CONTINUAL ROBOT POLICY LEARNING VIA VARIATIONAL NEURAL DYNAMICS

Anonymous authors
Paper under double-blind review

ABSTRACT

Robots deployed in the real world rarely operate under a single fixed dynamics model: wind changes, payloads vary, batteries drain, and hardware wears. Yet most learning-based controllers are trained once and deployed as if learning were complete. This prevents the robot from using deployment experience to recursively improve task performance. In this work, we propose a continual learning framework that uses real-world experience to improve robot policies under hidden and time-varying dynamics. Our method learns a condition-aware dynamics model from real state-action data by combining an analytical physics prior with a neural residual for unmodeled effects. Crucially, our continual learning occurs during real-world deployment: the robot repeatedly collects new interaction data, updates its learned dynamics model, and improves the policy, in a continual loop of about $10\,\mathrm{s}$. During execution, a recurrent encoder identifies the current hidden condition from recent interactions, allowing the policy to retain and reuse previously learned adaptations when conditions recur rather than adapting from scratch. Unlike meta-learning, our method can learn dynamics variations directly during deployment rather than from predefined, frozen training conditions. Through simulation studies and real-world experiments, we show that the framework improves policy performance across different platforms under a range of unobserved disturbances. In simulation under large disturbances, our method reduces the hover and tracking errors by 65.7% and 53.3% over state-of-the-art online adaptation approaches. On real quadrotor tracking task under changing disturbances, the policy recovers from recurring disturbances in roughly $1\,\mathrm{s}$, about $5\times$ faster than online residual re-fitting.

1 INTRODUCTION

Learning-based controllers have achieved remarkable performance across robotics domains, spanning agile quadrotor flight (Kaufmann et al., 2023; Geles et al., 2024; Xing et al., 2024), legged locomotion (Miki et al., 2022; Lee et al., 2020), and dexterous manipulation (Chi et al., 2025; Qi et al., 2023). These control policies are typically trained in simulation, transferred to hardware, and deployed with fixed parameters. This *train-once, deploy-forever* paradigm has enabled impressive demonstrations. However, the policy learning stops once the robot is deployed. Yet real-world dynamics are only partially known before deployment and may continue to change during operation (Aljalbout et al., 2025). The robot may therefore encounter conditions not fully captured in training, but the data it collects are never used to improve the policy.

In aerial robotics, wind, payload changes, battery state, and hardware degradation all affect the closed-loop system. These environment variations are often not directly observable, and some operating conditions may even recur over time. A continual learning system should therefore infer the current operating condition from recent interactions. When it encounters a new condition, it should learn from the resulting experience, and expand the range of dynamics it can handle. When a familiar condition returns, the robot should recognize it and reuse what it has already learned, rather than adapt from scratch, as existing one-shot adaptation approaches do.

Existing approaches address only parts of this problem. Domain randomization (Tobin et al., 2017) trains a policy across diverse simulated dynamics, while meta-learning and latent-adaptation methods (Rakelly et al., 2019; Kumar et al., 2021; Huang et al., 2023) additionally learn to infer the

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

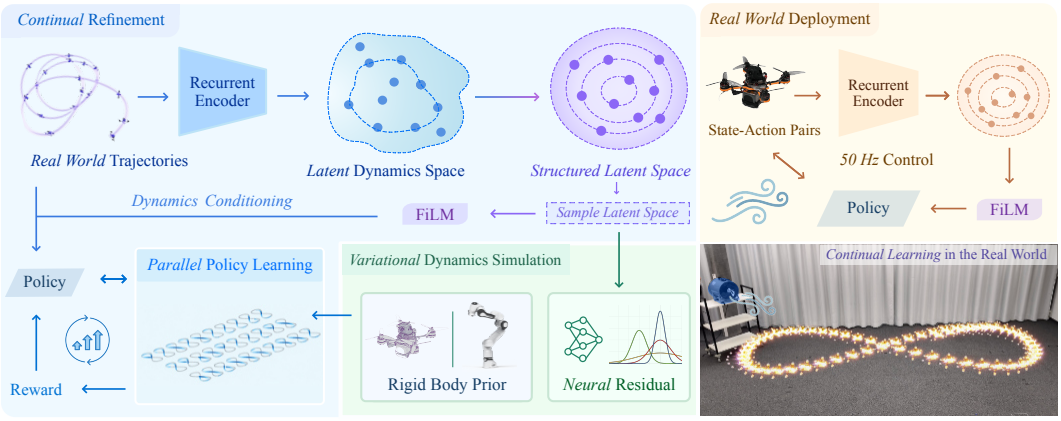


Figure 1: **Method overview.** Our framework learns latent-conditioned residual dynamics from real trajectories, augmenting a rigid-body prior with a FiLM-modulated neural residual inferred from recent state-action history. Sampled latents drive parallel differentiable policy training, while online inferred latents enable 50 Hz condition-aware deployment without privileged disturbance labels.

physical condition from interaction history. Both approaches help the policy handle a broader range of dynamics, but this range is defined in simulation before deployment, often using privileged parameters. Once deployed, neither approach uses new data to update the underlying model or policy. Real2Sim2Real methods (Kaufmann et al., 2023; Hwangbo et al., 2019) instead learn from real interactions, and recent methods can even fit residual dynamics from real flight data rapidly online (Pan et al., 2025). However, they typically fit one residual for the condition currently observed. When the dynamics change, the residual must be fitted again from scratch, even if the robot returns to a condition it has encountered before (Wang et al., 2025). Until refitting is complete, the controller operates with a model that does not match the current dynamics. For agile systems such as quadrotors, even a short mismatch can cause large tracking errors. If conditions change frequently, the robot need to adapt again before it has recovered from the previous change.

In this work, we propose a continual policy learning framework that recursively improves both its dynamics model and control policy during deployment. In each learning iteration, the robot executes its current policy and collects state-action trajectories. Here we introduce Variational Neural Dynamics (VND), which then uses these trajectories to learn a latent-conditioned correction to the robot dynamics model. VND models a distribution of residual dynamics across the conditions the robot has encountered, so it can learn from new conditions while retaining what it learned from earlier ones. A recurrent encoder maps recent state-action history to a latent embedding of the current condition. Both the residual model and the policy use embeddings from the same latent space.

During model training, we continually learn and expand this space from real trajectories and align its embeddings with a sampable distribution. During policy learning, we sample embeddings from this distribution and use the learned model to generate rollouts that reflect the dynamics encountered during deployment. We update the policy by backpropagating through these rollouts, then deploy it for the next iteration. When a familiar condition returns, the encoder infers its embedding from recent history, allowing the policy to reuse its earlier response without fitting a new residual. The entire continual learning process consists of repeated iterations of VND training and policy updates, with each iteration taking around 10 s during real-world deployment.

We study both our continual learning framework in controlled simulations of agile quadrotor flight and robotic manipulation, and in real-world experiments with a quadrotor under changing wind disturbances. The policy improves over successive deployment iterations as new data are incorporated. When a previous disturbance returns on hardware, the policy recovers in roughly 1 s, about $5\times$ faster than online residual re-fitting. In simulation under large disturbances, it reduces hover and tracking errors by 65.7% and 53.3%, respectively, compared with the state-of-the-art online residual adaptation baseline. Together, we show that a robot can continue improving after deployment and reuse what it learned under previous conditions, both of which we believe are essential for continual self-improvement in the real world.

2 Related Work

Online Policy Adaptation in the Real World. Adaptive controllers such as $\mathcal{L}_1$-MPC (Hanover et al., 2021) reject disturbances quickly but rely on a predefined model structure. Model-free RL (Schulman et al., 2017; 2015) can in principle keep learning during deployment, but it needs more samples than a physical robot can provide (Rudin et al., 2022). Recent work closes this gap with differentiable simulation: a residual dynamics model is fit from real flight data online and used to refine the policy (Pan et al., 2025; Ren et al., 2026). These methods, however, fit a single residual to the currently active condition and must refit it whenever the condition changes, even to one seen before. Our method instead remembers conditions it has already learned, so a recurring condition is recognized rather than refit.

Latent Conditioning on Unobserved Environmental Variables. Inferring hidden dynamics from recent interaction is a common way to adapt policies, from online system identification (Yu et al., 2017) and implicit adaptation under randomized dynamics (Peng et al., 2018) to meta-RL and meta-learned adaptation (Rakelly et al., 2019; O'Connell et al., 2022). In robotics, RMA (Kumar et al., 2021; 2022; Qi et al., 2023) condition policies on latents inferred from history for legged locomotion, in-hand manipulation, and quadrotor tracking. Common to these methods is that the distribution of conditions is fixed before deployment: the designer specifies it in simulation, and the encoder is trained there, often with privileged parameters (Tobin et al., 2017; Finn et al., 2017). Once deployed, the policy can adapt only within the fixed distribution it was trained on and cannot expand this distribution as new conditions arise. Our method instead learns the condition distribution directly from real-world data and continually refines the policy as new scenarios are encountered.

3 Methodology

Problem Formulation. We model the task execution as a hidden-parameter Markov decision process (Doshi-Velez & Konidaris, 2016) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{W}, \mathcal{T}, r, \gamma)$. Here $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{W}$ are the state, action, and hidden-condition spaces, $\mathcal{T}$ is the transition kernel, r is the reward, and γ is the discount factor. At time t, the robot observes its state $\mathbf{s}_t \in \mathcal{S}$ and applies an action $\mathbf{a}_t \in \mathcal{A}$, but the active condition $\mathbf{w} \in \mathcal{W}$, for example the present wind, is hidden. The transition depends on this hidden condition, $\mathbf{s}_{t+1} \sim \mathcal{T}(\cdot \mid \mathbf{s}_t, \mathbf{a}_t, \mathbf{w})$. We assume $\mathbf{w}$ is piecewise constant: it stays fixed for many control steps but may switch during deployment, and previously encountered conditions may recur.

The policy acts on a task observation $\mathbf{o}_t = O_{\text{task}}(\mathbf{s}_t, \mathbf{x}_t^{\text{ref}}, \mathbf{a}_{t-K:t-1})$ containing the state, task-specific reference features, and a short action history. Under a fixed condition, the return of policy π_θ over horizon H is $J(\theta; \mathbf{w}) = \mathbb{E}\big[\sum_{t=0}^{H-1} \gamma^t r(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})\big]$, with $\mathbf{s}_0 \sim \rho_0$ and transitions drawn from $\mathcal{T}(\cdot \mid \cdot, \cdot, \mathbf{w})$. Ideally, we would maximize the return under the conditions the robot actually encounters:

$$\max_\theta \ \mathbb{E}_{\mathbf{w} \sim p_{\text{deploy}}}\big[J(\theta; \mathbf{w})\big] \quad \longrightarrow \quad \max_\theta \ \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}\big[\hat{J}(\theta; \mathbf{z})\big]. \tag{1}$$

Here, the deployment distribution p_{deploy} is unknown and $\mathbf{w}$ is not observed. Domain randomization and meta-learning approximate p_{deploy} with a distribution over simulator parameters chosen by the designer. However, if this approximate distribution is too narrow, the policy can fail under conditions not encountered during training; if it is too wide, the policy can become overly conservative and lose performance on the conditions that actually occur.

Instead of specifying this distribution in advance, we learn it from the robot's deployment experience. We assume access only to a nominal model $f_{\text{prior}}(\mathbf{s}_t, \mathbf{a}_t)$ and deployment trajectories stored in a growing buffer $\mathcal{B}$. From $\mathcal{B}$, we learn an encoder that infers a latent representation $\mathbf{z} \in \mathbb{R}^d$ of the hidden condition $\mathbf{w}$ from recent state-action history, together with a dynamics model conditioned on $\mathbf{z}$. We align the encoded latents with a fixed sampling prior $p(\mathbf{z})$, so that different dynamics conditions can be sampled directly in the latent space during policy learning. For a sampled latent $\mathbf{z}$, the learned dynamics model defines the corresponding surrogate environment, and $\hat{J}(\theta; \mathbf{z})$ in Eq. (1) denotes the policy return in this environment. For the surrogate to be useful, the latent must satisfy two requirements. *(i) Inferable:* because $\mathbf{w}$ is hidden at deployment, $\mathbf{z}$ must be recoverable from recent interaction. This is possible because the recent trajectory carries information about the active dynamics. *(ii) Sampleable:* during policy training, latents drawn from $p(\mathbf{z})$ must produce realistic dynamics, so that the surrogate reflects conditions supported by the collected data.

In the remainder of this section, we will describe how we implement this formulation in practice, from the latent dynamics model to the continual policy learning loop.

System Overview. Our framework is a continual *loop* that runs throughout the robot's deployment (Algorithm 1, Figure 1). Each round of the loop has three phases. *(1) Collect and infer.* The robot executes its current policy at the control rate. At each step, a recurrent encoder E_ϕ reads the last C state-action pairs $\mathcal{H}_{t-C:t}$ and outputs a latent embedding $\mathbf{z}_t$ that stands in for the hidden condition $\mathbf{w}$, for example the present wind. The policy acts on both the task observation and this embedding, $\mathbf{a}_t = \pi_\theta(\mathbf{o}_t, \mathbf{z}_t)$, and every transition $(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})$ is appended to the buffer $\mathcal{B}$. *(2) Learn latent dynamics.* In parallel with data collection, we jointly update the encoder and a latent-conditioned residual model D_ψ on $\mathcal{B}$. By predicting the condition-dependent correction to f_{prior}, the model provides a learning signal that makes $\mathbf{z}$ capture the active dynamics and thus is inferable from recent history. The encoded latents are also aligned with the sampling prior $p(\mathbf{z})$, which makes the latent space sampleable to "simulate" real conditions. *(3) Improve policy.* With the encoder and dynamics model frozen, latents are sampled from $p(\mathbf{z})$, and the policy is trained by backpropagating the task reward through rollouts of the learned model, maximizing the surrogate objective in Eq. (1). The improved policy and the updated encoder are then deployed, and the next round begins. The loop starts from a base policy trained on f_{prior} alone. Throughout, the same latent space plays two roles: it is *inferred* from real history in phase (1) and *sampled* to generate training conditions in phase (3). The latent distribution is learned and grown during deployment data rather than frozen before deployment.

Algorithm 1: Continual Policy Learning via VND

Require: f_{prior}, buffer $\mathcal{B}$, prior $p(\mathbf{z})$
Init. $E_\phi, D_\psi, G_\eta, \pi_\theta$
REPEAT DURING DEPLOYMENT
1. Collect and infer
$\mathbf{z}_t \leftarrow E_\phi(\mathcal{H}_{t-C:t})$
$\mathbf{a}_t \leftarrow \pi_\theta(\mathbf{o}_t, \mathbf{z}_t)$
$\mathcal{B} \leftarrow \mathcal{B} \cup (\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})$
2. Learn latent dynamics
Update (ϕ, ψ, η) with
$\mathcal{L}_{\text{vnd}} = \mathcal{L}_{\text{dyn}} + \lambda_{\text{rec}}\mathcal{L}_{\text{rec}} + \lambda_{\text{mmd}}\,\text{MMD}(q_\phi, p)$
3. Improve policy
Freeze E_ϕ, D_ψ; sample $\mathbf{z} \sim p(\mathbf{z})$
Roll out with
$\mathbf{s}_{t+1} = f_{\text{prior}}(\mathbf{s}_t, \mathbf{a}_t) \oplus D_\psi(\mathbf{s}_t, \mathbf{a}_t, \mathbf{z})$
Update θ by BPTT on
$J(\theta) = \mathbb{E}_{\mathbf{z},\mathbf{s}_0} \sum_{h=0}^{H-1} \gamma^h r(\mathbf{s}_h, \pi_\theta(\mathbf{o}_h, \mathbf{z}), \mathbf{s}_{h+1})$

Latent-Conditioned Residual Dynamics. Our dynamics model has two components: a history encoder that infers the latent code, and a residual model that uses this code to predict the next state.

History encoder. Let $\mathcal{H}_{t-C:t} = \{(\mathbf{s}_i, \mathbf{a}_i)\}_{i=t-C}^{t-1}$ be a window of the last C state-action pairs. A GRU encoder maps this window to a low-dimensional latent, $\mathbf{z}_t = E_\phi(\mathcal{H}_{t-C:t}) \in \mathbb{R}^d$. We use a recurrent encoder because a single transition rarely contains enough evidence to identify the condition; wind, for example, appears as accumulated changes in acceleration and attitude over several control steps.

Residual dynamics. Rather than learning the full dynamics model from scratch, we keep the known physics in the nominal model f_{prior} and learn only the part it misses: $\hat{\mathbf{s}}_{t+1} = f_{\text{prior}}(\mathbf{s}_t, \mathbf{a}_t) \oplus D_\psi(\mathbf{s}_t, \mathbf{a}_t, \mathbf{z}_t)$, where $\oplus$ adds the Euclidean state components and the orientation correction through the exponential map. Learning only this discrepancy is more sample-efficient (Pan et al., 2025) and easier than learning the full transition, as our ablations confirm. The residual D_ψ captures condition-dependent effects such as wind, drag, payload changes, friction, or contact-induced deviations. Because the residual also takes $\mathbf{z}_t$ as input, the correction it applies to the nominal dynamics can change with the condition, hence here a single model is enough to handle many conditions.

Latent conditioning. D_ψ is an MLP whose hidden features $\mathbf{h}$ are modulated by the latent through FiLM (Perez et al., 2018), $\mathbf{h} \leftarrow \boldsymbol{\alpha}(\mathbf{z}) \odot \mathbf{h} + \boldsymbol{\beta}(\mathbf{z})$, where $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are learned functions of the latent. Compared with appending $\mathbf{z}$ to the input, FiLM lets the latent change how the network responds to the state and action, which improves prediction accuracy in our ablations..

Learning an Inferable Latent Space. To satisfy requirement (i), we train E_ϕ and D_ψ jointly to predict dynamics, without privileged labels such as wind direction, payload mass, friction coefficient, or hardware state. The key idea is that a latent inferred from one context window must explain several transitions from the same trajectory segment. Under the piecewise-constant assumption, these transitions share the same hidden condition, so a single latent can explain all of them only by capturing what they have in common: the condition itself.

Formally, let $\mathbf{y}_t = \mathbf{s}_{t+1} \ominus f_{\text{prior}}(\mathbf{s}_t, \mathbf{a}_t)$ be the residual target, that is, the part of the observed transition that the prior fails to explain. We sample context windows $\mathcal{H}_n$ from $\mathcal{B}$,

pair each with transitions from the same trajectory segment, and minimize $\mathcal{L}_{\text{dyn}}(\phi, \psi) = \frac{1}{M}\sum_{i=1}^{M} \ell_\delta\Big(D_\psi\big(\mathbf{s}_i, \mathbf{a}_i, E_\phi(\mathcal{H}_{n(i)})\big) - \mathbf{y}_i\Big)$, where ℓ_δ is the Huber loss, M is the batch size, and $n(i)$ is the index of the context window paired with transition i. Because different conditions require different residual corrections, they must be assigned different latents, so residual prediction acts as a self-supervised signal for the hidden condition. To help the latent retain more information, we add an auxiliary trajectory reconstruction loss. A decoder G_η takes the latent $E_\phi(\mathcal{H}_n)$ and reconstructs the encoder's own input, the sequence of C state-action pairs in the context window $\mathcal{H}_n$: $\mathcal{L}_{\text{rec}} = \frac{1}{N}\sum_{n=1}^{N} \|G_\eta(E_\phi(\mathcal{H}_n)) - \text{norm}(\mathcal{H}_n)\|_2^2$, where $\text{norm}(\cdot)$ normalizes each state and action channel with statistics computed on $\mathcal{B}$. We found that adding this term improves the accuracy of the learned dynamics (Figure 5).

Sampleable Latent Space. The dynamics loss makes latents inferable, but it does not control where they lie in $\mathbb{R}^d$. It only requires that different conditions receive different latents, so the encoded latents may end up in an arbitrary region. This is a problem for policy learning, which has no trajectory to encode and therefore samples latents from a prior $p(\mathbf{z})$ instead. If the prior and the encoded latents do not overlap, prior samples query the residual model in regions it was never trained on and produce unrealistic dynamics simulation.

We therefore align the encoded latents with a Gaussian prior $p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$. Let q_ϕ denote the empirical distribution of encoded latents $\{E_\phi(\mathcal{H}_n)\}_{n=1}^{N}$. We measure how far it is from the Gaussian p with the maximum mean discrepancy (MMD) (Gretton et al., 2012), which compares two distributions through their samples: $\mathcal{L}_{\text{mmd}}(q_\phi, p) = \mathbb{E}_{\mathbf{z},\mathbf{z}'\sim q_\phi}[k(\mathbf{z}, \mathbf{z}')] + \mathbb{E}_{\tilde{\mathbf{z}},\tilde{\mathbf{z}}'\sim p}[k(\tilde{\mathbf{z}}, \tilde{\mathbf{z}}')] - 2\,\mathbb{E}_{\mathbf{z}\sim q_\phi,\tilde{\mathbf{z}}\sim p}[k(\mathbf{z}, \tilde{\mathbf{z}})]$, where k is an RBF kernel that measures the similarity between two latents. The first two terms measure how similar samples are within each distribution, and the last term measures how similar they are across the two. Formally, $\mathcal{L}_{\text{mmd}}$ is the squared distance between the kernel mean embeddings of q_ϕ and p, and it vanishes only when the two distributions coincide. Unlike a KL divergence, which requires a density that our deterministic encoder does not provide, MMD aligns the latents as a group using only samples (estimator in Appendix A.9), so each condition keeps its own region of the latent space while together the latents match the prior. As a result, a sample from the prior falls near a condition the robot has already seen, or between such conditions. Note that $p(\mathbf{z})$ itself is fixed; what is learned is the residual model that turns each prior sample into a condition, so the distribution over dynamics obtained by sampling $p(\mathbf{z})$ and decoding through D_ψ is determined by the data. MMD matches the encoded latents to the prior in aggregate: it guarantees that prior samples land in regions the encoder populates, not that every such region decodes to a realistic condition, which we verify empirically in Section 4.

Joint Training Objective. The encoder, residual model, and decoder are trained jointly with $\mathcal{L}_{\text{vnd}} = \mathcal{L}_{\text{dyn}} + \lambda_{\text{rec}}\mathcal{L}_{\text{rec}} + \lambda_{\text{mmd}}\mathcal{L}_{\text{mmd}}$. We start with the dynamics loss alone and then gradually increase λ_{mmd} (Appendix A.9), so that the dynamics loss separates the conditions before the latents are aligned with the prior.

Policy Learning with Sampled Latent Dynamics. Once the dynamics model has been learned, we use it to improve the current policy, with hidden conditions sampled from the prior $p(\mathbf{z})$ and decoded by the learned residual model, so that the resulting distribution over dynamics reflects the conditions encountered in the real world. During policy learning, the encoder and residual model are kept frozen. We run N_{env} parallel rollouts. At the start of rollout j, we sample a latent $\mathbf{z}^{(j)} \sim p(\mathbf{z})$ and hold it fixed for the whole rollout. The same latent conditions both the policy and the learned dynamics: $\mathbf{a}_t = \pi_\theta(\mathbf{o}_t, \mathbf{z}^{(j)}), \hat{\mathbf{s}}_{t+1} = f_{\text{prior}}(\hat{\mathbf{s}}_t, \mathbf{a}_t) \oplus D_\psi(\hat{\mathbf{s}}_t, \mathbf{a}_t, \mathbf{z}^{(j)})$, where $\hat{\mathbf{s}}_0 \sim \rho_0$, and $\mathbf{o}_t$ is computed from $\hat{\mathbf{s}}_t$ with the same observation map O_{task} used at deployment. Averaging the discounted return $\sum_{t=0}^{H-1} \gamma^t r(\hat{\mathbf{s}}_t, \mathbf{a}_t, \hat{\mathbf{s}}_{t+1})$ over the rollouts gives an estimate of the surrogate objective in Eq. (1). Because f_{prior}, D_ψ, and π_θ are differentiable and each $\mathbf{z}^{(j)}$ is fixed, this estimate is a differentiable function of θ. Following differentiable policy learning (Freeman et al., 2021), we compute its gradient by backpropagation through time (BPTT) and update θ with Adam (Kingma & Ba, 2015). At deployment, the sampled latent is replaced by the encoder's estimate $\mathbf{z}_t = E_\phi(\mathcal{H}_{t-C:t})$, recomputed at every control step from the most recent C state-action pairs. The policy itself stays the same: it always receives a task observation and a latent, which is either sampled (during training) or inferred from recent trajectory history (during deployment).

4 EXPERIMENTS AND RESULTS

Our experiments evaluate whether the proposed framework enables continual policy learning on real robots. We organize them around five questions. (Q1) Does the loop improve a real robot's policy from deployment data, including under dynamics that the nominal model does not capture? (Q2) When a condition recurs, does the robot recognize it faster than methods that refit a residual model? (Q3) How does the resulting policy compare with state-of-the-art adaptive controllers? (Q4) Does training on latents sampled from the learned dynamics distribution produce better policies than other ways of using the same data, including beyond aerial robots? (Q5) Does the learned latent space satisfy the two requirements of Section 3, and which model components matter?

Setup. We evaluate our method on a real quadrotor and in simulation. On hardware, we use an Agilicious quadrotor (Foehn et al., 2022) controlled at 50 Hz. Its task is to track a reference trajectory under external disturbances, and we report position error over five trials per setting. In simulation, we evaluate four quadrotor tasks (hovering, trajectory tracking, landing, and vision-based hovering) and two manipulation tasks. Architectures and training settings are listed in Table 4, and task and reward details in Appendices A.2 and A.3. On hardware, the encoder and policy run off-board at 50 Hz, while model and policy updates run in parallel with flight on one NVIDIA RTX 4090 GPU. The encoder is a GRU with 128 hidden units that reads the last $C = 20$ state-action pairs (0.4 s) and outputs a 12-dimensional latent. The residual model is an MLP with two residual blocks of width 256, conditioned on the latent through FiLM. The policy is an MLP with two hidden layers of 512 units, trained with BPTT and Adam. Each round warm-starts from the previous one and runs 500 model updates and 50 policy epochs in 9.7 s this real-time budget applies only to the hardware loop.

Baselines. We compare against two groups of baselines, each answering a different question. The first group asks how our method compares with existing ways of handling hidden dynamics (Section 4). Base DiffSim is the base policy trained on f_{prior} alone, without any adaptation, it is also the starting point of our loop. $\mathcal{L}_1$-MPC (Hanover et al., 2021) is a classical adaptive controller that relies on a predefined model structure. RMA (Kumar et al., 2021) and DATT (Huang et al., 2023) infer the condition from recent history, but are trained in simulation on a designer-chosen range of conditions and cannot learn online. LOTF (Pan et al., 2025) learns from real data but refits a single residual dynamics model whenever conditions change.

The second group asks which part of our design matters for policy training. These baselines use our policy-learning pipeline and change only the conditions the policy is trained on. *GT Dist.* is an oracle trained on the true test-time disturbances. *Simple Residual* fits one residual model without a latent, so it cannot distinguish conditions. *Single Dist.* trains on a single disturbance condition, so it sees no variation. *RMA* adaptation trains on a designer-chosen distribution with privileged parameters, instead of one learned from data. For a fair comparison, all learned methods share the same state and action interfaces, reward, training and evaluation splits, and disturbance settings.

Continual refinement under intermittent wind (Q1). Starting from the base policy, the quadrotor tracks the figure-eight trajectory while the wind is repeatedly switched on and off, and the loop updates the model and policy from the collected flight data in successive rounds. As described in Section 3, each round, from collecting data to deploying the improved policy, takes about 10 s. Tracking error decreases from about 41 cm to about 9 cm within about 50 s of flight, that is, within roughly five rounds (Figure 2, middle), and the tracked trajectory becomes visibly closer to the reference (Figure 2, bottom). Videos of all hardware experiments are included in the supplementary material.

Adaptation to an unmodeled hardware fault (Q1). Wind is commonly approximated as an external force on the rigid body, so a range of wind conditions can be covered in simulation-based training through randomized disturbances, even if the approximation is imperfect. Hardware faults, in contrast, alter the actuation of the robot itself and are rarely captured by the nominal model or by predefined randomization ranges. To test whether our loop can learn such changes, we cut one of the propellers, which reduces the thrust of that rotor and makes the actuation asymmetric. We keep the nominal model unchanged, so the effect of the fault can only be learned from flight data. Using the same loop as in the wind experiments, tracking error drops from 52 cm to 24 cm within the first two rounds and continues to decrease to 12 cm, a 77% reduction overall.

Recovery from recurring wind (Q2). The experiments above show that the loop learns a new condition within a few rounds of about 10 s each. When a condition the robot has already learned returns, however, no further learning should be needed.

To test this, we switch the wind between previously encountered conditions (from calm air to wind, and from one wind direction to another) after the robot reaches steady tracking, and measure the recovery time, the time until the tracking error returns to and stays within its pre-switch range. Over five trials per switch, our policy recovers in 1.1 ± 0.2 s by inference alone: the encoder updates $\mathbf{z}$ from the recent context window, and the policy adjusts its behavior without retraining. LOTF keeps no memory of earlier conditions, so it must refit its residual model after every switch and needs 5.3 ± 0.6 s, making our method about $5\times$ faster (error traces, per-trial times, and the LOTF protocol in Appendix A.5). Across both switches, our method therefore recovers about $5\times$ faster, showing that recognizing a known condition is faster than relearning it.

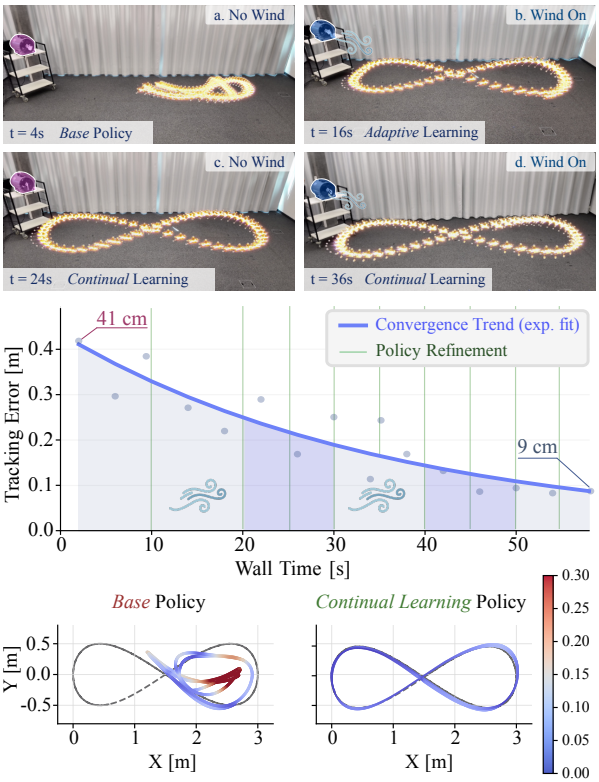


Figure 2: **Real-world policy refinement.** On hardware, continual policy learning reduces figure-eight tracking error from 41 cm for the base policy to 9 cm after refinement. The bottom panels show the flown xy trajectories before and after real-world learning. After adaptation, the quadrotor follows the reference much more closely. The supplementary video shows the full flight in real time.

Comparison with adaptive controllers (Q3). Table 1 compares our method with the first group of baselines in simulation from 8 trials, on hovering and trajectory tracking, each under small and large disturbances. Our method achieves the lowest error in three settings and ties with LOTF, the strongest baseline, on small-disturbance hovering. The differences are largest under large disturbances. Here, our method reduces the hover error of LOTF from 0.105 m to 0.036 m and its tracking error from 0.137 m to 0.064 m, improvements of 65.7% and 53.3%. Methods that rely on a simulation-defined condition range (RMA, DATT) or a fixed model structure ($\mathcal{L}_1$-MPC) degrade even more: their large-disturbance hover errors are 6–18$\times$ higher than ours, and DATT crashes in large-disturbance tracking. These results match the motivation of our method: the further the dynamics move from the nominal model, the greater the benefit of learning from the conditions the robot actually encounters.

Extension to manipulation (Q4). We further apply our method to two simulated manipulation tasks with a Franka arm (Figure 3). In *end-effector tracking*, the hidden condition is the mass of a payload, which changes the effective arm dynamics. The policy observes only the robot state and the task reference, so the payload must be inferred from recent state-action history. The policy is trained with BPTT using 256 parallel environments: the base policy trains in about 10 minutes, a refined policy in about 5 minutes, and fitting the residual model on 50k transitions takes 15–20 s. In *planar box pushing*, the hidden condition is the object mass and friction, which change the contact dynamics. Box-pushing policies are trained with PPO using 8192 parallel environments and 1.8×10^9 simulator steps, which takes about 3 hours per policy. We report Cartesian tracking error for end-effector tracking and success rate for box pushing, all from 20 rollouts. In the settings of Figure 3, our policy reaches 7.72 mm tracking error and a 70% pushing success rate, compared with 44.24 mm and 40% for the best baseline, Simple Residual. Across payload and box masses, our method stays close to the oracle, whereas RMA degrades as the mass grows and Simple Residual remains far from the oracle throughout (Figure 4).

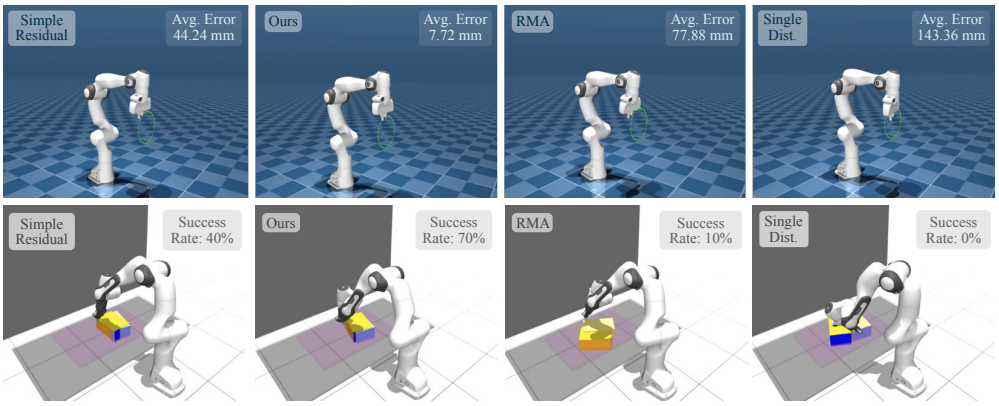


Figure 3: **Manipulation tasks.** Top: end-effector tracking with a hidden payload, with the average tracking error of each method. Bottom: planar box pushing with hidden object mass and friction, with the success rate of each method. Both tasks use the same continual learning loop as the quadrotor.

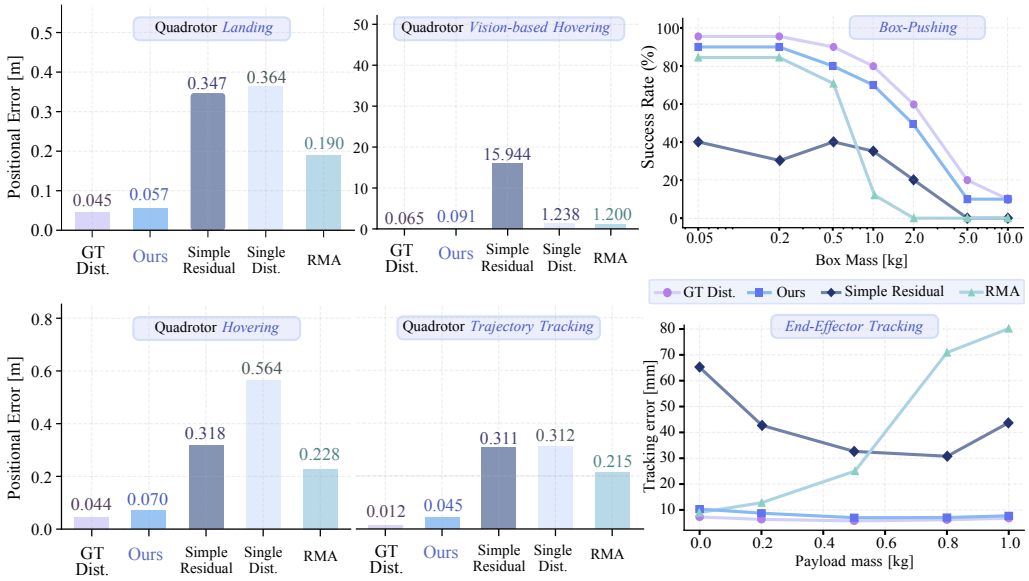


Figure 4: **Quantitative benchmark comparisons.** Across different platforms and tasks, the proposed latent dynamics framework approaches the ground-truth-disturbance oracle and outperforms residual, single-condition, and latent-adaptation baselines. Lower error is better except for box-pushing success rate.

Learning from the condition distribution (Q4). The previous experiments show that our method outperforms existing controllers. We next ask whether this benefit comes from how our method uses deployment data, using our learned condition distribution. Simulation lets us answer this question with an oracle that knows the true conditions. We compare against the second group of baselines on the simulated tasks for both quadrotor and manipulation (Figure 4). As a result, our method is closest to the GT Dist. oracle on all the tasks. In trajectory tracking, it reduces position error from 0.311–0.312 m for Simple Residual and Single Dist. to 0.045 m, and improves over RMA from 0.215 m to 0.045 m. Since the methods differ only in how their training conditions are obtained, this comparison isolates the effect of our learned dynamics distribution. Sampling from it reduces error by 3–13× relative to the best baseline on every task, even though that baseline, RMA, is trained with privileged disturbance parameters that our method never sees.

Method	Hover		Tracking	
	Small↓	*Large*↓	*Small*↓	*Large*↓
Base DiffSim	0.328	1.228	0.492	1.363
RMA	0.132	0.641	0.195	0.258
L1-MPC	0.134	0.552	0.121	0.281
DATT	0.009	0.231	0.082	*crash*
LOTF	0.008	0.105	0.045	0.137
Ours	0.008	0.036	0.042	0.064

Table 1: **Simulation Benchmark Comparison**. Position errors in meters over eight rollouts per setting, following the benchmark of Pan et al. (2025): small and large denote constant acceleration disturbances of $[0.5, 0.5, 0.5]$ and $[2, 2, 2]$ m/s^2. Best in green, second in orange.

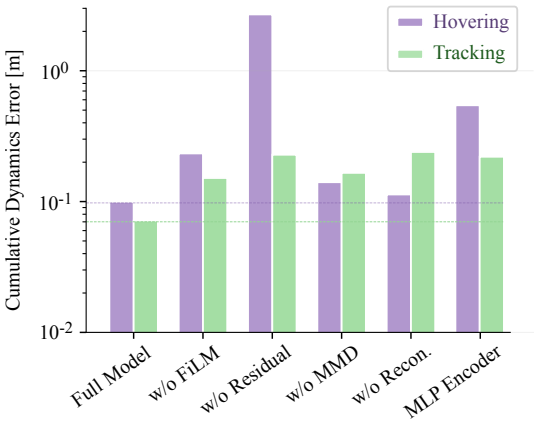


Figure 5: **Dynamics-model ablations.** The full model achieves the lowest cumulative rollout error on hovering and tracking, confirming the importance of latent-conditioned residual dynamics.

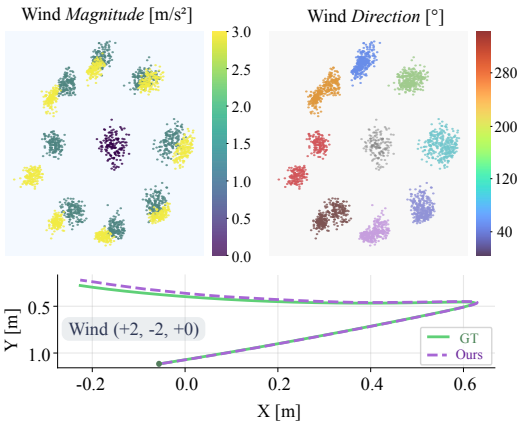


Figure 6: **Latent dynamics analysis.** Top: encoded latents projected to two dimensions with PCA, clustered by wind direction and magnitude. Bottom: the learned dynamics accurately reproduce a representative wind-conditioned open-loop rollout.

Ablation studies (Q5). Figure 5 reports the cumulative open-loop prediction error of the dynamics model over 40-step rollouts on hovering and trajectory tracking when individual design choices from Section 3 are removed. The full model gives the lowest error in both tasks. Removing the physics prior (w/o Residual) causes the largest increase, especially in hovering, confirming that learning only the discrepancy is easier than learning the full transition. Replacing the recurrent encoder with an MLP also substantially degrades hovering, confirming that recent history is important for inferring the hidden condition. Removing FiLM conditioning, MMD alignment, or the reconstruction term also increases error. Each component therefore contributes to accurate condition-specific prediction.

Latent space analysis (Q5). Finally, we examine whether the learned latent space has the two properties our method relies on: the latent must be inferable from recent state-action history, and latents sampled from $p(\mathbf{z})$ must produce realistic dynamics (Section 3). We use simulated wind conditions, whose true values are known (Appendix A.2). Although wind labels are never used in training, encoded context windows form clear clusters by wind direction and magnitude (Figure 6, top), so the encoder recovers the hidden condition from history, satisfying the first property. Given the same initial state and action sequence, an open-loop rollout of the latent-conditioned model stays within 0.1 m of the ground-truth trajectory (Figure 6, bottom), so the latent selects the correct condition-specific dynamics. The second property is supported by the policy results for Q4: policies trained only on sampled latents perform close to the oracle, which would not be possible if the sampled latents produced unrealistic dynamics.

5 Conclusion and Limitations

We presented a continual policy learning framework for robots whose dynamics change in ways they cannot directly observe. Rather than treating deployment as the end of learning, our method uses deployment data to learn the distribution of conditions the robot encounters and to continually improve the policy in response to them. Its core is a latent-conditioned residual dynamics model on top of an analytical nominal model, whose latent space is both inferable from recent deployment history and sampleable during policy training. On a real quadrotor, the loop reduced tracking error from 41 cm to 9 cm within about 50 s of flight, reduced the error under a damaged propeller by 77%, and recovered from recurring wind about 5× faster than online residual refitting. Simulation studies on aerial and manipulation tasks further showed that training on the learned dynamics distribution brings the policy close to an oracle that knows the true conditions.

Limitations. Our real-world experiments rely on accurate state estimates from motion capture. Onboard estimates from visual-inertial odometry or SLAM are noisy, delayed, and sometimes biased. Since the same estimates are used to learn the residual dynamics, sensor errors can be mistaken for unmodeled dynamics, leading to incorrect policy updates. Uncertainty-aware learning, which weights dynamics updates by state-estimation confidence, can be a promising future direction.

AI USE STATEMENT

In this work, we used generative AI tools to polish the writing, assist with literature summarization, and improve the efficiency of the training code. We did not use generative AI to formulate research ideas, theoretical claims, hypotheses, methodology, experiments, or data analysis. All AI-assisted text and code were reviewed and verified by the authors, who take full responsibility for the content.

ETHICS STATEMENT

This work presents no ethical concerns under the Code of Ethics, as it does not involve human subjects, sensitive data, privacy or security risks, harmful applications, or conflicts of interest.

REPRODUCIBILITY STATEMENT

The method and learning loop are described in Section 3 and Algorithm 1. Appendix A provides the experimental protocols, rewards, implementation details, hyperparameters, evaluation metrics, and compute information needed for reproduction. We will release the training and deployment code upon publication.

REFERENCES

Elie Aljalbout, Jiaxu Xing, Angel Romero, Iretiayo Akinola, Caelan Reed Garrett, Eric Heiden, Abhishek Gupta, Tucker Hermans, Yashraj Narang, Dieter Fox, et al. The reality gap in robotics: Challenges, solutions, and best practices. *Annual Review of Control, Robotics, and Autonomous Systems*, 9, 2025.

Cheng Chi, Zhenjia Xu, Siyuan Feng, et al. Diffusion policy: Visuomotor policy learning via action diffusion. *International Journal of Robotics Research*, 2025.

Finale Doshi-Velez and George Konidaris. Hidden parameter markov decision processes: A semi-parametric regression approach for discovering latent task parametrizations. In *IJCAI: proceedings of the conference*, volume 2016, pp. 1432, 2016.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pp. 1126–1135. PMLR, 2017.

Philipp Foehn, Elia Kaufmann, Angel Romero, Robert Penicka, Sihao Sun, Leonard Bauersfeld, Thomas Laengle, Giovanni Cioffi, Yunlong Song, Antonio Loquercio, and Davide Scaramuzza. Agilicious: Open-source and open-hardware agile quadrotor for vision-based flight. *Science Robotics*, 7(67), 2022. doi: 10.1126/scirobotics.abl6259.

C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax – a differentiable physics engine for large scale rigid body simulation. *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Ismail Geles, Leonard Bauersfeld, Angel Romero, Jiaxu Xing, and Davide Scaramuzza. Demonstrating agile flight from pixels without state estimation. *Robotics: Science and Systems*, 2024.

Arthur Gretton, Karsten M Borgwardt, Malte J Rasch, Bernhard Schölkopf, and Alexander Smola. A kernel two-sample test. *Journal of Machine Learning Research*, 13(1):723–773, 2012.

Drew Hanover, Philipp Foehn, Sihao Sun, Elia Kaufmann, and Davide Scaramuzza. Performance, precision, and payloads: Adaptive nonlinear mpc for quadrotors. *IEEE Robotics and Automation Letters*, 7(2):690–697, 2021.

Kevin Huang, Rwik Rana, Alexander Spitzer, Guanya Shi, and Byron Boots. DATT: Deep adaptive trajectory tracking for quadrotor control. In *Conference on Robot Learning*, 2023.

Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, et al. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26), 2019.

Elia Kaufmann, Leonard Bauersfeld, Antonio Loquercio, Matthias Müller, Vladlen Koltun, and Davide Scaramuzza. Champion-level drone racing using deep reinforcement learning. *Nature*, 620(7976):982–987, Aug 2023. ISSN 1476-4687.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. Rma: Rapid motor adaptation for legged robots. *Proceedings of Robotics: Science and Systems (RSS)*, 2021.

Ashish Kumar, Zhongyu Li, Jun Zeng, Deepak Pathak, Koushil Sreenath, and Jitendra Malik. Adapting rapid motor adaptation for bipedal robots. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1161–1168. IEEE, 2022.

Joonho Lee, Jemin Hwangbo, Lorenz Wellhausen, Vladlen Koltun, and Marco Hutter. Learning quadrupedal locomotion over challenging terrain. *Science Robotics*, 5(47), 2020.

Takahiro Miki, Joonho Lee, Jemin Hwangbo, Lorenz Wellhausen, Vladlen Koltun, and Marco Hutter. Learning robust perceptive locomotion for quadrupedal robots in the wild. *Science Robotics*, 7(62), 2022.

Michael O'Connell, Guanya Shi, Xichen Shi, Kamyar Azizzadenesheli, Anima Anandkumar, Yisong Yue, and Soon-Jo Chung. Neural-fly enables rapid learning for agile flight in strong winds. *Science Robotics*, 7(66):eabm6597, 2022.

Jiahe Pan, Jiaxu Xing, Rudolf Reiter, Yifan Zhai, Elie Aljalbout, and Davide Scaramuzza. Learning on the fly: Rapid policy adaptation via differentiable simulation. *IEEE Robotics and Automation Letters*, 2025.

Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE international conference on robotics and automation (ICRA)*, pp. 3803–3810. IEEE, 2018.

Ethan Perez, Florian Strub, Harm De Vries, Vincent Dumoulin, and Aaron Courville. Film: Visual reasoning with a general conditioning layer. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Haozhi Qi, Ashish Kumar, Roberto Calandra, Yi Ma, and Jitendra Malik. In-hand object rotation via rapid motor adaptation. In *Conference on Robot Learning*, pp. 1722–1732. PMLR, 2023.

Kate Rakelly, Aurick Zhou, Chelsea Finn, Sergey Levine, and Deirdre Quillen. Efficient off-policy meta-reinforcement learning via probabilistic context variables. In *International conference on machine learning*, pp. 5331–5340. PMLR, 2019.

Yunfan Ren, Zhiyuan Zhu, Jiaxu Xing, and Davide Scaramuzza. Learning agile quadrotor flight in the real world. In *arXiv Preprint*, 2026.

Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. Learning to walk in minutes using massively parallel deep reinforcement learning. In *Conference on robot learning*, pp. 91–100. PMLR, 2022.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pp. 1889–1897. PMLR, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *Proc. IROS*, 2017.

Hongze Wang, Jiaxu Xing, Nico Messikommer, and Davide Scaramuzza. Environment as policy: Learning to race in unseen tracks. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 11333–11339. IEEE, 2025.

Jiaxu Xing, Ismail Geles, Yunlong Song, Elie Aljalbout, and Davide Scaramuzza. Multi-task reinforcement learning for quadrotors. *IEEE Robotics and Automation Letters*, 10(3):2112–2119, 2024.

Wenhao Yu, Jie Tan, C Karen Liu, and Greg Turk. Preparing for the unknown: Learning a universal policy with online system identification. *arXiv preprint arXiv:1702.02453*, 2017.

A SUPPLEMENTARY MATERIALS

These supplementary materials provide implementation details, experimental protocols, and additional results for the main paper.

A.1 OVERVIEW

The supplementary materials are organized as follows.

- **Experimental setup.**
  - Section A.2 (Simulation Experiment Setup) describes the simulated quadrotor and manipulation tasks.
  - Section A.3 (Task Rewards) defines the task rewards.
  - Section A.5 (Hardware Setup) describes the real-world setup and the protocols of the hardware experiments.
- **Implementation and evaluation.**
  - Section A.6 (Training Curves and Compute) reports training curves and compute.
  - Section A.7 (Metrics and Statistical Reporting) defines the evaluation metrics.
  - Section A.8 (State, Action, and History Representation) specifies the state, action, and history representation and lists the model and training parameters.
- **Method details.**
  - Section A.4 (Policy Learning and Deployment) details policy optimization and deployment.
  - Section A.9 (Variational Neural Dynamics Details) details the latent dynamics model, its training objective, and the ablations.

A.2 SIMULATION EXPERIMENT SETUP

All simulated tasks follow the same template as the hardware loop. Starting from a base policy trained on the nominal model f_{prior}, the robot collects state-action transitions under hidden conditions. The buffer $\mathcal{B}$ trains the latent-conditioned residual dynamics model, and the policy is then optimized on latents sampled from $p(\mathbf{z})$ before evaluation on held-out hidden conditions. The tasks differ only in the robot model, the hidden condition, and the metric.

Quadrotor dynamics. The quadrotor state follows the main-paper convention $\mathbf{s} = (\mathbf{p}, \mathbf{q}, \mathbf{v}) \in \mathbb{R}^{10}$, where $\mathbf{q}$ is a unit quaternion and $R(\mathbf{q})$ is the corresponding rotation matrix. The action consists of the collective thrust T and the commanded body rates $\boldsymbol{\omega}$. The nominal model f_{prior} is

$$\dot{\mathbf{p}} = \mathbf{v}, \qquad \dot{\mathbf{v}} = \mathbf{g} + \frac{T}{m} R(\mathbf{q})\mathbf{e}_3, \qquad \mathbf{q}_{t+\Delta t} = \mathbf{q}_t \otimes \mathrm{Exp}_{\mathbb{H}}(\Delta t\, \boldsymbol{\omega}), \tag{2}$$

where $\mathrm{Exp}_{\mathbb{H}}$ maps a rotation vector to a unit quaternion. Position and velocity are integrated with RK4, and attitude is updated by quaternion composition through the exponential map. In the wind experiments, the simulated dynamics additionally contain the hidden condition $\mathbf{w}$ as a constant external acceleration,

$$\dot{\mathbf{v}} = \mathbf{g} + \mathbf{w} + \frac{T}{m} R(\mathbf{q})\mathbf{e}_3, \tag{3}$$

which is fixed within an episode, consistent with the piecewise-constant assumption, and never observed by the policy. The full-quadrotor configuration uses the Kolibri parameters: mass $0.192\,\mathrm{kg}$, per-motor thrust limit $3.5\,\mathrm{N}$, arm offsets $(\pm 0.041, \pm 0.041, 0)\,\mathrm{m}$, inertia $(1.42, 1.65, 2.13)\times 10^{-4}\,\mathrm{kg\,m^2}$, and motor time constant $0.0245\,\mathrm{s}$.

Quadrotor wind conditions. We use 17 steady wind conditions: calm flight and eight horizontal wind directions at two magnitudes, 1.5 and $3.0\,\mathrm{m/s^2}$. The wind labels are never used by the policy or by the latent dynamics objective; we use them only for evaluation and for the visualizations in Figures 6, 8, and 9.

Quadrotor hovering. Hovering rollouts start near a fixed target with randomized position, velocity, and attitude perturbations. The residual model predicts position and velocity corrections. We report the average position error under held-out wind conditions.

Quadrotor trajectory tracking. The quadrotor tracks a figure-eight reference from initial states randomized near the reference. The residual model predicts position, velocity, and orientation corrections. We report the mean tracking error with respect to the reference.

Landing and vision-based hovering. Landing uses the same pipeline and is evaluated by terminal position error and success criteria. Vision-based hovering uses a feature-based observation interface instead of explicit state estimation and follows the pretraining strategy of (Pan et al., 2025); we report position error.

Manipulation tasks. The manipulation tasks (main paper, Figure 3) use a simulated Franka arm. In end-effector tracking, the hidden condition is a payload mass between 0 and 1 kg (11 values), and each episode lasts 5 s. In planar box pushing, the hidden condition is the box mass, between 0.05 and 10 kg, together with friction-induced deviations from the nominal model. Box pushing uses the same data-to-latent-to-policy pipeline as every other task. The arm collects state-action transitions under hidden box masses and friction; the encoder E_ϕ and the latent-conditioned residual model D_ψ are trained on this buffer with $\mathcal{L}_{\text{vnd}}$; and the policy is then trained on latents sampled from $p(\mathbf{z})$, with D_ψ correcting the nominal simulator and the same latent conditioning the policy. The only difference is the policy optimizer: because contact makes the rollouts non-smooth, the policy is optimized with PPO rather than BPTT, so the learned model serves as a sampled-condition simulator without differentiating through it. At test time, the latent is inferred by the encoder from the recent state-action history, as in all other tasks. The encoder and residual settings are those of the Franka column in Table 4, and the PPO settings are listed in the Push PPO column.

A.3 Task Rewards

All methods within a task use the same reward. We write Δt for the control timestep and $\Delta \mathbf{a}_t = \mathbf{a}_t - \mathbf{a}_{t-1}$ for the action change. The reward terms use the position $\mathbf{p}$ (with height p_z), linear velocity $\mathbf{v}$, linear acceleration $\dot{\mathbf{v}}$, body rate $\boldsymbol{\omega}$, and attitude $R \in \mathrm{SO}(3)$. The per-step reward is the negative time-weighted cost, $r = -\Delta t\, c$, where the task cost c is a fixed, hand-tuned linear combination of penalty terms $c_{(\cdot)}$. Unless otherwise stated, every continuous term is a Huber penalty

$$\ell_\delta(e) = \begin{cases} \frac{1}{2}e^2, & |e| \le \delta, \\ \delta\big(|e| - \frac{1}{2}\delta\big), & |e| > \delta, \end{cases} \tag{4}$$

applied to the relevant error e (e.g., $c_v = \ell_\delta(\|\mathbf{v}\|)$), which keeps gradients bounded for large errors while remaining quadratic near zero.

Quadrotor hovering. The hovering cost penalizes the target-height error (c_{height}), linear velocity (c_v), body rate (c_ω), linear acceleration ($c_{\dot{v}}$), attitude deviation from upright (c_R), action deviation from hover thrust (c_a), and collisions ($c_{\text{collision}}$):

$$c_{\text{hover}} = c_{\text{height}} + 0.1\, c_v + 0.15\, c_\omega + 0.1\, c_{\dot{v}} + 0.1\, c_R + c_a + c_{\text{collision}}. \tag{5}$$

Quadrotor landing. Landing uses the hovering structure but replaces the height term with the position error to the landing target, c_p, keeping all other terms and weights:

$$c_{\text{land}} = c_p + 0.1\, c_v + 0.15\, c_\omega + 0.1\, c_{\dot{v}} + 0.1\, c_R + c_a + c_{\text{collision}}. \tag{6}$$

Quadrotor trajectory tracking. Trajectory tracking penalizes position, velocity, body-rate, acceleration, and attitude errors relative to the current reference waypoint, together with action-smoothness and collision terms:

$$c_{\text{track}} = c_p + 0.1\, c_v + 0.01\, c_\omega + 0.01\, c_{\dot{v}} + c_R + 0.15\, c_{\Delta a} + c_{\text{collision}}. \tag{7}$$

All tracking methods use the same reference, action limits, reset distribution, and reward in simulation and on hardware.

Franka end-effector tracking. The tracking cost combines the Cartesian position error (c_p), orientation error (c_R), action regularization (c_a), and a joint-limit penalty (c_{lim}) over a 5 s episode. Continuous errors pass through $\exp(-\beta e)$ with sharpness $\beta = 5.0$. The weights are $w_p = 3.0$, $w_R = 0.3$, $w_{\dot{q}} = 0$, $w_a = 0.01$, and $w_{\text{lim}} = 1.0$, where $\dot{q}$ denotes the joint velocity; the joint-limit term activates within 0.05 rad of a limit. These settings make Cartesian tracking dominate the gradient while avoiding unnecessary damping of sustained joint motion.

Box pushing. Unlike the tasks above, box pushing is written directly as a reward to be maximized by PPO: a weighted sum of shaped terms for box–target progress, box orientation, gripper–box proximity, side-contact shaping, joint velocity and its limit, robot target posture, and action regularization, plus a success bonus and a termination penalty,

$$\begin{aligned} r_{\text{box}} = {} & w_{\text{prog}}\, r_{\text{prog}} + w_{\text{ori}}\, r_{\text{ori}} + w_{\text{prox}}\, r_{\text{prox}} + w_{\text{contact}}\, r_{\text{contact}} + w_{\dot{q}}\, r_{\dot{q}} + w_{\dot{q}\text{-lim}}\, r_{\dot{q}\text{-lim}} \\ & + w_{\text{posture}}\, r_{\text{posture}} + w_{\Delta a}\, r_{\Delta a} + w_a\, r_a + r_{\text{succ}} - r_{\text{term}}, \end{aligned} \tag{8}$$

where the negative weights on the two action terms make them penalties. The setup uses $\Delta t = 0.005$ s, action repeat 4, observation history length 30, action history length 5, and delayed, noisy observations and actions during training. The reward scales are $w_{\text{prog}} = 8.0$, $w_{\text{ori}} = 6.0$, $w_{\text{prox}} = 2.0$, $w_{\text{contact}} = 1.0$, $w_{\dot{q}} = 1.0$, $w_{\dot{q}\text{-lim}} = 3.0$, $w_{\text{posture}} = 0.75$, $w_{\Delta a} = -0.1$, and $w_a = -0.1$. Success adds 500.0 after 30 consecutive success steps, with a 3.0 success-wait reward, and termination adds -50.0. The shaped terms use a 0.005 m box–target tolerance with 0.4 m margin, a 0.2 rad orientation tolerance with π margin, and a 0.1 m gripper–box side-contact tolerance with 1.0 m margin.

A.4 Policy Learning and Deployment

During policy optimization, the encoder and dynamics model are frozen. At the start of each simulated rollout j, a latent $\mathbf{z}^{(j)} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ is sampled and held fixed, and the same latent conditions both the residual model and the policy:

$$\mathbf{a}_t = \pi_\theta(\bar{\mathbf{o}}_t, \mathbf{z}^{(j)}), \qquad \hat{\mathbf{s}}_{t+1} = f_{\text{prior}}(\hat{\mathbf{s}}_t, \mathbf{a}_t) \oplus D_\psi(\hat{\mathbf{s}}_t, \mathbf{a}_t, \mathbf{z}^{(j)}). \tag{9}$$

Here $\bar{\mathbf{o}}_t = \text{norm}_o\big(O_{\text{task}}(\hat{\mathbf{s}}_t, \mathbf{x}_t^{\text{ref}}, \mathbf{a}_{t-K:t-1})\big)$ is the normalized task observation, where O_{task} selects the task-specific state, reference, and action-history features and norm_o applies the observation normalization used for policy training. The encoder input remains the separate context window $\mathcal{H}_{t-C:t}$. For box pushing, the policy is instead optimized with PPO on rollouts of the same learned model under sampled latents, without differentiating through it (Section A.2). At deployment, the sampled latent is replaced by the encoder output $\mathbf{z}_t = E_\phi(\mathcal{H}_{t-C:t})$, recomputed at every control step.

Fixed latents in training versus inferred latents at deployment. During policy optimization, each rollout uses a fixed latent $\mathbf{z}$, but the latent is resampled across rollouts from $p(\mathbf{z})$. This exposes the policy to a broad range of dynamics conditions and trains it to respond to different regions of the latent space. At deployment, $\mathbf{z}_t$ is instead inferred online from a sliding history window and may therefore change when the underlying dynamics change. Because the policy has already been trained across the latent distribution, these changes do not require policy adaptation: the encoder only needs to identify the new condition and move $\mathbf{z}_t$ toward the corresponding region of the latent space. With a 20-step context at 50 Hz, the encoder receives a fully refreshed history within 0.4 s after a switch. Our hardware experiments confirm that this inference-based transition is fast: across 20 disturbance switches, the policy recovers within 1.1 ± 0.2 s without additional policy learning. Thus, latent randomization during training provides robustness across dynamics conditions, while online inference enables rapid switching between them at deployment.

A.5 Hardware Setup

The hardware experiments use an Agilicious-based quadrotor (Foehn et al., 2022) controlled at 50 Hz with motion-capture state estimates; the encoder, policy, and learning loop run off-board. Unless stated otherwise, the quadrotor tracks a figure-eight reference, wind is generated externally by a fan and never observed by the policy, and position error is reported as mean $\pm$ standard deviation over five trials per setting, with the same reference and evaluation window for all methods.

Switch	Trial	Recovery time [s] ↓ Ours	LOTF	Speedup ↑
No Wind→Wind	1	0.9	4.8	5.3×
	2	1.2	5.5	4.6×
	3	1.0	5.1	5.1×
	4	1.4	6.0	4.3×
	5	0.8	4.6	5.8×
Top Wind→Bottom Wind	1	1.1	5.3	4.8×
	2	1.3	5.9	4.5×
	3	0.9	4.7	5.2×
	4	1.5	6.2	4.1×
	5	1.0	4.9	4.9×
Mean ± std		$\mathbf{1.1 \pm 0.2}$	5.3 ± 0.6	$\mathbf{4.9 \pm 0.5}\times$

Table 2: **Recovery from recurring wind on hardware, per trial.** The wind switches between two previously learned conditions, after the quadrotor reaches steady tracking. Recovery time is the time until the tracking error returns to and stays within its pre-switch envelope (Eq. (10)). Our policy recovers by inference alone, whereas LOTF refits its residual model after every switch. Speedup is LOTF's recovery time divided by ours.

Recovery from recurring wind: protocol. Before these trials, the loop has already learned the conditions used here: calm air and fan-generated wind from two directions (top and bottom). In each trial, the quadrotor tracks the figure-eight under one condition until its tracking error is steady, and at time t_{switch} the wind abruptly switches to another previously learned condition. We run five trials per switch (calm air to wind, and top wind to bottom wind) for each method, 20 trials in total. During these trials, our method uses the encoder, residual model, and policy from the last completed round, and the learning loop is paused, so recovery relies on inference alone. Let $e(t) = \|\mathbf{p}(t) - \mathbf{p}^{\text{ref}}(t)\|_2$ be the tracking error and $\bar{e}(t)$ its average over the trailing window $[t - w,\, t]$ with $w = 0.5\,\text{s}$. The pre-switch envelope is $\epsilon = (1 + m)\max_{t \in [t_{\text{switch}} - W,\, t_{\text{switch}})} \bar{e}(t)$, with margin $m = 0.1$ and a window W that covers at least one full lap of the reference. The recovery time is

$$T_{\text{rec}} = \inf\{\tau \geq 0 \mid \bar{e}(t_{\text{switch}} + \tau') \leq \epsilon \;\; \forall\, \tau' \in [\tau,\, T_{\max}]\}, \tag{10}$$

evaluated over a horizon of $T_{\max} = 15\,\text{s}$ after the switch. The trailing average uses no future samples, so no post-switch data enter the envelope; together with the margin, it prevents measurement noise from being counted as a failure to recover. Both settings are fixed in advance and identical for all methods.

LOTF protocol. LOTF (Pan et al., 2025) runs with its published settings on the same off-board GPU and receives the same flight data stream as our method. Before each trial, its residual model is fitted to the pre-switch condition, so both methods start from steady tracking under the same condition. After the switch, LOTF refits its residual model on the most recent flight data and refines its policy with its published update schedule, after which the refitted policy is deployed immediately. Both methods fly the same reference under the same wind conditions and switch protocol.

Recovery results. Table 2 lists the recovery time of every trial. Our method recovers in $1.1 \pm 0.2\,\text{s}$ and LOTF in $5.3 \pm 0.6\,\text{s}$ (mean ± std), about $5\times$ faster, and our method is faster in every paired trial.

A.6 Training Curves and Compute

Figure 7 shows a representative training curve for the quadrotor trajectory-tracking task, documenting the optimization behavior of one BPTT policy run; the evaluation metrics in the main paper are computed separately on held-out disturbance conditions.

Training from scratch. All training runs on a single NVIDIA RTX 4090 GPU. When trained from scratch, quadrotor hovering and landing policies use 512 parallel environments, 1500 epochs,

Stage	During flight	Work per round	Time [s]
Data collection	continuous	485 transitions (9.7 s)	–
Model update	yes	500 steps	3.5 [3.3, 3.8]
Policy update	yes	50 epochs	5.8 [5.6, 6.1]
Deployment	yes	–	0.4 [0.3, 0.4]
Full round	yes	–	9.7 [9.4, 10.1]
From scratch	no	5000 steps, 1500 epochs	275

Table 3: **Measured timing of one learning round on hardware**, median [interquartile range] over all rounds of the refinement experiment. Data collection runs continuously in the control process; all other stages run in the update process in parallel with flight. The model update trains the encoder and residual model, deployment covers the weight transfer and the swap, and a full round spans snapshot to deployment. The last row shows training from scratch for comparison.

and 250 simulator steps per epoch (1.92×10^8 differentiable transition evaluations, 5 s rollouts at $\Delta t = 0.02$ s); the trajectory-tracking policy uses 125-step rollouts (9.6×10^7 transition evaluations, 2.5 s rollouts). All quadrotor policies are MLPs with hidden sizes $(512, 512)$ and a four-dimensional thrust/body-rate action. For 1000 epochs with 512 environments, representative training times are 130.6 s for a wind task with the simple quadrotor model, 158.3 s with the full quadrotor model, and 188.5 s for latent-conditioned vision-based hovering. A Franka BPTT base policy trains in about 10 minutes and a warm-started refined policy in about 5 minutes; fitting the Franka residual model on 50k transitions takes 15–20 s. Box-pushing PPO policies use 8192 parallel environments and 1.8×10^9 simulator steps and take about 2.8–2.9 hours to train, plus about 45 s of JIT compilation. None of these simulation runs is subject to the real-time budget of the hardware loop.

Timing of one learning round on hardware. The hardware loop runs two processes concurrently. The control process runs the encoder and policy at 50 Hz and appends every transition to $\mathcal{B}$; it never pauses. The update process repeats three stages: it takes a snapshot of $\mathcal{B}$, updates the encoder and residual model, and then updates the policy, with both updates warm-started from the previous round. The new encoder and policy are then sent to the control process and swapped in together between two control steps. We log wall-clock timestamps at every stage boundary and report the median and interquartile range over all rounds of the refinement experiment in Table 3. A round takes 9.7 s from snapshot to deployment, and the flight data collected in the meantime enter the next round. Warm-starting is what makes this possible: a round runs 500 model steps and 50 policy epochs, far fewer than training from scratch (last row of Table 3).

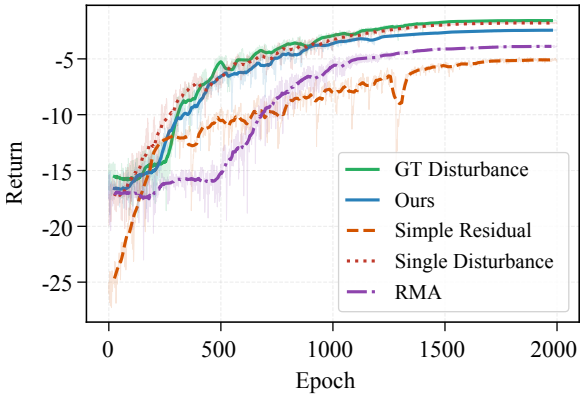


Figure 7: Training curves comparing return versus epoch across disturbance-modeling methods on the quadrotor task.

A.7 Metrics and Statistical Reporting

On hardware, the primary metric is position tracking error in meters over repeated trials, and recovery time is reported in seconds as defined in Section A.5. In simulation, quadrotor tasks report mean position error, landing additionally reports terminal success criteria, end-effector tracking reports Cartesian tracking error in millimeters, and box pushing reports success rate. When aggregate statistics are reported, all methods use the same seed set, disturbance split, and evaluation horizon within a task.

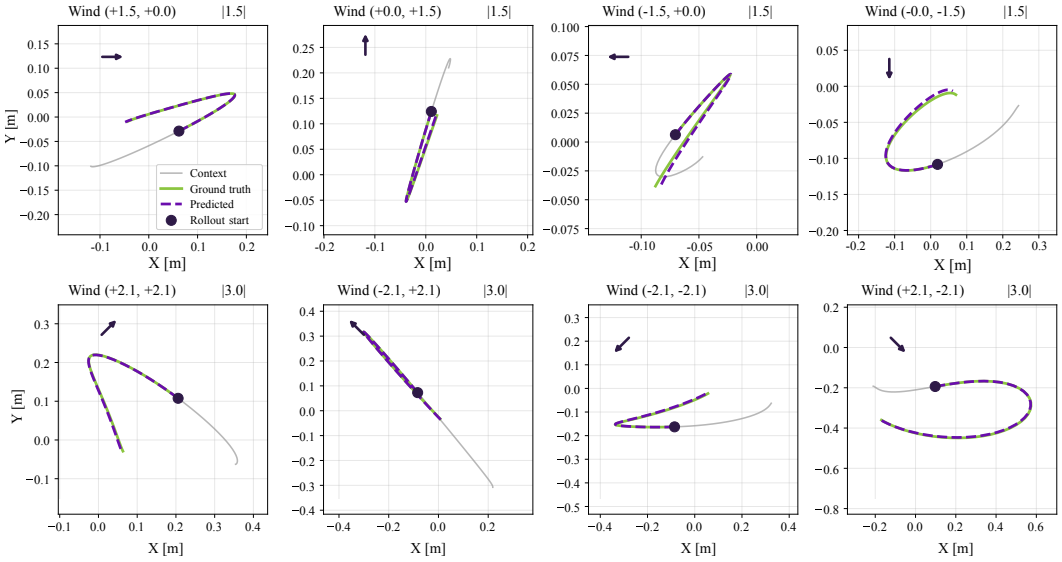


Figure 8: **Open-loop rollouts of the learned dynamics.** Over 40-step open-loop rollouts, the learned latent-conditioned model captures diverse disturbance conditions with low cumulative error.

A.8 State, Action, and History Representation

For the quadrotor experiments, the physical state is

$$\mathbf{s}_t = (\mathbf{p}_t, \mathbf{q}_t, \mathbf{v}_t) \in \mathbb{R}^{10}, \tag{11}$$

where $\mathbf{p}_t$ is position, $\mathbf{q}_t$ is a unit quaternion, and $\mathbf{v}_t$ is linear velocity. The action follows the main-paper formulation,

$$\mathbf{a}_t = (T_t, \omega_{x,t}, \omega_{y,t}, \omega_{z,t}) \in \mathbb{R}^4, \tag{12}$$

where T_t is the collective thrust and the remaining entries are the commanded body rates. The task observation $\mathbf{o}_t = O_{\text{task}}(\mathbf{s}_t, \mathbf{x}_t^{\text{ref}}, \mathbf{a}_{t-K:t-1})$ also includes recent actions, which lets the policy account for command delay and smoothness. Hovering observations contain the current position, quaternion, velocity, and a short action history, and trajectory-tracking observations additionally contain the error with respect to the current reference waypoint. Unless otherwise stated, $C = 20$ control steps, corresponding to $0.4\,\text{s}$ at the $50\,\text{Hz}$ control rate. The quadrotor model uses $\mathbf{z} \in \mathbb{R}^{12}$ and the Franka model $\mathbf{z} \in \mathbb{R}^4$ (Table 4). Input channels are normalized with statistics computed on $\mathcal{B}$, and the same normalization is used for our method and the history-conditioned baselines.

Parameter	Quad. VND	Franka VND	Push PPO
State-action dim.	13–14	21	state hist.
Residual output	6–9	14	14
Context length	20	20	20
Latent dim.	12	4	4
Encoder	GRU	GRU	GRU
Encoder hidden	128	256	256
Residual hidden	256	256	256
Residual blocks	2	3	3
Conditioning	FiLM	FiLM	FiLM
Hidden conditions	17 winds	11 masses	box mass, friction
VND epochs	per task	5000	5000
VND batch	per task	2048	2048
VND LR	per task	3×10^{-4}	3×10^{-4}
Policy hidden	512, 512	256, 256	64×4
Policy optimizer	Adam/BPTT	Adam/BPTT	PPO
Policy epochs	per task	500	–
Policy LR	per task	3×10^{-3}	6×10^{-4}
Horizon	per task	250	3000
Parallel envs	per task	256	8192
Control step	0.02 s	0.02 s	0.005 s

Table 4: **Model and training parameters.** "Quad. VND" lists the architecture shared by the quadrotor experiments; training-loop values are task specific. "Franka VND" lists the end-effector tracking settings, and "Push PPO" lists the box-pushing settings, whose encoder and residual model are identical to the Franka VND column and whose policy is trained with PPO.

A.9 Variational Neural Dynamics Details

Recurrent trajectory encoder.

Residual dynamics model. The residual network consists of residual MLP blocks (Table 4) whose hidden features are modulated by FiLM parameters computed from $\mathbf{z}$. Figure 8 shows 40-step open-loop rollouts of the learned model under eight wind conditions, and Figure 9 shows closed-loop tracking of the resulting policy under the same conditions.

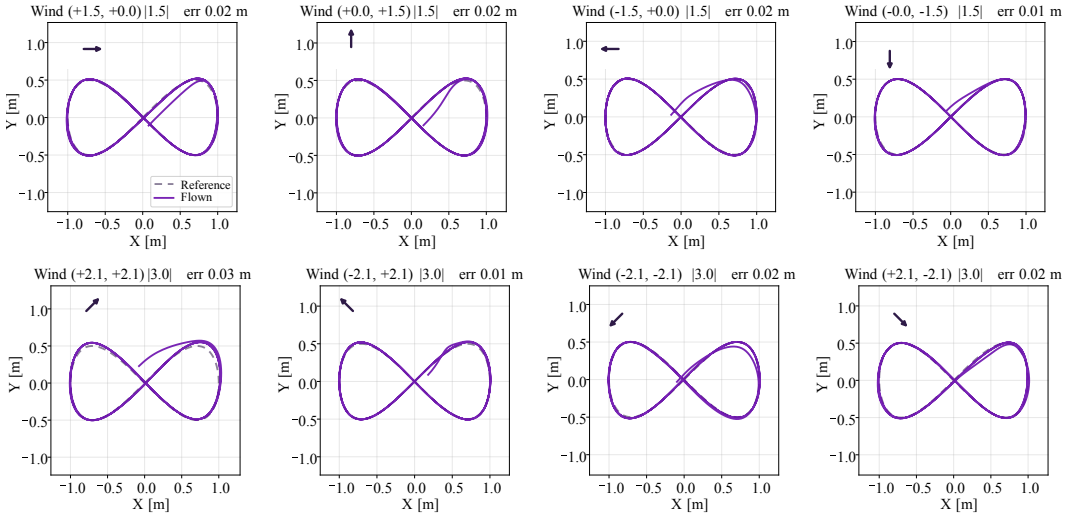


Figure 9: **Policy performance in diverse disturbance conditions.** Our learned policy through the continual learning framework is able to capture diverse external conditions.

MMD estimator. The MMD term aligns the empirical distribution q_ϕ of encoded latents with $p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$. For encoded latents $\{\mathbf{z}_n\}_{n=1}^N$ and prior samples $\{\tilde{\mathbf{z}}_m\}_{m=1}^N$, we use the biased finite-sample estimator

$$\mathcal{L}_{\text{mmd}} = \frac{1}{N^2}\sum_{n,n'} k(\mathbf{z}_n, \mathbf{z}_{n'}) + \frac{1}{N^2}\sum_{m,m'} k(\tilde{\mathbf{z}}_m, \tilde{\mathbf{z}}_{m'}) - \frac{2}{N^2}\sum_{n,m} k(\mathbf{z}_n, \tilde{\mathbf{z}}_m), \tag{13}$$

with the RBF kernel $k(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{\|\mathbf{x}-\mathbf{y}\|_2^2}{2\sigma^2}\right)$ and $\sigma = 2.0$ for the quadrotor experiments. The Gaussian prior is only a convenient sampling space, not a physical wind or payload model.

MMD schedule. The MMD weight is annealed as

$$\lambda_{\text{mmd}}(\kappa) = \lambda_{\text{mmd}}^{\max} \min\left(1, \max\left(0, \frac{\kappa - \kappa_0}{\kappa_{\text{ramp}}}\right)\right), \tag{14}$$

where κ is the latent-dynamics update index, κ_0 is the length of the dynamics-only warm-up, and κ_{ramp} is the ramp length.

Ablations. Table 5 summarizes the question that each ablation in the main paper answers.

Ablation	Question answered
w/o residual prior	Does the model benefit from predicting only the correction to a physics prior, rather than relearning the full transition?
w/o FiLM	Is latent feature-wise modulation useful beyond simple concatenation of $\mathbf{z}$ and the state-action vector?
w/o MMD	Can the encoder learn predictive latents that are also safe to sample from $\mathcal{N}(\mathbf{0}, \mathbf{I})$ without aggregate prior alignment?
w/o reconstruction	Does the auxiliary context reconstruction term help retain trajectory information and improve multi-step prediction?
MLP encoder	Is a recurrent encoder necessary, or can a non-recurrent history summary identify the active condition?

Table 5: **Ablation map.** The full model combines a physics prior, latent-conditioned residual dynamics, recurrent history encoding, context reconstruction, and annealed MMD prior alignment.