DeepRIoT: Continuous Integration and Deployment Of Robotic-IoT Applications

Meixun Qu*,†, Jie He*,†, Zlatan Tucaković†, Ezio Bartocci†, Dejan Ničković‡, Haris Isaković†, Radu Grosu†
Technische Universität Wien†, AIT Austrian Institute of Technology‡

ABSTRACT

We present DeepRIoT, a continuous integration and continuous deployment (CI/CD) based architecture that accelerates the learning and deployment of a Robotic-IoT system trained from deep reinforcement learning (RL). We adopted a multi-stage approach that agilely trains a multi-objective RL controller in the simulator. We then collected traces from the real robot to optimize its plant model, and used transfer learning to adapt the controller to the updated model. We automated our framework through CI/CD pipelines, and finally, with low cost, succeeded in deploying our controller in a real F1tenth car that is able to reach the goal and avoid collision from a virtual car through mixed reality.

KEYWORDS

Deep Reinforcement Learning, Sim2Real, DevOps, CI/CD

1 INTRODUCTION

Deep Reinforcement Learning [1] (RL) has been gaining momentum in the control tasks of Robotic-IoT systems [2–4]. Although RL has achieved state-of-the-art in some simulation benchmarks, there are still many factors that hinder its application in real Robotic-IoT systems, for example, the sample inefficiency problem of RL algorithms, the generalization problem of deep neural networks, the unavoidable noise in the sensors, and the deviation of the simulation from the real system (Sim2Real problem [5]).

Unfortunately, there is no systematic approach that takes the aforementioned problems into consideration, to guide the training and deployment of RL in real Robotic-IoT systems. In this paper, we fill in this blank by proposing DeepRIoT, a practical framework that aims at accelerating the learning and deployment for RL algorithms. To achieve this goal, we leverage the *DevOps* [6] practices that integrate the process of software development (Dev) with the monitoring of the real system during its operation (Ops). In our context, the collection of the execution traces of the real Robotic-IoT running the RL policy is used to improve the models and thus the RL policy in the simulation environment. Using DevOps machinery we can fully automate the integration of model changes w.r.t. the real system execution, RL policy retraining in the simulation environment and its continuous deployment in the real Robotic-IoT. These processes are referred as *Continuous Integration* [7] (CI) and *Continuous Deployment* [8] (CD). To accelerate RL in the simulation environment, we further enhance this process by using models of the real Robotic-IoT of different complexity.

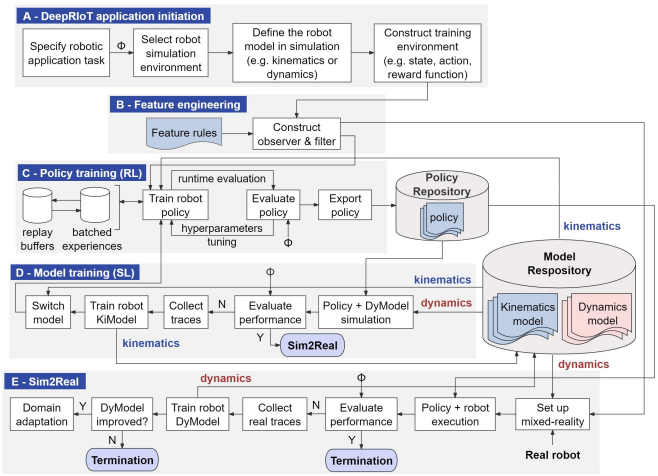


Figure 1: DeepRIoT architecture and workflow

We select a classic use case in motion planning to demonstrate our approach. We perform our experiments using F1tenth [9], an open-source autonomous vehicle platform. Our task is to teach an F1tenth car to reach the goal position from a starting point while avoiding collisions with static obstacles and other vehicles.

DeepRIoT Architecture. We sketch the architecture of DeepRIoT, depicted in Fig. 1. The architecture consists of Pipelines (A-E).

Pipeline A. Here we specify requirements Φ for the given robotic tasks. These requirements are given in the form of formal specifications using Signal Temporal Logic (STL) [10]. There are two major advantages of using formalized requirements. First, one can use runtime monitors to measure the degree to which the observed robotic behaviors satisfy or violate the specifications (see Pipelines C-E). Second, these same specifications are used to engineer the reward function during the agent training process [11] (Pipeline C). After selecting a suitable simulator for training the robotic tasks, we begin to construct the training environment in the context of RL, e.g., defining states, actions, and reward functions.

Pipeline B. This further constructs observers and filters for the states according to pre-defined feature rules. For example, we add Gaussian noises to the LiDAR model to make it more realistic. When we teach the car to avoid collision to obstacles, we only consider laser scans whose lengths are within a specified range.

Pipeline C. This is responsible for training the RL policy. It takes the inputs processed by the previous two pipelines, and starts from picking a simple *kinematics model* (KiModel) to simulate the dynamics of the car. Every step of training yields a four-tuple called *experience* that includes *current state*, *current action*, *next state* and

*Both authors contribute equally to this paper.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06.
https://doi.org/10.1145/3649329.3658250

associated reward. We classify each piece of experience into different types, and store them in different replay buffers according to their types. In each step, the trainer will proportionally sample batched experiences from these replay buffers and use them for training. For faster convergence, we divide the whole training procedure into different stages, and every stage emphasizes different requirements. We set the priority hierarchy by dynamically changing the sampling ratios for different types of experiences, and the weights of different reward functions. With specification Φ, we evaluate the success rate after each training stage, which is defined as the percentage of satisfying Φ among episodes. Finally, we export the satisfactory policy (Base Policy) to a corresponding repository. The success of training in Pipeline C, however, is only based on a simple plant model, very different from the actual dynamics of the car.

Pipeline D. This focuses on adapting the trained policy to richer dynamics. We first pick a *dynamic model* (DyModel) from the model repository, and test the performance of the so-far learned policy (trained with KiModel) on this model. If the policy passes the test, we directly go to the Sim2Real pipeline. Otherwise, we improve the policy and KiModel as follows. We first collect the *traces*, i.e. the sequences of output states from the Base Policy + DyModel collected in each time step. Then we use a nonlinear supervised-learning (SL) method, such as Particle Swarm Optimization [12] (PSO), to optimize the hyper-parameters of the KiModel, making its traces closer to the ones output from the DyModel. Next, we store the new version of the KiModel in the model repository, and go back to Pipeline C to train a second time our policy on the updated KiModel. Afterwards, we test the updated policy (Improved Policy) on the DyModel. *The success means that the new policy has adapted to the DyModel without directly being trained on it.* This further indicates that, without training the policy on the real car, which is extremely expensive, we can continue to update the policy by training it on a DyModel whose traces are as similar as the real car.

Pipeline E. Here we first evaluate the Improved Policy on the real car. If requirements in Φ are satisfied by the observed behaviors, this Sim2Real pipeline terminates. Otherwise, by taking the same approach as in Pipeline D, we collect traces from the car, and optimize the hyper-parameters of DyModel, reducing the difference between its output traces and the real car's ones. The updated DyModel is the best model whose dynamic behaviors is as close as possible to the real car. With this new DyModel, we go back to Pipeline C to train a third and last time our policy (Final Policy). This is the best policy we can obtain through our framework. Finally, we evaluate the Final Policy in a mixed-reality environment where a real car and a simulated car should cross each other without collision, and both reach their goal positions. The workflow defined by the above architecture and its pipelines is fully automated using the CI/CD integration, which is described in more details in Section 6.

Our contribution. We developed DeepRIoT, an efficient and low-cost CI/CD framework for training and deployment of RL policies in Robotic-IoTs. This narrows the sim2real gap in the RL community, and bridges the gap between robotics and CI/CD approaches in software engineering [7, 8]. The high success rate in our simulation and on-site experiments proves the feasibility of our approach.

2 RELATED WORK

Over the past years, the great potential of RL was increasingly exploited in more and more application domains, ranging from learning winning strategies in complex games such as AlphaGo [13], to continuous control of robotics tasks, as in [2–4]. However, RL generally requires millions of training episodes to be effective. Generating so many episodes in real-world Robotic-IoTs is practically infeasible. Furthermore, RL may be unsafe and costly, considering the potential damage and harm it may cause by applying a trial-and-error method directly on Robotic-IoTs.

Similarly to work in [3, 5, 14–17], we address this training problem by adopting a *Sim2Real* [4, 17] approach: We first train the RL algorithm with episodes generated in a simulation environment, a *digital twin* reproducing the physical behavior of the real system, and then we deploy the learned RL policy in the real-world scenario. However, as highlighted in [5], transferring a controller policy from a virtual to a physical environment is generally a very critical operation, because the simulated behavior is based on an approximated model of the physical system. Hence, undesired behaviors or failures may still appear only in the real Robotic-IoT. Thus, it is necessary to continuously update the digital twin(s) after observing the execution of a policy in the real Robotic-IoT.

In contrast to the works in [5, 14–16], here we propose a novel approach to reduce the gap between the real-world scenario and its digital twin, by employing a DevOps methodology based on CI/CD software development patterns [7, 8]. In particular, we show how to use digital-twin models of increasing complexity, to accelerate the convergence of RL on the digital twin, and how to use a mixed reality of virtual and actual Robotic-IoTs perceiving each other, to improve the learning and deployment of the RL policy in the real world. In contrast with other model-based DevOps proposed for cyber-physical systems [18–22], we design our approach for specifically accelerating the learning and deployment of deep reinforcement learning policies in Robotic-IoT systems.

3 PLANT MODEL

We assume that the *plant*, which in our example is the vehicle to be controlled by the policy we will synthesize, is given as a *Markov Decision Process (MDP)* $M = \langle \Sigma, A, P, R \rangle$. Here Σ is the state-space, A is the action space, P is a probabilistic transition relation, and R is a reward function. Given an initial state σ_0 and action sequence $a_{0:T-1}$, we denote the state sequence generated by M for T time steps as $\sigma_{0:T}$. Every state $\sigma' \in \Sigma$ has a scalar measure $R(\sigma)$, given by the reward function R. A trace is a pair $(a_{0:T-1}, \sigma_{0:T})$.

Vehicle Dynamics. This concertizes the transition relation of the MDP. As discussed in the introduction, we are using both a kinematic model [23], we call KiModel, and a dynamic bycicle model [23], we call DyModel, for this purpose. Both models are deterministic. However, their main sensor, the LiDAR scan, is probabilistic, as we add Gaussian noise. This captures the nature of observations received from the LiDAR mounted on the F1tenth car. As a consequence, the vehicle model itself becomes probabilistic. In order to design the reward, we start from a task specification.

Task Specification. Our goal is to train an RL policy that satisfies both reachability and safety properties. To this end, we first specify the *reachability* property in Signal Temporal Logic (STL) [10]:

$$\Phi_{\mathbf{R}} = G(s_0 \wedge \theta_0 \rightarrow F\ (|x - x_g| < \epsilon) \wedge (|y - y_g| < \epsilon))$$

It states that if we start the car in an arbitrary position $s_0 = (x_0, y_0)$ and with heading angle θ_0, then it should always (*G*lobally) be the case that, the car should eventually (*F*inally) reach the arbitrarily specified goal position $s_g = (x_g, y_g)$, within a given tolerance ϵ.

As *safety* property, we require that the car should always avoid collision with any fixed/movable obstacle: $\Phi_{\mathbf{S}} = G(d_{s,min} > d_0)$. To this end, we use laser scans d_s of the vehicle's LiDAR to measure the distance to obstacles in its vicinity, and require that the minimum distance $d_{s,min}$ is larger than a given safety threshold d_0,

Reward Design. To complete the MDP, we now proceed to concretize the reward function. To this end, we take advantage of the task specification, and structure the reward in two groups: reachability and safety, as in [11]. The first is concerned with the distance and progress (change) towards the goal, respectively.

Distance reward. The distance reward $r_d(t) = -(|\Delta x_t| + |\Delta y_t|)^{1/2}$ evaluates the distance to the goal from the car's current position. Its largest value equals to 0, which means the car reaches the goal.

Distance-change reward. We encourage the car to reach the goal as fast as possible by defining an approaching rate reward: $r_{dc}(t) = -\tanh[\Delta d_t \cdot \gamma_1] \cdot w_1$, where Δd_t is the change of Euclidean distance from the car to the goal position within two time steps, and γ_1 and w_1 decide the linear zone and bound of this reward.

Angle-change reward. Since we simulate a formula car that may get into an "oversteer" scenario due to undesired lateral control in the front wheel, we have to additionally design an angle reward to rid it out of such situation. Thus, we define h_t as the angle between the target and the wheelbase of the car. If the car is heading towards the target point between two time steps, i.e., $|h_t| - |h_{t-1}| <= 0$, we set a positive reward as: $r_{ac}(t) = (|h_{t-1}| - |h_t|)$. Otherwise, we set $r_{ac}(t) = -10$ to penalize the wrong steering behavior.

Reachability is effective only when the car is in no danger of collision. We created a pseudo bubble with $r = r_0$ around the center of the car. If the endpoint of any laser beams lies within this bubble, it means the car is in a "danger" state.

Collision-distance reward. We define $\bar{d}$ as the the average length of all short laser beams that fall into the aforementioned pseudo bubble, and use this variable as an indicator to provide a very negative reward as follows: $r_{cd}(t) = 400 - 40/(0.05 \cdot \bar{d}_t + 0.25)$.

Collision-rate reward. In order to help the car to recover from a dangerous situation quickly, we designed an auxiliary function that is positively correlated to its recovery rate: $r_{cr}(t) = \tanh(\Delta\bar{d}_t \cdot \gamma_2) \cdot w_2$, where $\Delta\bar{d}_t$ characterizes, between two time steps, the change of average laser lengths within the pseudo bubble.

4 POLICY TRAINING

Neural Network Design. We start by defining of the observation and action spaces and continue with the actor-critic architecture.

Observation space. The observation space can be divided into two groups. The *first group* includes five physical quantities observed from the car: $\mathbf{O_{1t}} = [\Delta x_t, \Delta y_t, \theta_t, v_{bx_t}, \omega_t]$, where Δx and Δy are the relative x and y positions from the target and the current position of the car; θ is the car's heading angle; v_{bx_t} is the car's longitudinal velocity, and ω_t is the angular velocity along z axis. At each time step, the car's lidar emits 1080 laser scans ($\mathbf{s_t}$) with a range of 270°. For the *second group*, we concatenate all the laser beams from the last three time steps, and arrange them into a matrix as $\mathbf{O_{2t}} = [\mathbf{s_t}, \mathbf{s_{t-1}}, \mathbf{s_{t-2}}]^{\mathrm{T}}$. Finally, the combination of the above two groups forms the complete observation space: $\mathbf{O_t} = [\mathbf{O_{1t}}, \mathbf{O_{2t}}]$.

Action space. The action space, denoted as $\mathbf{A_t}$, contains the desired steering angle δ_t^* and the desired longitudinal velocity $v_{bx_t}^*$.

We employ an actor-critic network architecture [24] for the policy architecture, using two neural networks, as follows.

Actor network. The inputs to the actor network are $\mathbf{O_{1t}}$ and $\mathbf{O_{2t}}$, and its main output is action $\mathbf{A_t}$. We first consecutively apply two 1D convolution layers (in_channels=1080/32, out_channels=32/32, kernel_size=5/3, stride=2/2, padding=1/1) to $\mathbf{O_{2t}}$, changing its size from $[1, 3, 1080]$ to $[1, 32, 270]$, and finally to $[1, 32 \times 270 = 8640]$ (flatten the last two dimensions). Next, we apply a linear neural network with parameter $(8640, 256)$ to further change the tensor to $\mathbf{O_{2t}}'$ with a shape of $[1, 256]$. Then we concatenate $\mathbf{O_{2t}}'$ and $\mathbf{O_{1t}}$ to finally form a fused tensor $\mathbf{O_t}$ with the size of $[1, 256 + 5 = 261]$. Finally, we applied a linear network to decrease the dimension of $\mathbf{O_t}$ from $[1, 256]$ to $[1, 2]$, where 2 is the dimension of actions. We used *tanh* activation to limit the output values into $(-1, 1)$ to facilitate scaling them to designated ranges. This way we obtained $\mathbf{A_t}$.

Critic network. We first use the same pre-processing method as described above to get a low-dimensional representation $\mathbf{O_{2t}}'$ for the laser scan. For the fusion procedure, this time we combined $\mathbf{O_{2t}}'$ with $\mathbf{O_{1t}}$ and $\mathbf{A_t}$, to obtain a compound tensor with a shape of $[1, 256 + 5 + 2 = 263]$. Finally, we applied two linear networks to obtain a single-number tensor, which evacuates how well the actions are given the input observation space.

Training Procedure Conventionally, the trainer stores all the experiences into one replay buffer, and uniformly samples batched experiences. However, since not all types of experiences are evenly distributed, this may lead to over-training or under-training for different tasks, or get training trapped into certain scenario with local minimum. We briefly discuss our remedy below.

Block-wise random initialization We divide the whole map into different blocks. The car and the target position can be initialized in each block with a given probability. In one block, we compute the possible range of initial positions and initial heading angles of the car, to keep it free from obstacles after initialization.

Symbolic abstraction We group the car's situation into three types according to its observation space at each time step. (1) T_1: The car is in no danger of collision, and it is steering towards the target; (2) T_2: The car is in no danger of collision, but it is steering far away from the target; (3) T_3: The car is in danger of collision. T_1 is encouraged while the rest two situations should be suppressed.

Dynamic allocation of replay buffer We build overall three replay buffers (*RB*). Here RB_i stores experiences from T_i $(i = 1, 2, 3)$. We can then dynamically change the sampling ratios for these replay buffers to satisfy different training preferences. We also adopt a "delayed" method by assigning a probability p for the trainer to resample an action if the situation belongs to T_2 and T_3.

Multi-stage training We split the training cycle into three stages, and for each stage, we set different a different goals: (1) S_1: We

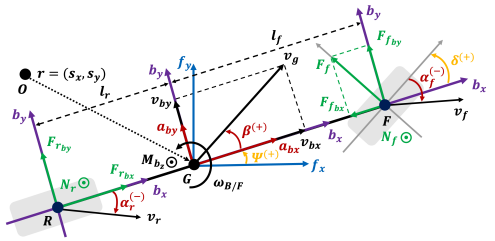


Figure 2: Bicycle Vehicle Model

encourage the car to reach the target; (2) S_2: We focus on collision avoidance; (3) S_3: We expect a quick convergence when the car approaches to the target.

Algorithm 1 Multi-stage Training

```
1:  S_lib = {S_1, S_2, S_3}
2:  for all s ∈ S_lib do
3:      Initialize network parameters, or load parameters from last stage
4:      while true do
5:          Set sampling ratios γ1, γ2, γ3 for RB1, RB2, RB3
6:          RB1 = {}, RB2 = {}, RB3 = {}
7:          for n_num = 1 to N do
8:              Initialize first state S1 = [O11, O21]
9:              for t = 1 to t_R do
10:                 At, St+1, r(t) = ACTIONSAMPLER(St)
11:                 Ti = Check type of state St+1
12:                 RBi = RBi ∪ {(St, At, r(t), St+1)}
13:                 if RB1, RB2, RB3 all get enough samples then
14:                     Sample M experiences from RB1-RB3 for training
15:         if Training results satisfied then
16:             Save weights for transfer learning in the next stage
17:             break
```

The multi-stage training approach is sketched in Algorithm 1. We sequentially train each stage and evaluate the results with accumulated rewards in training and success rate in test(lines 8-18). We first set the sampling ratios of each replay buffer and adjust them according to the periodical goals. Line 10 starts training from the first episode to the Nth episode. In the beginning of each episode, we randomly initialize the first state $\mathbf{S_1}$. Such randomness helps increasing the *reuse of experiences*, especially for those less encountered scenarios, e.g., the car is moving in the corners. Lines 12-18 implement the training process of each episode, where t_R is the maximum tolerable time for the car to reach the target. Line 13 executes state update. At time step t, we input current state $\mathbf{S_t}$, then the "delayed" action sampler will return the sampled action $\mathbf{A_t}$, the next state $\mathbf{S_{t+1}}$, and the corresponding reward value $r(t)$. Line 14 checks which type $\mathbf{S_{t+1}}$ falls into, then Line 15, assembles the current experience in form of a tuple as $(\mathbf{S_t}, \mathbf{A_t}, r(t), \mathbf{S_{t+1}})$, and adds this tuple to the corresponding replay buffer.

5 PLANT-MODELS TRAINING

This section discusses how we train the plant model of the car in Pipelines D/E to optimize its parameters.

Dynamic and Kinematics Models. Figure 2 shows the bicycle model used in automotive industry [23]. G is the gravity center, while R and F are centers of the rear and front wheels. The inertial frame F ($\boldsymbol{f}_{x/y/z}$) and the vehicle frame B ($\boldsymbol{b}_{x/y/z}$) are superimposed on G (Frame F has fixed origin O in the map, while Frame G has movable origin at G. The directions of $\boldsymbol{f_x}$ and $\boldsymbol{f_y}$ are fixed, while $\boldsymbol{b_x}$ always points to the nose of the wheelbase). The directions of the wheelbase and the rear wheel are the same, with a heading angle Ψ from the $\boldsymbol{f_x}$ axis. The car takes a left corner by firstly exerting a positive steering angle δ in the front wheel. However, tire velocities at R ($\boldsymbol{v}_r$) and F ($\boldsymbol{v}_f$) are not always pointing to the longitudinal axis of the tire. The deviated angles α_r, α_f determine the direction of the lateral friction forces $\boldsymbol{F}_f$ and $\boldsymbol{F}_{r_{by}}$. These two forces provide a positive moment $\boldsymbol{M}_{bz}$ that finally makes the car turn left, and deviate the velocity at G ($\boldsymbol{v}_g$) from the wheelbase by a slip angle β. We can build DyModel by collecting forces and moments in $\boldsymbol{b}_y$ and $\boldsymbol{b}_z$ such that $m\boldsymbol{a}_{by} = \boldsymbol{F}_{r_{by}} + \boldsymbol{F}_{f_{by}}$ and $I_z\ddot{\boldsymbol{\psi}} = -l_r\boldsymbol{F}_{r_{by}} + l_f\boldsymbol{F}_{f_{by}}$.

Compared to the DyModel, the KiModel assumes no slip angle β and considers no frictions. When we simulate a KiModel, the RL controller outputs desired steering angle and velocity to the model, which directly causes an update of v_g and δ. The heading angle is changed according to the car's geometric relation with δ such that $\dot{\Psi} = v_g tan\delta/(l_f + l_r)$. Finally, the G position of the car can be updated by $\dot{s}_x = v_g cos(\psi)$ and $\dot{s}_y = v_g sin(\psi)$.

Dynamic-Model Optimization. According to [23], the DyModel can be finally represented in the state-space form as $\dot{\boldsymbol{x}} = \boldsymbol{A_d x} + \boldsymbol{B_d u}$. The state vector is $\boldsymbol{x} = [s_x, s_y, \delta, v_g, \Psi, \dot{\psi}, \beta]$, the inputs are $\boldsymbol{u} = [f_\delta(\delta_t^*), f_v(v_{bx_t}^*)]$, where f_δ and f_v are the transfer functions of PID controllers. $\boldsymbol{A_d}$ and $\boldsymbol{B_d}$ are matrices with constant parameters. *Our goal in Pipelines D/E is to optimize the parameters in* $\boldsymbol{A_d}$ *and* $\boldsymbol{B_d}$ *with supervised learning, by using traces (labels) from the real car.*

In the real car, it is difficult to know f_δ and f_v, but the physical effect of f_v is equivalent to a_{bx}, which is measurable. Hence, we can still separate out the below equations independently from f_δ: $\ddot{\psi} = f_\psi(a_{bx}, \dot{\psi}, v_g, \beta, \delta; \boldsymbol{P})$ and $\dot{\beta} = f_\beta(a_{bx}, \dot{\psi}, v_g, \beta, \delta; \boldsymbol{P})$. $\boldsymbol{P}$ includes $\mu, m, I_z, l_f, l_r, C_{S,f}, C_{S,r}, h$, which are the parameters to be optimized.

For each training sample, the inputs-outputs are $[a_{bx}, \dot{\psi}, v_g, \beta, \delta]$ and $[\ddot{\psi}, \dot{\beta}]$ respectively. They are measured or estimated from the real car. We also provide the functional relations f_ψ and f_β. We initialize $\boldsymbol{P}$ with the default values in DyModel, then we can use the Mean-Square-Error (MSE) loss function to minimize the difference between model predictions and the labels by optimizing $\boldsymbol{P}$. We employ a Swarm-Optimization technique to train the DyModel.

Kinematic-Model Optimization. We write the KiModel in state-space form as: $\dot{\boldsymbol{x}} = \boldsymbol{A_k x} + \boldsymbol{B_k u}$, where $\boldsymbol{x} = [s_x, s_y, \delta, v_g, \Psi]$. Since KiModel is a geometric model, only l_f and l_r are the relevant parameters, so the only equation related two these two parameters is $\dot{\Psi} = v_g tan\delta/(l_f + l_r)$, which can be further simplified as $\dot{\psi} = g_\psi(v_g, \delta; l_f, l_r)$. Consequently, for each training sample, the inputs-output pairs are $[v_g, \delta]$ and $\dot{\psi}$. We can collect traces from the DyModel, build the labels, and finally optimize l_f and l_r to narrow the difference between the KiModel and DyModel.

State Measurement and Estimation Optimization of the KiModel does not need any state estimation, because the simulator provides all the data. However, training the DyModel requires the real-time values of $a_{bx}, \dot{\psi}, v_g, \beta, \delta$ from the car. The odometry directly provides $\dot{\psi}$ and v_{b_x}. We can get a_{bx} by taking the derivative of v_{bx} between

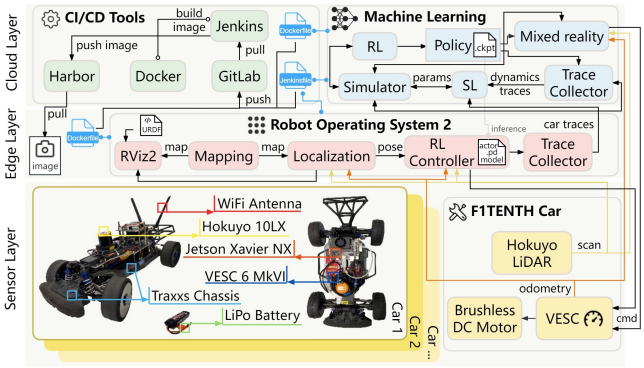
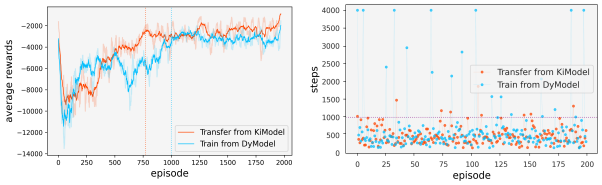


Figure 3: Hardware and Software Architecture



(a) Average rewards **(b) Steps**

Figure 4: Comparison between w/wo model training

two time steps. The localization module returns the G position (s_x, s_y). The velocity v_g can be estimated by computing the tangent value of each point along the car's motion curve.

We estimate β by considering the update of G's global x and y positions between two time steps: (1) $\Delta s_{x_k} = v_{g_k} cos(\Psi_k + \beta_k)\Delta t_k$; (2) $\Delta s_{y_k} = v_{g_k} sin(\Psi_k + \beta_k)\Delta t_k$. The left terms of the above two equations are the labels from localization, while the right terms can be viewed as the estimations. Since the localization module provides Ψ_k, and we have already computed v_{g_k}, we can iterate β_k from a reasonable range, and use the MSE function to pick the β_k value with the minimum loss value, then it is the best estimation.

As proposed in [25], δ can be approximated by $\delta = \arctan[(l_f + l_r)/(R_G cos\beta)]$. R_G is the radius of curvature at G such that $R_G = (s_x'^2 + s_y'^2)^{\frac{3}{2}}/|s_x' s_y'' - s_x'' s_y'|$, whose sign can be approximately determined by $\dot{\psi}$. Although δ is related to l_f and l_r, considering they will be optimized, this estimation can still be processed by the optimizer.

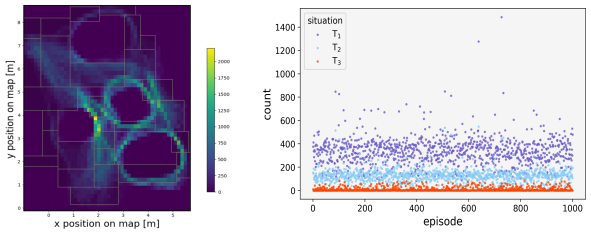
6 EXPERIMENTS

Hardware/Software Architecture. The HW/SW platform of our Robotic-IoT system is organized into the following three layers.

Sensor layer. The F1tenth car represents this layer. Its main components include: the JetsonXavier NX embedded module, the Hokuyo LiDAR used for perception, the odometry used to estimate the car posture, and the VESC controller used to drive the motor.

Edge layer. We consider the ROS2 middleware, running on the Jetson module, as the edge layer. It passes sensor data from the car to the RL policy, and policy actions back to the car. It is also responsible for building maps, processing localization, collecting traces from the car, and providing visualization through RViz2. Each program in ROS2 can be abstracted as a node, with the ROS2 launch file controlling the incorporation of each node.

Cloud layer. The cloud layer performs two major tasks: **(i) CI/CD**: The blue and red blocks in Fig. 3 represent the code that separately



(a) Trajectory heatmap **(b) Situation distribution**

Figure 5: Policy evaluation in simulation

(a) Heatmap **(b) Simulation** **(c) Deployment**

Figure 6: The deployment of trained policy

implements each module in Fig. 1. The scripts in the Jenkinsfile define the coordination between modules and the conditions that can automatically promote the development. All the code, including Jenkinsfile and Dockerfile are pushed to Gitlab, a code repository. At every Gitlab update, Jenkins is triggered to pull code from it, and execute the tasks defined in the Jenkinsfile. Next, Jenkins cooperates with Docker to build images that are stored in Harbor, the images repository. Finally, machines can pull images on demand and deploy them locally. **(ii) Mixed-reality**: (1) The virtual car and real car interact with each other through the *f1tenth_gym* [9] simulator; (2) The virtual car runs in the simulator and receives the posture from the real car, so it can "see" the real car through the LiDAR model. Then the virtual car updates its states through the RL controller and the DyModel inside the simulator; (3) The real car's LiDAR cannot physically perceive the virtual car, the fusion of the real laser scans and the LiDAR model will help the real car "see" the virtual car. (4) The real car updates its state by applying the action on the plant.

Experimental Evaluation. In the following two subsections, we discuss our simulation and sim2real experiments.

Simulation experiments. We achieved nearly 100% success rate in simulation. In Fig. 5a, the squares represent the blocks we used for initializing the car's starting and target positions. There are four obstacles located in the four circles, which are actual car trajectories. We use different color intensities to represent how frequently the car visits a position. Brighter colors are concentrated in the middle part of the map, which means these locations are trained more. In Fig. 5b, we visualize the ratios for different types of situations in 1000 tests. We find the first type ($T1$) overwhelmingly dominates the vast majority of cases, and the third type ($T3$) occurs least. This means that the car has learned how to approach the target location without getting into dangerous situations.

Fig. 4 shows the benefits of training plant model in simulation. In Fig. 4a (orange curve), we first trained a policy on the KiModel, then we collected traces from the DyModel to update its parameters, so this curve represents the training results when we did transfer

	Training	Test rounds	Success rate	Average steps
Simulation	from scratch	1000	98.1%	654.775
	transfer	1000	99.7%	488.555
	Deployment	**Test rounds**	**Success rate**	
Sim2Real	direct	60	68.33%	
	improved	60	71.67%	

Table 1: Experimental results for simulation and sim2real

learning on the updated KiModel. In Fig. 4a (blue curve), we directly trained a policy based on the DyModel. We find that from the 770th episode, the transfer-learning method (orange) begins to continuously achieve a high reward, while the direct-training method (blue) obtains the equivalent level of reward only starting from the 1000th episode. This yields a 23% of increase in efficiency.

Fig. 4b shows the time steps taken to finish the goal-reaching task. For the policy trained from the transfer-learning method (orange), it can finish most tests within 1000 time steps. However, for the direct-training method (blue), there are many cases where the policy takes long time to finish the task. As shown in the "Simulation" tab from Table 1, although the transfer-learning method has minor advantages on the success rate, the average steps to finish the task are much smaller than the direct-training method.

Sim2Real experiments. After deployment of the trained policy in the real car, the success rate drops to nearly 70%. We first analyzed the success rate in different locations. Fig. 6a shows that if we initialize the starting and target positions of the car in the four bright areas, the car will reach the highest success rate in the on-site experiments. We compare this result to Fig. 5a, and find that if the car wants to reach the four bright areas in Fig. 6a, it will mostly like traverse along the well-trained trajectories. This indicates that one reason for the drop of success rate is that the car is still not trained enough in other parts of the map, especially in the corners. We can anticipate that by sampling more positions in these areas would increase the success rate for the real car.

Despite a drop of success rate, we still achieve many successes in the mixed-reality application. Figure 6b shows the trajectories of a real car (red) and a virtual car (yellow). The real car and the virtual car started at s_1, s_2 respectively. At time t, they encountered, but they avoided each other and continued to reach their targets, as shown in g_1 and g_2. Figure 6c depicts the trajectory of the real car.

We finally evaluated the effects of Pipeline E in our approach. We first directly deployed the policy to the real car (*direct*). Next, we went through Pipeline E to improved the policy (*improve*). The "On-site" tab from Table 1 provides success rates for both methods a. We observed an increase of 3% in the success rate after improvement. The limited improvement can be attributed to the following reasons: (1) The car is still not well-trained in certain parts of the map, this may dilute the benefits of updating vehicle model; (2) Errors in state estimation due to noises. Nevertheless, DeepRIoT provides an automatic procedure to develop and debug Robotic-IoT systems.

7 CONCLUSION

We considered the automation design of RL-based Robotic-IoT systems by proposing DeepRIoT, a framework that accelerates both training and deployment. We narrow the sim2real gap by continuously updating the plant model via supervised learning and transfer learning, and automate the whole procedure through CI/CD. The high success rates and the low costs prove the effectiveness of our approach, and paves a way for further improvement in practice. Regarding future work, we plan to improve the effectiveness of DeepRIoT with considering more uncertainties and noises, and expand the generalizability by exploring more application scenarios.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments. This project has received funding from the Austrian FFG ICT of the Future program under grant agreement No 880811, and funding from the European Key Digital Technologies Joint Undertaking (KDT JU) under grant agreement No 101097300. This work was also partially supported by the WWTF project ICT22-023.

REFERENCES

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.
[2] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *Int. J. Robotics Res.*, 32(11):1238–1274, 2013.
[3] Saminda W. Abeyruwan et al. i-Sim2Real: Reinforcement learning of robotic policies in tight human-robot interaction loops. In *Proc. of CoRL 2022*, volume 205, pages 212–224. PMLR, 2022.
[4] Wenshuai Zhao et al. Sim-to-real transfer in deep reinforcement learning for robotics: a survey. In *Proc. of SSCI 2020*, pages 737–744, 2020.
[5] Andrea Stocco, Brian Pulfer, and Paolo Tonella. Mind the gap! A study on the transferability of virtual versus physical-world testing of autonomous driving systems. *IEEE Trans. Software Eng.*, 49(4):1928–1940, 2023.
[6] Christof Ebert et al. DevOps. *IEEE Softw.*, 33(3):94–100, 2016.
[7] Ade Miller. A hundred days of continuous integration. In *Agile 2008 Conference*, pages 289–293, 2008.
[8] Jez Humble and David G. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.
[9] Matthew O'Kelly et al. F1TENTH: an open-source evaluation environment for continuous control and reinforcement learning. In *NeurIPS 2019 Competition and Demonstration Track*, volume 123, pages 77–89. PMLR, 2019.
[10] Oded Maler and Dejan Nickovic. Monitoring properties of analog and mixed-signal circuits. *Int. J. Softw. Tools Technol. Transf.*, 15(3):247–268, 2013.
[11] Luigi Berducci and Radu Grosu. Safe policy improvement in constrained markov decision processes. In *Proc. of ISoLA'22*, volume 13701 of *LNCS*, pages 360–381. Springer, 2022.
[12] James Kennedy and Russell C. Eberhart. Particle swarm optimization. In *Proc. of the IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.
[13] David Silver et al. Mastering the game of Go without human knowledge. *Nat.*, 550(7676):354–359, 2017.
[14] Hao Ju et al. Transferring policy of deep reinforcement learning from simulation to reality for robotics. *Nat. Mac. Intell.*, 4(12):1077–1087, 2022.
[15] Fabio Muratore, Michael Gienger, and Jan Peters. Assessing transferability from simulation to reality for reinforcement learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(4):1172–1183, 2021.
[16] Jean Pierre Allamaa et al. Sim2real for autonomous vehicle control using executable digital twin. *IFAC-PapersOnLine*, 55(24):385–391, 2022.
[17] Sebastian Höfer et al. Sim2Real in robotics and automation: Applications and challenges. *IEEE Trans. on Autom. Science and Engineering*, 18(2):398–400, 2021.
[18] Benoît Combemale and Manuel Wimmer. Towards a model-based DevOps for cyber-physical systems. In *Proc. of DEVOPS 2019*, volume 12055 of *LNCS*, pages 84–94. Springer, 2019.
[19] Jerome Hugues et al. TwinOps - DevOps meets model-based engineering and digital twins for the engineering of cps. In *Proc. of MODELS '20*. ACM, 2020.
[20] Massimo Tisi et al. Towards twin-driven engineering: Overview of the state-of-the-art and research directions. In *APMS 2021*, volume 630 of *IFIP Advances in Information and Communication Technology*, pages 351–359. Springer, 2021.
[21] Aitor Gartziandia et al. Microservices for continuous deployment, monitoring and validation in cyber-physical systems: an industrial case study for elevators systems. In *Proc. of ICSA-C 2021*, pages 46–53, 2021.
[22] Hugo Bruneliere et al. AIDOaRt: Ai-augmented automation for DevOps, a model-based framework for continuous development in cyber-physical systems. *Microprocessors and Microsystems*, 94:104672, 2022.
[23] Matthias Althoff and Gerald Würsching. CommonRoad: Vehicle models, 2020.
[24] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Int. conf. on machine learning*, pages 1861–1870. PMLR, 2018.
[25] Jarrod M Snider et al. Automatic steering methods for autonomous automobile path tracking. *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RITR-09-08*, 2009.