Infrastructure Fault Detection and Prediction in Edge Cloud Environments

Mbarka Soualhia
Polytechnique Montréal, Canada
mbarka.soualhia@polymtl.ca

Chunyan Fu
Ericsson Research Canada, Canada
chunyan.fu@ericsson.com

Foutse Khomh
Polytechnique Montréal, Canada
foutse.khomh@polymtl.ca

ABSTRACT

As an emerging 5G system component, edge cloud becomes one of the key enablers to provide services such us mission critical, IoT and content delivery applications. However, because of limited fail-over mechanisms in edge clouds, faults (*e.g.,* CPU or HDD faults) are highly undesirable. When infrastructure faults occur in edge clouds, they can accumulate and propagate; leading to severe degradation of system and application performance. It is therefore crucial to identify these faults early on and mitigate them. In this paper, we propose a framework to detect and predict several faults at infrastructure-level of edge clouds using supervised machine learning and statistical techniques. The proposed framework is composed of three main components responsible for: (1) data pre-processing, (2) fault detection, and (3) fault prediction. The results show that the framework allows to timely detect and predict several faults online. For instance, using Support Vector Machine (SVM), Random Forest(RF) and Neural Network(NN)models, the framework is able to detect non-fatal CPU and HDD overload faults with an F1 score of more than 95%. For the prediction, the Convolutional Neural Network (CNN) and Long Short Term Memory (LSTM) have comparable accuracy at 96.47% vs. 96.88% for CPU-overload fault and 85.52% vs. 88.73% for network fault.

ACM Reference Format:
Mbarka Soualhia, Chunyan Fu, and Foutse Khomh. 2019. Infrastructure Fault Detection and Prediction in Edge Cloud Environments. In *The Fourth ACM/IEEE Symposium on Edge Computing (SEC 2019), November 7–9, 2019, Arlington, VA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3318216.3363305

SEC 2019, November 7–9, 2019, Arlington, VA, USA

ACM ISBN 978-1-4503-6733-2/19/11…$15.00
https://doi.org/10.1145/3318216.3363305

1 INTRODUCTION

Edge cloud has emerged as a key technology used in several areas including 5G mission critical applications, IoT, e-commerce services, autonomous vehicles and content delivery networks [28]. System monitoring is a common activity to ensure good quality of service for deployed applications in edge clouds. However, edge cloud servers and devices are prone to outages and faults. (*e.g.,* CPU or HDD faults). And once the faults occur, they can have undesirable impact on the deployed applications and may lead to a severe performance degradation on the entire cloud system. In fact, an infrastructure fault that happens at one node of an edge cloud, if not handled properly, may propagate. It may spread up towards applications or propagate across nodes in the same cluster of the edge cloud. Once spread, it can be hard to trace the root cause of the fault, and thus causes extra costs and delays in fault recovery. For instance, Google experienced system performance degradation due to an additional delay of 500 ms to the response time of some applications. This performance degradation caused a revenue loss of 20 % [14]. Applications at an edge cloud can often suffer from such performance degradation due to an unwanted communication latency or a slow computation speed at the infrastructure layer. In addition, the existing fault-tolerance mechanisms in a centralized cloud are often resource consuming and they cannot be easily applied to the edge. Contrary to the centralized cloud, the resources are constrained and limited in the edge due to the small processors and limited power capacity [19].

Although the built-in recovery mechanisms in edge cloud environments such as regular check-pointing and replication exist, a single fault recovery procedure can take a long time due to again the limited resources. Moreover, the workload on the edge often changes drastically and thus the built-in recovery mechanisms will compete with the running workload for these limited resources. Therefore, we believe that it is crucial to have mechanisms that can timely identify the occurrence of the infrastructure faults to quickly recover from them. Furthermore, we believe that if we can early identify circumstances leading to those faults, we can improve the system performance by preventing them on the fly. However, given the dynamic and heterogeneous nature of edge cloud environments, identification of those faults poses several

challenges such as fault types, recovery procedures and fault management costs. Anomaly detection is a well-established research domain with many theoretical and practical solutions in the literature. In current practice, there exist several techniques for fault identification in cloud environments including the use of test suites, performance metrics, system logs and machine learning techniques [12] [23] [29].

A typical approach to detect faults is the use of automated test suites to check the functionality of a service/software by testing whether it meets the specification or not (for example, those included in OpenStack Tempest test plugins [11] [21]). However, such an approach is often very expensive in terms of computation time, and it would require extra resources from the cloud system that would have been allocated or reserved to serve the users' requests. Running those test suites at the edge may negatively impact its performance because of the potential resource contentions.

Monitoring an edge cloud often requires controlling the states of the system over time by collecting performance metrics related to the status of the system (*e.g.,* CPU utilization, memory usages and IO error). For example, an alarm program can be put in place to control the average CPU utilization such that if it is greater than 90% over a specific time an alarm can be raised. There exist different studies [15, 26, 29, 30] that have been proposed in the literature and are in use today to detect faults based on checking if some metrics reach specific thresholds. In fact, they try to identify system symptoms that characterize the presence of faults. In general, the use of performance indicators is often more lightweight when compared to running test suites to detect faults symptom. The reason is that executing the test scenarios should be done periodically to regularly detect those faults. Hence, they can add a significant overhead to the applications, while in the metrics-based approach, only the filtered metrics will be monitored over time, which is more lightweight. However, the metrics-based method requires a lot of manual configuration and knowledge about the metrics that have a direct impact on the system faults.

In current practice of monitoring cloud systems, the output logs generated is one of the important information sources about the existence of faults in the system. A very simple way to detect faults is to define keywords that relate to the existence of faults (such as failure, fatal, error and warning). Existing literature is very rich in fault detection based on information extracted from the logs [12, 13, 23]. One key limitation of these techniques is that one should have an idea about the most common faults that a system may experience and their corresponding keywords. However, not all faults have attributes that can be logged. Moreover, log files are of noisy and lack information about system state changes over time.

Another approach to detect faults is to use machine learning techniques to identify abnormal behavior of the system according to collected data from the monitored cloud infrastructure [16, 23, 31]. The machine learning models are trained with various performance metrics to identify possible dependencies between the metrics and the fault occurrences. They are particularly useful when identifying faults that are difficult to detect, and the metrics related to these faults are hard to monitor. This requires a model designers to know that a specific data point is correspondent to a fault or normal behavior, so that they can label the data and train the models. This can usually be obtained from the managed system when an alarm is generated, in the presence of a fault, at the data collection time. Most of the published machine learning-driven models [23, 31] include only an offline evaluation of the proposed fault detection algorithms, but they do not measure the impact of the trained models when integrated on a real time cloud system. In fact, the online evaluation is a necessary step to demonstrate the effectiveness of these models to capture system changes that may lead to faults. In addition, the trained models were not evaluated in an edge environment with resource constraint devices. Moreover, none of those works include a method to predict fault occurrence in such environments to prevent it from happening.

Considering the above facts, it poses a critical and open challenge to design efficient detection mechanisms to deal with dynamic and heterogeneous edge cloud environments. Hence, it is of interest to design a timely and efficient fault detection and prediction functionality for such management systems. In this paper, we propose a framework to detect and predict several faults at infrastructure-level of edge clouds using supervised machine learning and statistical techniques. The proposed framework is composed of three main components responsible for: (1) data pre-processing, (2) fault detection and (3) fault prediction. For data pre-processing, we use Random Forest (RF) classifier and Recursive Feature Elimination (RFE) classifier to select the most relevant features. For fault detection, we apply various models including RF, Support Vector Machine (SVM), Neural Network (NN), AutoRegressive Integrated Moving Average (ARIMA), Markov Chain (MC), Bayesian Network (BN) and Tree-Augmented Naive Bayes (TAN). For fault prediction, we use deep learning techniques including Long Short-Term Memory (LSTM) and Convolutional Neural Network (CNN) models. We implement and test the proposed framework on an edge cloud testbed and a resource constraints edge device (Raspberry Pi) and we perform a case study to show its effectiveness. The experimental results show that the framework allows to timely detect and predict several faults online. In addition, it allows to detect and predict faults at infrastructure layer using the data collected from an upper layer to help avoiding the propagation of the fault by early finding the

appropriate mitigation procedures (*e.g.,* reschedule applications from a faulty node to a healthy one). Moreover, it is possible to detect faults experienced by cluster slaves in edge clouds through the cluster master to early perform recovery mechanisms and reschedule the workload accordingly. To achieve those goals, our proposed framework is using contextual and behavioral information from its environments. Our proposed framework shows good results for detection and prediction of several faults. For instance, using SVM, RF and NN models, the framework is able to detect non-fatal CPU and HDD overload faults with an F1-Score of more than 95%. For the prediction, both CNN and LSTM have comparable accuracy 96.47% vs. 96.88% for CPU-overload fault and 85.52% vs. 88.73% for network fault.

The remainder of this paper is organized as follows: Section 2 presents our proposed framework for fault detection and prediction. Section 3 describes our case study and discusses the obtained results. Section 4 discusses threats to the validity of our work. Section 5 summarizes the related literature, and Section 6 concludes the paper and outlines some prospects for future works.

2 PROPOSED FRAMEWORK

2.1 General Overview

In this section, we present the architecture of our proposed framework that uses the data from the edge cloud environment to 1) detect faults (*e.g.,* CPU, Packet Loss (PL), and HDD faults) when they happened, and 2) predict faults in advance before they actually happen, using machine learning techniques. Early detection of faults can help cloud providers applying remedy solutions and thus to reduce the possibility of a fault propagating through the cloud. Predictions can further help the providers proactively preventing these faults before they happen. The overview of the methodology is as follows. First, we collect time series data from the monitored cloud system in order to get information about the system status. Next, we analyze the collected data and extract the most relevant ones that have an impact on the fault occurrence. Finally we apply machine learning and statistical techniques to build fault detection and prediction models. Figure 1 gives an overview of the structure of our framework. It is comprised of three main components: "Data Pre-Processing", "Fault Detection", and "Fault Prediction". The remainder of this section elaborates more on each of these components.

2.2 Detailed Description

2.2.1 Data Pre-Processing. This component is responsible for collecting time series data from the monitored edge cloud environment. Next, it transforms the raw data into the correct format for machine learning. This includes data format

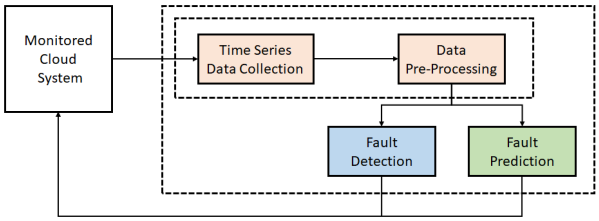


Figure 1: Architecture of the Proposed Framework

translation, missing sample interpolation and data normalization and labeling. After that, it identifies the features that are likely the most important ones reflecting the changes in the presence of a fault in edge cloud environments. In fact, the quality of features used to train a given model has a huge impact on the performance a model can achieve (*e.g.,* accuracy and time). Various techniques can be used for this purpose. One technique would be identifying possible relationships/connections between metrics (*i.e.,* features) and the system status (*e.g.,* fault or no fault) in order to select the most relevant ones. This allows to reduce the complexity of the model training and its associated execution time. Specifically, we used a RF classifier [7] and a RFE classifier [8] for selecting features. Indeed, they have been widely used for feature selection as they come with a built-in modules that measure the importance of features. So, they can help selecting only the features having a direct impact on the model output. The Pre-Processing component is preliminary to fault detection and prediction components.

2.2.2 Fault Detection. This component is responsible for the identification of symptoms/circumstances in the edge cloud that may lead to the occurrence of a fault. This is based on the data of the selected features from the previous step. Specifically, our goal is to detect several infrastructure-level faults including the CPU overload, HDD overload, and PL in edge clouds. Our framework aims at detecting the faults at both hosts and virtual machines (VMs) using their corresponding data. In addition to detecting faults directly from the initial faulty node, the framework also supports detecting faults that occur at a cluster slave node from the master node. This is since some faults experienced by the slave nodes affect metrics at both the slave and the master nodes. Our framework filters those affected features from the master to identify potential faults at the slave.

To build the fault detection component, we train several machine learning, probabilistic and statistical models and explore their capabilities to identify a potential system fault based on its status (*e.g.,* collected metrics about its performance and usage). Here, we select probabilistic models such as MC, BN and TAN because they are probabilistic classifiers that can determine the probability distribution over a set of classes given an input data. Furthermore, we use statistical

models to implement the proposed component to overcome the uncertainty in machine learning models. This is based on statistics of the input data used to train those models in order to determine the expected system status. For the machine learning models, we choose several regression and classification algorithms including RF, SVM, and NN. On the other hand, we choose a very widely used statistical algorithm ARIMA for detecting faults.

For training the models, we first inject faults and collect time series data on an edge cloud environment. The data were collected over a fixed time interval. Next, we measure and compare the performance of those models in terms of accuracy and F1 score when applying 10-fold random cross validation method [18]. In the cross validation process, the collected data are randomly split into ten folds where nine folds serve as a training data set and the rest serves as the testing set while evaluating the model performance.

2.2.3 Fault Prediction. Using this component in our framework, we aim to explore the possibility of predicting a potential fault in advance based on the collected time series data and some deep learning techniques. We believe that if we can early identify the occurrence of faults, we can prevent several system failures from happening. Anticipating those faults can help preventing several performance issues by, for examples, early notifying the scheduler to make adaptive scheduling decisions or to adjust the workload balancing to avoid application performance degradation. In our framework, we also address a multi-step prediction that allows to timely identify the faults in several time steps advance. For instance, the prediction can be made according to the sampling window used to collect the data from the edge cloud (*e.g.,* 10 s and 20 s.). Furthermore, our framework supports prediction of faults through different layers. For instance, it allows to use data from the container layer to make predictions on faults that a host can experience.

For the predictive models, we choose to apply deep learning techniques (LSTM and CNN) to determine whether a fault will occur or not, given an input data. Recurrent Neural Networks (RNN) are used in several prediction problems since they work well with sequence data and learn the changes of input features. LSTM is one of the most useful RNNs while CNN represents another class of deep learning that can be used for prediction problems. It is generally used on data that has an ordered relationship in the time steps of a time series. Therefore, we apply the LSTM and CNN models to perform the prediction of faults in our framework. To do the time-series prediction, the moving window technique is used [20].

We train the prediction models with data representing the state of the system and their corresponding "labels" to forecast the next state of the system. In other terms, the inputs of the machine learning model are the values of the identified "predictors" and the output is the "system status" (either fault or no-fault). The trained model will learn the changes of the system over time from the sequence of input data. We use a 1 hour to 7 days data to train the models. Similar to the detection component, we evaluate the obtained results for the tested models in terms of accuracy.

3 EXPERIMENTAL RESULTS

In this section, we present the obtained results when implementing our proposed framework on an edge cloud environment.

3.1 Experimental Setup

3.1.1 Lab setup. Figure 2 shows the lab setup where we conduct the experiments. There are 7 HP servers simulating an edge computing environment (*e.g.,* edge sites). The servers that represent edge sites are running Ubuntu 16.04 with two types of setups: Kubernetes on VMs (provisioned by Kolla OpenStack version Rocky) and Kubernetes on hosts, with a total of four Kubernetes clusters. We install Kubefed-v2 [4] for Kubernetes federation. Weave net and core-dns are used for the networking. We also deploy a couple of Raspberry Pis (Model 3B) with Ubuntu MATE installed for simulating edge devices.

3.1.2 Data collection: We use Prometheus [5] for monitoring and data collection. Prometheus is an open-source system monitoring and alerting toolkit that has been adopted by many companies and organizations. We installed it one instance per Kubernetes cluster and per Raspberry Pi. Prometheus uses scrapers for collecting data. We install the node-exporter scraper for node-level (*e.g.,* CPU, memory, I/O, network) statistics. In the "Kubernetes on hosts" setup (*i.e.,* Cloud 1, 2 and 4), the node-level data reflects the physical host statistics. While on the "Kubernetes on VMs" setup (*i.e.,* Cloud 3), the node-level data reflects the VM statistics. We also install cAdvisor for container statistics. The data collection rate is set to 10 seconds for hosts/VMs and 30 seconds for Raspberry Pis.

3.1.3 Injected Faults: We inject two types of faults: recurrent faults and accumulative faults. For the recurrent faults, we have experimented with CPU stress, HDD stress (using stress-ng [9]) and PL (using Linux Traffic Control [10]). We define a recurrent fault as follows: 1) a fault is injected for 60 seconds and 2) the system is cooling down for 180 seconds, 3) repeat steps 1) and 2) till total testing times out. For the accumulative faults, we have tested accumulative memory stress (using stress-ng), HDD stress and network flooding (include TCP flooding and Ping flooding). The accumulative faults are the ones that grow in a few steps over time (4 steps for

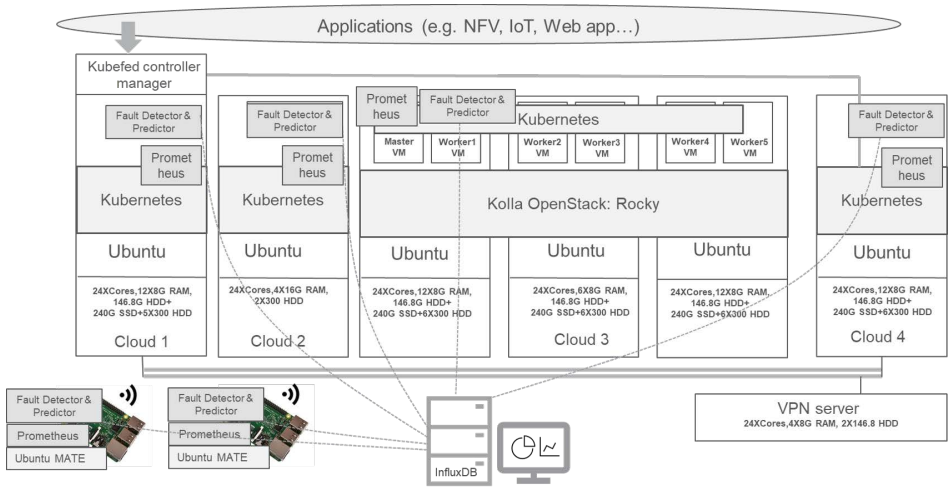


Figure 2: Lab Setup

hosts/VMs, and 2 steps for Raspberry Pis), where each step lasts 60 seconds. For example, we stress 20%, 40%, 60%, and 80% of the total host memory or we send pings with the intervals of 0.2 second, 0.1 second, 0.05 second and finally in flooding mode. A cool down time of 240 seconds is set between two fault injections.

3.1.4 Data Labeling: Labels are manually added to the collected data based on the fault injection timestamps. The data for recurrent faults is labeled as two classes: 0 as non-fault and 1 as fault. The accumulative fault data is labeled as 5 classes: 0 as non-fault, and 1-4 as the 4 levels of faults. The data of accumulative Pi faults is labeled as 3 classes: 0 as non-fault, and 1-2 as the 2 levels of faults.

3.1.5 Coding tools: We use *Python 3.6* on hosts and VMs and python 2.7 on Raspberry Pis for coding. The detection and prediction models are built using *Python Scikit-learn* and *Keras* (using *Tensor-Flow*) libraries.

3.1.6 Machine Learning Models: We experiment with various models for fault detection and prediction. For recurrent fault detection, we use RF, SVM, feedforward NN, MC, BN, TAN and ARIMA. For a recurrent fault prediction, we use LSTM, CNN and a combined CNN and LSTM network. For an accumulative fault prediction, we use LSTM, multi-channel CNN and multi-headed CNN for multi-step predictions. Model parameters are tuned while we do the experiments. In order to reduce the data transmission cost, models are trained in priority on the same host of the data collection point, *i.e.,* on each Kubernetes master. For the Raspberry Pis, Cloud 1 is used for their model training since they do not have enough resources to train the prediction models. For example, we got CPU overload errors and Pi overheat warnings when training the LSTM and CNN and it could not be completed. The detailed procedure is as follows. When data is collected, the Pi first uses the RF classifier to select features. It then sends data with only the selected features to the host for model training. Once models are trained, the host sends back the models, and the Pi uses the models to do online prediction. The procedure is repeated when there is new data for model re-training. Figure 3 shows the data flow of the Raspberry Pi. Finally, the online detection and prediction results are stored in a centralized time-series database InfluxDB [6].

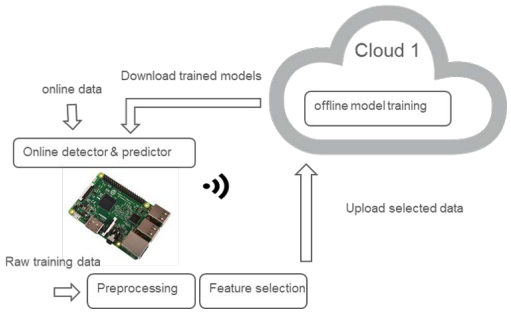


Figure 3: Edge Device Data Flow

3.2 Experimental Results

3.2.1 Data Pre-Processing: The raw data from Prometheus contains both gauges and counters. Since the counters are ever increasing or decreasing, they cannot be directly used in our framework. Therefore, we convert counters into gauges by: (1) replacing the data value with the difference between values of the two sequential data samples (called gauged data) or (2) replacing the values with the difference between the current value and the value from a time interval (*e.g.,* one minute) ago divided by the time interval (called rated data). Furthermore, we notice that some data samples from

Prometheus have all-empty values and hence, we use a one-dimensional linear interpolation to impute the data.
While collecting the data from Prometheus, we find out that the node-level data contains (a) around 1200 metrics of the host, (b) around 700 metrics of the VM and (c) around 470 metrics of the Raspberry Pi. On the other hand, the container-level contains data with 4000-10000 features. We observe that the feature dimensions are too high for detection and prediction models to perform. Consequently, we experiment with four approaches to do feature reduction and selection. First, we try non-changing elimination. That is, features that do not change with and without faults may not help in fault detection or prediction. Therefore, we eliminate those features from the training data. Table 1 shows the number of features reduced following this method with two data sets.

Features	Before	After
Node-level data sample	1251	337
Container-level data sample	9982	733

Table 1: Non-Changing Feature Elimination (Host Data)

As a second step, we eliminate highly correlated features to remove the redundant information on the features values. To do that, we first reorder the features based on their importance, using feature importance returned by a common used decision tree classifier: ExtraTreeClassifier. We then use the Pearson and Spearman to do correlation analysis and remove the highly (>0.75 or <-0.75) correlated features.

Third, we perform non-stationary elimination. Statistical methods such as ARIMA convert non-stationary time-series into stationary for prediction. However, we found that some time-series, even after the pre-processing, cannot be converted into stationary data. Removing such time-series can help improving the online prediction accuracy for ARIMA. Thus, we use the Augmented Dickey Fuller (ADF) to do the stationary analysis and remove the non-stationary features.

Finally, we experiment on feature importance-based elimination. Three most used methods are tested: the feature importance from RF classifier, the permutation importance from NN and the Logistic Regression based RFE. They analyze the feature importance, so that the features with less impact on the results can be removed. Among the three methods, we compare their performance to reduce features from 1060 to 10 for a host node-level data set (with 6480 samples). The model training time is first compared and then with the outcome of the 10 features, the 4 hours and 7 days offline CPU fault prediction accuracy (using CNN) are compared.

Table 2 shows the obtained results. We can see that on the same training host (*i.e.,* "Cloud 1" in Figure 2), RF only takes 51.25 seconds to finish the training, while NN takes more than 2 hours and RFE takes more than half an hour. In terms of accuracy, the NN and RFE show a bit better result (*i.e.,* nearly 2% more accurate prediction is achieved). We can argue that RF is a better method for features reduction tasks under computing resource constrained environments, while RFE and NN are better if there are enough computing resources in the system. Thus we use RF in Raspberry Pis. In the other cases, between NN and RFE, we use RFE since its training time is much less than the time taken by the NN method.

(1060 ->10) features	Training Time (s)	6 hours offline accuracy (%)	7 days offline accuracy (%)
RF (10000 trees)	51.25	91.84	96.06
NN (3 layers, epochs=50)	8624.24	93.60	98.08
RFE (with Logistic Regression as model)	2513.96	94.20	97.87

Table 2: Feature Reduction using RF, NN and RFE

Comparing the above-mentioned four approaches, we find that the most useful one is the feature importance-based elimination. This is because it reduces the features based on their correlations on the labels. Following this way, the features that have more impact on the results are selected, so the trained models can achieve a better result in detection and prediction. We observe the shortcomings of the other approaches as follows:

- The non-change removal has a risk of removing some useful features that are not changed in training time but changed in testing time.
- The highly-correlated feature removal removes the features that have impact on results while keeping irrelevant features thus reducing the prediction accuracy.
- The non-stationary elimination can help improving online detection accuracy for statistical methods such as ARIMA, but it may remove useful features for the other models.

3.2.2 Recurrent Fault Detection: We have experimented with seven models for fault detection. We first compared the models' F1 scores for the validation dataset when detecting the faults (*i.e.*, host CPU stress) using different numbers of features when training the models. That helps us deciding the number of features to be used for training each model. With the number of features decided, we compare the model training time, the offline and online F1 scores results for the three recurrent faults (CPU stress, HDD stress, and 40% PL). The offline test is done using 10-fold cross validations, while the online test result is calculated via collecting 1 hour's detection results with 10 second detection interval. In the last

part of this section, we will also show the study results for one of the fault propagation scenarios – a fault that happened on a Kubernetes worker node propagating to its cluster master. The CPU stress fault is injected on one of the worker nodes on Cloud 3 (see Figure 2), and we evaluate the detection accuracy using the data collected only from the master node.

(a) Model Performance: Figure 4 presents the seven models' performances when using different number of features obtained using RFE. We tested 1-10 features for ARIMA and 10-90 features for the other models. It can be observed that ARIMA performs best when only 1 feature is used. Its detection F1 score drops >30% when using >4 features. We also test 20 features to train ARIMA and we find out that the model cannot converge in some tries. Thus, we decide to use 1 feature for ARIMA fault detection. This actually makes sense because ARIMA is known for uni-variate predictions. Even though supported multivariate, it does not perform with a big number of features. For all the other models we start from 10 features and we find that more than 10 features do not help much enhancing the model accuracy. For example, only < 3% of F1 score increment achieved for NN when increasing the number of features from 10 to 40. However, there is no significant F1 score differences when using 10 features or 90 features for the other models for CPU fault detection. Thus, we decide to use 10 features for all the other models for fault detection.

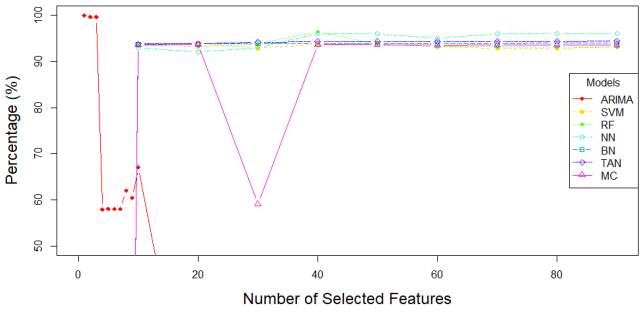


Figure 4: Offline Recurrent CPU Fault Detection F1 scores

(b) Model Training Time and Results: We further compare the training time when using 10 features (except for ARIMA using 1 feature), 15 hours of data set. Table 3 shows the obtained results. We observe that NN is the most time-consuming model for training, followed by the BN/TAN in the second place. SVM is the most lightweight one in terms of the training time.

Figure 5 presents the offline fault detection F1 score results. We find that ARIMA performs best (100%) for the CPU fault because it uses the one feature that reflects changes when the fault is injected. The other models obtained around 93% of F1 score. For the HDD fault, RF and NN performs best with 99%, followed by the SVM and ARIMA with 98%. The BN, TAN and MC performs worse than the other models for the fault. This is because when the HDD fault is injected, Prometheus often lost data samples. Since BN and TAN are built based on the states and transitions between relative states, the missing sample will affect the accuracy of their detection. For the PL fault, the BN and TAN performs best with 75%, followed by the RF, MC and NN with 68% and the SVM and ARIMA at <65%. We can see that all the models in general are less performant in detecting VM PL compared to the F1 scores of the other two faults. The reason is due to the bigger drift (*e.g.,* changes on the statistical properties for the used data values) of the VM data. To alleviate this issue, possible solutions are 1) to use a longer training set and 2) to use techniques such as transferred learning or incremental learning. We will explore enhancing the VM fault detection accuracy in future work. Figure 6 shows the online results. We can see that the results are comparable to the offline results for the host CPU fault and VM PL. Similarly, the BN and TAN do not perform as good as the other models for HDD fault.

Model	RF	SVM	NN	MC	BN/TAN	ARIMA
Training Time (s)	0.119	0.006	239.934	21.392	206.349	0.401

Table 3: Models' Training Times Comparison

(c) Detecting Worker's Fault from Kubernetes Master: Table 4 shows the offline and online results of the detection accuracy. From the experiment, we can claim that our framework is able to detect worker's fault from a Kubernetes master node using RF and NN. We see that the two models can achieve > 80% online detection accuracy. We have also tested with the other models and they performed with < 10% accuracy.

Accuracy (%)	RF	NN
Offline	83.88	74.31
Online	84.47	87.0

Table 4: Detection Accuracy for CPU Fault From Master

3.2.3 Recurrent Fault Prediction. For the prediction of faults, we use neural networks since they are generally more capable in executing multi-variate predictions. In this section, we present the online and offline results obtained using LSTM, CNN and CNN-LSTM for the three types of recurrent faults. We measure the training time and the offline and online one-step prediction accuracy when using the number of selected

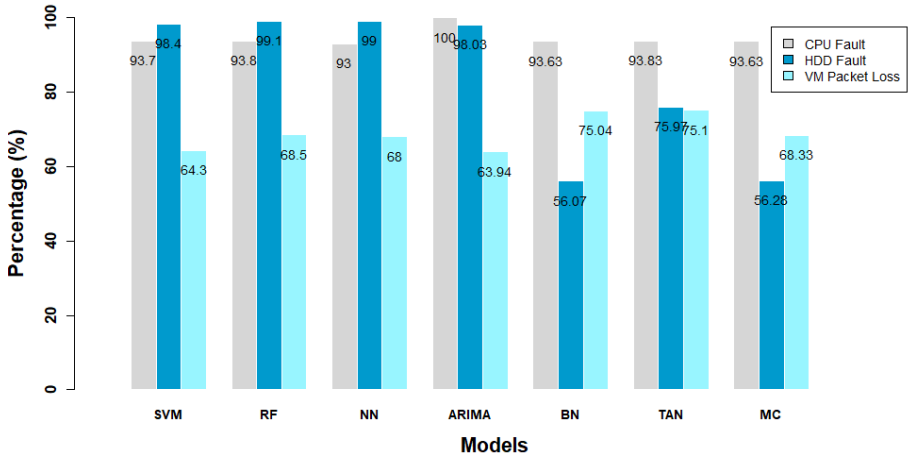


Figure 5: Offline Recurrent Fault Detection F1 score

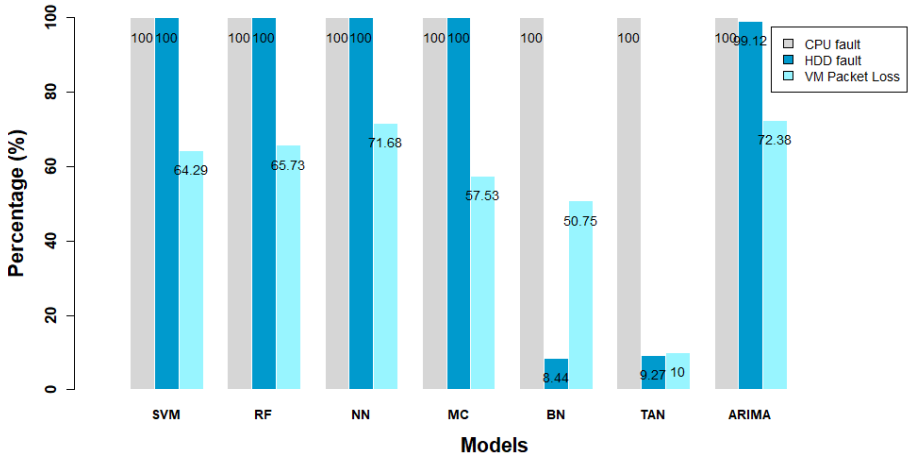


Figure 6: Online Recurrent Fault Detection F1 score

features. We also test the CPU fault prediction on Raspberry Pis and measure offline and online accuracy. In the last part of this section, we will describe the obtained result of another fault propagation scenario, where explore the possibility to predict host CPU fault using container-level data.

(a) Host and VM Recurrent Fault Prediction: Both LSTM and CNN are often used for time-series prediction. People also combine the two models in some circumstances to achieve better performance. From the experiments on training the 3 models, we find that the most time-consuming model is the LSTM. CNN only takes <10% of the training time of the LSTM. Table 5 shows the example of training a 24-hour data set.

Model	LSTM	CNN	LSTM-CNN
Training Time (s)	252.79	24.87	128.33

Table 5: Training Time Comparison on 24 hours Data Set

When evaluating the offline prediction, the training data set is split to the 75% and 25% for training and testing respectively. The input window is set to 24 steps and output to 1. Figure 7 shows the offline prediction results in terms of the number of features selected by RFE. We find that there is not much difference to use 10 features or 80 features for predicting host and VM CPU faults (See Figure 7a and 7b). For the host PL, the result is a little bit better (around 3% accuracy enhancement) when using 20 features than when

using 10 features. There is no much difference among using 20 to 90 features (see Figure 7c). For the HDD fault, all the models perform best when using 10 features. The prediction accuracy drops 5-20% when using 20 to 90 features (see Figure 7d).

In terms of prediction accuracy, we can observe that the three models have comparable results – where the LSTM performs a little bit better when predicting the VM CPU fault and host PL, CNN is a little better for host CPU fault, and CNN-LSTM outperforms the other models when predicting host HDD fault. The offline prediction accuracy is between 90% to 95% for CPU fault and host PL, while the value is between 60% to 87% for the HDD faults. The reason behind the lower results for HDD is again due to the missing samples caused by the HDD faults.

Figure 8 shows the online prediction results. The host CPU and PL fault prediction accuracy are comparable to the offline ones. The HDD prediction is a little better due to the low number of missing samples during the hour when we collect the results. The VM CPU fault prediction accuracy is (10%-15%) worse than the offline ones, the result is especially bad for LSTM. Similar to the VM PL fault detection result, this is due to the big drift of the VM data.

(b) Raspberry Pi CPU Fault Prediction: We evaluate the offline and online prediction results for our framework on a Raspberry Pis. For the recurrent CPU fault on the Pis, Table 6 presents the accuracy results. The result is lower than the host CPU fault prediction due to significant data drift on some of the features. Further investigation is required for handling the data drifting.

Model	LSTM	CNN	LSTM-CNN
Offline Accuracy (%)	76.01	78.40	78.12
Online Accuracy (%)	77.41	77.84	78.12

Table 6: CPU fault Prediction Results on Raspberry Pis

(c) Predict CPU Fault using Container Data: Predicting node-level fault using higher-level measurement data can help application users early detecting symptoms indicating a performance drop at the infrastructure layer. This is in order to timely relocate or reschedule their applications. We experiment using data collected from containers (via cAdviser scraper) to predict the recurrent CPU faults. Table 7 shows the offline and online testing results. Overall, the 3 models have comparable offline prediction accuracy results (at 91%). For the online testing, instead of using the mean/deviation from the training data set for online data sample normalization as we do for the other online testing, we use a walk-forward mean/deviation. That is, we keep calculating the mean/deviation from the online samples and using the historical values to normalize the current sample. The reason of doing this is because the fast drifting nature of the container data. When we use mean/deviation from the training data, we observe that the normalized data range has significantly change and caused near-zero prediction accuracy. With the walk-forward mean, we could achieve a prediction accuracy at 87.3%.

Model	LSTM	CNN	LSTM-CNN
Offline Accuracy (%)	90.81	91.02	91.36
Online Accuracy (%)	67.20	87.30	74.60

Table 7: CPU Fault Prediction Using Container Data

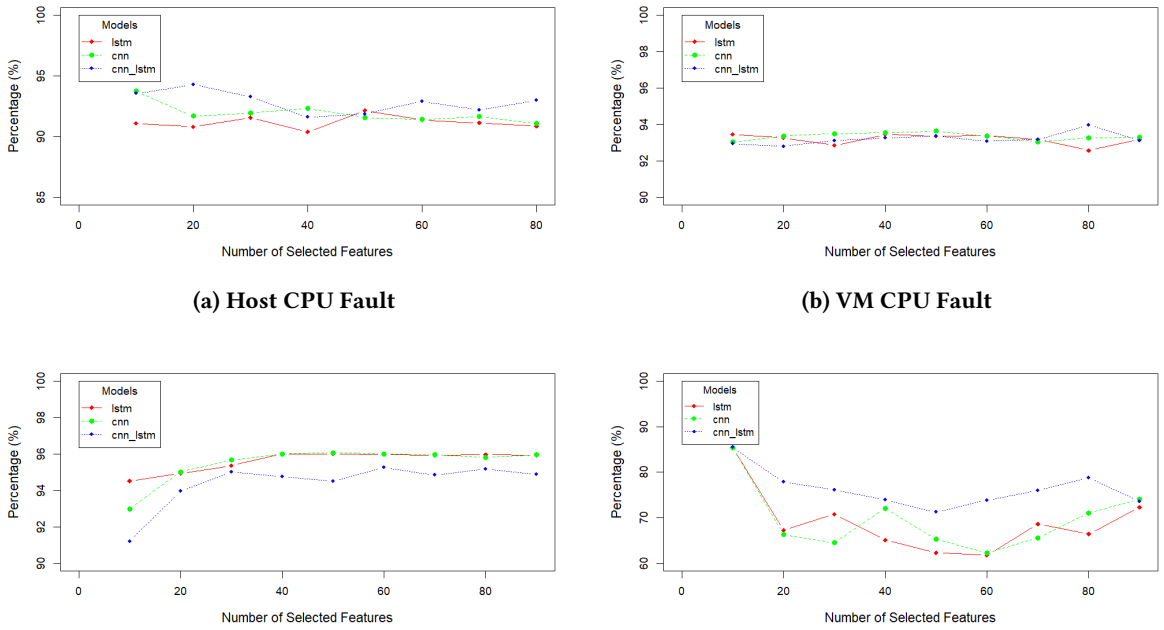
3.2.4 Accumulative Faults Prediction. We use LSTM and CNN for accumulative fault prediction. In the experiments, we compare three models: LSTM, multichannel CNN (MC-CNN) [32] and multi-headed CNN (MH-CNN) [3]. The purpose here is to predict a fault when the first sign of the fault happens. Since a fault is accumulated in 4 steps (240 seconds) in our test, we try to predict the next 24 steps of the collected time series. The 100%, 90%, 80% and 75% of 24 steps accuracy is evaluated, where 100% means that each of the 24 predicted result matches the actual result, and 75% means that 18 of the 24 predicted results matches the actual results. The offline and online prediction results are presented for the three types of the accumulative faults: memory stress, HDD stress and network congestion. In addition, we present the evaluation result for the memory stress fault prediction on Raspberry Pis in the last part of the section.

(a) Host and VM Accumulative Fault Prediction. Table 8 presents the results in terms of time used for training using the 24 hours and 19 hours data sets collected from a host and a Raspberry Pi respectively. Similarly to the previous results, CNN models take much less training time than the time taken by LSTM. There is not much difference between the two CNN models with the given number of features. However, since the input of MH-CNN model is based on the number of features, we also observe that with number of features increasing, the training time of MH-CNN significantly increases too.

Training on Host	LSTM	MC-CNN	MH-CNN
Host data (24h, 10 features)	1635.27	210.25	240.10
Pi data (19h, 4 features)	113.87	26.38	19.76

Table 8: LSTM, MC-CNN and MH-CNN Training Times on 24 Hours Data Set

Figure 9 presents the offline and online prediction accuracy results. For the offline testing, we split the training set into 75% and 25% for training and testing respectively. The input window is set to 48 steps and output to 24. Figures 9a, 9c



(a) Host CPU Fault

(b) VM CPU Fault

(c) Host PL Fault

(d) Host HDD fault

Figure 7: Offline Recurrent Faults Prediction: 1 Step Prediction-Accuracy

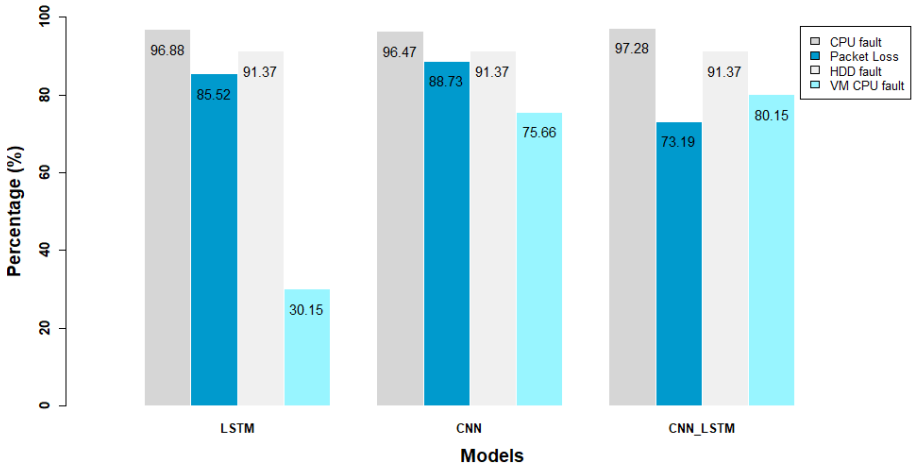


Figure 8: Online Recurrent Faults Prediction: 1 Step Prediction-Accuracy

and 9e show the offline accuracy for the 3 faults. We can find that the models perform best for predicting the memory stress, with 83.43% of 100% 24 step matching and 96.1% of 21 (24 * 90%) step matching. The 24-step matching rate is 45.22% for HDD stress, however, it can achieve 90.58% of accuracy for 21 step matching. Similar to HDD stress, the accuracy for VM network congestion is 45.64% for 24 step matching, and 91.83% of 21 step matching. Comparing the 3 models, the MH-CNN performs better for 24 step prediction of memory stress and network congestion, while MC-CNN performs better in the HDD stress 24 step prediction. Figures 9b, 9d and 9f present the online accuracy results for the 3 types of faults. The online accuracy results are much lower compared to the offline ones. A bit better results are achieved for memory

stress fault, and it reaches 86.57% for the 18 step matching. The HDD stress and VM network congestion accuracy are as low as 19.2% and 38.06% for the 18 step matching. The main reason behind those results is that some features drift from the data range of the training set. Thus, we can conclude that the current prediction models are capable of predicting memory stress. More work needs to be done for HDD stress and network congestion faults.

(b) Raspberry Pi Memory Stress Prediction. We experiment with memory stress faults on Raspberry Pis. As described in Section 3.1.3, the memory is stressed in 2 steps (120 seconds). With a 30 second data sampling rate, we target to predict the next 4 steps. Due to the sample time already causing a delay, we decide to predict 6 steps instead. We first experiment 10 selected features (by the RF classifier) and find out that 6 of them drifted a lot when we did online prediction, which causes a very low (<10%) accuracy for the online predictions. We then manually select 4 features from the 10 and found that the online accuracy is significantly increased. Figure 10 shows the offline and online results for 6 step prediction using 4 features. We can observe 92.43% accuracy in 6 step matching for offline evaluation. Due to a sampling time delay (30s) for online prediction, we observe that the first result of the 6 steps is always out of date, we thus evaluate the online accuracy based on the last 5 steps. We observe an 86.99% accuracy for 5 step matching and 95.66% accuracy for 4 step matching.

4 THREATS TO VALIDITY

4.1 Construct validity threats

Construct validity threats concern the relation between theory and observation. When building the detective and predictive models in our framework, we did not include contextual information about the edge cloud environments. This is because they are changing a lot and are difficult to monitor in edge clouds. However, it is possible that some faults occurred due to changes in the environment where it was detected. Our framework uses the most relevant features collected from the edge cloud that we carefully checked for their impacts by doing offline and online experiments. Furthermore, we observe that some of collected samples from Prometheus were not available when injecting the HDD fault. Those samples were lost because of the nature of the HDD fault that affect the read and write operation to some files. To overcome this issue, we carefully check the consistency of the collected data by mapping the labels to their corresponding data. Hence, this step could help improving the accuracy of the trained models.

4.2 Internal validity threats

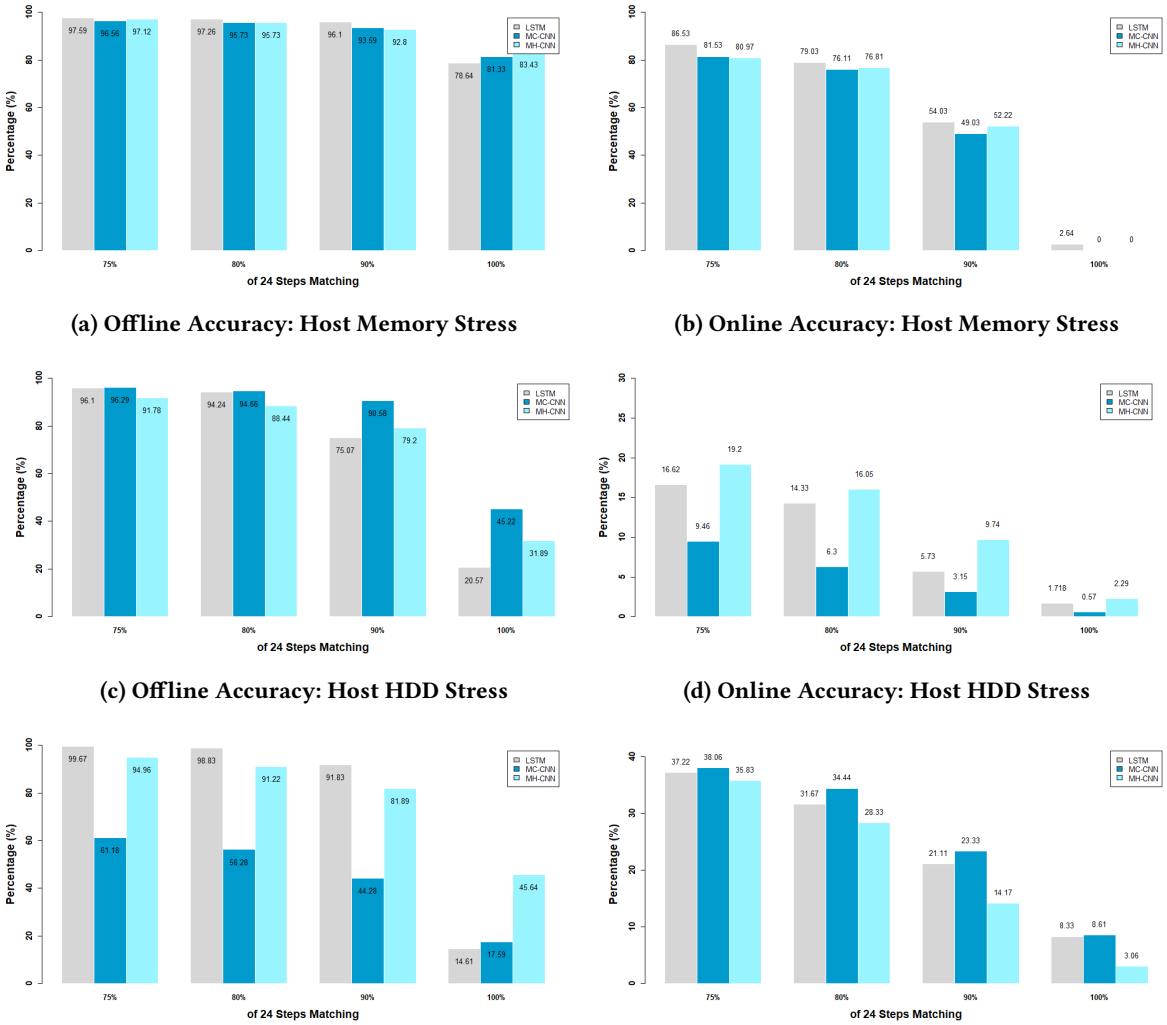
Internal validity threats concern the tools used to implement our proposed framework. We used the commands *stress-ng* [9], *Linux Traffic Control* [10] and *TCP ping flood* to inject different faults at the infrastructure level. We followed two different patterns when injecting the faults: recurrent and accumulative patterns. However, it is possible that the edge cloud does not experience such high rate of faults as those described in Section 3.1.3. Here, our aim is to train the models with sufficient data about the faults to be able to identify the occurrence of faults. Future work should be performed on different fault rates and different edge cloud test beds such as Azure IoT Edge [1] and Google Edge TPU [2]. In addition, the trained data is often changing because of the dynamic behavior of cloud systems and may affect the validity of the models. Therefore, the machine learning models require to be frequently trained over time in order to capture new changes in the cloud system and learn new dependencies between the system metrics and the faults to be detected. We retrain the models over regular intervals to solve the problem of data drift [25] that affect the statistical properties of the used features for the models. Although the retraining phase is time and resource consuming, we carefully check that its overhead and that it does not lead to some faults in the monitored system.

4.3 Conclusion validity threats

Conclusion validity threats concern the relation between the treatment and the outcome. The implemented models are trained with a prior knowledge about the faults (*e.g.,* time of injected faults). This allows us to perform the labeling of the collected data. In fact, performing this step in an edge cloud can be challenging because they require a prior knowledge about the monitored cloud system. Furthermore, the insufficient label data makes it difficult to accurately detect and predict the faults. On the other side, we used RFE to perform the data pre-processing which is an important step to ensure the good quality for the proposed models. We run RFE several times to make sure that the selected features are the most important ones and they have a direct impact on the fault occurrence. Finally, we check the time spent by the proposed algorithms to identify the occurrence of faults in a way that it can timely send the faults results and do not generate extra overhead times (*e.g.,* 2 milliseconds for the detection algorithms).

4.4 Reliability validity threats

Reliability validity threats concern the possibility to replicate our proposed framework. We believe that our proposed framework can be reused on other edge cloud platforms including Azure IoT Edge and Google Edge TPU. To do so,



(a) Offline Accuracy: Host Memory Stress

(b) Online Accuracy: Host Memory Stress

(c) Offline Accuracy: Host HDD Stress

(d) Online Accuracy: Host HDD Stress

(e) Offline Accuracy: VM Network Congestion

(f) Online Accuracy: VM Network Congestion

Figure 9: Offline and Online Accumulative Faults Prediction: 24 Step Prediction Accuracy

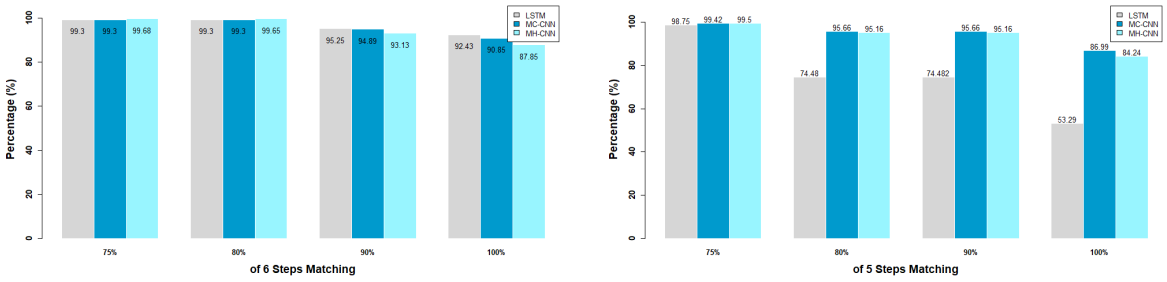
one should collect data from the monitored cloud system to train the proposed predictive and detective models. The new trained models can be unseasoned within other cloud platforms to timely detect and predict faults. On the other hand, we can use unsupervised learning methods when implementing our framework, that have less requirements with respect to system expertise or data labeling. For this purpose, we have to collect large amount of data about the status of the system (where there is no fault). This is due to the fact that the result is highly dependent on the quality of data used to train the unsupervised models. Then we could train the unsupervised models with those data and hence, any outlier to the used data to train the models will be considered as a fault.

4.5 External validity threats

External validity threats concern the generalization of our proposed framework. In fact, further validation on a larger system should be done using different fault rates and several fault types. Furthermore, more tests should be done on resource-constrained environments to show the effectiveness of the proposed models in such systems.

5 RELATED WORK

In this section, we will present the most relevant work that addressed fault detection using machine learning techniques. Gulenko *et al.* [17] describe a framework to detect anomalies in cloud-based infrastructures. The framework is based on indicators using supervised and unsupervised algorithms to



(a) Offline Memory Stress **(b) Online Memory Stress**

Figure 10: Raspberry Pi Offline and Online 6 Step Prediction Accuracy

detect degradation on the system performance in real-time. They conducted experiments on OpenStack to build their proposed framework and the obtained results showed a percentage of correctly identified anomalies. In [31], the authors proposed an approach to mine log files from Hadoop Distributed Files (HDFS) using source code analysis to generate composite features (e.g., state variables and object identifiers). Those features are used then to train machine learning models to identify operational issues. Although the proposed approach could detect some operational issues from the logs, it fails on several cases. This is because it uses a static source code analysis to extract structure from logs and hence, some part of log message will be considered as unparsed string. In [27], authors proposed a method to predict failure in cloud datacenters by learning patterns from the generated messages by identifying similarities between content of messages indicating the presence of failures. The proposed method does not depend on the format of the generated messages from the datacenter. It classifies the messages from the real time system by their similarities to automatically learn message pattern on the fly. The experiments show good prediction results that covered 90% of failure occurrences. The work of Lavin *et al.* [23] described an approach to detect abnormal behaviors of services and hosts in a cloud infrastructure using machine learning techniques. Indeed, they used 13 offline classification algorithms (*e.g. J48, Logistic Model Tree, Hoeffding Tree, Random Tree*, etc.) to build models to detect anomalies on a set of hosts in the cloud. The experimental results showed that good performance in terms of precision and recall for the tested models for the offline anomalies detection. In [24], machine learning models were used to explore the possibility to improve accuracy of failure prediction in virtualized high performance clouds. The authors applied different machine learning algorithms on collected time-series data including SVM, RF, k-nearest neighbors (KNN), etc. The applied models showed good prediction results to identify possible application failures within the system with an accuracy of 90%. The goal of work proposed in [22] is to find the main features that identify application failures in cloud. In addition, they present a prediction model to early identify failure or successful execution of tasks in a cloud application using LSTM. Indeed, the LSTM-based model takes the performance data for each job and task as input and gives the termination status (e.g., failed and finished etc.) with an accuracy of 87%.

Our work is different in the use of contextual information from the monitored cloud. Moreover, it is used in an online edge cloud, which allows to show its effectiveness to timely detect and predict faults compared to [24] [23]. Furthermore, our proposed framework allows to predict faults from other layers (*e.g.,* using container data) and detect a worker fault through the master node (using data from the master node) which can improve the recovery mechanism and have a good impact on the scheduling and workload balancing in an edge cloud.

6 CONCLUSION AND FUTURE WORK

In this paper, we propose a framework for fault detection and prediction in edge cloud environments. The proposed framework is able to timely identify the occurrence of faults accordingly using machine learning techniques to avoid their propagation through the cloud. For fault detection, we apply various models including RF, SVM, NN, ARIMA, MC, BN and TAN. For fault prediction, we use deep learning techniques including LSTM and CNN. To demonstrate the effectiveness of our framework, we implement and test the proposed framework on an edge cloud testbed and we perform a case study to show its effectiveness. The results show that the framework allows to timely detect and predict several faults online. For instance, using SVM, RF and NN models, the framework is able to detect non-fatal CPU and HDD overload faults with an F1-Score of more than 95%. For the prediction, both CNN and LSTM have comparable accuracy 96.47% vs. 96.88% for CPU-overload fault and 85.52% vs. 88.73% for network fault.

All in all, we made a comprehensive comparison of different machine learning models for fault detection and prediction in an edge cloud environment. Throughout the experiments, we first found out that the pre-processing component in our framework is a crucial one since it allows to get the most relevant data for the trained models. Furthermore, it reduces the cost associated with the training phase as well as it enhances the model performance when running online. Moreover, we observed that some faults (*e.g.,* CPU fault) can be detected using either data collected from master node or data collected from a container layer. Hence, this can help cloud providers efficiently manage (*e.g.,* application rescheduling, load balancing) the running applications. Thus, it could reduce the impact of infrastructure faults on the application layers.

As a future work, we plan to extend our framework using unsupervised learning algorithms and assess its performance compared to the used supervised models especially to cover the faults that are hard to detect.

REFERENCES

[1] 2019 (accessed May, 2019). Azure IoT Edge. https://azure.microsoft.com/en-ca/services/iot-edge/.

[2] 2019 (accessed May, 2019). Edge TPU. https://cloud.google.com/edge-tpu/.

[3] 2019 (accessed May, 2019). How to Develop Convolutional Neural Networks for Multi-Step Time Series Forecasting. https://machinelearningmastery.com/how-to-develop-convolutional-neural-networks-for-multi-step-time-series-forecasting/.

[4] 2019 (accessed May, 2019). Kubernetes federation v2. https://github.com/kubernetes-sigs/federation-v2.

[5] 2019 (accessed May, 2019). Prometheus. https://prometheus.io/.

[6] 2019 (accessed May, 2019). Real-time visibility into stacks, sensors and systems. https://www.influxdata.com/.

[7] 2019 (accessed May, 2019). Selecting good features – Part III: random forests. https://blog.datadive.net/selecting-good-features-part-iii-random-forests/.

[8] 2019 (accessed May, 2019). Selecting good features – Part IV: stability selection, RFE and everything side by side. https://blog.datadive.net/selecting-good-features-part-iv-stability-selection-rfe-and-everything-side-by-side/.

[9] 2019 (accessed May, 2019). Stress-ng. https://wiki.ubuntu.com/Kernel/Reference/stress-ng.

[10] 2019 (accessed May, 2019). TC. http://manpages.ubuntu.com/manpages/xenial/man8/tc.8.html.

[11] 2019 (accessed May, 2019). Tempest. https://theiotlearninginitiative.gitbook.io/edgecomputingsolutions/introduction/stacks/openstack/testing/akraino/tempest.

[12] A. Cauveri and R. Kalpana. 2017. Dynamic fault diagnosis framework for virtual machine rolling upgrade operation in google cloud platform. In *International Conference on Power and Embedded Drive Control (ICPEDC)*. 235–241.

[13] M. Farshchi, J. Schneider, I. Weber, and J. Grundy. 2015. Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis. In *IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. 24–34.

[14] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. 2008. The Cost of a Cloud: Research Problems in Data Center Networks. *SIGCOMM Comput. Commun. Rev.* 39, 1 (2008), 68–73.

[15] Q. Guan and S. Fu. 2013. Adaptive Anomaly Identification by Exploring Metric Subspace in Cloud Computing Infrastructures. In *IEEE International Symposium on Reliable Distributed Systems*. 205–214.

[16] A. Gulenko, M. Wallschläger, F. Schmidt, O. Kao, and F. Liu. 2016. Evaluating machine learning algorithms for anomaly detection in clouds. In *IEEE International Conference on Big Data (Big Data)*. 2716–2721.

[17] Anton Gulenko, Marcel Wallschläger, Florian Schmidt, Odej Kao, and Feng Liu. 2016. A System Architecture for Real-time Anomaly Detection in Large-scale NFV Systems. *Procedia Computer Science* 94 (2016), 491 – 496.

[18] T. Gunasegaran and Y. Cheah. 2017. Evolutionary cross validation. In *International Conference on Information Technology (ICIT)*. 89–95.

[19] Cheol-Ho Hong and Blesson Varghese. 2018. Resource Management in Fog/Edge Computing: A Survey. *CoRR* abs/1810.00305 (2018). http://arxiv.org/abs/1810.00305

[20] Shrivas AK Hota HS, Handa R. 2017. Time Series Data Prediction Using Sliding Window Based RBF Neural Network. *International Journal of Computational Intelligence Research* 17, 5 (2017), 1145–1156.

[21] S. Huang, X. Xu, Y. Xiao, and W. Wang. 2012. Cloud Based Test Coverage Service. In *IEEE 19th International Conference on Web Services*. 648–649.

[22] T. Islam and D. Manivannan. 2017. Predicting Application Failure in Cloud: A Machine Learning Approach. In *IEEE International Conference on Cognitive Computing (ICCC)*. 24–31.

[23] A. Lavin and S. Ahmad. 2015. Evaluating Real-Time Anomaly Detection Algorithms – The Numenta Anomaly Benchmark. In *IEEE Conference on Machine Learning and Applications*. 38–44.

[24] Bashir Mohammed, Irfan Awan, Hassan Ugail, and Muhammad Younas. 2019. Failure prediction using machine learning in a virtualised HPC system and application. *Cluster Computing* (03 2019). https://doi.org/10.1007/s10586-019-02917-1

[25] S. K. Mukkavilli and S. Shetty. 2012. Mining Concept Drifting Network Traffic in Cloud Computing Environments. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 721–722.

[26] H. S. Pannu, J. Liu, Q. Guan, and S. Fu. 2012. AFD: Adaptive failure detection system for cloud computing infrastructures. In *IEEE International Performance Computing and Communications Conference*. 71–80.

[27] Pedro Henriques dos Santos Teixeira, Ricardo Gomes Clemente, Ronald Andreu Kaiser, and Denis Almeida Vieira, Jr. 2010. HOLMES: An Event-driven Solution to Monitor Data Centers Through Continuous Queries and Machine Learning. In *ACM International Conference on Distributed Event-Based Systems (DEBS '10)*. 216–221.

[28] H. Truong and M. Karan. 2018. Analytics of Performance and Data Quality for Mobile Edge Cloud Applications. In *IEEE 11th International Conference on Cloud Computing (CLOUD)*. 660–667.

[29] Chengwei Wang, , V. Talwar, K. Schwan, and P. Ranganathan. 2010. Online detection of utility cloud anomalies using metric distributions. In *IEEE Network Operations and Management Symposium*. 96–103.

[30] C. Wang, K. Viswanathan, L. Choudur, V. Talwar, W. Satterfield, and K. Schwan. 2011. Statistical techniques for online anomaly detection in data centers. In *IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*. 385–392.

[31] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting Large-scale System Problems by Mining Console Logs. In *ACM Symposium on Operating Systems Principles*. 117–132.

[32] Yi Zheng, Qi Liu, Enhong Chen, Yong Ge, and J. Leon Zhao. 2014. Time Series Classification Using Multi-Channels Deep Convolutional Neural Networks. In *Web-Age Information Management*. Springer International Publishing, 298–310.