A Code and Further Resources

We provide complementary code in the supplementary of the paper. All models were implemented in Python using the PyTorch library. The total training computation required for optimizing all models and baselines presented in this works amounts to roughly 150 hours (wall clock) on a single standard GPU. Visualization and additional material will be uploaded here https://sites.google.com/view/velap-corl/home.

B Hyperparameters and algorithm details

Here we present a description of the hyperparameter used in method used during the training and inference phases.

B.1 Model architectures

Table 2: Hyperparameters of the encoder ϕ

Parameter	Value
Batch-norm.	yes
Filters	[32,32,64,64]
Kernels	[4,4,4,4]
Strides	[2,2,2,2]
Activation	LeakyRelu
Dense layers	[256, 128, 32]

Table 3: Hyperparameters of dynamics model h

Parameter	Value
Batch-norm.	yes
Activation	LeakyRelu
Dense layers	[128,128,128,128,32]

Table 4: Hyperparameters of action sampler g (β-VAE)

Parameter	Value
Batch-norm.	yes
Activation	LeakyRelu
Latent dimension	16
β (kl-weight)	0.01
Encoder dense layers	[128,128,128, 2*16]
Decoder dense layers	[128,128,128, d_{action},]

Table 5: Hyperparameters of policy networks π^l and π^g

Parameter	Value
Batch-norm.	yes
Activation	LeakyRelu
Dense layers	[128,128,128, d_{action}]

Table 6: Hyperparameters of critic networks Q_k^l and Q^g

Parameter	Value
Batch-norm.	yes
Activation	LeakyRelu
Dense layers	[128,128,128, 1]

B.2 Training hyperparameters

Table 7: Hyperparameters during training (model learning)

Parameter	Value
batch size	64
learning rate (all models)	0.0003
c_0	0.2
c_1 (*SpiralMaze*)	0.001
c_1 (*ObstacleMaze*)	1.0
c_1 (metaworld tasks)	0.01
c_2	0.001
c_3	0.001
c_3 (expert)	0.5
γ (discount factor)	0.96
$d_{\mathcal{Z}}$	32
T (temperature parameter)	1.0
n_{ens}	3

B.3 Planner and controller hyperparameters

Table 8: Hyperparameters during inference (planning)

Name	Description	Value
n_{iter}	Number of planner iterations	250 (500 in *ButtonWall*)
n_{sim}	Number of simulation steps during tree expansion	5 (10 in *SpiralMaze*, *ButtonWall* and *DrawerButton*)
$\tau_{\text{discard}}^{\text{high}}$	Q-value threshold for discarding node if too close to existing nodes in the tree	γ^2
$\tau_{\text{discard}}^{\text{low}}$	Q-value threshold for discarding node if too far from expansion node	$\gamma^{n_{\text{sim}}}$
$\tau_{\text{discard}}^{\text{std}}$	Q-value threshold for discarding node if standard deviation of ensemble prediction is too high	$1.0 - \gamma$
τ_{neigh}	Q-value threshold to determine neighboring nodes	γ^3
τ_{goal}	Q-value threshold to determine goal nodes	γ^1
d_{neigh}	Euclidean distance threshold to determine candidate neighbors	3 x upper 5-percentile of Eucl. distances between encoding of subsequent states

Table 9: Hyperparameters during inference (controller)

Parameter	Description	Value
n_{replan}	Planning frequency	15 (25 in *SpiralMaze*, *ButtonWall*)
ϵ_{goal}	Q-threshold to determine vicinity to the goal	γ^5
ϵ_{wp}	Q-threshold for switching to the next waypoint	γ^3

B.4 Training of policy and value functions

We use TD3-BC [48] as the base offline RL algorithm to train our local and goal policies π^l and π^g, respectively state-action value functions a Q_k^l and Q^g. Within our planning framework Q_k^l takes an important role as it provides us with a distance proxy. To further improve the accuracy of Q_k^l, we use k Q networks (instead of 2 usually used in TD3). During the training update of the Q-network, we then determine the Q-target by taking the minimum value among the predictions given by the ensemble of Q-networks (similar to [49]). The ensemble further allows us to filter out unlikely or out-of-distribution transitions generated during the tree expansion by thresholding the resulting Q-values based on the minimum predicted ensemble value and the standard deviation among the predicted values.

Our models π^l and Q_k^l describe goal-reaching navigation policy and state-action value functions which require a set of goal-conditioned reaching experiences for training. Since our original dataset $\mathcal{D}$ might not provide such data, we augment it using data augmentation via hindsight relabeling. In particular, we create a new dataset $\mathcal{D}'$ creating transitions $(z_t, a_t, r_t, z_{t+1}, z^g, \gamma) \in \mathcal{D}'$ based on the existing transitions in $\mathcal{D}$ by relabeling the values of r_t, γ (γ also indicates terminal condition, i.e. $\gamma = 0$) and adding a new goal state z^g. In this regards, we apply a combination of three different relabeling strategies (a) set goal z^g to be next state of the original transitions and set $\gamma = 0$ and $r_t = 1$ (b) sample z^g from the set of future states within the same trajectory and set $\gamma = 0$; $r_t = 0$ (c) sample z^g from another trajectory in the data and $\gamma = 0$ and $r_t = 0$.

B.5 Additional details about planning method

Neighbor computation To determine if a newly sampled node z_{new} is novel, we check its similarity to existing nodes in the tree by evaluating the state-action value function. Yet, evaluating the values network for all nodes in the tree results in an enormous computations overhead. Yet, we can significantly reduce this computation by first determining a set of candidate neighbors around z_{new} using the Euclidean metric and a distance threshold d_{neigh}. In practice, we found it useful to define d_{neigh} based on the statistics of Euclidean distances between subsequent states in the dataset (see App. B.3).

Batch processing The method in Alg. 1 describes an iterative schema for which at every expansion step one new node is generated and evaluated. Yet, some steps can be compuated in parallel on a GPU in order to speed up the planning time. For a practical implementation, we therefore suggest to parallelize the tree expansion by sampling multiple expansion nodes at once and generating new nodes by passing batches through the neural network dynamics model. Similarly, we can compute state-action values in batches instead of assessing one new nodes at a time. For discussions about highly-parallelized implementations of classical RRT-like planners, we refer to [58, 59].

B.6 Additional details about MPC controller

The below Alg. 2 presents the pseudocode for our MPC controller. The fucntion `update_waypoint`(z_{curr}, g^*) determines the next waypoint z_{wp} which we seeks to achieve using our local policy. In particular, we estimate the value between the current state and waypoint and switch to the next element in g^* if the predicted value surpasses a threshold ϵ_{wp}, i.e $Q_{\text{min}}^{\text{curr,wp}} > \epsilon_{\text{wp}}$. Once the z_{curr} gets close to the goal, we disable planning and determine actions based on π^g. To determine vicinity to the goal, we check if the prediction of the global value function Q^g exceeds a threshold ϵ_{goal}.

Algorithm 2 MPC controller

Given: s_{init}, n_{replan} $n_{\text{max steps}}$, ϕ, π^l
$z_{\text{curr}} \leftarrow \phi(s_{\text{init}})$ ▷ Map state to latent encoding
$i \leftarrow 0$
while goal not achieved and $i < n_{\text{max steps}}$ **do**
  if not $i \mod n_{\text{replan}}$ **then** ▷ Replan every n_{replan} steps
    Build tree $\mathcal{T}$ rooted at z_{curr} for n_{iter} steps (Alg. 1).
    Determine $g^* = \{z_{curr}, z_1, .., z_n\}$ given $\mathcal{T}$ (Eq. 8)
  end if
  $z_{\text{wp}} \leftarrow$ `update_waypoint`(z_{curr}, g^*) ▷ Update waypoint if close enough
  $a \leftarrow \pi^l(z_{\text{curr}}, z_{\text{wp}})$ ▷ Compute action given policy
  Execute a and update state
  $z_{\text{curr}} \leftarrow \phi(s_{\text{curr}})$
end while

C Evaluation Environments

C.1 Description of block environments

Similar to the evaluation environments in [14], we implement two long-horizon navigation tasks whose underlying state space is relatively low-dimensional in order to facilitate illustration and visual inspect of learned embeddings using dimensionality reduction techniques such as Isomap [60]. For both environments, the a block robot is controlled using velocity commands while it's position in constrained to a planar surface.

C.1.1 SpiralMaze

To solve this task, the block agent must navigate form the outer end of the spiral-shaped corridor to the inner region (colored in red; see Fig. 2). The maximum allowed number of episode steps is limited to 300. To generate training data, the agent is placed randomly at a collision free position in the workspace and random actions sequences are applied by subsequently adding Gaussian noise an initially sampled random uniform action. For testing, the agent's position is sampled uniformly within a small region close to the outer end of the spiral-shaped corridor.

C.1.2 ObstacleMaze

In this environment, the agent must navigate towards the upper wall of the workspace (color in red; see Fig. 2). To achieve the goal the agent must take actions around two obstacles which are randomly placed around the center of the workspace at the beginning of each new episode. The maximum allows number of environment steps is set to 100. For testing, the agent is initialized to random configuration close to wall which is opposite to the goal. We used the same random data collection policy as for the *SpiralMaze* task.

C.2 Description of manipulation environments

We adapted and implemented several robot manipulation environments based on the Metaworld [17] robot benchmark tasks. The underlying physics simulator in this regard is Mujoco [61]. To enable visual manipulation, similar to the problems studied in [6], we enable background rendering of RGB images from a static viewpoint. The robot is controlled by commanding desired endeffector and gripper opening displacements resulting in a 4-dimensional action space. While *WindowClose* and *FaucetClose* were with small modifications adapted from the [17], we evaluate two new scenarios *ButtonWall* and *DrawerButton* which were particularly desired to study our method under extreme sparse reward conditions over a long temporal horizon which requires trajectory "stitching" to find a solution policy.

For data collection, we use a suboptimal policies which takes random actions (additive Gaussian noise) most of the time and with a low probability takes an action generated by a scripted expert

policy. Table. 10 provides insight about the number of samples and trajectories in the training data and as well presents the portion of successful actions (reward=1). For all manipulation tasks, we set the maximum permitted environmental steps at 150, with the exception of the "ButtonWall" scenario, where we allow up to 250 steps during the evaluation phase.

C.2.1 WindowClose

In order to accomplish this task, the robotic arm must successfully open a window by shifting a specific handle sideways. We implement environmental variability by randomly determining the x-y location of the window object in each episode. During the data collection stage, we randomly set the positioning of the end-effector above the surface of the table. However, we restrict the sampling of expert actions to areas close to the objective (window handle). This approach is intended to guarantee that the strategy employed necessitates to "stitch" different trajectories together to reach the objective and complete the task when starting from states that are farther away. To ensure challenging planning situations during testing, we initiate the robot at a significant distance away from the target.

C.2.2 FaucetClose

This task is similar to the *WindowClose* task, but it requires the agent to use its end-effector to close a faucet instead. In addition, we employ analogous strategies for data gathering and scenario creation as those used in the *WindowClose* environment.

C.2.3 ButtonWall

In this scenario, the robot's end-effector is required to navigate around a wall structure before pressing a button. The wall's location is randomly set at the beginning of each episode. Furthermore, a height limitation is imposed on the end-effector to ensure that the agent takes a more extended path around the wall, as opposed to simply elevating the end-effector. The dataset was produced by either placing the agent in front of the wall, near the button, or far behind the wall. However, expert samples in the dataset only exist for scenarios when starting closer to the goal. For testing purposes, the end-effector is sampled within a restricted region behind the wall to increase the planning task's complexity.

C.2.4 DrawerButton

In this scenario, the agent is tasked to first close a drawer using its end-effector and subsequently press a button. To train the agent, we develop a dataset by separately collecting trajectories for each subtask. This approach necessitates the use of a method capable of combining different trajectories in the data to devise a solution that achieves the overall task goal.

C.3 Composition of training dataset

The table below presents the composition of our training datasets. Each context in this regards, refers to a new environment initialization (excl. agent) such as the position of obstacles.

Table 10: Composition of training datasets for each environment

Environment	Num. contexts	Traj. per context	Max. traj. length	Successful transitions
SpiralMaze	1	1000	20	0.12 %
ObstacleMaze	250	20	20	0.11 %
WindowClose	200	10	50	0.48 %
ButtonWall	200	10	50	0.16 %
FaucetClose	200	10	50	0.31 %
DrawerButton	150	20	50	0.16 %

D Baseline methods

To enable a fair comparison between different methods, we use the same underlying representation/encoder ϕ and dynamics model h in the evaluation of all baselines. For assessing the quality and impact of our representation learner, please review the experimental ablation study in App. E.2.

D.1 BC and BC ($\mathcal{D}^*$)

Simple behavioral cloning baselines for which we use the same network architecture as our policy networks (see Table 5) and train using a mean-squared error objective. For $\mathcal{D}^*$ we train only on the subset of successful episodes in the dataset. The train both methods for $3 \cdot 10^5$ iterations using a learning rate of $3 \cdot 10^{-4}$ and batches of size 128.

D.2 TD3-BC [48]

This baselines resembles the underlying global policy π^g used in `VELAP`. It provides us with an intuition how well pure offline RL performs without adding any planning methods on top.

D.3 IQL [54]

This method presents a state of the art model-free offline RL baseline which utilizes expectile regression to estimate state-conditional expectiles of the target values in order to avoid querying values of out-of-sample actions during training. The train IQL for $3 \cdot 10^5$ iterations, a learning rate of $3 \cdot 10^{-4}$, batches of size 256 and $\tau = 0.7$ and $\beta = 3.0$.

D.4 MPPI

We implement a trajectory optimization baseline similar to the model-based planning algorithm introduced in [7]. The method in [7] presents an adaption of MPPI specifically for the online reinforcement learning setting which seeks to optimize the expected return of sampled trajectories. To estimate the return, a learned model is used to predict the reward for each trajectory node while a learned Q-function predicts the future return beyond the specified planning horizon. Since rewards in our evaluation environments are sparse, a learned model of the environment reward carries guidance for the trajectories optimization as most states have 0 reward. Therefore, we adapt the objective in [7] and instead use the accumulated sum of state-action values as the optimization criterion. This type of scoring function in model-based RL has recently been discussed in [62]. To implement this baseline, we utilize the Q-function of TD3-BC. For all environments, we use 1000 samples per iteration, a planning horizon of 50, elite size 64 and 5 iterations. Replanning is done every 5 environment steps.

D.5 MBOP [35]

MBOP presents an adaptation of MPPI which was particularly designed for the offline RL setting. It generates new candidate trajectories by addding small amount of Gaussian noise to the actions predicted by a behavioral-cloned policy. To evaluate the quality of the rollouts it uses a truncated value function trained on the offline data. Due to the sparse nature of rewards in our experiments, we found that both the behavioral-cloned policies and the truncated value function were insufficient to generated farsighted behaviors that solve our tasks. To accommodate for the long planning horizons, we instead sample action for a TD3-BC policy and use the corresponding Q-functions to score candidate trajectory during the optimization update. For all environments, we use 1000 samples per iteration, a planning horizon of 50, elite size 64, 5 iterations and a β parameter of 0.7. Replanning is done every 5 environment steps.

D.6 IRIS [36]

IRIS presents an offline RL methods that was desired particularly for sparse reward environments. It uses a hierarchical decomposition of the planning agent into a low-level and a high-level policy. A conditional VAE model is trained on offline trajectory data and predicts a set of suitable subgoals states that are n-step ahead of the current states. The high-level policy is essentially represented by a Q-function which is chooses the highest values subgoal states among the set of generated candidate states. The low-level policy is then used to navigate towards the predicted subgoal. In all experiments we implement IRIS by training an conditional VAE to predict subgoal states at a horizon of 5 and sample a candidate set of size 256. To implement both the low-level and high-level policy, we use TD3-BC as the base RL algorithm.

D.7 IRIS (multi-step)

We evaluate an extension of IRIS in which we use the state prediction model to generate multi-step rollouts of suitable subgoals. This strategy increases the exploration horizon and allows to choose the best subgoal from a larger and potentially more diverse set of states. It exploration strategy can also be seen as random shooting of coarse subgoal sequences. In all experiments, we generate 256 different trajectories using rollouts of length 5 and a conditional generative model to predict states for a horizon of 5. In our evaluation, we found this method to sometimes perform worse than IRIS. We attribute this to the fact that the global policy doesn't align with the capabilities of the local one, which occasionally results in the selection of subgoal states that might not be attainable.

E Supplementary Experiments and Analysis

E.1 Physical hardware experiments

For the real-world validation of our method, we collected 200 episodes of data for the sponge ($\sim$ 15000 samples) and 150 episodes of data for the rope manipulation ($\sim$ 15000 samples) tasks. Training data was generated by operating the robot through a gamepad and took less than 1 hour per task. The collected dataset consist largely of suboptimal trajectories. Successful transition (positive reward + episode termination) were labeled during data. We form states by stacking three subsequent images taken by a static camera. The results of a comparison against BC, BC ($\mathcal{D}^*$) and IRIS are presented in Table 11.

Table 11: Results of phyiscal robot experiments (successful episodes)

Environment	BC	BC $\mathcal{D}^*$	IRIS	VELAP
Sponge	5/20	6/20	6/20	14/20
Rope	0/20	0/20	2/20	8/20

E.2 Ablating the impact of the learned representation

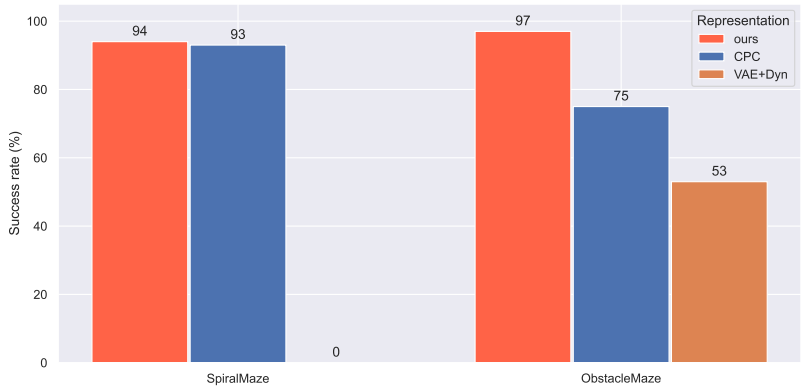

Figure 6: Impact of type of representation on the performance of our planner.

E.3 Influence of the dynamics loss

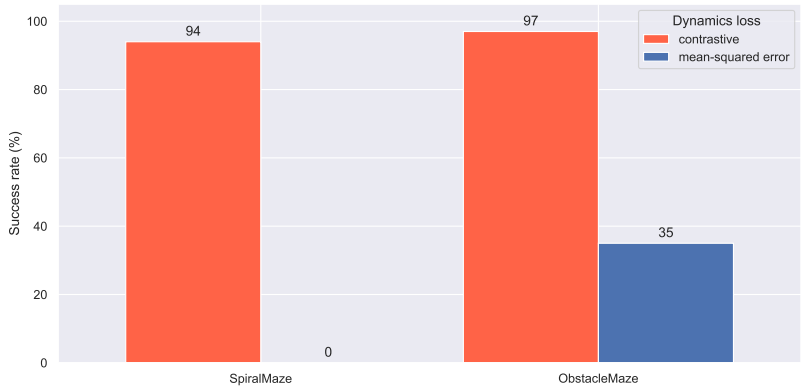

Figure 7: Impact of type of representation on the performance of our planner.