Appendix

A Implementation Details

A.0.1 Skill Embedding and Skill Prior Learning

In this section, we provide a detailed overview of the network architectures used to learn the two different skill modules in this work. The training dataset consists of sequences of state-action pairs (skills) collected by running the robot in the environment. We detail the data collection process in Appendix E. For the Fetch push, cleanup and stacking tasks, the state information consisted of a 19-dimensional vector that describes the configuration of the robot and the pose of the red block in the environment. For the hook task, we additionally concatenate the pose of the hook and its velocity to give a resulting state vector of 43-dimensions. The action space for all tasks consists of a 4-dimensional vector that encapsulates the agent's end-effector pose in 3-dimensional space and the gripper position. The skill-horizon H was set as 10 across all environments.

We parameterise both the VAE skill embedding and normalising flows skill prior modules as deep neural networks and detail their implementation below.

VAE embedding module The VAE encoder first processes individual concatenated state-action pairs in sequence using a one-layer LSTM with 128 hidden units. The hidden layer after processing the final observation is then fed into an MLP block comprised of 3 linear layers with batch normalisation and ReLU activation units with two output heads over the last layer, yielding parameters μ_z, σ_z of the variational posterior $\log q_\phi(z \mid \boldsymbol{s}, \boldsymbol{a}) \sim \mathcal{N}(\mu_z, \sigma_z)$. We set our latent space to be 4-dimensional, i.e. $\mathcal{Z} \cong \mathbb{R}^4$. The decoder network mirrors the architecture of the MLP block used in the encoder, taking as input concatenated latent skill vector z and current state s_t. The final layer has a single head and a `tanh` layer to ensure actions are bounded between $(-1, 1)$. The full action sequence ***a'*** is decoded sequentially, with observed states s_t given to the encoder at each step. The overall loss function for the embedding module is as provided in (1), noting the expectation is computed by drawing a single sample from posterior $q_\phi(z \mid \boldsymbol{s}, \boldsymbol{a})$. Furthermore, the VAE reconstruction loss term is the mean-squared error, i.e. $\log p_\theta (a_t \mid z, s_t) \propto (a_t - a'_t)^2$.

State-conditioned skill prior sampling module The network parameterising the skill prior $f : \mathcal{Z} \times \mathcal{S} \to G$ is a conditional real NVP [27] which consists of four affine coupling layers, where each coupling layer takes as input the output of the previous coupling layer, and the robot state vector s_0 from the start of the skill sequence. We use a standard Gaussian $p_\mathcal{G}(g) \sim \mathcal{N}(0, I)$ as our base distribution for our generative model. Our architecture for f is identical to the conditional Real NVP network used by Singh et al. [4], see Singh et al. [4] for an in-depth explanation. The loss function for a single example is given by

$$\mathcal{L}_{prior} = \log p_\mathcal{G}(f(z, s_0)) + \log \left| \det \frac{\partial f}{\partial z^\top} \right| . \tag{2}$$

During training, we pass the initial state vector of the robot before applying the skill as conditioning information. We optimise our model using the Adam optimiser with a learning rate of 1e-4 and batch size of 128. The overall objective for training the skills model for a single training observation is given as

$$\mathcal{L}_{skills} = \mathcal{L}_{embed} + \mathcal{L}_{prior}, \tag{3}$$

noting that gradients of the skills prior loss w.r.t. z, i.e. $\partial \mathcal{L}_{prior}/\partial z$ are blocked. What this means is that the VAE embedding module is trained without being influenced by the skills prior objective, however, the skills prior training is affected because the topology of latent space determined by the embedding module is evolving during training. We found this joint training yielded more expressive skill priors compared to first training on $\mathcal{L}_{embed}$ and subsequently training the skills prior on $\mathcal{L}_{prior}$ after the embedding module has finished training.

A.0.2 Reinforcement Learning Setup

We jointly train the high-level and low-level policy in a hierarchical manner, where the high-level policy leverages the resulting cumulative reward after a complete skill execution, while the residual component is updated using the reward information generated after every environment step. By jointly training these two systems, we make effective use of all the online experience collected by the agent to update both the high-level and low-level policies, allowing for better overall sample efficiency of downstream learning.

We utilise Proximal Policy Optimisation [33] as the underlying RL algorithm for both the high and low-level policies, using the standard hyper-parameters given in the SpinningUp implementation [35] with a clip ratio of 0.2, policy learning rate of 0.0003 and discount factor γ = 0.99. We found it important to train the high-level policy without the residual for an initial 20k steps before allowing the residual policy to modify the underlying skills. This allowed the high-level policy to experience the useful skills suggested by the skill-prior without being distorted by the random initial outputs of the residual policy. As opposed to a hard introduction of the residual action, we gradually introduce it by weighting this additive component using a smooth gating function which increases from 0 to 1 over the course of the first 20k steps. We found that this stabilised the training of the hierarchical agent. While there are various ways to schedule this weighting parameter w, we utilised the logistic function - with centre C at 10k steps and a growth rate k of 0.0003.

$$w = \frac{1}{1 + e^{-k(x-C)}} \tag{4}$$

B Tasks

We describe the downstream RL evaluation tasks in detail below. Note that each of the task environments exhibit dynamical and physical variations from the data collection environment used for skill acquisition.

Slippery Push The agent is required to push a block to a given goal, however, we lowered the friction of the table surface from that seen in the data collection push task. This makes fine-grained control of the block more difficult. The agent receives a reward of 1 only once the block is at the goal location, otherwise, it receives a reward of 0. The task is episodic and terminates after 100 timesteps.

Table Cleanup We introduce a rigid tray object in the scene, into which the agent must place the given block. The tray was not present in the data collection environment and the edges act as an obstacle that the downstream agent must overcome with its available skills. The agent receives a reward of 1 only once the block is placed in the tray, otherwise, it receives a reward of 0. The task is episodic and terminates after 50 timesteps.

Pyramid Stack Stacking task where the agent is required to place a small red block on top of a larger blue block. The prior controllers used for data collection are unable to move the gripper above the height of the larger block. Additionally, this task requires precise placement of the red block. The agent receives a reward of 1 only once the red block is successfully balanced on top of the blue block, otherwise, it receives a reward of 0. The task is episodic and terminates after 50 timesteps.

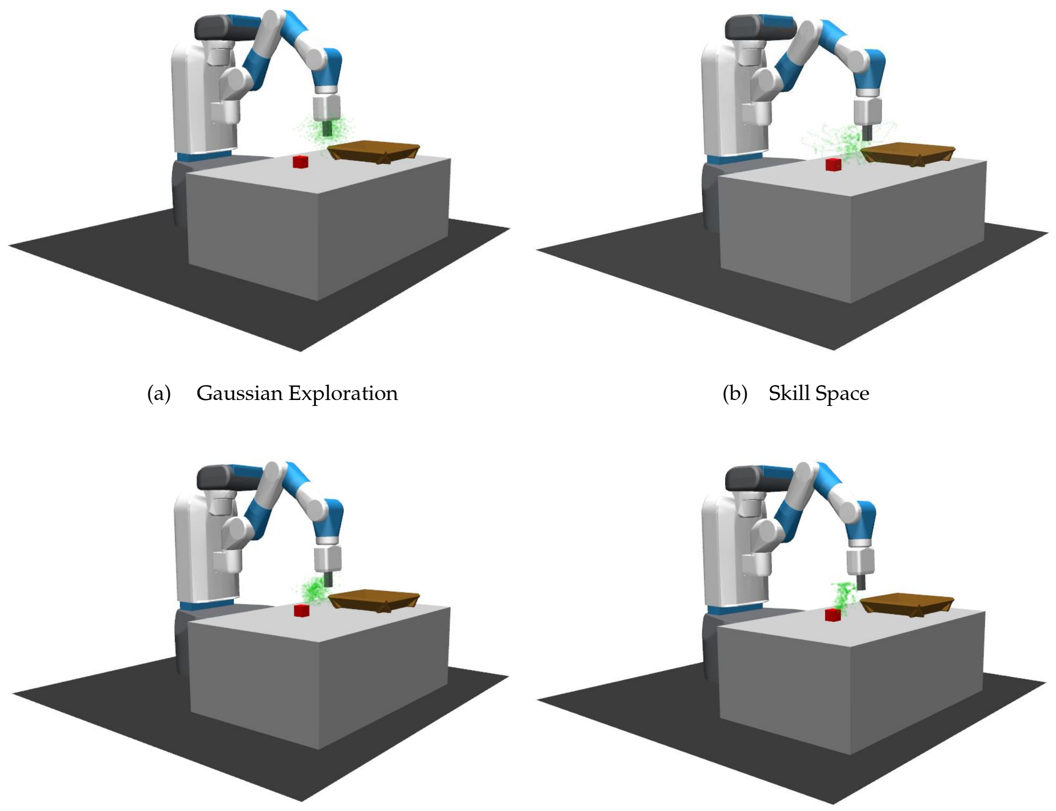
Complex Hook The agent is required to move an object to a target location, however, the robot cannot directly reach the object with its gripper. We introduce a hook that the agent must use to manipulate the object. To add to the complexity of this task, the objects are drawn randomly from an unseen dataset of random objects and the table surface is scattered with random ”bumps” that act as rigid obstacles when manipulating the object. The agent receives a reward of 1 only once the block is at the goal location, otherwise, it receives a reward of 0. The task is episodic and terminates after 100 timesteps.

C Skill Prior Ablation

In this section, we analyse the impact that our proposed skill prior has on the exploratory behaviours of the agent during the early stages of training. For a quantitative evaluation, we denote the percentage of the first 20k steps that result in some form of manipulation of objects placed in the scene. The intuition here is that for manipulation-based tasks, ”meaningful” behaviours would involve manipulation of the objects in the scene in order for the agent to make progress towards solving the task at hand, as opposed to the extensive random exploration of irrelevant, non-manipulation-based behaviours. We evaluate our skill prior based exploration strategy to approaches used in existing

Table 1: Skill prior impact on exploration. The table denotes the proportion of exploratory steps that result in some form of manipulation of relevant objects in the environment during the first 20k steps.

	Skill Prior (Ours)	Skill Space	Behaviour Prior	Gaussian Exploration
Object Interaction	**45.4%**	9.39%	4.72%	0.560%



(a) Gaussian Exploration (b) Skill Space

(c) Behaviour Prior (d) Skill Prior (Ours)

Figure 7: **Exploratory Trajectories** We plot the trajectories taken by four different strategies used in skill-based and single-step RL approaches. Note how the skill prior significantly directs the exploratory trajectories towards the object in the environment while still allowing the agent to explore a diverse set of surrounding skills.

RL literature for both skill-based and standard single-step agents. Table 1 summarises the results. We additionally provide a visual depiction of the trajectories taken by each exploration strategy in Figure 7 to better understand how the skill prior impacts exploration.

As indicated by the results, our NVP-based skill prior attains the highest proportion (>45%) of meaningful exploratory behaviours and we can attribute this to the ability of the skill prior to sample relevant skills that the agent can directly execute in the environment. We note here that while this prior does heavily bias the agent's behaviours towards manipulating the block, it does not completely constrain the agent's ability to explore other skills that could be potentially better for the downstream task. The Skill Space variant tested in this ablation encompasses skill-based algorithms that sample behaviours from the skill space using the stochastic policy output $\pi_{HL}(z|s)$. This is reminiscent of the strategy used in SPiRL [3], which additionally regularises this distribution towards a learned prior during training. This strategy achieved only 9.39% of relevant behaviours which we could use to explain the discrepancy in the asymptotic performance of SPiRL and ReSkill (No Residual). We note here that this value would vary based on the strength of the regularisation towards the prior,

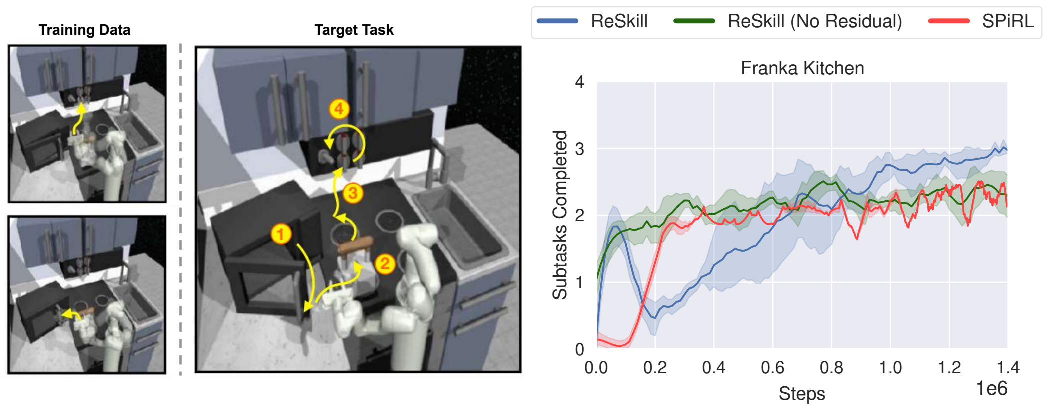


Figure 8: **Evaluation on Franka Kitchen Environment.** (**Left**) Franka Kitchen environment proposed by Gupta *et al.* [36] which requires the agent to manipulate a kitchen setup to reach a target configuration. (**Right**) Note how ReSkill exhibits faster convergence to a higher reward than SPiRL.

and should increase as the policy output gets better regularised towards the prior over the course of training. The Behaviour Prior tested in this work is the single-step prior proposed by Singh *et al.* [4] and while it shares a similar operation to our skill prior we note it attains much lower interaction than our skill-based variant. We could attribute this to the temporal nature of skills, which are less noisy than constantly sampling single-step action allowing for more directed exploration towards the block. Figure 7 (c) and (d) provides a visual depiction of this. Gaussian exploration has almost no interaction with the block, which significantly impacts the agent's ability to learn as shown in the training curves for SAC, PPO and HAC in Figure 5.

D Evaluation on Long Horizon Manipulation Tasks

To demonstrate the broad applicability of ReSkill to higher dimensional tasks involving long-horizon goals, we provide an additional evaluation in the Franka Kitchen environment introduced by Gupta *et al.* [36]. To train the skills modules, we use the demonstration data provided in the D4RL benchmark [37], which consists of 400 teleoperated sequences in which a 7-DoF Franka robot arm manipulates objects in the scene (e.g. switch on the stove, open microwave, slide cabinet door). During downstream learning, the agent has to execute an unseen sequence of multiple sub-tasks. The agent receives a sparse, binary reward for each successfully completed sub-task. The action space of the agent consists of a 7-dimensional vector that corresponds to each of the robot joints as well as a 2-dimensional continuous gripper opening/closing action. The state space consists of a 60-dimensional vector that consists of the agent's joint velocities as well as the poses of the manipulable objects. We summarise the results in Figure 8.

The training curves illustrate that ReSkill can effectively handle higher dimensional and long-horizon tasks, and can substantially outperform SPiRL in both sample efficiency and convergence to higher final policy performance. We note the high variance of ReSkill during the gradual introduction of the residual, which we can attribute to the long-horizon nature of the task and the multiple strategies that the low-level agent can identify to adapt to the task at hand. It is important to note however that this agent gradually converges towards a stable solution that can yield a much higher reward after this transition phase. Without the residual, the ReSkill agent attains the same final performance as SPiRL however still demonstrates faster convergence given the direct exploratory guidance provided by our proposed skill prior.

E Data Collection

In order to re-purpose existing controllers for a wide range of tasks, we decompose their behaviours into task-agnostic skills. We firstly collect a dataset of demonstration trajectories, consisting of state-action pairs by executing the handcrafted controllers in the data collection environment. For the block manipulation tasks, we script 2 simple controllers for a 7-DoF robotic arm each capable of completing *pushing* and *grasping* manipulation tasks on an empty table as described in Algorithm 2 and 3. This dataset was used to train a single skills module that was later used for downstream RL learning across *Slippery-Push*, *Table-Cleanup* and the *Pyramid-Stacking* task. For the hook task, we scripted a simple controller that can manipulate a block using a hook object on an empty table which we used to collect the required dataset. We note here that these controllers are suboptimal with respect to the downstream tasks which each introduce additional complexities to the environment that the RL agent will have to adapt to as described in Appendix B. To increase the diversity of skills collected, we add Perlin noise [38] to the controller outputs. Perlin noise is correlated across the trajectory, allowing for smooth deviations in trajectory space. Before adding these trajectories to the dataset, we filter them based on a predefined rule: if it is longer than the skill horizon H we add it to our dataset. Skills can be extracted from a trajectory, by randomly slicing a H dimensional skill consisting of a sequence of actions $\boldsymbol{a} = \{a_t, ..., a_{t+H-1}\}$ and the corresponding states $\boldsymbol{s} = \{s_t, ..., s_{t+H-1}\}$ that these actions were executed in. For all experiments conducted in this work, we set the skill horizon H to 10. Figure 9 shows a summary of the data collection process and the notion of a skill from a recorded trajectory. We collect a total of 40k trajectories to train the skills module for the Fetch manipulation tasks.

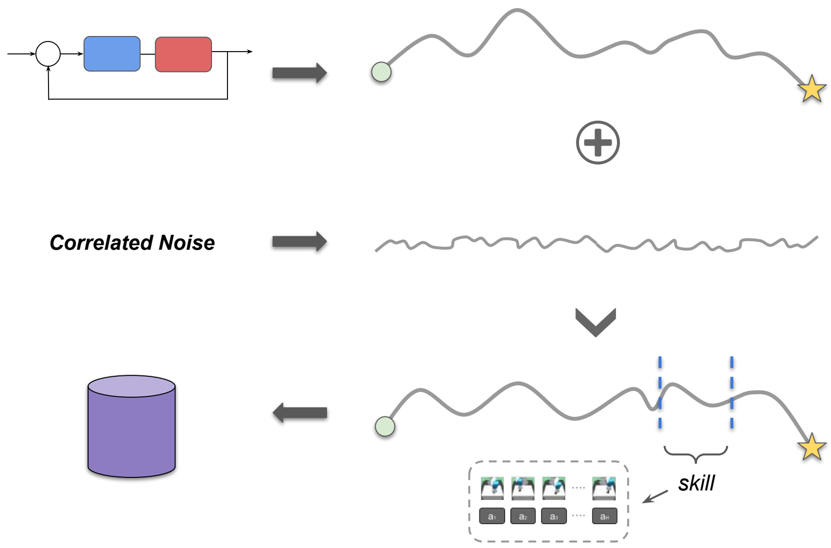


Figure 9: **Data collection.** Skill extraction from trajectories collected from a handcrafted controller. To obtain a diverse range of skills we add correlated Perlin noise to the trajectory rollouts before skill extraction.

E.1 Handcrafted Controllers

We describe each of the controllers derived to complete simple tasks on an empty table with a single object placed in front of it as shown in Figure 4 (a). This was the environment used for collecting data for training the skill modules.

E.1.1 Reactive Push Controller

This controller is designed for pushing an object to a target location. Although this policy performs well in the original Push task, its performance drops dramatically when the sliding friction on the block is reduced as in the *SlipperyPush* task. Pseudo-code for the controller is provided in Algorithm 2.

Algorithm 2 Reactive Push

```
Given: threshold
Input: state
Output: action
1: if distance(targetObjectPose, objectPose) < threshold then
       action ← 0
2: else if gripper_at_pushloc(gripperPose, pushLoc) then
       action ← push object to target
3: else
       action ← move gripper to push location
4: end if
5: return action
```

E.1.2 Pick and Place Controller

This controller is designed to move to an object location, grasp the object and move it towards a target goal. The controller does not lift the block higher than 3cm above the surface of the table and therefore will fail if either the object has to be placed on a higher surface or alternatively, the target is on the other side of a barrier. These failure cases are present in both the *Pyramid-Stack* and *Table-Cleanup* tasks. Pseudo-code for the controller is provided in Algorithm 3.

Algorithm 3 Pick and Place

```
Given: threshold
Input: state
Output: action
1: if distance(targetObjectPose, objectPose) < threshold then
       action ← 0
2: else if is_grasped(gripperPose, objectPose) then
       action ← move to target
3: else if object_in_gripper(gripperPose, objectPose) then
       action ← close gripper
4: else if gripper_above_object(gripperPose, objectPose) then
       action ← move gripper down
5: else
       action ← move above object
6: end if
7: return action
```

E.1.3 Reactive Hook Controller

This controller is designed to pick up a hook, move it behind and to the right of an object and pull the object towards a target location. The controller and environment are adapted from the set of tasks presented by Silver *et al.* [31]. The controller works well when the table is empty and when the object being manipulated is a simple block. In the *Complex-Hook* environment, however, this controller falls suboptimal as we introduce additional objects of varying masses and shapes as well as rigid "bumps" which make the surface of the table uneven. This causes the hook to get stuck when sliding objects along the table. Pseudo-code for the controller is provided in Algorithm 4.

Algorithm 4 Reactive Hook

```
Given: threshold
Input: state
Output: action
1: if distance(targetObjectPose, objectPose) $<$ threshold then
       action $\leftarrow$ 0
2: else if hook_is_not_grasped(gripperPose, hookPose) then
       action $\leftarrow$ grasp hook
3: else if hook_in_position(hookPose, objectPose) then
       action $\leftarrow$ place hook to the right of object
4: else
       action $\leftarrow$ move object to target
5: end if
6: return action
```