# Supplementary Materials for CodeDiffuser

Guang Yin[3*] Yitong Li[4*] Yixuan Wang[1*] Dale McConachie[2] Paarth Shah[2]
Kunimatsu Hashimoto[2] Huan Zhang[3] Katherine Liu[2] Yunzhu Li[1]
[1]Columbia University [2]Toyota Research Institute [3]University of Illinois Urbana-Champaign [4]Tsinghua University
https://robopil.github.io/code-diffuser/

## I. Method

### A. Clustering and Segmentation

For clustering, we first leverage DINO features to measure the similarity between the source feature and the features of point cloud. A predefined threshold is used to filter attention points based on the cosine similarity value. To ensure uniform point density across different parts, we apply Furthest Point Sampling (FPS) to downsample the point cloud to 8,000 points. Next, we employ a density-based clustering method on the selected attention points. Specifically, the clustering algorithm groups points into clusters based on a predefined L2-norm distance threshold—0.1 for packing a battery and 0.15 for hanging a mug.

We found DBSCAN performs robustly in our experiments, as evidenced by the 3D attention benchmark (Table I) and the failure breakdown analysis (Main Figure 9). Additionally, due to modular design, it can be easily replaced with a more robust alternative as needed.

For methods utilizing 2D attention, we project the 3D point cloud with attention onto 2D images, generating a binary mask where attention regions are marked as 1 and non-attention regions as 0. Due to the sparse and fragmented nature of the initial projection, we apply dilation to refine the mask, ensuring a more cohesive and continuous attention region.

### B. Spatial Selection through CodeGen

We utilize code as an interface to detect the target object based on spatial instructions, such as "the rightmost mug." Given a language instruction, the LLM generates the corresponding code and leverages NumPy APIs to analyze the spatial relationships between attention clusters, ultimately identifying the specified target.

For example, if the language instruction is "*Pick a battery into the slot on the middle columns.*", the generated code would be:

```
# compose API summary
# INSTRUCTION
# Pick a battery into the slot on the middle
    columns.

# RESPONSE
# Determine target object, battery
battery = self.detect("battery")[0]["pointer"]

# Determine target object, the slot on the
    middle column

# Detect slots
slot_list = self.detect("slot")
slot_centroid_list = []
for obj in slot_list:
   slot_centroid_list.append(np.mean(obj["pcd"],
       axis=0)[np.newaxis, :])
slot_centroid_numpy =
    np.concatenate(slot_centroid_list, axis=0)

# Extract x coordinate
slot_x = slot_centroid_numpy[:, 0] # use x to
    determine right or left

# The slot with the max x value and the min x
    value
slot_max_x = np.max(slot_x)
slot_min_x = np.min(slot_x)

# Determine the x coordinate range for the
    middle column
middle_x_threshold = (slot_max_x +
    slot_min_x) / 2

# Find the index of the slot closest to the
    middle_x_threshold
tgt_slot_idx = np.argmin(np.abs(slot_x -
    middle_x_threshold))

# Extract the points of the target slot
tgt_slot = slot_list[tgt_slot_idx]["pointer"]

output_var = [battery, tgt_slot]
# -------------
# detection API summary
# PROMPT
# ['battery', 'slot']
# battery

# RESPONSE
battery_list = self.get_obj('battery')
output_var = battery_list
# -------------
# detection API summary
# PROMPT
# ['battery', 'slot']
# slot

# RESPONSE
slot_list = self.get_obj('slot')
output_var = slot_list
```

*Denotes equal contribution.

### C. Visual Selection through VLM

We employ a Vision-Language Model (VLM) to detect the target object based on visual instructions such as "the orange book." When visual detection is required, the generated code can invoke a predefined API, *find instance in category*, which annotates instances in RGB images with corresponding labels. The VLM then interprets the language instruction and provides the index of the matching label as shown in Main Figure 4. For example, if the language instruction is "*I want to use the blue mug to drink some water. Put away the other one on a branch.*", the generated code would be:

```
# compose API summary
# INSTRUCTION
# I want to use the blue mug to drink some
    water. Put away the other one on a branch.

# RESPONSE
# determine target object, the other mug (not
    blue)
blue_mug = self.detect("blue mug")[0]
mug_list = self.detect("mug")
tgt_mug_list = []
for mug in mug_list:
  if mug != blue_mug:
    tgt_mug_list.append(mug)
tgt_mug = tgt_mug_list[0]["pointer"]

# determine target object, branch
branch = self.detect("branch")[0]["pointer"]

output_var = [tgt_mug, branch]
# -------------
# detection API summary
# PROMPT
# ['mug', 'branch']
# blue mug

# RESPONSE
mug_list = self.get_obj('mug')
tgt_idx =
    self.find_instance_in_category(instance =
    'blue mug', category = 'mug')
tgt_mug = []

for i in tgt_idx:
   tgt_mug.append(mug_list[i])
output_var = tgt_mug
# -------------
# find_instance_in_category API summary
# PROMPT
# blue mug


# RESPONSE
# The object labeled with index 1 is a blue
    mug.
1
# -------------
# detection API summary
# PROMPT
# ['mug', 'branch']
# mug

# RESPONSE
mug_list = self.get_obj('mug')
output_var = mug_list
# -------------
# detection API summary
# PROMPT
# ['mug', 'branch']
# branch

# RESPONSE
branch_list = self.get_obj('branch')
output_var = branch_list
```

## II. Additional 3D Attention Evaluation

In addition, we build a benchmark in the simulation to quantitatively evaluate the language-to-3D attention pipeline, which can automatically generate scenes, prompts, and corresponding ground truth 3D attention maps. We measure the distance between ground truth 3D attention maps and generated 3D attention maps, and a test is considered successful if they are close enough. Specifically, we use Chamfer Distance to evaluate alignment between GT and generated 3D attention. A threshold of 0.005 is used to decide spatial proximity. In this experiment, we compare with the following two baselines:

- G-SAM2: We directly prompt Grounded-SAM2 to segment out the object to attend to.
- G-SAM2+GPT-4o: We first segment out objects for each object category using Grounded-SAM2 and then ask GPT4o to select the object instance.

| Scene | Ours | G-SAM2 | G-SAM2+GPT-4o |
|---|---|---|---|
| Hang Mug | **97/100** | 0/100 | 0/100 |
| Pack Battery | **94/100** | 0/100 | 0/100 |
| Total | **191/200** | 0/100 | 0/100 |

**Table I: Additonal Attention Quantitative Evaluation**. We quantitatively evaluate the pipeline from language instructions to 3D attention maps in simulation. Our results demonstrate that our pipeline effectively attends to task-relevant areas, whereas baseline methods fail due to their limited fine-grained visual-semantic reasoning and spatial understanding capabilities.

Table I summarizes our quantitative evaluation of the pipeline from language to attention. We find that our method outperforms the baselines, as it leverages the powerful visualsemantic understanding capabilities of VLMs and benefits from explicit spatial relation reasoning using 3D representations. In contrast, Grounded-SAM2 lacks the capability for fine-grained semantic reasoning, such as detecting branches and slot locations, resulting in zero success. Qualitative comparisons between our method and G-SAM2 are shown in Figure 1.

## III. Simulation Experiments

### A. Training Data Generation and Labeling

In this project, we conduct experiments in Sapien using two scenarios: packing batteries and hanging mugs. We use scripted policies to generate demonstrations. During data generation, after defining the task scenes, we randomly select the

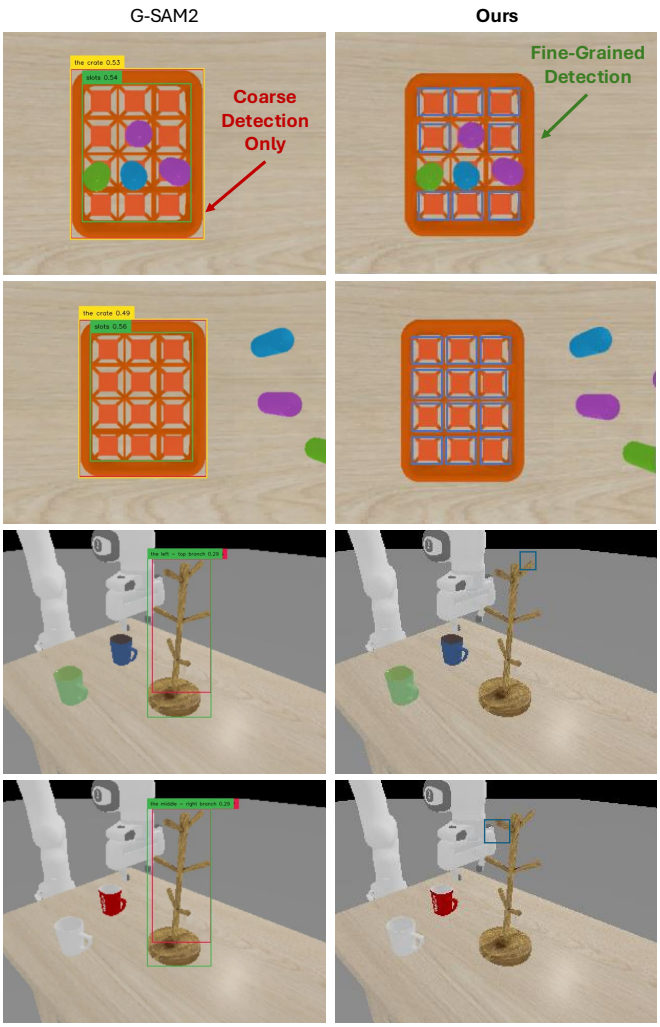



Fig. 1: **Comparisons with Baselines on 3D Attention.** Compared to G-SAM2, our method allows for more fine-grained visual understanding, such as detecting slots and branches.

target objects (e.g., a battery and slot, or a mug and branch) and generate the corresponding manipulation trajectories. At the same time, we log the ground truth attention locations for future labeling. This includes details such as which object to manipulate and where to place it.

For the labeling process, we use DINO features to query the attention point cloud, cluster the points into distinct parts, and identify the target part based on the ground truth location information. For methods involving 2D attention, we generate a point cloud with attention points and project it onto images as segmentation masks.

### *B. Evaluation Benchmark with Language Instruction*

For system evaluation, the main objective is to assess whether the policy can accurately perform tasks based on language instructions. The system's inputs for evaluation include visual observations and language prompts. Specifically, the visual observations are RGBD images captured from five different viewpoints, while the prompts are generated based on the task scene. These prompts are then validated using a rule-based approach to ensure their feasibility and coherence. Additionally, we define success criteria derived from the given prompts and task scenes. To be considered successful, the system must meet these criteria within a predefined time limit (e.g., 240 frames).

The scenario generation process can be broken down into several parts.

- **Descriptive Components Selection and Instruction Generation**: Descriptive components (e.g., "blue mug," "furthest branch" for hanging a mug, "slot in the first row," "left battery" for packing a battery) are randomly sampled from a predefined library. Once the descriptive components are selected, we randomly choose a template from an instruction library and combine it to form a complete language instruction. A more detailed explanation can be found in the next subsection.
- **Task Scene Randomization and Matching**: Using the selected descriptive components, we generate task scenes in Sapiens (e.g., initial position, pose, number, and color of batteries). It is crucial that the correlation between the physical scenes and the prompts is accurate. For example, if the prompt is "hang a blue mug," there should be at least one blue mug on the table. Similarly, if the prompt is "put the battery into the slot in the first row," at least one slot in the first row must be available. We use a predefined check function, along with the descriptive components, to ensure that the generated scene matches the instructions. If the scene does not meet the requirements, a new one is generated and checked again. This process continues until the generated scene satisfies all conditions.
- **Success Criteria Specification**: We can also obtain the indices of objects using a predefined extraction function, which works in conjunction with the descriptive components. This function is designed to identify target objects. For example, the *blue_mug_extract* function retrieves the indices of all blue mugs on the table. By using this function, we can determine the positions of the target objects referenced in the instructions and get access to their poses through API defined in Sapiens.

In this way, we can automatically evaluate whether the policy executes successfully based on our language instructions, which are also generated without manual effort. The detailed criteria are listed below:

- **Pack Batteries**: The reward function determines success based on three criteria: the object's horizontal distance from the target must be less than 0.03 units, the object must be aligned with the global Z-axis (i.e., nearly upright) with a product value greater than 0.99, and the object's height must be less than 0.009 units. If all these conditions are met for any object-target pair, the function returns a reward of 1.0, indicating success; otherwise, it returns 0.0, indicating failure.
- **Hang Mugs**: The reward function evaluates success based on the position of an object relative to a target. Specifically,

the object's X-coordinate must be within 0.05 units of the target branch's X-coordinate, and the object's Z-coordinate must be between the target branch's Z-coordinate and a lower offset of 0.1 units below it. If both conditions are met for any object-target pair, the function returns a reward of 1.0, indicating success. If no pair satisfies these criteria, the function returns 0.0, indicating failure.

The object-target pairs are determined based on the language instructions, specifying which pairs meet the requirements of the given instruction. The process for this determination is explained in **Success Criteria Specification**.

### C. Details of Language Instruction Generation

The first step in generating language instructions is sampling descriptive components. These are object-centric descriptions, such as "blue mug", " front-most battery", "furthest branch". We consider both spatial information (e.g., "furthest", "right") and visual information (e.g., "red", "white") when selecting these components. The libraries for hanging mugs and packing batteries are listed below:

- **Hang Mugs**: *left-most mug*, *right-most mug*, *white mug*, *red mug*, *blue mug*, *green mug*, *furthest branch*, *right-topmost branch*, *left-topmost branch* and *right-middle branch*.
- **Pack Batteries**: *left-most battery*, *right-most battery*, *front-most battery*, *back-most battery*, *furthest slot*, *nearest slot*, *slot on the front-most row*, *slot on the middle row*, *slot on the back-most row*, *slot on left column*, *slot on the middle columns* and *slot on the right column*.

Once the descriptive components are determined, we randomly select a template from an instruction library and combine it to form a complete language instruction. Below is the full list of templates for hanging mugs:

- **No Slackness**:
  *"Hang the {mug} on the {branch}."*
  *"I want to use the {other_mug} to drink some water. Put away the other one on the {branch}."*
  *"I want to use the {other_mug} to drink some water. Put away the other mug on the {branch}."*
  *"I will use the {other_mug} to drink some water. Hang the {mug} on the {branch}."*
  *"Hang the {mug} on the {other_branch}. Sorry, the {branch}."*
  *"There are two mugs. Keep {other_mug} on the table, put the other one on {branch}."*
  *"I will not use this {mug} now. Hang it on {branch}"*
  *"Put Bob's mug, the {mug}, on the {branch}."*
- **Mug Slackness**:
  *"Hang a mug on the {branch}."*
  *"Put away a mug on the {branch}."*
  *"Hang a mug on the {other_branch}. Sorry, the {branch}."*
- **Branch Slackness**:
  *"Hang the {mug} on a branch."*
  *"I want to use the {other_mug} to drink some water. Put away the other one on a branch."*
  *"I want to use the {other_mug} to drink some water. Put away the other mug on a branch."*
  *"I will use the {other_mug} to drink some water. Hang the {mug} on a branch."*
  *"Hang the {other_mug} on a branch. Sorry, the {mug}."*
  *"There are two mugs. Keep {other_mug} on the table, put the other one on a branch."*
  *"I will not use this {mug} now. Hang it on a branch"*
  *"Put Bob's mug, the {mug}, on the a branch."*
- **Both Slackness**:
  *"Hang a mug on a branch."*
  *"There are two mugs. Keep one on the table, put the other one on a branch."*

This is the full list of template for packing batteries:

- **No Slackness**:
  *"Pick the {battery} outside the crate into the {slot}."*
  *"I need to use the {other_battery}. Put away the {battery} outside the box in the {slot}."*
  *"The desk is too messy. Put away the {battery} outside the box into the {slot}."*
  *"I want to put the {battery} outside the crate into the {other_slot}, oh sorry, the {slot}."*
  *"You should make the table more tidy, Just start from putting the {battery} outside the crate into the {slot}."*
- **Battery Slackness**:
  *"Pick a battery outside the crate into the {slot}."*
  *"The desk is too messy. Put away a battery outside the crate into the {slot}."*
  *"I want to put a battery outside the crate into the {other_slot}, oh sorry, the {slot}."*
  *"I want to put a battery outside the crate into the {slot}."*
- **Slot Slackness**:
  *"Pick the {battery} outside the crate into a slot."*
  *"I need to use the {other_battery}. Put away the {battery} outside the crate in a slot."*
  *"The desk is too messy. Put away the {battery} outside the crate into a slot."*
  *"I want to put the {other_battery} into a slot, oh sorry, the {battery} outside the crate."*
  *"You should make the table more tidy, Just start from putting the {battery} outside the crate into a slot."*
- **Both Slackness**:
  *"Put a battery outside the crate on a slot."*
  *"There are some batteries outside the crate. Put one into a slot."*

### D. Baseline Details

- **Ours with 2D Attention**: Instead of mapping the multi-view RGBD observation into 3D space, this baseline uses a 2D attention mechanism to segment objects of interest. The masked observation is then input into visuomotor policy. Specifically, we first obtain 3d point cloud from images and ground attention on it through our proposed pipeline. Then the 3d attention is projected onto 2d images in the format of segmentation mask, where attention region is 1 and none-attention region is 0. We use attention masks to multiply corresponding RGB images. We train diffusion policy based on this observation. The total training epoch number is 360

and the learning rate scheduler is cosine with 600 epochs maximum limitation. The training batchsize is 64.

- **Ours without Residual Connection**: In our policy, we include a residual connection in PointNet++ for visual feature extraction. This baseline ablates the residual connection to evaluate its contribution to performance. We train diffusion policy based on typical 3d point cloud with 3d attention as ours. The total training epoch number is 360 and the learning rate scheduler is cosine with 600 epochs maximum limitation. The training batchsize is 64.
- **Lang-DP (RGB)**: This baseline extends DP (RGB) by conditioning the policy on language using a frozen CLIP encoder. The extracted language features are concatenated with visual features to condition the diffusion policy. The initial embedding dimension is 1536, which is reduced into 128 after processed through a linear layer. We train diffusion policy based on RGB images and textual embedding. The total training epoch number is 360 and the learning rate scheduler is cosine with 600 epochs maximum limitation. The training batchsize is 64.
- **Lang-DP (PCD)**: Similar to Lang-DP (RGB), this baseline adds language conditioning to DP (PCD) using a CLIP-based language encoder. The initial embedding dimension is 1536, which is reduced into 128 after processed through a linear layer. We train diffusion policy based on 3d point cloud without attention, but with RGB channel. RGB information should be accessible for language reasoning, especially for visual reasoning such as "blue mug". The total training epoch number is 360 and the learning rate scheduler is cosine with 600 epochs maximum limitation. The training batchsize is 64.
- **Lang-ACT**: This baseline augments the vanilla ACT framework with language features, similar to Lang-DP. For language embedding, the initial embedding dimension is 1536, which is reduced into 128 after processed through a linear layer. We regard it as an additional token, which are processed similarly as RGB image tokens. The total training epoch number is 540, where we find ACT will show better performance with a bigger epochs number. For fair comparison, we take the advantages of all methods. The learning rate scheduler is constant. The training batchsize is 64.
- **Lang-ACT with 3D Attention**: Unlike the vanilla ACT, which uses multi-view RGB observations, this baseline inputs 3D attention maps into Lang-ACT to assess whether the attention module consistently improves the performance of base imitation learning algorithms. The total training epoch number is 540, where we find ACT will show better performance with a bigger epochs number and the learning rate scheduler is constant. The training batchsize is 64.

### *E. Single vs. Recursive VLM Calls*

While we can use a single VLM call, we choose the recursive approach to leverage the modular design for problem decomposition. To ablate this design choice, we evaluated the generated 3D attention map by single VLM call, following the experiment in Section IV.C and Table I. We observed that the success rate drops from 94% and 97% to 90% and 91%. We will add this to the revised paper. Second, recursive calls are also standard in existing works [1].

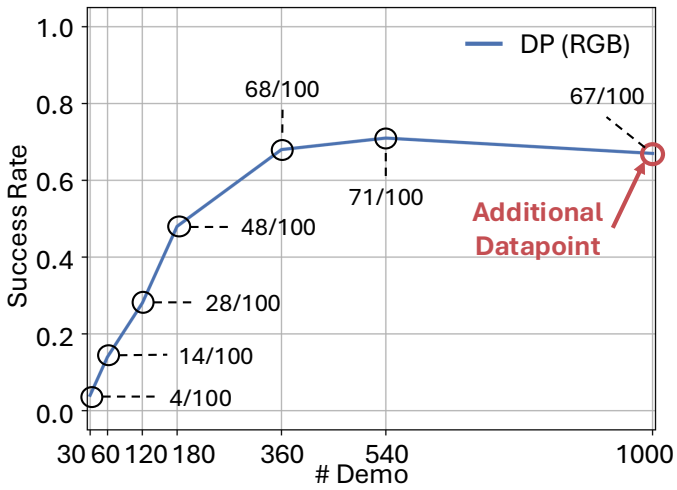



Fig. 2: **Scaling Plot.** We increase the number of demonstrations to 1000 and observe the performance plateaus.

### *F. Scaling Plot*

To further demonstrate that the scaling plot (Fig. 4) for DP (RGB) is saturated, we increased the number of demonstrations to 1,000 and performed the same evaluation. The performance plateaued, confirming saturation (Fig. 2).

## IV. Real-World Experiments

### *A. Training Data Generation*

For the real-world experiment, datasets are collected, and policies are evaluated on Aloha [2] for three different tasks: packing a battery, hanging a mug, and stowing a book. The same dataset generation logic used in training applies to real experiments, but with several key differences. In the real experiment setup, there are five cameras: four positioned at each top corner and one at the center on top.

For data generation, the first step is to sample the required number of initial configurations for training demonstrations. According to predefined randomness—for example, the mug's position follows a uniform distribution within a fixed area—a program generates two output files: (1) a visualization image that helps the operator reset the scene before and after each demonstration as shown in Figure 3, and (2) a YAML script that describes the scene in a structured format, which is used for instruction generation. The same program is used to generate initial configurations for evaluation. During instruction generation, the instruction generator produces instructions based on the YAML scene description file and an instruction template file.

For labeling, we use an intuitive GUI labeling tool that allows the operator to quickly select objects with attention. The labeling tool for the packing battery and hanging mug tasks projects the point cloud of the entire scene in a top-down view.

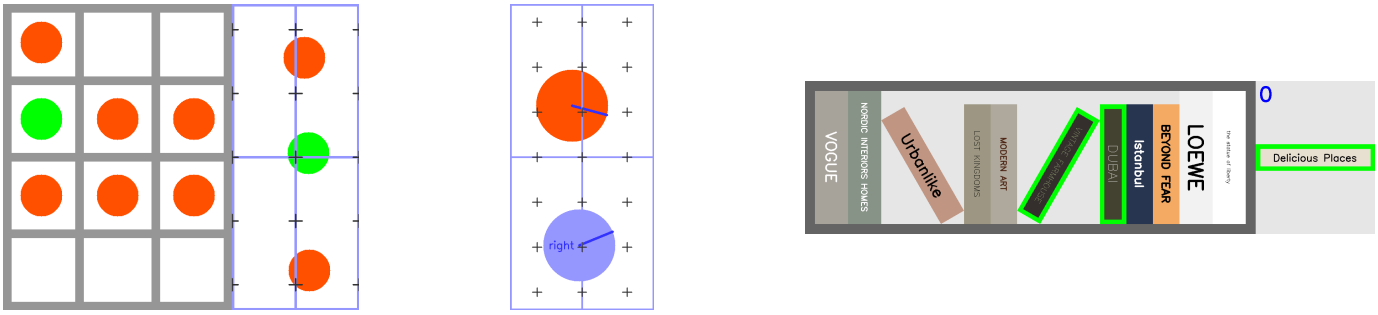



Fig. 3: **Initial Configuration Visualization.** Initial configuration examples are generated by a program for real experiments. The leftmost visualization image corresponds to the packing a battery task, where orange circles represent batteries, and green circles indicate the target battery and target slot. The middle image represents the hanging a mug task. Circles of different colors correspond to mugs of the same color, while blue lines within the circles indicate the handle angles of each mug. The label "Right" signifies that the specified mug should be picked up and placed on the right branch. The cross markers on the pad image have exact corresponding locations to the cross markers on the real pad. This facilitates the operator in quickly identifying the relative positions of objects. Finally, the rightmost image corresponds to the stowing a book task, where books with green boundaries indicate the target book to be picked and the designated slot for placement.

Fig. 4: **Dataset labeling interface.** The left interface corresponds to the packing a battery task, while the middle interface is for the hanging a mug task. In both tasks, the target objects are highlighted in orange. The right interface represents the stowing a book task, where the target objects are highlighted with green boundaries. The operator only needs to move the cursor close to the target object and click to select it. Clicking the selected object again will deselect it.

However, this approach is not suitable for the stowing book task because the books are placed too close together, making the top-down view cluttered. Instead, we project the point cloud from the perspective of one of the cameras, allowing the operator to directly select objects within the camera's view. The labeling interfaces for each task are shown in Figure 4

### *B. Task Design*

We carefully design each task to be complex enough to evaluate the effectiveness of our pipeline while ensuring an appropriate level of randomness, allowing the low-level policy to converge with a limited number of human demonstrations. The randomness visualization is shown in Figure 5.

*a) Packing a Battery:* In this task, a crate with 12 slots is arranged in 3 rows and 4 columns. In front of the crate, a random number of batteries, ranging from 1 to 3, are randomly placed on an orange pad. Additionally, some batteries may already be positioned inside the crate. The goal of the task is to place a battery on the pad into a slot according to a given linguistic instruction.

We define four types of instructions based on the level of specificity: **no slackness, battery slackness, slot slackness**, and **both slackness**.

- **No Slackness:** Both the battery and the slot are explicitly specified. For example, *"Put the leftmost battery into the slot in the back row."* The task is only considered successful if the specified battery is placed into the designated slot.
- **Battery Slackness:** Only the slot is explicitly specified. For example, *"Place a battery into the slot in the right column."* Any battery can be selected, but it must be placed in the designated slot for the task to be considered successful.
- **Slot Slackness:** Only the battery is explicitly specified. For example, *"I need to use the leftmost battery. Put away the middle battery in a slot."* The specified battery must be picked up, but it can be placed in any available slot for the task to be considered successful.
- **Both Slackness:** Neither the battery nor the slot is explicitly specified. For example, *"There are some batteries. Put one into a slot."* In this case, placing any battery into any slot is considered successful.

The instruction types are randomly selected, and the instructions are generated using the instruction generator mentioned earlier.

*b) Hanging a Mug:* The experiment scene consists of a mug tree with four branches (the rear branch is not used) and two mugs randomly placed on a white pad. The angle of each mug handle ranges from $-30°$ to $30°$. Each mug has a color randomly selected from red, blue, or green, ensuring that no two mugs on the pad share the same color. The goal of this task is to pick up a mug and place it on a branch according to a given linguistic instruction.

We define four types of instructions based on the level of specificity: **no slackness, mug slackness, branch slackness**, and **both slackness**.

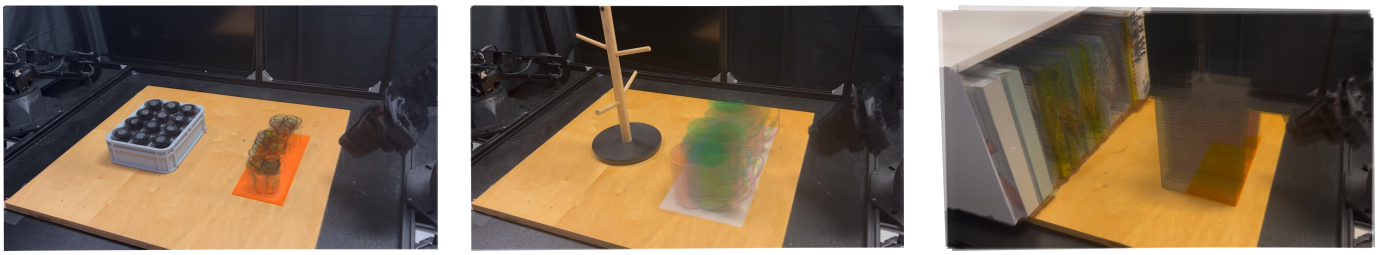


Fig. 5: **Initial Configuration Overlay.** The randomness for each task is visualized as an overlay of initial configurations.

- **No Slackness:** Both the mug and the branch are explicitly specified. For example, *"Put Bob's mug, the red mug, on the top branch."* The task is only considered successful if the specified mug is placed on the designated branch.
- **Mug Slackness:** Only the branch is explicitly specified. For example, *"Put away a mug on the right branch."* Any mug can be selected, but it must be placed on the designated branch for the task to be considered successful.
- **Branch Slackness:** Only the mug is explicitly specified. For example, *"Hang the green mug on a branch. Sorry, the blue mug."* The specified mug must be picked up, but it can be placed on any available branch for the task to be considered successful.
- **Both Slackness:** Neither the mug nor the branch is explicitly specified. For example, *"Hang a mug on a branch."* In this case, placing any mug on any branch is considered successful.

*c) Stowing a Book:* A fixed number of books are placed on the bookshelf, with 1 to 2 of them tilted, creating 2 to 4 potential slots for stowing. Except for the 4 large books on the sides of the shelf, the order of the other books are randomized. There is also another randomly positioned book on a orange pad for picking. The goal of this task is to pick up the book on pad and stow it into a slot in the book shelf according to a given linguistic instruction.

We define three types of instructions based on the level of specificity: **no slackness, side slackness**, and **range slackness**.

- **No Slackness:** The slot is explicitly specified. For example, *"Place the front-most book so it sits directly next to DUBAI on the right side."* The task is only considered successful if the book on the pad is stowed in the designated slot.
- **Side Slackness:** The book should be placed on the specified side of the specified book. For example, *"Ensure LOST KINGDOMS is shelved on the orange book's right side. Sorry, the left side."*
- **Range Slackness:** The target slot is between two specified books. For example, *"Put the Neo's book, Istanbul, onto the shelf so that it is between Urbanlike and the black book."*

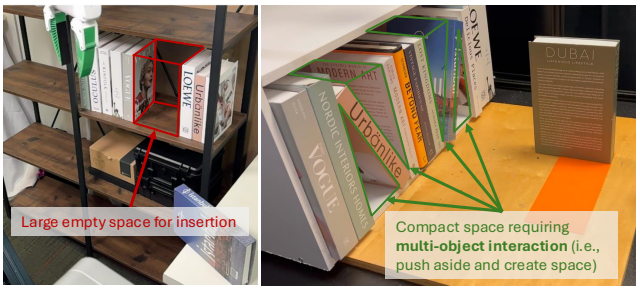
### C. Baseline Details

- **Lang-DP (PCD with color)**: The architecture of this baseline is essentially the same as Lang-DP (PCD) in the simulation experiments. The only difference is that the observation includes not only the point cloud of the scene but also additional RGB color channels for each point. In other words, each point consists of three channels representing its spatial location and three additional channels representing its color.
- **DP (PCD)**: It is similar to the Lang-DP (PCD) baseline in the simulation experiments. However, it can only observe raw point cloud without any hint from language.
- **Lang-DP (RGB)**: It is the same as the Lang-DP (RGB) used in the simulation experiments.

## V. Comparisons with VoxPoser

VoxPoser relies on off-the-shelf motion planners to execute low-level skills [1] and assumes simplified dynamics. For instance, once an object is grasped, it is treated as rigidly attached to the end-effector. As a result, VoxPoser is limited in handling scenarios involving complex dynamics or contact-rich interactions.

In contrast, many of the tasks in our work—such as book stowing—involve multi-object interactions and complex contact dynamics. As illustrated in Figure 6, prior work assumes an empty slot for book insertion. In our task setting, however, the robot must push adjacent books aside and squeeze out space—capabilities made possible by learning visuomotor policies from demonstrations.




(a) ReKep Stowing Book (B) Our Stowing Book

Fig. 6: **Experiment Setup Comparison.** ReKep's setup assumes an empty slot for book insertion, whereas our setup requires complex multi-object interaction to push books aside and create space.

We also conduct additional experiments in simulation, with example rollouts shown in Figure 7. Since VoxPoser relies on a pre-trained grasp planner, it may fail in scenarios that are not well-supported in the training data.

## References

[1] Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. Voxposer: Composable 3d value maps for robotic manipulation with language

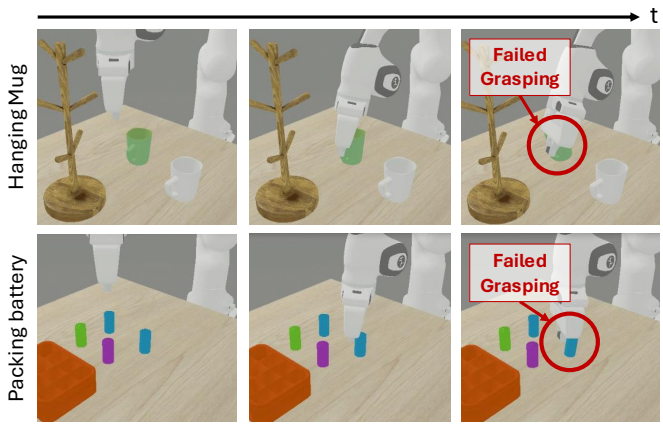



Fig. 7: **VoxPoser Rollout Examples.** VoxPoser rollouts in our simulation environments demonstrate that VoxPoser fails to grasp the target objects.

models. In *Conference on Robot Learning*, pages 540–562. PMLR, 2023.

[2] Tony Z Zhao, Vikash Kumar, Sergey Levine, and Chelsea Finn. Learning fine-grained bimanual manipulation with low-cost hardware. *arXiv preprint arXiv:2304.13705*, 2023.