Appendices

A Visual Perception Module

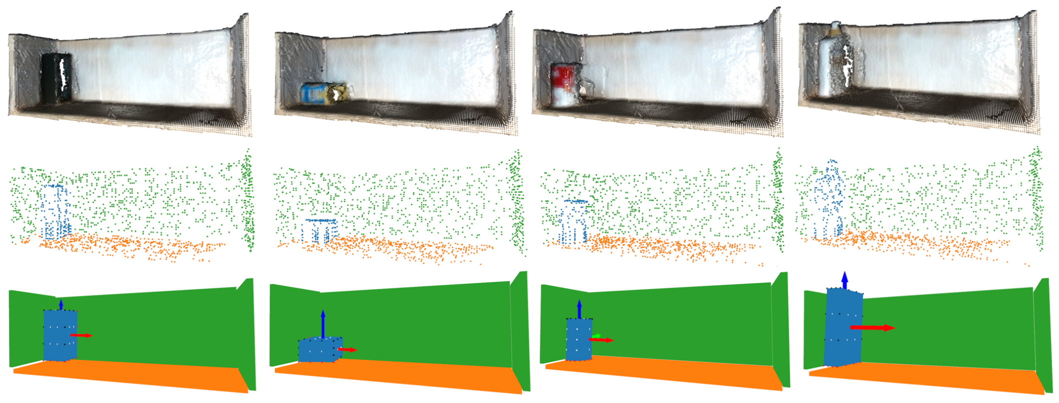
Figure 5: Additional example outputs of our visual perception module on real-world observations. Top row are visualizations of TSDF volumes fused from 6 RGBD images taken around the shelf. Note that these are partial observations, and there are missing data in the point clouds. Middle row are visualizations of segmented points. There are only two segmentation classes the precondition function uses — object and shelf. We color the shelf's bottom points in visualization for clarity. The bottom row are estimated geometries for both the objects and the shelf, as well as the estimated object pose. Black points on the object surface are potential contact points that are not in contact; red points are contact points that are in contact. White points are potential initial robot-object contact points used for parameter generation.

Here we provide additional details on the visual perception module. It segments object and environment classes from point clouds, and then it extracts object primitive geometries, object pose, and environmental planar constraints from the segmented point clouds. In our domain that has a rectangular shelf, environment points are segmented out by fitting an oriented bounding box to the entire scene. Then, we fit the corresponding object primitive geometry to the remaining object points, the centroid and principal axes of which form the object pose. We initially planned to train neural networks to perform these vision tasks, but we found that our existing pipeline (using Open3D [49] and Trimesh [50]) was already sufficiently robust for our domain.

For obtaining environment information, finding environmental planar constraints and performing object-environment segmentation can be done together. We assume the environment has a cuboid shape (e.g. a shelf), so finding points that belong to the environment can be done by first fitting an oriented bounding box and then labeling all points close to the surface of that bounding box as environment points.

For finding the object bounding box and pose of cuboids, we first fit an oriented bounding box to the segmented object points. This is done by clustering the object normals and treating the cluster centers with the 2 highest counts as the principal directions for the oriented bounding box. The dimensions of the bounding box can be directly computed by projecting object points onto these directions and computing the corresponding extents. These directions, along with the centroid of the oriented bounding box, are used to form the object pose.

For fitting cylinder primitives, we first fit an oriented bounding box on the partial point cloud, then use the bounding box's rotation axes as candidates for the cylinder's primary axis (about which there is rotational symmetry). For each candidate axis we rotate the partial point cloud around it by 90 degrees and fit another oriented bounding box on both the original and the rotated points. The candidate axis that corresponds to the new bounding box with the smallest volume is chosen as the primary rotation axis, and we extract the radius and pose of the cylinder from the dimensions and pose of this bounding box.

While the visual perception module was sufficient for our experiments, it does have limitations and failure modes that hinder more general use. Specifically, fitting object geometry primitives to noisy point cloud

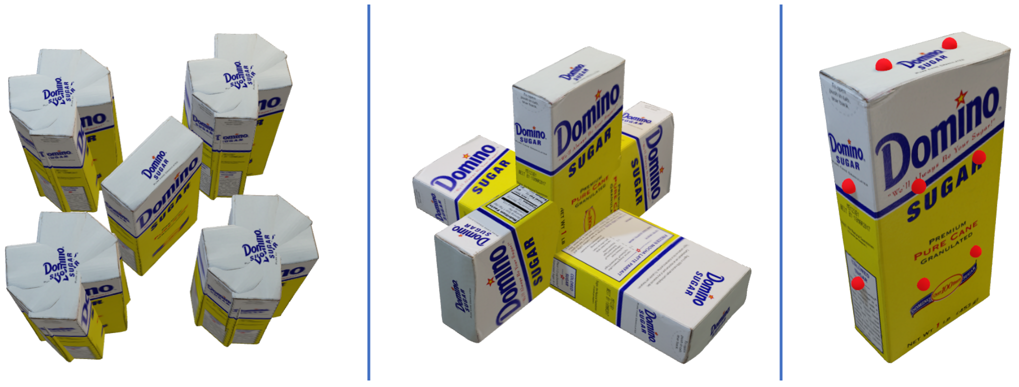
Figure 6: Subgoal generation. Left: planar subgoal poses — combination of 4 translations and 3 rotations (middle object is the original pose). Translations have been exaggerated for visual clarity. Middle: subgoals that are in neighboring stable poses (middle object is original pose). Right: Red dots are initial robot-object contact points come from a grid on the object surface.

data is not as reliable in the real world as it is in simulation. Incorrectly estimated object shapes and poses, leading to infeasible skill parameters, is the most common cause of real-world skill execution failures. The visual perception module can be improved by using learned models, and this may also allow it to generalize to a broader range of object geometries. We did not pursue furthering the capabilities of the visual perception module to focus time and engineering effort toward the contributions of this paper, but we are confident that our method could be extended to more complex object geometries as the neither precondition model nor HFVC synthesis rely on these geometric primitives (the precondition model uses point clouds, and HFVC synthesis uses sparse contact points which can be inferred by learned methods).

B HFVC Skill

B.1 HFVC Parameter Generation

Here we provide more details on HFVC skill parameter generation. See Figure 6 Recall there are two components of the generated parameter — the desired object subgoal pose and a robot-object contact point.

There are two types of subgoals we generate — ones in the same stable pose, and ones that are in neighboring stable poses (rotate by $90°$). For the subgoals in the same stable pose, we form 4 delta translations along the positive and negative principal directions of the environment bounding box. This allows us to sample subgoals that move the object along the walls of the environment. The translation magnitudes are the object bounding box dimensions in the corresponding directions. This allows the translation magnitudes to automatically scale with object size. In addition, we form 3 planar rotations, with degrees $[-30°, 0°, 30°]$ that are applied to each of the 4 delta translations. For subgoal poses in different stable poses, we obtain "neighboring" stable poses that are $90°$ apart by rotating the current object pose around the object bounding box's principal axes. We do not apply additional planar rotations to these subgoal poses. The planar rotations and different stable poses allow the HFVC skill to achieve complex behaviors, like pivoting and toppling. There are at most $4 \times 3 + 4 = 16$ subgoal poses generated, and we filter out the ones that intersect with the environment. In the context of planning, these subgoal poses can be thought of as motion primitives that are object-centric instead of robot-centric.

For cylinders, when they are vertical, we generate neighboring stable poses by applying 90 degree rotations in the 2 axes that are not the axis of symmetry. This looks similar to the cuboid stable pose figure in the Appendix. When cylinders are not vertical, only 1 axis would give a different neighboring stable pose (horizontal -¿ vertical), so we only use that one. There are infinite pairs of axes that are orthogonal to the axis of symmetry - we pick the pair that's aligned with the shelf walls, so these subgoals can result in motions that pivot against shelf walls.

For robot-object contact points, we compute them from a evenly spaced grid on the object primitive surface. This gives 24 points for cuboids and 46 for cylinders.

During execution, the robot approaches the given object contact point, and it proceeds to executing the HFVC after a contact has been detected. The achieved robot-object contact point is most likely different from the contact point in the parameter, because the object bounding box may not accurately capture the underlying object geometry. Sometimes, this mismatch leads to HFVC execution failures, and the precondition function would learn to classify these parameters as such, so the planner can avoid such failures during task execution.

B.2 HFVC Synthesis

Our HFVC synthesis algorithm outputs the hybrid force velocity control commands in the 6D end effector task space. An HFVC command $h \in \mathcal{H}$ include the velocity control directions $T_v \in \mathbb{R}^{n_v \times 6}$ and magnitude $\eta_v \in \mathbb{R}^{n_v}$, and the force control direction $T_f \in \mathbb{R}^{n_f \times 6}$ and magnitude $\eta_f \in \mathbb{R}^{n_f}$. While which direction to perform force or velocity control can be arbitrarily chosen, here we adopt maximum velocity control where $n_v = 5$ and $n_f = 1$.

During the controller construction, our algorithm take as inputs the start(current) object pose ${}^W\mathcal{T}_O$, desired object pose ${}^W\mathcal{T}_{sg}$, partially observed object environment contacts, and the robot object contact(s). The algorithm first reason about the 3D environment contact mode of the desired motion (i.e. which contacts are separating and which contacts are sliding), by comparing the environment contact locations at the current object pose and desired object poses. Our controller will maintain this environment contact mode during execution. For robot object contact(s), we assume they are either a sticking point contact or a sticking small patch contact with the object (can be approximated by over three point contacts). Now we get the contact mode of all the contacts in the system — environment contacts could be sticking, sliding or separating. Robot contacts are always sticking. And we fix this contact mode. A fixed contact mode enables us to safely use continuous constrained dynamics in our following computations. More importantly, by choosing to maintain a contact mode, we don't need to perform contact detection and estimation (which is expensive and slow) during execution.

During the execution, at each timestep, the controller first update the current object pose estimation ${}^W\mathcal{T}_O$. Assuming the object-environment contacts are fixed in the world frame (not moving), we update the environment contact locations and normals in the object frame. There are three steps in computing the hybrid force velocity control parameters:

1. Compute velocity control directions and magnitudes
2. Compute force control directions
3. Compute force control magnitudes

In the first step, we compute the desired contact mode constrained hand velocity $v_h \in \mathbb{R}^6$ (in body twist form). Velocity command v_h is solved with a quadratic programming with which the cost is to get the constrained object velocity v_o as close as possible to the desired object velocity v_o^{des}. v_o^{des} is computed by the desired object pose and current object pose from using the first order Euler integration.

$$
\begin{aligned}
& \min_{v_o, v_h} \|v_o^{\text{des}} - v_o\|_2^2 \\
\text{s.t.} \quad & N \begin{bmatrix} v_o \\ v_h \end{bmatrix} = 0 \text{ (contact mode velocity constraints)} \\
& {}^W\mathcal{T}_{sg} = {}^W\mathcal{T}_O \cdot \exp(v_o^{\text{des}}) \text{ (first order Euler integration of object pose)}
\end{aligned} \tag{1}
$$

We scale this hand velocity to be no larger than some threshold.

In the second step, we determine the force control direction. The force control direction is chosen to be as close as possible to the robot object contact normal direction n_h while being as orthogonal as possible to the desired hand velocity direction v_h by having $T_f = \operatorname{argmin}_{T_f}(\|T_f v_h\| + \|T_f - n_h\|)$. If the robot-object contact normal is parallel to the desired hand velocity direction, we only do velocity control by letting $n_f = 0, n_v = 6$ (this often results in a pushing motion).

In the third step, we solve for the force control magnitude by trying to maintain a small amount of normal contact forces on every non-separating environment contact under quasi-dynamic assumption. Denote the contact forces as $\lambda = \begin{bmatrix}\lambda_e \\ \lambda_h\end{bmatrix}$, where λ_e is the object-environment contact forces and λ_h is the object-hand contact forces. Denote $\eta = \begin{bmatrix}\eta_f \\ \eta_v\end{bmatrix}$, where η_f is the force control magnitude, and η_v are the reaction forces of the hand in the velocity control directions. Here, J_λ^{oT} is the contact jacobian that transforms all contact forces in the wrench space in the object body frame, and $J_{\lambda_h}^{h\ T}$ is the contact jacobian than transforms hand contact forces in the wrench space of the hand body frame. Matrix T describe the directions of force and velocity control axes. Let M_o be the object inertia and a be the quasi-dynamic object acceleration. The optimization problem can be written as follows:

$$\begin{aligned} &\min_{\lambda,\eta} \|\lambda - \lambda^{\text{des}}\|_2^2 \\ \text{s.t.}\quad & FC(\lambda) \leq 0 \text{ (friction cone constraints)} \\ & J_\lambda^{oT}\lambda + F_{external} = M_o a \text{ (quasi-dynamic balance on the object)} \\ & a \in PCC_{v_o} \text{ (object acceleration constraint)} \\ & J_{\lambda_h}^{h\ T}\lambda_h + T^{-1}\eta = 0 \text{ (force balance on the hand)} \end{aligned} \tag{2}$$

Friction cone constraints ensures all the contact forces satisfy the Coulomb Friction law. The object quasi-dynamic balance constraints allowing object to have acceleration a which has the direction in a small polyhedral convex cone aligned with desired object velocity v_o, while ignoring Coriolis forces. The force balance on the hand constraint enables us to compute the force control magnitude η_f, and the passive reaction forces η_v in the velocity control directions.

Our HFVC synthesis algorithm is similar to [51], but we introduce several modifications that make the algorithm feasible for our domain. First, we do not require pre-designed feasible trajectories for the HFVC to follow. These trajectories are typically manually engineered or generated with simulations [11] with ground truth object feedback, which is impractical in our domain. Instead, we interpolate a trajectory from start and goal object pose. While this will result in infeasible object trajectories that would sometimes cause execution failures, our learned precondition function would classify this and help the planner avoid such executions. Second, compared to the quasi-static assumption in [51], we also allow short dynamic motions to allow object dropping. Nevertheless, the quasi-dynamic assumption still prevents us from doing fully dynamic manipulation. However, our models are more limited than the general robot model in [51]. The action space of the robot is just 6D end-effector task space, assuming the end-effector making one sticking point contact or a sticking patch contact. We are also using maximum velocity control while [51] allows the users to choose from maximum or minimum velocity control.

B.3 HFVC execution

	S_v	S_f	D
Simulation	$300I$	I	$diag([10,10,10,1,1,1])$
Real World	$1800I$	$2I$	$diag([10,10,10,1,1,1])$

Table 4: Gains used by low-level controller to follow HFVC targets in simulation and real world.

To execute a given HFVC command h through our torque control scheme, our low-level controller first compute the current desired generalized force, velocity $f_e \in \mathbb{R}^6$ and $v_e \in \mathbb{R}^6$ from h: $f_e = T_f \eta_f$, and $v_e = T_v \eta_v$. These are then used to compute commanded robot torques:

$$\tau = J^\top (S_v v_e + S_f f_e - Dv) \tag{3}$$

where S_v, S_f, and D are diagonal matrices, the first two correspond to velocity and position error gains, and the third is for a damping term to achieve smoother control with v being the generalized end-effector velocity. See Table 4 for values of these matrices. For simplicity, we did not write terms that correspond to compensating for gravity and Coriolis forces.

The HFVC skill terminates if both the position and rotational difference between the estimated object pose and the subgoal pose do not improve for more than 0.1 seconds.

B.4 HFVC Precondition Learning

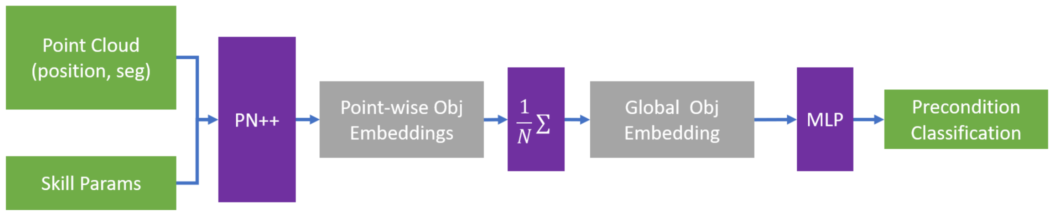
Point Cloud (position, seg)
Skill Params
PN++
Point-wise Obj Embeddings
$\frac{1}{N}\Sigma$
Global Obj Embedding
MLP
Precondition Classification

Figure 7: Precondition learning model architecture

For sampling object scales during data generation, we apply random non-uniform scaling of a range $[0.6, 1.3]$ along each principal axis, with objects that have scales near the 0.6 and 1.3 being used in the test set (10% of the 440k data collected) are in the test set.

During training, we randomly subsample 256 object points and 256 environment points to pass into our network. Because the object tends to be much smaller than the environment, this subsampling ensures the network sees a sufficient amount of object points relative to environment points.

Training the model takes about 20 epochs for validation loss to converge, which takes about 100 hours on an Nvidia A6000.

The following is a list of things we experimented with to improve precondition prediction performance that ultimately had negligible or even negative effects:

1. **Adding point cloud normals as an input feature.** While this had no effect on performance on simulation data, it drastically worsened real-world precondition predictions, because real-world normals, even ones extracted from TSDFs, are very noisy.
2. **Adding point-wise differences to environment points as an additional feature.** This is an $\mathbb{R}^3$ vector that starts from an object point and ends in the closest environment point such that the vector is also within 30° of the point's normal direction. The motivation for this feature is to better capture object-environment contact relationships, but this had negligible effects on learning performance.
3. **Using robust labels.** Instead of labeling a (point cloud, parameter) tuple as positive or negative by the result of running HFVC on that parameter, a robust label is generated by running many HFVC trials on perturbed parameters near the original parameter and labeling the original positive if more than 50% of these noisy executions also satisfy preconditions. The motivation for this is to obtain cleaner and focused data that may be simpler to learn. However, using robust labels had negligible effect on performance, and generating the data for these labels took much longer than just using a single execution.
4. **Adding parameter features to the object embeddings after PointNet++, instead of adding it to each object point before.** This change actually made performance slightly worse.
5. **Using point clouds to represent parameters.** This means having 2 sets of point clouds, both parsed by PointNet++ and their outputs combined to make a prediction. The first set contains the environmental points, the initial object points, and the robot end-effector points placed at the initial robot-object contact point. The second set of points contain the environmental points, the object points where the object is placed at the subgoal, and corresponding end-effector points at the subgoal. The motivation for this representation is to allow the network to more directly parse geometric relationships. However, this change made performance slightly worse and because it needs to process more points, it made training much slower.
6. **Adding contrastive losses.** We experimented with adding contrastive losses to the global object embedding to contrast between object types and parameter types (see Section D.2 how these are defined). These losses had negligible effect on prediction performance.
7. **Dynamics Learning.** We tried training the model to jointly predict preconditions and skill-level dynamics. The motivation for this is that 1) the richer training signal from dynamics losses may

	Ours	Ours RW	No-Params	No-Params RW
Accuracy	79%	69%	65%	44%
Precision	78%	53%	60%	35%
Recall	77%	86%	79%	71%

Table 5: Precondition prediction results with different input representations on real world (RW) data.

improve precondition learning and 2) the learned dynamics prediction may be more accurate than subgoals, leading to better planning performance. For 1), we found predicting dynamics has negligible effect on precondition performance. While our learned dynamics does indeed have much better aggregate next-state prediction errors (ADD 2.3cm, see Figure 12) than directly using subgoals (ADD 3.4cm, see Figure 9), it led to worse planning performance (59.1%, compared to 73.2% achieved by using only precondition model and subgoal dynamics).

We tried many variants of dynamics learning — predicting 6D poses ($\mathrm{SE}(3)$), predicting 3D poses ($\mathrm{SE}(2)$) from the subgoal (the reached pose is in the same stable pose as the subgoal pose, otherwise the preconditions would not be satisfied), computing dynamics loss from pose predictions, and computing dynamics loss from the Average Discrepancy Distance (ADD, this is the mean distance between corresponding points of two point clouds). While some variants performed better than others, they were not able to result in better planning performance.

Please see more detailed results in Section D.2, D.4, and D.5.

B.5 HFVC Precondition Sim-to-Real Gap

To characterize the sim-to-real gap of the learned precondition function, We performed 61 skill executions across the real world objects in Figure 4 and recorded the prediction results in Table 5. We show both the numbers for our approach and No-Params in simulation (from Table 2) and their performance on the real-world dataset (RW). There is a 10% accuracy drop for our approach and a much larger 21% drop for No-Params. Most of the accuracy drop came from reduced precision, meaning there are many samples where the model predicted were successful but in reality were not. Recall actually increased for our approach, meaning a higher proportion of actual positives were predicted to be positive. We note this is generally an acceptable trade-off for planning, as our replanning scheme allows recovering from false positive errors, but having high false negatives would prevent the planner from finding feasible solutions when they exist. Lastly, we note that due to the degraded performance of the visual perception module on real-world data, the real-world precondition labels, which rely on real-world object pose estimates, are likely to be noisy, so we expect our models to actually perform better than these numbers would suggest.

C Planner Details

We use a planning algorithm similar to Real-time A* but with a few modifications that make the planner greedier but faster. The planner has a planning budget of 30 seconds and expands at most 10 nodes with a maximum search depth of 3. While we only expand at most 10 nodes, this actually requires generating many more successors because of the high branching factor (can be more than 100). To improve planning speed, we prune parameters that reach the same subgoal pose by only keeping one of them (there are many parameters that reach the same subgoal pose via different robot-object contact points). This choice is made by sampling a probability distribution that is proportional to their predicted precondition satisfaction probabilities, where weights are computed via softmax. In addition, we also terminate the search after a path to goal is found, instead of waiting until the goal state is removed from the open list. This is because due to inaccurate transition models, we are not confident in the actual cost improvements of other plans, and we prefer to executing the first action then replan as needed. While our algorithm is coded as a graph search problem, in practice it more closely resembles a tree search, because only a very rare occasions do two generated successor states coincide to be the same state (have the same object pose).

The goal pose is satisfied if the object reaches within 1.5cm and 10° tolerance. The cost of each planning edge is $c = d + 0.5\theta$, where d is the distance between the initial and reached object poses, and θ the angle. To guide the search, we use the heuristic $pD({}^W\mathcal{T}_O, {}^W\mathcal{T}_g) + (1-p)1$, where p is the prediction precondition satisfaction probability of the action that corresponds to the edge before the current state,

${}^{W}\mathcal{T}_O$ is the object pose of the current planning state, and ${}^{W}\mathcal{T}_g$ the goal pose. Here, D computes the distance between two poses the same way the cost is computed above. The $(1-p)1$ term can be thought of as applying a penalty of 1 if the skill fails. This heuristic is inadmissible, but we found it works well in practice when coupled with replanning.

D Experiment Details

D.1 Task Domain Setup

The Franka Arm gripper is outfitted with finger extensions — this helps the robot to reach farther into the shelf without incurring collisions. The finger extensions are covered with Plasti Dip to increase friction. From both simulated and real-world robot arms we receive feedback on end-effector poses, velocities, and external force-torques. To obtain point clouds of the scene, in simulation we directly fuse together deprojected depth images from 3 simulated cameras whose combined viewpoints cover the entire shelf area. In the real-world, this is done by capturing a series of depth images from a wrist-mounted RealSense D415 camera, and a final point cloud is produced via TSDF Fusion with Open3D. See Figure 4 for an illustration.

Pick-and-Place Skill. In order to more efficiently complete the task, we also use a Pick-and-Place skill that can grasp and move objects to subgoal poses. The parameter for the Pick-and-Place skill contains two elements — an initial antipodal grasp pose for the robot and a final placement pose for the object. Grasp poses are generated from the estimated object primitives, and placement poses are sampled in a grid on the shelf surface. We additionally sample neighboring stable poses and planar rotations for object placement poses. These parameters are filtered by collision checking — the robot should not collide with the environment and the grasp and placement poses, and the object should not collide with the environment at the placement pose.

D.2 Data Generation Results

In Figures 8 we plot precondition success ratios and the mean subgoal pose error ADDs; in Figure 9 we plot the distribution of subgoal pose error ADDs. Pose errors are only computed over executions that satisfy preconditions. In both figures, we separate the aggregate statistics into different data segments. One is across all objects, one is across the 8 YCB objects we used, one is across parameter types, and one is across whether or not the initial state is close to a shelf wall. Parameter type is denoted by concatenating an adjective (front, top, side) and a verb (slide, push, pivot, topple). The adjective refers to the position of the initial robot-object contact point, whether it is in front of the object (toward the shelf opening), top of the object, or the side of the object. The verb refers to the motion achieved by the combination of the robot-object contact point and the desired subgoal pose. For slide and push, the subgoal pose only differs from the initial pose via translation, while pivot and topple have rotation components. The difference between slide and push is that comparing to the delta pose direction, the robot-object contact normal is more orthogonal for slide than it is for push. The difference between pivot and topple is that pivot does not change the stable pose, while topple does. We compute these parameter types only for debugging and visualization purposes — they are not used by the precondition model.

D.3 Precondition Learning Results

In Figure 10 we plot precondition performance curves (True Positive Rate vs. True Negative Rate, Precision vs. Recall), and in Figure 11 we further breakdown Precision and Recall numbers into different data segments, similar to the data generation results above.

D.4 Dynamics Learning Results

For dynamics learning, we train a model to jointly predict next object pose and the preconditions by adding a new output head from the global object embedding. To represent the next pose prediction, our model predicts a planar 3D delta pose (x-y translation and z-axis rotation) that when applied to the subgoal pose, should give the actual reached pose. In experiments this yielded better prediction performance than directly

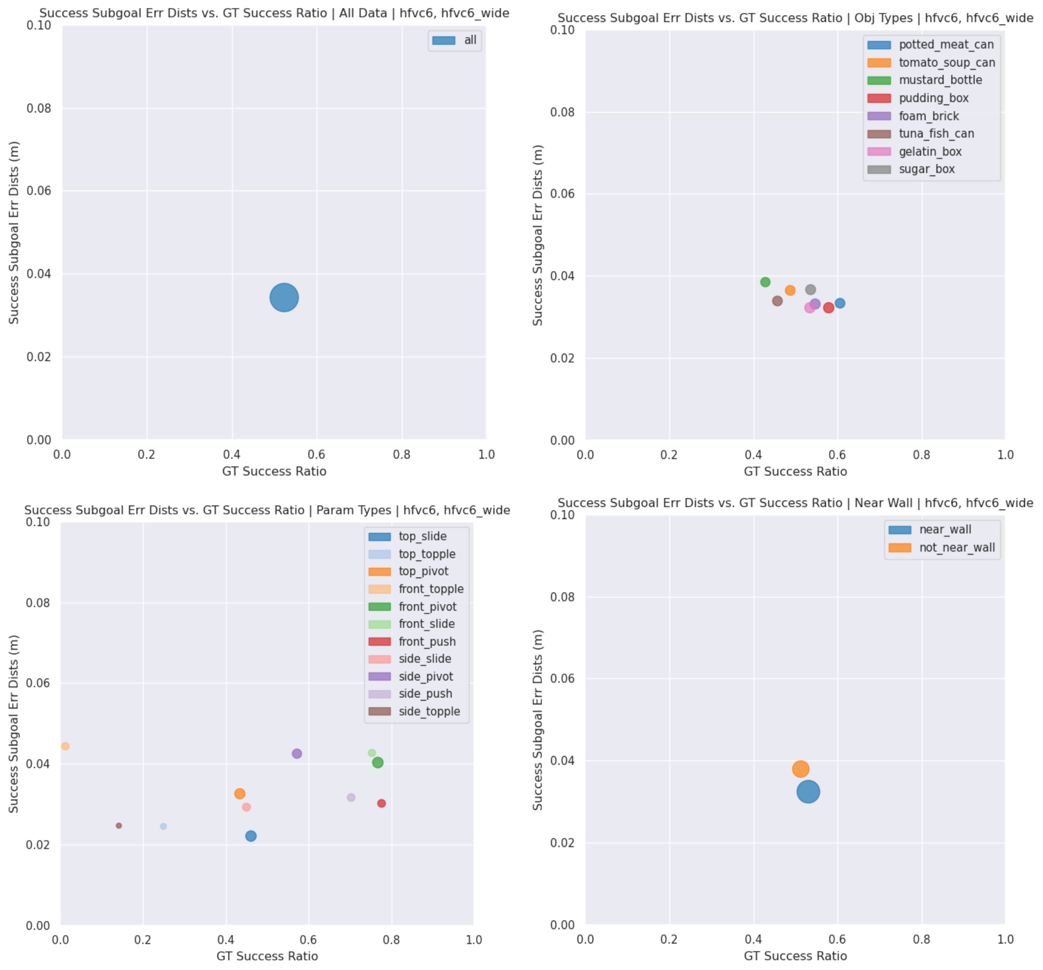

Figure 8: Precondition data generation statistics across different data segments. X-axis is the precondition success ratio. Y-axis is the ADD between the subgoal pose and the actual reached pose after skill execution; this is only aggregated over executions that satisfy preconditions. The area of each circle corresponds to the ratio of data size — the number of data points in for that type of data divided by the the number of all data points. The four plots shows the these statistics across different data segments. Top left: all data. Top right: separated by object type. Bottom left: separated by parameter type. Bottom right: separated by whether or not the initial pose was near a shelf wall.

predicting the 6D delta pose from the initial pose. The dynamics loss is represented by the ADD between the object points at the predicted pose than at the actual pose. This gave better prediction performance than directly regressing to the ground truth delta pose values. Dynamics loss is only back-propagated for data points that have positive precondition labels. See dynamics prediction results in Figure 12, where we plot distributions of predicted object pose errors over executions that satisfy ground truth preconditions. Note that this distribution has lower mean errors and less outliers than directly using subgoals (Figure 9).

D.5 Planning Results

Planning with learned dynamics. We report planning results of using a model that jointly predicts skill preconditions and dynamics (Dyn) in Table 6. This achieves a success rate of 59.1%, which is lower than our method that uses only the learned preconditions with subgoals (73.2%), but it is comparable to Est-Primitive (61.1%). Note that this method has a significantly longer total plan time (mean 89.1s, as opposed to the mean of 43.1s for our method), which is indicative of needing to replan more frequently. In Figure 13 (explained in more detail below), the most dramatic difference between our approach and Dyn is for tasks where both the initial and goal object poses are near shelf walls. This points to the model's inability

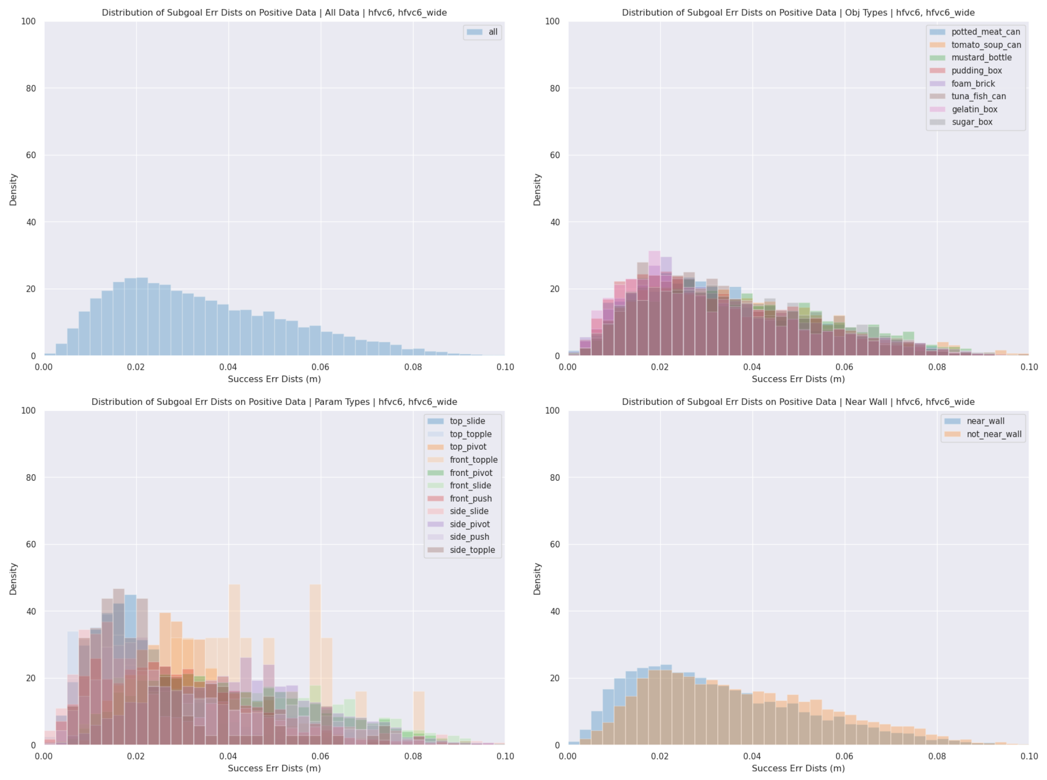

Figure 9: Subgoal dynamics error distributions across different data segments. We plot histograms of the ADD between subgoal and actual reached pose after skill execution; this is only aggregated over executions that satisfy preconditions. The four plots shows the these statistics across different data segments. Top left: all data. Top right: separated by object type. Bottom left: separated by parameter type. Bottom right: separated by whether or not the initial pose was near a shelf wall.

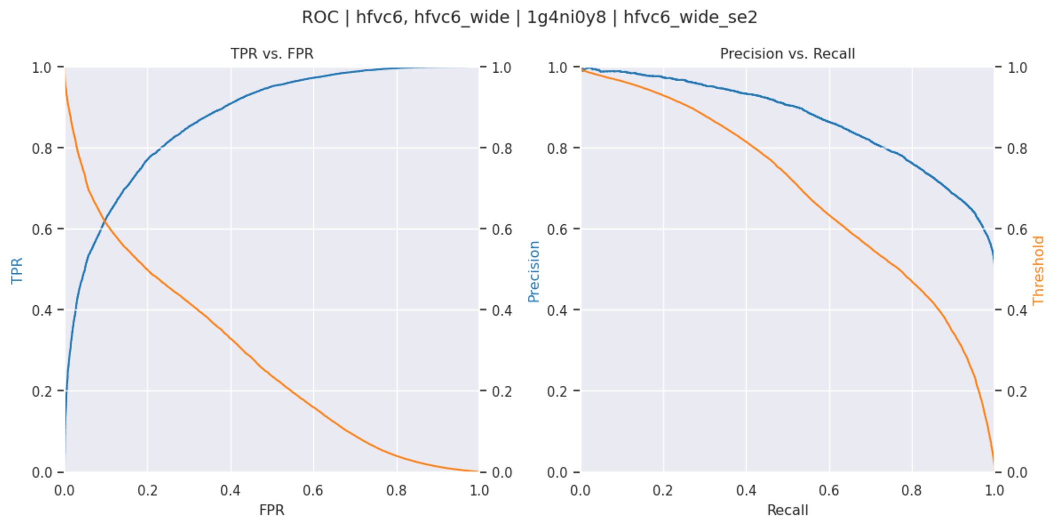

Figure 10: Precondition performance curves. Left: True Positive Rate vs. True Negative Rate. Right: Precision vs. Recall. Points on the threshold curves correspond to the threshold used for the corresponding point (same x-axis location) on the blue curves, where the threshold is the y-axis value on the right.

to sufficiently take into account object-environment interactions to accurately predict skill dynamics in these scenarios.

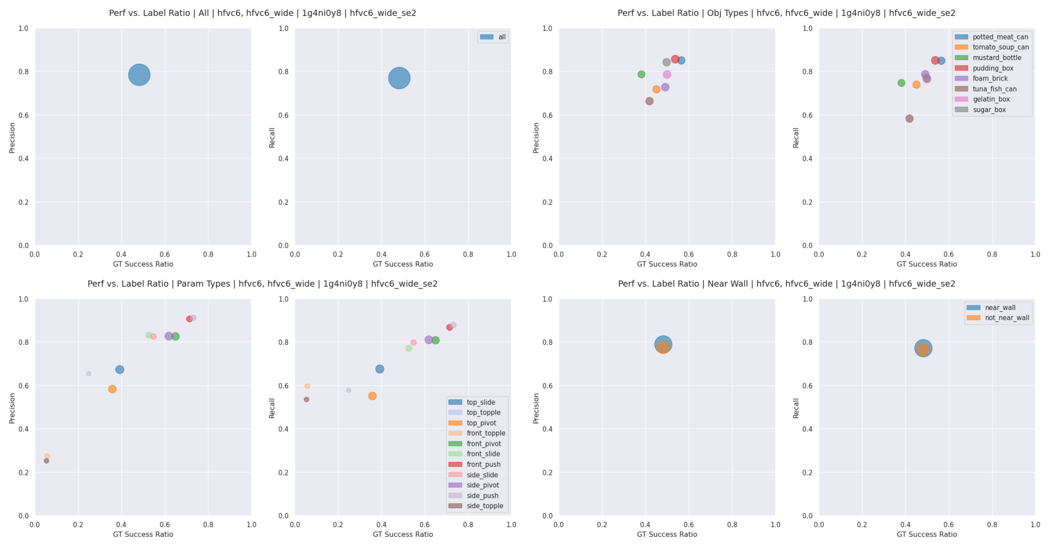

Figure 11: Precondition precision, recall vs. ground truth precondition success ratio across data types. X-axis is the ground truth precondition success ratio. Y-axis plots precision and recall. The area of each circle corresponds to the ratio of data size — the number of data points in for that type of data divided by the the number of all data points. The four plots shows the these statistics across different data segments. Top left: all data. Top right: separated by object type. Bottom left: separated by parameter type. Bottom right: separated by whether or not the initial pose was near a shelf wall.

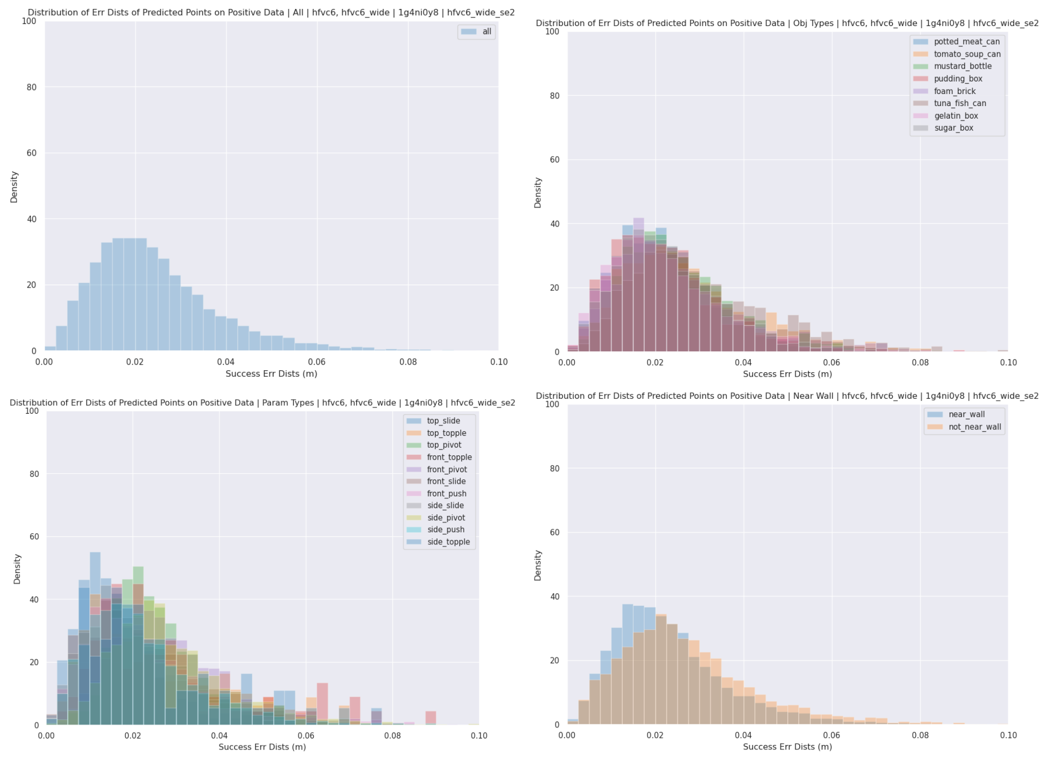

Figure 12: Predicted dynamics error distributions across different data segments. We plot histograms of the ADD between subgoal and actual reached pose after skill execution; this is only aggregated over executions that satisfy preconditions. The four plots shows the these statistics across different data segments. Top left: all data. Top right: separated by object type. Bottom left: separated by parameter type. Bottom right: separated by whether or not the initial pose was near a shelf wall.

	Plan Success	Plan Time (s)	Plan Length
Dyn	59.1%	89.1±60.9	2.6±1.4

Table 6: Planning performance of using skill model that jointly predicts skill preconditions and dynamics.

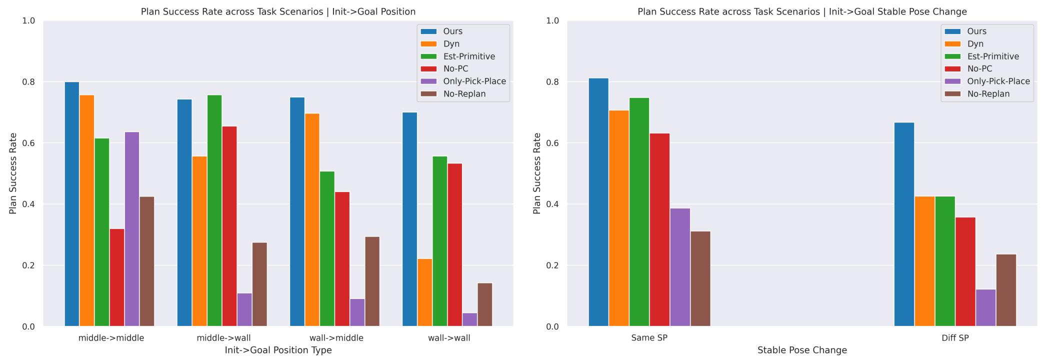

Figure 13: Task success rate for ablations across different task scenarios. Left: initial to goal object position types. Right: whether or not initial and object poses are in stable pose configurations

Planning performance across task scenarios. To better understand the differences among the 4 variants we compared in our main planning result, we plot success rates for all methods across different task scenarios in Figure 13. We report two types task scenario comparisons, one on whether the initial and goal object poses are in the middle of the shelf or near the shelf walls, and another one on whether these two poses are in the same stable pose configurations. For init→goal position types, No-PC has the worst performance in middle→middle. This is because many HFVC motions, like pushing and sliding, are infeasible or not robust when the object is not close environmental constraints, so the planner is overly optimistic on the types of plans it can achieve. Only-Pick-Place is able to find many successful plans for middle→middle, but less so for other scenarios. This is expected as most object poses near shelf walls are not graspable. No-Replan has much worse performance than our approach across the board, indicating that subgoal poses are not accurate enough to use as a transition model for multi-step planning. While all methods perform worse when the stable poses change, our method is still able to complete over 60% of the tasks, while the best alternative (Est-Primitive) is in the low 40%s.

D.6 Example Plans

Figures 14 and 15 show two example plan executions in simulation. In both cases, the object starts and ends near the shelf walls, so the robot needs to use HFVC skills to manipulate the object absence of available grasps. Figures 16, 17, 18, and 19 have example plan executions in the real world. For videos and additional plans, please see the website `https://sites.google.com/view/constrained-manipulation/`.

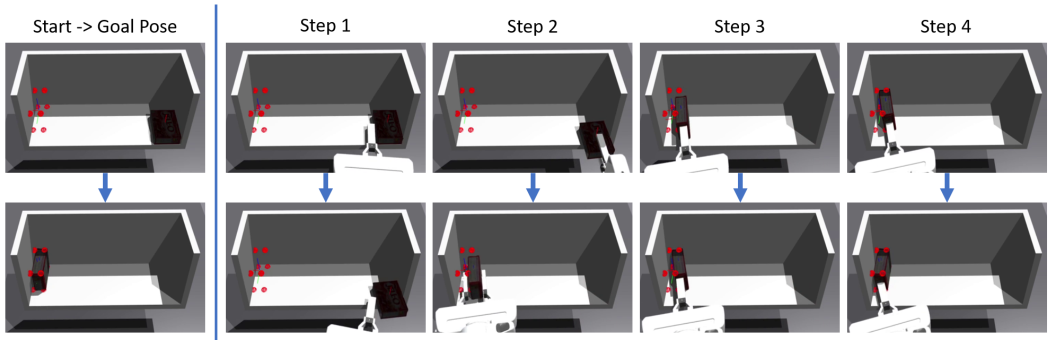

Figure 14: Example plan execution in simulation for the gelatin box. Left column is the start and goal state (note they are in different stable poses). The red circles indicate the intended goal pose, while the object on the bottom pictures is the reached final pose. Right columns are the series of HFVC and Pick-and-Place skills the planner found to complete the task. Note the use of HFVC skills in the beginning and in the end when the object grasps are occluded by shelf walls.

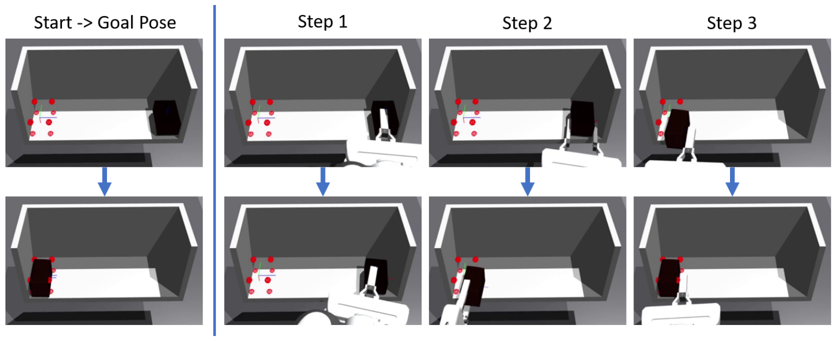

Figure 15: Example plan execution in simulation for the foam brick. Left column contains the start and goal states (note they are in different stable poses). The red circles indicate the intended goal pose, while the object on the bottom pictures is the reached final pose. Right columns are the series of HFVC and Pick-and-Place skills the planner found to complete the task. Note the use of HFVC skills in the beginning and in the end when the object grasps are occluded by shelf walls.

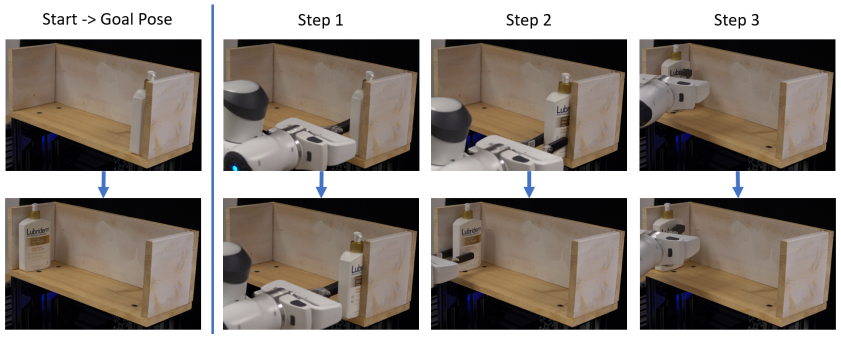

Figure 16: Example plan execution in the real world. Left column contains the start and goal states (note they are in different stable poses). Right columns are the series of HFVC and Pick-and-Place skills the planner found to complete the task.

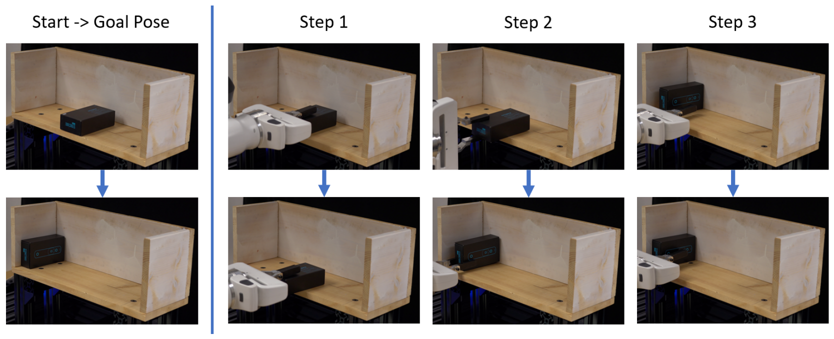

Figure 17: Example plan execution in the real world. Left column contains the start and goal states (note they are in different stable poses). Right columns are the series of HFVC and Pick-and-Place skills the planner found to complete the task.

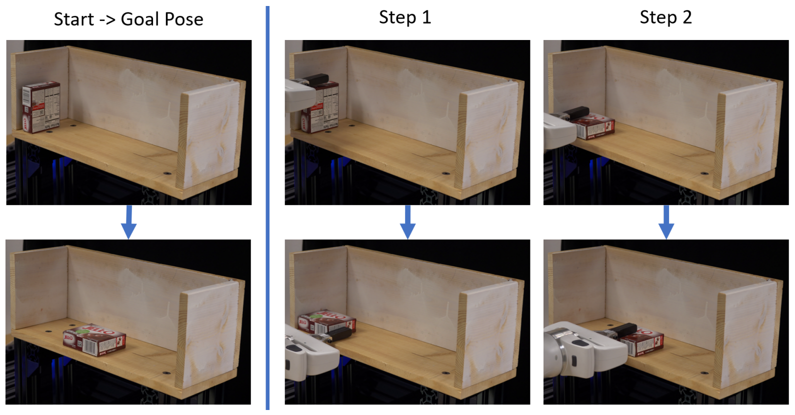

Figure 18: Example plan execution in the real world. Left column contains the start and goal states (note they are in different stable poses). Right columns are the series of HFVC and Pick-and-Place skills the planner found to complete the task.

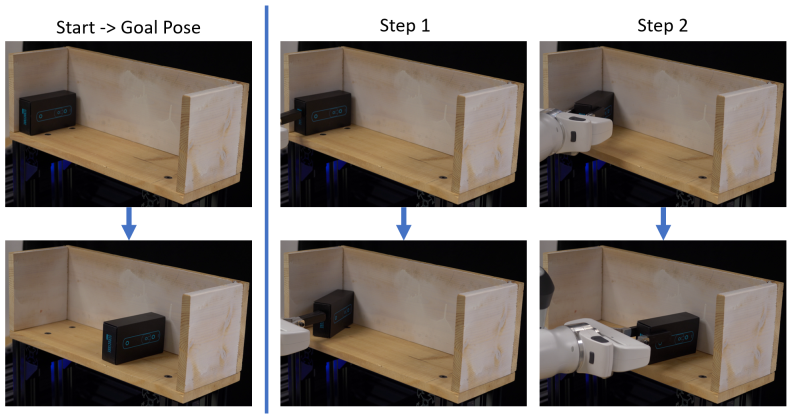

Figure 19: Example plan execution in the real world. Left column contains the start and goal states (note they are in different stable poses). Right columns are the series of HFVC and Pick-and-Place skills the planner found to complete the task.