Automated Concept Drift Handling for Fault Prediction in Edge Clouds using Reinforcement Learning

Behshid Shayesteh[†], Chunyan Fu[*], Amin Ebrahimzadeh[†], and Roch Glitho[†**], *Senior Member, IEEE*
[†]CIISE, Concordia University, Montréal, QC, Canada
[*]Ericsson Research, Montréal, QC, Canada
[**]Computer Science Programme, University of Western Cape, Capetown, South Africa

***Abstract*— Fault management systems that use real-time analytics based on Machine Learning (ML) help provide the reliability required in edge clouds, though they suffer from frequent changes in data distribution (concept drift) caused by highly dynamic traffic in edge clouds, requiring frequent adaptations of the ML model. We propose an automated concept drift handling framework for fault prediction in edge clouds using Reinforcement Learning (RL) to select the most appropriate drift adaptation method as well as the amount of data needed for adaptation, while considering edge cloud operator's requirements. We implemented an edge cloud testbed and introduced infrastructure and network faults to it in the presence of abrupt and incremental concept drift. According to the obtained results, our proposed framework achieves up to 40% higher accuracy compared to a system without drift handling, and up to 13× and 30× less regret for selecting adaptation methods and amount of data, respectively, compared to other approaches.**

***Index Terms*—Fault Prediction, Edge Cloud, Concept Drift, Machine Learning, Reinforcement Learning**

I. Introduction

EDGE cloud environments are required to be self-sustainably reliable, with the ability to recognize potential faults and failures and mitigate their effect, since as a fault occurs, it may propagate across nodes in the same cluster of the edge cloud, causing complexity in tracing the root cause of a fault, and extra costs and delays in fault recovery procedures [1]. To achieve this, while monitoring an edge cloud, a Machine Learning (ML) model can be trained to learn the correlation between the statistics of the monitored system and the occurrence of faults and failures in order to become capable of detecting and predicting the known faults [1]. However, ML-based techniques that use real-time statistics for fault prediction are constrained by several limitations [2]. One of these limitations is that although the statistics of edge clouds can have stable patterns, they can be affected by unexpected events, such as sudden or gradual changes in the workload within a certain time period [3], permanent or ephemeral anomalies [4], and/or highly dynamic traffic in edge clouds, causing frequent changes in the distribution of data that is used for training fault prediction models, a phenomenon commonly known as *concept drift* [5]. To ensure service reliability, it is important to prevent the performance degradation of the underlying prediction models caused by concept drift by means of effective model adaptation.

Traditionally, experts analyze the severity and type of the drift in data, choose the proper technology (e.g., partial updating, ensemble learning, or retraining) for drift adaptation [6], and determine the amount of data to be used for drift adaptation. However, adapting an ML model to the drift in a heterogeneous, large-scale edge cloud system is complex due to the diversity of the concept drift impacts, which requires different handling methods. On the other hand, the edge cloud operator wants to benefit from accurate predictions while minimizing the overall management overhead. This is well aligned with the so-called *zero-touch network service management*, which requires automated management of communication systems with no human intervention [2] [7]. Thus, concept drift must be handled in a more systematic way in edge cloud environments that automates the handling process while considering the performance requirements of the edge cloud operator. Given the resource-constrained characteristic of edge cloud systems and the time-sensitivity of use cases, the edge cloud operator's requirements for the concept drift adaptation could include (i) minimizing the time for adaptation so that the impact of the concept drift can be minimal, (ii) constraining adaptation resource utilization to the limited edge resources, (iii) maintaining an adapted model that has accuracy comparable to that of the original model. Given the diverse architectures of ML models deployed at edge sites, diverse requirements of edge cloud operator, and wide variety of drift types with different severity, it is challenging to automatically handle the drifts while considering all possible variations of ML models, requirements, and drifts.

Recently, in [8], we proposed an auto-adaptive fault prediction system for edge clouds in the presence of concept drift that utilized Convolutional Neural Networks (CNNs), Long-Short

(Corresponding author: Behshid Shayesteh)
Behshid Shayesteh and Amin Ebrahimzadeh are with the Concordia Institute of Information System Engineering (CIISE), Concordia University, Montréal, QC H3G 1M8, Canada e-mail: (b_shayes@encs.concordia.ca, amin.ebrahimzadeh@concordia.ca).
Chunyan Fu is with Ericsson Research, Montréal, QC H4S 0B6, Canada e-mail: (chunyan.fu@ericsson.com).
Roch Glitho is with the CIISE, Concordia University, Montréal, QC H3G 1M8, Canada and also with the Computer Science Program, University of Western Cape, Cape Town 7535, South Africa e-mail: (glitho@ece.concordia.ca).

Term Memory (LSTM), and their combination to predict CPU over-utilization and network congestion faults. Moreover, to handle the concept drift, we manually selected the amount of data for adaptation and evaluated a limited set of drift detection and adaptation methods, while only considering the model's accuracy after adaptation. In this work, we build on [8] and aim to solve the problem of automated selection of the amount of data for adaptation as well as the proper drift adaptation method among all available methods while considering the edge cloud operator's requirements. Furthermore, additional ML models that predict new types of faults, i.e., network packet loss and HDD over-utilization fault are trained in this work in addition to the faults studied in [8].

In this paper, we leverage on Reinforcement Learning (RL) [9] to propose our framework, which automatically selects the most appropriate drift adaptation method as well as the amount of data required for the adaptation process considering the operator's requirements. More specifically, we use Q-learning [10], a model-free RL algorithm. To select the proper drift adaptation method and data size, the gap between the method's performance and the operator's requirements should be minimized. In addition, both problems can be modeled as a sequential decision process, where the future state of the system only depends on the current state and decision, and thus, can be modeled as a Markov Decision Process (MDP). This, along with the fact that the variations of our problem are large but finite, and the success of RL in tackling similar problems that involve searching a design space to make an optimized decision [11]–[13], motivate us to utilize an RL-based approach to build an automated concept drift handling framework. To evaluate the proposed framework, we implemented a real-world edge cloud testbed using Kubernetes and injected four specific faults to it. We trained neural network ML models to predict these faults, and as new data becomes available, while ensuring that various types of concept drifts occur in this data, we train the RL agents to select the drift adaptation method and amount of data according to the edge cloud operator's requirements and evaluate our framework against the decisions of a human expert. The key contributions of this paper are as follows:

1) We propose a framework that automatically handles concept drift by selecting the drift adaptation method and data size for adaptation in heterogeneous edge cloud environments while considering the edge cloud operator's requirements of time, resource consumption, and accuracy.
2) We adopt a Q-Learning approach and design and train RL agents that are aware of the operator's requirements and the architecture of the fault prediction models that need to be adapted to a concept drift. Our RL agents select the adaptation method and amount of data for adaptation with the main objective of meeting the operator's requirements.
3) We run a comprehensive evaluation of our proposed framework on a real-world edge cloud testbed, where we injected and collected real fault data and evaluated our proposed framework upon arrival of the new data that is prone to concept drift. The obtained results indicate that our proposed framework can achieve up to a 40% higher accuracy compared to a system without drift handling, and has up to 13× and 30× less regret for selecting adaptation methods and amount of data, respectively, compared to other drift handling approaches.

The remainder of the paper is organized as follows. In Section II, we review the literature. The system model and the problem statement are explained in Section III. Our proposed RL-based automated concept drift handling framework is presented in Section IV. We illustrate the lab setup and the implementation results in Section V, and conclude the paper and identify future work in Section VI.

II. Related Work

In the following, we first review the prior research on fault prediction in edge cloud environments followed by the existing works on ML-based concept drift adaptation.

A. Fault Prediction in Edge Cloud

The literature on fault prediction for edge cloud environments is quite scarce. In [1], an infrastructure fault detection and prediction system for edge cloud environments was proposed. It utilized ML and deep learning algorithms for detecting and predicting faults such as CPU, HDD, memory over-utilization, and network congestion and packet loss faults using system performance metrics. The authors of [14] utilized a Hidden Markov Model to detect and predict faults by mapping container level observations (i.e., response time, resource utilization) to the underlying infrastructure problems in edge cluster environments. Ref. [14] was extended in [15], where a self-adaptive healing approach was proposed to recover from the faults that were predicted to occur and were detected. While the above mentioned works predict faults in edge cloud environments, they do not consider the problem of concept drift, which is a common challenge due to the dynamic nature of edge cloud systems and ever-changing patterns of data in edge clouds. In our previous work [8], we designed CPU over-utilization and network congestion fault prediction models using deep learning for time series forecasting and monitoring system performance metrics. Furthermore, we implemented and evaluated the models in a real-world testbed. We note, however, that in [8], human intervention was still required to handle concept drift in various edge cloud configurations and concept drift occurrence settings. Thus, the method presented in [8] is not scalable in heterogeneous large-scale edge clouds nor is it automated.

B. Concept Drift Adaptation

Concept drift adaptation is a relatively well-established research area in the literature of domains that work with evolving streaming data. However, here we specifically explore concept drift adaptation when the base learners are ML and deep learning models, and we categorize these papers by the underlying drift adaptation techniques, e.g., retraining, ensemble learning, and partial updating.

1) Retraining: This method trains a new model from scratch with the data after the drift, and then discards the obsolete model. The work in [16] presented a data selection mechanism for business process mining use case to compensate for concept drift by retraining ML models, including in particular, the Naive Bayes classifier. The authors of [17] proposed a new detection algorithm to handle concept drift while using Naive Bayes classifiers. While they use both incremental learning (partial updating) and retraining techniques for drift detection, they only use retraining to adapt the classifier to the drift. In [4], a drift-adaptive ML-based framework for IoT streaming data is proposed and evaluated on intrusion detection use case. It uses a Light Gradient Boosting Machine model, which is an ML model based on an ensemble of decision trees, for training a classifier. Furthermore, it performs drift detection and adaptation using their proposed method, which uses a window-based approach for drift detection, and uses retraining and hyper-parameter optimization for adaptation. The work in [18] studied the effect of concept drift on the performance of Automated ML (AutoML) methods. It has defined various drift adaptation strategies and evaluated them on open-source autoML libraries while various drift types occurred. These strategies consist of retraining the AutoML pipeline while imposing a variety of possible constraints on the AutoML's search space. In [19], a drift region-based data sample filtering method that removes obsolete data, i.e., the data before the drift, and uses the new concept data for retraining the base learner is proposed. This method is not limited to a specific classifier, but has been evaluated on k-Nearest Neighbors (kNN), Naive Bayes, decision trees, and Support Vector Machine (SVM) ML classifiers. Retraining is a time- and resource-intensive method since it requires large amount of data to train a model from scratch. However, it is the most straightforward method.

2) Ensemble Learning: The papers in this category, preserve a set of ML models (i.e., an ensemble) to handle concept drift. In [20], an ensemble learning method based on transfer learning [21] was proposed to tackle the problem of concept drift for ML models in general, and was particularly evaluated for decision trees. In this method, as new chunks of data arrive, it both adapts the historical preserved models to the new data, and retrain a new model with the new data and adds the latter to an ensemble set of preserved models, while keeping the ensemble diverse. Ensemble learning provides a better generalization ability while being a resource-consuming method, since multiple models are preserved for inference.

3) Partial Updating: The papers in this category, instead of discarding the obsolete ML model after the occurrence of the drift, partially update it to fit the new concept. In [22], the authors presented an anomaly detection and concept drift adaptation method for Recurrent Neural Networks (RNNs) by incrementally updating the prediction model without detecting a drift first. The authors of [23] proposed a concept drift adaptation method for CNNs using transfer learning to partially update the CNN model after a drift is detected. This method was proposed for image object detection use case, and it finds the obsolete layers of CNN, and then retrain those layers with the images after drift. The authors of [24] proposed an online adaptive RNN for electricity load forecasting use case, that adapts the deep learning model to the concept drift by re-tuning hyper-parameters (learning rate) and incrementally updating the model. In partial updating, it is challenging to know which part of the model (e.g., branches in decision trees or layers in a neural network) should be updated to get a well-performing adapted model.

While the reviewed papers aim at solving the concept drift problem for specific ML models, none has addressed the necessity of automating the process of selecting the proper drift adaptation method considering the time and resource consumption as well as post-adaptation performance of the underlying ML model.

III. System Model and Problem Statement

Fig. 1 illustrates a high-level view of an edge cloud environment consisting of one central cloud site and one edge site, as a representative of many. As shown in Fig. 1, we consider a deep learning time series forecasting model, which predicts infrastructure faults such as CPU or HDD over-utilization, network congestion, or network packet loss. We assume that an operator manages several types of edge cloud sites using these fault prediction models. The edge site has an Edge Site Monitor that collects raw training data, i.e., node-level performance metrics, and passes this data to the cloud for pre-processing that cleans the data from inconsistencies (e.g., outliers, null values, etc.), and labels them if necessary. The data is further used for feature selection to select the features or metrics that have the greatest correlation with the occurrence of a fault. The selected features of the pre-processed data are further used to train the fault prediction models in an offline mode, and then the trained models are deployed in the edge site for prediction. The Edge Site Monitor collects online data and feeds them to the trained fault prediction model to generate online fault predictions. Due to the high dynamicity in configurations and workloads of the edge cloud, the distribution of the arriving data is subject to frequent changes over time (i.e., concept drift), which causes false predictions and results in a degradation of the prediction model's performance. To tackle this problem, assuming that the true labels of the arriving data are available by using the timestamp-based labeling method defined in [1], the Drift Detector in the edge site receives the status of the fault prediction (i.e., whether the prediction was correct or incorrect) as an input, detects the occurrence time of a possible concept drift and generates a drift alert if a concept drift was detected. Next, upon receiving a drift alert, some new data after the drift is collected and the Drift Adaptor in the edge site adapts the fault prediction model to the concept drift using the newly collected data. The edge cloud's operator also specifies the concept drift handling requirement's Key Performance Indicators (KPIs), which can specify the time and resource consumption of adapting the prediction model, as well as the prediction model's inference accuracy after the concept drift adaptation.

To adapt the deployed deep learning time series forecasting model, we consider three approaches: retraining, partial updating, and ensemble learning. In retraining, the learned

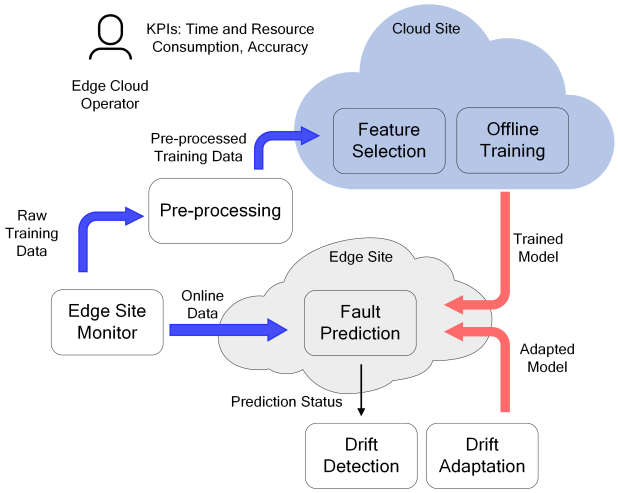


Fig. 1: Generic system model of an edge cloud environment.

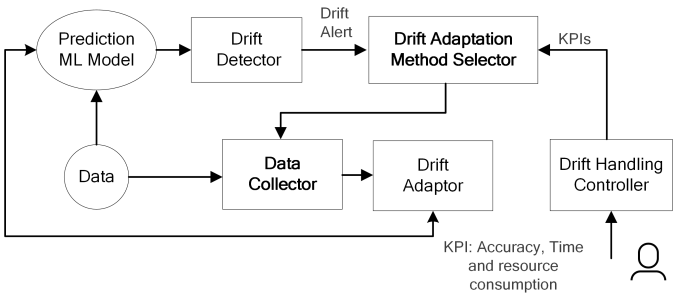


Fig. 2: Architecture of the proposed automated drift handling framework.

parameters of the neural network before the drift are discarded and a new model is trained from scratch using the data collected after the concept drift. In partially updating a neural network, the learned parameters of the old neural network are further fine-tuned as a starting point for the new model, which is also known as transfer learning, since some knowledge is transferred from a source model to a target model. In our previous work [8], we defined three transfer learning scenarios. However, there are more possible scenarios that specify which layers should be frozen (keeping the weights of the old model), and which should be fine-tuned, especially as the size of the models grow. In ensemble learning, the old and new neural network models are preserved in a set, and the final inference is a combination of the inferences of both models following a specific rule, e.g., weighted sum. The new preserved model in the ensemble could have been adapted using either retraining or the partial updating approach.

In the edge cloud setting described above, the operator sets various requirements for each edge site specifying that the drift adaptation process should not exceed $)_A$ seconds, consume less than $_A$ amount of resources, and achieve a minimum post-adaptation accuracy $_A$, given that a maximum amount $<OG$ of data is available for adaptation. We aim to solve the problem of automated selection of the appropriate drift adaptation method among all available methods (i.e., retraining, partial updating, ensemble learning) as well as the amount of data required for adaptation by minimizing the gap between the time and resource consumption, and accuracy of the selected adaptation method and desired $)_A$, $_A$, and $_A$ (i.e., the requirements of the edge cloud operator). Given the diverse architectures of the prediction models deployed at edge sites, the diverse requirements of the edge cloud operator, and various drift types, it is challenging to automatically handle the drifts while considering all the possible variations. Our proposed framework aims at solving this problem by providing an automated solution to handle the concept drift in heterogeneous edge cloud environments while considering the operator's requirements.

IV. RL-BASED AUTOMATED CONCEPT DRIFT HANDLING FRAMEWORK

In this section, we present our proposed automated drift handling framework. Fig. 2 illustrates the architecture of our proposed framework, in which it is assumed that a fault prediction model, specifically, a deep learning time series forecasting model, is previously trained using offline data. The fault Prediction Model receives pre-processed streaming data from the Edge Site Monitor component and generates its predictions on the fault status of the system. The Drift Detector component receives the status of the fault prediction as an input, detects the occurrence time of a possible concept drift and generates a drift alert if a concept drift is detected. Upon receiving a drift alert, the Drift Adaptation Method Selector component is initialized and reads the edge operator's required KPIs from the Drift Handling Controller component. The Drift Adaptation Method Selector uses the previously trained RL agent to infer the proper drift adaptation method considering the KPIs. The Drift Adaptation Method Selector initializes the Data Collector component that trains an RL agent to learn the amount of data to collect that will result in the highest model performance. Once the amount of data for collection has been decided, the Drift Adaptor component performs the drift adaptation process and updates the fault prediction model.

Algorithm 1 illustrates the different steps of our proposed automated concept drift handling framework. The algorithm to train RL agents for the Drift Adaptation Method Selector and the Data Collector are presented in Algorithm 2 and Algorithm 3, respectively. Algorithm 1 starts by loading a fault prediction model ("). When the new data (-) with a label (.) arrives, it is fed to the fault prediction model ("). The inference of the prediction model ($\hat{.}$) from data (-) along with the true label (.) of (-) is used to get the prediction status. While new data arrives, the model's prediction status is monitored to detect possible drifts (see lines 4-9). In case of drift detection, the set of operator's requirements ({) - - }), i.e., the adaptation's time and resource consumption ( ) and), and the model's accuracy ( ) after adaptation, along with the Drift Adaptation Method Selector's Q-table (& " ( ) that was previously trained using the Drift Adaptation Method Selector (Algorithm 2) are used to select the drift adaptation method (see lines 11-13). Next, a new Data Collector agent is trained using Algorithm 3, and the Q-table returned by the Data Collector (&) is further used to select the data

Algorithm 1 Automated Concept Drift Handling

1: **Initialize:**
2: Drift Detector, Edge Site Monitor (ESM),
3: Drift Handling Controller (DHC)
4: $\mathcal{M} \leftarrow$ Load trained fault prediction model
5: **while** data arrives **do**
6: $\mathbf{x} \leftarrow$ Read pre-processed data from ESM
7: $\hat{y} \leftarrow \mathcal{M}(\mathbf{x})$
8: prediction_status $\leftarrow$ get_prediction_status($\mathbf{x}, \hat{y}$)
9: drift_alert $\leftarrow$ drift_detector(prediction_status)
10: **if** drift_alert **then**
11: {KPIs} $\leftarrow$ read KPIs from DHC
12: $Q_{table} \leftarrow$ get Q-table returned by Algo. 2
13: adaptor $\leftarrow$ get_adapt_method (Q_{table}, {KPIs})
14: $Q_{table} \leftarrow$ train and get Q-table from Algo. 3
15: data_size $\leftarrow$ get_data_size (Q_{table})
16: new_model $\leftarrow$ perform_adapt(adaptor, data_size)
17: $\mathcal{M} \leftarrow$ new_model
18: **end if**
19: **end while**

size (see lines 14-15). Finally, the Drift Adaptor receives the adaptation method and the data size to perform the adaptation process and then substitutes the adapted model for the old prediction (see lines 16-17). In the following, we elaborate on the two main proposed components, i.e., the Drift Adaptation Method Selector and the Data Collector.

A. Q-learning

Two of the main components in our proposed framework (Drift Adaptation Method Selector and Data Collector) utilize Q-learning, and here we summarize theoretical formulations of Q-learning corresponding to our problems. Q-learning is an off-policy RL algorithm, which means it can learn the optimal policy by behaving under a non-optimal policy, e.g., greedy. Moreover, it is a model-free algorithm, which means it does not need to explicitly understand the dynamics of the environment to learn how to act optimally in it. Q-learning fits our problems because in both Drift Adaptation Method Selector and Data Collector, we only need to learn a function that optimizes the agent's behavior and leads the agent on how to act in a given situation without modeling the environment.

In a general RL problem, an agent continually interacts with an environment, and the environment accordingly presents new situations to the agent, along with rewards, which the agent tries to maximize over time [9]. Given that the agent and the environment interact in discrete time steps t, the agent is presented with a representation of the environment, i.e., state $s_t \in \mathcal{S}$ at time t, where $\mathcal{S}$ is the set of all possible states in the environment. Based on this state, the agent takes the action $a_t \in \mathcal{A}(s_t)$, where $\mathcal{A}(s_t)$ is the set of possible actions in state s_t. Consequently, the agent receives a numerical reward $r_{t+1} \in \mathcal{R} \subset \mathbb{R}$ in the next time step and ends up in state s_{t+1}. The agent's goal at each time step t is to maximize the reward it receives over the long run, i.e., the amount of return G_t, given by:

$$G_t = \sum_{k=0}^{T-1} \gamma^k r_{t+k+1} \quad (1)$$

where T is the final time step and γ is a parameter called the discount factor, where $0 \leq \gamma \leq 1$, which is the weight given to the immediate rewards over the long-term rewards. The expected return starting in state s taking an action a is called the Q-value denoted as $Q(s,a)$ and defined as follows:

$$Q(s,a) = \mathbb{E}[G_t \mid s_t = s, a_t = a] \quad (2)$$

where Q is called the action-value function. The Q-learning algorithm starts by initializing the Q-value of all possible state and action pairs to a pre-defined value, e.g., zero. Next, it iteratively updates the Q-values for each state and action pair using the Bellman's equation given by:

$$Q(s_t,a_t) = (1-\alpha)Q(s_t,a_t) + \alpha[r_t + \gamma \max_{a'} Q(s_{t+1},a')] \quad (3)$$

where α is the learning rate parameter that indicates the importance of the recent information over the older information. If the state s and action a pair is sampled infinitely, $Q(s,a)$ will converge to the maximum expected reward $Q^*(s,a)$ [25]. Given that the number of state and action pairs are finite, the Q-values are preserved in a table called the Q-table. After the action-value function converges, the agent uses the Q-table to decide what action to take in each time step.

In our algorithm design, the agent follows the ϵ-greedy policy [26] while training. In this policy, the agent acts randomly with the probability of ϵ, and acts greedy with the probability of $1-\epsilon$. The ϵ value starts from 1 and is gradually decayed to a near-zero value. By reducing the ϵ value gradually, the agent acts randomly at the beginning and explores many state and action pairs, gradually exploiting the information it has learned. Using ϵ-greedy policy can improve convergence as it suggests an approach that avoids the issues of over-exploration and over-exploitation, which typically results in slow convergence and convergence to local optimum, respectively. Moreover, in our algorithm, the agent uses the experience replay technique [27], where the explored paths and their corresponding rewards are stored. The agent periodically samples from experience replay to update the Q-values. This, as a result, speeds up the convergence of the action-value function and increases data efficiency as each sample is used multiple times to update the Q-table, which is particularly beneficial when the exploration is costly or the state space is large [26].

B. Drift Adaptation Method Selector

This component is responsible for selecting the proper drift adaptation method considering the KPIs from the Drift Handling Controller. The Drift Adaptation Method Selector considers all the possible methods to adapt a neural network (i.e., retraining, transfer learning, and ensemble learning) and then makes a selection that meets the operator's requirements.

Fig. 3 illustrates two main steps to train an RL agent to select the proper drift adaptation method using Q-learning. In the first step, inspired by [11], the agent samples a drift adaptation

method by deciding which drift adaptation approach to use for each layer of the neural network model, given a pre-defined behavior pattern. Next, the sampled drift adaptation method is applied to the data after the concept drift and the performance metric, i.e., the accuracy of a validation set, as well as time and resource consumption of drift adaptation is saved in the agent's replay memory along with the adaptation method. Furthermore, the agent samples from the replay memory and learns by updating the Q-table. The following sub-sections elaborate on the state and action spaces and reward function of the Drift Adaptation Method Selector.

1) State Space: In this problem, each state is defined in a three-tuple.

(i) The neural network layer depth that represents the position of a layer in a neural network. The agent needs to know the layer depth, since some actions are only possible in specific layer positions.
(ii) The drift adaptation applied to that layer, which can be either Retraining (R), Fine-tuning (F), Freezing (Z), Preserving (P), or Discarding (D). The first three adaptations indicate that the parameters in the corresponding layer were retrained from scratch, fine-tuned from the model before the adaptation, or were frozen, respectively. The last two adaptations indicate that the old model is preserved as an ensemble with the adapted model, or is discarded, respectively.
(iii) The operator's adaptation requirements (KPIs), a set that consists of the drift adaptation method's time consumption $)$, resource consumption , and the model's performance after the adaptation, i.e., its accuracy on a validation set. The three requirements can co-exist at the same time. It is assumed that the operator can define a finite number of requirement sets $\{)$- - $\}$ to keep the state space of the problem finite.

The representation of a state that performs an adaptation approach G on the 8^{C} layer of a neural network, while the operator's requirement equals $\{)$- - $\}$, is as follows:

$$B = ((8\text{-}G\text{-}\{)\text{-} \quad \text{-} \quad \})\bullet \tag{4}$$

2) Action Space: In each state B, the agent can take action $0 \in \mathcal{A}(B)$. There are five actions defined in this problem: Retraining (R), Fine-tuning (F), Freezing (Z), Preserving (P), or Discarding (D). However, not all actions are available in all states. In the initial state $(_0$, the agent decides which drift adaptation approach to choose for the first layer, where the action is selected from $\mathcal{A}((_0) = \{\mathrm{R, F, Z}\}$. Moreover, if the current state's adaptation approach is $'$, the agent can only select $'$ as its future action, since in the retraining adaptation method, all layers are retrained from scratch. Similarly, if the current state's adaptation approach is or $/$, the next action is selected from $\mathcal{A}(() = \{\mathrm{F, Z}\}$, and any combination of layers can be fine-tuned or frozen. Finally, after the agent decides what action to select for the last layer, it decides to form an ensemble of this model and the old model, or to discard the old model. Hence, in the last layer, $\mathcal{A}(() = \{\mathrm{P, D}\}$. Available action(s) in each state B is given by:

$$\mathcal{A}(B) = \begin{cases} \{\mathrm{R, F, Z}\}\text{-} & B = (_0 \\ \{\mathrm{R}\}\text{-} & B = ((\{1\text{-}\bullet\bullet\bullet\text{-}\#-1\}\text{-}\ '\text{-}\cdot) \\ \{\mathrm{F, Z}\}\text{-} & B = ((\{1\text{-}\bullet\bullet\bullet\text{-}\#-1\}\text{-}\{/\cup\ \ \}\text{-}\cdot) \\ \{\mathrm{P, D}\}\text{-} & B = ((\#\text{-}\cdot\text{-}\cdot)\text{-} \end{cases} \tag{5}$$

where $\#$ is the total number of layers and $\{1\text{-}\bullet\bullet\bullet\text{-}\#-1\}$ is the set of all the layers, except the last one.

3) Reward Function: The agent receives a reward after performing the drift adaptation method it selected. We define the reward of action 0_8 at a given state B_8 as a function of the drift adaptation's time consumption $)_8$, resource consumption $_8$, and accuracy of the model after adaptation $_8$, as follows:

$$\mathcal{R}(B\text{-}0) = :_1\frac{(\ _8 - \ _A)}{\ _A} - :_2\frac{()_8 -)_A)}{)_A} - :_3\frac{(\ _8 - \ _A)}{\ _A}\text{-} \tag{6}$$

where $_A$ is the operator's required accuracy, and $)_A$ and $_A$ are the maximum time and resources available for adaptation as specified in operator's requirements. The $:_1$, $:_2$, and $:_3$ coefficients reflect how exceeding each requirement (obtaining the target accuracy and remaining under the time and resource consumption limits) can penalize or promote the reward, respectively. According to Eq. (6), the Drift Adaptation Method Selector considers all three requirements at the same time in the calculation of the reward, and the actions that meet all requirements are assigned the highest rewards. Similarly, if all or some of the requirements cannot be met, since performing drift adaptation is necessary, the action that has the closest $)$, , and values to those of the requirements will be assigned the highest reward, and consequently will be selected by the Drift Adaptation Method Selector.

It is assumed that this RL agent is trained offline, and the Drift Adaptation Method Selector component infers the proper action from the Q-table. The RL agent can be further trained if the operator needs to find the proper drift adaptation method for a new requirement set not known by the agent. Algorithm 2 illustrates the steps of our proposed Drift Adaptation Method Selector.

C. Data Collector

Once the proper drift adaptation method to fulfill the operator's requirements is selected by the Drift Adaptation Method Selector, the Data Collector trains a new Q-learning RL agent in a different environment (see lines 5 - 28) to learn the amount of data to collect to start the drift adaptation process.

1) State Space: In this problem, the size of the data to be collected is represented as a state. The possible data sizes are defined within discrete intervals .

2) Action Space: In this problem, in each state, the agent can take two actions to either decrease or increase the current data size with an amount of . Assuming that the initial state is to collect an amount of 3_0 data ($B_{\ -0} = 3_0$), and the agent increases the data size, the next state would be $B_{\ -1} = 3_0 + $. It is further assumed that the operator specifies the maximum data size available for data collection $_A$ as the upper limit.

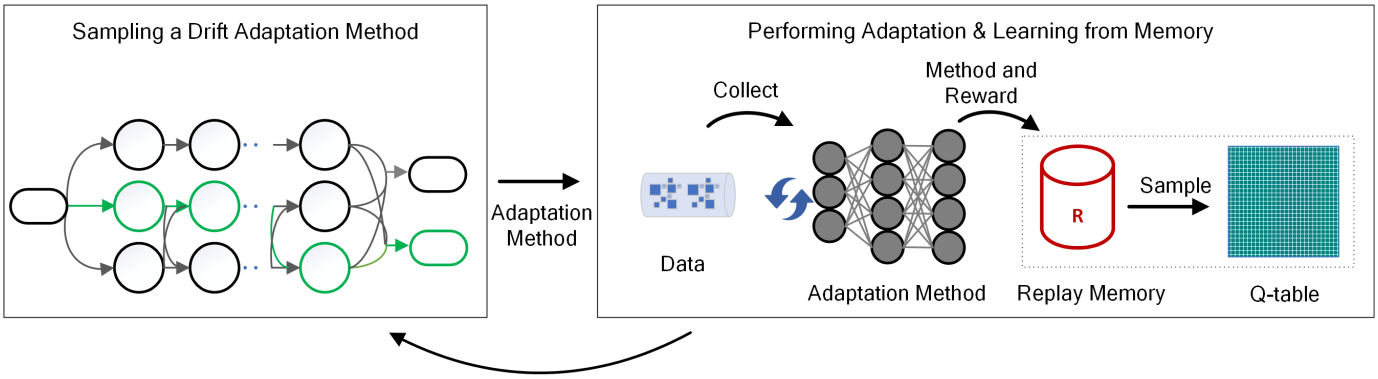


Fig. 3: Drift Adaptation Method Selector Steps.

Algorithm 2 Drift Adaptation Method Selector

Inputs: Number of model's layers (#), Number of explorations for each n ( ), Total number of episodes ( )
Initialize:
replay_memory ← []
& ← 0 for all state-action pairs
for (episode = 1 to) **do**
  Visited states (= [($_0$]
  Taken actions $\mathcal{A}$ = []
  for (layer = 1 to #) **do**
    generate random number $4 \in [0\text{-}1]$
    $\mathcal{A}$(([−1]) ← get possible actions for ([−1] from Eq. (5)
    if 4 ¡ n **then**
      $0 \leftarrow \max_{0 \in \mathcal{A}(([-1])}$ &(([−1]-0)
    else
      0 ← random selection in $\mathcal{A}$(([−1])
    end if
    $\mathcal{A}$.append(0)
    ($_{=4F}$ ← perform action 0
    (.append(($_{=4F}$)
  end for
  $\mathcal{R}$ ← perform_adaptation(() get reward from Eq. (6)
  memory.append((-$\mathcal{A}$-$\mathcal{R}$)
  {($_{<}$-$\mathcal{A}_{<}$-$\mathcal{R}_{<}$} ← " random selections from memory
  for ((-$\mathcal{A}$-$\mathcal{R}$ in ($_{<}$-$\mathcal{A}_{<}$-$\mathcal{R}_{<}$) **do**
    for (B-0 in (-$\mathcal{A}$) **do**
      update &(B-0) using Eq. (3) & reward '
    end for
  end for
  ← − 1
  if equals 0 **then**
    decay n & re-initialize E
  end if
end for
return &

3) Reward Function: The reward function for Data Collector ($\mathcal{R}$) is a function of the gained accuracy using the collected data $_2$ and the operator's required accuracy $_A$, as follows:

$$\mathcal{R} \quad (B \quad -0 \quad) = \quad _2 - \quad _A\text{-} \tag{7}$$

which indicates that the data sizes resulting in closer accuracies to the operator's required accuracy get higher rewards. Algorithm 3 illustrates the steps of our proposed Data Collector.

D. Complexity Analysis

Here we present a complexity analysis of our proposed automated concept drift handling, Drift Adaptation Method Selector, and Data Collector algorithms. The time complexity of an n-greedy Q-learning algorithm is from the order of the number of steps in an episode and the length of each episode [28]. Similarly, the space complexity is from the order of the episode length and the number of states and actions [28]. Therefore, the time complexity of the Data Collector (Algorithm 3) is $O($ $\cdot$ $!$), and the space complexity is $O(!$ $\cdot|\mathcal{A}$ $|\cdot($), where (= ($_A$ − $_0$)/ is the number of states. For the Drift Adaptation Method Selector (Algorithm 2), since a replay buffer of size " is used, the time complexity is $O($ $\cdot\#\cdot$ "). For the space complexity, since each state is a three-tuple, assuming that there are requirement sets, the number of states is $|\mathcal{A}|\cdot\#\cdot$, and the space complexity would be $O(|\mathcal{A}|^2\cdot\#^2\cdot$ $\cdot$ "). As for the complexity of automated concept drift handling (Algorithm 1), assuming that the size of the arriving data is , , the time complexity is $O($, $\cdot)$ $\cdot$ $!$) and the space complexity is $O($, $\cdot$ $!$ $\cdot\mathcal{A}$ $\cdot($).

V. Results

In this section, we evaluate the performance of our proposed framework in a real-world testbed. Similar to [29]–[31], we implemented a Kubernetes[1]-based edge cloud testbed to evaluate our proposed automated concept drift handling framework. We have used Kubernetes to realize an edge cloud environment since our goal is to expand this Proof-of-Concept (PoC) to a fully containerized solution so that it can be easily deployed and scaled. In the following, after presenting the implementation settings, i.e., the lab setup and dataset

[1]Kubernetes is a widely used open-source tool for container orchestration that can easily manage scaling, failovers, and load balancing, among others.

Algorithm 3 Data Collector

Inputs: Minimum and Maximum data available ($_{0}$- $_{A}$), Total number of episodes ( ), Episode length ($!$), Number of exploration for each n ( ), Interval between two consecutive data sizes
Initialize:
$\mathcal{A}$ $\leftarrow$ {increase, decrease}
$\&$ $\leftarrow 0$ for all state-action pairs
for (episode = 1 to) **do**
  B $\leftarrow$ start in a random initial state
  is_episode_done $\leftarrow$ False
  while ! is_episode_done **do**
    generate random number $4 \in [0\text{-}1]$
    if 4 ¡ n **then**
      $0 \leftarrow \max_{0 \in \mathcal{A}_{DC}} \&$ (B -0)
    else
      $0 \leftarrow$ random selection from $\mathcal{A}$
    end if
    $\mathcal{R}$ $\leftarrow$ do action 0 and get reward from Eq. (7)
    update $\&$ (B -0) using Eq. (3) & reward $\mathcal{R}$
    ($\leftarrow$ perform action 0
    $! \leftarrow ! - 1$
    if $!$ equals 0 or $\mathcal{R}$ ≥ 0 or B $<$ $_{0}$/ or B $>$ ($_{A}$ − $_{0}$)/ **then**
      is_episode_done $\leftarrow$ True
      re-initialize L
    end if
  end while
    = − 1
  if equals 0 **then**
    decay n & re-initialize E
  end if
end for
return $\&$

and coding tools, we present the experimental results of our evaluations.

A. Implementation Settings

We describe our lab setup, datasets, and coding tools in detail in the following two sub-sections.

1) Lab Setup: Fig. 4 depicts our lab setup, which consists of three Kubernetes clusters with a total of 10 Virtual Machines (VMs) running *Ubuntu 18.04*. The Kubernetes clusters are deployed at Ericsson Research's private cloud Xerces, which consists of around 300 servers, being managed by the Infrastructure-as-a-Service (IaaS) OpenStack. One cluster represents the central site while the other two clusters are the edge sites. One of the VMs in the central site cluster acts as the master node and three VMs are the worker nodes in this cluster, and a physical machine is connected to this site for training the fault prediction models. Similarly, each of the two edge site clusters has one VM as its master node and two VMs acting as worker nodes. The configurations of the VMs are summarized in Table I. To cause drifts in the data (i.e., the performance metrics of the nodes) used for training the

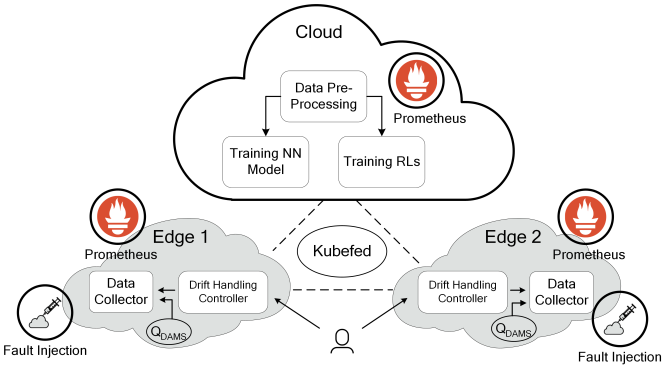


Fig. 4: Lab setup for evaluating the proposed automated concept drift handling framework.

TABLE I: Lab setup parameters and default values.

Site	Parameter	Value
Central Site	Number of VMs	4 VMs
	CPU	4 cores for each VM 1 core i7-8700 for training
	RAM	16G for each VM 16G for training
	HDD	100G for each VM 1T for training
Edge Sites	Number of VMs	3 VMs
	CPU	4 cores for each VM
	RAM	8G for each worker VM 16G for master VM
	HDD	100G for each VM

fault prediction models, we installed Stan's Robot Shop [32], a CPU-intensive application, on all the master nodes. The load generator of Stan's Robot Shop increases the CPU, memory and network load causing drifts in the nodes' performance metrics. The architecture illustrated in Fig. 4 is for a small-scale setup. At larger scales, based on the edge nodes' capacity, it would be possible to move the data pre-processing component, or the set of all components, into the edge nodes to avoid big data transfers over the network. We utilized *Prometheus* [33] to monitor the VMs in Kubernetes clusters and collect the data. For each cluster, we have one instance of Prometheus. We used Kubernetes Cluster Federation, or Kubefed for short [34], to facilitate application deployments, e.g., deploying the Prometheus monitoring service across the clusters. Kubefed provides a unified mechanism to coordinate configurations and manage multiple Kubernetes clusters from a single set of APIs in a hosting cluster, i.e., the central site in our setup.

2) Datasets and Coding Tools: We used *Python 3.8* to implement all the entities. The CPU and HDD over-utilization faults were injected to VMs using the *Stress-ng* tool [35]. As infrastructure faults exhibit themselves as predominant saturation in resources, in this experiment, we stress CPU or HDD of a VM with a high load, e.g., 80% utilization, to inject CPU or HDD over-utilization faults. The network congestion and packet loss faults were initiated targeting VMs using Ping flood and *Linux Traffic Control*, respectively. The fault injections have a recurrent pattern, i.e., they are recurrently interrupted with a cool down time. Following [36], we assume that the duration of an injection follows a Normal distribution

TABLE II: Experiment parameter settings and default values.

Parameters	Value
Coefficient $:_1$-$:_2$-$:_3$	0.8, 0.1, 0.1
Operator's requirement (Congestion)	{50s, 10MB, 86%}
Operator's requirement (HDD)	{25s, 5MB, 83%}
Operator's requirement (Packet Loss)	{15s, 9MB, 89%}
Operator's requirement (CPU)	{10s, 7MB, 90%}

(with a mean of 120 s and standard deviation of 6 s in our experiment), while the inter-arrival time of the fault injections follows an exponentiated Weibull distribution (with a shape parameter 10 and a shifted value of 120 s in our experiment). The data was collected every 10 s, when we collected the node-level VM statistics (e.g., the CPU, memory, and network statistics). The data for these recurrent faults are manually labelled in two classes based on the timestamp of the fault injection: 0 as non-fault, and 1 as fault. We used the so-called Recursive Feature Elimination (RFE) classifier [37] for feature selection. For the building and training of the fault prediction neural network models, we used the *Keras* (with a *Tensorflow* backend) and *Scikit-learn* libraries, and to realize the environments of our Q-learning models we used OpenAI Gym [38].

TABLE III: Hyper-parameter space of our considered ML models.

Parameter	Value Range
Number of neurons in Convolution, LSTM, Dense layers	[2, 256]
Optimizers	[Adam, Nadam]
Loss functions	[MAE, RMSE]
Number of batch sizes	[8, 128]
Number of epochs	[8, 256]

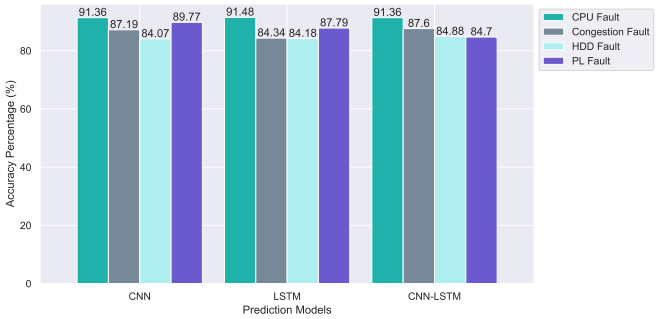


Fig. 5: Accuracy of trained prediction models on two days of data.

B. Evaluation Results

In this section, we initially evaluate and compare the experimental results of our fault prediction models. Next, to evaluate our proposed automated concept drift handling framework, we evaluate the performance of the Drift Adaptation Method Selector and Data Collector RL agents. We illustrate the effectiveness of our framework by demonstrating its persistent accuracy on two weeks data in the presence of multiple drifts during our experiment. Finally, we compare our proposed framework in terms of end-to-end performance with various approaches that use a combination of adaptation methods (e.g., retraining) and data collectors (e.g., increasing the data size gradually).

1) Fault Prediction Results: The purpose of this experiment is to compare the accuracy of the trained prediction models to find the model with the highest prediction accuracy for each type of fault. We trained CNN, LSTM, and CNN-LSTM models with CPU and HDD over-utilization, network congestion, and packet loss fault data. The architecture of the CNN model consists of two Convolutional, one Pooling, one Flatten, and two Dense layers, and the LSTM model has an LSTM and a Dense layer. The CNN-LSTM model has both CNN and LSTM layers, and consists of a sequence of one Convolution, one Pooling, two LSTMs, and one Dense layer(s). Based on our findings in [1], we used RFE to select 10 features out of all the metrics collected by Prometheus to train our models. The features selected by RFE are node-level metrics that are different for various faults, and they include CPU load, available and allocated memory bytes, received and transmitted bytes and packets. We trained multi-variate multi-step time series forecasting models, and classified the results based on the forecast values, where the model's inputs are the selected features and the predicting output is the fault status of the system. In this experiment, the input window was set to 12 samples, and the output window (prediction duration) to 6 samples. This means that we looked at 12 previous samples to forecast 6 samples, which is equivalent to predicting one minute ahead in this case, as we collected the data every 10 s. Note that the sizes of the input and output window are configurable parameters. One minute is set for demonstrative purposes in this experiment, which allows an operator to pro-actively handle the fault, e.g., scaling-up a node for allocating more resources or redirecting traffic to a different node when there is a fault occurrence in a node. We used the first two days out of two weeks data for training and evaluating the models, splitting this dataset into training and testing data with a ratio of 80% and 20%, respectively. The hyper-parameters of our considered models were optimized using Tree-structured Parzen Estimator (TPE) [39]. The optimizers were selected between Adam [40] and Nadam [41] versions of stochastic gradient descent, and the loss function was selected between the Mean Absolute Error (MAE) and the Root Mean Square Error (RMSE) by the TPE. Table III presents the loss functions, optimizers, and the searching space for tuning other hyper-parameters. Fig. 5 illustrates the accuracy of our trained models for CPU, HDD, network congestion, and packet loss faults, and Table IV presents the training times for each model.

As illustrated in Fig. 5, for each type of fault, all trained models have very similar prediction accuracies. For CPU fault, the LSTM model has the highest accuracy of 91.48%, while for packet loss fault, the CNN model has the highest accuracy, with 89.77%. The CNN-LSTM model has the highest accuracy of 87.6% and 84.88% for network congestion and HDD faults, respectively. As shown in Table IV, the training time of different fault prediction models varies. The reason for this is that each model is associated with a different set of hyper-parameters, i.e., number of epochs, neurons, batch size. For example, the LSTM model that predicts HDD over-

TABLE IV: Training time of considered ML models on two days of data.

Model	CNN	LSTM	CNN-LSTM
CPU Fault Training Time (s)	15.8	158.9	135.0
Network Fault Training Time (s)	60.2	114.4	362.4
HDD Fault Training Time (s)	58.3	1104.5	456.3
Packet Loss Fault Training Time (s)	44.3	102.8	925.2

utilization fault has a large number of epochs and neurons in the LSTM layer along with a small batch size, which results in longer training time compared to other prediction models. The training time of the CNN models for all faults is noticeably shorter than the LSTM and CNN-LSTM models, due to the recursive characteristics of LSTM layers. Since the CNN takes a significantly shorter training time to provide a prediction model with an accuracy close to that of the other models, for all four types of faults, we chose to use the CNN model to operate as the prediction model in our proposed framework.

2) Drift Adaptation Method Selector Results: In this experiment, we evaluate the performance of the proposed Drift Adaptation Method Selector component. It is assumed that the edge operator has defined four sets of adaptation KPI requirements, each corresponding to one type of fault, i.e., CPU and HDD over-utilization, network congestion, and network packet loss. The time and resource consumptions in the requirement sets have diverse values and are defined in a way that not all adaptation methods can meet them. We set the accuracy value of the requirement considering the original accuracy of the prediction models before the drift. The operator's requirement sets are listed in Table II. The CNN fault prediction model was selected for all faults based on the results from Section V-B1, and the models have four layers with trainable parameters. This experiment was performed twice training separate RL agents, once with a dataset that contains abrupt drifts [6], and once with a dataset that contains incremental drifts [6]. A maximum data size of 7000 samples is available for performing the adaptations. The coefficients of the reward function were set experimentally using an exhaustive search approach considering the importance of each requirement. The time and resource consumption have the same priority, while the model's accuracy after adaptation has a higher priority. We have experimentally examined various values that meet the aforementioned criteria for setting the coefficients, and selected the parameters that resulted in the most similar choices of the trained agent and a human expert. Training an agent without tuning the coefficients of the reward function may result in a poor selection of the agent, as it may neglect some requirements against the others. For example, by assigning larger values of λ_1 and smaller values of λ_2 and λ_3, the agent will get rewarded for selecting the action that meets the accuracy requirement, while only getting a small negligible penalization for not meeting the time and resource requirements. The coefficients of the reward function and the operator's requirement sets are listed in Table II.

Training and Convergence Evaluation: First, we evaluate the training process and convergence behavior of the RL agent. In this experiment, n is initiated with 1 and the agent continues exploring for 2000 iterations, and then n is decayed by 0•1 every 200 iterations and holds a minimum value of 0•1 for the exploitation phase. Fig. 6 depicts the moving average (with window size 100) of the collected reward by the agent over 12000 training iterations when data contains abrupt drifts, and in each iteration, the agent selects one adaptation method. An RL agent trained with a dataset that contains incremental drifts exhibits a similar collected reward over the training iteration result.

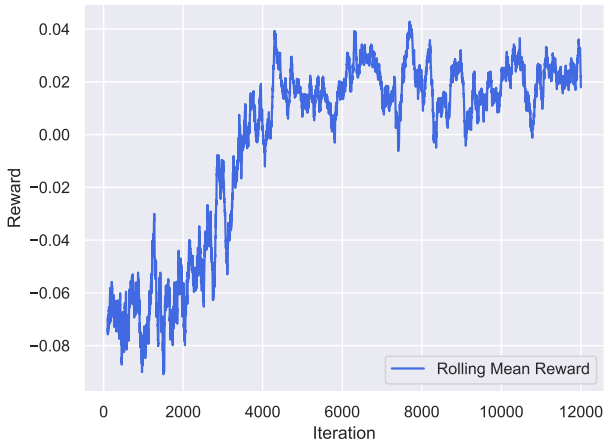


Fig. 6: Reward collected by the Drift Adaptation Method Selector agent while training.

As illustrated in Fig. 6, during the training, as n decreases, the reward collected by the agent increases and converges to positive rewards. This demonstrates that the agent is learning how to select drift adaptation methods that have the minimum adaptation time and resource consumption gap and accuracy gap with the requirements of the operator. Note that according to Eq. (6), the drift adaptation method that fulfills the operator's requirements gains the reward of 0 or higher, and any reward value below 0 does not meet the requirements. The reason for the oscillations of the reward during training (even in the exploitation phase) is that depending on the initial state, the operator's requirements are different and a different value of reward will be collected by the agent. The initial state is set randomly at the beginning of each iteration, which explains the changes in rewards collected by the agent during training. The training time of the Drift Adaptation Method Selector with abrupt drift is 3 hours and 17 minutes for 12000 training iterations. During the training, all state and action pairs are visited several times, which ensures the convergence of the agent after training.

Regret Evaluation: Once the successful training of the RL agent is demonstrated, we compare the choices of the agent with two other algorithms, i.e., choosing a drift adaptation method randomly, and always selecting a specific drift adaptation method (e.g., retraining, partially updating the whole neural network (called TL-3), its lower layers (called TL-1),

(a) Regret evaluation of the Drift Adaptation Method Selector in the presence of abrupt drifts

(b) Regret evaluation of the Drift Adaptation Method Selector in the presence of incremental drifts

Fig. 7: Regret evaluation of the Drift Adaptation Method Selector.

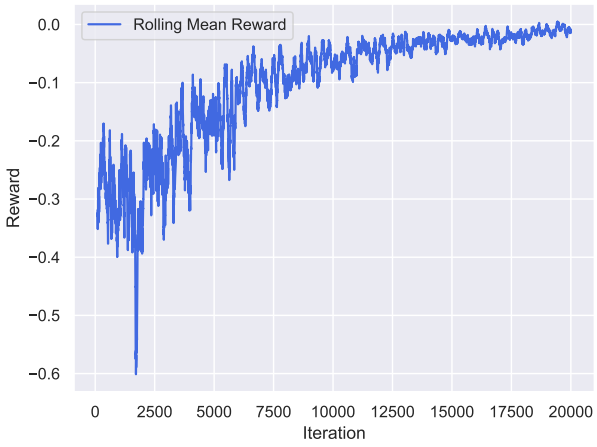


Fig. 8: Reward collected by the Data Collector agent while training.

or its end layers (called TL-2) [8]). We find the difference between the sum of rewards collected by the agent and the two other algorithms for all four types of faults with the sum of rewards collected by the choices of a human expert. This difference is presented as a sum of regret term, indicating how different an algorithm acts compared to a human expert. The algorithm that has the lowest regret is the one closest to a human expert and is the one that performs better than the others. In this experiment, to study the impact of limited available data size on the Drift Adaptation Method Selector, in addition to the existing KPIs in the operator's requirement set (i.e., time and resource consumption, and the model's accuracy), we add a constraint on the data available for adaptation (3000 data samples), and append it to the operator's requirement set as a new KPI. The experiment was performed once with the original KPI set, and once with the new one, with both abrupt and incremental drifts.

Fig. 7a and Fig. 7b illustrate the regret for each drift type, respectively. As illustrated in Fig. 7a and Fig. 7b, for both requirement sets and both abrupt and incremental drifts, the regret of the trained agent is lower compared to other algorithms, which shows that the drift adaptation methods selected by the agent collected the reward closest to human the expert's choices and meets the operator's requirements. The Drift Adaptation Method Selector agent achieves up to 13× lower regret compared to other agents. In Fig. 7a, the constraint on available data does not affect the regret term for the trained agent, and has only a minor effect on other algorithms, except for the random algorithm. However, as shown in Fig. 7b, the regret term changes more significantly for other algorithms while presenting a slight change for the trained agent when the available data constraint is added. This is because when limited data is available, the incremental drift may still be in transition from one concept to the other(s). Therefore, the human expert's choice changes with more probability compared to when abrupt drifts occur, and so the regret of the other algorithms against the human expert changes.

3) Data Collector Results: In this experiment, we evaluate the performance of the proposed Data Collector component. This experiment is performed two times, once with a dataset that contains abrupt drifts, and once with a dataset that contains incremental drifts. We present the results of the CPU fault, as the other faults exhibited similar results.

Training and Convergence Evaluation: First, we evaluate the training process and convergence behavior of the RL agent. The result is presented for the CPU over-utilization fault prediction model and the transfer learning drift adaptation method. Similar results can be obtained for other prediction models and drift adaptation methods. In each episode, the agent's starting state (i.e., the initial data size), is set randomly to make the agent explore the whole state space. The minimum and the maximum data size available for selection is 1000 and 7000, respectively, with granularity of 100. Fig. 8 illustrates the moving average (with window size of 100) of the reward collected by the agent in each iteration during the training time with the data that contains abrupt drifts, while n is modified. The n is initiated with 1 and the agent continues exploring for 2000 iterations. Next, n is decayed by 0•1 every 200 iterations, and holds a minimum value of 0•1 for the exploitation phase.

As illustrated in Fig. 8, as more iterations are passed and the value of n decreases, the reward collected by the agent increases and converges to rewards near 0. This means that the agent learns what data size to select to realize an adapted model that has the closest accuracy to the operator's required

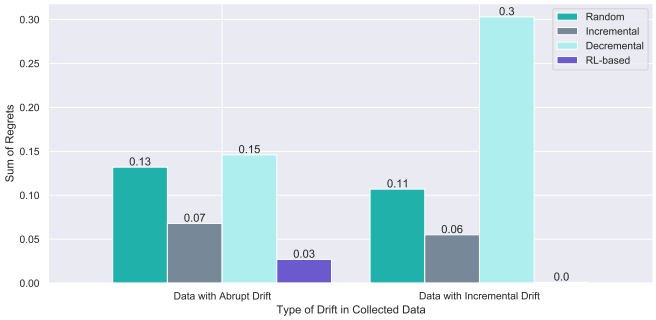
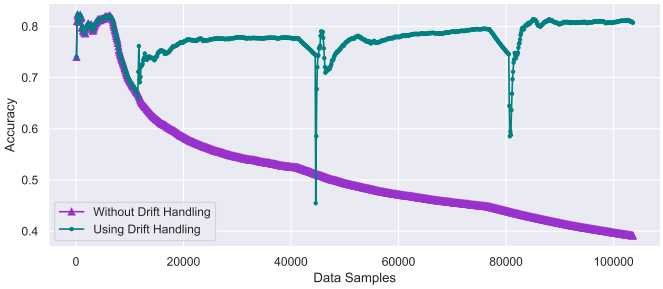
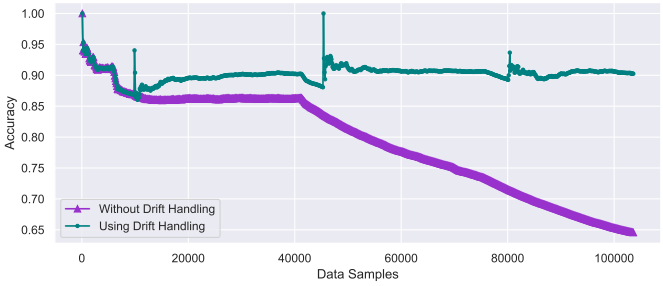


Fig. 9: Regret evaluation of the Data Collector in the presence of abrupt and incremental drifts.



(a) Accuracy over time (during a twelve-day experiment) of trained HDD fault prediction model.



(b) Accuracy over time (during a twelve-day experiment) of trained CPU fault prediction model.

Fig. 10: Accuracy over time of fault prediction models.

accuracy. Note that according to Eq. (7), the drift adaptation method that fulfills the operator's requirement gains the reward of 0 or higher, and any reward value below 0 does not meet the requirements. The reason for the oscillations of the reward during training is that the initial state (i.e., the data size) in each iteration is set randomly so that the agent explores all the state space. Thus, the agent collects different amounts of rewards, especially during the exploration phase. As agent moves to the exploitation phase, even with random initialization, the agent learns which action(s) to select to maximize the reward. The training time of the Data Collector with abrupt drift is 2369.6 seconds for 20000 training iterations. Visiting all the state and action pairs several times during the training, ensures the convergence of the agent.

Regret Evaluation: To evaluate the performance of the trained RL agent, we compare the decisions of the RL agent with two other algorithms, i.e., randomly selecting a data size, and always performing one action (decreasing or increasing the initial data size) until one episode is terminated (called decremental or incremental data collection). The initial data size is set randomly. We find the difference between the sum of the rewards collected by the agent and the two other algorithms on all four type of faults with the sum of rewards collected by the choices of a human expert. This difference is presented as a sum of regret term, indicating how different an algorithm acts compared to a human expert.

Fig. 9 illustrates the sum of the regret terms for both drift types. As illustrated in Fig. 9, for both the abrupt and incremental drifts, the regret of the trained agent is lower compared to that of other algorithms, which shows that the data size that the agent selected resulted in the closest reward to the human expert's choices and fulfills the operator's requirements. The Data Collector agent achieves up to 30× lower regret compared to other agents. The decremental data collection has the highest regret since it tends to collect small amount of data that is not enough for training a model, which leads to lower prediction accuracy and is not selected by the human expert. For incremental drift, the amounts of data that the trained agent selected were the same as the human expert's choices, which resulted in 0 regret value.

4) Accuracy of the Proposed Framework Over Time: After the superiority of the proposed Drift Adaptation Method Selector and Data Collector components were demonstrated, we evaluated the performance of our proposed automated concept drift handling framework and compare it to a system without any drift handling entity. We evaluated our framework on the last twelve days worth of of HDD and CPU over-utilization fault data with multiple abrupt drifts and monitored the accuracy of the HDD and CPU fault prediction models. A similar prediction accuracy result was exhibited when multiple incremental drifts occurred in the data.

Fig. 10 depicts the persistent accuracy of our proposed framework over twelve days while three abrupt drifts occur. As illustrated in Fig. 10a, the first drift occurs around data sample 6000, and the accuracy of the HDD fault prediction model starts to drop. The Drift Adaptation Method Selector agent selects transfer learning, where the first Dense layer in CNN model is fine-tuned, and the Data Collector agent selects 5300 data samples for adaptation. Around data sample 11000, the adaptation process is complete, and the accuracy of the HDD fault prediction model starts increasing to near its original accuracy. However, the accuracy of the model that was not adapted after the drift decreased to nearly 65% after the drift occurred. For the two consecutive drifts, the agents selected the same transfer learning approach and 3900 and 4300 data samples, respectively, and managed to keep the prediction accuracy of the model close to its original value. After the model is adapted, the accuracy may drop for a few samples and then start to improve. This behavior is manifested because the accuracy is calculated in every hundred data samples considering all the model's predictions so far, and after the adaptation, the first accuracy is reported on only one hundred data samples, which is not enough to estimate the true accuracy of the model. However, as more

TABLE V: End-to-end comparison of proposed framework with conventional approaches.

Approach	End-to-End Time (s)	Adaptation Time (s)	Adaptation Resource (MB)	Accuracy (%)	Data Size
RL-based	83.1	13.6	4.7	86.08	4250
Retrain + Inc.	370.1	22.7	6.4	84.23	5825
Retrain + Dec.	235.4	7.3	5.5	75.4	2400
TL-1 + Inc.	140.1	18.0	5.0	86.7	4875
TL-1 + Dec.	120.5	10.8	5.1	78.64	2950
TL-2 + Inc.	291.1	13.4	5.4	82.75	6175
TL-2 + Dec.	123.0	3.9	4.5	79.25	2375
TL-3 + Inc.	373.9	12.4	5.6	84.24	5800
TL-3 + Dec.	183.6	10.7	5.6	84.31	2675

predictions become available, the true accuracy of the model can be calculated. Similarly, Fig. 10b illustrates the accuracy of the CPU fault prediction model in the presence of three abrupt drifts. The Drift Adaptation Method Selector agent selects transfer learning, where the first convolutional layer in CNN model is fine-tuned and the Data Collector agent selects 3900 data samples for adapting to the first drift, and 4100 samples for adapting to the second and third drift (see Fig. 10b). In summary, Fig. 10 shows the effectiveness of our proposed framework in maintaining a persistent accuracy of up to 40% higher compared to a system without any drift handling entity during our twelve-day experiment while satisfying the requirements of the operator.

5) Comparative Results: Since there is no similar framework in the literature that automates handling concept drifts, we compare the end-to-end performance of our framework to various approaches that use a combination of adaptation methods including retraining, TL-1, TL-2, TL-3, and data collectors such as incremental and decremental data collection. The comparison is done in terms of end-to-end time consumption (i.e., the time from when the drift is detected until the adaptation procedure is finished), adaptation time and resource use, accuracy, and selected data size. This experiment was performed on all four fault types (CPU and HDD over-utilization, network congestion, and network packet loss faults), and the results are the average amounts over the four faults. The initial data size for collecting was set to 4000, which is the mid value size in the data collection searching range. It is assumed that the RL agents are trained and that their Q-tables are available for inferencing.

Table V presents the end-to-end performance comparison of our framework with several approaches. As shown in Table V, our proposed solution and the combination of TL-1 drift adaptation method and incremental data collection (TL-1 + Inc.) approach have the highest accuracy after adaptation. However, our proposed solution has a lower end-to-end time and adaptation time, requires less adaptation resources, and collects fewer data samples for adaptation. Furthermore, our proposed framework is flexible and can adjust its choices if new drift types occur or if more complex prediction models need to be adapted. Note that if a new type of drift other than abrupt and incremental drifts occur, it will be necessary to train new RL agents for Drift Adaptation Method Selector and Data Collector. The approaches that use decremental data collection tend to collect small amount of data for training. Therefore, they have lower accuracy and training time and consume less resources compared to the incremental approach. Among the approaches that use incremental data collection, they all have comparable accuracy. However, TL-2 has the lowest accuracy since it preserves the weights of the lower layers, which should be updated since they extract the basic data features that have been drifting.

VI. Conclusions and Future Work

We proposed an automated concept drift handling framework for fault prediction in edge cloud environments using RL. This framework considers the edge cloud operator's requirements (drift adaptation time and resource consumption, and the prediction model's accuracy after adaptation), and uses RL to select the most appropriate drift adaptation method to update the fault prediction model that fulfills the operator's requirements. Moreover, it utilizes RL to automatically select the data size for adaptation to fulfill the operator's requirements. To demonstrate the effectiveness of our proposed framework, we experimentally validated a proof-of-concept of our framework on an edge cloud testbed using Kubernetes. Our proposed Drift Adaptation Method Selector and Data Collector trained agents presented up to 13× and 30× less regret, respectively, against a human expert as compared to other algorithms. Moreover, our proposed framework achieved up to a 40% higher accuracy compared to a system without drift handling, and had superior end-to-end performance compared to other approaches in terms of end-to-end time, adaptation time, adaptation resource, and collected data samples for adaptation. Using our proposed framework, the edge cloud operator can specify a set of adaptation requirements, and the drifts are automatically handled considering those requirements; the operator does not need to examine each drift.

To further improve our proposed automated concept drift handling framework, we suggest several future research directions. The first interesting research avenue is to explore semi-supervised or unsupervised drift detection and adaptation methods to relax the assumption of most drift handling techniques on the availability of labeled data after prediction. Another research direction is to further improve the scalability of the proposed framework so that in case of unplanned downtimes in the cloud site [42], some edge nodes with higher computational power can train the agents to prevent service interruptions. Further, minimizing the data size required for adaptation and performing drift adaptation with limited available data will also play a key role in improving the framework. Another promising research avenue is to evaluate

our framework in large-scale scenarios with large state space, e.g., a high number of requirement sets are defined by the operator, or the neural network has complex architectures with many trainable layers. In such scenarios, depending on the size of the problem, using Q-tables can lead to slow training and convergence or be infeasible. However, since there will be no changes in the action space and it will remain discrete even in the large-scale scenarios, value-based Deep RL (DRL), where a neural network is used to approximate the Q-values can be used to train the agent. Using DRL in large state spaces with limited actions has shown promising performance while using n-greedy strategy, experience replay, target networks, and fine-tuning the approximator to ensure convergence [43] [44].

ACKNOWLEDGMENT

This work is funded by the Ericsson/ENCQOR-5G Senior Industrial Research Chair on Cloud and Edge Computing for 5G and Beyond.

REFERENCES

[1] M. Soualhia, C. Fu, and F. Khomh, "Infrastructure fault detection and prediction in edge cloud environments," in *Proc. ACM/IEEE Symposium on Edge Computing*, Nov. 2019, pp. 222–235.

[2] C. Benzaid and T. Taleb, "AI-driven zero touch network and service management in 5G and beyond: Challenges and research directions," *IEEE Network*, vol. 34, no. 2, pp. 186–194, Feb. 2020.

[3] Z. Khan, J. Lehtomäki, A. Shahid, and I. Moerman, "Real-time edge analytics and concept drift computation for efficient deep learning from spectrum data," in *Proc. IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Jul. 2020, pp. 1290–1291.

[4] L. Yang and A. Shami, "A lightweight concept drift detection and adaptation framework for iot data streams," *IEEE Internet of Things Magazine*, May. 2021.

[5] G. Widmer and M. Kubat, "Learning in the presence of concept drift and hidden contexts," *Machine learning*, vol. 23, no. 1, pp. 69–101, Nov. 1996.

[6] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, "Learning under concept drift: A review," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 12, pp. 2346–2363, Oct. 2018.

[7] A. Shaghaghi, A. Zakeri, N. Mokari, M. R. Javan, M. Behdadfar, and E. A. Jorswieck, "Proactive and AoI-aware failure recovery for stateful NFV-enabled zero-touch 6G networks: Model-free DRL approach," *IEEE Transactions on Network and Service Management*, pp. 1–1, Sep. 2021.

[8] B. Shayesteh, C. Fu, A. Ebrahimzadeh, and R. Glitho, "Auto-adaptive fault prediction system in edge cloud environments in the presence of concept drift," in *Proc. IEEE International Conference on Cloud Engineering (IC2E)*, Oct. 2021, pp. 217–223.

[9] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[10] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, May 1992.

[11] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," in *Proc. IEEE International Conference on Learning Representations (ICLR)*, Apr. 2017.

[12] M. de Prado, N. Pazos, and L. Benini, "Learning to infer: RL-based search for DNN primitive selection on heterogeneous embedded systems," in *Proc. IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar. 2019, pp. 1409–1414.

[13] B. Lyu, H. Yuan, L. Lu, and Y. Zhang, "Resource-constrained neural architecture search on edge devices," *IEEE Transactions on Network Science and Engineering*, Jan. 2021.

[14] A. Samir and C. Pahl, "Detecting and predicting anomalies for edge cluster environments using hidden Markov models," in *Proc. IEEE International Conference on Fog and Mobile Edge Computing (FMEC)*, Jun. 2019, pp. 21–28.

[15] ——, "Self-adaptive healing for containerized cluster architectures with hidden Markov models," in *Proc. IEEE International Conference on Fog and Mobile Edge Computing (FMEC)*, Jun. 2019, pp. 68–73.

[16] L. Baier, J. Reimold, and N. Kühl, "Handling concept drift for predictions in business process mining," in *Proc. IEEE on Business Informatics (CBI)*, Antwerp, Belgium, Jul. 2020, pp. 76–83.

[17] A. Liu, G. Zhang, K. Wang, and J. Lu, "Fast switch naïve bayes to avoid redundant update for concept drift learning," in *2020 International Joint Conference on Neural Networks (IJCNN)*, Jul. 2020, pp. 1–7.

[18] B. Celik and J. Vanschoren, "Adaptation strategies for automated machine learning on evolving data," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Mar. 2021.

[19] F. Dong, J. Lu, Y. Song, F. Liu, and G. Zhang, "A drift region-based data sample filtering method," *IEEE Transactions on Cybernetics*, Feb. 2021.

[20] Y. Sun, K. Tang, Z. Zhu, and X. Yao, "Concept drift adaptation by exploiting historical knowledge," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 10, pp. 4822–4832, Feb. 2018.

[21] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *Advances in Neural Information Processing Systems*, Dec. 2014, pp. 3320–3328.

[22] S. Saurav, P. Malhotra, V. TV, N. Gugulothu, L. Vig, P. Agarwal, and G. Shroff, "Online anomaly detection with concept drift adaptation using recurrent neural networks," in *Proc. ACM India Joint International Conference on Data Science and Management of Data*, Jan. 2018, pp. 78–87.

[23] S. Disabato and M. Roveri, "Learning convolutional neural networks in presence of concept drift," in *Proc. IEEE International Joint Conference on Neural Networks (IJCNN)*, Jun. 2019, pp. 1–8.

[24] M. N. Fekri, H. Patel, K. Grolinger, and V. Sharma, "Deep learning for load forecasting with smart meter data: Online adaptive recurrent neural network," *Elsevier Applied Energy*, vol. 282, p. 116177, Jan. 2021.

[25] D. P. Bertsekas and A. Scientific, *Convex Optimization Algorithms*. Athena Scientific, 2015.

[26] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[27] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Springer Machine learning*, vol. 8, no. 3-4, pp. 293–321, 1992.

[28] C. Jin, Z. Allen-Zhu, S. Bubeck, and M. I. Jordan, "Is Q-learning provably efficient?" in *Proc. International Conference on Neural Information Processing Systems*, Dec. 2018, pp. 4863–4873.

[29] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Machine learning-based scaling management for kubernetes edge clusters," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 958–972, Jan. 2021.

[30] K. Kaur, S. Garg, G. Kaddoum, S. H. Ahmed, and M. Atiquzzaman, "Keids: Kubernetes-based energy and interference driven scheduler for industrial IoT in edge-cloud ecosystem," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4228–4237, Sep. 2019.

[31] Y. Han, S. Shen, X. Wang, S. Wang, and V. C. Leung, "Tailored learning-based scheduling for kubernetes-oriented edge-cloud system," in *Proc. IEEE Conference on Computer Communications (INFOCOM)*, May 2021, pp. 1–10.

[32] "Stan's robot shop," https://github.com/instana/robot-shop, accessed Feb. 2022.

[33] "Prometheus," https://prometheus.io, accessed Feb., 2022.

[34] "Kubefed," https://github.com/kubernetes-sigs/kubefed, accessed Feb. 2022.

[35] "Stress-ng," https://wiki.ubuntu.com/Kernel/Reference/stress-ng, accessed Feb., 2022.

[36] A. Netti, Z. Kiziltan, O. Babaoglu, A. Sîrbu, A. Bartolini, and A. Borghesi, "A machine learning approach to online fault classification in HPC systems," *Elsevier Future Generation Computer Systems*, vol. 110, pp. 1009–1022, Sep. 2020.

[37] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik, "Gene selection for cancer classification using support vector machines," *Springer Machine Learning*, vol. 46, no. 1-3, pp. 389–422, Jan. 2002.

[38] "OpenAI Gym," https://gym.openai.com, accessed Feb., 2022.

[39] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," *Advances in Neural Information Processing Systems*, vol. 24, pp. 2546–2554, Dec. 2011.

[40] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. IEEE International Conference on Learning Representations (ICLR)*, May 2015.

[41] T. Dozat, "Incorporating Nesterov momentum into Adam," in *Proc. IEEE International Conference on Learning Representations (ICLR)*, May 2016.