# Scaling Up and Distilling Down: Language-Guided Robot Skill Acquisition

Supplementary Material

## Contents



## 1 Policy Rollout Visualizations

Our policy's 6DoF manipulation behavior is best visualized through videos. Please visit scalingup-distillingdown.github.io to view the videos.

7th Conference on Robot Learning (CoRL 2023), Atlanta, USA.

## 2 LLM Prompts

Below, we include all prompts used in our approach. We use the same LLM pipeline and prompts in all domains and tasks. We first outline the rationale behind our design of the LLM pipeline (§ 2.1). Next, we describe in detail the LLM modules and how they are used in the data generation stage (§ 2.2), summarize the general prompt structure (§ 2.3), and outline the API supplied to the LLM for success condition inference (§ 2.4). Finally, we show some examples of LLM completions (§ 2.5).

In all of our experiments, we use GPT3 (text-davinci-003) with temperature 0.0.

### 2.1 LLM Pipeline Design

Our LLM pipeline is factorized into multiple LLM modules, allowing each module's prompt to speciallize in a small reasoning skill (*e.g.* one set of prompts for deciding whether a task involves a single or multiple objects). We found that this not only improves the LLM's performance, but also makes designing and maintaining prompts easy. For instance, during development, if the LLM outputs an unexpected task tree, the error could be traced back to a single module, and only that module's prompt needs to be updated. Another convenient feature of this approach is that it also saves on token usage. Since each module's task is small (*e.g.* answer only "one" or "multiple"), the amount of completion tokens is significantly smaller than a monolithic prompt. Further, when a module's prompt is updated, only that module's outputs needs to be updated, allowing cost-effective approaches to cache-ing LLM's completions.

### 2.2 LLM Pipeline

The recursive LLM-based planner starts with an ambiguous task description handler (Listing 1), which transforms ambiguous task descriptions such as "move the block onto the catapult then shoot the block into the furthest bin" into more specific task descriptions like "move the block onto the catapult then shoot the block into the furthest bin by pressing the catapult's button". While this handler's task can occasionally overlap with the LLM planner's task, we found that it was more effective to keep them separate.

Next, given a un-ambiguous task description, the LLM planner first decides whether the planning step is necessary by checking whether the task involves touching only a single object or requiring further decomposition (Listing 2). If the task involves multiple objects, it proceeds with planning (explained in the next paragraph). If the task involves only one object part, an LLM identifies which object part name it should interact with (Listing 3). If the object part name is a single-link rigid object, the LLM is asked for which object it should move (the pick object part) and where (the place object part) using the prompt in Listing 4.

In the planning step, the LLM planner outputs a list of subtasks (Listing 5). Given the recursive nature of this planning module, parent tasks also need to keep track of and propagate the current state of the environment to child tasks. For instance, the "open the fridge" subtask should be followed with "with the fridge door opened, move the eggs from the fridge ...", such that the recursive call for moving the eggs knows it does not need to open the fridge door again.

After it has inferred the full task tree, the LLM also infers a success condition for every task in the task tree (Listing 6) in the form of a code-snippet. Similar to [1], we inform the LLM which state API utilities are available for its usage by including import statements at the top of the file and demonstrating how they are used in the examples.

### 2.3 Prompt Structure

All prompts start with instructions to explain to the LLM what the task is (*e.g.* "given an input task description, the goal is to output a list of subtasks .."), followed by a few "shots" of examples, separated by a "#" symbol (in text-based prompts) or a multi-line comment (in code-based prompts). Each shot starts with a structured text encoding of the scene's object's and their parts' names in the form of a bullet list. In the planning, success condition inference, single-or-multiple , pick-and-place, and ambiguous task description LLM tasks, we found that it was helpful to encourage the LLM to output its reasoning (either with an explicit "reasoning:" field or through in-line code comments). In contrast, we found the object part identifier task to be more effective without this explicit reasoning field.

### 2.4 APIs for Success Condition Code Generation

All functions take as the first argument the simulation state, which contains information on object and part names, kinematic structure, contact, all degrees of freedom, and collision meshes.

**Contact.** This function takes as input two object (part) names, and returns whether they (or any of their parts) are in contact.

**Activation.** A pair of functions, `check_activated` and `check_deactivated`, take as input an object part name and checks whether the revolute/prismatic joint connecting the object part to its parent link are near their maximum or minimum values, respectively. This is useful for checking whether a lid is opened/closed or a button is pressed/released.

**Spatial Relations.** We provide two spatial relations, `check_on_top_of` and `check_inside`, which takes two object (part) names and returns whether the first object (part) is on top of the second object (part) or inside the second object (part), respectively. An object is on top of another if they are in contact and the contact normal's dot product with the up direction is greater than 0.99. An object is inside a container if that the intersection of that object's axis-aligned bounding boxes with the container's axis-aligned bounding boxes is at least 75% of the object's axis-aligned bounding box's volume. This axis-aligned bounding box information can be parsed from the collision checker of most physics simulators.

Listing 1: Ambiguous task description handler's prompts

```
instructions:
given an input task description, the goal is rephrase the task such that it is not ambiguous.
if the task is already specific enough, just return the original task description.
below are some examples:
#
task: stack the blocks on top of each other
scene:
 - navy block
 - maroon block
 - violet block
reasoning: the block stacking order is ambiguous. we can specify which block should be placed on which, in which order.
answer: move the maroon block onto the navy block, and the violet block on the maroon block.
#
task: move the lilac block onto the brown block
scene:
 - brown block
 - lilac block
 - yellow block
reasoning: the blocks to interact with are fully specified, so just return the original task description.
answer: move the lilac block onto the brown block.
#
task: sort the blocks based on their color's temperature onto corresponding plates
scene:
 - red block
 - orange block
 - blue block
 - purple block
 - red plate
 - blue plate
reasoning: which blocks and plates belong to the same color temperature group are ambiguous. we can specify exactly which blocks should be placed on which plate.
answer: move the red and orange blocks onto the red plate, and the purple and blue blocks onto the blue plate.
#
task: open the jar
scene:
 - jar
    + jar lid
reasoning: opening a jar is a primitive action and is fully specified, so just return the original task description.
answer: open the jar.
#
task: close the second drawer
scene:
 - drawer
    + first drawer
       + first drawer handle
    + second drawer
       + second drawer handle
```

```
    + third drawer
       + third drawer handle
reasoning: closing the second drawer is a
     primitive action towards a specific drawer, so just return the original task description.
answer: close the second drawer.
#
task: move the ingredients for the omelette onto the kitchen counter
scene:
 - kitchen counter
    + cupboard
       + cupboard door
          + cupboard door handle
       + salt
       + pepper
 - fridge
    + fridge door
       + fridge door handle
    + fridge top shelf
       + eggs
       + butter
       + cheese
       + milk
    + fridge bottom shelf
       + mushrooms
       + broccoli
    + freezer
       + lamb shank
       + trader joe's dumplings
       + tilapia fillet
reasoning: which ingredients belong to
      the omelette is ambiguous. we can specify exactly which items to take out of the fridge.
answer: move the eggs, butter, cheese,
      and mushrooms onto the kitchen counter and the salt and pepper onto the kitchen counter.
#
task: open the fridge, move the cheese onto the kitchen counter, and then close the fridge.
scene:
 - kitchen counter
    + cupboard
       + cupboard door
          + cupboard door handle
       + salt
       + pepper
 - fridge
    + fridge door
       + fridge door handle
    + fridge top shelf
       + eggs
       + butter
       + cheese
       + milk
    + fridge bottom shelf
       + mushrooms
       + broccoli
    + freezer
       + lamb shank
       + trader joe's dumplings
       + tilapia fillet
reasoning: which actions to perform
      and in which order is fully specified, so just return the original task description.
answer: open the fridge, move the cheese onto the kitchen counter, and then close the fridge.
```

Listing 2: One-or-Multiple module's prompts

```
instructions:
given an input task description, the goal is to classify whether
      performing the task will involve touching only "one" object or "multiple" objects.
all objects start in a de-activated
      state (e.g., doors, drawers, cabinets, cupboards, and other objects with doors are
      closed, lights are off, etc.) unless specified otherwise (e.g., with the door opened).
after performing the task, objects should be reset to their de-activated state if relevant.
below are some examples:
#
task: move the blue block onto the plate
scene:
 - green block
 - blue block
 - red block
 - plate
```

```
reasoning: "moving the blue block onto the plate" involves two objects, the blue block and the plate. moving the blue block requires touching it. the plate does not have any activation state, so does not need to be touched.
answer: one.
#
task: stack the blocks on the plate
scene:
 - green block
 - plate
 - red block
 - blue block
reasoning: "stack the blocks" can be decomposed into moving the red block onto the plate, moving the green block onto the red block, and moving the blue block onto the green block. performing these steps involve touching multiple blocks.
answer: multiple.
#
task: with the red block on the plate and the orange block on the red block, move the green block onto the pink block
scene:
 - orange block
 - pink block
 - plate
 - green block
 - red block
reasoning: "moving the green block onto the pink block" involves two objects, the green block and the pink block. moving the green block requires touching it. the pink block does not have any activation state, so does not need to be touched.
answer: one.
#
task: move the lasagna into the microwave
scene:
 - microwave
    + microwave door
       + microwave door handle
 - kitchen counter
 - fridge
    + fridge door
       + fridge door handle
 - lasagna
reasoning: "moving the pasta into the microwave" involves only two objects, the lasagna and the microwave. however, it is not a primitive task because the microwave has a door (activation state), but it starts off being closed (de-activated). opening the microwave involves touching the microwave.
answer: multiple.
#
task: with the microwave opened, move the pasta into the microwave
scene:
 - microwave
    + microwave door
       + microwave door handle
 - kitchen counter
 - fridge
    + fridge door
       + fridge door handle
 - pasta
reasoning
    : "moving the pasta into the microwave" involves two objects, the pasta and the microwave
    . the microwave's door needs to be opened (activation state), but it is already opened
    . since the task asserts that the microwave is opened, it also does not need to be closed afterwards. this means performing the task does not involve touching the microwave.
answer: multiple.
#
task: open the microwave
scene:
 - fridge
    + fridge door
       + fridge door handle
 - dumplings
 - microwave
    + microwave door
       + microwave door handle
 - kitchen counter
reasoning:
     "opening the microwave" is a primitive task. it involves only one object, the microwave.
answer: one.
#
task: with the microwave opened and the sandwich in the microwave, close the microwave
scene:
 - fridge
    + fridge door
       + fridge door handle
```

```
 - sandwich
 - microwave
    + microwave door
       + microwave door handle
 - kitchen counter
reasoning:
      "closing the microwave" is a primitive task. it involves only one object, the microwave.
answer: one.
```

Listing 3: Object part identifier's prompts

```
instructions: given an input task
      description, the goal is to identify which object part from the scene to interact with.

below are some examples:
#
task: stack the blue block on the plate
scene:
 - red block
 - blue block
 - green block
 - plate
answer: blue block.
#
task: with the red block on the plate, stack the green block on the red block
scene:
 - red block
 - blue block
 - green block
 - plate
answer: green block.
#
task: turn on the lights
scene:
 - light switch
 - ceiling light
 - wall
answer: light switch.
#
task: open the microwave
scene:
 - microwave
    + microwave door
       + microwave door handle
    + microwave start button
    + microwave plate
 - kitchen counter
    + cupboard
       + cupboard door
          + cupboard door handle
answer: microwave door handle.
#
task: with microwave
      opened and the lasagna on the kitchen counter, move the lasagna into the microwave
scene:
 - kitchen counter
    + cupboard
       + cupboard door
          + cupboard door handle
 - fridge
    + fridge door
       + fridge door handle
    + fridge top shelf
    + fridge bottom shelf
    + freezer
 - lasagna
 - microwave
    + microwave door
       + microwave door handle
    + microwave start button
    + microwave plate
answer: lasagna.
#
task: with the fridge door opened, open the cupboard
scene:
 - microwave
    + microwave door
       + microwave door handle
    + microwave start button
    + microwave plate
```

```
 - kitchen counter
    + cupboard
       + cupboard door
          + cupboard door handle
 - fridge
    + fridge door
       + fridge door handle
    + fridge top shelf
    + fridge bottom shelf
    + freezer
 - lasagna
answer: cupboard door handle.
```

Listing 4: Pick & place handler's prompts

```
instructions: given an input pick and place description, the goal is to
      identify which object to pick and where to place among the objects listed in the scene.

below are some examples:
#
task: move the blue block on the plate
scene:
 - red block
 - blue block
 - green block
 - plate
pick: blue block.
place: plate.
#
task: with the red block on the plate, move the green block to the top of the red block
scene:
 - red block
 - blue block
 - green block
 - plate
pick: green block.
place: red block.
#
task: with microwave
      opened and the lasagna on the kitchen counter, move the lasagna into the microwave
scene:
 - kitchen counter
    + cupboard
       + cupboard door
          + cupboard door handle
 - fridge
    + fridge door
       + fridge door handle
    + fridge top shelf
    + fridge bottom shelf
    + freezer
 - lasagna
 - microwave
    + microwave door
       + microwave door handle
    + microwave start button
    + microwave plate
pick: lasagna.
place: microwave plate.
```

Listing 5: Planning module's prompts

```
instructions: given a input task description
     , the goal is to output a list of subtasks, which, when performed in sequence would
      solve the input task. all objects start in a de-activated state (e.g., doors, drawers
     , cabinets, cupboards, and other objects with doors are closed, lights are off, etc
     .) unless specified otherwise (e.g., with the door opened). after performing the task,
     objects should be reset to their de-activated state if possible. below are some examples:
#
task: move the red block
      onto the plate, the blue block onto the red block, and the green block on the blue block
scene:
 - red block
 - blue block
 - green block
 - plate
reasoning
     : no objects have activation states. the blocks can be directly placed onto the plates.
answer:
```

```
 - 1. move the red block onto the plate
 - 2. with the red block on the plate, move the blue block onto the red block
 - 3. with the red block on
       the plate and the blue block on the red block, move the green block onto the blue block
#
task: move the eggs, salt, and pepper onto the kitchen counter
scene:
 - kitchen counter
    + cupboard
       + cupboard door
          + cupboard door handle
       + salt
       + pepper
 - fridge
    + fridge door
       + fridge door handle
    + fridge top shelf
       + eggs
       + butter
       + cheese
       + milk
    + fridge bottom shelf
    + freezer door
       + freezer door handle
reasoning: the fridge and cupboard has doors
     (activation states) which start off closed (de-activated). they need to be opened before
     objects can be taken out of them. after the task is done, they need to be closed (reset).
answer:
 - 1. open the fridge
 - 2. with the fridge door opened, move the eggs from the fridge onto the kitchen counter
 - 3. with the eggs on the kitchen counter, close the fridge
 - 4. with the eggs on the kitchen counter, open the cupboard
 - 5. with the eggs on
      the kitchen counter and the cupboard door opened, move the salt onto the kitchen counter
 - 6. with the eggs and salt on the
       kitchen counter and the cupboard door opened, move the pepper onto the kitchen counter
 - 7. with the eggs, salt, and pepper on the kitchen counter, close the cupboard door
#
task: with the fridge door opened, move the eggs, salt, and pepper onto the kitchen counter
scene:
 - kitchen counter
    + cupboard
       + cupboard door
          + cupboard door handle
       + salt
       + pepper
 - fridge
    + fridge door
       + fridge door handle
    + fridge top shelf
       + eggs
       + butter
       + cheese
       + milk
    + fridge bottom shelf
    + freezer door
       + freezer door handle
reasoning: the fridge and cupboard has doors (activation states
     ). the fridge's door is already opened (activated) and so don't need to be reset. the
      cupboard's door starts off closed (de-activated) but needs to be opened before objects
      can be taken out of it. after the task is done, the cupboard need to be closed (reset).
answer:
 - 1. with the fridge door opened, move the eggs from the fridge onto the kitchen counter
 - 2. with the fridge door opened and the eggs on the kitchen counter, open the cupboard
 - 3. with the fridge door opened, the eggs on the
       kitchen counter, and the cupboard door opened, move the salt onto the kitchen counter
 - 4. with the fridge door opened, the eggs and salt on the
       kitchen counter, and the cupboard door opened, move the pepper onto the kitchen counter
 - 5. with the fridge
       door opened, the eggs, salt, and pepper on the kitchen counter, close the cupboard door
```

Listing 6: Success Condition Inference module's prompts

```
from utils import (
    check_contact,
    check_activated,
    check_deactivated,
    check_inside,
    check_on_top_of,
    EnvState,
```

```
)


"""
instructions:
given a input task description, the goal is to output the success condition for
that task. unless otherwise specified, all objects start in a de-activated state
(e.g., doors, drawers, cabinets, cupboards, and other containers are closed,
lights are off, etc.) unless specified otherwise (e.g., with the door opened).
after performing the task, objects should be reset to original state if possible.
"""


# robot task: touch the apple
# scene:
# - apple
#    + apple body
#    + apple stem
def touching_apple(init_state: EnvState, final_state: EnvState):
    return check_contact(
        final_state, "robotiq left finger", "apple body"
    ) and check_contact(final_state, "robotiq right finger", "apple body")


# robot task: release the cup
# scene:
# - cup
#    + cup body
#    + cup handle
def released_cup(init_state: EnvState, final_state: EnvState):
    finally_touching_cup = check_contact(
        final_state, "robotiq left finger", "cup handle"
    ) and check_contact(final_state, "robotiq right finger", "cup handle")
    finally_released_cup = (not finally_touching_cup) and (
        not final_state.gripper_command
    )
    return finally_released_cup


# robot task: move the milk carton into the shelf
# scene:
# - milk carton
# - coke can
# - shelf
def milk_carton_is_on_shelf(init_state: EnvState, final_state: EnvState):
    return check_on_top_of(final_state, "milk carton", "shelf")


# robot task: move the milk carton from the shelf
# scene:
# - milk carton
# - coke can
# - shelf
def milk_carton_is_not_on_shelf(init_state: EnvState, final_state: EnvState):
    return not check_on_top_of(final_state, "milk carton", "shelf")


# robot task: open the washing machine
# scene:
# - washing machine
#    + washing machine door
#       + washing machine door handle
#    + control panel
#       + on off button
def washing_machine_opened(init_state: EnvState, final_state: EnvState):
    return check_activated(final_state, "washing machine door")


# robot task: move the sock into the washing machine
# scene:
# - washing machine
#    + washing machine door
#       + washing machine door handle
#    + control panel
#       + on off button
# - sock
def sock_inside_washing_machine(init_state: EnvState, final_state: EnvState):
    # the washing machine can be opened (activated state) or closed (de-activated
    # state). since its activation state was not specified, the washing machine starts
    # off closed. therefore, it needs to be closed after the sock is moved inside.
    sock_inside_washing_machine = check_inside(final_state, "sock", "washing machine")
```

```
    washing_machine_door_closed = check_deactivated(final_state, "washing machine door")
    return sock_inside_washing_machine and washing_machine_door_closed


# robot task: with the washing machine opened, move the sock into the washing machine
# scene:
# - washing machine
#    + washing machine door
#       + washing machine door handle
#    + control panel
#       + on off button
# - sock
def sock_inside_washing_machine_with_washing_machine_opened(
    init_state: EnvState, final_state: EnvState
):
    # the washing machine can be opened (activated state) or closed (de-activated
    # state). the washing machine starts off opened, so it does not need to be closed
    # after the sock is moved inside.
    sock_inside_washing_machine = check_inside(final_state, "sock", "washing machine")
    return sock_inside_washing_machine
```

### 2.5 Example Completions

Most modules are designed such that their answers are multi-choice (*e.g.* single or multiple). We show examples of more interesting open-ended completions for the ambiguous task description handler, planning, and success condition inference modules. The LLM's outputs are highlighted in blue.

#### 2.5.1 Ambiguous Task Description Handler

```
task: move the block onto the catapult arm, then press the button to shoot the block into the
closest box
scene:
- catapult
    + button
    + catapult arm
- yellow block
- closest box
- furthest box
- middle box
- table
reasoning: which block to move onto the catapult arm is ambiguous. we can specify exactly
which block to move onto the catapult arm.
answer: move the yellow block onto the catapult arm, then press the button to shoot the block
into the closest box.
```

```
task: send the amazon package for return
scene:
- mailbox
   + mailbox lid
      + mailbox lid handle
   + mailbox flag
- amazon package
- table
reasoning: which actions to perform and in which order is ambiguous. we can specify exactly
 which actions to take.
answer: open the mailbox lid, place the amazon package inside the mailbox, raise the mailbox
flag, and then close the mailbox lid.
```

### 2.5.2 Planning

```
task: move the yellow block onto the catapult arm, then press the button to shoot the block
into the closest box
scene:
- catapult
  + button
  + catapult arm
- yellow block
- closest box
- furthest box
- middle box
- table
reasoning: the catapult has a button (activation state) which starts off de-activated. it
needs to be pressed to shoot the block. after the task is done, the button should be reset to
its de-activated state.
answer:
 - 1. move the yellow block onto the catapult arm
 - 2. with the yellow block on the catapult arm, press the button to shoot the block into the
 closest box
 - 3. with the yellow block in the closest box, reset the button to its de-activated state
```

```
task: open the mailbox lid, place the amazon package inside the mailbox, raise the mailbox
flag, and then close the mailbox lid
scene:
- mailbox
  + mailbox lid
    + mailbox lid handle
  + mailbox flag
- amazon package
- table
reasoning: the mailbox lid has an activation state (closed/de-activated). it needs to be
opened before the package can be placed inside. after the task is done, the lid needs to be
closed (reset).
answer:
- 1. open the mailbox lid
- 2. with the mailbox lid opened, move the amazon package from the table into the mailbox
- 3. with the amazon package in the mailbox, raise the mailbox flag
- 4. with the amazon package in the mailbox and the mailbox flag raised, close the mailbox lid
```

### 2.5.3 Success Condition Inference

```
# robot task: move the yellow block onto the catapult arm
# scene:
# - catapult
#   + button
#   + catapult arm
# - yellow block
# - closest box
# - furthest box
# - middle box
# - table
def yellow_block_on_catapult_arm(init_state: EnvState, final_state: EnvState):
return check_on_top_of(final_state, "yellow block", "catapult arm")
```

```
# robot task: open the mailbox lid
# scene:
# - mailbox
#   + mailbox lid
#     + mailbox lid handle
#   + mailbox flag
# - amazon package
# - table
def mailbox_lid_opened(init_state: EnvState, final_state: EnvState):
return check_activated(final_state, "mailbox lid")
```

# 3 Training & Data Details.

## 3.1 Data Generation

Our data-collection policy uses the 6DoF Exploration Primitives with the Verify & Retry step. For each domain, we run data generation until we get at least 500 successful trajectories per task. Although this can be costly when tasks are long horizon with low success rates (the mailbox domain took 2 days on 256 CPU cores Intel Xeon Gold 6230R CPU @ 2.10GHz), data generation happens only once.

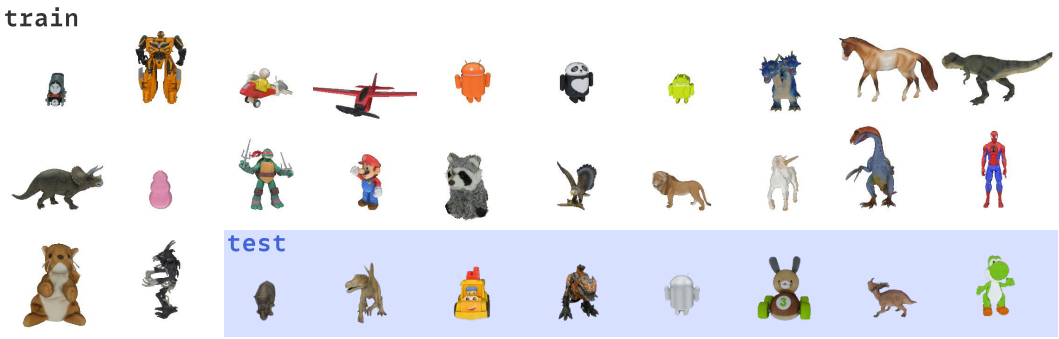


Figure 1: **Generalization to Novel Objects.** The Transport domain requires generalization to diverse and novel object shapes and colors. Trained to transport 22 toys, our distilled policy generalizes to 8 novel toys (in blue section). All objects rendered from a fixed camera to show diversity of object size.

### 3.2 Network Architecture & Hyperparameters

We use the same network architecture and hyperparameters for all domains. Our task descriptions are encoded using CLIP B/32's text encoder [2], and projected into a 512-dimensional vector. For each of the two camera view, we learn a separate Resnet18-based vision encoder, whose features are flattened, concatenated, and projected into a 512-dimensional vector. The Resnet18 architecture is pre-processed by replacing BatchNorm with GroupNorm and replacing the final average pool layer with a spatial softmax pooling [3]. We use an image resolution of $80 \times 120$ for each view, processed with a random crop to $72 \times 108$. Finally, the proprioception is concatenated with the vision and text encoder as the condition into the diffusion policy. We use the convolution network-based diffusion policy architecture [3]. The final network has 108 million parameters. All networks are optimized end-to-end with the AdamW optimizer, with 5e-5 learning rate and 1e-6 weight decay, and a cosine learning rate scheduler. For evaluation, we use an exponential moving average of all networks with a decay rate of 0.75.

### 3.3 Training

We train a separate multi-task policy for each domain using the same hyperparameters and network architecture. For domains with only a single task, this amounts to a single-task policy. All networks are trained for 72 hours on a single NVIDIA A6000, and the best checkpoint's performance is reported.

## 4 Utilities Implementation

For motion planning, we implemented rapidly-exploring random trees (RRT [4]) with grasped-object-aware collision checking, allowing the robot to motion plan with dynamic grasping constraints. The geometry-based grasp and placement sampler is implemented using point clouds created from depth maps, camera matrices, and segmentation maps from the simulator. While our grasp sampler uses only geometry, kinematics, and contact information, including other grasp quality metrics (*e.g.* stability analysis) can improve its performance. In the placement sampler, we sample candidate place positions at points whose estimated contact normal is aligned against the gravity direction. The revolute and prismatic joint motion primitives are implemented by checking the grasp pose relative to the joint (*e.g.* mailbox lid handle grasp relative to the mailbox lid hinge), then performing a circular motion around the joint axis or a linear motion along the joint axis, respectively.

## 5 Benchmark

Our benchmark is built on top of the Mujoco [5] simulator, using assets from the Google Scanned Objects dataset [6, 7]. We use a table-top manipulation set-up, with a WSG50 gripper and Toyota Research Institute Finray fingers mounted on a UR5e. The workspace has two cameras, one front view, which observes the entire workspace and robot, and a wrist-mounted camera, which is used to help with fine-grained manipulation [8]. Since all tasks are quasi-static, we opted for a 2Hz control rate to allow objects more time

| Approach | Pencilcase | | | Crayon | | | Vitamin | | | Horse | | | Avg. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Top | Mid | Bot | Top | Mid | Bot | Top | Mid | Bot | Top | Mid | Bot | |
| LLM-as-Policy (2D) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| (+) 6DoF Robot Utils | 8 | 8 | 8 | 18 | 42 | 16 | 0 | 7 | 0 | 6 | 7 | 2 | 10 |
| (+) Verify & Retry | 32 | 42 | 46 | 56 | 76 | 58 | 16 | 40 | 38 | 28 | 22 | 18 | 39 |
| (+) Distill (Ours) | 18 | 46 | 54 | 44 | 56 | 54 | 8 | 14 | 32 | 10 | 26 | 28 | 33 |

Table 1: **Drawer Quantitative Results (Success Rate %).**

to settle (*e.g.* after being dropped) between control cycles and reduce the number of control horizon. We end episodes when any object is dropped to the floor. Below, we clarify how we design the tasks for each domain.

### 5.1 Mailbox

To be considered successful, the mailbox needs to be closed with the package inside the mailbox, with the mailbox flag raised within 200 seconds (400 control cycles). During data generation and testing, the package's planar position is uniformly random in a planar bound of dimensions [20cm, 10cm]. At evaluation, the policy has to generalize to unseen package positions. The amazon has is a rigid object with 6DoF. The mailbox is a fixed rigid object, with one degree of freedom for each of its revolute joints, one for the mailbox lid, and one for the mailbox flag.

### 5.2 Transport

To be considered successful, the toy needs to be inside the left bin within 100 seconds. At the beginning of each episode, a random toy 3D asset is sampled. During data generation and testing, the toy's position is uniformly random inside the right bin, and orientation uniformly random along all three euler axes. On top of novel randomized poses, the policy also has to generalize to unseen object instances with novel geometry. We use 22 toys for data generation, and 8 for testing (Fig. 1). The toy is a rigid object with 6DoF, while the bins are fixed rigid objects with no DoF. The bin asset names corresponds with their spatial location (*e.g.* the left bin is called "left bin" when the scene is presented to the LLM).

### 5.3 Drawer

This is a multi-task domain with 12 tasks, where each task involves moving one of the four objects (vitamin bottle, pencil case, crayon box, horse) into one of the three drawers (top, middle, bottom). The task description follows the template "move the ⟨*object*⟩ into the ⟨ *drawer*⟩". To be considered successful, the specified object needs to be inside the specified drawer within 120 seconds. During data generation and testing, each of the four object's position is uniformly random within a planar bound of dimensions [10cm,10cm], centered around 4 evenly spaced locations along the table. At test time, the policy has to generalize to unseen object positions in the same distribution as its data generation.

All four objects are rigid objects with 6DoF. The drawer is a fixed articulated object with 3 DoF, one for each of the drawers.

### 5.4 Catapult

This is a multi-task domain with 3 tasks, one for each of the three bins. The task description follows the template "move the block onto the catapult arm, then press the button to shoot the block into the ⟨*bin*⟩" where ⟨*bin*⟩ is either closest, middle, or furthest bin. The bin asset names corresponds with their spatial location (*e.g.* the furthest bin is called "furthest bin" when the scene is presented to the LLM).

In order to be considered successful, the block needs to be inside the specified bin within 40 seconds. This is a short amount of time, which prevents policies from retrying after failure. The block is a rigid object with 6DoF. The bins are fixed rigid objects with no degrees of freedom. The catapult has two degrees of freedom, one revolute joint for the catapult arm, and one prismatic joint for the button. This task is designed to study tool-use, and does not have any pose randomization. Thus, different seeds affect only the policy's pseudo random samplers or the diffusion process.

We implement the catapult with a special callback function which checks whether the button sliding joint is near its max value. If it is, then the constraint that holds the catapult arm down is disabled, releasing the spring loaded catapult arm hinge joint.

| Approach | Balance | Catapult | | | Transport | | Mailbox |
|---|---|---|---|---|---|---|---|
| | | Near | Mid | Far | Train | Test | |
| LLM-as-Policy (2D) | 36.0 | 0.0 | 100.0 | 0.0 | — | 18.0 | 0.0 |
| (+) 6DoF Robot Utils | 4.0 | 5.3 | 0.8 | 0.7 | — | 42.0 | 0.0 |
| (+) Verify & Retry | 24.0 | 5.3 | 0.8 | 0.7 | — | 72 | 6.0 |
| (+) Distill (Ours) | 72.0 | 70.0 | 100.0 | 70 | 50.0 | 50.0 | 48.0 |

Table 2: **Full Quantitative Results (Success Rate %).**

### 5.5 Bus Balance

In order to be considered successful, the bus needs to be fully balanced on top of the block within 100 seconds. On top of testing for intuitive physics, this high precision requirement of this task was also used to test the policy's precision and ability to recover from failure, which is why we allow a generous time budget. The task description is "balance the bus on the block".

The bus is a rigid object with 6DoF, dropped from a fixed location above the table with uniformly random orientation. This means when the bus drops, it lands in different positions and orientations. The block is fixed with no degrees of freedom.

## 6 Full Results

We include the full results for all tasks in the drawer domain in Table 1, and all other domains in Table 2. We omit data generation baseline numbers on the train set in the transport domain, since they are non-learning approaches. All approaches are evaluated on 50 different seeds, which controls pose randomization, which asset is sampled, the pseudo-random robotic utility samplers, and the pseudo-random diffusion process. We make one exception in the catapult domain, where we evaluate on 1000 seeds due to the low success rates of getting the block into the middle and far bin. Since the time limit for the catapult is short, the data-collection policy will not have enough time to retry, leading to identical numbers with the baseline data-collection policy without verify & retry.

In the drawer domain, we observe that the task is more difficult for:

1. **Larger objects:** The most challenging objects are the vitamin bottle and the horse toy, both of which are too large to fit the drawer if they are in an upright orientation. This means to be effective at this task, the robot should perform sideway grasps on these objects, such that downstream placement is easier. In contrast, the small crayon box is has the highest success rates amongst the data-collection policies.
2. **Top drawer:** We observe interacting with this drawer often brings the robot close to its kinematic reach range. This means slight imprecision in the policy's predicted actions or small shifts in the grasped object (which is unaccounted for during motion planning) in execution could lead to failure. For instance, while moving the objects inside the top drawer, the grasped object could collide with the drawer, causing the grasped object to drop or the drawer to close.
3. **Planar Action Primitives:** A top-down grasp on the drawer handle will typically be in collision with the drawer's body. Thus, in LLM-as-Policy (2D)'s first action to open the drawer, its call to the motion planner will fail due to an invalid goal configuration.

## References


[1] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng. Code as policies: Language model programs for embodied control. In *arXiv preprint arXiv:2209.07753*, 2022.

[2] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, et al. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, pages 8748–8763. PMLR, 2021.

[3] C. Chi, S. Feng, Y. Du, Z. Xu, E. Cousineau, B. Burchfiel, and S. Song. Diffusion policy: Visuomotor policy learning via action diffusion. In *Proceedings of Robotics: Science and Systems (RSS)*, 2023.

[4] S. M. LaValle et al. Rapidly-exploring random trees: A new tool for path planning.

[5] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012. doi:10.1109/IROS.2012.6386109.

[6] L. Downs, A. Francis, N. Koenig, B. Kinman, R. Hickman, K. Reymann, T. B. McHugh, and V. Vanhoucke. Google scanned objects: A high-quality dataset of 3d scanned household items, 2022. URL https://arxiv.org/abs/2204.11918.

[7] K. Zakka. Scanned Objects MuJoCo Models, 7 2022. URL https://github.com/kevinzakka/mujoco_scanned_objects.

[8] A. Mandlekar, D. Xu, J. Wong, S. Nasiriany, C. Wang, R. Kulkarni, L. Fei-Fei, S. Savarese, Y. Zhu, and R. Martín-Martín. What matters in learning from offline human demonstrations for robot manipulation. In *arXiv preprint arXiv:2108.03298*, 2021.