# RL-Based Task Offloading in a K3s Cluster

This tutorial presents a step-by-step guide to set up and implement a task-offloading application based on reinforcement-learning (RL) in a k3s cluster consisting of one Edge node (Ubuntu 22.04) and one IoT node (Raspberry Pi OS 11).

## Install docker (both Edge and IoT nodes)

1. For Ubuntu, refer to https://docs.docker.com/engine/install/ubuntu/
2. For RPi, refer to <https://docs.docker.com/engine/install/debian/>

## Install Helm (Edge node only)

1. Refer to <https://helm.sh/docs/intro/install/>

## Install k3s

1. For the Edge node, install k3s as a master node,

curl -sfL https://get.k3s.io | INSTALL\_K3S\_VERSION=v1.23.14+k3s1 sh -s - server --cluster-init --kube-apiserver-arg="default-not-ready-toleration-seconds=20" --kube-apiserver-arg="default-unreachable-toleration-seconds=20" --write-kubeconfig-mode 644

A kubeconfig file will be written to /etc/rancher/k3s/k3s.yaml and the kubectl installed by K3s will automatically use it.

K3s does not require any special configuration to support Helm. Just be sure you properly set the kubeconfig path as per the guidelines in <https://docs.k3s.io/cluster-access>. Specifically, export the KUBECONFIG environment variable:

export KUBECONFIG=/etc/rancher/k3s/k3s.yaml

1. For the IoT node, install as agent node and add it to the cluster

curl -sfL https://get.k3s.io | K3S\_URL=https://myserver:6443 K3S\_TOKEN=mynodetoken sh -

Replace myserver with the hostname of the master node. The value to use for K3S\_TOKEN is stored at /var/lib/rancher/k3s/server/node-token on your master node.

\*if iot k3s failed to start at “cgroup\_memory=1 cgroup\_enable=memory” in /boot/cmdline.txt

## Install Prometheus (Edge node)

### Deploy Prometheus Operator to a Kubernetes Cluster using Helm

kube-prometheus-stack: a collection of Kubernetes manifests, Grafana dashboards, and Prometheus rules combined with documentation and scripts to provide easy to operate end-to-end Kubernetes cluster monitoring with Prometheus using the Prometheus Operator.

By default this chart installs additional, dependent charts:

* prometheus-community/kube-state-metrics
* prometheus-community/prometheus-node-exporter
* Grafana/grafana

The Prometheus Operator introduces custom resources in Kubernetes to declare the desired state of a Prometheus and Alertmanager cluster as well as the Prometheus configuration. For this guide, the resources of interest are:

* Prometheus
* ServiceMonitor
* PodMonitor

The Prometheus resource declaratively describes the desired state of a Prometheus deployment, while ServiceMonitor and PodMonitor resources describe the targets to be monitored by Prometheus.

#### Step 1: Installing the Prometheus Operator

Create a new namespace for Prometheus Operator using the following command:

$ kubectl create namespace monitoring

Add the Prometheus Operator Helm repository:

$ helm repo add prometheus-community <https://prometheus-community.github.io/helm-charts>

Update the Helm repositories:

$ helm repo update

Install the Prometheus Operator using Helm:

$ helm install prometheus-operator prometheus-community/kube-prometheus-stack -n monitoring --set node-exporter.nodeSelector."kubernetes\\.io/os"=linux --set node-exporter.nodeSelector."kubernetes\\.io/arch"=amd64

#### Step 2: Verifying the Deployment

After installing the Prometheus Operator, verify that the deployment is successful:

Check the status of the deployed Prometheus Operator components, ensuring that all of the pods are in “Running” state.

$ kubectl get pods -n monitoring

#### Step 3: Accessing Prometheus and Grafana Dashboards

To access the Prometheus and Grafana dashboards, follow these steps:

Check the service endpoints of Prometheus and Grafana. This will tell you where these endpoints are exposed to.

$ kubectl get svc -n monitoring

Use port forwarding to access the Prometheus dashboard locally:

$ kubectl port-forward svc/prometheus-operator-kube-p-prometheus -n monitoring 9090:9090

Access the Prometheus dashboard by opening <http://localhost:9090> in your browser.

Similarly, we can access the Grafana dashboard using port forwarding:

kubectl port-forward svc/prometheus-operator-grafana -n monitoring 3000:80

Access the Grafana dashboard by opening <http://localhost:3000> in your browser. Use the default credentials (admin/prom-operator) to log in.

\*Note (optional): To retrieve the password:

kubectl -n monitoring get secret prometheus-operator-grafana -o jsonpath='{.data.admin-password}' | base64 –decode

\*Note (optional): To reset to the default password (admin):

kubectl -n monitoring get secret prometheus-operator-grafana -o jsonpath='{.data.admin-password}' \

| base64 --decode \

| xargs -I {} kubectl -n monitoring exec -it <grafana pod name> -- grafana-cli --homepath /usr/share/grafana admin reset-admin-password {}

## Deploy MediaMTX’s RTSP server pod

The RTSP server is to stream the video from the camera installed on the IoT node.

Create the Kubernetes manifest deploy-rtsp.yaml using the codes below.

apiVersion: apps/v1

kind: Deployment

metadata:

name: mediamtx-deployment

spec:

replicas: 1

selector:

matchLabels:

app: mediamtx

template:

metadata:

labels:

app: mediamtx

spec:

nodeSelector:

kubernetes.io/hostname: iot2

containers:

- name: mediamtx-container

image: bluenviron/mediamtx:latest-rpi

env:

- name: MTX\_PROTOCOLS

value: "tcp"

- name: MTX\_PATHS\_UNICAST\_SOURCE

value: "rpiCamera"

- name: MTX\_PATHDEFAULTS\_RPICAMERAWIDTH

value: "640"

- name: MTX\_PATHDEFAULTS\_RPICAMERAHEIGHT

value: "480"

- name: MTX\_PATHDEFAULTS\_RPICAMERAFPS

value: "10"

securityContext:

privileged: true

volumeMounts:

- mountPath: /run/udev

name: udev

readOnly: true

- mountPath: /dev/shm

name: shm-volume

readOnly: false

ports:

- containerPort: 8554

name: rtsp

volumes:

- name: udev

hostPath:

path: /run/udev

- name: shm-volume

emptyDir:

medium: Memory

---

apiVersion: v1

kind: Service

metadata:

name: mediamtx-service

spec:

ports:

- name: rtsp

port: 8554

protocol: TCP

targetPort: 8554

nodePort: 30000

selector:

app: mediamtx

type: NodePort

The RTSP server service is exposed as a NodePort service. To deploy the server, use the command below:

kubectl apply -f deploy-rtsp.yaml

Use the following commands to check the status of the deployment and the running pods:

kubectl get deployments –o wide

kubectl get pods –o wide

## Deploy the Objection Detection App

in siong23/classify-detect:latest the detect.py ip is different so chg the ip by replace the detect.py by adding configuration in yaml file

# Copyright 2019 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

#     https://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

"""A demo that runs object detection on camera frames using OpenCV.

TEST\_DATA=../all\_models

Run face detection model:

python3 detect.py \

  --model ${TEST\_DATA}/mobilenet\_ssd\_v2\_face\_quant\_postprocess\_edgetpu.tflite

Run coco model:

python3 detect.py \

  --model ${TEST\_DATA}/mobilenet\_ssd\_v2\_coco\_quant\_postprocess\_edgetpu.tflite \

  --labels ${TEST\_DATA}/coco\_labels.txt

"""

import argparse

import cv2

import os

from pycoral.adapters.common import input\_size

from pycoral.adapters.detect import get\_objects

from pycoral.utils.dataset import read\_label\_file

from pycoral.utils.edgetpu import make\_interpreter

from pycoral.utils.edgetpu import run\_inference

def main():

    default\_model\_dir = '../all\_models'

    default\_model = 'mobilenet\_ssd\_v2\_coco\_quant\_postprocess\_edgetpu.tflite'

    default\_labels = 'coco\_labels.txt'

    parser = argparse.ArgumentParser()

    parser.add\_argument('--model', help='.tflite model path',

                        default=os.path.join(default\_model\_dir,default\_model))

    parser.add\_argument('--labels', help='label file path',

                        default=os.path.join(default\_model\_dir, default\_labels))

    parser.add\_argument('--top\_k', type=int, default=3,

                        help='number of categories with highest score to display')

    parser.add\_argument('--camera\_idx', type=int, help='Index of which video source to use. ', default = 0)

    parser.add\_argument('--threshold', type=float, default=0.1,

                        help='classifier score threshold')

    args = parser.parse\_args()

    print('Loading {} with {} labels.'.format(args.model, args.labels))

    interpreter = make\_interpreter(args.model)

    interpreter.allocate\_tensors()

    output\_details = interpreter.get\_output\_details()

    print('Model output details:', output\_details)

    labels = read\_label\_file(args.labels)

    inference\_size = input\_size(interpreter)

    #cap = cv2.VideoCapture(args.camera\_idx)

    #cap = cv2.VideoCapture('rtsp://192.168.0.150:30000/unicast')

    cap = cv2.VideoCapture('rtsp://admin:mmuzte123@192.168.254.2:554/cam/realmonitor?channel=1&subtype=1')

    while cap.isOpened():

        ret, frame = cap.read()

        if not ret:

            break

        cv2\_im = frame

        cv2\_im\_rgb = cv2.cvtColor(cv2\_im, cv2.COLOR\_BGR2RGB)

        cv2\_im\_rgb = cv2.resize(cv2\_im\_rgb, inference\_size)

        run\_inference(interpreter, cv2\_im\_rgb.tobytes())

        objs = get\_objects(interpreter, args.threshold)[:args.top\_k]

        cv2\_im = append\_objs\_to\_img(cv2\_im, inference\_size, objs, labels)

        cv2.imshow('frame', cv2\_im)

        if cv2.waitKey(1) & 0xFF == ord('q'):

            break

    cap.release()

    cv2.destroyAllWindows()

def append\_objs\_to\_img(cv2\_im, inference\_size, objs, labels):

    height, width, channels = cv2\_im.shape

    scale\_x, scale\_y = width / inference\_size[0], height / inference\_size[1]

    for obj in objs:

        bbox = obj.bbox.scale(scale\_x, scale\_y)

        x0, y0 = int(bbox.xmin), int(bbox.ymin)

        x1, y1 = int(bbox.xmax), int(bbox.ymax)

        percent = int(100 \* obj.score)

        label = '{}% {}'.format(percent, labels.get(obj.id, obj.id))

        cv2\_im = cv2.rectangle(cv2\_im, (x0, y0), (x1, y1), (0, 255, 0), 2)

        cv2\_im = cv2.putText(cv2\_im, label, (x0, y0+30),

                             cv2.FONT\_HERSHEY\_SIMPLEX, 1.0, (255, 0, 0), 2)

    return cv2\_im

if \_\_name\_\_ == '\_\_main\_\_':

    main()

Create the configmap:

kubectl create configmap myapp-config --from-file=detect.py=/home/pi/detect.py

Create the Kubernetes manifest myapp-deployment.yaml below:

apiVersion: apps/v1

kind: Deployment

metadata:

  name: myapp-deployment

  labels:

    app: myapp

spec:

  replicas: 1

  selector:

    matchLabels:

      app: myapp

  template:

    metadata:

      labels:

        app: myapp

    spec:

      nodeSelector:

        kubernetes.io/hostname: iot2

      containers:

      - env:

        - name: DISPLAY

          value: :0

        - name: LD\_LIBRARY\_PATH

          value: /opt/vc/lib:/lib:/usr/lib:/usr/local/lib

        name: myapp

        image: siong23/classify-detect:latest

        imagePullPolicy: IfNotPresent

        securityContext:

          privileged: true

        volumeMounts:

        - mountPath: /dev/vchiq

          name: dev-vchiq

        - mountPath: /dev/bus/usb

          name: dev-bus-usb

        - mountPath: /dev/vc

          name: dev-vc

        - mountPath: /dev/video0

          name: dev-video0

        - mountPath: /tmp/.X11-unix

          name: tmp-x11-unix

        - mountPath: /home/user/.Xauthority

          name: xauthority

        - mountPath: /coral/examples-camera/opencv/detect.py

          name: config-volume

          subPath: detect.py

#       command: ["tail","-f","/dev/null"]

#       command: ["/bin/bash"]

      volumes:

      - name: dev-vchiq

        hostPath:

          path: /dev/vchiq

      - name: dev-bus-usb

        hostPath:

          path: /dev/bus/usb

      - name: dev-vc

        hostPath:

          path: /dev/vc

      - name: dev-video0

        hostPath:

          path: /dev/video0

      - name: tmp-x11-unix

        hostPath:

          path: /tmp/.X11-unix

      - name: xauthority

        hostPath:

          path: /home/pi/.Xauthority

      - name: config-volume

        configMap:

          name: myapp-config

#      hostNetwork: true

#      dnsPolicy: ClusterFirstWithHostNet

Deploy the app using the command below:

kubectl apply -f myapp-deployment.yaml

Check the status of the deployment and the running pods:

kubectl get deployments –o wide

kubectl get pods –o wide

## Task Offloading Algorithm

A screenshot of a computer task

Description automatically generatedA white paper with black text

Description automatically generated

A white paper with black text

Description automatically generatedA white background with black text

Description automatically generated

A white paper with black text

Description automatically generated

A screenshot of a math problem

Description automatically generated

Only 1 iot device and 1 edge server

|  |  |
| --- | --- |
| Computation parameters | Value |
| Task size, *H* | 124,500 bits |
| Computation intensity, | 6,650 cycles per bit |
| IoT capacity, | 7.2 GHz(CPU cycles per second) |
| Edge capacity, | 13.2 GHz(CPU cycles per second) |
| Delay bound, *D* | 0.5 sec. |
| Transmission rate, *r\_dt​* | Min(1e7), Max(2e8) |
| Q-Learning parameters | Value |
| Learning rate | 0.1 |
| Discount factor | 0.9 |
| Episode | 200 |
| Penalty (E1, E2) | 2000, 10000 |

Beta = 0.9999

Base iot device cpu usage = 5 %

Base edge device cpu usage = 5 %

Cpu usage variance = 0.1

### Model Training Code

<https://colab.research.google.com/drive/1-m1oqN25Q_49DfXfGa_LmvUgG2KI6Hnq#scrollTo=BAd4xGUiaBZp>

import numpy as np

import matplotlib.pyplot as plt

from collections import defaultdict

import pickle

# Constants

H = 124500  # Task size in bits

C\_iot = 7.2e9  # IoT capacity in CPU cycles per second

C\_edge = 13.2e9  # Edge server capacity in CPU cycles per second

phi = 6650  # Computation intensity in CPU cycles per bit

E1, E2 = 2000, 10000  # Penalties

LEARNING\_RATE = 0.1

DISCOUNT\_FACTOR = 0.9

EPISODES = 200

NUM\_IOT\_DEVICES = 1  # Fixed to 1

BETA = 0.9999 # Weight for balancing C\_1t and C\_2t

# Constants for CPU usage

BASE\_CPU\_USAGE\_IOT = 0.05  # 5% base CPU usage for IoT device

BASE\_CPU\_USAGE\_EDGE = 0.05  # 5% base CPU usage for edge server

CPU\_USAGE\_VARIANCE = 0.1

TRANSMISSION\_RATE\_MIN = 1e7

TRANSMISSION\_RATE\_MAX = 2e8

T = 0.5  # Edge Time slot duration in seconds

D = 0.5  # Edge Delay bound in seconds

def discretize\_state(iot\_cpu, edge\_cpu, rate, prev\_action):

    # Discretize CPU usage into 10 levels (0.0-1.0)

    iot\_cpu\_level = min(int(iot\_cpu \* 10), 9)

    edge\_cpu\_level = min(int(edge\_cpu \* 10), 9)

    # Discretize transmission rates into 5 levels each

    rate\_level = int((rate - TRANSMISSION\_RATE\_MIN) /

                        (TRANSMISSION\_RATE\_MAX - TRANSMISSION\_RATE\_MIN) \* 5)

    # Clip to valid ranges

    rate\_level = max(0, min(4, rate\_level))

    # Combine all state components into a single tuple

    return (iot\_cpu\_level, edge\_cpu\_level, rate\_level, prev\_action)

class Environment:

    def \_\_init\_\_(self):

        self.reset()

    def reset(self):

        self.state = 0

        self.prev\_action = 0

        self.N\_dt = 1

        # Initialize transmission rates

        self.r\_dt = np.random.uniform(TRANSMISSION\_RATE\_MIN, TRANSMISSION\_RATE\_MAX)

        # Initialize CPU usage

        self.iot\_cpu\_usage = BASE\_CPU\_USAGE\_IOT

        self.edge\_cpu\_usage = BASE\_CPU\_USAGE\_EDGE

        # Return complete state tuple

        return discretize\_state(self.iot\_cpu\_usage, self.edge\_cpu\_usage, self.r\_dt, self.prev\_action)

    def update\_cpu\_usage(self, action):

        # Update base CPU usage with some randomness

        self.iot\_cpu\_usage = BASE\_CPU\_USAGE\_IOT + np.random.normal(0, CPU\_USAGE\_VARIANCE)

        self.edge\_cpu\_usage = BASE\_CPU\_USAGE\_EDGE + np.random.normal(0, CPU\_USAGE\_VARIANCE)

        # Calculate CPU usage increase based on task execution

        iot\_task\_load = (1 - action) \* (phi \* H) / (C\_iot \* T)

        edge\_task\_load = action \* (phi \* H) / (C\_edge \* T)

        # Increase CPU usage based on task load

        self.iot\_cpu\_usage += iot\_task\_load

        self.edge\_cpu\_usage += edge\_task\_load

        # Ensure CPU usage is between 0 and 1

        self.iot\_cpu\_usage = np.clip(self.iot\_cpu\_usage, 0.05, 0.99)

        self.edge\_cpu\_usage = np.clip(self.edge\_cpu\_usage, 0.05, 0.99)

    def step(self, action):

        self.update\_cpu\_usage(action)

        # Calculate available CPU capacity

        epsilon = 1e-6

        available\_iot\_capacity = max(C\_iot \* (1 - self.iot\_cpu\_usage), epsilon)

        available\_edge\_capacity = max(C\_edge \* (1 - self.edge\_cpu\_usage), epsilon)

        # Calculate loads

        L\_et = action \* (phi \* H) / (available\_edge\_capacity \* T)

        L\_dt = (1 - action) \* (phi \* H) / (available\_iot\_capacity \* T)

        # Calculate violations

        if action == 1:

            capacity\_violation = max(0, L\_et - available\_edge\_capacity \* T / (phi \* H))

        else:

            capacity\_violation = max(0, L\_dt - available\_iot\_capacity \* T / (phi \* H))

        latency\_violation = max(0, (H / self.r\_dt) - D)

        # Calculate costs

        C\_1t = min(np.abs(L\_dt - L\_et), 1e6)

        C\_2t = action \* self.prev\_action

        total\_cost = BETA \* C\_1t + (1 - BETA) \* C\_2t

        reward = -total\_cost - E1 \* capacity\_violation - E2 \* latency\_violation

        self.prev\_action = action

        # Update transmission rates

        self.r\_dt = np.random.uniform(TRANSMISSION\_RATE\_MIN, TRANSMISSION\_RATE\_MAX)

        # Return complete state tuple

        next\_state = discretize\_state(self.iot\_cpu\_usage, self.edge\_cpu\_usage, self.r\_dt, self.prev\_action)

        return (next\_state, reward, C\_1t, C\_2t, total\_cost, capacity\_violation, latency\_violation, self.iot\_cpu\_usage, self.edge\_cpu\_usage)

class QLearningAgent:

    def \_\_init\_\_(self):

        self.q\_table = defaultdict(lambda: np.zeros(2))

    def get\_action(self, state, epsilon):

        if np.random.random() < epsilon:

            return np.random.randint(0, 2)

        else:

            return np.argmax(self.q\_table[state])

    def update\_q\_table(self, state, action, reward, next\_state):

        current\_q = self.q\_table[state][action]

        next\_max\_q = np.max(self.q\_table[next\_state])

        new\_q = (1 - LEARNING\_RATE) \* current\_q + LEARNING\_RATE \* (reward + DISCOUNT\_FACTOR \* next\_max\_q)

        self.q\_table[state][action] = new\_q

    def export\_model(self, filename):

        with open(filename, 'wb') as f:

            pickle.dump(dict(self.q\_table), f)

    @classmethod

    def import\_model(cls, filename):

        with open(filename, 'rb') as f:

            q\_table = defaultdict(lambda: np.zeros(2), pickle.load(f))

        agent = cls()

        agent.q\_table = q\_table

        return agent

    def print\_q\_table(self):

        print("\nQ-Table (showing non-zero entries):")

        for state, actions in self.q\_table.items():

            if np.any(actions != 0):

                iot\_cpu, edge\_cpu, rate, prev\_action = state  # Extract prev\_action

                print(f"\nState:")

                print(f"  IoT CPU Level: {iot\_cpu}/9")

                print(f"  Edge CPU Level: {edge\_cpu}/9")

                print(f"  Rate Level: {rate}/4")

                print(f"  Previous Action: {'Offload' if prev\_action == 1 else 'Local'}")

                print(f"Actions: [Local: {actions[0]:.4f}, Offload: {actions[1]:.4f}]")

def run\_simulation():

    env = Environment()

    agent = QLearningAgent()

    rewards\_per\_episode = []

    cost\_1\_per\_episode = []

    cost\_2\_per\_episode = []

    total\_cost\_per\_episode = []

    iot\_cpu\_usage\_per\_episode = []

    edge\_cpu\_usage\_per\_episode = []

    offload\_ratio\_per\_episode = []

    for episode in range(EPISODES):

        state = env.reset()

        total\_reward = 0

        total\_cost\_1 = 0

        total\_cost\_2 = 0

        total\_cost = 0

        decisions = []

        penalties = []

        iot\_cpu\_usage = []

        edge\_cpu\_usage = []

        for step in range(100):  # 100 steps per episode

            action = agent.get\_action(state, epsilon=1.0 - episode / EPISODES)

            next\_state, reward, cost\_1, cost\_2, step\_total\_cost, capacity\_violation, latency\_violation, iot\_cpu, edge\_cpu = env.step(action)

            agent.update\_q\_table(state, action, reward, next\_state)

            total\_reward += reward

            total\_cost\_1 += cost\_1

            total\_cost\_2 += cost\_2

            total\_cost += step\_total\_cost

            decisions.append(action)

            penalties.append((capacity\_violation, latency\_violation))

            iot\_cpu\_usage.append(iot\_cpu)

            edge\_cpu\_usage.append(edge\_cpu)

            state = next\_state

        rewards\_per\_episode.append(total\_reward)

        cost\_1\_per\_episode.append(total\_cost\_1)

        cost\_2\_per\_episode.append(total\_cost\_2)

        total\_cost\_per\_episode.append(total\_cost)

        iot\_cpu\_usage\_per\_episode.append(np.mean(iot\_cpu\_usage))

        edge\_cpu\_usage\_per\_episode.append(np.mean(edge\_cpu\_usage))

        # Calculate offloading ratio for this episode

        offload\_ratio = sum(decisions) / len(decisions)

        offload\_ratio\_per\_episode.append(offload\_ratio)

        print(f"Episode {episode + 1}:")

        print(f"Total offloading decisions: {sum(decisions)}")

        print(f"Total capacity violations: {sum(p[0] for p in penalties)}")

        print(f"Total latency violations: {sum(p[1] for p in penalties)}")

        print(f"Total reward: {total\_reward}")

        print(f"Total cost 1: {total\_cost\_1}")

        print(f"Total cost 2: {total\_cost\_2}")

        print(f"Total cost: {total\_cost}")

        print(f"Average IoT CPU usage: {np.mean(iot\_cpu\_usage):.2f}")

        print(f"Average Edge CPU usage: {np.mean(edge\_cpu\_usage):.2f}")

        print(f"Offloading ratio: {offload\_ratio:.2f}")

        print("---")

    # Plotting

    fig, axs = plt.subplots(3, 2, figsize=(16, 18))

    fig.suptitle('Simulation Results per Episode')

    # Reward plot

    axs[0, 0].plot(rewards\_per\_episode, 'b-')

    axs[0, 0].set\_title('Reward')

    axs[0, 0].set\_xlabel('Episode')

    axs[0, 0].set\_ylabel('Value')

    axs[0, 0].grid(True)

    # Cost 1 plot

    axs[0, 1].plot(cost\_1\_per\_episode, 'r-')

    axs[0, 1].set\_title('Cost 1 (Load Imbalance)')

    axs[0, 1].set\_xlabel('Episode')

    axs[0, 1].set\_ylabel('Value')

    axs[0, 1].grid(True)

    # Cost 2 plot

    axs[1, 0].plot(cost\_2\_per\_episode, 'g-')

    axs[1, 0].set\_title('Cost 2 (Switching)')

    axs[1, 0].set\_xlabel('Episode')

    axs[1, 0].set\_ylabel('Value')

    axs[1, 0].grid(True)

    # Total Cost plot

    axs[1, 1].plot(total\_cost\_per\_episode, 'm-')

    axs[1, 1].set\_title('Total Cost')

    axs[1, 1].set\_xlabel('Episode')

    axs[1, 1].set\_ylabel('Value')

    axs[1, 1].grid(True)

    # CPU Usage plot

    axs[2, 0].plot(iot\_cpu\_usage\_per\_episode, 'b-', label='IoT')

    axs[2, 0].plot(edge\_cpu\_usage\_per\_episode, 'r-', label='Edge')

    axs[2, 0].set\_title('Average CPU Usage')

    axs[2, 0].set\_xlabel('Episode')

    axs[2, 0].set\_ylabel('CPU Usage')

    axs[2, 0].legend()

    axs[2, 0].grid(True)

    # Offloading Decision plot

    axs[2, 1].plot(offload\_ratio\_per\_episode, 'c-')

    axs[2, 1].set\_title('Offloading Ratio')

    axs[2, 1].set\_xlabel('Episode')

    axs[2, 1].set\_ylabel('Ratio')

    axs[2, 1].set\_ylim(0, 1)

    axs[2, 1].grid(True)

    plt.tight\_layout()

    plt.show()

    # Export the model

    agent.export\_model('q\_learning\_model.pkl')

    print("Model exported to q\_learning\_model.pkl")

if \_\_name\_\_ == "\_\_main\_\_":

    run\_simulation()

### K3s Model Application Code

import numpy as np

import pickle

from collections import defaultdict

from kubernetes import client, config

import requests

import urllib.parse

import time

import subprocess

import re

# Constants

H = 124500 # Task size in bits

C\_iot = 7.2e9 # IoT capacity in CPU cycles per second

C\_edge = 13.2e9 # Edge server capacity in CPU cycles per second

TRANSMISSION\_RATE\_MIN = 1e7

TRANSMISSION\_RATE\_MAX = 2e8

# IP Addresses

IOT\_IP = "192.168.0.150"

EDGE\_IP = "192.168.0.172"

# Prometheus configuration

PROMETHEUS\_URL = "http://localhost:9090/api/v1/query"

PROMETHEUS\_QUERIES = {

"iot\_device": f'''(1 - sum without (mode) (rate(node\_cpu\_seconds\_total{{job="node-exporter", mode=~"idle|iowait|steal", instance="{IOT\_IP}:9100", cluster=""}}[5m]))) / ignoring(cpu) group\_left count without (cpu, mode) (node\_cpu\_seconds\_total{{job="node-exporter", mode="idle", instance="{IOT\_IP}:9100", cluster=""}})''',

"edge\_device": f'''(1 - sum without (mode) (rate(node\_cpu\_seconds\_total{{job="node-exporter", mode=~"idle|iowait|steal", instance="{EDGE\_IP}:9100", cluster=""}}[5m]))) / ignoring(cpu) group\_left count without (cpu, mode) (node\_cpu\_seconds\_total{{job="node-exporter", mode="idle", instance="{EDGE\_IP}:9100", cluster=""}})'''

}

def query\_prometheus(query):

encoded\_query = urllib.parse.quote(query, safe="()[],")

full\_url = f"{PROMETHEUS\_URL}?query={encoded\_query}"

response = requests.get(full\_url)

data = response.json()

if 'data' in data and 'result' in data['data']:

return data['data']['result']

else:

return None

def get\_cpu\_usage():

iot\_cpu\_usage = query\_prometheus(PROMETHEUS\_QUERIES['iot\_device'])

edge\_cpu\_usage = query\_prometheus(PROMETHEUS\_QUERIES['edge\_device'])

iot\_total\_usage = sum(float(metric['value'][1]) for metric in iot\_cpu\_usage)

edge\_total\_usage = sum(float(metric['value'][1]) for metric in edge\_cpu\_usage)

return iot\_total\_usage, edge\_total\_usage

def ping(host, count=5):

"""Ping the specified host and return the average round-trip time after removing min and max."""

try:

command = ["ping", "-c", str(count), host]

output = subprocess.check\_output(command, universal\_newlines=True)

rtt\_values = re.findall(r'time=(\d+\.\d+) ms', output)

rtt\_values = [float(rtt) for rtt in rtt\_values]

if len(rtt\_values) > 2: # Ensure there are at least 3 values to remove min and max

rtt\_values.remove(min(rtt\_values)) # Remove the minimum RTT

rtt\_values.remove(max(rtt\_values)) # Remove the maximum RTT

avg\_rtt = sum(rtt\_values) / len(rtt\_values) if rtt\_values else None

return avg\_rtt

except subprocess.CalledProcessError:

print(f"Failed to ping {host}.")

return None

def calculate\_transmission\_rate(avg\_rtt, packet\_size\_bytes):

"""Calculate the transmission rate based on average RTT and packet size."""

if avg\_rtt is None:

return None

avg\_rtt\_seconds = avg\_rtt / 1000.0

packet\_size\_bits = packet\_size\_bytes \* 8

transmission\_rate = packet\_size\_bits / avg\_rtt\_seconds

return transmission\_rate

def get\_transmission\_rate(IP):

packet\_size = 124500 # size for ping packets in bytes

avg\_rtt = ping(IP) # Ping

if avg\_rtt is not None:

return calculate\_transmission\_rate(avg\_rtt, packet\_size)

return None

def swap\_deployment\_nodes(decision):

config.load\_kube\_config(config\_file='/etc/rancher/k3s/k3s.yaml')

api = client.AppsV1Api()

deployment\_name = "myapp-deployment"

namespace = "default"

try:

# Get the current deployment

deployment = api.read\_namespaced\_deployment(deployment\_name, namespace)

# Get the current node selector

current\_node\_selector = deployment.spec.template.spec.node\_selector

current\_node = current\_node\_selector.get("kubernetes.io/hostname")

# Determine the new node based on the decision

if decision == 0: # Deploy to IoT node

new\_node = "iot2"

else: # Deploy to Edge node

new\_node = "pi-edge2"

# If the decision is to keep the deployment on the same node, skip latency check and do nothing

if current\_node == new\_node:

print(f"No change needed. Current node is already '{current\_node}'.\n")

return

# Update the nodeSelector to swap the node

deployment.spec.template.spec.node\_selector = {"kubernetes.io/hostname": new\_node}

# Update the deployment with the new node

api.patch\_namespaced\_deployment(deployment\_name, namespace, deployment)

print(f"Deployment '{deployment\_name}' is being swapped to run on '{new\_node}'.")

# Monitor the deployment status until it is running on the new node

wait\_for\_deployment\_to\_run(api, deployment\_name, namespace)

except Exception as e:

print(f"An error occurred: {e}")

def wait\_for\_deployment\_to\_run(api, deployment\_name, namespace):

while True:

deployment\_status = api.read\_namespaced\_deployment\_status(deployment\_name, namespace)

available\_replicas = deployment\_status.status.available\_replicas

if available\_replicas and available\_replicas > 0:

print(f"Deployment '{deployment\_name}' is now running with {available\_replicas} replica(s) on the new node.\n")

break

else:

print(f"Waiting for deployment '{deployment\_name}' to be in Running state...\n")

time.sleep(10)

def discretize\_state(iot\_cpu, edge\_cpu, rate, prev\_action):

# Discretize CPU usage into 10 levels (0.0-1.0)

iot\_cpu\_level = min(int(iot\_cpu \* 10), 9)

edge\_cpu\_level = min(int(edge\_cpu \* 10), 9)

# Discretize transmission rate into 5 levels

rate\_level = int((rate - TRANSMISSION\_RATE\_MIN) /

(TRANSMISSION\_RATE\_MAX - TRANSMISSION\_RATE\_MIN) \* 5)

# Clip to valid range

rate\_level = max(0, min(4, rate\_level))

# Combine all state components into a single tuple, including the previous action

return (iot\_cpu\_level, edge\_cpu\_level, rate\_level, prev\_action)

def make\_offloading\_decision(iot\_usage, edge\_usage, rate, prev\_action, model\_path='q\_learning\_model.pkl'):

# Load the trained model

with open(model\_path, 'rb') as f:

q\_table = defaultdict(lambda: np.zeros(2), pickle.load(f))

# Discretize the state, including the previous action

state = discretize\_state(iot\_usage, edge\_usage, rate, prev\_action)

# Choose the action with the highest Q-value

action = np.argmax(q\_table[state])

return action

def main():

prev\_action = 0 # Initialize previous action (0 = Execute on IoT, 1 = Execute on Edge)

while True:

# Get the current CPU usage from IoT and Edge devices

iot\_usage, edge\_usage = get\_cpu\_usage()

# Get the current transmission rate

rate = get\_transmission\_rate(IOT\_IP)

print(f"Current state: IoT CPU Usage: {iot\_usage:.2f}, Edge CPU Usage: {edge\_usage:.2f}")

print(f"Transmission rate: {rate:.2e} bps")

# Make the offloading decision, passing the previous action

decision = make\_offloading\_decision(iot\_usage, edge\_usage, rate, prev\_action)

print(f"Decision: {'Execute on Edge' if decision == 1 else 'Execute on IoT'}")

# Swap the deployment nodes based on the decision

swap\_deployment\_nodes(decision)

# Update the previous action for the next iteration

prev\_action = decision

# Wait for a specified time before making the next decision

time.sleep(60) # Adjust the sleep time (in seconds) as per your requirement

if \_\_name\_\_ == "\_\_main\_\_":

main()

# Appendix

A K3s cluster can still be deployed on nodes which do not share a common private network and are not directly connected (e.g. nodes in different public clouds). K3s can integrate with Tailscale so that nodes use the Tailscale VPN service to build a mesh between nodes. Further details:

https://docs.k3s.io/networking/distributed-multicloud

<https://blog.csdn.net/easylife206/article/details/131950269>