages over other methods, including identifying clusters of arbitrary shapes, eliminating the need to predefine the number of clusters, and its effectiveness in high-density regions.

C. Environment Configuration

We first provide detailed descriptions of the simulators (CLIPort [19], Omnigibson [11], RLBench [8]) and real-world setups used in our study. We then discuss the low-level control policies implemented in these environments. Finally, we present the baselines and their implementation specific to each environment. **Notably, our framework, CaM, is policy agnostic, meaning it can be adapted to any control policy without requiring any modification.**

C.1. Environmental Setup

The CLIPort [19] simulator¹ is a robotic manipulation benchmark to gather extensive data for imitation learning

¹<https://github.com/cliport/cliport>

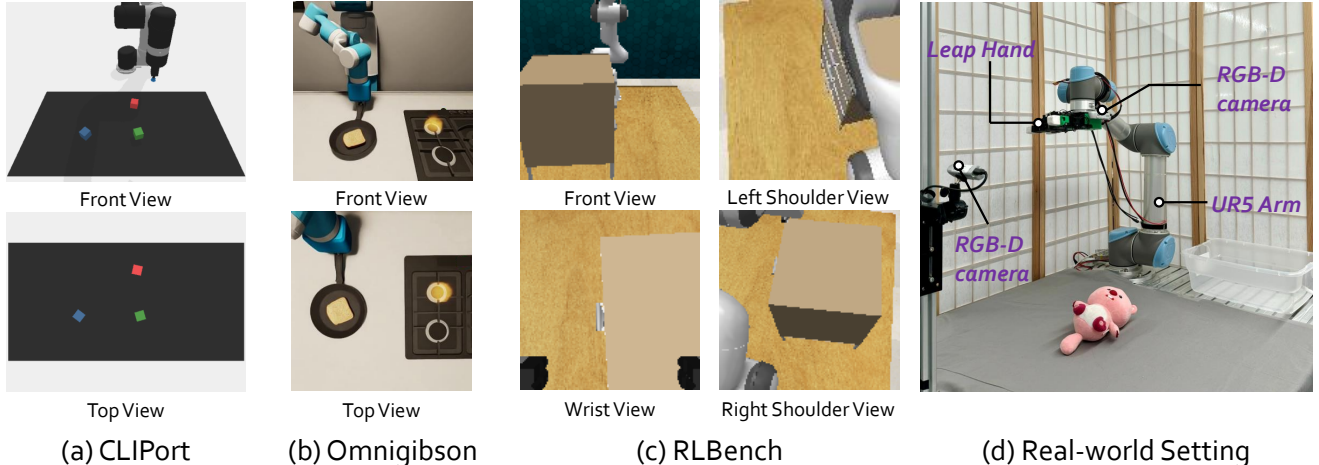


Figure 2. Environmental Setup of three simulators and one real-world setting. For CLIPort [19] and OmniGibson [11], we provide third-person front and top views and the robot platforms are the UR5 arm and Fetch, respectively. RLBench [8] offers four camera views, including front left shoulder, right shoulder, and wrist views, with the robot platform being Franka equipped with a gripper. We provide a wrist and a third-person front view for the real-world setting, utilizing a UR5 robot equipped with a Leap Hand [18].

and train a language-conditioned multi-task low-level control policy. The environment features a UR5 robotic arm with a suction cup as the end effector for pick-and-place tasks and a spatula as the end effector for pushing tasks, both operating on a black tabletop. We use two cameras: a third-person front view and a top view to provide comprehensive perspectives of the tabletop, as shown in Fig. 2 (a).

The Omnigibson [11] simulator² offers a realistic setting including a physics engine capable of supporting features such as lighting rendering, gravity effects, and temperature variations impacting objects within the environment. This platform also provides an extensive array of pre-configured scenes and objects, enabling researchers to customize setups and train mobile manipulation robots. We select version 1.1.0 for our study. This simulator involves a Fetch robot equipped with a gripper as the end effector, operating on a white tabletop. We utilize two cameras: a third-person front view and a top view to provide comprehensive perspectives of the tabletop, as shown in Fig. 2 (b).

RLBench [8] simulator³ is a widely used benchmark for robot manipulation, featuring tasks such as articulated objects and tool use. Researchers can gather data and train low-level control policies using imitation learning or reinforcement learning within this environment. RLBench features a Franka robotic arm with a gripper as its end effector, operating on a brown tabletop. Four cameras provide comprehensive tabletop coverage, as shown in Fig. 2 (c). Additionally, RLBench uses a sampling-based motion planner for motion planning given the next predicted action/pose.

In the real-world setup depicted in Fig. 2 (d), we utilize

a fixed UR5 robotic arm with a Leap Hand [18] as the end effector. Two RealSense D415 RGB-D cameras capture the scene, one mounted on the wrist and the other positioned for a third-person front view.

C.2. Control Policy

In CLIPort, we use a pre-trained low-level policy the CLIPort [19] simulator provides to control the robotic arm and end effector. This policy can execute multi-task operations based on language instructions with RGB observations and its performance approaches perfection due to extensive imitation learning training. Notably, the policy is open-loop, meaning it does not adjust its actions in response to dynamic environmental changes (*e.g.*, it will not immediately pick up a dropped block during movement but will continue with the previously planned actions).

In Omnigibson, We utilize ReKep [7] as our low-level control policy, transforming long-horizon tasks into a set of relationships between fixed keypoints at different stages. At each stage, an optimization algorithm computes these relationships to generate actions, enabling language-conditioned closed-loop control. Notably, ReKep employs a pre-trained large vision model (*i.e.*, DINOv2 [16]) to process raw RGB data, extracting semantically relevant keypoints. This approach also serves as the compared method in our ablation study for extracting 3D points through constraint-aware segmentation, showing our superiority.

In RLBench, we employ the Autoregressive Policy (ARP) [22] as the control policy, which generates the next action based on historical observations and action sequences through an autoregressive process. This method achieves state-of-the-art performance in the RLBench.

²<https://github.com/StanfordVL/Omnigibson>

³<https://github.com/stepjam/RLBench>

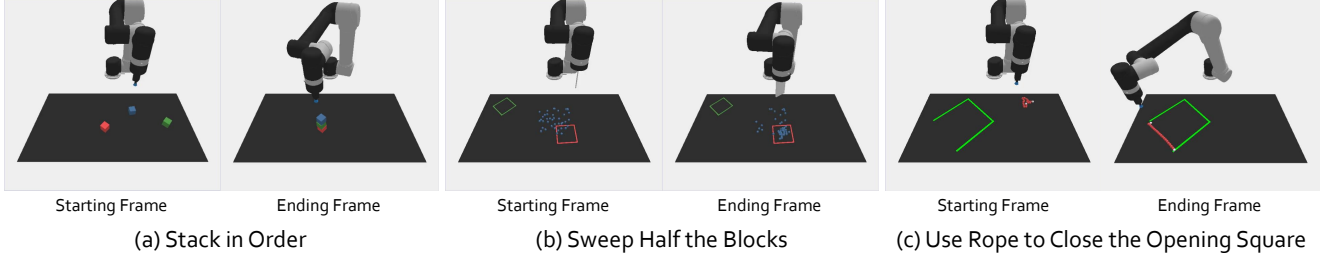


Figure 3. CLIPort task demonstration. we present three types of tasks in our experiments, including the starting and ending frames.

In the real-world setting, We employ DexGraspNet 2.0 [21] as our low-level policy, which predicts the dexterous hand’s grasping pose based on the scene’s point cloud and facilitates the robotic arm’s action trajectory through motion planning to achieve robust generalized grasping. Notably, DexGraspNet 2.0 is an open-loop policy, which means it does not adjust to environmental changes during action execution. For example, if the target object’s position shifts while the arm moves, the system does not modify its motion plan. Therefore, it continues to execute toward the originally intended location to complete the grasp, failing.

C.3. Baseline Details

In CLIPort, we compare three baselines: CLIPort [19], CLIPort with Inner Monologue [6], and CLIPort with DoReMi [5]. We slightly modify the original implementations of these three baselines to suit our task requirements. (1) For CLIPort [19], the sole modification involves substituting the original oracle success detector with an off-the-shelf VLM (*i.e.*, GPT-4o [1]) used as a failure detector. This change enables the robot system to determine whether to transition to the next subgoal using image-based vision question answering (VQA). Notably, CLIPort decomposes the instructions into a list of subgoals before the task begins and does not dynamically adjust or revert to previous subgoals upon detecting a failure. (2) For Inner Monologue [6], we replicate the implementation detailed in the original literature, by employing CLIPort for the low-level policy and an off-the-shelf VLM (*i.e.*, GPT-4o [1]) as the planner. This pipeline determines the next subgoal based on the completed subgoals and current observations after each subgoal concludes. Notably, Inner Monologue queries the VLM only at the end of each subgoal, without considering events that occur during execution. (3) For DoReMi [5], we reproduce the implementation according to the original DoReMi paper and enhance it by replacing the VLM initially (*i.e.*, BLIP2 [13]) used for repeated VQA-style queries during robotic execution with a more powerful VLM (*i.e.*, GPT-4o [1]). Additionally, we substitute its LLM, which lacks environmental awareness, with the same GPT-4o to serve as the task planner.

In Omnigibson, we compare two baselines: ReKep [7]

and ReKep with DoReMi [5]. (1) For ReKep, we directly implement it using the official codebase. (2) For DoReMi, it is implemented as described above.

In RLBench, we compare two baselines: ReKep [7] and ReKep with DoReMi [5]. (1) For ReKep, we directly implement it using the official codebase. (2) For DoReMi, it is implemented as described above.

For the real-world evaluation, we compare two baselines: DexGraspNet 2.0 [21] and DexGraspNet 2.0 with DoReMi [5]. (1) For DexGraspNet 2.0, we directly implement it using the official codebase. (2) For DoReMi, it is implemented as described above.

D. Evaluation Details

In this section, we first detail the task specifications within the simulator and real-world evaluations. Then, we introduce the disturbances introduced in each task and the evaluation metrics used. Finally, we report the detailed experimental results and our analyses, with additional results not included in the main text due to space constraints.

D.1. CLIPort

D.1.1 Task, Disturbance and Metric Details

As shown in Fig. 3, we evaluate three tasks in CLIPort: (1) “Stack in Order”: Given blocks in red, green, and blue on a table, the robot must stack them with red at the bottom, green in the middle, and blue on top. (2) “Sweep Half the Blocks” With 40 blocks on the table, the robot must sweep approximately half of them (with a permissible error margin of $\pm 10\%$, *i.e.*, 16 to 24 blocks) to a designated area. (3) “Use Rope to Close the Opening Square”: The robot should use a rope to enclose an open square, to enclose the area sufficiently, rather than form a perfectly closed square.

We introduce two types of disturbances to the “Stack in Order” task: (1) after the suction cup grasps a block, there is a probability p at each step that the block will be released, causing it to drop; (2) The predicted placement position by the policy is perturbed by a uniform noise in the range $[0, q]$ cm, potentially leading to block tower collapse.

For each task and disturbance type, we conduct 5 trials using different seeds, each comprising 12 episodes. We as-

Table 1. Detailed Performance in CLIPort. We report the success rate and execution time for three tasks, compared to baseline methods.

Tasks with disturbance	Success Rate(%) \uparrow				Execution Time(s) \downarrow		
	CLIPort	+Inner Monologue	+DoReMi	+Ours	+Inner Monologue	+DoReMi	+Ours
Stack in order with drop $p=0.0$	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00	13.40 ± 1.82	13.40 ± 1.82	13.40 ± 1.82
$p=0.15$	56.67 ± 6.11	81.67 ± 6.11	83.33 ± 5.17	95.00 ± 4.00	34.80 ± 3.12	26.00 ± 2.77	21.00 ± 1.75
$p=0.3$	21.67 ± 8.33	75.00 ± 8.95	76.67 ± 9.52	88.33 ± 6.53	42.80 ± 3.18	34.20 ± 2.73	25.40 ± 2.95
Stack in order with noise $q=1$	90.00 ± 6.11	90.00 ± 6.11	96.67 ± 4.00	98.33 ± 3.27	24.80 ± 4.08	24.60 ± 4.66	24.20 ± 4.65
$q=2$	41.67 ± 7.30	71.67 ± 8.33	75.00 ± 5.17	83.33 ± 5.17	39.40 ± 5.87	37.00 ± 6.29	29.20 ± 4.61
$q=3$	15.00 ± 6.11	40.00 ± 8.00	40.00 ± 6.11	63.33 ± 8.33	58.20 ± 4.74	54.20 ± 6.02	36.80 ± 4.61
Sweep Half the Blocks	0.00 ± 0.00	18.33 ± 6.11	16.67 ± 8.95	75.00 ± 11.55	22.00 ± 2.91	16.60 ± 1.33	16.40 ± 1.00
Use Rope to Close the Opening Square	0.00 ± 0.00	68.33 ± 9.52	58.33 ± 18.62	76.67 ± 6.11	41.60 ± 6.34	65.80 ± 7.40	34.60 ± 2.81

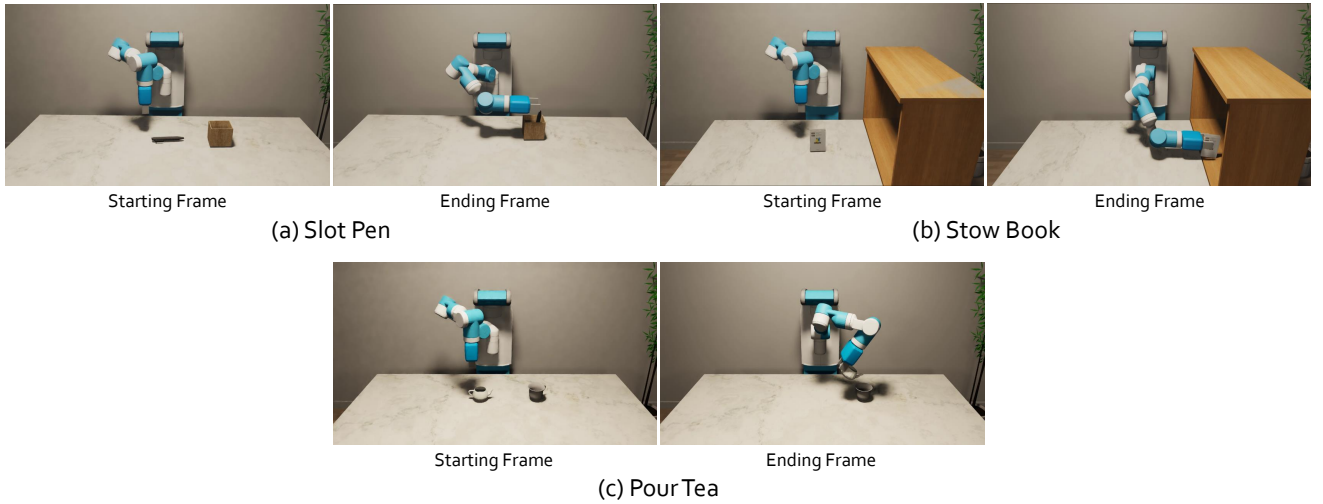


Figure 4. Omnigibson task demonstration. we present three types of tasks in our experiments, including the starting and ending frames.

sess performance based on success rate and execution time, **excluding** the computational time for invoking the VLM. Results are reported as mean values with 95% confidence intervals. In the “Stack in Order” task, the robot must successfully stack the blocks in the specified order into a tower within 70 seconds, despite the two perturbations above. For the “Sweep Half the Blocks” task, the pre-trained policy aims to sweep blocks into a designated area. The robot must stop the policy once half of the blocks are in the target region. If, after 30 seconds, the number of blocks in the area falls within the required range (16–24), the task is considered successful. For the “Use Rope to Close the Opening Square” task, the pre-trained policy attempts to close an open rectangle into a perfect square using a rope. The robot must detect when the rectangle is sufficiently enclosed and immediately stop execution. Success is achieved if the robot halts within 70 seconds, and the enclosure is complete.

D.1.2 Detailed Experiment Results

In Tab. 1, we present detailed results in CLIPort, including those discussed in the main text and additional results.

In the “Stack in Order” task under severe interference conditions, our *CaM* shows an improvement of 18.33% and 17.5% in success rate over Inner Monologue [6] and DoReMi [5], respectively, while also reducing execution times by 38.7% and 14.4% compared to Inner Monologue and DoReMi, respectively. The failure detection and recovery processes are shown in Fig. 6 and Fig. 7.

In the “Sweep Half the Blocks” task, our *CaM* achieved success rates that are 4.1 and 4.5 times higher than those of Inner Monologue [6] and DoReMi [5], respectively. However, the success rate is not very high even in distraction-free scenarios. This is attributed to the high density of tracking points in the scene, which increases the likelihood of confusion and tracking errors, leading to inconsistent block counts within the target area. The completion process of the

task is also illustrated in Fig. 8.

In the “Use Rope to Close the Opening Square” task, our approach outperforms Inner Monologue [6] and DoReMi [5] by 8.34% and 18.34% in success rates, respectively, while also reducing execution times by 16.82% and 47.43% compared to Inner Monologue and DoReMi, respectively. We find that calculating the distance between the rope ends and the opening’s edges to determine closure is more accurate than directly querying a VLM with an image, allowing for earlier termination of the policy execution. The complete processes of the task are illustrated in Fig. 9.

D.2. OmniGibson

D.2.1 Task, Disturbance and Metric Details

As shown in Fig. 4, in the OmniGibson environment, we evaluated three distinct tasks: (1) Slot Pen: A pen placed on a desk is picked up, rotated to a near-vertical position, moved above a pen holder, and then inserted into the holder. (2) Stow Book: A book located on a desk is picked up and vertically positioned on a bookshelf. (3) Pour Tea: A teapot on the desk is lifted, horizontally moved above a teacup, and then tilted to pour tea into the cup.

We introduce three types of disturbances with varying constraint elements for each task: (1) Slot Pen Task: Point-level disturbances are applied as follows: (a) moving the pen while the robot is grasping it, (b) forcing the robot to release the pen mid-transfer, causing it to drop onto the table, and (c) moving the pen holder while the robot attempts to insert the pen. Despite these disturbances, the task is considered successful only if the robot can insert the pen into the holder. (2) Stow Book Task: Line-level disturbances include: (a) rotating the book during the robot’s grasping process, (b) altering the book’s pose during transfer to disrupt its vertical alignment, and (c) reorienting the book horizontally after it has been placed vertically on the shelf. Success requires the robot to place the book vertically on the shelf despite these disturbances. (3) Pour Tea Task: Surface-level disturbances involve: (a) tilting the container forward or backward during transfer, (b) inducing lateral tilts during movement, and (c) restoring the container to a level position during pouring. To succeed, the robot must prevent spillage and complete the pouring task under these disturbances.

We conduct experiments on three tasks, each consisting of one no-disturbance trial and three specific-disturbance trials. Each trial was repeated 10 times, and the performance was evaluated based on success rate, execution time (including the computation time for invoking the VLM), and the number of tokens used.

D.2.2 Detailed Experiment Results

Specific experimental results are detailed in the main text; here, we present additional demonstrations in Sec. F.2.

D.3. RL Bench

D.3.1 Task, Disturbance and Metric Details

As shown in Fig. 5, in RL Bench, we evaluate six tasks across three categories of manipulation: (1) Articulated Object Interaction: (a) Open Drawer—Open the top drawer (b) Put in Drawer—Open the drawer and place an item into the open drawer. (2) Rotational Manipulation: (a) Screw Bulb—Screw in the red light bulb. (b) Turn Tap—Turn the left tap. (3) Tool Use: (a) Drag Stick—Use a stick to drag the cube onto the red target. (b) Sweep to Dustpan—Sweep dirt into the tall dustpan.

In RL Bench, we avoid introducing additional disturbances, as its control policy naturally generates diverse failures to validate the effectiveness of our framework. The RL Bench-trained policy lacks failure recovery mechanisms; thus, any episode flagged as a failure by the detection framework is deemed invalid, and a new episode is initiated.

For each task, we evaluate performance over 1,000 valid episodes (maximum 25 steps each), measured by the average success rate.

D.3.2 Detailed Experiment Results

Code with elements can generalize better to monitor diverse tasks. Notably, despite our training data lacking information on articulated objects at both the instance and part levels, our method effectively handles them, accurately segmenting parts such as drawer handles. In the “Open Drawer” task, CaM achieves a 98.1% success rate, significantly outperforming DRM’s 90.6%. For “Put in Drawer”, CaM reaches 98.3%, surpassing DRM’s 87.7% by 10.6 percentage points. **We attribute this generalization to two factors: (1) the inherent prior world knowledge of extensively pre-trained VLMs (e.g., SAM, LLaVA), which enables generalization to unseen tasks; and (2) our minimalist scheme that abstracts articulated objects into geometric components via constraint elements, ignoring irrelevant details, enhancing the generalizability.**

We also demonstrate our method on additional unseen tasks, such as rotational manipulation and tool use, where it consistently outperforms baseline methods.

D.4. Real-world Evaluation

D.4.1 Task, Disturbance and Metric Details

We conduct real-world evaluations on two tasks: (1) Simple Pick & Place: The robot should pick and place objects within 70 seconds. We include four kinds of objects: Deformable, Transparent, Small Rigid, and Large Geometric, with three objects in each category. The deformable objects are Loopy, Dog, and Rabbit toys, which undergo deformation when grasped by the dexterous hand. The transparent

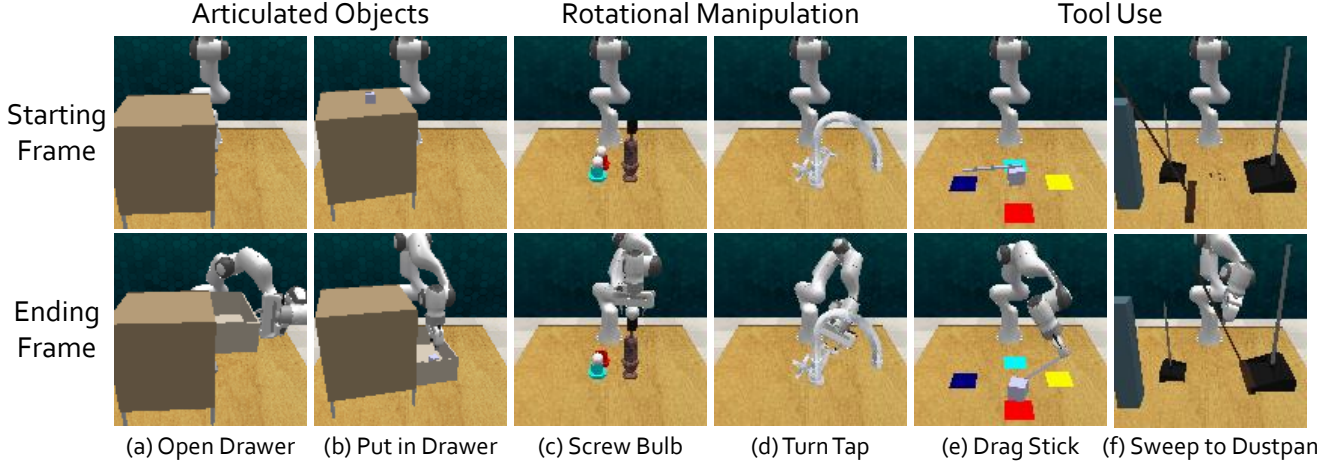


Figure 5. RL Bench task demonstration. we present six types of tasks in our experiments, including the starting and ending frames.

Table 2. Performance in RL Bench. We report the success rate, compared to baseline methods.

Method	Avg. Success Rate (%) \uparrow	Articulated Object		Tool-Use		Tool-Use	
		Open Drawer	Put in Drawer	Screw Bulb	Turn Tap	Drag Stick	Sweep to Dustpan
RVT2 [4]	89.83	90.3	97.6	86.6	91.0	93.8	79.7
ARP [22]	91.27	93.9	91.0	86.4	96.6	88.1	91.6
+DRM [5]	87.97	90.6	87.7	83.1	93.3	84.8	88.3
+Ours	97.08	98.1	98.3	97.5	97.9	95.6	94.0

objects are a clear beverage bottle, a transparent glass, and a clear shampoo bottle. The small rigid objects consist of apple, pear, and peach models, while the large geometric objects include a large plate, ball, and pyramid. (2) Reasoning Pick & Place: In cluttered scenes, the robot must perform long-horizon tasks where the instructions contain ambiguous terms (*e.g.*, “animals” or “fruits” without specifying particular types). Specifically, the tasks are: (a) “Clear all objects on the table except for animals”, and (b) “Grasp the animals according to their distances to fruits, from nearest to farthest”, with ambiguous terms underlined.

For both tasks, we introduce two identical disturbances: (1) Moving the object during the robot’s grasping. (2) Removing the object from the robot’s dexterous hand during movement after grasping.

For each task, and each object involved in Simple Pick & Place, we conduct 10 trials. For each long-horizon task in Reasoning Pick & Place, we also conduct 10 trials. We evaluate performance based on success rate and execution time, **including** the computational time invoking the VLM. Results are reported as mean values with 95% confidence intervals. For the Simple Pick & Place, the robot has only one opportunity to autonomously release the object held in its gripper at a designated location. Any disturbance the robot encounters allows for a return and reattempt at grasping if the robot successfully detects it. Success is defined as

meeting these conditions within 90 seconds. For the Reasoning Pick & Place task, the robot must clear all objects on the table except for animals within 4 minutes for “Clear all objects on the table except for animals”. In the task “Grasp the animals according to their distances to fruits, from nearest to farthest”, the robot must sequentially grasp the animals in the correct order within 2 minutes, despite human-induced distractions such as moving animals or fruits. **Notably, this task is particularly challenging because the robot operates under an open-loop policy, preventing it from using closed-loop feedback to handle the dynamic distances between fruits and animals caused by external disturbances. Therefore, a failure detection framework is necessary to enable both reactive and proactive real-time detection with high precision, monitoring the distance changes and adjusting the grasping sequence accordingly.**

D.4.2 Detailed Experiment Results

In Tab. 3, we present detailed results of Simple Pick & Place. CaM achieves success rates surpassing DoReMi by 20.4% when handling different kinds of objects. We show real-world demonstrations of Simple Pick & Place and Reasoning Pick & Place in Sec. F.3.

Table 3. Detailed Performance of Single Pick & Place. We report the success rate and execution time. DGN is DexGraspNet 2.0 [21].

Tasks with disturbance	Object types	Object Name	Success Rate(%) \uparrow			Execution Time(s) \downarrow	
			DGN	+DoReMi	+Ours	+DoReMi	+Ours
Pick & Place with the objects being moved during grasping	Deformable	Toy Loopy	0.00	80.00	100.00	64.91 ± 2.83	46.02 ± 3.11
		Toy Dog	0.00	80.00	100.00	60.68 ± 4.00	47.06 ± 3.24
		Toy Rabbit	0.00	90.00	90.00	59.83 ± 1.82	45.77 ± 2.03
	Transparent	Beverage Bottle	0.00	60.00	100.00	69.97 ± 7.89	47.61 ± 2.58
		Glass Cup	0.00	70.00	90.00	76.99 ± 4.60	48.32 ± 3.22
		Shampoo Bottle	0.00	70.00	90.00	70.91 ± 5.68	48.31 ± 3.08
	Small Rigid	Apple Model	0.00	80.00	100.00	64.65 ± 4.34	45.39 ± 0.71
		Pear Model	0.00	90.00	90.00	67.11 ± 1.10	45.48 ± 1.01
		Peach Model	0.00	70.00	90.00	65.48 ± 2.90	45.37 ± 0.64
	Large Geometric	Plate	0.00	80.00	100.00	69.86 ± 2.64	45.18 ± 0.65
		Ball	0.00	90.00	100.00	67.43 ± 2.63	45.37 ± 0.70
		Pyramid	0.00	90.00	90.00	69.14 ± 3.32	45.42 ± 0.72
Pick & Place with the objects being removed during movement	Deformable	Toy Loopy	0.00	80.00	90.00	69.29 ± 4.87	60.86 ± 3.41
		Toy Dog	0.00	70.00	100.00	66.09 ± 2.99	63.12 ± 3.75
		Toy Rabbit	0.00	80.00	90.00	70.86 ± 4.56	63.40 ± 3.88
	Transparent	Beverage Bottle	0.00	50.00	90.00	77.90 ± 2.89	61.97 ± 3.90
		Glass Cup	0.00	70.00	90.00	70.00 ± 3.46	63.22 ± 4.35
		Shampoo Bottle	0.00	60.00	90.00	60 ± 4.28	63.00 ± 3.81
	Small Rigid	Apple Model	0.00	70.00	90.00	70.21 ± 4.30	63.71 ± 3.91
		Pear Model	0.00	60.00	100.00	72.70 ± 4.84	58.61 ± 2.41
		Peach Model	0.00	60.00	90.00	66.48 ± 3.32	59.19 ± 2.59
	Large Geometric	Plate	0.00	90.00	100.00	72.00 ± 2.77	59.21 ± 2.61
		Ball	0.00	70.00	100.00	70.92 ± 3.37	61.57 ± 3.80
		Pyramid	0.00	70.00	90.00	73.83 ± 2.82	60.25 ± 3.25

E. More ablation studies

Segmentation model ablations. Tab. 4 presents further ablation studies, replacing *ConSeg* with LISA and PixelLM. Our *ConSeg* shows superior overall framework performance (check Tab. 4 ID A & B & E), which represents a key technical contribution.

Failure detection mode ablations. (1) In Tab. 4 (ID C), proactive failure detection alone yields a lower success rate due to its inability to handle unforeseen failures; (2) In Tab. 4 (ID D), reactive detection alone achieves a slightly higher success rate but incurs longer execution times, as it only responds post-failure; (3) In Tab. 4 (ID E), the synergy of both modes achieves the best performance by addressing the limitations of each mode.

F. More Demonstrations and Prompts

This section presents additional demonstrations, including simulations and real-world scenarios of failure detection and recovery, as well as constraint-aware segmentation results and prompts.

F.1. CLIPort

Here, we present demonstrations of three tasks in CLIPort: “Stack in Order”, “Sweep Half the Blocks”, and “Use Rope

to Close the Opening Square”.

Fig. 6 demonstrates how our framework detects failures and assists in recovery when the placement positions predicted by the policy for the “Stack in Order” task are subject to a uniform $[0, q]$ cm interference.

Fig. 7 illustrates how our framework performs failure detection and aids in recovery when, in the “Stack in Order” task, there is a probability p that blocks will fall due to being released by the robot’s suction cup at each step.

Fig. 8 shows the “Sweep Half the Blocks” task, where our framework precisely counts the blocks within a specified area and timely halts the policy execution to complete the task. In contrast, DoReMi [5] fails to stop the policy execution in time, leading to task failure.

Fig. 9 depicts the “Use Rope to Close the Opening Square” task. Our framework effectively detects when the rope closes the opening square and promptly stops the policy execution to complete the task successfully. Conversely, DoReMi fails to halt the policy execution on time; although it eventually succeeds in closing the opening, the excessive execution time results in task failure.

F.2. OmniGibson

As shown in Fig. 10, Fig. 11 and Fig. 12, we show how our framework detects failures and assists in recovery when

Table 4. Following the Omnigibson evaluation protocol, we report the average success rate under disturbance (SR) and execution time to assess the impact of *ConSeg* and various failure detection modes on overall framework performance.

ID	Method	Slot Pen		Stow Book		Pour Tea	
		SR(%) \uparrow	Time(s) \downarrow	SR(%) \uparrow	Time(s) \downarrow	SR(%) \uparrow	Time(s) \downarrow
A	LISA	30.00	126.89	42.50	118.93	24.00	218.92
B	PixelLM	29.50	134.10	41.00	124.26	24.50	214.04
C	Only Proactive	37.50	130.15	50.00	109.47	32.50	192.23
D	Only Reactive	42.50	157.63	57.50	147.95	35.50	284.15
E	Ours	47.50	101.85	65.00	93.08	40.00	174.55

facing point-, line- and surface -level disturbances.

F.3. Real-world Evaluation

Fig. 13 demonstrates the task “Clear all objects on the table except for animals”, where our framework achieves both reactive failure detection (*e.g.*, detecting unexpected failures when humans remove objects from the robot’s grasp) and proactive failure detection (*e.g.*, identifying target object movement during grasping to prevent foreseeable failures). This effectively enhances the task success rate and reduces the execution time.

F.4. Constraint-aware segmentation

As shown in Fig. 14, Fig. 15, Fig. 16, and Fig. 17 we present additional results on constraint-aware segmentation, including instance and part-level results. **To demonstrate generalizability, we utilize out-of-distribution (OOD) data, including the RoboFail Dataset from REFLECT [14], datasets from the Open6DOF benchmark [3], and the RT-1 dataset [2].** Additionally, we showcase segmentation results from the OmniGibson.

F.5. Prompts

As illustrated in Fig. 18, we detail the prompt used to invoke an off-the-shelf VLM, *i.e.*, GPT-4o [1], to generate Python code for monitoring.

Stack in Order — The placement position is perturbed by a uniform noise

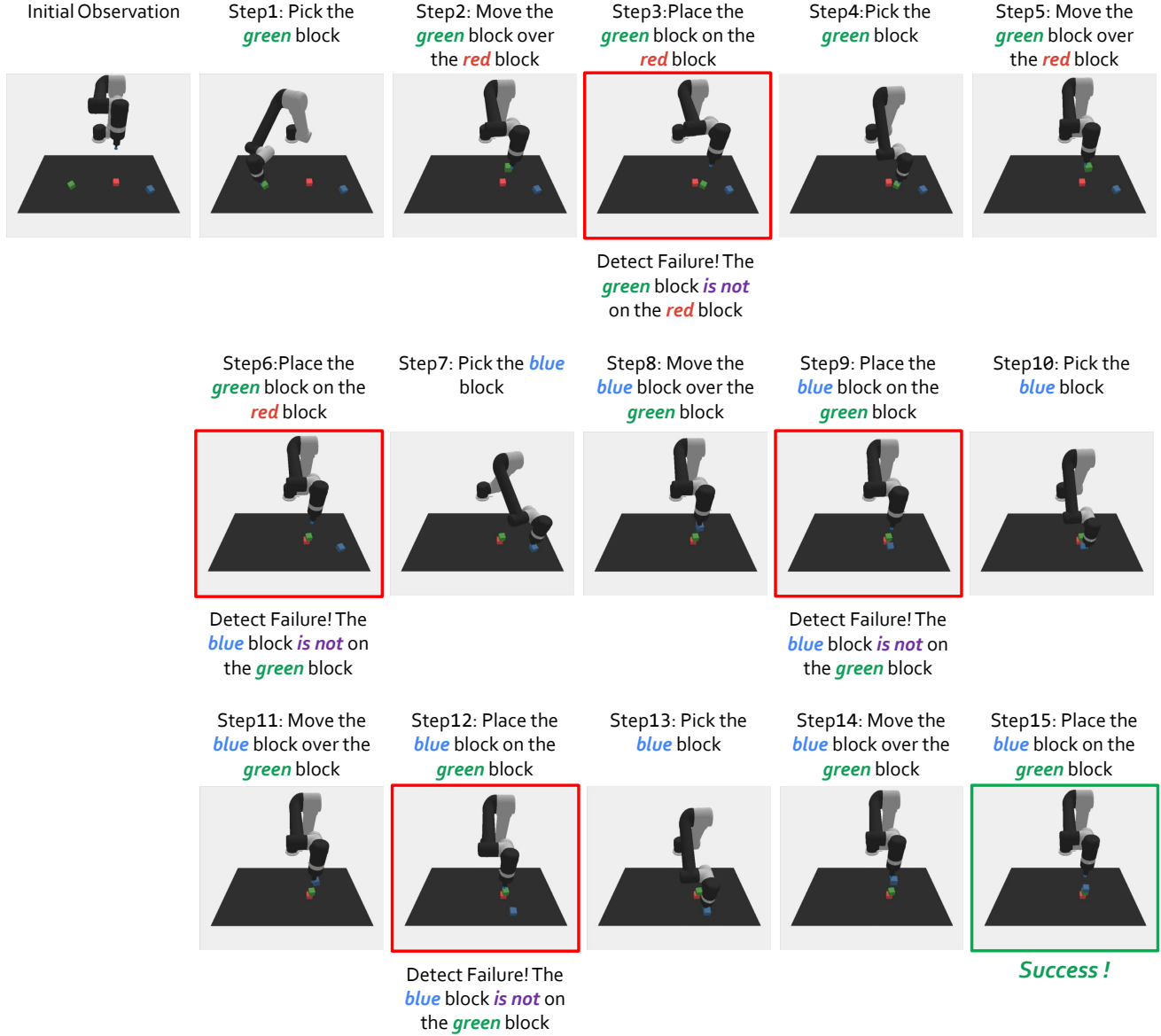


Figure 6. Demonstration of “Stack in Order”. We show how our framework detects failures and assists in recovery when the placement positions predicted by the policy for the “Stack in Order” task are subject to a uniform $[0, q]$ cm interference. Red boxes indicate the occurrence of failures, while green boxes signify successful task execution.

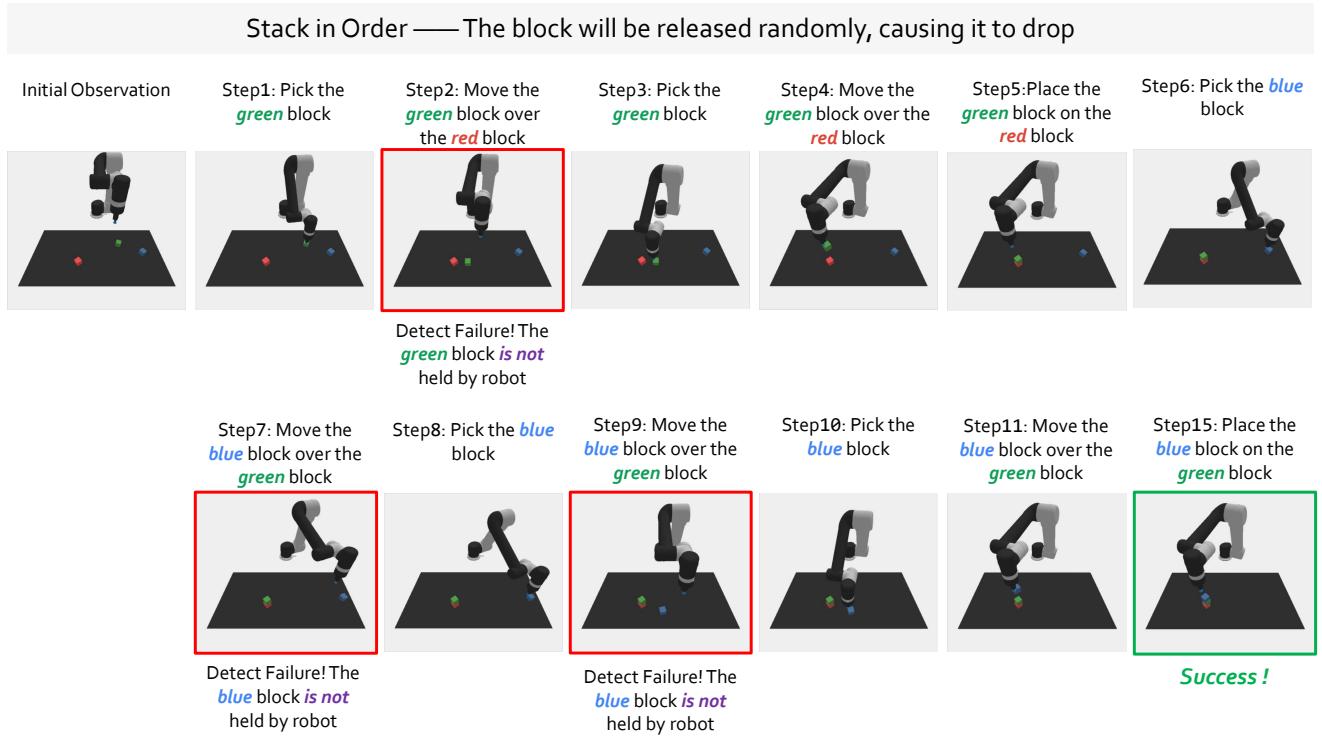


Figure 7. Demonstration of “Stack in Order”. We show how our framework performs failure detection and aids in recovery when, in the “Stack in Order” task, there is a probability p that blocks will fall due to being released by the robot’s suction cup at each step. Red boxes indicate the occurrence of failures, while green boxes signify successful task execution.

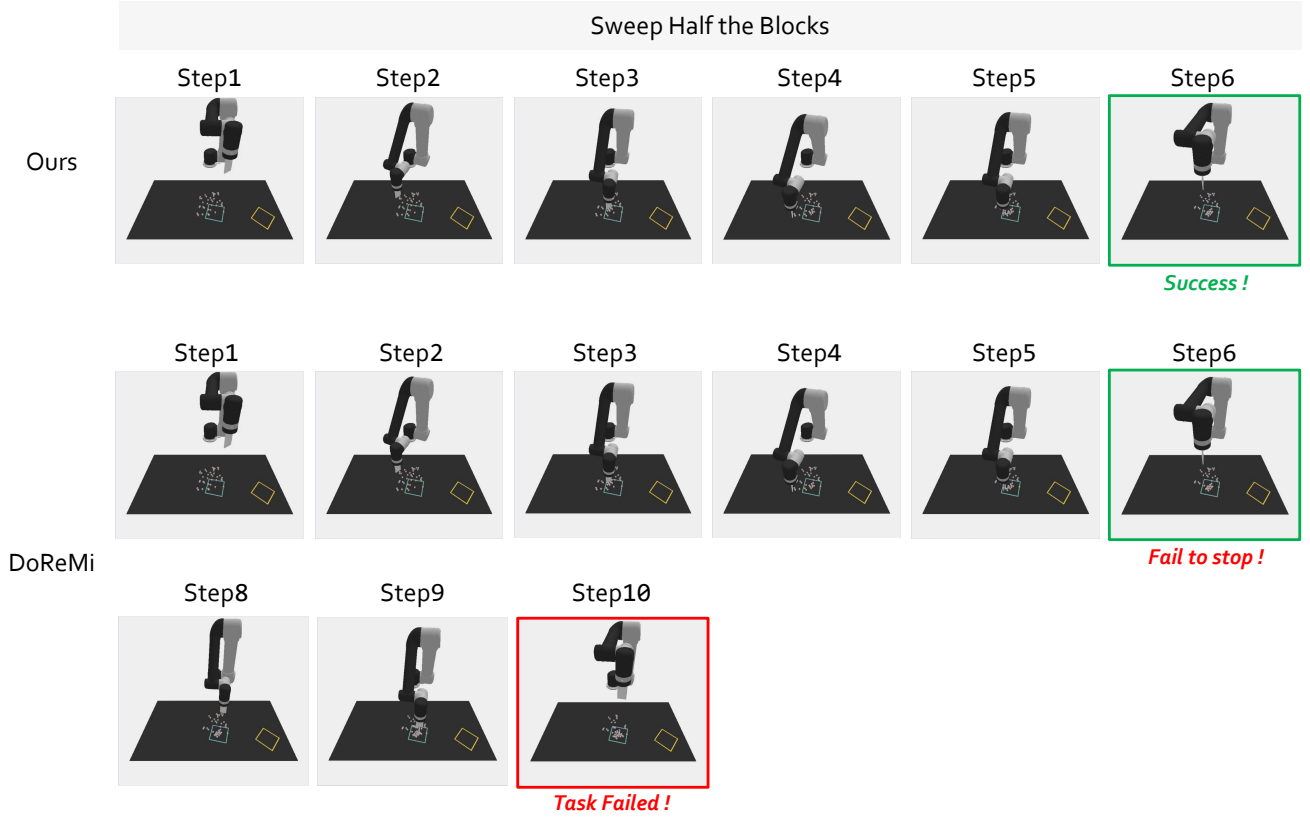


Figure 8. Demonstration of “Sweep Half the Blocks” and comparison to baseline. We show our framework can precisely count the blocks within a specified area and timely halts the policy execution to complete the task. In contrast, DoReMi [5] fails to stop the policy execution in time, leading to task failure. Red boxes indicate the occurrence of failures, while green boxes signify successful task execution.

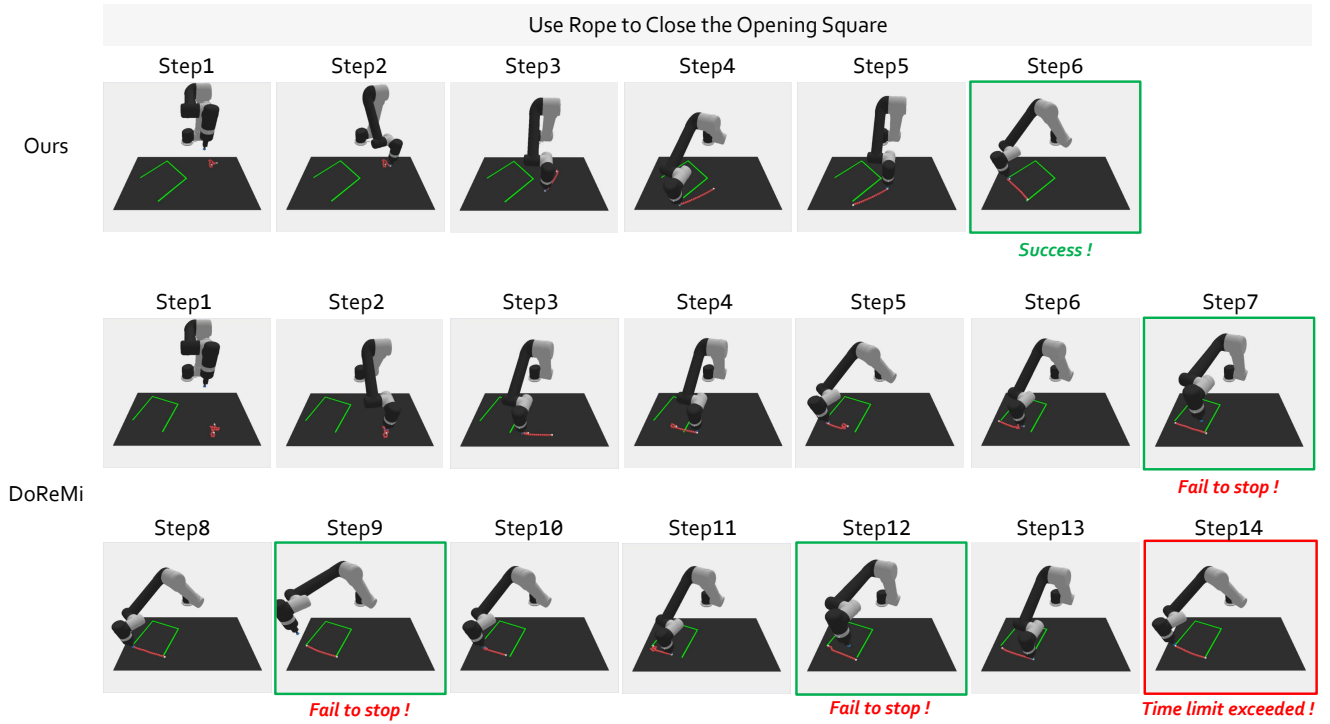


Figure 9. Demonstration of “Use Rope to Close the Opening Square” and comparison to baseline. We show that our framework effectively detects when the rope closes the opening square and promptly stops the policy execution to complete the task successfully. Conversely, DoReMi fails to halt the policy execution on time; although it eventually succeeds in closing the opening, the excessive execution time results in task failure. Red boxes indicate the occurrence of failures, while green boxes signify successful task execution.

Slot Pen Task — With Three Point-Level Disturbances

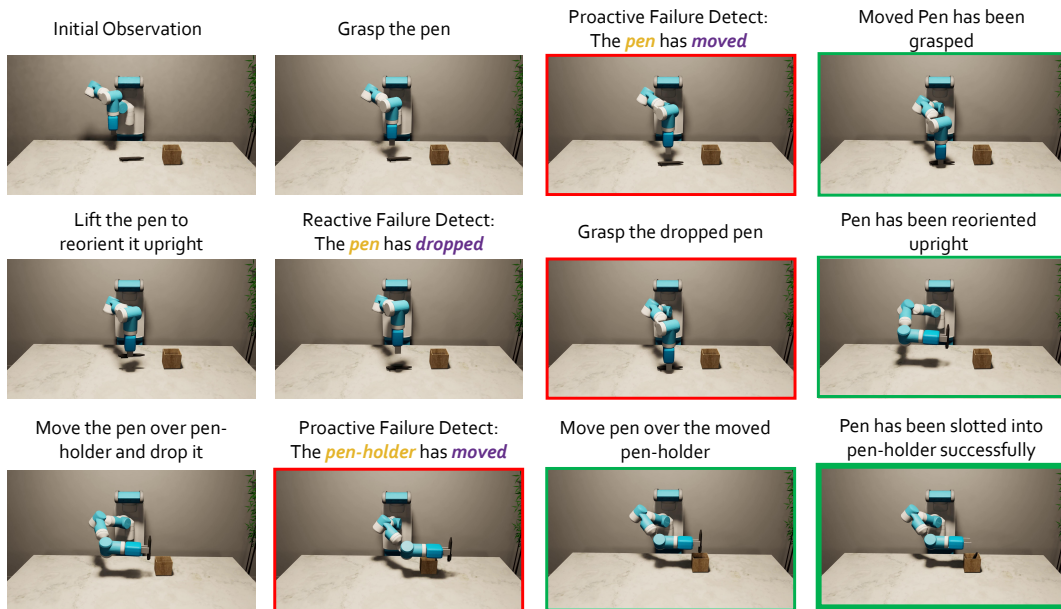


Figure 10. Demonstration of “Slot Pen”. We show how our framework detects failures and assists in recovery when facing point-level disturbances. Red boxes indicate the occurrence of failures, light green indicates the recovery with subgoal success and dark green boxes signify successful task execution.

Stow Book Task — With Three Line-Level Disturbances

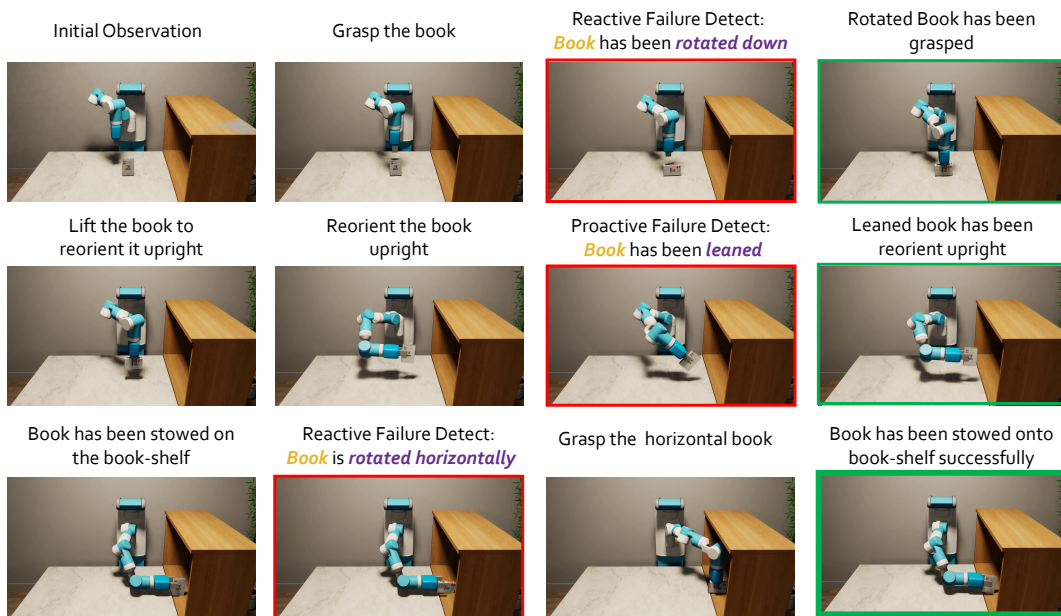


Figure 11. Demonstration of “Stow Book”. We show how our framework detects failures and assists in recovery when facing line-level disturbances. Red boxes indicate the occurrence of failures, light green indicates the recovery with subgoal success and dark green boxes signify successful task execution.

Pour Tea Task — With Three Surface-Level Disturbances

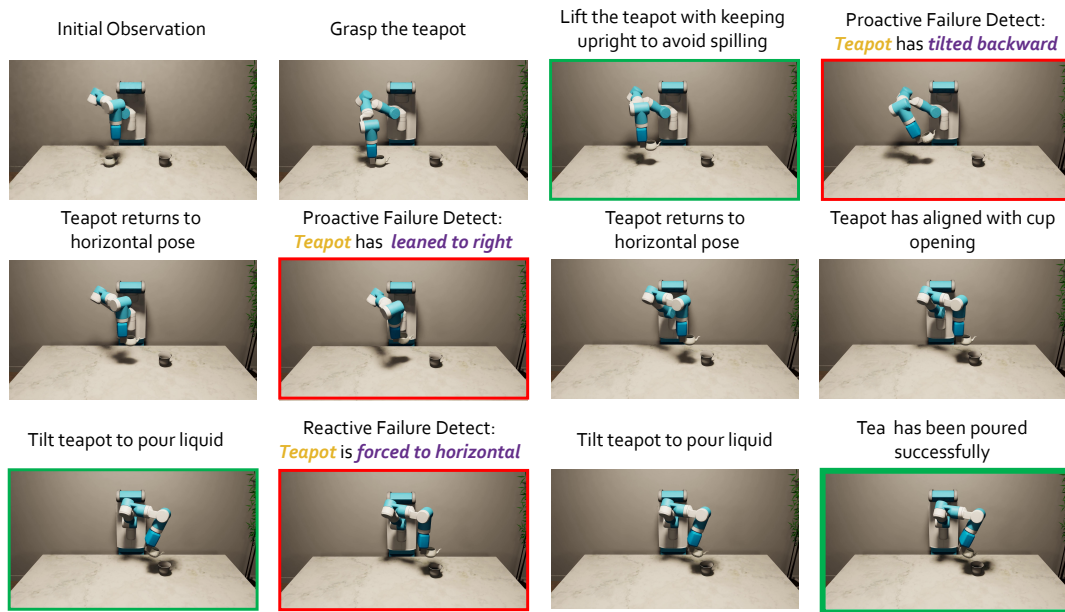


Figure 12. Demonstration of “Pour Tea”. We show how our framework detects failures and assists in recovery when facing surface-level disturbances. Red boxes indicate the occurrence of failures, light green indicates the recovery with subgoal success and dark green boxes signify successful task execution.

Reasoning Pick & Place Task: Clear all *objects* on table *except for animals*.

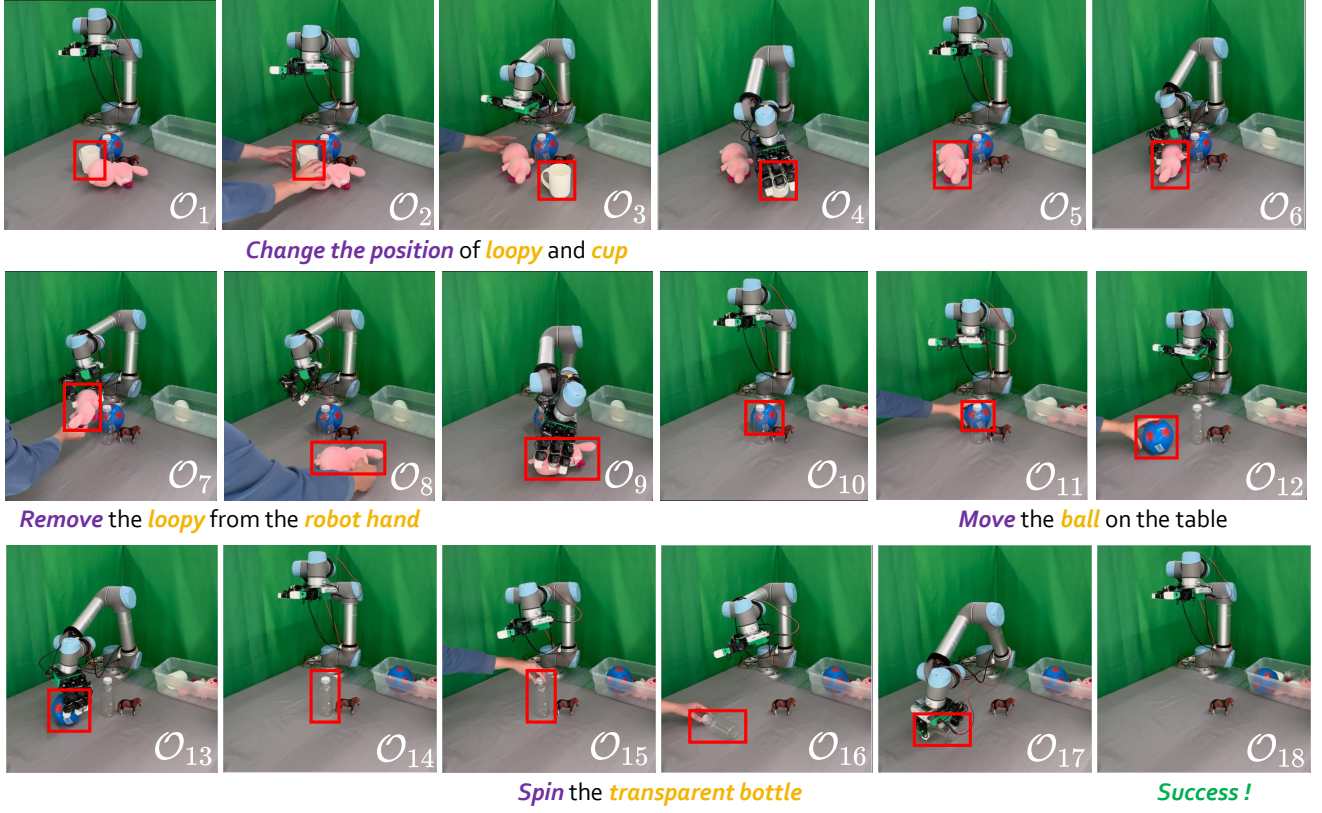


Figure 13. Demonstration of “Clear all objects on the table except for animals”. We show that our framework achieves both reactive failure detection (*e.g.*, detecting unexpected failures when humans remove objects from the robot’s grasp) and proactive failure detection (*e.g.*, identifying target object movement during grasping to prevent foreseeable failures). This effectively enhances the task success rate and reduces the execution time.
















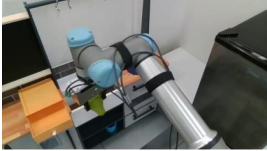


RoboFail Dataset (Out of Distribution)			
Task, Subgoal, Constraint	Observation	ConSeg	
		Instance-level	part-level
<u>Task</u> : Sauté carrot slice in a saucepan. <u>Subgoal</u> : Grasp the knife. <u>Constraint</u> : <i>Align</i> the <i>end effector</i> with the <i>handle</i> of the <i>knife</i> .			
<u>Task</u> : Sauté carrot slice in a saucepan. <u>Subgoal</u> : Slice carrot. <u>Constraint</u> : The <i>blade</i> should be <i>perpendicular</i> to the <i>carrot</i> .			
<u>Task</u> : Boil water in a pot. <u>Subgoal</u> : Pick up the pot. <u>Constraint</u> : <i>Align</i> the <i>end effector</i> with the <i>handle</i> on the <i>pot</i> .			
<u>Task</u> : Boil water in a pot. <u>Subgoal</u> : Place the pot on the stove. <u>Constraint</u> : The <i>pot</i> must be <i>10cm</i> <i>above</i> the <i>stove</i> .			
<u>Task</u> : Secure pear and knife. <u>Subgoal</u> : Open the drawer. <u>Constraint</u> : <i>Align</i> the <i>end effector</i> with the <i>handle</i> of the <i>drawer</i> .			
<u>Task</u> : Secure pear and knife. <u>Subgoal</u> : Put pear in drawer. <u>Constraint</u> : The <i>pears</i> <i>on</i> the <i>bottom</i> <i>surface</i> of the <i>drawer</i> .			

Figure 14. Visualization of constraint-aware segmentation for the RoboFail Dataset [14]. This dataset is not included in the training data.


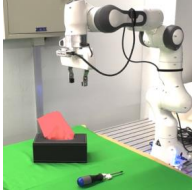
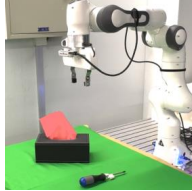












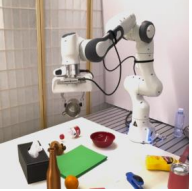

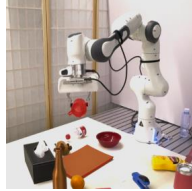
Open6DOR Benchmark (Out of Distribution)			
Task, Subgoal, Constraint	Observation	ConSeg	
		Instance-level	part-level
<u>Task</u> : Take a piece of paper and lay it over the screwdriver. <u>Subgoal</u> : Grasp a piece of paper. <u>Constraint</u> : <i>Align</i> the <i>end effector</i> with the <i>paper</i> .			
<u>Task</u> : Take a piece of paper and lay it over the screwdriver. <u>Subgoal</u> : Lay the paper over the screwdriver. <u>Constraint</u> : Move the <i>paper 10cm above</i> the <i>screwdriver</i> .			
<u>Task</u> : Put the ball into the upper drawer. <u>Subgoal</u> : Grasp the ball. <u>Constraint</u> : <i>Align</i> the <i>end effector</i> with the <i>ball</i> .			
<u>Task</u> : Put the ball into the upper drawer. <u>Subgoal</u> : Put the ball into the upper drawer. <u>Constraint</u> : The distance between the <i>ball</i> and the upper <i>drawer</i> should be <i>less than 10cm</i> .			
<u>Task</u> : Place the mug on top of the green paper. <u>Subgoal</u> : Grasp the mug. <u>Constraint</u> : <i>Align</i> the <i>end effector</i> with the <i>handle</i> of the <i>mug</i> .			
<u>Task</u> : Place the mug on top of the green paper. <u>Subgoal</u> : Move the mug on top of the green paper. <u>Constraint</u> : Move the <i>mug 10cm above</i> the <i>green paper</i> .			

Figure 15. Visualization of constraint-aware segmentation for the Open6DOF [3]. This dataset is not included in the training data.



















RT-1 Dataset (Out of Distribution)			
Task, Subgoal, Constraint	Observation	ConSeg	
		Instance-level	part-level
<u>Task</u> : Place red bull can in middle drawer. <u>Subgoal</u> : Grasp the red bull can. <u>Constraint</u> : <i>Align the end effector with the red bull can.</i>			
<u>Task</u> : Place red bull can in middle drawer. <u>Subgoal</u> : Place the red bull can into the middle drawer. <u>Constraint</u> : The red bull can must be 10cm above the drawer.			
<u>Task</u> : Place coke can upright. <u>Subgoal</u> : Grasp the coke can. <u>Constraint</u> : <i>Align the end effector with the coke can.</i>			
<u>Task</u> : Place coke can upright. <u>Subgoal</u> : Place the can upright. <u>Constraint</u> : Let the line formed by the top and bottom of the can be parallel to the z-axis.			
<u>Task</u> : Move sponge to green jalapeno chips. <u>Subgoal</u> : Grasp the sponge. <u>Constraint</u> : <i>Align the end effector with the sponge.</i>			
<u>Task</u> : Move sponge to green jalapeno chips. <u>Subgoal</u> : Move sponge to green jalapeno chips. <u>Constraint</u> : Make sure the distance between the sponge and the green jalapeno chips is less than 10cm.			

Figure 16. Visualization of constraint-aware segmentation for the RT-1 dataset [2]. This dataset is not included in the training data.




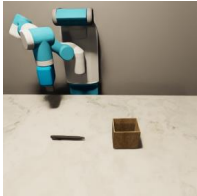
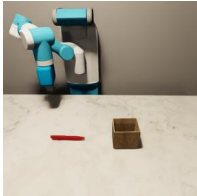
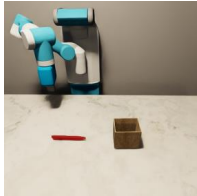

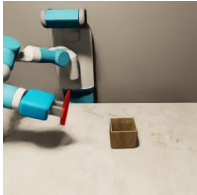
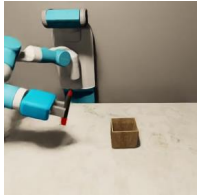









Omnigibson Dataset			
Task, Subgoal, Constraint	Observation	ConSeg	
		Instance-level	part-level
<u>Task</u> : Put the pen into the pen holder. <u>Subgoal</u> : Grasp the pen. <u>Constraint</u> : <i>Align</i> the <i>end effector</i> with the <i>pen</i> .			
<u>Task</u> : Put the pen into the pen holder. <u>Subgoal</u> : Grasp the pen. <u>Constraint</u> : <i>Align</i> the <i>end effector</i> with the <i>pen</i> .			
<u>Task</u> : Put the pen into the pen holder. <u>Subgoal</u> : Move the pen on the holder. <u>Constraint</u> : Keep the <i>pen upright</i> .			
<u>Task</u> : Put the pen into the pen holder. <u>Subgoal</u> : Move the pen on the holder. <u>Constraint</u> : Keep the <i>pen upright</i> .			
<u>Task</u> : Pour the tea from the teapot into the teacup. <u>Subgoal</u> : Grasp the teapot. <u>Constraint</u> : <i>Align</i> the <i>end effector</i> with the <i>handle of teapot</i> .			
<u>Task</u> : Pour the tea from the teapot into the teacup. <u>Subgoal</u> : Grasp the teapot. <u>Constraint</u> : <i>Align</i> the <i>end effector</i> with the <i>handle of teapot</i> .			

Figure 17. Visualization of constraint-aware segmentation for the Omnigibson simulator.

Imagine you are monitoring a robot performing manipulation tasks by writing monitor code in Python. The monitors provides you with two images of the environment: one captured from the head camera showing a first-person perspective, and the other from a recorder camera showing a third-person perspective. Additionally, you receive a brief text instruction describing the next subgoal of the task that the robot needs to execute. These images are overlaid with our proposed elements, which consist of 3D points. Each element is associated with its own indices, labeled on each point. For every given task, please follow these steps:

- For the given subgoal, specify two types of constraints to monitor: **"constraints during execution"** and **"constraints upon completion"**. Some examples are provided below:
 - Task: "Place the red block on top of the blue block":
 - Subgoal: "Grasp the red block":
 - Constraints upon completion: "The end-effector is aligned with the red block."
 - Constraints during execution: None.
 - Subgoal: "Move the red block over the blue block":
 - Constraints upon completion: "The red block is positioned higher than the blue block."
 - Constraints during execution: "The red block is held by the end-effector."
 - Subgoal: "Place the red block on the blue block":
 - Constraints upon completion: "The red block is on the blue block."
 - Constraints during execution: None.

****Note:****

- Each constraint function should take a dummy **end-effector position** and a set of **element positions** represented by 3D points, along with their past positions, as input. It should return two outputs:
 1. A **boolean value** indicating whether the spatial positions satisfy the required constraints.
 2. A **textual explanation** of the constraint being checked.
- ****Inputs to the constraints**:**
 - `end_effector`: A NumPy array of shape `(T, 3)` representing the positions of the end-effector over the past `T` time steps, including the current position.
 - `element_position`: A NumPy array of shape `(T, E, K, 3)` representing the positions of `E` elements, each with `K` points, tracked over the past `T` time steps.
 - `is_finished`: A boolean flag indicating whether this is a **"constraint upon completion"**.
- ****Indexing**:**
 - Elements marked on the image correspond to indices starting from `0`, matching the indices in the `element_position` array.
- ****Allowed Libraries**:**
 - You may only use **native Python functions** and **NumPy** functions.
- ****Function Return**:**
 - The **last return statement** in the function must:
 - Return a **boolean value** (`True` or `False`), and
 - Be at the **outermost level** of the function.
- ****Guidelines for Writing Constraints**:**
 - Avoid contradictory or overly detailed constraints.
 - Ensure the constraints are logically consistent and suitable for the specific task.

****Structure your output in a single python code block as follows:****

```

python
def constraint_monitor(end_effector, element_position, is_finished):
    """Put your explanation here."""
    ...
    return True, reason

```

Query

Query Task: "{task_instruction}"

Query Current Subgoal: "{next_textual_subgoal}"

Query Image:

Figure 18. Prompt of monitor code generation. We use this prompt, combined with additional task instructions, the current subgoal, and images from two perspectives, to enable an off-the-shelf VLM, *i.e.*, GPT-4o [1], to generate Python code for monitoring.

References

- [1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv*, 2023. 3, 5, 10, 23
- [2] Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Joseph Dabis, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Jasmine Hsu, et al. Rt-1: Robotics transformer for real-world control at scale. *arXiv*, 2022. 10, 21
- [3] Yufei Ding, Haoran Geng, Chaoyi Xu, Xiaomeng Fang, Jiazhao Zhang, Songlin Wei, Qiyu Dai, Zhizheng Zhang, and He Wang. Open6dor: Benchmarking open-instruction 6-dof object rearrangement and a vlm-based approach. *IROS*, 2024. 10, 20
- [4] Ankit Goyal, Valts Blukis, Jie Xu, Yijie Guo, Yu-Wei Chao, and Dieter Fox. Rvt2: Learning precise manipulation from few demonstrations. *RSS*, 2024. 8
- [5] Yanjiang Guo, Yen-Jen Wang, Lihan Zha, Zheyuan Jiang, and Jianyu Chen. Doremi: Grounding language model by detecting and recovering from plan-execution misalignment. *arXiv*, 2023. 5, 6, 7, 8, 9, 13
- [6] Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv*, 2022. 5, 6, 7
- [7] Wenlong Huang, Chen Wang, Yunzhu Li, Ruohan Zhang, and Li Fei-Fei. Rekep: Spatio-temporal reasoning of relational keypoint constraints for robotic manipulation. *arXiv*, 2024. 1, 4, 5
- [8] Stephen James, Zicong Ma, David Rovick Arrojo, and Andrew J. Davison. Rlbench: The robot learning benchmark & learning environment. *RAL*, 2020. 3, 4
- [9] Xin Lai, Zhuotao Tian, Yukang Chen, Yanwei Li, Yuhui Yuan, Shu Liu, and Jiaya Jia. Lisa: Reasoning segmentation via large language model. *CVPR*, 2024. 2
- [10] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *NeurIPS*, 2020. 2
- [11] Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-Martín, Chen Wang, Gabriel Levine, Michael Lingelbach, Jiankai Sun, Mona Anvari, Minjune Hwang, Manasi Sharma, Arman Aydin, Dhruva Bansal, Samuel Hunter, Kyu-Young Kim, Alan Lou, Caleb R Matthews, Ivan Villa-Renteria, Jerry Huayang Tang, Claire Tang, Fei Xia, Silvio Savarese, Hyowon Gweon, Karen Liu, Jiajun Wu, and Li Fei-Fei. Behavior-1k: A benchmark for embodied ai with 1,000 everyday activities and realistic simulation. *CoRL*, 2022. 3, 4
- [12] Feng Li, Hao Zhang, Peize Sun, Xueyan Zou, Shilong Liu, Jianwei Yang, Chunyuan Li, Lei Zhang, and Jianfeng Gao. Semantic-sam: Segment and recognize anything at any granularity. *arXiv*, 2023. 2, 3
- [13] Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. *ICML*, 2023. 5
- [14] Zeyi Liu, Arpit Bahety, and Shuran Song. Reflect: Summarizing robot experiences for failure explanation and correction. *CoRL*, 2023. 10, 19
- [15] Pasquale Minervini et al. awesome-hallucination-detection. <https://github.com/EdinburghNLP/awesome-hallucination-detection>, 2024. 2
- [16] 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*, 2023. 4
- [17] Tianhe Ren, Shilong Liu, Ailing Zeng, Jing Lin, Kun-chang Li, He Cao, Jiayu Chen, Xinyu Huang, Yukang Chen, Feng Yan, Zhaoyang Zeng, Hao Zhang, Feng Li, Jie Yang, Hongyang Li, Qing Jiang, and Lei Zhang. Grounded sam: Assembling open-world models for diverse visual tasks. *arXiv*, 2024. 2, 3
- [18] Kenneth Shaw, Ananye Agarwal, and Deepak Pathak. Leap hand: Low-cost, efficient, and anthropomorphic hand for robot learning. *arXiv*, 2023. 4
- [19] Mohit Shridhar, Lucas Manuelli, and Dieter Fox. Cliport: What and where pathways for robotic manipulation. *CoRL*, 2021. 3, 4, 5
- [20] Homer Rich Walke, Kevin Black, Tony Z Zhao, Quan Vuong, Chongyi Zheng, Philippe Hansen-Estruch, Andre Wang He, Vivek Myers, Moo Jin Kim, Max Du, et al. Bridgedata v2: A dataset for robot learning at scale. *CoRL*, 2023. 2
- [21] Jialiang Zhang, Haoran Liu, Danshi Li, XinQiang Yu, Haoran Geng, Yufei Ding, Jiayi Chen, and He Wang. Dexgraspnet 2.0: Learning generative dexterous grasping in large-scale synthetic cluttered scenes. *CoRL*, 2024. 5, 9
- [22] Xinyu Zhang, Yuhuan Liu, Haonan Chang, Liam Schramm, and Abdeslam Boularias. Autoregressive action sequence learning for robotic manipulation. *arXiv*, 2024. 4, 8