Machine Learning for Fault Prediction in Clouds

Behshid Shayesteh

A Thesis

in the

Concordia Institute

for

Information Systems Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of

Doctor of Philosophy (Information Systems Engineering)

at Concordia University

Montréal, Québec, Canada

July 2024

© Behshid Shayesteh, 2024

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Behshid Shayesteh**

Entitled: **Machine Learning for Fault Prediction in Clouds**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Information Systems Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

______________________________ Chair
Dr. Ferhat Khendek

______________________________ External Examiner
Dr. Alberto Leon-Garcia

______________________________ External to Program
Dr. Anjali Agarwal

______________________________ Examiner
Dr. Jamal Bentahar

______________________________ Examiner
Dr. Chadi Assi

______________________________ Supervisor
Dr. Roch Glitho

Approved by ______________________________
Dr. Jun Yan
Chair of Department or Graduate Program Director

July 18, 2024
____
Date of Defence

____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Machine Learning for Fault Prediction in Clouds

Behshid Shayesteh, Ph.D.

Concordia University, 2024

The vast adoption of cloud computing has increased the size and complexity of data centers, increasing possibility of faults. Fault can negatively impact the performance, availability, and reliability of cloud services, leading to significant maintenance cost and revenue loss for cloud service providers. Therefore, fault prediction in clouds is a critical task. Machine Learning (ML) is increasingly used for this purpose due to their pattern recognition capabilities. While predicting faults in clouds using ML enables a proactive approach to prevent faults, building accurate prediction models that can maintain their performance in dynamic clouds is challenging. One problem is concept drift, where changes in data distribution can degrade model performance. Similarly, feature drift, which is changes in feature relevancy, can also degrade the model performance. Additionally, models accuracy is influenced by data-related parameters, necessitating selection of these parameters to achieve a high model performance. Existing ML-based fault prediction solutions do not focus on adaptability to dynamic conditions like concept or feature drift. Additionally, selecting data-related parameters to balance model performance and resource consumption is not addressed in current literature.

This thesis mainly focuses on addressing the challenges of employing ML models for predicting faults and predicting application performance degradation caused by faults in cloud environments. We first propose a concept drift adaptation algorithm for fault prediction in clouds using Reinforcement Learning (RL). This algorithm considers the cloud operator's requirements, and uses RL to select the most appropriate drift adaptation method as well as data size for adaptation that fulfills

the requirements. Second, we propose a feature drift adaptation solution for adapting the model to feature drifts while predicting application performance degradation in clouds. This solution consists of a feature drift detector that monitors the performance of the prediction model as well as the feature importance, and a feature drift adaptor that measures the drift severity to adapt the prediction model. Finally, we propose a multi-objective optimization algorithm to select the training data size, data sampling interval, input window, and prediction horizon for training an ML model that predicts application performance degradation in clouds.

Acknowledgments

First and foremost, I express my sincere gratitude to my Ph.D. supervisor, Prof. Roch Glitho. I appreciate all his contributions of time, ideas, and resources to make my Ph.D. experience productive and exciting. Beyond his guidance, I am grateful for the opportunities and experiences he facilitated, which have been essential in my academic growth.

I gratefully acknowledge my Ph.D. committee members, Prof. Anjali Agarwal, Prof. Chadi Assi, and Prof. Jamal Bentahar for their time, effort, and constructive comments. I would also like to extend my appreciation to the external examiner, Prof. Alberto Leon-Garcia, for accepting to serve on my Ph.D. thesis committee.

I am thankful to my collaborator at Ericsson, Dr. Chunyan Fu, for all the enlightening discussions, comments, and collaboration. Working with you was a great pleasure that shaped me to be a motivated, dedicated, and detail-oriented researcher. I would also like to thank Dr. Amin Ebrahimzadeh for all the insightful comments, contributions, and collaborations.

I am thankful to my colleagues in the TSE lab for their companionship and support. Special thanks to Mahsa Raeiszadeh for all the insightful discussions and the fun lunch and coffee breaks.

Many thanks to my friends for their support and encouragement. Special thanks to Dr. Maryam Amini for being a constant source of motivation since the days we were preparing to start our Ph.D. journeys back in IUST. I am also grateful to Mahdokht Shashaei and Nastaran Hakimi for always being there for me, no matter where we lived in the world.

I owe a special thanks to my brother Behnoud. I could always count on you since I remember. Thank you for being so supportive that for the past five years, I always felt you are so close, no matter how far. To my parents, I am profoundly grateful for the immeasurable sacrifices and the

unwavering support you provided, enabling me to pursue my dreams. Words fall short of expressing how grateful I am for all the love you have shown me along the way.

Last but not least, to my better half, Mohsen Amoei, I am blessed beyond words to have had your love, encouragement, and support in ups and downs of this journey. Thank you for always being by my side, for being extremely understanding and supportive, for always cheering me up and motivating me, and for believing in me. Without you and your support, none of this would have been possible.

Table of Contents

List of Figures x

List of Tables xii

List of Abbreviation xiv

1 Introduction **1**

1.1 Overview 1

1.2 Challenges 2

1.3 Thesis Contributions 4

1.3.1 Concept Drift Adaptation Algorithm for ML-based Fault Prediction [1][2] . 4

1.3.2 Feature Drift Adaptation Algorithm for ML-based Fault Prediction [3][4] . 5

1.3.3 Data-related Parameter Selection Algorithm for ML-based Fault Prediction [5] 5

1.4 Background Information 6

1.4.1 Cloud Computing 6

1.4.2 Edge Computing 6

1.4.3 Fault Prediction 7

1.4.4 Machine Learning 7

1.5 Thesis Outline 10

2 Requirements and Related Work **11**

2.1 Illustrative Use Case 11

2.2 Requirements 11

2.2.1 General Requirements of ML-based Fault Prediction in Clouds 12
2.2.2 Requirements of Concept Drift Adaptation in ML-based Fault Prediction . 13
2.2.3 Requirements of Feature Drift Adaptation in ML-based Fault Prediction . . 14
2.2.4 Requirements of Data-related Parameter Selection in ML-based Fault Prediction 14
2.3 Related Work 15
2.3.1 ML-based Fault Prediction in Clouds [6] 15
2.3.2 Concept Drift Adaptation in ML-based Fault Prediction 18
2.3.3 Feature Drift Adaptation in ML-based Fault Prediction 21
2.3.4 Data-related Parameter Selection in ML-based Fault Prediction 23
2.4 Conclusion 25

3 Concept Drift Adaptation Algorithm for ML-based Fault Prediction 26

3.1 Introduction 26
3.2 Illustrative Use case 27
3.3 System Model 27
3.4 Proposed Concept Drift Adaptation Algorithm 29
3.4.1 Drift Adaptation Method Selector 33
3.4.2 Data Collector 36
3.4.3 Complexity Analysis 37
3.5 Performance Evaluation 38
3.5.1 Experiment Settings 38
3.5.2 Evaluation Results 40
3.6 Conclusion 50

4 Feature Drift Adaptation Algorithm for ML-based Fault Prediction 51

4.1 Introduction 51
4.2 Illustrative Use Case 52
4.3 System Model 53
4.4 Proposed Feature Drift Adaptation Algorithm 55

4.4.1 Feature Drift Detector 58
4.4.2 Feature Drift Adaptor 59
4.4.3 Complexity Analysis 61
4.5 Performance Evaluation 62
4.5.1 Experiment Settings 62
4.5.2 Evaluation Results 64
4.6 Conclusion 77

5 Data-related Parameter Selection Algorithm for ML-based Fault Prediction 79
5.1 Introduction 79
5.2 Illustrative Use case 80
5.3 System Model 82
5.4 Problem Formulation 84
5.5 Proposed Data-related Parameter Selection Algorithm 86
5.6 Performance Evaluation 89
5.6.1 Experiment Settings 90
5.6.2 Evaluation Results 92
5.7 Conclusion 98

6 Conclusion, Discussion, and Future Work 99
6.1 Conclusion 99
6.2 Impact of Proposed Algorithms on Fault Management in Clouds 100
6.3 Future Work 101
6.3.1 Concept Drift and Feature Drift Adaptation 101
6.3.2 Integration of Fault Prediction and Prevention Solutions 102
6.3.3 Multi-fault Prediction 102
6.3.4 Explainability of Fault Prediction Models 102

Bibliography 103

List of Figures

2.1 An illustrative use case of an application deployed in cloud environment [6]. 12

3.1 Generic system model of a cloud environment. 29

3.2 Architecture of the proposed automated drift adaptation solution. 30

3.3 Drift Adaptation Method Selector steps. 33

3.4 Lab setup for evaluating the proposed concept drift adaptation solution. 39

3.5 Accuracy of trained prediction models on two days of data. 41

3.6 Reward collected by the Drift Adaptation Method Selector agent while training. . . 44

3.7 Regret evaluation of the Drift Adaptation Method Selector. 45

3.8 Reward collected by the Data Collector agent while training. 46

3.9 Regret evaluation of the Data Collector in the presence of abrupt and incremental drifts. 47

3.10 Accuracy over time of fault prediction models. 48

4.1 Generic system model of an ML model predicting application performance degradation by monitoring application KPIs in a cloud environment. 54

4.2 Architecture of the proposed feature drift adaptation solution. 57

4.3 Lab setup for evaluating the proposed feature drift adaptation algorithm. 63

4.4 Post-adaptation F1-score vs. number of data samples used for adaptation for migration, Change-Fault-Type (CFT)-1, and CFT-2 datasets. 74

4.5 F1-score over time of application performance degradation prediction model. . . . 75

5.1 Illustrative use case with 5G core services deployed in cloud. 81

5.2 Generic system model of a prediction optimizer in clouds environments. 83

5.3 Steps of the proposed data-related parameter selection algorithm. 87

5.4 Lab setup for evaluating the proposed surrogate-assisted NSGA-II multi-objective optimization algorithm. 90

5.5 Pareto front approximations of the considered optimization algorithms. 94

5.6 Normalized hypervolume obtained by the considered optimization algorithms on UERT and UERR datasets. 95

5.7 Search time of the considered optimization algorithms to obtain the Pareto front approximation on UERT and UERR datasets. 95

List of Tables

2.1 Evaluation of the related work on ML-based fault prediction in clouds. 18

2.2 Evaluation of the related work with respect to the requirements of concept drift adaptation algorithm. 20

2.3 Evaluation of the related work with respect to the requirements of feature drift adaptation algorithm. 22

2.4 Evaluation of the related work with respect to the requirements of data-related parameter selection algorithm. 25

3.1 Lab setup parameters and default values. 39

3.2 Hyper-parameter space of our considered ML models. 41

3.3 Training time of considered ML models on two days of data. 41

3.4 Experiment parameter settings and default values. 43

3.5 End-to-end comparison of proposed solution with conventional approaches. 49

4.1 Evaluation of the feature selection approach and prediction model using the considered datasets. 66

4.2 Evaluation of the proposed feature drift detector on migration, CFT-1, and CFT-2 datasets. 70

4.3 Feature drift adaptation evaluation of the proposed solution on migration, CFT-1, and CFT-2 datasets. 73

5.1 Summary of key notations. 84

5.2 Lab setup parameters and default values. 90

5.3 Comparison of the optimal data-related parameters obtained in the Pareto front approximation of the considered algorithms on UERT and UERR datasets. 97

List of Abbreviations

3GPP Third Generation Partnership Project

ADWIN Adaptive Windowing

AI Artificial Intelligence

AMF Access and Mobility Management Function

ANN Artificial Neural Network

AUSF Authentication Server Function

AutoML Automated Machine Learning

Bi-LSTM Bi-directional Long-Short Term Memory

CFT Change-Fault-Type

CNN Convolutional Neural Network

CUSUM Cumulative Sum

DCFS Dynamic Correlation-based Feature Selection

DNN Deep Neural Network

ELDA Expanded Linear Discriminant Analysis

ELM Extreme Learning Machine

GA Genetic Algorithm

GAN Generative Adversarial Network

GBRT Gradient Boosted Regression Tree

GRU Gated Recurrent Unit

HDD Hard Disk Drive

IaaS Infrastructure-as-a-Service

ICMP Internet Control Message Protocol

IMSI International Mobile Subscriber Identity

IoT Internet of Things

k-NN K-Nearest Neighbor

KPI Key Performance Indicator

LDA Linear Discriminant Analysis

LET Limited Epoch Training

LR Logistic Regression

LSTM Long-Short Term Memory

MAE Mean Absolute Error

ML Machine Learning

MLP Multi-Layer Perceptron

NAS Neural Architecture Search

NSGA-II Non-dominated Sorting Genetic Algorithm-II

OSS/BSS The Operation/Business Support System

PCF Policy Control Function

PM Polynomial Mutation

QDA Quadratic Discriminant Analysis

QoS Quality-of-Service

RAN Radio Access Network

RF Random Forest

RFE Recursive Feature Elimination

RGF Regularized Greedy Forest

RL Reinforcement Learning

RMSE Root Mean Square Error

RNN Recursive Neural Network

RS Random Search

SBX Simulated Binary Crossover

SMART Self-Monitoring, Analysis and Reporting Technology

SMF Session Management Function

SVM Suport Vector Machine

TL Transfer Learning

TLCC Time-Lagged Cross Correlation

TPE Tree-structured Parzen Estimator

UDM Unified Data Management

UE User Equipment

VM Virtual Machine

VNF Virtual Network Functions

Chapter 1

Introduction

1.1 Overview

The vast adoption of cloud computing has led to a significant increase in the size and complexity of data centers, resulting in an increased possibility of the occurrence of faults [7]. A fault is an event that occurs in a system that disrupts the intended normal operation of the system [8]. The occurrence of fault can negatively impact the performance, availability, and reliability of cloud services and result in significant maintenance cost and loss of revenue for cloud service providers. Therefore, it is crucial to recognize the potential faults and predict them before occurrence.

The large-scale cloud environments and the diversity of potential faults that may occur due to various causes including misconfigurations, hardware malfunctions, or network connectivity issues necessitates an automated approach for fault prediction [8]. Moreover, the diverse and high-dimensional nature of the data generated in cloud including system logs, performance metrics of resources, and application data makes it challenging to analyze the data and design proper fault prediction solutions [9]. Machine Learning (ML) is capable of discovering patterns and automatically extracting insights from such large amount of data of various heterogeneous sources [9]. This positions ML as an effective solution for analyzing data and predicting faults in complex large-scale settings such as cloud environments. Additionally, ML can fulfill the vision of zero-touch networks, where fault prediction is achieved without human intervention [9]. Automated ML-based fault prediction systems can self-sustainably build and maintain models to predict a diverse range of faults

that occur in cloud environments [2].

While predicting faults in cloud using ML enables a proactive approach to prevent faults in clouds, building accurate prediction models that can maintain their performance in dynamic cloud environments is not an easy task. One problem is occurrence of concept drift [10]), where changes in the data distribution of cloud performance metrics, which are used for training these models, can cause the models' performance to degrade over time. Similarly, feature drift [11], which refers to the changes in the relevance of features used for training the model for fault prediction, can also degrade model performance over time. Additionally, the accuracy of ML models is influenced by several data-related parameters, necessitating selection of these parameters to achieve a high model performance.

This thesis mainly focuses on addressing these challenges of employing ML for predicting faults as well as predicting application performance degradations caused by these faults in cloud environments. In the following subsections, we first discuss the challenges of employing ML-based fault prediction models followed by the main thesis contributions. Next, we provide background information about important concepts related to the thesis. Finally, we present the thesis outline.

1.2 Challenges

The challenges of employing ML model for fault prediction in clouds are summarized as follows:

- **Concept drift adaptation in ML-based fault prediction in clouds:** When employing ML models for fault prediction, it is usually assumed that the underlying distribution of the data used for training the model is stationary. However, the metrics of cloud systems can be affected by unexpected events, such as sudden or gradual changes in the workload within a certain time period [12], and/or permanent or ephemeral anomalies [13], causing frequent changes in the distribution of data that is used for training fault prediction models, a phenomenon commonly known as concept drift. To ensure service performance, availability, and reliability, it is important to prevent the performance degradation of the 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 technique (e.g., partial updating, ensemble learning, or retraining) for drift adaptation [14], and determine the amount of data to be used for drift adaptation. However, adapting an ML model to the drift in large-scale systems like clouds is complex due to the diversity of the concept drift impacts, which requires different adaptation methods. Thus, to minimize the overall model management overhead, there is a need for an automated concept drift adaptation solution.

- **Feature drift adaptation in ML-based fault prediction in clouds:** Another challenge when maintaining the performance of ML models is adapting the model to feature drifts, which refers to the changing relevance of selected features to the learning task, leading to the model's inaccurate predictions. Feature drifts in clouds may occur due to dynamic nature of cloud environments, where frequent workload changes such as adding, removing or migrating workloads may be observed, or hardware infrastructure configurations may change. For example, when using ML models to predict performance degradation of applications deployed in clouds caused by infrastructure faults, changes in the type of infrastructure faults leading to application performance degradation can significantly affect the performance of the ML model. This is due to the fact that once the underlying fault causing an application performance degradation changes, the relevance between the selected features (i.e., performance metrics of a cloud system, such as CPU or memory usage) and application performance degradation indicators may change. This highlights the need for a mechanism to detect feature drifts, update the features, and adapt the model to the feature drift to maintain the performance of the prediction model.

- **Data-related parameter selection in ML-based fault prediction in clouds:** The performance of ML models is influenced by several factors, including training data size (i.e., total number of data samples available for training the prediction model), data sampling interval (i.e., the time difference between two consecutive collected data samples), input window (i.e., the number of data samples the model considers to make a prediction), and prediction horizon (i.e., the number of future data samples the model should predict) [15]. Traditionally, the process of determining these data-related parameters for training an ML model has relied

on experimental methods and the analytical skills of human experts. However, this approach faces significant challenges in cloud environments where multiple ML models may be required to predict various faults or performance degradation across various applications. The scale and complexity of these environments make it impractical for human experts to manually test and select the optimal set of parameters for each model. Consequently, there is a need for an automated solution that can select these parameters. Such a solution should aim to balance two key objectives, i.e., maximizing the prediction accuracy of the models, which is the primary indicator of their performance, and minimizing the resources required for collecting and storing data, with an intent to reduce data collection and storage costs and computational overhead of processing large datasets for the model training.

1.3 Thesis Contributions

The existing ML-based fault prediction solutions for clouds do not address the challenges described in Section 1.2. This thesis aims at tackling the challenges of employing ML models for fault prediction in clouds. It makes three main contributions presented in this section. Each of the contributions corresponds to a challenge addressed by this thesis.

1.3.1 Concept Drift Adaptation Algorithm for ML-based Fault Prediction [1][2]

In the first contribution, we tackle the challenge of concept drift adaptation while predicting faults in clouds using ML models. We train deep learning models to predict infrastructure faults such as CPU over-utilization faults in cloud environments. To maintain the model performance, an effective technique (e.g., partial updating, ensemble learning, or retraining) for drift adaptation should be selected. We propose an algorithm that considers the cloud operator's requirements (i.e., drift adaptation time and resource consumption, and the prediction model's accuracy after adaptation), and uses Reinforcement Learning (RL) to automatically select a drift adaptation method that best fulfills the operator's requirements to update the fault prediction. Moreover, the algorithm utilizes RL to select the amount of data required for adaptation so that the operator's requirements are fulfilled. We experimentally validate a proof-of-concept of our algorithm on a Kubernetes-based

testbed. The results show the proposed algorithm can effectively maintain the ML model performance in presence of concept drifts and had superior performance compared to other approaches in terms of drift adaptation time, adaptation resource, and number of data samples for adaptation.

1.3.2 Feature Drift Adaptation Algorithm for ML-based Fault Prediction [3][4]

In the second contribution, we tackle the challenge of feature drift adaptation while predicting application performance degradation caused by faults in clouds using ML models. We propose a feature drift adaptation algorithm to automatically adapt to feature drifts while predicting application performance degradation in cloud environments. This algorithm detects feature drifts by monitoring the performance of the prediction model as well as the feature importance. The algorithm measures the feature drift severity to adapt the prediction model to the drift by performing either feature re-selection and re-training the model or dropping the irrelevant features and fine-tuning the prediction model. We experimentally build a proof-of-concept of our algorithm in a Kubernetes-based testbed, and evaluate and compare the proposed solution to four benchmarks. Our results demonstrate that the proposed algorithm can effectively detect the feature drift and update the features and adapt the prediction model to the drift and can maintain the performance of the prediction model.

1.3.3 Data-related Parameter Selection Algorithm for ML-based Fault Prediction [5]

In the third contribution, we tackle the challenge of selecting data-related parameters to train an ML model that predicts application performance degradation in clouds. We propose a surrogate-assisted multi-objective optimization algorithm based on Non-dominated Sorting Genetic Algorithm-II (NSGA-II) to automate selection of the training data size, the data sampling interval, the input window, and the prediction horizon for training an ML model that predicts application performance degradation in clouds. This algorithm maximizes the performance of the prediction model while minimizing resource consumption of data collection and storage. To evaluate the effectiveness of the proposed algorithm, we experiment on a Kubernetes-based cloud testbed, where a 5G core is deployed. The data-related parameters selected by the algorithm are used for training models that predict the degradation of the 5G core Key Performance Indicator (KPI)s. The results demonstrated

that the proposed algorithm can achieve optimal solutions in two scenarios while reducing the solution search time compared to the considered benchmarks.

1.4 Background Information

In this section, we present brief background information related to this thesis. The background information covers cloud computing, edge computing, fault prediction, and ML.

1.4.1 Cloud Computing

Cloud computing is most commonly defined as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [16]." Cloud computing has a distinct set of characteristics that make this paradigm unique and powerful, such as the appearance of infinite computing resources available on-demand, elimination of an upfront commitment by users, ability to pay per use, multiple tenancies, scalability, elasticity, and rapid provisioning of services and applications [17]. The most important enabling technology behind the unique characteristics of the cloud is virtualization. This technology provides virtual resources from real physical resources with the goal of efficient usage of limited physical resources. Each cloud data center is composed of physical machines, and each physical machine can support a set of Virtual Machine (VM).

1.4.2 Edge Computing

In the cloud computing paradigm, most computations happen in the centralized cloud. However, such a computing paradigm may suffer from longer latency, which weakens the user experience. To this end, the edge computing paradigm was introduced, which refers to the enabling technologies allowing computation to be performed at the edge of the network in the proximity of data sources. Edge can refer to any computing and network resources along the path between data sources and cloud data centers. Examples of edge include smartphones, gateways, and micro data center [18]. Edge cloud is an environment that consists of one central cloud and one or multiple edges.

1.4.3 Fault Prediction

A fault is an event that occurs in a system that disrupts the intended normal operation of the system [8]. Cloud environments are susceptible to various fault types. At the infrastructure level, hardware faults (e.g., over-utilization in resources such as CPU, HDD, or memory) and network faults (e.g., packet loss or network congestion) can occur in both physical and virtual resources. Moreover, software faults (e.g., unhandled exceptions) can occur at the software level. Such faults, whether related to infrastructure or software, may result in the failure of jobs or degradation in performance of applications deployed in clouds [6]. These faults can originate from misconfigurations, human (maintenance) errors, malicious intrusions, and/or excessive load [19].

Since the occurrence of faults can negatively impact the performance, availability, and reliability of cloud services and result in significant maintenance cost and loss of revenue for cloud service providers, it is crucial to recognize the potential faults and predict them before occurrence while system is fully functional [20][6]. Fault prediction is identifying whether a fault will occur in the near future based on an assessment of the current state of a monitored system.

1.4.4 Machine Learning

ML is a subset of Artificial Intelligence (AI) which provides machines the ability to learn automatically and improve from experience without being explicitly programmed [21]. ML models with shallow structures including Artificial Neural Network (ANN) and Extreme Learning Machine (ELM) are used for training fault prediction models [6]. ANNs are neural networks with one hidden layer trained by processing data to iteratively adjust internal parameters (weights) to minimize the difference between the model's predictions and the actual fault occurrence status [22]. ELM, a single-hidden-layer neural network, trains the model by initializing input layer weights randomly and computing output layer weights analytically, leading to faster training [23].

With advancements in ML algorithms, deep learning algorithms have gained prominence in data analytics tasks. Deep learning is a class of ML algorithms based on ANN that utilizes multiple hidden layers to capture inter-dependencies between features and can handle high-dimensional

data [24]. Therefore, deep learning algorithms including Deep Neural Network (DNN), Convolutional Neural Network (CNN), Recursive Neural Network (RNN), Long-Short Term Memory (LSTM), Bi-directional Long-Short Term Memory (Bi-LSTM), and hybrid CNN-LSTM are also employed for fault prediction tasks [6]. CNNs were designed to analyze structured, arrayed data including images, signals or time series. RNNs excel at processing sequential data where the order matters, such as time-dependent metrics. LSTMs improve RNNs by effectively learning long sequences without losing crucial information, and Bi-LSTMs enhance LSTMs by considering past and future context [25] [26]. Hybrid models like CNN-LSTM combine CNN's sequence processing with LSTM's long-term memory capabilities. The ability in sequence and pattern analysis makes CNN, RNN, LSTM, Bi-LSTM, and CNN-LSTM models well-suited for learning from cloud system performance metrics with temporal dependencies [6]. Ensemble learning algorithms are also used for fault prediction in clouds by combining the prediction of multiple models using bagging or boosting approach. The bagging approach, such as Random Forest (RF), trains multiple models of the same type, e.g., decision trees, using random subsets of the training data and combines their predictions [27]. The boosting approach such as AdaBoost, trains multiple models, e.g., decision trees, sequentially, with each new model correcting its predecessor's errors. The final prediction is a combination of all models' outputs [28]. By combining multiple prediction models, ensemble learning algorithms can improve the overall performance and generalization of the combined model.

Training an ML model involves several steps: 1) data collection is carried out to gather the necessary data for training the model, 2) the collected data undergoes pre-processing, which includes transforming the data into a desired format, cleaning it of any inconsistencies, labeling the data when required, selecting the most relevant features or engineering new features, and dividing the data into training, validation, and testing sets, 3) the actual training of the ML model is performed, where the training data is fed to the specific ML model, 4) the model is refined using the validation dataset, where the performance of the model is evaluated, and measures to further enhance its performance, such as hyperparameter tuning, are undertaken, 5) the model is evaluated using the testing data to assess its performance and generalization ability, ensuring it can perform well on unseen data.

While there are a few types of problems in ML, the four well-known problems are classification,

regression, clustering, and timeseries forecasting, which are defined in the following.

- Classification: when the output of the ML model is one of a finite set of values, the learning problem is called classification and is called Boolean or binary classification if there are only two values [29]. Some well-known classification algorithms are ANN, Logistic Regression, and Decision Trees.

- Regression: when the output of the ML model is a numerical value, the learning problem is called regression. ANN and Linear Regression are well-known regression algorithms.

- Clustering: when the ML model aims to group similar examples of data in a cluster, the learning problem is called clustering. K-means and mean-shift clustering are examples of clustering algorithms.

- Timeseries forecasting: when the ML model aims to predict a number based on the timeseries data, it is called a timeseries forecasting problem. Timeseries data is a sequence of numerical data points that have time dependencies. RNN and LSTM are among well-known timeseries forecasting algorithms.

The problem of predicting faults can be seen as a timeseries forecasting problem or a classification problem. In timeseries forecasting, the pattern of the performance metrics of cloud or the occurrence of faults is predicted. It may or may not need a classification method to map the forecasted output as fault or no-fault. In classification, the training data is labeled with fault or no-fault. The trained model can classify whether the upcoming data is predicted to be fault or non-fault.

There are three well-known types of ML algorithms, i.e., supervised learning, unsupervised learning, and RL, that are defined as follows.

- Supervised Learning: the algorithm observes some example input-output pairs and learns a function that maps from input to output [29]. It uses labeled data to train the ML model.

- Unsupervised Learning: the algorithm learns patterns in the input, even though no explicit feedback and guidance are supplied [29]. It uses unlabeled data to train the ML model.

- RL: the algorithm (agent) interacts with an environment and learns from a series of reinforcements, i.e., rewards or punishments [29]. RL is a reward-based algorithm that trains an agent with no predefined data and no supervision. Q-learning [30] is one of the most widely known RL algorithm that seeks to find the best action to take given the current state by successively updating the evaluation of the long-term reward (the Q-value) of actions at each state.

Our literature review indicated that fault prediction in cloud is most commonly tackled by supervised learning, which relies on learning from labeled data. Fault prediction algorithms require labeled historical data that describes the state of the system, enabling the algorithms to predict the system's future state based on its current state. The performance of prediction algorithms is subject to the quality and availability of datasets, which includes the performance metrics, logs, or alarms of various severities collected from real-world cloud environments. While obtaining realistic datasets for fault prediction can be challenging, in the literature, there are both publicly available (e.g., [25][31]) and proprietary datasets (e.g., [2][27]) collected from real-world clouds to train fault prediction models.

1.5 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 discusses the requirements of ML-based fault prediction in clouds and provides a critical review of the state-of-the-art. Chapter 3 presents an algorithm for adaptation concept drift while predicting faults in clouds, and Chapter 4 presents a feature drift adaptation algorithm for ML-based fault prediction in clouds. Chapter 5 presents an algorithm for data-related parameters selection for training ML models predicting faults in clouds. Chapter 6 concludes this thesis and provides future research direction for this research work.

Chapter 2

Requirements and Related Work

In this chapter, we first describe an illustrative use case. Next, we use the use case to derive a set of requirements for ML-based fault prediction in clouds. Finally, we review the related work in light of these requirements.

2.1 Illustrative Use Case

In this section, we present an illustrative use case to derive the general requirements of ML-based fault prediction in cloud environments. Fig. 2.1 presents an illustrative use case of a web application is deployed in cloud environment provided as Infrastructure-as-a-Service (IaaS), where the end-users send requests to access the service provided by the web application. Faults can occur in different resources (i.e., compute, storage, or network) in data centers, including physical and virtual resources, potentially leading to performance degradation or service failure of the web application deployed in this environment. ML models are trained using the cloud performance metrics to predict these faults or application performance degradation caused by these faults so that preventive steps can be taken to avoid fault occurrences.

2.2 Requirements

In this section, considering the illustrative use case, we discuss the general requirements of ML-based fault prediction in cloud environments. Next, we present the requirements specific to each

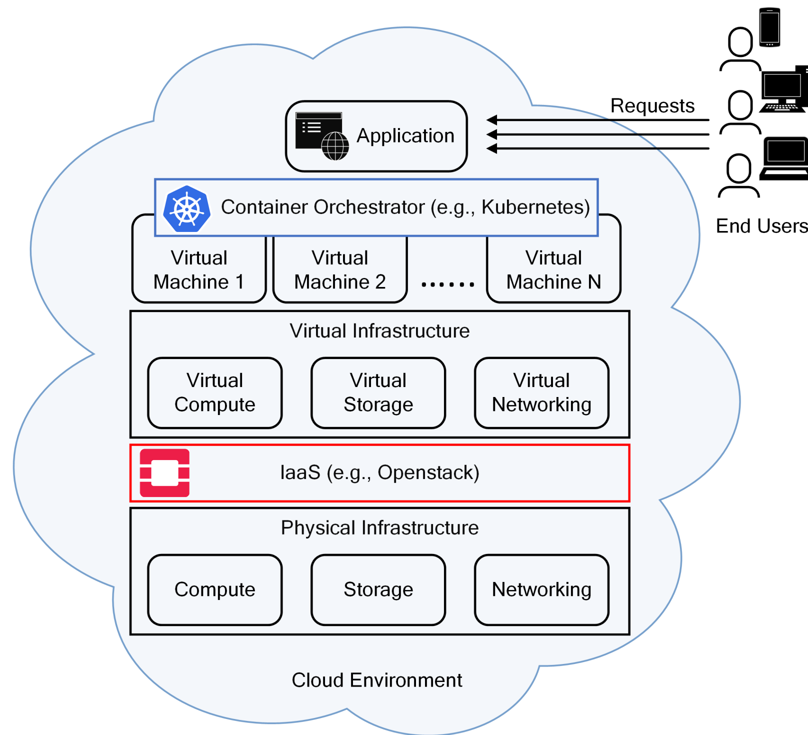


Figure 2.1: An illustrative use case of an application deployed in cloud environment [6].

solution that address the challenges of employing ML for fault prediction in clouds identified in Section 1.2 of this thesis.

2.2.1 General Requirements of ML-based Fault Prediction in Clouds

Accuracy: The ML models should achieve a high accuracy in predicting faults, while minimizing false alarms (false positives) and missed faults (false negatives). False alarms can trigger unnecessary prevention mechanisms, while missed faults can propagate in clouds and result in service failures.

Timely prediction: The ML models should be capable of delivering timely prediction of faults considering the prevention mechanism required to avert those faults. For example, if the ML model predicts server failures, the prediction horizon should span several days to accommodate the time required for replacing or repairing a failed server. However, for predicting CPU over-utilization in virtual resource, the prediction horizon can be within a few minutes. This is because scaling up a virtual machine and allocating more resources when there are enough resources available in the

cloud is not a time-consuming prevention mechanism.

Scalability: The ML models should ensure scalability while processing large amount of data and features. The fault prediction algorithm should not run out of computing resources and can train models within reasonable time frame without loss of accuracy as the size of the input data and features grow.

Adaptability: The ML models should have adaptability to the dynamic cloud environment. The fault prediction model should maintain its performance in dynamic conditions of cloud environment. For example, the dynamic changes in the workload of an application deployed in clouds can cause frequent changes in the distribution of the data (i.e., resource usage metrics) used for training the model. This can result in poor model performance since the data used for training the model and the data for inferencing have different distributions.

Explainability: The ML model should provide explainability about the decision process behind the fault predictions. The decisions of the fault prediction model should be explainable to provide insights into the underlying cause of a fault. In an automated fault prediction system, explainability can help selecting proper prevention mechanism to avert the fault.

2.2.2 Requirements of Concept Drift Adaptation in ML-based Fault Prediction

While the requirements identified in Section 2.2.1 were general to ML fault prediction models, in this section, we present the requirements specific to a solution that adapts to concept drifts while predicting faults in clouds.

Autonomy: The algorithm that adapts the prediction models to concept drift should be an automated solution with no to limited human intervention. This is because in large-scale cloud systems, there are diverse ML models with different architectures that may suffer from performance drops due to concept drifts of various severities. Therefore, an automated solution is required to monitor and adapt all models to potential concept drifts.

Post-adaptation performance: The concept drift adaptation algorithm should consider the model post-adaptation performance while adapting the model to the concept drift. This is to ensure that the model's prediction performance after adaptation is comparable to that of the original model.

Adaptation time consumption: The concept drift adaptation algorithm is required to minimize

the concept drift adaptation time so that the impact of the concept drift can be minimal, i.e., the performance of the prediction model recovers fast from the occurrence of drift.

Adaptation resource consumption: The concept drift adaptation algorithm can be required to constrain the resource consumption for performing concept drift adaptation in case of adapting to concept drift at resource-constrained edge.

2.2.3 Requirements of Feature Drift Adaptation in ML-based Fault Prediction

In this section, we present the requirements specific to a feature drift adaptation solution while predicting faults in clouds using ML models.

Feature drift detection: The feature drift adaptation algorithm is required to detect feature drifts before performing feature drift adaptation to avoid unnecessary adaptations in a high-dimensional feature space.

Applicable to high-dimensional feature space: The feature drift adaptation algorithm is required to be applicable to high-dimensional feature space like performance metrics of a cloud system that can consist of thousands of features.

Post-adaptation performance: The feature drift adaptation algorithm is required to effectively update the features and adapt the prediction model to the drift so that the model's post-adaptation performance is comparable to its original performance.

Resource efficient: The feature drift adaptation algorithm can be required to constrain the resource consumption for feature drift adaptation in case of adaptation at resource-constrained edge.

2.2.4 Requirements of Data-related Parameter Selection in ML-based Fault Prediction

In this section, we present the requirements specific to a data-related parameter selection algorithm for training ML-based fault prediction models in clouds.

Data-related parameter selection: The algorithm is required to select the data-related parameters such as training data size, data sampling interval, input window, and prediction horizon for training a ML prediction model.

Autonomy: The data-related parameter selection algorithm is required to be an automated solution with no human intervention. This is because numerous ML models may be employed in cloud environments to predict various faults with different characteristics, which makes it impractical for human experts to manually test and select the optimal set of parameters for each model.

Multi-objective: While selecting the data-related parameters, the algorithm is required to consider performance of the ML model as well as the resource consumption for data collection and storage as two criteria for parameter selection.

Model performance estimation: Considering that training an ML model to evaluate its performance for every set of data-related parameters is time and resource consuming, the parameter selection algorithm is required to estimate the performance of the model accurately without fully training prediction models for each evaluation.

2.3 Related Work

In this section, we present the state-of-the-art for the challenges identified in this thesis. First, we review the existing ML-based fault prediction solutions in clouds. Next, we review the prior-art on the three main contributions of this thesis, i.e., concept drift adaptation in ML-based fault prediction, feature drift adaptation for ML-based fault prediction, and data-related parameter selection for training fault prediction models.

2.3.1 ML-based Fault Prediction in Clouds [6] [1]

In this section, we review the most representative works that use ML for predicting faults in clouds. Based on the type of ML algorithm used by these works, we divide them to three categories of shallow ML, deep learning, and ensemble learning. While these works predict faults, they did not address the challenges identified in Section 1.2 tackled in this thesis.

[1] This sub-section is based on a published paper: [6] B. Shayesteh, A. Ebrahimzadeh, R. Glitho, "Machine Learning for Predicting Infrastructure Faults and Job Failures in Clouds: a Survey", in *IEEE Communication Magazine*, 2024, to appear

A) Shallow Machine Learning

An ANN-based algorithm was introduced in [22] to predict job execution failures in cloud data centers. The authors provided failure analysis of unsuccessful executions, uncovering the pattern and the inter-dependencies among task and job failures. The evaluation results indicated that the ANN-based approach exhibited superior predictive performance compared to Linear Discriminant Analysis (LDA), Expanded Linear Discriminant Analysis (ELDA), Quadratic Discriminant Analysis (QDA), Logistic Regression (LR), and Suport Vector Machine (SVM) for job failure prediction. An ELM-based algorithm for online job failure prediction in cloud was proposed in [23]. This work predicted the submitted jobs' status and updated the prediction model incrementally as new sequences of jobs arrived. Utilizing an online sequential ELM enhances adaptability in dynamic cloud environments, enabling the model to adjust to changes such as concept drifts by incremental model updating.

B) Deep Learning

Deep learning algorithms consume more time and computational resources for training fault prediction models compared to shallow ML. However, deep learning excel at handling large amounts of data, and can effectively learn complex fault patterns and dependencies between various indicators of faults in cloud.

The authors of [32] studied using RNNs for assessing the health status and predicting failures of Hard Disk Drive (HDD)s using Self-Monitoring, Analysis and Reporting Technology (SMART) data attributes in large-scale storage systems such as cloud data centers. SMART attributes include power-on hours, spin up time, and temperature. Their findings indicated that the proposed RNN-based models can outperform previous sequence independent models and short-term sequence dependent models. An algorithm based on multi-layer Bi-LSTM for predicting task and job failures in cloud data centers was proposed in [25]. The evaluations revealed that the Bi-LSTM-based solution outperformed RNN and LSTM without increasing training time, since Bi-LSTM's bidirectional learning improves recognition of sequence patterns. A DNN-based algorithm proposed in [7] aimed

to predict probability of task failure in clouds. The failure prediction was used to develop an optimized task scheduling strategy to prevent task execution failures and minimize the total energy consumption. The findings highlighted the superior predictive performance of DNN over SVM and Naive Bayes models in anticipating task failure probabilities. The authors of [33] proposed an LSTM-based algorithm to predict HDD health level according to its time to failure in cloud data centers. The algorithm used SMART attributes and temporal analysis to predict HDD health level. The results illustrated the superior performance of LSTM-based models compared to DNN and RF in predicting the health status of HDD. A framework for predicting task failures in cloud is proposed in [31]. They leveraged on DNN and CNN to predict task failures. The framework optimized remedy actions selection process for a given task. The evaluation results demonstrated that the proactive failure handling framework reduced resource usage in the cloud. An algorithm based on Bi-LSTM is proposed in [34] to predict CPU over-utilization fault in virtual resources of edge clouds realized using Kubernetes. The predictions were used to trigger a service migration mechanism for service recovery. The proposed Bi-LSTM algorithm outperformed CNN-LSTM, Gated Recurrent Unit (GRU), and LSTM models. Moreover, the proposed proactive migration framework helped avoiding significant Quality-of-Service (QoS) latency violations compared to the default Kubernetes migration method.

C) Ensemble Learning

Training ensemble learning-based fault prediction models is complex and may require more time and computational resources compared to shallow ML and deep learning, which train a single fault prediction model. However, ensemble learning models offer models with better generalization since they combine multiple models.

A solution to predict node failures in an ultra-large-scale cloud environment is proposed in [27]. The solution consisted of a data analytics pipeline combining a data pre-processing step with training of an RF-based prediction model. The authors made the RF model explainable through studying the importance scores of the features that contributed to the prediction output. The evaluation results demonstrated the superior performance of RF compared to LSTM and a hybrid RF and LSTM

model in terms of prediction performance and training time. An AdaBoost-based algorithm to predict HDD failures for large-scale storage systems like cloud data centers was proposed in [28]. The authors employed transfer learning to predict failures in minority disks, i.e., the new disks that gradually enter data centers, using the knowledge learned from existing (majority) disks. SMART data attributes were used to train AdaBoost models with Gradient Boosted Regression Tree (GBRT), Regularized Greedy Forest (RGF), SVM, and RNN as weak classifiers.

Table 2.1 presents a summary of the existing ML-based fault prediction solution for clouds. The challenges identified in thesis such as concept drift and feature drift adaptation are a subset of adaptability, which has not been addressed by the current literature. Moreover, selecting data-related parameters for training models was not studied in the literature of ML-based fault prediction in clouds.

Table 2.1: Evaluation of the related work on ML-based fault prediction in clouds.

References	**Requirements**				
	Accuracy	**Timely Prediction**	**Scalability**	**Adaptability**	**Explainability**
[22][25][28]	✓	✓	-	-	-
[23]	✓	-	✓	-	-
[32][7][33] [31]	-	✓	-	-	-
[34]	-	-	-	-	-
[27]	-	✓	✓	-	✓

2.3.2 Concept Drift Adaptation in ML-based Fault Prediction

In this section, we review the concept drift adaptation approaches that use ML model for their learning task, and we categorize these works by the concept drift adaptation techniques they used, i.e., retraining, ensemble learning, and partial updating. 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 ML model.

A) Retraining

To adapt the model to the concept drift, retraining method trains a new model from scratch with the data after the drift, and then discards the obsolete model. The work in [35] 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 [36] proposed a new detection algorithm for concept drift adaptation 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 [13], a drift-adaptive ML-based framework for Internet of Things (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 hyperparameter optimization for adaptation. The work in [37] studied the effect of concept drift on the performance of Automated Machine Learning (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 [38], 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 Neighbor (k-NN), Naive Bayes, decision trees, and 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.

B) Ensemble learning

The papers in this category, preserve a set of ML models (i.e., an ensemble) for concept drift adaptation. In [39], an ensemble learning method based on transfer learning [40] 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 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.

C) 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 data distribution. In [41], the authors presented an anomaly detection and concept drift adaptation method for RNNs by incrementally updating the prediction model without detecting a drift first. The authors of [42] 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 [43] proposed an online adaptive RNN for electricity load forecasting use case, that adapts the deep learning model to the concept drift by re-tuning hyperparameters (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.

Table 2.2 presents a summary of the works that adapt to concept drifts when using ML models. Since the focus of these works is not fault prediction in clouds, the general requirements of ML-based fault prediction models is not included in the table.

Table 2.2: Evaluation of the related work with respect to the requirements of concept drift adaptation algorithm.

References	**Requirements**			
	Autonomy	**Post-adaptation Performance**	**Adaptation Time Consumption**	**Adaptation Resource Consumption**
[35][36][37][38] [43][41][42]	-	✓	-	-
[13][39]	-	✓	-	✓

2.3.3 Feature Drift Adaptation in ML-based Fault Prediction

In this section, we review the works that adapt ML models to feature drifts. Although concept drift adaptation is a relatively well-established research area, the literature is quite scarce on feature drift adaptation. We categorize the feature drift adaptation techniques into two groups of active or passive adaptation approaches, considering whether they detect a feature drift to trigger adaptation or they continuously update the selected features and/or prediction model to reflect the latest changes in the feature relevancy. While the reviewed paper aim at solving the feature drift adaptation problem, the passive approaches do not detect a drift and are not applicable to high-dimensional feature space like metrics of a cloud system, since periodically performing feature selection is very time and resource consuming and may not be feasible. Moreover, the existing passive approaches either do not discriminate between concept drift or feature drift and cannot effectively adapt the model to the drift, or perform evolutionary-based feature selection for adaptation, which has a high time and resource overhead and is not guaranteed to converge in high-dimensional feature space.

A) Passive Feature Drift Adaptation

The papers in this category implicitly adapt to feature drift by updating the features that are relevant to the learning task as new data arrives without detecting the occurrence of drifts. In [44], a feature drift adaptation approach is proposed specifically for k-NN algorithm. The distances in k-NN were weighted dynamically according to the current discriminative power of each feature calculated using entropy over a sliding window. Therefore, as features became irrelevant, they contributed less in distance calculation of k-NN. The authors in [45] proposed a dynamic feature selection for clustering data streams. It continuously performed feature selection over a sliding window and stored a score for each feature and defined a threshold to decide selection of the feature. The changes in features relevancy were reflected in the score enabling feature drift adaptation. In [46] a dynamic feature selection based on genetic algorithm is proposed to change the selected features over time as new streaming data arrived. It trained classifiers using different combinations from an initial set of features, and the features used for top performing classifiers were further used for cross-over and mutation so that weak features are removed for next iterations.

B) Active Feature Drift Adaptation

As opposed to passive adaptation approaches that do not detect a feature drift to start the adaptation, the papers in this category detect a feature drift to trigger adaptation procedure, which makes them more suitable for high-dimensional feature spaces, where continuous feature selection is not feasible. The work in [47] proposed Dynamic Correlation-based Feature Selection (DCFS), an active feature drift adaptation solution for data streams. It detected the feature drift using commonly used concept drift detection algorithms, e.g., Adaptive Windowing (ADWIN) [48]. Upon receiving a drift warning, i.e., a drop in based learner's performance, it used feature re-selection using backward elimination and re-trained the base learner. The authors of [49] proposed a framework to classify data streams that may contain feature drifts. They detected feature drift by monitoring the relevance of features and class label using mutual information. For feature drift adaptation, they updated the selected features using an evolutionary algorithm and retrained the base classifier. The work was further extended in [50] by introducing a memory mechanism to store the features that were selected before the occurrence of feature drift to re-use for accelerating feature selection in case of recurrent feature drifts. The authors in [51] did not detect feature drift over time, but they proposed a feature selection solution based on degree of feature drift and genetic algorithm for device identification in IoT. They used an entropy-based metric to identify and discard features that exhibit a drift across different datasets. They further used genetic algorithms to select features for the classification task.

Table 2.3 presents a summary of the feature drift adaptation works. Since the focus of these works is not fault prediction in clouds, the general requirements of ML-based fault prediction models is not included in the table.

Table 2.3: Evaluation of the related work with respect to the requirements of feature drift adaptation algorithm.

References	Approach	Requirements			
		Feature Drift Detection	Applicable to High-dimension Feature Space	Post-adaptation Performance	Resource Efficient
[44][45][46]	Passive	-	-	✓	-
[47][49][50]	Active	✓	-	✓	-

2.3.4 Data-related Parameter Selection in ML-based Fault Prediction

In this section, we review the existing work on selecting data-related parameters (training data size, data sampling interval, input window, or prediction horizon) while training an ML model, followed by existing multi-objective optimization techniques adopted for optimizing ML models. The works that studied selecting the data-related parameters tested these parameters in a limited scope for a specific model, highlighting the need for an automated and general solution to select these parameters. Moreover, the existing work only consider one objective, i.e., model performance, to select the parameters and other objectives such as data storage resource consumption was not considered while selecting the parameters. On the other hand, the works that optimize multiple objectives when refining an ML model, focus on optimizing architectures or number of features rather than tuning data-related parameters.

A) Selecting Data-related Parameters

The works in [15] [52][53] tune parameters such as training data size, data sampling interval, input window, or prediction horizon while training an ML model. In [15], the authors studied the impact of training data size, data sampling interval, and prediction horizon on the accuracy of an ML-based QoS prediction model. They reported Mean Absolute Error (MAE) of the prediction model when trained using various combinations of the parameters. Ref. [52] studied the effect of data sampling interval on wearable-based fall detection system on four ML models with the purpose of finding the minimum data sampling interval that results in the highest performance. The authors of [53] studied how different sampling intervals, ML models and hyperparameters influence accuracy of an ML model predicting power consumption of a wind turbine. They evaluated two ML models, four sampling rates and prediction horizons to tune the parameters.

B) Multi-Objective Optimization for ML models

The works in [54][55][56][57][58] utilized multi-objective optimization for optimizing various aspects of ML models, such as model hyperparameters, architectures, and model types. The authors of [54] proposed a multi-objective optimization for AutoML to maximize the performance of the

ML model while minimizing model size to enable deployment on mobile devices. They proposed a Genetic Algorithm (GA)-based approach with customized search space sampling. In [55], a framework for DNN Neural Architecture Search (NAS) to maximize the performance of the model while minimizing model size is proposed. This framework used an evolutionary algorithm to find models suitable for deployment resource-constrained devices. The authors of [56] proposed a resource-aware AutoML framework to build ML models while considering multiple objectives such as resource consumption, inference time, and performance simultaneously. The platform employed a combination of Bayesian Optimization and GA to find Pareto optimal candidates. In [57], the authors simultaneously optimized the model hyperparameters and features that will be used for training, focusing on maximizing predictive performance while minimizing the number of features. They employed and compared Bayesian Optimization and NSGA-II to solve their optimization problem. The authors of [58] proposed a multi-objective AutoML approach to find a trade-off between model performance and computational complexity of the ML algorithm. They utilized an NSGA-II-based approach with a modified dominance and cross-over definition. While these works optimize two conflicting objectives, they overlooked the significant costs associated with training ML models to assess these objectives during optimization. Moreover, these works did not focus on tuning data-related parameters.

Another group of papers [59][60][61] have adopted surrogate-assisted multi-objective optimization for ML models. This method employs surrogate models to estimate the objective functions, which are expensive to evaluate due to the high cost of training the ML model for each evaluation. The authors of [59] proposed a surrogate-assisted hardware-aware NAS by predicting the Pareto front rankings of architectures for multiple objectives, such as accuracy and resource consumption. They extended the work in [60] by enhancing the inference time of the surrogate model. Ref. [61] trained an LSTM model offline as a surrogate for multi-objective optimization in a wrapper-based feature selection problem in air quality forecasting. The objective was to find a trade-off between model performance and number of features.

Table 2.4 presents a summary of the literature review on data-related parameter selection. Since the focus of these works is not fault prediction in clouds, the general requirements of ML-based fault prediction models is not included in the table.

Table 2.4: Evaluation of the related work with respect to the requirements of data-related parameter selection algorithm.

References	Requirements			
	Data-related Parameter Selection	**Autonomy**	**Multi-objective**	**Model Performance Estimation**
[15][52][53]	✓	-	-	-
[54][55][56] [57][58]	-	✓	✓	-
[59][60][61]	-	✓	✓	✓

2.4 Conclusion

In this chapter, we first described an illustrative use case and derived the requirements for ML-based fault prediction in clouds. Then, we reviewed the existing algorithms for ML-based fault prediction in the literature and evaluated them based on the identified requirements. Our literature review indicates that the existing fault prediction algorithms do not address the three challenges discussed in Section 1.2.

Chapter 3

Concept Drift Adaptation Algorithm for ML-based Fault Prediction [1]

3.1 Introduction

One limitation of ML-based solutions that use cloud performance metrics for fault prediction is that their predictive performance can be affected by concept drifts. It is important to prevent the performance degradation of the prediction models caused by concept drift by means of effective model adaptation. It is challenging is to automatically select the best concept drift adaptation technique and data size for diverse ML models in large-scale clouds, accounting for varying drift types, model architectures, and cloud operator requirements.

To address this challenge, this chapter proposes an algorithm that considers the cloud operator's requirements (i.e., 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.

The rest of this chapter is organized as follows. First, we present an illustrative use case followed

[1] This chapter is based on two published papers: [1] B. Shayesteh, C. Fu, A. Ebrahimzadeh, R. Glitho, "Auto-adaptive Fault Prediction System for Edge Cloud Environments in the Presence of Concept Drift". In *Proc. IEEE International Conference on Cloud Engineering (IC2E)*, Oct. 2021, pp. 217-223, and [2] B. Shayesteh, C. Fu, A. Ebrahimzadeh, R. Glitho, "Automated Concept Drift Handling for Fault Prediction in Edge Clouds Using Reinforcement Learning," in *IEEE Transactions on Network and Service Management*, vol. 19, no. 2, pp. 1321-1335, June 2022.

by the system model and our proposed RL-based automated concept drift adaptation solution. Next, we describe the lab setup and the implementation results and conclude this chapter.

3.2 Illustrative Use case

In this section, we describe an illustrative use case that highlights the importance of a concept drift adaptation solution for ML models that predict faults in cloud. Let us consider a cloud environment consisting of a central cloud and an edge. An ML model is utilized in this environment to predict impending infrastructure faults, enabling preemptive actions to prevent faults. The prediction model is trained using infrastructure-level metrics relevant to the occurrence of a specific fault. For instance, to predict CPU over-utilization faults, metrics such as CPU load or the time a process waits for CPU are used to train the model.

Maintaining the performance of the fault prediction model in highly dynamic cloud environments poses a significant challenge. A particular challenge is the phenomenon of concept drift, where the distribution of the data used for training the model and the data used for prediction are different. The concept drift degrades the ML model's performance. For example, let us assume that CPU-intensive applications are deployed across the cloud infrastructure, or there is a significant increase in workload due to seasonal business activities or unexpected events. These events can change the CPU usage patterns within the cloud environment. The original CPU over-utilization fault prediction model, which was trained on historical data reflecting a different level of CPU demand, may no longer accurately predict CPU over-utilization under these new conditions. In this case, a concept drift adaptation solution is required to detect concept drifts, collect new data after concept drift occurrence, and determine how to adapt the prediction model to the drift to maintain its performance.

3.3 System Model

Fig. 3.1 illustrates a high-level view of a cloud consisting of one central cloud and one edge, as a representative of many. As shown in Fig. 3.1, we consider a deep learning time series forecasting model, which predicts infrastructure faults such as CPU or HDD over-utilization faults. We assume

that an operator manages several clouds using these fault prediction models. The edge site has an Edge Site Monitor that collects raw training data, i.e., infrastructure-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 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, 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. We assume that the true labels of the arriving data are available using the timestamp-based labeling method defined in [62], which records the fault injection timestamp and labels each sample as fault or non-fault. 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.

In the setting described above, the operator sets various requirements for each edge site specifying that the drift adaptation process should not exceed T_r seconds, consume less than C_r amount of resources, and achieve a minimum post-adaptation accuracy A_r, given that a maximum amount D_{max} 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 T_r, C_r, and A_r (i.e., the requirements of the cloud operator). In retraining, the learned 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

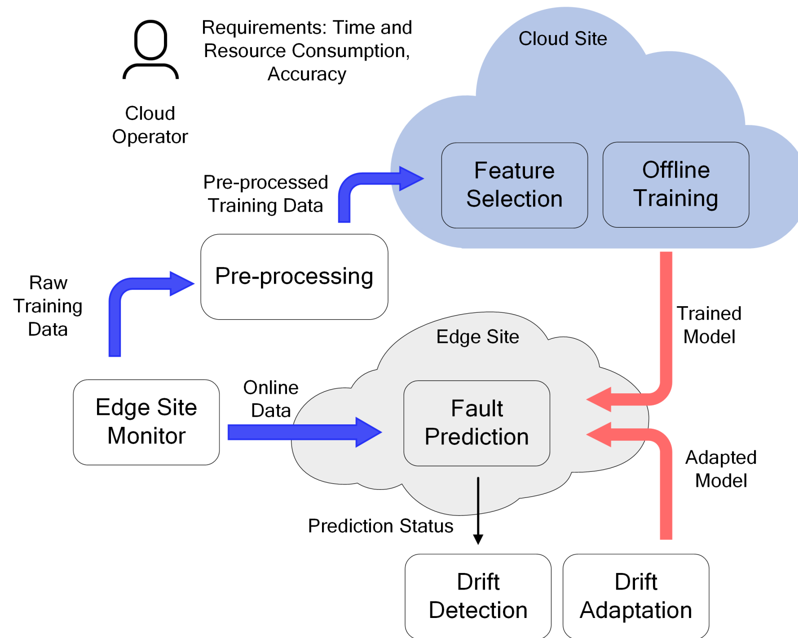


Figure 3.1: Generic system model of a cloud environment.

model, which is also known as Transfer Learning (TL), since some knowledge is transferred from a source model to a target model. 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.

3.4 Proposed Concept Drift Adaptation Algorithm

In this section, we present our proposed automated concept drift adaptation solution. Fig. 3.2 illustrates the architecture of our proposed solution, 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. We assume that the true labels of the arriving data are available using the timestamp-based labeling method defined in [62], which records the fault injection timestamp and labels each sample as fault or non-fault. Using this labeling mechanism, the Drift Detector component receives as input whether the model's

prediction is correct or not (prediction status) and detects the occurrence time of a possible concept drift and generates a drift alert if a concept drift is detected. Error-based concept drift detection techniques (reviewed in [14]) including Cumulative Sum (CUSUM) [63] and ADWIN [48] can be used for concept drift detection. Our previous experiments for concept drift detection in [1] found that CUSUM had a better performance in terms of True Positive ratio and False Positive ratio compared to other error-based concept drift detections. CUSUM monitors the mean of the model's error and alarms a drift when significant deviation is observed in the mean error value. Upon receiving a drift alert, the Drift Adaptation Method Selector component is initialized and reads the cloud operator's requirements from the Drift Adaptation Controller component. The Drift Adaptation Method Selector uses the previously trained RL agent to infer the proper drift adaptation method considering the requirements. 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.

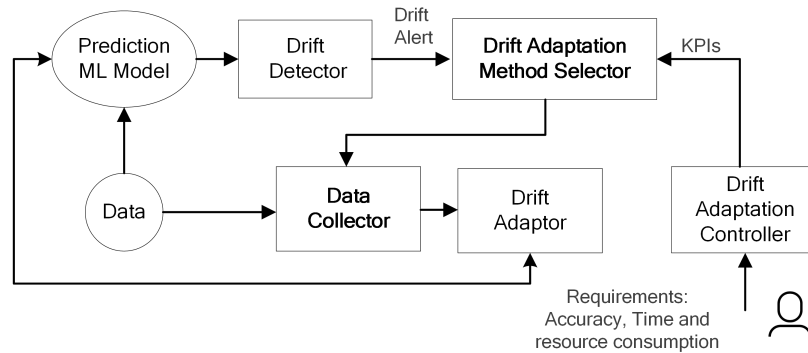


Figure 3.2: Architecture of the proposed automated drift adaptation solution.

Algorithm 1 illustrates the different steps of our proposed automated concept drift adaptation solution. It starts by loading a fault prediction model (M). When the new data (X) with a label (Y) arrives, it is fed to the fault prediction model (M). The inference of the prediction model ($\hat{Y}$) from data (X) along with the true label (Y) of (X) 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 ($\{T_r, C_r, A_r\}$), i.e., the adaptation's time and resource consumption (T_r and C_r), and the model's accuracy (A_r) after adaptation, along with the

Drift Adaptation Method Selector's Q-table (Q_{DAMS}) that was previously trained using the Drift Adaptation Method Selector Algorithm are used to select the drift adaptation method (see lines 11-13). Next, a new Data Collector agent is trained and the Q-table returned by the Data Collector (Q_{DC}) is further used to select the data 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).

Algorithm 1 Automated Concept Drift Adaptation

```
 1: Initialize: Drift Detector, Edge Site Monitor (ESM), Drift Adaptation Controller (DAC)
 2: M ← Load trained fault prediction model
 3: while data arrives do
 4:     X ← Read pre-processed data from ESM
 5:     Ŷ ← M(X)
 6:     prediction_status ← get_prediction_status(Y, Ŷ)
 7:     drift_alert ← drift_detector(prediction_status)
 8:     if drift_alert then
 9:         {T_r, C_r, A_r} ← read requirements from DAC
10:         Q_DAMS ← get Q-table returned by DAMS
11:         adaptor ← get_adapt_method (Q_DAMS, {T, C, A})
12:         Q_DC ← train and get Q-table from Data Collector
13:         data_size ← get_data_size (Q_DC)
14:         new_model ← perform_adapt(adaptor, data_size)
15:         M ← new_model
16:     end if
17: end while
```

Q-Learning: Since the Drift Adaptation Method Selector and Data Collector utilize Q-learning, 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 [64]. 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:

$$G_t = \sum_{k=0}^{T} \gamma^k R_{t+k+1}, \tag{3.1}$$

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:

$$Q(s, a) = \mathbb{E}[\, G_t \mid S_t = s, A_t = a] \,, \tag{3.2}$$

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:

$$Q(S_t, A_t) = (1 - \alpha) Q(S_t, A_t) + \alpha [R_t + \gamma \max_{A'} Q(S_{t+1}, A')], \tag{3.3}$$

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 $Q^*(s, a)$ [65]. 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 [66] 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 [67], 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 [66].

3.4.1 Drift Adaptation Method Selector

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

Fig. 3.3 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 [68], 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 a performance evaluation 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. In the following, we elaborate on the state and action spaces and reward function of the Drift Adaptation Method Selector.

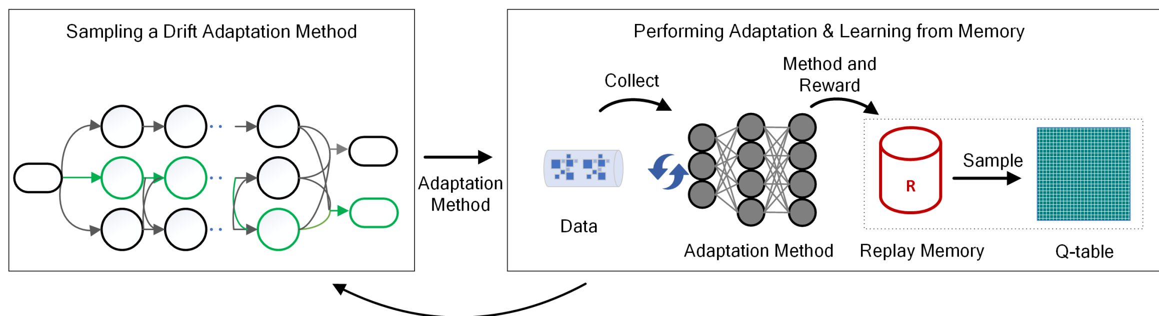


Figure 3.3: Drift Adaptation Method Selector steps.

A) 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, a set that consists of the drift adaptation method's time consumption T_r, resource consumption C_r, and the model's performance after the adaptation, i.e., its accuracy A_r 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 $\{T_r, C_r, A_r\}$ to keep the state space of the problem finite.

The representation of a state that performs an adaptation approach x on the ith layer of a neural network, while the operator's requirement equals $\{T_r, C_r, A_r\}$, is as follows:

$$s = S(i, x, \{T_r, C_r, A_r\}). \tag{3.4}$$

B) Action Space: In each state s, the agent can take action $a \in \mathcal{A}(s)$. 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 S_0, the agent decides which drift adaptation approach to choose for the first layer, where the action is selected from $\mathcal{A}(S_0) = \{$R, F, Z$\}$. Moreover, if the current state's adaptation approach is R, the agent can only select R 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 F or Z, the next action is selected from $\mathcal{A}(S) = \{$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, $\mathcal{A}(S) = \{$P, D$\}$. Available

action(s) in each state s is given by:

$$\mathcal{A}(s) = \begin{cases} \{\text{R, F, Z}\}, & s = S_0 \\ \{\text{R}\}, & s = S(\{1, \ldots, N-1\}, R, \cdot) \\ \{\text{F, Z}\}, & s = S(\{1, \ldots, N-1\}, \{Z \cup F\}, \cdot) \\ \{\text{P, D}\}, & s = S(N, \cdot, \cdot), \end{cases} \tag{3.5}$$

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

C) Reward Function: The agent receives a reward after performing the drift adaptation method it selected. We define the reward of action a_i at a given state s_i as a function of the drift adaptation's time consumption T_i, resource consumption C_i, and accuracy of the model after adaptation A_i, as follows:

$$\mathcal{R}(s, a) = k_1 \frac{(A_i - A_r)}{A_r} - k_2 \frac{(T_i - T_r)}{T_r} - k_3 \frac{(C_i - C_r)}{C_r}, \tag{3.6}$$

where A_r is the operator's required accuracy, and T_r and C_r are the maximum time and resources available for adaptation as specified in operator's requirements. The k_1, k_2, and k_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. (3.6), 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 adaptation time and resource consumption and accuracy 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.

Algorithm 2 Drift Adaptation Method Selector

```
1: Inputs: Number of model's layers (N), Number of explorations for each ϵ (E), Total number of episodes (B)
2: Initialize:
3: replay_memory ← []
4: Q ← 0 for all state-action pairs
5: for (episode = 1 to B) do
6:     Visited states S = [S_0]
7:     Taken actions 𝒜 = []
8:     for (layer = 1 to N) do
9:         generate random number e ∈ [0, 1]
10:        𝒜(S[−1]) ← get possible actions for S[−1] from Eq. (3.5)
11:        if e > ϵ then
12:            a ← max_{a∈𝒜(S[−1])} Q(S[−1], a)
13:        else
14:            a ← random selection in 𝒜(S[−1])
15:        end if
16:        𝒜.append(a)
17:        S_new ← perform action a
18:        S.append(S_new)
19:    end for
20:    ℛ ← perform_adaptation(S) get reward from Eq. (3.6)
21:    memory.append(S, 𝒜, ℛ)
22:    {S_m, 𝒜_m, ℛ_m} ← M random selections from memory
23:    for (S, 𝒜, ℛ in S_m, 𝒜_m, ℛ_m) do
24:        for (s, a in S, 𝒜) do
25:            update Q(s, a) using Eq. 3.3 & reward R
26:        end for
27:    end for
28:    E ← E − 1
29:    if E equals 0 then
30:        decay ϵ & re-initialize E
31:    end if
32: end for
33: return Q
```

3.4.2 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 to learn the amount of data to collect to start the drift adaptation process.

A) 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 I.

B) 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 I. Assuming that the initial state is to collect an amount of d_0 data ($s_{DC,0} = d_0$), and the agent increases the data size, the next state would be $s_{DC,1} = d_0 + I$. It is further assumed that the operator specifies the maximum data size available for data collection D_r as the upper limit.

C) Reward Function: The reward function for Data Collector ($\mathcal{R}_{DC}$) is a function of the gained

accuracy using the collected data A_c and the operator's required accuracy A_r, as follows:

$$\mathcal{R}_{DC}(s_{DC}, a_{DC}) = A_c - A_r, \tag{3.7}$$

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

Algorithm 3 Data Collector

```
1: Inputs: Minimum and Maximum data available (D_0, D_r), Total number of episodes (B_DC), Episode length (L), Number of
   exploration for each ϵ (E), Interval between two consecutive data sizes I
2: Initialize:
3: A_DC ← {increase, decrease}
4: Q_DC ← 0 for all state-action pairs
5: for (episode = 1 to B_DC) do
6:     s_DC ← start in a random initial state
7:     is_episode_done ← False
8:     while ! is_episode_done do
9:         generate random number e ∈ [0, 1]
10:        if e > ϵ then
11:            a ← max_{a∈A_DC} Q_DC(s_DC, a)
12:        else
13:            a ← random selection from A_DC
14:        end if
15:        R_DC ← do action a and get reward from Eq. (3.7)
16:        update Q_DC(s_DC, a) using Eq. 3.3 & reward R_DC
17:        S_DC ← perform action a
18:        L ← L − 1
19:        if L equals 0 or R_DC ≥ 0 or s_DC < D_0/I or s_DC > (D_r − D_0)/I then
20:            is_episode_done ← True
21:            re-initialize L
22:        end if
23:    end while
24:    E = E − 1
25:    if E equals 0 then
26:        decay ϵ & re-initialize E
27:    end if
28: end for
29: return Q_DC
```

3.4.3 Complexity Analysis

In this sub-section, we present a complexity analysis of our proposed concept drift adaptation, Drift Adaptation Method Selector, and Data Collector algorithms. The time complexity of an ϵ-greedy Q-learning algorithm is from the order of the number of steps in an episode and the length of each episode [69]. The space complexity of Q-learning is influenced by the number of states and actions [69]. Therefore, the time complexity of the Data Collector (Algorithm 3) is $\mathcal{O}(B_{DC} \cdot L_{DC})$, and the space complexity is $\mathcal{O}(|\mathcal{A}_{DC}| \cdot S_{DC})$, where $S_{DC} = (D_r - D_0)/I$ is the number of states.

For the Drift Adaptation Method Selector (Algorithm 2), since a replay buffer of size M is used, the time complexity is $\mathcal{O}(B \cdot N \cdot M)$. For the space complexity, since each state is a three-tuple, assuming that there are H requirement sets, the number of states is $|\mathcal{A}| \cdot N \cdot H$, and the space complexity would be $\mathcal{O}(|\mathcal{A}|^2 \cdot N \cdot H \cdot M)$. As for the complexity of automated concept drift adaptation (Algorithm 1), assuming that the size of the arriving data is W, the time complexity is $\mathcal{O}(W \cdot B_{DC} \cdot L_{DC})$ and the space complexity is $\mathcal{O}(W \cdot \mathcal{A}_{DC} \cdot S_{DC})$.

3.5 Performance Evaluation

In this section, we evaluate the performance of our proposed solution in a Kubernetes-based cloud testbed. In the following, after presenting the experiment settings, i.e., the lab setup and fault injection specifications, we present the results of our evaluations.

3.5.1 Experiment Settings

A) Lab Setup: Fig. 3.4 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. One cluster represents the central site while the other two clusters are the edge sites. To cause drifts in the data (i.e., the performance metrics of the nodes) used for training the fault prediction models, we installed Stan's Robot Shop [70], a CPU-intensive application. The load generator of Stan's Robot Shop increases the CPU, memory and network load causing drifts in the nodes' performance metrics. We utilized *Prometheus* [71] to monitor the VMs in Kubernetes clusters and collect the data. The configurations of the VMs are summarized in Table 3.1.

B) Fault Injection: The CPU and HDD over-utilization faults were injected to VMs using the *Stress-ng* tool [72]. 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 [73], we assume that the duration of an injection follows a Normal distribution (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

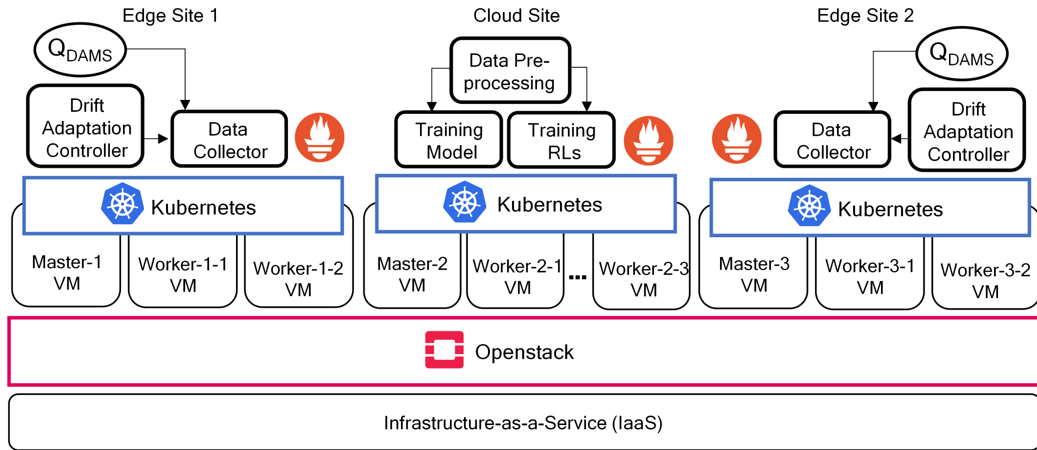


Figure 3.4: Lab setup for evaluating the proposed concept drift adaptation solution.

Table 3.1: 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

exponentiated Weibull distribution (with a shape parameter 10 and a shifted value of 120 s in our experiment). The data was collected every 10 seconds, when we collected the node-level VM metrics (e.g., the CPU, memory, and network metrics). 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 [74] 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 [75].

3.5.2 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 adaptation algorithm, we evaluate the performance of the Drift Adaptation Method Selector and Data Collector RL agents. We illustrate the effectiveness of our solution by demonstrating its persistent accuracy on two weeks data in the presence of multiple drifts during our experiment. Finally, we compare our proposed solution 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).

A) 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 [62], 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. 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) [76]. The optimizers were selected between

Adam [77] and Nadam [78] versions of stochastic gradient descent, and the loss function was selected between the MAE and the Root Mean Square Error (RMSE) by the TPE. Table 3.2 presents the loss functions, optimizers, and the searching space for tuning other hyperparameters. Fig. 3.5 illustrates the accuracy of our trained models for CPU, HDD, network congestion, and packet loss faults, and Table 3.3 presents the training times for each model.

Table 3.2: 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]

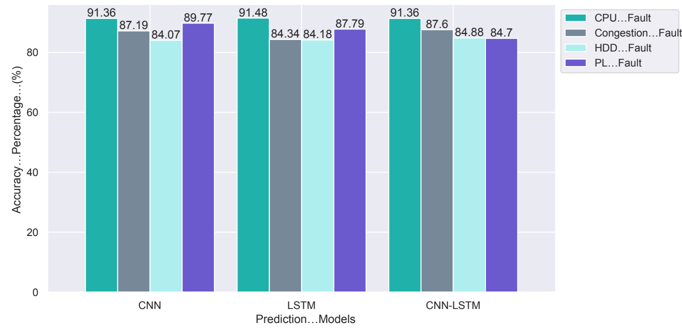


Figure 3.5: Accuracy of trained prediction models on two days of data.

Table 3.3: 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

As illustrated in Fig. 3.5, 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 3.3, the training time of different fault prediction models varies. The reason for this is that each model is associated with a different set of hyperparameters, i.e., number of epochs, neurons, batch size. For example, the LSTM model that predicts HDD over-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 solution.

B) 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 cloud operator has defined four sets of adaptation 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 3.4. The CNN fault prediction model was selected for all faults based on the results from fault prediction results section, 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 [14], and once with a dataset that contains incremental drifts [14]. Abrupt or sudden concept drift is a type of drift that occurs when the data distribution changes in a short time, whereas incremental concept drift occurs when the old data distribution replaces the new distribution over a period of time [14]. 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 k_1 and smaller values of k_2 and k_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 3.4.

Table 3.4: Experiment parameter settings and default values.

Parameters	**Value**
Coefficient k_1, k_2, k_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)	$\{$10s, 7MB, 90%$\}$

Training and Convergence Evaluation: First, we evaluate the training process and convergence behavior of the RL agent. In this experiment, ϵ is initiated with 1 and the agent continues exploring for 2000 iterations, and then ϵ is decayed by 0.1 every 200 iterations and holds a minimum value of 0.1 for the exploitation phase. Fig. 3.6 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.

As illustrated in Fig. 3.6, during the training, as ϵ 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. (3.6), 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

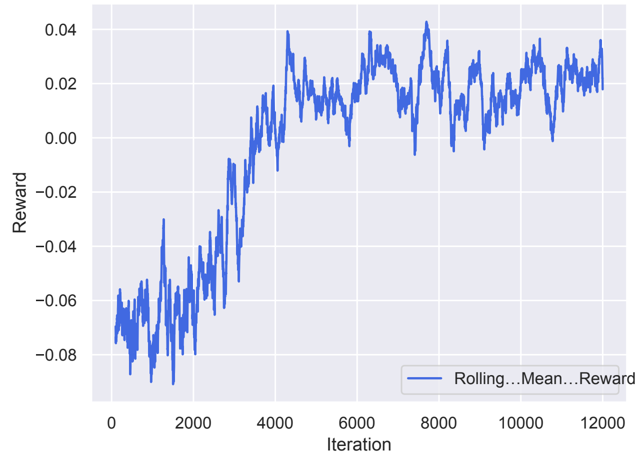
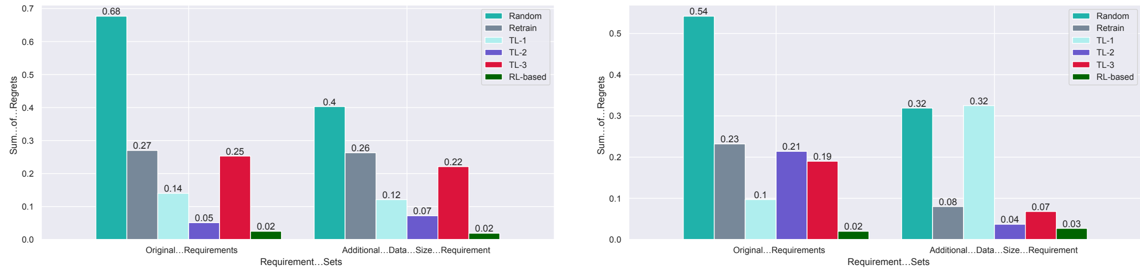


Figure 3.6: Reward collected by the Drift Adaptation Method Selector agent while training.

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), or its end layers (called TL-2) [1]). 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 operator's requirements (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 requirements. The experiment was performed once with the original requirements, and once with the new one, with both abrupt and incremental drifts.



(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

Figure 3.7: Regret evaluation of the Drift Adaptation Method Selector.

Fig. 3.7a and Fig. 3.7b illustrate the regret for each drift type, respectively. As illustrated in Fig. 3.7a and Fig. 3.7b, 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$\times$ lower regret compared to other agents. In Fig. 3.7a, 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. 3.7b, 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.

C) 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 TL 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. 3.8 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 ϵ is modified. The ϵ is initiated with 1 and the agent continues exploring for 2000 iterations. Next, ϵ is decayed by 0.1 every 200 iterations, and holds a minimum value of 0.1 for the exploitation phase.

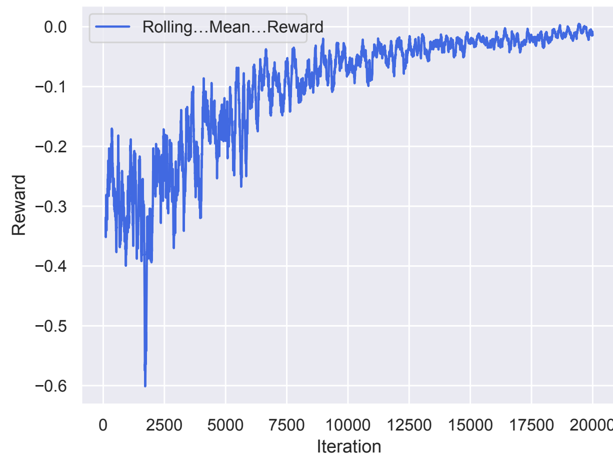


Figure 3.8: Reward collected by the Data Collector agent while training.

As illustrated in Fig. 3.8, as more iterations are passed and the value of ϵ 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 accuracy. Note that according to Eq. (3.7), 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.

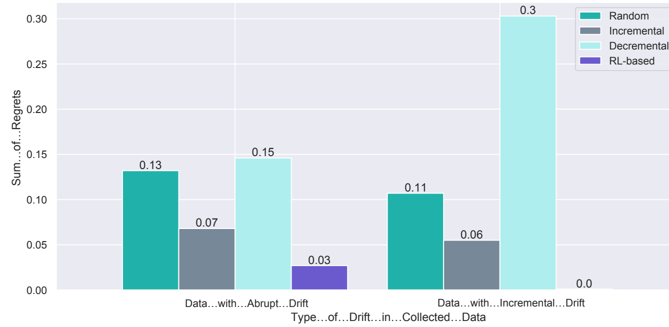
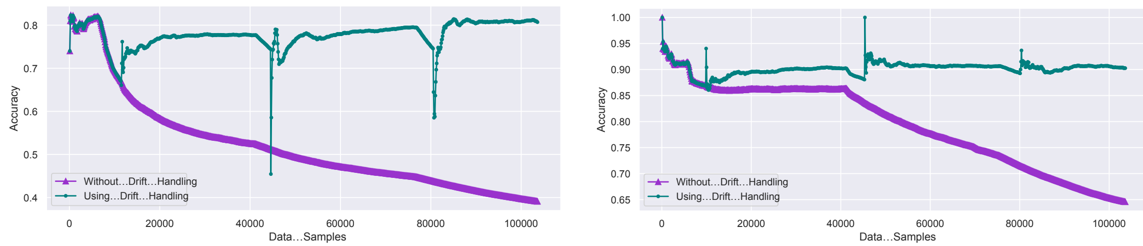


Figure 3.9: Regret evaluation of the Data Collector in the presence of abrupt and incremental drifts.

Fig. 3.9 illustrates the sum of the regret terms for both drift types. As illustrated in Fig. 3.9, 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$\times$ 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.

D) Accuracy of the Proposed Solution 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 adaptation solution and compare it to a system without any drift adaptation entity. We evaluated our solution on the last twelve days worth 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.



(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.

Figure 3.10: Accuracy over time of fault prediction models.

Fig. 3.10 depicts the persistent accuracy of our proposed solution over twelve days while three abrupt drifts occur. As illustrated in Fig. 3.10a, 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 TL, 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 TL 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 predictions become available, the

true accuracy of the model can be calculated. Similarly, Fig. 3.10b illustrates the accuracy of the CPU fault prediction model in the presence of three abrupt drifts. The Drift Adaptation Method Selector agent selects TL, 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. 3.10b). In summary, Fig. 3.10 shows the effectiveness of our proposed solution in maintaining a persistent accuracy of up to 40% higher compared to a system without any drift adaptation entity during our twelve-day experiment while satisfying the requirements of the operator.

E) Comparative Results

Since there is no similar solution in the literature that automates concept drift adaptation, we compare the end-to-end performance of our solution 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 3.5: End-to-end comparison of proposed solution 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

Table 3.5 presents the end-to-end performance comparison of our solution with several approaches. As shown in Table 3.5, 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 solution 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.

3.6 Conclusion

In this chapter, we proposed a concept drift adaptation algorithm for fault prediction in cloud environments using RL. This algorithm considers the 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 as well as data size for adaptation that fulfills the operator's requirements. To demonstrate the effectiveness of our proposed solution, we experimentally validated a proof-of-concept on a Kubernetes-based testbed. Our proposed Drift Adaptation Method Selector and Data Collector trained agents presented up to $13\times$ and $30\times$ less regret, respectively, against a human expert as compared to other algorithms. Moreover, our proposed solution achieved up to a 40% higher accuracy compared to a system without drift adaptation, 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.

Chapter 4

Feature Drift Adaptation Algorithm for ML-based Fault Prediction [1]

4.1 Introduction

Maintaining the performance of an ML model that can predict application performance degradation caused by infrastructure faults in cloud environments is challenging. One challenge is the occurrence of feature drifts leading to the model inaccurate predictions. To effectively adapt prediction models to feature drifts, this chapter proposes a feature drift adaptation solution for cloud environments. This solution detects feature drifts based on the changes in the performance of the prediction model as well as the changes in the importance of the features used for training the model. The solution assesses the severity of the feature drift to decide how to adapt the features as well as the model to the drift. In case of non-severe drifts, it eliminates the drifting features and fine-tunes the model using TL to adapt to feature drift; otherwise, for the severe feature drifts, the solution performs a feature re-selection and a model re-training to adapt to feature drifts.

The rest of this chapter is organized as follows. First, we present an illustrative use case followed

[1] This chapter is based on one published and one submitted paper: [3] B. Shayesteh, C. Fu, A. Ebrahimzadeh, R. Glitho, "Causal-Temporal Analysis-Based Feature Selection for Predicting Application Performance Degradation in Edge Clouds," In *Proc. IEEE International Conference on Communications (ICC)*, May 2023, pp. 5496-5501, and [4] B. Shayesteh, C. Fu, A. Ebrahimzadeh, R. Glitho, "Adaptive Feature Selection for Predicting Application Performance Degradation in Edge Cloud Environments," Submitted to *IEEE Transactions on Network and Service Management*, under review.

by system model and our proposed feature drift adaptation algorithm. Next, we describe the lab setup and the implementation results and conclude this chapter.

4.2 Illustrative Use Case

In this section, we describe an illustrative use case that highlights the importance of feature drift adaptation solutions for ML models. Let us consider an online shopping web application comprising multiple microservices deployed in a cloud environment consisting of a central cloud and an edge to guarantee low latency and high availability for its users. An ML model is utilized to predict potential performance degradation of the application, enabling preemptive actions to avoid such degradation and ensure that a consistent QoS is delivered to end users. The application is declared to experience a performance degradation if its KPI, i.e., response time, surpasses a given threshold, e.g., the expected response time from the application. The prediction model is trained using infrastructure-level metrics relevant to the underlying reason of application performance degradation. It is important to note that training the ML model on all features associated with every possible underlying fault type that causes performance degradation introduces a significant amount of noisy input, leading to inaccurate prediction models. Therefore, the model is specifically trained on features related to the common faults that may cause performance degradation in the online shopping application, such as CPU over-utilization or network congestion during high-traffic periods like special sales. Features such as the CPU load of VMs, and received/transmitted packets, are used to predict potential performance degradations, facilitating preemptive actions like resource scaling or load balancing.

Maintaining the performance of the ML model in highly dynamic cloud environments poses a significant challenge. A particular challenge is the phenomenon of feature drift, where some or all of the features used for training the model become irrelevant to application performance degradation over time due to environment changes. This irrelevance degrades the ML model's performance. For instance, in addition to the common faults, due to some issues causing excessive logging, the system may experience HDD over-utilization fault, which also has impact on the application performance. The initial prediction model would suffer from inaccurate predictions since the features it was trained on do not fully represent all conditions and can partially lose their relevance, while

HDD-related features will become relevant to the learning task. In another scenario, in the considered application, resource optimization techniques can be incorporated in the cloud environment to effectively adjust the resources according to the load, which can avert CPU over-utilization faults and deprecate the relevancy of CPU-related features used for training the ML model. In both cases, a feature drift adaptation solution is required to continuously monitor the performance of the model and relevance of the features to adapt the features and the model to feature drift.

4.3 System Model

Fig. 4.1 presents a high-level view of a cloud environment consisting of one central cloud and one edge, as a representative of many. As illustrated in Fig. 4.1, an application is deployed in this environment, and the end users of this application can send requests to use the functionalities provided by the application. We consider a time series forecasting model deployed in this environment, which predicts possible performance degradation of the deployed application ahead of time. Application performance degradation can be manifested through various application KPIs, e.g., response time. The underlying cause of application performance degradation are diverse and can include infrastructure faults such as CPU or HDD over-utilization. The edge site is equipped with an Edge Site Monitor, which collects raw training data, i.e., infrastructure-level and application-level performance metrics. Infrastructure-level performance metrics can include resource utilization such as CPU, memory, disk, and network metrics, while application-level performance metrics can include various application KPIs such as response time or throughout. The raw training data is passed to the ML pipeline in the cloud site, where the Data Pre-processing cleans the data from inconsistencies (e.g., outliers, null values, etc.), and normalizes the data. The normalization is necessary to ensure all features used for training the model are within the same scale. Each feature is normalized separately using its mean and standard deviation. Next, the pre-processed data is used by Feature Selection to choose the features, i.e., infrastructure-level and application-level performance metrics that have the highest correlation with the occurrence of degradation in the application KPI. The feature space can consist of thousands of features considering the diversity of performance metrics of various resources in the edge site. The selected features of the pre-processed data are further used to

train the application performance degradation models in an offline mode, and then the trained model is deployed in the edge site for prediction. The Edge Site Monitor collects online data and feeds them to the Online Prediction component to perform the same pre-processing procedure. The Online Prediction executes the prediction model to produce KPI predictions. Due to the high diversity of underlying causes of an application performance degradation deployed in cloud, the relevancy between the selected features used for training the prediction model and application KPI is subject to changes over time (i.e., feature drift). The Feature Drift Adaptation tackles the problem of feature drift by receiving the features used for training the prediction model along with the prediction status as inputs, and updates the feature set and adapts the prediction model to the feature drift. The prediction status indicates whether the model's prediction about application performance degradation was correct or incorrect. The prediction model predicts the value of the application kpi. For example for response time, a performance degradation is identified if the kpi value exceeds a predefined threshold that represents the application's expected performance level.

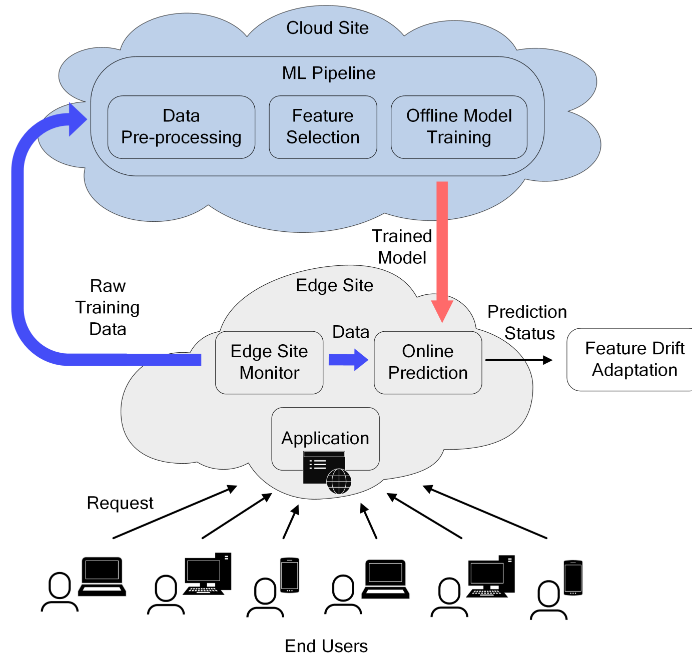


Figure 4.1: Generic system model of an ML model predicting application performance degradation by monitoring application KPIs in a cloud environment.

In the setting described above, let the data that will be used for training the prediction model be $S = \{(x_1, y_1), (x_2, y_2)..., (x_n, y_n)\}$, where n is the total number of data samples. Each x_i (i

ranging from 1 to n) is a vector of features belonging to feature set D with size $|D|$, and $y_i \in \{C_{degrade}, C_{no-degrade}\}$ is the class label denoting whether application performance will degrade or not for vector x_i. $C_{degrade}$ indicates the class label for occurrence of application performance degradation, while $C_{no-degrade}$ indicates the class label for normal application performance. Let the set of features selected for training a prediction model at time t_j be $D^*_{t_j} \subseteq D$, and the prediction model $F_{t_j} : x_i \rightarrow y_i$ be trained to predict performance degradation using data characterized by feature set $D^*_{t_j}$. The model F_{t_j} can predict performance degradation of a given vector x_i as $\hat{y}_i = F_{t_j}(x_i)$ with the accuracy of A_{t_j} percent. A feature drift can occur at any time instance t_j, where $t_k = t_j + \Delta$, when $D^*_{t_j} \neq D^*_{t_k}$ [11], which means that the set of most relevant features that are selected are not equal in time the reference time t_j and t_k. To define feature drift for each feature $d_i \in D$, let $r(d_i, t_j) \in 0, 1$ define the relevancy of feature d_i in time t_j, with $r(d_i, t_j) = 1$ stating that $d_i \in D^*_{t_j}$ and $r(d_i, t_j) = 0$ stating $d_i \notin D^*_{t_j}$, then feature d_i drifts between time t_j and t_k as follows [11]:

$$\exists t_j \exists t_k, t_j < t_k, r(d_i, t_j) \neq r(d_i, t_k). \tag{4.1}$$

The problem of feature drift adaptation can include detecting feature drift, i.e., finding the time t_k when the feature drift occurs, selecting the new set of features $D^*_{t_k} \subseteq D$, and adapting the prediction model to $F_{t_k}(x_i)$ so that the post-adaptation accuracy of the prediction model $A_{t_k} \approx A_{t_j}$.

We aim to solve the problem of feature drift adaptation while predicting application performance degradation in cloud by detecting a feature drift, and updating the features and adapting the prediction model to the drift to maintain the prediction model's performance.

4.4 Proposed Feature Drift Adaptation Algorithm

In this section, we present our proposed feature drift adaptation algorithm and its main components. Fig. 4.2 illustrates the architecture of our proposed solution. A deep learning time series forecasting model for predicting application performance degradation is trained offline using training data characterized with an initial set of features selected offline, i.e., a selected set of cloud

performance metrics, using techniques described in [3]. The prediction model receives the pre-processed online data consisting of the selected features as its input and makes a prediction about future status of the application, i.e., whether the performance of the application will degrade or not. The Feature Drift Detector component receives the online data and prediction model as inputs and detects the occurrence time of a feature drift and generates a feature drift alert. The feature drifts are detected with the help of two components, i.e., Model Performance Analysis and Feature Importance Analysis. In Model Performance Analysis, the prediction status of the prediction model is received as an input, and in case of consistent drop in the model performance, the output, a performance drop notification, is sent to the Feature Importance Analysis to further analyze the changes in feature importance of the data used for inferencing. A feature drift alert is generated if both the prediction model's performance and the importance of features used for prediction have degraded. Upon receiving a feature drift alert, the Feature Drift Adaptor component updates the set of features that are relevant to the application performance degradation after drift and adapts the model to the feature drift. The Feature Drift Adaptor consists of two components, i.e., Drift Severity Analysis and Adaptation Method Selection. Upon receiving a feature drift detection alert, the Drift Severity Analysis component finds the severity of the detected feature drift, and sends its finding to the following component, i.e., Adaptation Method Selection, to select the appropriate approach for updating the set of relevant features as well as the proper method for adapting the prediction model to the feature drift. Based on this decision, the Feature Drift Adaptor updates the features and adapts the prediction model. By focusing on the indicators of feature drift such as change in model performance and feature importance rather than specific scenarios causing the feature drift, e.g., change in fault type, the proposed solution can be applicable to detect feature drifts and adapt to them in various settings.

Algorithm 4 illustrates the different steps of our proposed feature drift adaptation solution. The inputs of the algorithm are the set of training data ($\{(X_{train}, Y_{train})\} \in S$) that will be used to train the application performance degradation prediction model, list of selected features (D^*), and amount of data (L) that will be used for feature drift adaptation. Algorithm 4 starts by loading the performance degradation prediction model (F) that was trained offline using $\{(X_{train}, Y_{train})\}$. When the new online data (X) with prediction result (Y) arrives, the data along with the prediction

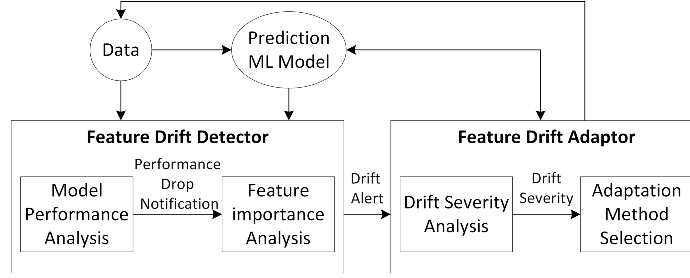


Figure 4.2: Architecture of the proposed feature drift adaptation solution.

model (F) are passed to the drift detector function to detect possible drifts. The feature drift detector returns a binary value indicating occurrence of a drift as well as the list of feature importance (FI) after the drift, if applicable (see lines 2-5). The detailed description on how feature drift detector works is provided in Section 4.4.1 and Algorithm 5. In case of feature drift detection, the feature drift adaptor is initialized by receiving the feature importance (FI) after drift as input and returns the proper approach to update the selected features (i.e., feature_selector), and the proper approach to adapt the prediction model to the drift (i.e., drift_adaptor) (see lines 6-7). The detailed description on how feature drift adaptor works is provided in Section 4.4.2 and Algorithm 6. A pre-defined amount of data (L) is collected to perform feature update as well as prediction model adaptation. Following the adaptation, the set of selected features and prediction model are replaced with the new set of features and adapted prediction model, respectively (see lines 8-11). The process of feature drift adaptation, i.e., detecting feature drift and adapting to the feature drift, continues as long as online data is received.

Algorithm 4 Feature Drift Adaptation

1: **Inputs**: Training data $(X_{train}, Y_{train}) \in S$, Set of selected features (D^*), Amount of data required for adaptation L
2: $F \leftarrow$ Load trained prediction model
3: **while** data arrives **do**
4: $(X, Y) \leftarrow$ Receive data with features from D^*
5: $is_drift, FI \leftarrow$ drift_detector(F, X, Y)
6: **if** is_drift **then**
7: model_adapt, feature_selector $\leftarrow$ drift_adaptor(FI)
8: $X_{adapt}, Y_{adapt} \leftarrow$ collect data of size L
9: $D^* \leftarrow$ feature_selector(X_{adapt}, Y_{adapt})
10: $new_model \leftarrow$ model_adapt($F, X_{adapt}, Y_{adapt}, D^*$)
11: $F \leftarrow new_model$
12: **end if**
13: **end while**

4.4.1 Feature Drift Detector

This component is responsible for detecting the feature drifts while an ML model is predicting application performance degradation. As illustrated in Fig. 4.2, Feature Drift Detector consists of two main components, i.e., Model Performance Analysis and Feature Importance Analysis.

Feature drift occurs when the features used for training the prediction model cease to be relevant to the learning task. Therefore, one of the main indicators of feature drift occurrence is a drop in the performance of the prediction model. The responsibility of the Model Performance Analysis component is to raise an alert if there is noticeable constant drop in the performance of the prediction model. Therefore, this component needs to monitor the prediction model's error to detect changes. Error-based concept drift detection techniques (reviewed in [14]) including CUSUM [63] and ADWIN [48] monitor the prediction's model error to detect drifts. Our previous experiments for concept drift detection in [1] showed that CUSUM had a better performance in terms of True Positive ratio and False Positive ratio compared to other error-based concept drift detections. CUSUM monitors the mean of the model's error and alarms a drift when significant deviation is observed in the mean error value.

Although the Model Performance Analysis component can detect degradation in model's performance, which is one of the main indications of the feature drift, there could be other reasons why model's performance is degrading, such as concept drift, i.e., changes in distribution of data. Therefore, to detect feature drifts it is important to also monitor the changes in the relevancy of the feature to the learning task. In the Feature Drift Detector component, we propose using Feature Importance Analysis to measure the effect and contribution of each feature to the performance of the prediction model. We use a lightweight perturbation-based method, where an importance score is assigned to each feature based on the difference in model's prediction when that particular feature is replaced with random samples drawn from the feature's distribution of data [79]. A drop in features' importance score is an indicator that the performance drop notification raised by the Model Performance Analysis component is due to changes in the feature relevancy. Therefore, if the changes in feature importance after occurrence of a performance drop is greater than a given threshold α, the Drift Detector can detect that a feature drift has occurred. The reason for adopting this feature importance

analysis method instead of only monitoring feature relevance over time is that perturbation-based feature importance monitors the direct impact of each feature on model performance. Therefore, if a previously relevant feature is experiencing a decrease in its importance score, it does not only show a decline in relevance to the target KPI, but it also manifests an impact on the model performance. Algorithm 5 illustrates the steps of our proposed Feature Drift Detector component.

Algorithm 5 Feature Drift Detector

```
1: Inputs: Prediction model (F), Online data ((X, Y)), List of selected features (D*), Threshold indicating feature importance change (α)
2: performance_drop ← False
3: FI_base ← load initial feature importance of each feature in the training data calculated offline using lines 9-16 of this algorithm
4: FI ← []
5: Ŷ ← F(X)
6: pred_status ← get_prediction_status(Y, Ŷ)
7: performance_drop ← CUSUM(pred_status)
8: if performance_drop then
9:     base_error ← calculate_pred_error(Y, Ŷ)
10:    for (i = 0 to |D*|) do
11:        X' ← make a copy of X
12:        X'[i] ← draw random samples from P(X'[i])
13:        Ŷ' ← F(X')
14:        error ← calculate_pred_error(Y, Ŷ')
15:        FI[i] ← error − base_error
16:    end for
17:    for (i = 0 to |D*|) do
18:        if FI_base[i] − FI[i] ≥ α then
19:            return True, FI
20:        end if
21:    end for
22: end if
23: return False, FI
```

4.4.2 Feature Drift Adaptor

This component is responsible for updating the set of selected features after feature drift occurrence and adapt the prediction model to the feature drifts. As illustrated in Fig. 4.2, Feature Drift Adaptor consists of two main components, i.e., Drift Severity Analysis and Adaptation Method Selection.

To make a decision on how to update the set of features used for training the ML model (e.g., re-use a subset of features or re-select new features), and how to adapt the prediction model to the feature drift (e.g., fine-tune the model or re-train a model from scratch), we are required to know the severity of the feature drift. In severe feature drifts, the majority of the features with high contribution to the prediction model's performance become irrelevant to the prediction task,

resulting in significant degradation in performance of the prediction model. On the other hand, in non-severe feature drifts, the features that do not have high contribution to the prediction model's performance or a minority of the high contributing features become irrelevant to the prediction task. In this case, the degradation in performance of the model is not significant since the majority of high contributing features are still relevant to the prediction task. The degree of contribution of each feature is determined using feature importance scores. The Drift Severity Analysis receives the feature importance scores of all features before and after drift as input and it calculates a binary value indicating whether the feature drift was severe or not as an output. The severity analysis is performed by considering which feature has experienced a decrease in feature importance. If the majority of the features with the high contribution to model's performance have lost their importance, the drift is considered a severe feature drift. Otherwise, the drift is considered as a non-severe feature drift. Threshold β is used to determine if the feature has a great or small contribution to the model's performance.

The Adaptation Method Selection component decides on how to update the feature set and adapt the prediction model while receiving the decision of the Drift Severity Analysis component as an input. In case of severe feature drift, the adaptation solution is to update the features by re-selecting the entire feature set and adapt the prediction model by re-training a new model from scratch using the re-selected feature set. Otherwise, for the non-severe feature drifts, where the majority of features with high contribution to the learning task before the drift are still relevant to the learning task after the drift, the decision is to eliminate the drifting features and fine-tune the prediction model using TL [40]. TL uses a pre-trained model as a starting point and fine-tunes it on a new task with new data. Fine-tuning involves updating the weights of a pre-trained model during training with new data. This can be done by either updating the weights of all the layers or by freezing the weights in certain layer(s) of the model so that they are not updated during the training, while updating the weights of the rest of layer(s) [1]. Therefore, to adapt the prediction model to non-severe feature drifts, first the drifting features are eliminated from the feature set. Next, the input shape of the first layer of the prediction model is adjusted according to the new size of the feature set. Finally, the model is fine-tuned using the data collected after drift occurrence. Algorithm 6 illustrates the steps of our proposed Feature Drift Adaptor component.

Algorithm 6 Feature Drift Adaptor

1: **Inputs**: List of selected features before drift (D^*), Prediction model before drift (F), Feature importance of each feature after drift (FI), Amount of data required for adaptation (L), Threshold indicating feature importance change (α), Threshold indicating features with high contribution (β)
2: $FI_{base} \leftarrow$ load initial feature importance of each feature in the training data calculated offline using lines 9-16 of Algorithm 5
3: $is_severe \leftarrow$ False
4: $drifting_features \leftarrow$ []
5: $high_contrib_features \leftarrow$ []
6: $counter = 0$
7: **for** ($i = 0$ to $|D^*|$) **do**
8:     **if** $FI_{base}[i] - FI[i] \geq \alpha$ **then**
9:         $drifting_features$.append(i)
10:     **end if**
11:     **if** $FI_{base}[i] \geq \beta$ **then**
12:         $high_contrib_features$.append(i)
13:         **if** i in $drifting_features$ **then**
14:             $counter = counter + 1$
15:         **end if**
16:     **end if**
17: **end for**
18: **if** $counter \geq$ size($high_contrib_features$) / 2 **then**
19:     $is_severe \leftarrow$ True
20: **end if**
21: $(X_{adapt}, Y_{adapt}) \leftarrow$ collect new data of size L
22: **if** is_severe **then**
23:     $D^* \leftarrow$ select_features(X_{adapt}, Y_{adapt})
24:     $F \leftarrow$ retrain($X_{adapt}, Y_{adapt}, D^*$)
25: **else**
26:     $D^* \leftarrow$ drop_features($drifting_features$)
27:     $F \leftarrow$ fine_tune($F, X_{adapt}, Y_{adapt}, D^*$)
28: **end if**
29: **return** D^*, F

4.4.3 Complexity Analysis

In the following, we present the complexity analysis of our proposed feature drift adaptation, Feature Drift Detector, and Feature Drift Adaptor algorithms. The time complexity of the Feature Drift Detector (shown in Algorithm 5) is $\mathcal{O}(|D^*|)$, where $|D^*|$ is the size of the selected features that the algorithm iterates over to calculate feature importance. Similarly, the time complexity of the Feature Drift Adaptor (shown in Algorithm 6) is $\mathcal{O}(|D^*|)$ since the algorithm iterates over all features to identify drift severity. As for the complexity of feature drift adaptation (shown in Algorithm 4), assuming that the arriving data includes W samples, which are needed to be processed for detection and adaptation purposes, the time complexity is $\mathcal{O}(W \cdot |D^*|)$, since feature drift detection and adaptation are performed periodically on the arriving data.

4.5 Performance Evaluation

In this section, we evaluate the performance of our proposed feature drift adaptation solution in a real-world testbed. We deployed a microservice-based web application on Kubernetes and trained a prediction model to predict the performance degradation of the application while automatically adapt to feature drifts. In the following, we present the experiment settings and evaluation results.

4.5.1 Experiment Settings

We describe the details of our lab setup, fault injection specification, and datasets of the experiments in the following sub-sections.

A) Lab Setup: Figure 4.3 depicts our lab setup, which consists of three Kubernetes clusters with a total of 10 VMs running *Ubuntu 18.04*. The clusters are deployed in a private cloud managed by the Infrastructure-as-a-Service OpenStack. In each cluster, one of the VMs is the master node while the other VMs in the cluster are worker nodes. The configurations of the VMs are summarized in Table 3.1. We deployed an open-source microservice-based online shopping web application called *Sock Shop* [80] in the edge site clusters. The performance degradation is manifested in the application KPI, i.e., the amount of time in seconds spent serving HTTP requests. Moreover, we deployed a *Locust* [81] load generator in the cloud site to impose a constant load of 100 requests per second on the applications deployed in both edge sites. We deployed *Prometheus* [71] in the edge sites to monitor the clusters and to collect data from edge sites every 10 seconds to reflect the latest changes in the metrics. We collected the node- and container-level metrics (i.e., VM and container metrics such as the CPU, memory, and network metrics), as well as application-level metrics (i.e., Sock Shop application performance metrics).

B) Fault Injection Specifications: We initiated node-level CPU over-utilization, HDD over-utilization, and packet loss faults, which were injected to the VM hosting the Sock Shop application. The node-level CPU over-utilization and HDD over-utilization faults were injected using the *Stress-ng* [72] tool. The packet loss fault was injected using *Linux Traffic Control* [82], which allows us to drop a certain portion of the packets (e.g., 10%). The fault injections are recurrent, meaning

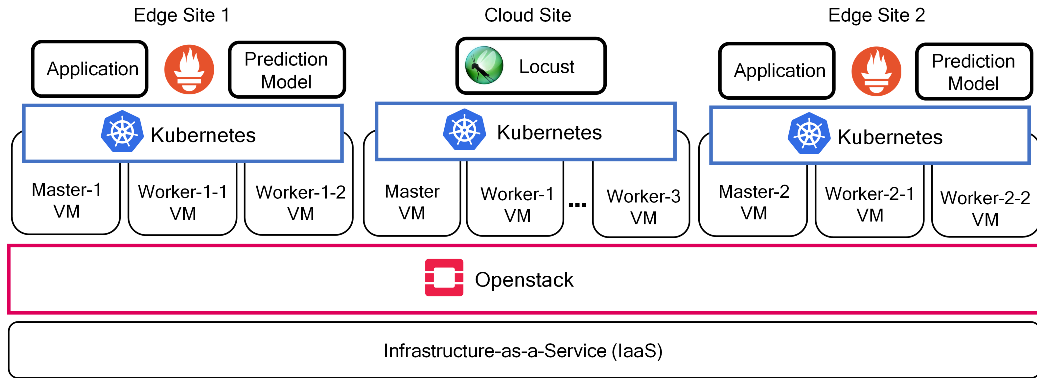


Figure 4.3: Lab setup for evaluating the proposed feature drift adaptation algorithm.

that the injections are recurrently interrupted with a cool-down time [62]. The duration and inter-arrival of faults follows a Normal distribution and exponentiated Weibull distribution, respectively, as described in Section 3.5.1.

C) Datasets: We have examined multiple scenarios that often occur in cloud systems that may cause feature drifts including changes in software and hardware configuration, load modification, scaling the services, migrating the fault injection target, and changing the underlying fault type. Among these scenarios, migrating the fault injection target and changing the underlying fault type caused feature drifts. Therefore, we considered these two scenarios to collect datasets that include feature drift. In the first scenario, we started the experiment by injecting node-level CPU over-utilization fault to the VM that hosts the microservice whose response time was the KPI of interest. To cause a feature drift, we migrated that specific microservice to another VM and also changed the node-level CPU over-utilization fault injection target to the migration destination VM. Since the application VM is the response time of that particular microservice, the migration scenario described above can result in the metrics of the migration destination VM become more relevant to the KPI degradation after migration compared to the metrics of the source VM. We refer to this dataset as the migration dataset. The duration of the migration dataset is 40 hours. In the second scenario, we changed the type of underlying infrastructure-level fault that caused an application performance degradation. Specifically, we collected two separate datasets in this scenario. For the first dataset, we started the experiment by injecting node-level CPU over-utilization fault to the VM that hosts the microservice whose response time was the KPI of interest. Toward the middle

of the experiment, we changed the injected fault type to packet loss on the same VM. Similarly, for the second dataset, we started by injecting HDD over-utilization fault to the VM hosting the microservice of interest, and we changed the injected fault type to CPU over-utilization on the same VM. The change in the underlying fault type can also cause a feature drift since metrics relevant to application performance degradation can change depending on the underlying fault type. We refer to these datasets as Changed-Fault-Type-1 (CFT-1) and CFT-2 datasets. The duration of the CFT-1 and CFT-2 datasets is 54 hours.

We collected all datasets from Prometheus. Each timeseries metric scraped by Prometheus is uniquely identified by the name of the metric and a series of tags [83]. The metric's name indicates the function of that metric. For instance, *http_request_duration_seconds_count* is the total number of HTTP requests. The series of tags associated with this metric can clarify more details about the metric, e.g., the tag *name=front-end* can specify the total number of HTTP requests reported by this metric is made to the front-end microservice of the application. Similarly, Prometheus collects many other metrics with the same name but different tags, which makes each metric unique. The metrics collected by Prometheus composed the feature space of the collected datasets for evaluations.

4.5.2 Evaluation Results

In this section, we first evaluate the performance of feature selection and prediction models that predict application performance degradation in terms of MAE, F1-score, model training time, and feature selection time. Next, to evaluate our proposed feature drift adaptation solution, we first evaluate the performance of the Feature Drift Detector component in terms of drift detection delay and accuracy and then evaluate the Feature Drift Adaptor component in terms of post-adaptation F1-score and MAE. Finally, we compare our proposed solution to four benchmarks, namely, DCFS [47], Modified DCFS [47], retraining the model with new data and without feature re-selection (a common solution for concept drift adaptation), and no feature drift adaptation.

A) Feature Selection and Prediction Results

The purpose of this experiment is to present the features selected for training the prediction models and the performance of the models trained using these features. In [3], we proposed a

causal-temporal analysis feature selection system, consisted of two phases, namely, similarity analysis and causal-temporal analysis. In the similarity analysis phase, the feature space is reduced by evaluating the correlation between infrastructure-level performance metrics and application KPI. Subsequently, in the causal-temporal analysis phase, causal-temporal relationships between the application KPI and the most correlated infrastructure metrics from the initial phase is determined. Only those features that have a causal relationship with the application KPI are selected for training models for predicting application performance degradation. Therefore, the number of features used for training the prediction model is not predetermined, but rather it depends on the number of infrastructure-level metrics that have a causal relationship with application KPI as identified by the feature selection system. In this experiment, we explored the performance of an approach that previously showed promising results in [3], i.e., Pearson Correlation Coefficient combined with Time-Lagged Cross Correlation (TLCC) (Pearson + TLCC). The Pearson Correlation Coefficient identifies the infrastructure-level metrics most significantly correlated to the application KPI. The subsequent use of TLCC helps in finding the causal relationships between these metrics and the application KPI to selected features for training the prediction model.

For training the prediction model, we trained hybrid CNN-LSTM prediction models using the selected features for each dataset to predict application performance degradation. The architecture of the CNN-LSTM model consists of a sequence of one Convolution, one Pooling, two LSTMs, and one Dense layer(s). The prediction models are multi-variate multi-step time series prediction models, where the model's inputs are the data with selected features and the prediction output is the value of the application KPI (i.e., response time). The datasets are chronologically divided into training and testing sets at a ratio of 80% and 20%, respectively, ensuring that the training data precedes the testing data to prevent any potential data leakage. In the migration and CFT datasets, the portion of data used for training the prediction models does not include any feature drifts. The hyperparameters of our prediction models were optimized using TPE [76], with the same search space described in Section 3.5.2.

The evaluation metrics for this experiment are MAE and F1-score of the prediction model, model training time, and the feature selection time to show the duration of resource occupation for feature selection. MAE is the difference of the actual and predicted values of the KPI. To

calculate the F1-score of the prediction model, the KPI values, i.e., response times, above 1 second are assigned a class label of performance degradation, while response times under 1 second are assigned the normal performance class label. The 1 second threshold is set considering the response time of the application when it receives a normal load of 100 requests per second. Tables 4.1 present the features size and feature selection time of Pearson + TLCC approach, as well as the MAE, F1-score, and training time of a model trained using these features for all considered datasets.

Table 4.1: Evaluation of the feature selection approach and prediction model using the considered datasets.

Dataset	**Feature Size**	**F1score (%)**	**MAE**	**Training Time (s)**	**Feature Selection Time (s)**
Migration	4	81.2	0.231	44.3	90.5
CFT-1	4	77.49	0.491	57.1	112.1
CFT-2	8	81.17	0.231	39.6	136.7

According to Table 4.1, in the migration dataset, the F1-score of the prediction model trained using the features selected by Pearson + TLCC feature selection approach is 81.2% and its MAE is 0.231. The migration datasets consists of 2890 features, which the feature selection approach spends 81.2 seconds to select the most relevant features from. Pearson + TLCC selects the following features of the VM experiencing the CPU over-utilization fault for training the prediction model: *node_cpu_seconds_total*: the average amount of CPU time spent in user mode, which was selected for two different CPU cores (core 2 and 3), *node_schedstat_runinng_seconds_total*: number of seconds CPU spent running a process, *node_schedstat_waiting_seconds_total*: number of seconds a process spent waiting for this CPU.

Similarly, for the CFT-1 dataset, top 4 features from total of 3057 features are selected in 57.1 seconds. The prediction model trained using these features achieves an F1-score of 77.49% and MAE of 0.491. The list of features selected by Pearson + TLCC for training the prediction model is as follows (all features belong to the VM experiencing CPU over-utilization fault): *node_netstat_Icmp_OutMsgs*: number of sent Internet Control Message Protocol (ICMP) messages, *container_cpu_usage_seconds_total*: amount of CPU time, *node_memory_Slab_bytes*: the amount of slab memory allocated in bytes, *node_memory_SUnreclaim_bytes*: the amount of unreclaimable

memory allocated in bytes.

The CFT-2 dataset consists of 3257 dataset, and the Pearson + TLCC select the top 8 relevant features for training the prediction model with F1-score of 81.17% and MAE of 0.231. Pearson + TLCC selects the following features for training the prediction model. Note that all features are from the VM experiencing HDD over-utilization fault: *node_schedstat_waiting_seconds_total*: number of seconds a process spent waiting for one CPU core, *node_cpu_seconds_total*: the average amount of CPU time spent in system mode for one CPU core, *node_disk_io_time_seconds_total*: number of seconds I/O operations have been in progress on a virtual disk, *node_disk_writes_completed_total*: number of write operations completed successfully on a virtual disk, *node_disk_written_bytes_total*: number of bytes written to a virtual disk, *node_vmstat_pgpgout*: number of pages that have been paged out from the memory, *node_memory_Cached_bytes*: number of bytes used in the memory used for caching purposes, *node_memory_Dirty_bytes*: number of bytes in the memory used to store modified data that has not been written to the disk.

It can be observed from the list of the features that although the type of underlying fault causing application performance degradation is the same in the migration and CFT-1 datasets, i.e., CPU over-utilization fault, the features selected in the two datasets are different. This is due to the fact that these datasets are collected from different clusters with different settings (e.g., Kubernetes version). Moreover, the monitored application KPIs indicating application performance degradation belong to different services in the two datasets. In the migration dataset, since we migrated the *Catalogue* service of the application to another VM, the KPI that we monitored is the response time of the *Catalogue* service of the application. However, in the CFT-1 and CFT-2 datasets, the KPI that we monitored is the response time of the *Front-end* service, which is the main service that receives all the requests. It can also be observed that in the CFT-2 dataset, in addition to the disk-related features, some features related to CPU are also selected. This is due to the fact that stressing the disk can also affect the CPU, e.g., by increasing wait times for data to be read or written to the disk.

B) Feature Drift Detection Results

In this experiment, we evaluate the performance of the proposed feature drift detector in terms of feature drift detection delay and accuracy of drift detection. In the migration dataset, there is

one feature drift that occurs once we migrated an specific microservice (i.e., catalogue service) to another VM and also changed the node-level CPU over-utilization fault injection target to the migration destination VM. For feature drift detection purpose in the migration dataset, we consider the last 27 hours out of 40 hours worth of data, which consists of the testing data used for evaluating the prediction model and the data after the occurrence of feature drift. The feature drift occurs around the data sample 1200 of the dataset. The model performance analysis component periodically calculates the performance of the prediction model in terms of F1-score after receiving predictions on 100 data samples, and passes the performance to CUSUM to detect consistent performance drops. CUSUM detects a consistent drop in the prediction model's performance around the data sample 1600, which means that it had received 16 inputs (predictions) when a drift was detected. Once a performance drop is detected, to detect feature drift, the Feature Importance Analysis component finds if the importance of any features have dropped compared to its initial importance. It is important for the Feature Importance Analysis component not to be too sensitive to normal fluctuations in feature importance scores. Therefore, threshold α is set considering the fluctuations of the feature importance on the historical data used for training the prediction model. To set the threshold, importance score of each feature is calculated repeatedly using the subsets of the training data. The threshold is set equal to the standard deviation of the calculated feature importance scores for each feature, which shows the range of typical changes feature importance scores. Although it is possible to set a single threshold for all features, e.g., average of standard deviations among all features, if the threshold is smaller than normal changes of feature importance for a given feature, it can increase number of false feature drift detection. On the other hand, if the threshold is too large, the feature drift detector would miss the performance drops due to feature drifts. Similarly, the initial importance for each feature is set to average among the calculated feature importance scores for each feature in the subsets of the training data. In this experiment, to set the threshold, we calculated the feature importance scores offline for every 100 data sample in the training data, which shows 0.06, 0.06, 0.02, and 0.05 standard deviation for each feature, respectively. The amount of feature importance drop at the time CUSUM detected a performance drop is 0.19, 0.12, 0.06, and 0.18 for each feature, respectively. Therefore, for all features, the feature importance drop is above the determined thresholds, which confirms that there is a change in feature importance and a feature

drift has been detected. The amount of time spent performing feature importance analysis is 0.17 seconds. Knowing the ground truth that the drift occurrence time is after receiving predictions on 1200 data samples, the drift detector has a delay of 4 updates or 400 data samples. Considering that the time spent for performing feature importance analysis is short (0.17 seconds) compared to CUSUM , we report the feature drift detection delay only considering the delay of CUSUM.

Similarly, for the CFT-1 dataset, one feature drift occurs when the underlying fault type is changed from node-level CPU over-utilization fault to packet loss fault. We analyzed the last 30 hours out of 54 hours worth of data for detecting feature drifts. The feature drift occurs around data sample 4000. The CUSUM detects a consistent drop in performance of the prediction model occurs around data sample 4700, which means 47 predictions have been made by the model. The Feature Importance Analysis component finds feature importance drops of 0.16, 0.24, 0.38, and 0.31 for each feature, respectively. Threshold α with values 0.08, 0.1, 0.08, and 0.07 were set using historical training data, which confirms occurrence of a feature drift since the feature importance drop is greater than the determined thresholds for all features. The amount of time spent performing feature importance analysis is 0.15 seconds. Consulting the ground truth of drift occurrence time, i.e., drift occurrence after receiving predictions on 4000 data samples, the drift detector has a delay of 7 updates or 700 data samples.

Similar to CFT-1 dataset, in CFT-2 one feature drift occurs due to changing the underlying fault type from HDD over-utilization to CPU over-utilization. The last 30 hours out of 54 hours worth of data is analyzed for feature drift detection purposes. The CUSUM detects a consistent drop in performance of the prediction model around data sample 4400, while the feature drift occurs around sample 4000. Therefore, feature drift detector has a delay of 4 updates or 400 data samples. Upon feature drift detection, the Feature Importance Analysis component finds feature importance decrease of 0.01, 0.004, 0.02, 0.03, 0.06, 0.04, 0.003, and 0.18 for each of the 8 features of the CFT-2 datasets, respectively. Threshold α is set offline using the training data for each feature with values 0.09, 0.06, 0.01, 0.01, 0.01, 0.01, 0.02, and 0.04, respectively. The feature importance drop is higher than the corresponding α for 5 features out of 8 features, which shows that 5 features experience a feature drift while three other features are not drifting.

Table 4.2 summarizes the feature drift detection delay and accuracy of the proposed solution

Table 4.2: Evaluation of the proposed feature drift detector on migration, CFT-1, and CFT-2 datasets.

Dataset	Drift Detection Delay (Data Samples)	Drift Detection Accuracy (%)
Migration	400	100
CFT-1	700	100
CFT-2	400	100

across the three studied datasets. As shown in Table 4.2, the feature drift detection accuracy is 100% for all datasets. However, the average feature drift detection delay is 500 data samples. Knowing an average feature drift detection delay, a buffer can be considered to store the data to be used for drift adaptation purposes in addition to the data that will be collected after feature drift detection.

C) Feature Drift Adaptation Results

Next, we evaluate the performance of the proposed feature drift adaptor in terms of the prediction model's post-adaptation F1-score and MAE. After feature drift detection, the Drift Severity Analysis component determines whether the feature drift is severe or not by verifying whether the features with high contribution to the prediction model's performance experienced a drift. Threshold β specifies whether the feature has a high contribution or not. We set threshold β to the average initial feature importance across all features. Therefore, if a feature's importance is greater than the average initial importance, it is considered as a high-contributing feature, whose drift may result in a significant degradation in model's performance. Recall from Section 4.5.2 that in the migration dataset, all features experienced a feature drift. Therefore, all the features including the ones with high contribution experienced a feature drift, which makes this a severe drift. To perform feature drift adaptation for a severe drift, new features are selected and a new prediction model is trained using the newly selected features. Similarly, for the CFT-1 dataset, the feature importance drop is severe since all the features experience a feature drift. Therefore, new features are selected for training a new prediction model. For the CFT-2 dataset, Feature Drift Detector component showed that five features out of eight features drifted. Therefore, the Drift Severity Analysis component should find whether the majority of the high-contributing features are drifting in order decide if the detected feature drift is severe or non-severe. In CFT-2 dataset, threshold β, i.e., the average

initial feature importance across all features, is 0.08, where the initial feature importance for each feature were 0.22, 0.11, 0.02, 0.01, 0.03, 0.02, 0.02, and 0.17, respectively. This shows that the first, second, and last feature are the high-contributing features. Considering the findings of the Feature Drift Detector discussed in Section 4.5.2, only the last feature among the high-contributing features has experienced a drift. Therefore, the majority of the high-contributing features are still relevant to the learning task, and the Drift Severity Analysis declares this drift a non-severe feature drift. Therefore, to adapt to this drift, the drifting features should be eliminated (without performing feature re-selection) and the model should be fine-tuned.

To perform feature drift adaptation, we need to collect some new data after the drift. To set the amount of data required for adaptation, we considered the evaluation results in [1] and [2] that studied determining the suitable amount of data to learn a new task from scratch for concept drift adaptation. The evaluations of these work showed that 3900 to 5400 data samples were required for adaptation to various drifts. If the size of collected data for adaptation is too small, the performance of the prediction model will not recover after adaptation. On the other hand, if a large amount of data is collected for adaptation, although the performance of the model after adaptation would recover, it would take a longer time to perform drift adaptation. Based on these findings, we have collected 4000 data samples to perform the adaptations in all datasets. As discussed in Section 4.5.2, we use Pearson + TLCC approach for feature selection. The features selected to perform feature drift adaptation for the migration dataset are as follows: *node_schedstat_waiting_seconds_total*: number of seconds a process spent waiting for this CPU, which is for one CPU core of the VM that was the destination of the service migration and is experiencing the CPU over-utilization fault, *node_softnet_processed_total*: number of processed packets, which is for one CPU core of the VM that was the destination of the service migration and is experiencing the CPU over-utilization fault, *container_cpu_usage_seconds_total*: amount of CPU time spent in a pod scheduled on the VM that was the destination of the service migration.

We observe from the feature list above that the features selected after a feature drift are different from the features selected before the drift. More specifically, the features selected after the drift are the features related to the CPU cores of the VM that was the destination of the service migration and was experiencing the CPU over-utilization fault. This indicates that the proposed solution can

effectively detect a drift and update the set of features accordingly. The features selected for feature drift adaptation for the CFT-1 dataset are as follows: *node_netstat_Tcp_RetransSegs*: total number of segments re-transmitted in the VM experiencing packet loss, *node_network_receive_packets_total*: network statistics indicating amount of received packets in the VM that is experiencing packet loss, *node_network_receive_bytes_total*: network statistics indicating amount of received bytes on the VM that is experiencing packet loss, *node_nf_conntrack_entries*: number of currently allocated flow entries for connection tracking of the VM experiencing packet loss.

It can be observed from the above-mentioned list of selected features that after occurrence of the feature drift, the selected features are network statistics of the applications deployed on the VM experiencing packet loss. However, before the drift, when the underlying fault causing application performance degradation was CPU over-utilization fault, the features were related to the CPU time of the VM. For the CFT-2 dataset, since the feature drift was non-severe, the drifting features are eliminated from the list of features and feature re-selection is not performed. The features that did not drift and are used for adaptation are as follows (as described in Section 4.5.2): *node_schedstat_waiting_seconds_total*, *node_cpu_seconds_total*, *node_memory_Cached_bytes*.

The above list of feature shows that most of the remaining relevant features after occurrence of feature drift are the features related to CPU. Considering that in CFT-2 dataset the underlying fault type is changed from HDD to CPU over-utilization, this indicates that the proposed solution can effectively locate the drifting features based on the changes in their feature importance. We have also performed a feature re-selection just to realize whether the re-selected features are similar to the above-mentioned feature list. We observed that although the first two features, i.e., *node_schedstat_waiting_seconds_total* and *node_cpu_seconds_total* are re-selected, two additional CPU-related features, i.e., *node_schedstat_running_seconds_total* and *node_schedstat_timeslices_total*, are also selected.

Table 4.3 presents the post-adaptation F1-score of the prediction model adapted using the proposed feature drift adaptor for the migration, CFT-1, and CFT-2 datasets. Recall from Sections 4.5.2 and 4.5.2 that for the migration dataset, the prediction F1-score before the occurrence of the drift is 81.2%. A feature drift is detected around the sample 1600, and 4000 data samples are collected for adaptation. Once the drift adaptation is performed by selecting 3 new features and retraining the

Table 4.3: Feature drift adaptation evaluation of the proposed solution on migration, CFT-1, and CFT-2 datasets.

Dataset	Feature Size	Post-Adaptation F1score (%)	Post-Adaptation MAE
Migration	3	81.73	0.2384
CFT-1	4	79.23	0.3462
CFT-2	3	83.09	0.265

prediction model, as shown in Table 4.3, the model's F1-score recovers to 81.73%. For the CFT-1 dataset, it is observed in Table 4.1 that the original F1-score of the prediction model before occurrence of drift is 77.49%. The feature drift is detected around sample 4700 and 4000 data samples is then collected for adaptation purposes. After the adaptation is performed by selecting 4 new features and retraining the model, the prediction model recovers and shows an initial F1-score of 79.23%. For the CFT-2 dataset, the original F1-score of the prediction model was 81.17% (as indicated in Table 4.1). A non-severe feature drift was detected at sample 4400, and 4000 data samples were collected for adaptation. Once the model is adapted to the feature drift through feature elimination and fine-tuning, the the model shows an F1-score of 83.09%.

The F1-score values reported in Table 4.3 show the post-adaptation F1-score when 4000 data samples are collected to perform the adaptation. However, in time-sensitive use-cases where collecting 4000 data samples can be time-consuming, collecting a smaller amount of data while sacrificing the post-adaptation performance of the model can be considered. Although the number of data samples required for adaptation should be set prior to performing adaptation, we have conducted an experiment to show the trend of changes in post-adaptation F1-score when various number of data samples are collected for adaptation. In this experiment, we modified the data size to be collected after feature drift from 500 to 7000 within 100 intervals, adapted the model using this data, and measured the F1-score of the adapted model on the remaining portion of the data. The experiment is repeated across all datasets.

Fig. 4.4 depicts post-adaptation F1-score vs. number of data samples used for adaptation for migration, CFT-1, and CFT-2 datasets. Given that there are fluctuations in F1-score while changing the data size with 100 intervals, Fig. 4.4 shows a moving average of the F1-scores with a window size of 10. As illustrated in Fig. 4.4, for all datasets, it can be observed as a general trend that as

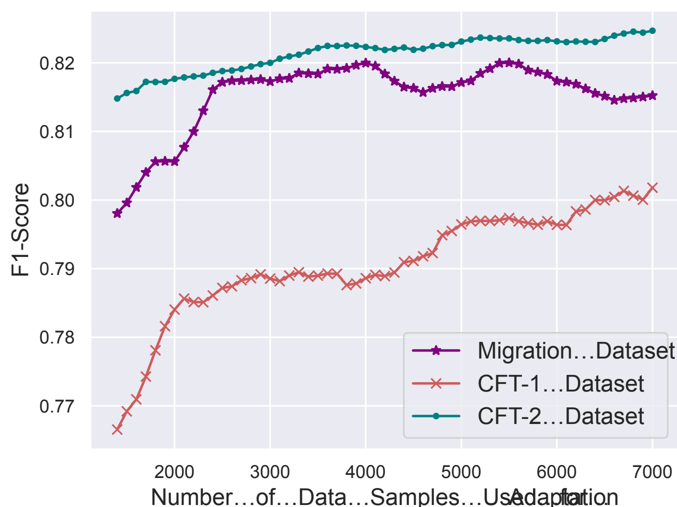


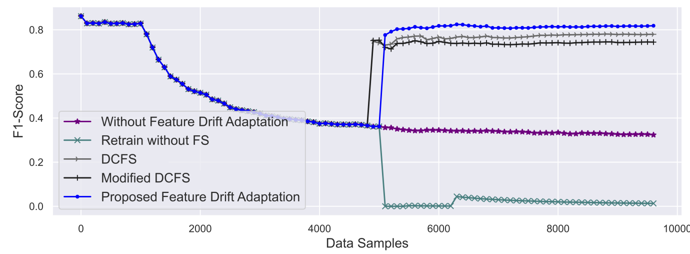
Figure 4.4: Post-adaptation F1-score vs. number of data samples used for adaptation for migration, CFT-1, and CFT-2 datasets.

the number of data samples used for adaptation increases, the post-adaptation F1-score is improved, although some fluctuations in F1-score is observed due to change in the training data size. It can also be observed that changing the number of data samples does not affect the post-adaptation F1-score as much in CFT-2 dataset compared to CFT-1 and migration datasets. This is because there was a non-severe feature drift in CFT-2 dataset, and the model was fine-tuned for adaptation. Therefore, even adapting with fewer data samples can result in an F1-score close to that of the original model. Another observation is that using as few as 500 data samples for adaptation is feasible and results in a slight reduction in the post-adaptation F1-score: approximately 1% for CFT-2 and around 2% for the migration and CFT-1 datasets. This is specifically helpful for time-sensitive scenarios, where the objective is to minimize the number of data samples for adaptation or when the resources for gathering a large volume of data are limited.

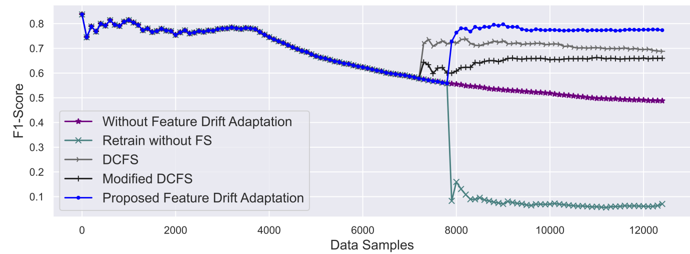
D) Comparative Results

In this experiment, we evaluate our proposed feature drift adaptation algorithm against four benchmarks to assess its performance. These benchmarks include the DCFS [47] and its modified version, alongside a common strategy for concept drift adaptation which involves retraining the model without re-selecting features, and a scenario where no drift adaptation is implemented. DCFS

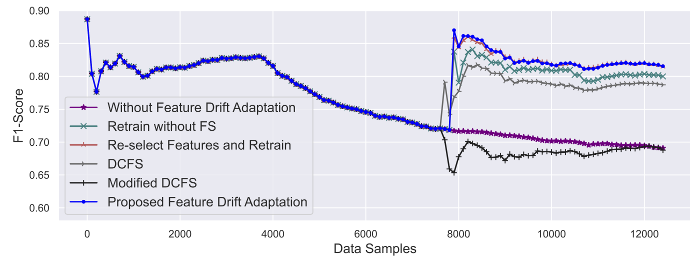
leverages ADWIN for detecting feature drifts, and adapts by re-selecting features and retraining the model. It employs a backward feature elimination technique based on a merit score to assess feature relevance. Given the time-consuming nature of applying backward feature elimination in scenarios with high-dimensional feature spaces like metrics of a cloud system, we eliminate features in batches, e.g., 10 at a time, to improve feasibility. However, due to the time-consuming nature of even batch-wise feature elimination in high-dimensional feature spaces, we also evaluate a modified DCFS approach. This variation simplifies the process by calculating the merit score for all features once, and then selecting the top features based on this score.



(a) F1-score of the trained model for predicting application performance degradation over time for the migration dataset.



(b) F1-score of the trained model for predicting application performance degradation over time for the CFT-1 dataset.



(c) F1-score of the trained model for predicting application performance degradation over time for the CFT-2 dataset.

Figure 4.5: F1-score over time of application performance degradation prediction model.

Fig. 4.5a illustrates the post-adaptation F1-scores of the prediction model over time using the

proposed feature drift adaptation solution in comparison with DCFS, modified DCFS, no drift adaptation mechanism, and retraining without feature re-selection for the migration dataset. As illustrated in Fig. 4.5a, the prediction F1-score before the occurrence of the drift is 81.2%. The feature drift is detected around the sample 1600 and amount of 4000 data samples are collected for adaptation. Once the drift adaptation is performed, the model's F1-score recovers to its original F1-score. However, the F1-score of retraining without any feature re-selection is almost zero, which shows that the initially selected feature cannot represent any valuable information for learning application performance degradation after occurrence of the drift. When no feature drift adaptation mechanism is applied, the F1-score is around 32.35%. This shows that the knowledge learned by relevant features before the drift is more useful than training a new model with the same set of features. This is due to the fact that pattern of fault injection does not change after the drift and the old model can still perform better than retraining a model with irrelevant features. DCFS and modified DCFS use ADWIN and detect feature drift around the sample 1400. For fair comparison, we use the same amount of data, i.e., 4000 data samples, for feature re-selection and model re-training. DCFS takes around 14 hours to select features using backward feature elimination, while modified DCFS performs feature re-selection in 57.7 seconds. It can be observed that DCFS achieves F1-score of 77.94% while F1-score of modified DCFS is 74.4%. This shows that for the migration dataset, DCFS can select features resulting in slightly better performance compared to modified DCFS in significantly longer time. However, neither of these approach can achieve the original F1-score of the model, due to their feature selection approach. Another shortcoming of DCFS and modified DCFS is the feature drift detection approach, where they only monitor model performance and not feature importance or correlation. Therefore, if a concept drift occurs, DCFS approaches perform unnecessary feature re-selection while only re-training the model would have adapt the model.

The same behaviour is exhibited in Fig. 4.5b for the CFT-1 dataset. The original F1-score of the prediction model before occurrence of drift is 77.49%. It can be observed that after the adaptation, the prediction model recovers and shows an F1-score of 77.9% after receiving the last data sample. The F1-score of retraining without any feature re-selection approach is near zero, which is even lower than when no feature drift adaptation is applied, i.e., F1-score of 48.78%. DCFS has a slightly higher F1-score compared to modified DCFS, i.e., 68.8% compared to 65.98%. However, DCFS

takes around 12 hours to re-select features while modified DCFS takes 69.4 seconds. Similar to the results from migration dataset, DCFS approaches cannot select features that reach the original F1-score of the model.

For the CFT-2 dataset, as illustrated in Fig. 4.5c the F1-score of the prediction model before occurrence of feature drift was 81.17%. After occurrence of the feature drift and adapting the prediction model through feature elimination and model fine-tuning, the F1-score of the model recovers to 81.53% after receiving the last data sample. The F1-score of retraining without any feature re-selection approach is 79.98%, which is significantly higher compared to migration and CFT-1 datasets. This is because in CFT-2 a non-severe feature drift was observed, which means that some features are still relevant to the learning task. Therefore, retraining the model using the same set of features can also offer a good performance with 1.5% smaller F1-score. When no feature drift adaptation is applied, the F1-score of the model is 69.09%, which shows smaller loss of F1-score compared to the migration and CFT-1 dataset since the feature drift is non-severe. We also considered an additional adaptation approach for comparison in Fig. 4.5c to compare feature elimination and fine-tuning approach to feature re-selection and retraining approach. It can be observed that re-selecting features and re-training the model shows almost the same performance (F1-score of 81.46%) as the feature elimination and fine-tuning approach suggested by the proposed feature drift adaptation solution. Similar to CFT-1 dataset, DCFS results in higher F1-score compared to modified DCFS while taking a significantly longer time for feature re-selection. The F1-core of DCFS is 78.7%, which is around 2.37% smaller that the original model.

4.6 Conclusion

In this chapter, we proposed a feature drift adaptation solution for adapting to feature drifts while predicting application performance degradation in cloud environments. This solution consists of a feature drift detector that detects feature drifts by monitoring the performance of the prediction model as well as the feature importance, and a feature drift adaptor that measures the drift severity to adapt the prediction model by performing either feature re-selection and re-training the model or

dropping the irrelevant features and fine-tuning the prediction model. To demonstrate the effectiveness of our proposed solution, we experimentally built a proof-of-concept of our solution on a cloud testbed using Kubernetes, and evaluated and compared the proposed solution to four benchmarks. Our results demonstrate that the proposed Feature Drift Detector and Feature Drift Adaptor can effectively detect the feature drift and update the features and adapt the prediction model to the drift, respectively. Moreover, the proposed feature drift adaptation solution can maintain the performance of the prediction model close to its original F1-score.

Chapter 5

Data-related Parameter Selection Algorithm for ML-based Fault Prediction [1]

5.1 Introduction

The performance of ML models, i.e., their accuracy in predicting faults or application performance degradation caused by faults, is influenced by several factors, including training data size, data sampling interval, input window, and prediction horizon [15]. A longer prediction horizon can provide lead time for proactive measures but may reduce performance due to the increasing uncertainty over extended periods, while a shorter prediction horizon might yield higher prediction performance but offer limited foresight for preemptive actions [15]. On the other hand, larger training data sizes typically enhance model performance due to the richer information available for learning [15]. Additionally, increasing the data sampling interval can lead to more granular data, potentially improving the model's ability to predict subtle performance changes [52]. This can also result in a larger volume of data, which needs more storage and network resources for data collection. Cloud service providers prefer building models that minimize resource consumption while

[1] This chapter is based on a submitted paper: [5] B. Shayesteh, C. Fu, A. Ebrahimzadeh, R. Glitho, "Data-related Parameter Selection for Predicting Application Performance Degradation in Clouds using Deep Learning Models," Submitted to *IEEE Transactions on Cloud Computing*, under review.

expecting the models to accurately predict application performance degradation. In many cases, the performance and resource consumption requirements are conflicting.

To address this challenge, in this chapter, we propose an NSGA-II-based multi-objective optimization algorithm to automate selection of the data-related parameters for training an ML model that predicts application performance degradation in clouds, with the objective of maximizing the performance of the model while minimizing resource consumption of data collection and storage. The algorithm uses surrogates to estimate the model performance for each set of data-related parameters instead of testing the performance of each model trained by the set of the parameters, to accelerate the optimization time.

The rest of this chapter is organized as follows. First, we present an illustrative use case followed by the system model and the problem formulation. Next, we describe the proposed data-related parameter selection algorithm. Finally, we describe the lab setup and the implementation results and conclude this chapter.

5.2 Illustrative Use case

This section presents the motivation for selecting data-related parameters for training models predicting application performance degradation in clouds, through an illustrative use case of a 5G mobile network. In this scenario, a 5G mobile network provider rents cloud services from a cloud service provider with the objective of minimizing rental costs while ensuring the availability of the 5G services. To achieve these objectives, the mobile network provider needs to optimize resource usage for proactive service assurance while ensuring the effectiveness of this service assurance scheme. Such a scheme can be based on ML models and would require accurate predictions of performance degradations for timely prevention to maintain service availability.

The 5G mobile network system consists of three components: User Equipment (UE), Radio Access Network (RAN), and the 5G Core [84], where the 5G Core services are hosted in cloud as visualized in Fig. 5.1. The 5G system connects the UEs to the Data Network or other UEs by using the services of 5G Core. According to The Third Generation Partnership Project (3GPP), 5G core architecture consists of several network functions or services depicted in Fig. 5.1. To

register a UE in 5G networks, several steps involving different services are carried: 1) the UE begins the process by sending a registration request to RAN, which is forwarded to the Access and Mobility Management Function (AMF), 2) the AMF initiates the verification process by requesting the UE to confirm its identity by sending its International Mobile Subscriber Identity (IMSI), with the RAN acting as the intermediary, 3) the AMF checks if the UE is allowed to join the network by sending a request to the Authentication Server Function (AUSF), 4) the AUSF exchange request and responses with the Unified Data Management (UDM) to retrieve authentication information and authenticates the UE, 5) upon authentication completion, the AMF exchange messages with the UDM to register this UE in the network, 6) the AMF works with the Policy Control Function (PCF) to establish network usage policies for the UE's session, 7) the AMF communicates with the Session Management Function (SMF) to set up the session for the UE, 8) the AMF sends a message to the RAN to signal registration acceptance, and 9) the UE confirms the registration by sending a registration completion message to the AMF. The UE registration time and the number of UE registration requests per minute are two example KPIs that may be impacted by 5G network and core failures. The 5G mobile network provider may want to predict these KPIs to proactively handle the potential failures that will have impact on UE registrations.

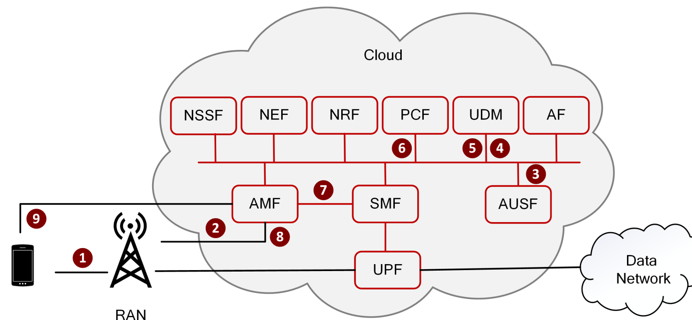


Figure 5.1: Illustrative use case with 5G core services deployed in cloud.

Given that the 5G mobile network provider wants to use ML models to predict these KPIs in order to take prevention actions that prevent KPI degradations, it needs to collect training data that includes KPIs and other 5G core services as well as cloud infrastructure performance metrics like computing resource and network usage, to learn to predict patterns that indicate impending

degradations in KPIs. However, collecting and processing this high-dimensional data is resource-intensive. The provider aims to ensure that the ML model can predict KPI degradations with the highest possible performance while minimizing the resource usage for data collection, processing, and storage. To achieve this, the cloud service provider needs to select data-related parameters that influence model's performance, so that the model can predict KPIs with the highest performance while minimizing resource usage.

5.3 System Model

Fig. 5.2 presents a high-level view of a cloud environment hosting 5G core services. As illustrated in Fig. 5.2, Cloud Infrastructure provides computational resources, storage, and networking capabilities to facilitates the deployment and scaling of Virtual Network Functions (VNF)s, e.g., 5G core services. The The Operation/Business Support System (OSS/BSS) interfaces with both the ML Pipeline and the 5G Core VNFs to monitor the 5G core services and provide operational support and business management. The ML Pipeline performs the life cycle management of ML models trained for management tasks. We consider a time series forecasting ML model deployed in OSS/BSS, which predicts degrdations in KPI(s) of 5G core. The UE registration time is an example KPI that can be impacted by the failures in 5G core. The prediction model will predict the value of the KPI, and KPI values above a given threshold are determined as degradation in the KPI. The underlying cause of performance degradations in 5G core services are diverse and can include infrastructure faults such as CPU or HDD over-utilization, network congestion, or network packet loss. The ML Pipeline would receive the data it requires for training such prediction model from infrastructure monitoring that monitors the infrastructure and OSS/BSS that monitors the 5G core services. Infrastructure-level performance metrics can include resource utilization such as CPU, memory, disk, and network metrics, while 5G core performance metrics can include various KPIs. Before training the prediction model, the Prediction Optimizer is responsible for selecting data-related parameters such as training data size, data sampling interval, input window, and prediction horizon of the model, so that this model would achieve the highest possible prediction performance while minimizing the resources required for data collection and storage. The Prediction Optimizer

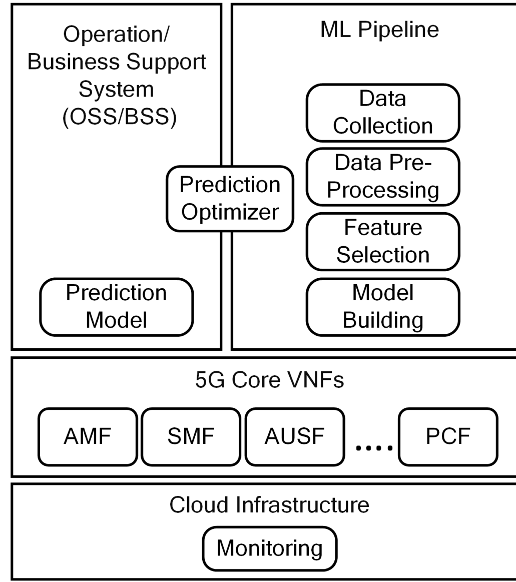


Figure 5.2: Generic system model of a prediction optimizer in clouds environments.

obtains the search space, i.e., upper and lower boundaries of each parameter, from the OSS/BSS to perform parameter selection in the ML Pipeline. When these parameters are selected, the raw training data is collected according to the training data size and sampling interval and passed to the ML Pipeline, where the Data Pre-processing cleans the data from inconsistencies (e.g., outliers, null values, etc.), and normalizes the data. Next, the pre-processed data is used to select the features, i.e., infrastructure-level and 5G core performance metrics that have the highest correlation with KPI degradation. Finally, these features are used for training the prediction model with a prediction horizon defined by Prediction Optimizer.

In the cloud environment described above, let all the data available for training the prediction model be $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, where n is the total number of data samples available for training a prediction model. Let us further assume every s^{th} sample in D is selected to form a subset of data that will be used for training the prediction model, ensuring $n > s$ for feasible sub-sampling. We define a sub-sampled dataset $D' = \{(x_s, y_s), (x_{2s}, y_{2s}), \ldots, (x_{ms}, y_{ms})\}$, where $m = \left\lfloor \frac{n}{s} \right\rfloor$ represents the total number of selected samples for training the model. In this sub-sampled dataset D', each x_{is} (for $i = 1, 2, \ldots, m$) is a vector of features from the feature set R with r size, and $y_{is} \in \{\mathcal{C}_{degrade}, \mathcal{C}_{\textit{no-degrade}}\}$ is the class label denoting whether application performance will degrade or not for vector x_{is}. $\mathcal{C}_{degrade}$ indicates the class label for the occurrence of

application performance degradation, while $\mathcal{C}_{no\text{-}degrade}$ indicates the class label for normal application performance. For training the prediction model, given that this is a time series forecasting task, we define an input window size w and a prediction horizon h. This means that for every set of w consecutive samples in D', the model will predict the outcomes for the next h samples. Let the prediction model trained for predicting application performance degradation using the training dataset D' be $F : (x_{is}, x_{(i+1)s}, \ldots, x_{(i+w-1)s}) \rightarrow (y_{(i+w)s}, \ldots, y_{(i+w+h-1)s})$. The model F can predict performance degradation of a given input window. Table 5.1 presents a summary of the key notations used in this chapter.

Table 5.1: Summary of key notations.

Notation	**Description**
D	Full dataset comprising all available data samples
n	Total number of data samples available in D
s	Data sampling interval to sample from dataset D
D'	Training dataset by selecting every s^{th} sample of D
r	Feature size of training dataset D'
m	Total number of selected samples for training the model
w	Input window (number of samples used for a prediction)
h	Prediction horizon (number of future samples to predict)
F	Prediction model
A	Performance of the prediction model F
C	Resource consumption of data collection and storage

5.4 Problem Formulation

To formulate the problem of optimizing data-related parameters, we define four decision variables as follows:

- h: An integer variable, indicating the prediction horizon, i.e., the number of future data samples the prediction model F is expected to predict.
- w: An integer variable, indicating the input window size, i.e., the number of data samples the prediction model F should consider to make a prediction.
- s: An integer variable, indicating the sampling interval for selecting every s^{th} sample from the dataset of all available data for training.

- m: An integer variable, indicating the number of training data samples that will be used for training the prediction model F.

Next, we describe the constraints on the decision variables that should be considered to ensure the feasibility of the solution.

Constraint (1): the number of data samples the prediction model F uses as input to make a prediction (i.e., the input window size, w) shall be equal to or greater than the number of future data samples the prediction model F is expected to predict (i.e., the prediction horizon, h). This constraints can ensure that the model F has enough historical context to generate predictions for the specified horizon.

$$w \geq h \tag{5.1}$$

Constraint (2): the available dataset D, with n total data samples, shall be large enough to allow for the selection of every s^{th} sample and still obtain m data samples for training the prediction model F. This condition ensures that the sub-sampling strategy does not exceed the bounds of the dataset, preserving the integrity of the training process.

$$m \cdot s \leq n \tag{5.2}$$

Constraint (3): the training data size m shall be large enough to allow splitting into one or more of the input window w number of data samples and the prediction horizon h number of data samples.

$$m \geq w + h \tag{5.3}$$

The first objective of our problem is to maximize the performance of model F. Note that performance of a prediction model can be evaluated using various metrics such as accuracy, precision, recall, or F1-score. Here, we define the first objective as a general performance function A of the model F. Assuming a testing dataset is available for evaluating the model defined as $T = \{(x'_1, y'_1), (x'_2, y'_2), \ldots, (x'_q, y'_q)\}$, where q is the number of samples in the testing dataset, the

performance of the model F over the testing data is then given by:

$$A = \sum_{i=1}^{q-w} L(y'_i, \hat{y}'_i) \tag{5.4}$$

Where $L(y'_i, \hat{y}'_i)$ is a loss function measuring the difference between the actual value y'_i and predicted value $\hat{y}'_i$ for the ith data.

The second objective of our problem is to minimize resource consumption of data collection and storage for training model F. The resource consumption for storing the data is given by:

$$C = \alpha \cdot m \cdot r \tag{5.5}$$

Where α is a coefficients reflecting the unit of storage for one data sample and r is the feature size.

Consequently, our multi-objective optimization problem can be formulated as follows:

$$\begin{aligned} &\text{Minimize} \quad C \\ &\text{Maximize} \quad A \\ &\text{subject to constraints (1)-(3).} \end{aligned} \tag{5.6}$$

In multi-objective optimization problems, the goal is to find a trade-off between the two objective as it is often not possible to optimize one objective without making the other objective worse. Instead, the multi-objective optimization aims at finding a set of non-dominated solutions called the Pareto front. A solution is said to be non-dominated by other solutions if there is no other solution that improves some objectives without worsening the other objective [85].

5.5 Proposed Data-related Parameter Selection Algorithm

In this section, we present our proposed surrogate-assisted multi-objective optimization algorithm to select data-related parameters for training an application performance degradation prediction model, while balancing the model performance and resource consumption for data collection and storage. The proposed approach is based on the evolutionary multi-objective algorithm,

NSGA-II. Fig. 5.3 illustrates the flowchart of the proposed data-related parameter selection algorithm, which consists of the following steps.

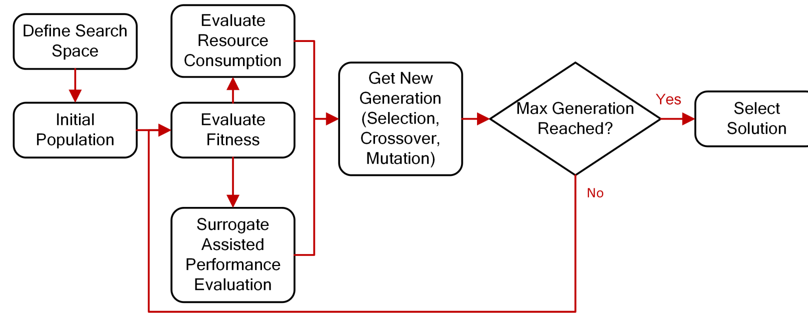


Figure 5.3: Steps of the proposed data-related parameter selection algorithm.

1) Define Search Space: To start selecting the parameters, we need to define the search space for the solutions, i.e., lower bound and upper bound for each decision variable. Each solution in the search space is a set of our four decision variables. One solution (individual) in the search space can be defined as $I_i = \{h_i, w_i, s_i, m_i\}$ for $i = 1, \ldots, z$, where z is the number of solutions in the search space.

2) Initial Population: In context of an evolutionary algorithm, initial population is the first set of potential solutions, on which the objective functions will be evaluated. The initial population is generated randomly within the search space.

3) Fitness Function Evaluation: The optimization algorithm starts by evaluating the fitness of the current population, which involves quantifying how well each solution in the population meets the optimization objectives. In our problem, fitness evaluation consists of evaluating resource consumption as well as model performance for the solutions within the population.

4) Evaluate Resource Consumption: As demonstrated in Eq. 5.5, to evaluate the resource consumption for collecting and storing the training data, we consider the data dimension, i.e., the number of data samples that will be used for training (m) and the feature size (r). Since r is constant for all the solutions in the population, resource consumption is a linear function of m. Therefore, the less data we use for training, the lower the resource consumption.

5) Surrogate-Assisted Performance Evaluation: To evaluate the performance of a prediction

model for each solution, we need to train separate model for each solution, characterized by a specific set of parameters including the prediction horizon, input window, data sampling interval, and training data size (h_i, w_i, s_i, m_i). This process, while thorough, demands substantial computational resources and time due to the necessity of training a separate model for each configuration to obtain its performance. To avoid this, we propose employing a surrogate model that can approximate the model performance for each solution. The surrogate models commonly used in the literature include Multi-Layer Perceptron (MLP), RF, and Kriging regression models [86]. The surrogate model is trained offline using a limited number of solutions and their observed model performance. In our problem, since we have discrete values for our decision variables, i.e., the data-related parameters, we use a RF surrogate [86].

6) Get New Generation: To evolve the solutions from one generation to the next, evolutionary algorithms perform variations through selection, crossover, and mutation. The selection identifies solutions with lower prediction error and resource consumption, guiding the algorithm toward optimal solution. Crossover combines the decision variables from pairs of selected solutions (parents) to create new solutions (offspring) that may perform better in fulfilling the objectives. Mutation randomly changes the value of the decision variables in a solution, e.g., changing the prediction horizon h, to explore potentially better values not presented in the current population. The NSGA-II algorithm performs selection using a non-dominated sorting approach to classify the population into different fronts based on Pareto dominance and selecting solutions based on their rankings. Crossover in NSGA-II is performed using Simulated Binary Crossover (SBX) while mutation is performed using Polynomial Mutation (PM).

7) Select Solution: Upon reaching the predefined maximum number of generations, the evolutionary algorithm concludes in a final generation from which the optimal solution, situated on the Pareto front, must be identified. To select this optimal solution, we use a weighted sum method that assigns a score to each candidate solution on the Pareto front to compare the solutions. This approach does not only account for the primary objectives of our problem but also incorporates the prediction horizon (h) as a decision-making factor. This is because a model with larger prediction horizon can provide a longer lead time for implementing preventative actions. The score V to select the optimal solution is given by:

$$V = \omega_1 \cdot A + \omega_2 \cdot C + \omega_3 \cdot h \tag{5.7}$$

Where $\omega_1, \omega_2, \omega_3$ are coefficients of A, C, and h, respectively. The coefficients indicate the importance of each objective and ω_3 should be smaller than ω_1 and ω_2 to avoid assigning more importance to prediction horizon than the two objectives of the problem.

Algorithm 7 NSGA-II-based Data-related Parameter Selection

1: **Inputs**: Dataset (D), Population size (L), Maximum generations (G_{max}), Trained performance surrogate model (H), Prediction Model (F), Resource Function (C)
2: **Initialize**: Random initial population (P_0), Generate off-springs (Q_0), $t = 0$
3: **while** $t < G_{max}$ **do**
4: $B_t = P_t \cup Q_t$
5: **for** each solution I in B_t **do**
6: $ranks \leftarrow$ rank solution using $H(I)$ and $C(I)$
7: **end for**
8: $\mathcal{F}_{1,2...} \leftarrow$ fast-non-dominated-sort($B_t, ranks$)
9: $P_{t+1} = \varnothing$
10: $i = 0$
11: **while** $|P_{t+1}| + |\mathcal{F}_i| \leq L$ **do**
12: $\mathcal{F}_i \leftarrow$ sort solutions in $\mathcal{F}_i$ using crowding distance
13: $P_{t+1} = P_{t+1} \cup \mathcal{F}_i$
14: $i = i + 1$
15: **end while**
16: $P_{t+1} = P_{t+1} \cup \mathcal{F}_i[0 : (L - |P_{t+1}|)]$
17: $Q_{t+1} \leftarrow$ make-new-population(P_{t+1})
18: $t = t + 1$
19: **end while**
20: **return** Solution I in $\mathcal{F}_1$ with the highest score based on Eq. 5.7

Algorithm 7 illustrates the steps of our proposed data-related parameter selection algorithm. The time complexity of the NSGA-II algorithm for each generation is $\mathcal{O}(L^2)$, where L is the population size [85]. This is because time complexity of performing fast-non-dominated-sorting to sort the combined population (B_t) with size $2L$ is $\mathcal{O}((2L)^2)$. Considering that the algorithm is repeated for G_{max} generations, the overall time complexity of the algorithm is $\mathcal{O}(G_{max}.L^2)$.

5.6 Performance Evaluation

In this section, we evaluate the performance of the proposed surrogate-assisted NSGA-II optimization algorithm for selecting the data-related parameters while predicting application performance degradation in a real-world testbed. We deployed 5G core on Kubernetes with the objective of predicting two of its KPIs. In the following, we present the experiment settings and evaluation results.

5.6.1 Experiment Settings

We describe the details of our lab setup, fault injection specification, datasets, and algorithm search space for the experiments in the following sub-sections.

A) Lab Setup: Figure 5.4 depicts our lab setup, which consists of a Kubernetes cluster with a total of 3 VMs running *Ubuntu 20.04*. The cluster is hosted in a private cloud, managed by the Infrastructure-as-a-Service OpenStack. The configurations of the VMs are summarized in Table 5.2. We deployed an open-source 5G core called *Open5GS* [87] in this cluster using the helm charts generated and maintained by [88], which includes microservices of 5G core. We also deployed UEs and a RAN using *UERANSIM* [89], which is connected to the 5G core. Moreover, we deployed *Prometheus* [71] to monitor the cluster and to collect data every 5 seconds to reflect the latest changes in the metrics. We collected the node-level and container-level metrics (i.e., VM and container metrics such as the CPU, memory, and network metrics), as well as application-level metrics (i.e., 5G core performance metrics).

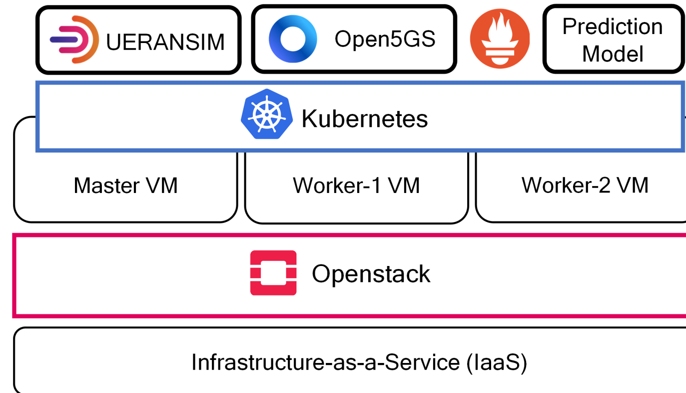


Figure 5.4: Lab setup for evaluating the proposed surrogate-assisted NSGA-II multi-objective optimization algorithm.

Table 5.2: Lab setup parameters and default values.

Site	Parameter	Value
Cloud Site	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

B) Fault Injection Specifications: We initiated node-level CPU over-utilization faults, where the faults were injected to the VM hosting AMF 5G core microservice, which is responsible for handling UE registration requests. The node-level CPU over-utilization faults were injected using the *Stress-ng* [72] tool. The fault injections are recurrent, and the duration and inter-arrival of faults follows a Normal distribution and exponentiated Weibull distribution, respectively, as described in Section 3.5.1.

C) Datasets: We evaluate the performance of the proposed algorithm for selecting the data-related parameters using two datasets explained in the following.

UE Registration Time (UERT) Dataset: Using this dataset, the objective is to predict the UE registration time of the 5G core, i.e., the amount of time in seconds between UE sending a registration request until it is registered. We use UERANSIM to create 6 UEs where UE de-registration and registration requests are sent to the 5G core every 0.2 seconds. While this traffic is introduced to the 5G core, we inject CPU over-utilization faults to the VM where the AMF service that handles registrations requests is hosted on to cause degradations of the KPI. We collect 1 day of data to train a prediction model. The original data consists of 2320 features including node-level, container-level and application-level metrics from the monitored VMs. To find the most relevant features to the target KPI, we used a feature selection mechanism proposed in [3]. The feature selection uses Pearson Correlation Coefficient [90] to find highly correlated features and uses TLCC [91] to find features with causal relationships to the KPI among the highly correlated features. A total of 8 features are selected using this feature selection approach.

UE Registration Requests (UERR) Dataset: In this dataset, we followed the same scenario for sending UE registration requests and injecting CPU over-utilization faults. However, the KPI under study is the number of UE registration requests per minute. Similar to UERT dataset, we collected 1 day of data and used the same feature selection technique to select the most relevant features. A total of 9 features are selected using this feature selection approach.

D) Search Space: The search space for parameter selection is defined by setting the lower and upper bounds for each decision variable, considering the 5G network operator requirements such as resource constraints or minimal prediction horizon. In this experiment, we defined the search space tailored to the characteristics of the two considered datasets. However, the proposed

algorithm can support different value ranges based on the use case. The prediction horizon ranges from 6 to 15, where a minimum of 6 samples (corresponding to a 30-second forecast window, given our dataset's 5-second sampling interval) is selected to ensure the model is capable of taking preventive actions at least 30 seconds before predicted performance degradation. The input window size is adjustable up to twice the size of the prediction horizon, varying from 6 to 30, to provide context for accurate predictions. The data sampling interval is set between 1 and 10, allowing for a range from sampling every 5 seconds to every 50 seconds. This range is chosen to balance detail with computational efficiency, noting that wider sampling intervals might miss performance degradations, which typically last about 3 minutes on average. Lastly, the size of the training dataset is varied from a minimum of 500 samples to a maximum of $13,800$, considering the total dataset length for a day is $17,280$ samples, with 80% allocated for training while 20% reserved for testing.

5.6.2 Evaluation Results

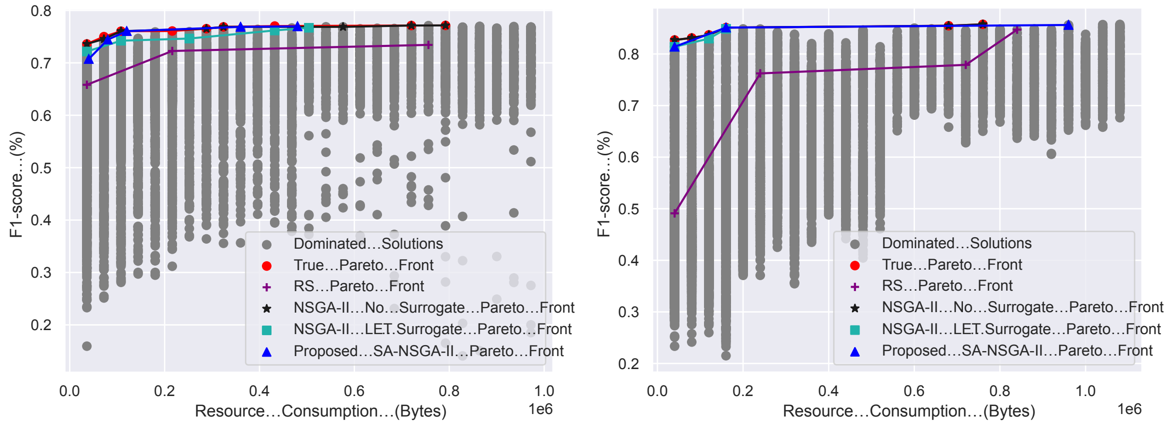
In this section, we present the evaluation of our proposed Surrogate-Assisted NSGA-II (SA-NSGA-II) algorithm aimed at selecting data-related parameters. The effectiveness of our algorithm is benchmarked against four approaches: Random Search (RS), NSGA-II with a Limited Epoch Training (LET) surrogate model, NSGA-II without surrogates, and a comprehensive brute-force search to obtain the true Pareto front. The proposed algorithm uses a RF model trained using 120 observations to approximate the model performance (i.e., F1-scores). All NSGA-II-based approaches are executed with population size 50 for 25 generations. The RS approach randomly samples 50 sets of decision variables, which is equal to the population size of the NSGA-II-based benchmarks, to identify a Pareto front among them, serving as a baseline for comparison. NSGA-II with LET surrogate employs a surrogate model based on the performance of the model when only trained for limited epochs (4 epochs). This method assumes that if models have high performance over a few initial training epochs, they will likely approach a high performance if they are trained fully. Thus, it can significantly reduce the resource and time required for model evaluation. NSGA-II without Surrogates relies on complete model training for performance evaluation. This approach ensures accurate fitness assessments at the expense of higher computational costs. Brute-force search involves exhaustively exploring all potential solutions within the search space to identify the true

Pareto front. Although this method guarantees the discovery of the optimal set of solutions, it is resource-intensive and is conducted solely for comparative analysis. The criteria for comparison to benchmarks include the time required to identify an approximate Pareto front and the optimality gap between the true Pareto front and those produced by each method. To quantify the optimality gap, we visualize the Pareto fronts and employ the normalized hypervolume metric, which calculates the space between the Pareto front and a designated reference point, chosen as the most distant point from the true Pareto front within the search space. Additionally, we examine the data-related parameters resulting from each approach, comparing them to those obtained by the true Pareto front, to highlight the performance and decision-making provided by each method.

A) Pareto Front Analysis and Search Time Results

In this section, we compare the Pareto front approximations obtained by our proposed algorithm and various benchmarks to the true Pareto front. The prediction model is a hybrid CNN-LSTM model designed for predicting application performance degradation. This model architecture includes a convolutional layer, a pooling layer, two LSTM layers, and a dense layer, used for multivariate multi-step time series prediction. The model predicts KPIs, specifically UE registration time for the UERT dataset and the number of UE registration requests per minute for the UERR dataset, using selected features as inputs. For the model performance objective, we utilize the F1-score, which is derived from precision and recall measurements. In the UERT dataset, KPI values exceeding a specific threshold (measured in seconds) are labeled as performance degradation, while those below are considered normal. This threshold is determined based on the registration time under typical request loads. Similarly, for the UERR dataset, a threshold for the number of UE registration requests per minute distinguishes normal operation from performance degradation. A higher F1-score indicates the model's effectiveness in accurately predicting both degradation and normal performance, thus minimizing false alarms that could trigger unnecessary preventative actions. The F1-score values are calculated separately for each prediction horizon of the model and then averaged across all the data samples in the testing data. For evaluating the data collection and storage resource consumption, we measure resource usage by using built-in methods of *Pandas* library storing the training data. This library allocates 8 bytes (i.e., α in Eq. 5.5) for storing a float number. To

measure the total resource consumption, it multiplies 8 bytes by the total number of samples and features size. We observed a fixed overhead of 128 bytes used for creating the dataset regardless of its size.



(a) Pareto front approximations of the considered optimization algorithms on UERT dataset.

(b) Pareto front approximations of the considered optimization algorithms on UERR dataset.

Figure 5.5: Pareto front approximations of the considered optimization algorithms.

Fig. 5.5a and Fig. 5.5b illustrate the Pareto fronts approximation of the proposed algorithm, the considered benchmarks, and the true Pareto front for the UERT and UERR datasets, respectively. The dominated solutions illustrate all sets of decision variables dominated by the solutions in the true Pareto front. It can be observed that the NSGA-II approach operating without surrogates achieves the most accurate approximation to the actual Pareto front. In contrast, the RS benchmark exhibits the least accurate approximation. The LET surrogate and our proposed algorithm, both of which leverage surrogates to approximate the Pareto front, demonstrate close approximations to the true front. This is achieved without fully training prediction models for performance evaluation, with our algorithm showing marginally superior performance to the LET surrogate. However, the surrogate-assisted Pareto front approximation falls short of the NSGA-II with no surrogates. This is because the surrogate-assisted methods do not fully train models to obtain the F1-score, thereby introducing approximation errors relative to the NSGA-II's full-training approach. We note that to illustrate the Pareto fronts approximated by the proposed algorithm and the LET surrogate, we fully trained prediction models using the final decision variables they selected and used their F1-score to find the Pareto front.

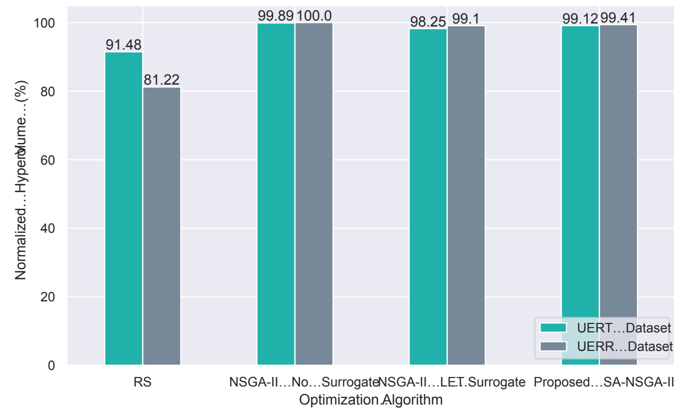


Figure 5.6: Normalized hypervolume obtained by the considered optimization algorithms on UERT and UERR datasets.

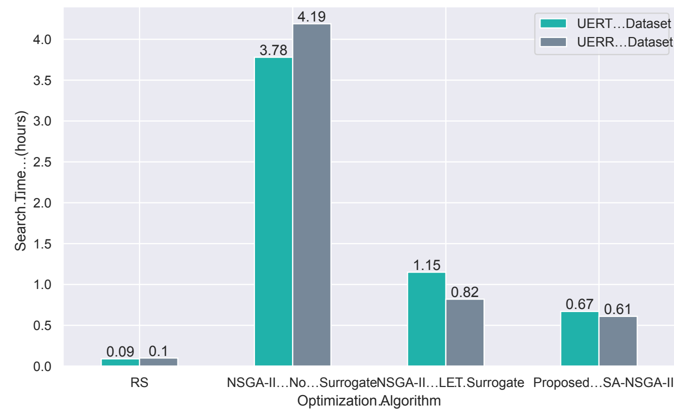


Figure 5.7: Search time of the considered optimization algorithms to obtain the Pareto front approximation on UERT and UERR datasets.

Fig. 5.6 illustrates the normalized hypervolume obtained by the considered algorithms on UERT and UERR datasets. To calculate the normalized hypervolume for each algorithm, we divided its hypervolume by that of the true Pareto front, which was determined using a brute-force approach. Consequently, a normalized hypervolume nearing 100% indicates a closer approximation to the true Pareto front. Consistent with previous observations on Pareto front approximations, the NSGA-II algorithm operating without surrogate assistance, achieved the highest normalized hypervolume values of 99.89% for UERT and 100% for UERR datasets. This underscores its superior approximation capability. Similarly, the proposed algorithm and the LET surrogate demonstrated comparable performance, with the proposed algorithm reaching normalized hypervolumes of 99.12% for UERT

and 99.41% for UERR. On the other hand, the RS algorithm showed the lowest performance, with normalized hypervolumes of 91.48% for UERT and 81.22% for UERR, indicating the least accurate approximation of the true Pareto front. It is important to note, however, that while the NSGA-II without surrogates excels in approximation accuracy, it requires significantly more time to reach the Pareto front approximation compared to surrogate-assisted methods. Fig. 5.7 further illustrates the search time required by each optimization algorithm to achieve their respective Pareto front approximations on the UERT and UERR datasets.

As illustrated in Fig. 5.7, the NSGA-II algorithm operating without surrogate assistance, requires 3.78 hours for UERT and 4.19 hours for UERR, to achieve Pareto front approximations. In contrast, the proposed algorithm significantly reduces approximation time to 0.67 hours for UERT and 0.61 hours for UERR. This efficiency is achieved by employing a RF surrogate model trained on 120 observations, which necessitates full training of the prediction model to obtain F1-scores. The LET surrogate, which approximates the F1-score of the prediction model by limiting training to 4 epochs, also demonstrates reduced approximation times of 1.15 hours for UERT and 0.82 hours for UERR. Meanwhile, the RS algorithm, which derives a Pareto front from 50 randomly sampled solutions, exhibits the shortest search times of 0.09 hours for UERT and 0.1 hours for UERR. We note that obtaining the true Pareto front using a brute-force approach requires 88.38 hours for UERT and 86.06 hours for UERR dataset.

B) Comparison of Optimal Data-related Parameters Results

In this section, we present the data-related parameters obtained by our proposed algorithm and the considered benchmarks and compare them to those obtained by the true Pareto front, to highlight the objective fulfillment status and the decision variables provided by each algorithm. As discussed in Equation 5.7, a weighted sum approach is used to select the set of parameters that find a trade-off between the two objectives. We allocated the weights as ω_1 and ω_2 both at 0.4, and ω_3 at 0.2, to reflect our prioritization that both model performance and resource consumption are treated equally as main objectives, while longer prediction is treated as the secondary objective. Table 5.3 shows a comparison of the objective fulfillment status and the decision variables obtained from the benchmarks and our proposed algorithm on UERT and UERR datasets.

Table 5.3: Comparison of the optimal data-related parameters obtained in the Pareto front approximation of the considered algorithms on UERT and UERR datasets.

Dataset	Optimization Algorithm	Objectives		Decision Variables			
		F1-Score (%)	Resource (Bytes)	Sampling Interval	Prediction Horizon	Input Window	Training Data Size
UERT	True Pareto Front	76.04	108,128	1	6	29	1500
	RS	72.26	216,128	1	9	26	3000
	NSGA-II No Surrogate	76.04	108,128	1	6	29	1500
	NSGA-II LET Surrogate	72.85	36,128	3	7	30	500
	Proposed SA-NSGA-II	76.04	108,128	1	6	29	1500
UERR	True Pareto Front	85.11	160,128	1	6	15	2000
	RS	76.23	240,128	1	9	26	3000
	NSGA-II No Surrogate	85.11	160,128	1	6	15	2000
	NSGA-II LET Surrogate	81.39	40,128	1	6	6	500
	Proposed SA-NSGA-II	85.11	160,128	1	6	15	2000

As shown in Table 5.3, for the UERT dataset, the highest F1-score achieved is 76.04%, with resource consumption of $108,128$ bytes. It can be observed that while no method reached a 100% normalized hypervolume, both the NSGA-II algorithm without surrogates and our proposed algorithm successfully identified the optimal combination of decision variables. This optimal combination matches the performance and resource consumption of the true Pareto front, highlighting their effectiveness in finding the best solution in the true front. However, the LET surrogate's performance, despite being close to the actual front, shows a different pattern. Its solutions were not evenly distributed across the entire front, with a noticeable focus on minimizing resource consumption. Specifically, the LET surrogate opted for a solution that reduces resource use to $36,128$ bytes at the expense of achieving a lower F1-score of 72.85%. The table further shows that the optimal decision variables, NSGA-II without surrogates and the proposed algorithm output a data sampling interval

of 1, a prediction horizon of 6, an input window of 29, and a dataset size of 1500 samples. The sampling interval and prediction horizon are the smallest values in the search space, which contribute to higher performance of the model, while small training data size 1500 is selected to minimize the resource consumption. The LET surrogate favors resource consumption over performance by selecting only 500 data samples for training.

A similar trend can be observed in the UERR dataset. The true Pareto front identifies an optimal solution characterized by an F1-score of 85.11% and resource consumption of $160,128$ bytes. Both the NSGA-II algorithm without surrogates and the proposed algorithm identify this optimal solution. Conversely, the LET surrogate opts for a solution that prioritizes minimal resource usage, consuming only $40,128$ bytes, at the cost of reducing the F1-score by 3.72%. The decision-making trend regarding the selection of decision variables remains consistent with that observed in the UERT dataset, where a small data sampling interval, prediction horizon, and training data size are favored against other possible values for decision variables.

5.7 Conclusion

In this chapter, we proposed a surrogate-assisted multi-objective optimization algorithm based on NSGA-II to automate selection of the training data size, the data sampling interval, the input window, and the prediction horizon for training an ML model that predicts application performance degradation caused by infrastructure faults in clouds. This algorithm maximizes the performance of the prediction model while minimizing resource consumption of data collection and storage. To evaluate the effectiveness of the proposed solution, we conducted experiments on a Kubernetes-based cloud testbed, where a 5G core was deployed. The proposed algorithm was compared to the true Pareto front, RS, NSGA-II without surrogates, and NSGA-II with LET surrogates benchmarks. The results demonstrated that the proposed algorithm can achieve a normalized hypervolume of 99.41% relative to the true Pareto front and reduce search time by 0.48 hours compared to other surrogates and by 3.58 hours compared to NSGA-II using no surrogates.

Chapter 6

Conclusion, Discussion, and Future Work

6.1 Conclusion

Although predicting faults in cloud environments using ML enables a proactive approach to prevent faults in clouds, building accurate prediction models that can maintain their performance in dynamic cloud environments is not an easy task. Occurrence of concept drifts and feature drifts can degrade the accuracy of fault prediction models over time, requiring effective adaptation of prediction models to the drifts. Additionally, the accuracy of ML models is influenced by several data-related parameters, necessitating selection of these parameters to achieve a high model performance. In this thesis, we studied and addressed these challenges by proposing algorithms to adapt prediction models effectively and optimizing data-related parameters selection for enhanced model performance.

In Chapter 3, we introduced a concept drift adaptation algorithm for fault prediction in cloud environments using RL. This algorithm considers the cloud operator's requirements of 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 as well as data size for adaptation that fulfills the operator's requirements. The evaluation results showed the proposed algorithm can effectively maintain the ML model performance in presence of concept drifts and had superior performance

compared to other approaches in terms of drift adaptation time, adaptation resource, and number of data samples for adaptation.

Chapter 4 studied the problem of feature drift adaptation and proposed a feature drift adaptation algorithm to adapt models to feature drifts while predicting application performance degradation in cloud environments. This solution consisted of a feature drift detector that detected feature drifts by monitoring the performance of the prediction model as well as the feature importance, and a feature drift adaptor that measured the drift severity to adapt the prediction model by performing either feature re-selection and re-training the model or dropping the irrelevant features and fine-tuning the prediction model. Our results demonstrated that the proposed Feature Drift Detector and Feature Drift Adaptor can effectively detect the feature drift and update the features and adapt the prediction model to the drift, respectively. Moreover, the proposed feature drift adaptation solution can maintain the performance of the prediction model close to its original F1-score.

Finally, in Chapter 5, we proposed a multi-objective optimization algorithm based on NSGA-II to automate selection of the training data size, the data sampling interval, the input window, and the prediction horizon for training an ML model that predicts application performance degradation caused by infrastructure faults in clouds. This algorithm maximizes the performance of the prediction model while minimizing resource consumption of data collection and storage. The data-related parameters selected by the algorithm are used for training models that predict the degradation of 5G core KPIs. The results demonstrated that the proposed algorithm can achieve optimal solutions in two scenarios while reducing the solution search time compared to the considered benchmarks.

6.2 Impact of Proposed Algorithms on Fault Management in Clouds

ML-based fault prediction solutions can work as a key component of fault management systems and work alongside detection, localization, and mitigation of faults. By predicting faults before they occur, fault prediction enables fault management systems to not only respond to faults as they occur, but also to take preventive actions to avoid potential faults [25] [19]. The proposed algorithms ensure that the fault prediction model accurately predicts impending faults and maintains its accuracy in dynamic cloud environments, despite challenges like concept drift and feature drift that

can otherwise degrade model performance. Accurate fault prediction models ensure that the fault management system can effectively trigger fault prevention mechanisms, since failing to predict faults can result in service failure and false prediction can trigger unnecessary preventive measures that increase the operational costs of the fault management system. Regardless of the type of deep learning model used for fault prediction, the proposed algorithms ensure an accurate model that can maintain its performance in dynamic cloud environments. This is achieved by focusing on the symptoms of concept drift or feature drift and taking model-agnostic adaptation solutions.

While the proposed ML-based fault prediction solutions do not focus on localizing and identifying the root causes of impending faults, the causal-temporal analysis-based feature selection approach used for training the model can provide insights into the underlying cause of faults or application performance degradation caused by faults. These insights can be leveraged to localize impending faults and trigger appropriate prevention mechanisms to avert them.

6.3 Future Work

This thesis presented three contributions on addressing the challenges of employing ML for fault prediction in clouds. However, there still exists several research directions for the future.

6.3.1 Concept Drift and Feature Drift Adaptation

To enhance the concept drift adaptation and feature drift adaptation solutions introduced in Chapters 3 and 4, an interesting research direction involves reducing the amount of data required for drift adaptation. This reduction could decrease the time it takes for the prediction model to regain its pre-drift performance. Currently, our solutions necessitate collecting sufficient data after drift to effectively regain the prediction model performance prior to the drift. An interesting alternative to avoid extensive data collection is the exploration of Generative Adversarial Network (GAN)s to create synthetic data following a drift.

6.3.2 Integration of Fault Prediction and Prevention Solutions

In addition to the challenges studied in this thesis, other challenges of employing ML models for fault prediction should be tackled. While the primary objective of fault prediction models in cloud is preventing faults, limited works in the literature integrate prediction models with prevention mechanisms. Current studies focus on triggering prevention mechanisms for a given fault type, but a system that can trigger various mechanisms based on the type of predicted fault and its underlying cause is needed. Creating such a systems requires identification of the impending fault's cause(s), and a mapping solution designed by domain experts linking the impending faults to a prevention action, considering factors including fault cause, remaining time to failure, and available resources for prevention.

6.3.3 Multi-fault Prediction

Current fault prediction models in clouds, including the solutions proposed in this thesis, are developed to predict specific fault types. However, training and maintaining individual models for each fault type can be challenging given the multitude of fault types with diverse underlying causes. Future research can aim at developing prediction models that can predict multiple fault types. This can be achieved by identifying faults with common symptoms and using multi-task learning techniques to train models that can predict these faults.

6.3.4 Explainability of Fault Prediction Models

While deep learning models are popular for predicting faults in clouds and have been used in this thesis, they are often black-boxes lacking clear decision-making process. Therefore, they cannot provide the rationale behind fault predictions to guide prevention actions. Developing new approaches to improve the explainability of fault prediction models is essential [27] and can be achieved through integrating Explainable AI methods with fault prediction solutions. Adapting Explainable AI techniques to suit characteristics of fault prediction solutions, especially in RNN-based models, remains a challenge that future research can tackle [9].

Bibliography

[1] Behshid Shayesteh, Chunyan Fu, Amin Ebrahimzadeh, and Roch 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)*, pages 217–223, Oct. 2021.

[2] Behshid Shayesteh, Chunyan Fu, Amin Ebrahimzadeh, and Roch H Glitho. Automated concept drift handling for fault prediction in edge clouds using reinforcement learning. *IEEE Transactions on Network and Service Management*, 19(2):1321–1335, Feb. 2022.

[3] Behshid Shayesteh, Chunyan Fu, Amin Ebrahimzadeh, and Roch Glitho. Causal-temporal analysis-based feature selection for predicting application performance degradation in edge clouds. In *Proc. IEEE International Conference on Communications (ICC)*, pages 5496–5501, May 2023.

[4] Behshid Shayesteh, Chunyan Fu, Amin Ebrahimzadeh, and Roch H Glitho. Adaptive feature selection for predicting application performance degradation in edge cloud environments. *IEEE Transactions on Network and Service Management*, 2024. in revision.

[5] Behshid Shayesteh, Chunyan Fu, Amin Ebrahimzadeh, and Roch H Glitho. Data-related parameter selection for training deep learning models predicting application performance degradation in clouds. *IEEE Transactions on Cloud Computing*, 2024. under review.

[6] Behshid Shayesteh, Amin Ebrahimzadeh, and Roch H Glitho. Machine learning for predicting infrastructure faults and job failures in clouds: A survey. *IEEE Communication Magazine*, 2024. to appear.

[7] Avinab Marahatta, Qin Xin, Ce Chi, Fa Zhang, and Zhiyong Liu. PEFS: AI-driven prediction based energy-aware fault-tolerant scheduling scheme for cloud data center. *IEEE Transactions on Sustainable Computing*, 6(4):655–666, Aug. 2020.

[8] Mukosi Abraham Mukwevho and Turgay Celik. Toward a smart cloud: A review of fault-tolerance methods in cloud systems. *IEEE Transactions on Services Computing*, 14(2):589–605, Mar. 2018.

[9] Tianzhu Zhang, Han Qiu, Marco Mellia, Yuanjie Li, Hewu Li, and Ke Xu. Interpreting AI for networking: Where we are and where we are going. *IEEE Communications Magazine*, 60(2): 25–31, Feb. 2022.

[10] Gerhard Widmer and Miroslav Kubat. Learning in the presence of concept drift and hidden contexts. *Machine learning*, 23(1):69–101, Nov. 1996.

[11] Jean Paul Barddal, Heitor Murilo Gomes, Fabrício Enembreck, and Bernhard Pfahringer. A survey on feature drift adaptation: Definition, benchmark, challenges and future directions. *Journal of Systems and Software*, 127:278–294, May 2017.

[12] Zaheer Khan, Janne Lehtomäki, Adnan Shahid, and Ingrid 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), pages 1290–1291, Jul. 2020.

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

[14] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, Joao Gama, and Guangquan Zhang. Learning under concept drift: A review. *IEEE Transactions on Knowledge and Data Engineering*, 31(12): 2346–2363, Oct. 2018.

[15] Yang Syu, Chien-Min Wang, and Yong-Yi Fanjiang. Modeling and forecasting of time-aware dynamic QoS attributes for cloud services. *IEEE Transactions on Network and Service Management*, 16(1):56–71, Dec. 2018.

[16] Peter Mell, Tim Grance, et al. The nist definition of cloud computing, Sep. 2011.

[17] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, Apr. 2010.

[18] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, Jun. 2016.

[19] Sihem Cherrared, Sofiane Imadali, Eric Fabre, Gregor Gössler, and Imen Grida Ben Yahia. A survey of fault management in network virtualization environments: Challenges and solutions. *IEEE Transactions on Network and Service Management*, 16(4):1537–1551, 2019.

[20] David Jauk, Dai Yang, and Martin Schulz. Predicting faults in high performance computing systems: An in-depth survey of the state-of-the-practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, Nov. 2019.

[21] Tom M Mitchell et al. Machine learning, 1997.

[22] Andrea Rosà, Lydia Y Chen, and Walter Binder. Failure analysis and prediction for big-data systems. *IEEE Transactions on Services Computing*, 10(6):984–998, Mar. 2016.

[23] Chunhong Liu, Jingjing Han, Yanlei Shang, Chuanchang Liu, Bo Cheng, and Junliang Chen. Predicting of job failure in compute cloud based on online extreme learning machine: a comparative study. *IEEE Access*, 5:9359–9368, May 2017.

[24] Li Deng and Dong Yu. Deep learning: methods and applications. *Foundations and trends in signal processing*, 7(3–4):197–387, Jun. 2014.

[25] Jiechao Gao, Haoyu Wang, and Haiying Shen. Task failure prediction in cloud data centers using deep learning. *IEEE Transactions on Services Computing*, May 2020.

[26] Chaoyun Zhang, Paul Patras, and Hamed Haddadi. Deep learning in mobile and wireless networking: A survey. *IEEE Communications surveys & tutorials*, 21(3):2224–2287, Mar. 2019.

[27] Yangguang Li, Zhen Ming Jiang, Heng Li, Ahmed E Hassan, Cheng He, Ruirui Huang, Zhengda Zeng, Mian Wang, and Pinan Chen. Predicting node failures in an ultra-large-scale cloud computing platform: an AIOps solution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(2):1–24, Apr. 2020.

[28] Ji Zhang, Ke Zhou, Ping Huang, Xubin He, Ming Xie, Bin Cheng, Yongguang Ji, and Yinhu Wang. Minority disk failure prediction based on transfer learning in large data centers of heterogeneous disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 31(9): 2155–2169, Apr. 2020.

[29] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach, 2002.

[30] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, May 1992.

[31] Yanal Alahmad, Tariq Daradkeh, and Anjali Agarwal. Proactive failure-aware task scheduling framework for cloud computing. *IEEE Access*, 9:106152–106168, Jul. 2021.

[32] Chang Xu, Gang Wang, Xiaoguang Liu, Dongdong Guo, and Tie-Yan Liu. Health status assessment and failure prediction for hard drives with recurrent neural networks. *IEEE Transactions on Computers*, 65(11):3502–3508, Mar. 2016.

[33] Aniello De Santo, Antonio Galli, Michela Gravina, Vincenzo Moscato, and Giancarlo Sperlì. Deep learning for HDD health assessment: An application based on lstm. *IEEE Transactions on Computers*, 71(1):69–80, Dec. 2020.

[34] Minh-Ngoc Tran, Xuan Tuong Vu, and Younghan Kim. Proactive stateful fault-tolerant system for kubernetes containerized services. *IEEE Access*, 10:102181–102194, 2022.

[35] Lucas Baier, Josua Reimold, and Niklas Kühl. Handling concept drift for predictions in business process mining. In *Proc. IEEE on Business Informatics (CBI)*, pages 76–83, Antwerp, Belgium, Jul. 2020.

[36] Anjin Liu, Guangquan Zhang, Kun Wang, and Jie Lu. Fast switch naïve bayes to avoid redundant update for concept drift learning. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7, Jul. 2020.

[37] Bilge Celik and Joaquin Vanschoren. Adaptation strategies for automated machine learning on evolving data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Mar. 2021.

[38] Fan Dong, Jie Lu, Yiliao Song, Feng Liu, and Guangquan Zhang. A drift region-based data sample filtering method. *IEEE Transactions on Cybernetics*, Feb. 2021.

[39] Yu Sun, Ke Tang, Zexuan Zhu, and Xin Yao. Concept drift adaptation by exploiting historical knowledge. *IEEE Transactions on Neural Networks and Learning Systems*, 29(10):4822–4832, Feb. 2018.

[40] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems*, pages 3320–3328, Dec. 2014.

[41] Sakti Saurav, Pankaj Malhotra, Vishnu TV, Narendhar Gugulothu, Lovekesh Vig, Puneet Agarwal, and Gautam 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*, pages 78–87, Jan. 2018.

[42] Simone Disabato and Manuel Roveri. Learning convolutional neural networks in presence of concept drift. In *Proc. IEEE International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, Jun. 2019.

[43] Mohammad Navid Fekri, Harsh Patel, Katarina Grolinger, and Vinay Sharma. Deep learning for load forecasting with smart meter data: Online adaptive recurrent neural network. *Elsevier Applied Energy*, 282:116177, Jan. 2021.

[44] Jean Paul Barddal, Heitor Murilo Gomes, Jones Granatyr, Alceu de Souza Britto, and Fabrício Enembreck. Overcoming feature drifts via dynamic feature weighted k-nearest neighbor learning. In *Proc. IEEE International Conference on Pattern Recognition (ICPR)*, pages 2186–2191, Dec. 2016.

[45] Conor Fahy and Shengxiang Yang. Dynamic feature selection for clustering high dimensional data streams. *IEEE Access*, 7:127128–127140, Jul. 2019.

[46] Boshra Pishgoo, Ahmad Akbari Azirani, and Bijan Raahemi. A dynamic feature selection and

intelligent model serving for hybrid batch-stream processing. *Knowledge-Based Systems*, 256: 109749, Nov. 2022.

[47] Jorge C Chamby-Diaz, Mariana Recamonde-Mendoza, and Ana LC Bazzan. Dynamic correlation-based feature selection for feature drifts in data streams. In *Proc. IEEE Brazilian Conference on Intelligent Systems (BRACIS)*, pages 198–203, 2019.

[48] Albert Bifet and Ricard Gavalda. Learning from time-changing data with adaptive windowing. In *Proc. SIAM international conference on data mining*, pages 443–448, 2007.

[49] Shaaban Sahmoud and Haluk Rahmi Topcuoglu. A general framework based on dynamic multi-objective evolutionary algorithms for handling feature drifts on data streams. *Future Generation Computer Systems*, 102:42–52, Jan. 2020.

[50] Shaaban Sahmoud and Haluk Rahmi Topcuoglu. Memory-assisted dynamic multi-objective evolutionary algorithm for feature drift problem. In *Proc. IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, Jul. 2020.

[51] Qi Chen, Yubo Song, Brendan Jennings, Fan Zhang, Bin Xiao, and Shang Gao. Iot-id: robust iot device identification based on feature drift adaptation. In *Proc. IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, Dec. 2021.

[52] Kai-Chun Liu, Chia-Yeh Hsieh, Steen Jun-Ping Hsu, and Chia-Tai Chan. Impact of sampling rate on wearable-based fall detection systems based on machine learning models. *IEEE Sensors Journal*, 18(23):9882–9890, Sep. 2018.

[53] Daniel Vázquez Pombo, Tuhfe Göçmen, Kaushik Das, and Poul Sørensen. Multi-horizon data-driven wind power forecast: From nowcast to 2 days-ahead. In *Proc. IEEE International Conference on Smart Energy Systems and Technologies (SEST)*, pages 1–6, Sep. 2021.

[54] Steven Gardner, Oleg Golovidov, Joshua Griffin, Patrick Koch, Wayne Thompson, Brett Wujek, and Yan Xu. Constrained multi-objective optimization for automated machine learning. In *IEEE Proc. on data science and advanced analytics (DSAA)*, pages 364–373, 2019.

[55] Mohammad Loni, Sima Sinaei, Ali Zoljodi, Masoud Daneshtalab, and Mikael Sjödin. Deepmaker: A multi-objective optimization framework for deep neural networks in embedded systems. *Microprocessors and Microsystems*, 73:102989, Mar. 2020.

[56] Yao Yang, Andrew Nam, Mohamad Nasr-Azadani, and Teresa Tung. Resource-aware pareto-optimal automated machine learning platform. In *Proc. IEEE international seminar on research of information technology and intelligent systems (ISRITI)*, pages 1–6, Dec. 2020.

[57] Martin Binder, Julia Moosbauer, Janek Thomas, and Bernd Bischl. Multi-objective hyperparameter tuning and feature selection using filter ensembles. In *ACM Proc. on genetic and evolutionary computation conference*, pages 471–479, Jun. 2020.

[58] Ritam Guha, Wei Ao, Stephen Kelly, Vishnu Boddeti, Erik Goodman, Wolfgang Banzhaf, and Kalyanmoy Deb. Moaz: A multi-objective automl-zero framework. In *Proc. ACM Genetic and Evolutionary Computation Conference*, pages 485–492, Jul. 2023.

[59] Hadjer Benmeziane, Smail Niar, Hamza Ouarnoughi, and Kaoutar El Maghraoui. Pareto rank surrogate model for hardware-aware neural architecture search. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 267–276, May 2022.

[60] Hadjer Benmeziane, Hamza Ouarnoughi, Kaoutar El Maghraoui, and Smail Niar. Multi-objective hardware-aware neural architecture search with pareto rank-preserving surrogate models. *ACM Transactions on Architecture and Code Optimization*, 20(2):1–21, Apr. 2023.

[61] Raquel Espinosa, Fernando Jiménez, and José Palma. Surrogate-assisted and filter-based multiobjective evolutionary feature selection for deep learning. *IEEE Transactions on Neural Networks and Learning Systems*, Jan. 2023.

[62] Mbarka Soualhia, Chunyan Fu, and Foutse Khomh. Infrastructure fault detection and prediction in edge cloud environments. In *Proc. ACM/IEEE Symposium on Edge Computing*, pages 222–235, Nov. 2019.

[63] Michele Basseville, Igor V Nikiforov, et al. *Detection of abrupt changes: theory and application*, volume 104. prentice Hall Englewood Cliffs, 1993.

[64] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[65] Dimitri P Bertsekas and Athena Scientific. *Convex Optimization Algorithms*. Athena Scientific, 2015.

[66] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015.

[67] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Springer Machine learning*, 8(3-4):293–321, 1992.

[68] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. In *Proc. IEEE International Conference on Learning Representations (ICLR)*, Apr. 2017.

[69] Chi Jin, Zeyuan Allen-Zhu, Sebastien Bubeck, and Michael I Jordan. Is Q-learning provably efficient? In *Proc. International Conference on Neural Information Processing Systems*, pages 4863–4873, Dec. 2018.

[70] Stan's robot shop. `https://github.com/instana/robot-shop`. Accessed May 2024.

[71] Prometheus. `https://prometheus.io`. Accessed May 2024.

[72] Stress-ng. `https://wiki.ubuntu.com/Kernel/Reference/stress-ng`. Accessed May 2024.

[73] Alessio Netti, Zeynep Kiziltan, Ozalp Babaoglu, Alina Sîrbu, Andrea Bartolini, and Andrea Borghesi. A machine learning approach to online fault classification in HPC systems. *Elsevier Future Generation Computer Systems*, 110:1009–1022, Sep. 2020.

[74] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. Gene selection for cancer classification using support vector machines. *Springer Machine Learning*, 46(1-3): 389–422, Jan. 2002.

[75] OpenAI Gym. `https://gym.openai.com`. Accessed May 2024.

[76] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in Neural Information Processing Systems*, 24:2546–2554, Dec. 2011.

[77] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proc. IEEE International Conference on Learning Representations (ICLR)*, May 2015.

[78] Timothy Dozat. Incorporating Nesterov momentum into Adam. In *Proc. IEEE International Conference on Learning Representations (ICLR)*, May 2016.

[79] Sana Tonekaboni, Shalmali Joshi, Kieran Campbell, David K Duvenaud, and Anna Goldenberg. What went wrong and when? instance-wise feature importance for time-series black-box models. *Advances in Neural Information Processing Systems*, 33:799–809, 2020.

[80] Sock shop application. https://microservices-demo.github.io. Accessed May 2024.
[81] Locust. https://locust.io. Accessed May 2024.
[82] Linux traffic control. https://https://manpages.ubuntu.com/tc. Accessed May 2024.
[83] Yu-Wei Chan, Halim Fathoni, Hao-Yi Yen, and Chao-Tung Yang. Implementation of a cluster-based heterogeneous edge computing system for resource monitoring and performance evaluation. *IEEE Access*, 10:38458–38471, Jan. 2022.
[84] Olaonipekun Oluwafemi Erunkulu, Adamu Murtala Zungeru, Caspar K Lebekwe, Modisa Mosalaosi, and Joseph M Chuma. 5g mobile communication applications: A survey and comparison of use cases. *IEEE Access*, 9:97251–97295, Jun. 2021.
[85] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, 2001.
[86] Jian-Yu Li, Zhi-Hui Zhan, and Jun Zhang. Evolutionary computation for expensive optimization: A survey. *Machine Intelligence Research*, 19(1):3–23, Feb. 2022.
[87] Open5gs. https://open5gs.org/. Accessed May 2024.
[88] Gradient 5g charts. https://github.com/Gradiant/5g-charts. Accessed May 2024.
[89] Ueransim. https://github.com/aligungr/UERANSIM. Accessed May 2024.
[90] Joseph Lee Rodgers and W Alan Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42(1):59–66, Jun. 1988.
[91] Chenhua Shen. Analysis of detrended time-lagged cross-correlation between two nonstationary time series. *Physics Letters A*, 379(7):680–687, Mar. 2015.