Detecting and Predicting Anomalies for Edge Cluster Environments using Hidden Markov Models

Areeg Samir and Claus Pahl
Faculty of Computer Science
Free University of Bozen-Bolzano
Bolzano, Italy
Email: {areegsamir,Claus.Pahl}@unibz.it

***Abstract*—Edge cloud environments are often build as virtualized coordinated clusters of possibly heterogeneous devices. Their problem is that infrastructure metrics are only partially available and observable performance needs to be linked to underlying infrastructure problems in case of observed anomalies in order to remedy problems effectively. This paper presents an anomaly detection and prediction model based on Hidden Markov Model (HMM) that addresses the problem of mapping observations to underlying infrastructure problems. The model aims at detecting anomalies but also predicting them at runtime in order to optimize system availability and performance. The model detects changes in response time based on their resource utilization. We target a cluster architecture for edge computing where applications are deployed in the form of lightweight containers. To evaluate the proposed model, experiments were conducted considering CPU utilization, response time, and throughput as metrics. The results show that our HMM detection and prediction performs well and achieves accurate fault prediction.**

***Index Terms*—Cloud, Cluster, Anomaly detection, Fault injection, Prediction, Containers, Hidden Markov Model.**

I. Introduction

Edge cloud computing requires applications to be deployed on distributed virtualized clusters of nodes that are possibly heterogeneous in nature. We target cluster architectures for edge cloud computing where applications are deployed in the form of lightweight containers on these possible mobile virtualized nodes, for instance using Docker Swarm or Kubernetes for cluster orchestration [1], [2]. Here, not necessarily all infrastructure information is accessible and only application-level observations are possible. Resources that run deployed containers may cause performance anomalies. A performance anomaly arises when a resource behaviour deviates from its expectation. Performance anomalies can impact on required system behaviour, but it is difficult to link observable performance anomalies with underlying but invisible resource problems. Typical anomalies in an edge cloud setting are response time degradations caused by overloaded low-capacity devices. Another typical anomaly arises if connected sensors produce irregular amounts of data, which in high volume cases cause spike demands for data processing and storage.

An anomaly detection system aims at identifying anomalous patterns of behaviour. Many solutions have been proposed, [4], [5], [6], however, linking observations to underlying hidden infrastructure in an edge environment has not been sufficiently addressed [3], [14]. Detecting and predicting anomalies in such a clustered environment is recognized as a research gap [9], [15], [10]. To detect deviations from expected behaviour, and to predict their future occurrence, a performance anomaly prediction model is proposed based on Hidden Markov Model (HMM). A HMM [11] is a probabilistic model that is represented by a distribution of probabilities for all the possible results. HMMs can specifically address the correct mapping between observable and hidden properties here.

In practice, containerized cluster monitoring tools and controllers provide static rule-based auto-scaling approaches, but are not flexible enough to adjust to dynamic root cause mappings during runtime [7], [8]. Thus, predicting possible future anomalies aids in deciding to what extent the proposed approach should be adapted to observed changes in the dynamic environment. Our contribution is to detect and predict anomalous behaviour of containers based on the observed response time of containers through HMMs. An analysis of response time is performed to determine and predict the resource utilization of containers based on probabilities captured in the model. Metrics are collected from the monitored resources of the system at different abstraction levels [12]: (1) Resource utilization: proportion of the resources in use (CPU, Memory) to the total number of resources provided per time unit. (2) Response time: time taken by a request until the arrival of the response. (3) Throughput: the average number of interactions completed per time unit during an interval.

To validate the proposed HMM model, the evaluation is based on analyzing a dataset extracted from multiple containers in a cluster. The experiment shows that once the HMM is configured properly, it can predict anomalous behaviour with sufficient efficiency and with an accuracy of 96%.

The paper is organized as follows: using HMM to detect and predict anomaly is addressed in Section II. The implementation and evaluation to asses the proposed model are presented in Section III. Related work is introduced in Section IV. Conclusions and future work are made in Section V.

II. The Anomaly Detection and Prediction Model

Anomaly detection matches current behaviour against the normal behaviour models and calculates the probability with which non-normal behaviour occurs. This section proposes a

Markov model for detecting and predicting anomalies in container behaviour on the basis of past behaviour data collected from metrics and logs to obtain probabilities.

The model will be developed for cluster architectures of nodes $N_{i.j}$ for edge cloud computing where we assume applications to be deployed on the nodes in the form of lightweight containers $C_{i.j}$, for instance using Docker Swarm or Kubernetes for orchestration. This reflects a variety of edge cloud settings, down to using mobile or small single-board devices for an edge architectures that host container clusters as we have done in [13], [16], see Fig. 1. We assume the metrics required here to be provided by e.g. Docker.

A. Hidden Markov Model (HMM) for Anomaly Detection

A Hidden Markov Model (HMM) is a statistical Markov model in which the system being modelled is assumed to be a Markov process with hidden states (here infrastructure states). The hidden states are unobserved during the process. HMM is identified by $\lambda = \{A, B, \pi\}$ with:

1) π: initial state distribution, $\pi = P(q_1 = S_i)$.
2) N: number of states, $State_{Space} = \{N * N\}$.
3) $A = S_{ij}$: state transition probability matrix. The system transfers from state S_i to state S_j with probability S_{ij}.
4) O: the number of observation symbols per state, here representing output of containers being modelled.
5) $B = b_j(o_k)$ observation probability matrix. Each state S_j generates output with a probability distribution function $b_j(o_k) = P(O_t = o_k | q_t = S_j)$, where O_t is the observation vector at time t, O the observation symbols per state. The observation symbols represent output of the model.

We align the HMM annotations with the assumed system topology of containers deployed on cluster nodes. The observable states are associated with containers. The observations O are associated to the response time emissions R that are emitted from containers. In our model we map states S_i to containers C_i. We use performance metrics such as response time and resource utilization to describe the different behaviour of containers.

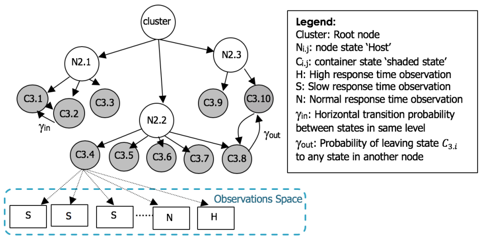
cluster
N2.1
N2.2
N2.3
C3.1
C3.2
C3.3
C3.4
C3.5
C3.6
C3.7
C3.8
C3.9
C3.10
γ_{in}
γ_{out}
Observations Space
S
S
S
N
H
Legend:
Cluster: Root node
$N_{i.j}$: node state 'Host'
$C_{i.j}$: container state 'shaded state'
H: High response time observation
S: Slow response time observation
N: Normal response time observation
γ_{in}: Horizontal transition probability between states in same level
γ_{out}: Probability of leaving state $C_{3.i}$ to any state in another node

Fig. 1: Anomaly Detection and Prediction for Edge Cluster Architecture with HMM

B. Model Construction

The defined HMM reflects the structure of our edge architecture. A HMM is a useful tool to model sequences of information in a graph with 'N' nodes called 'states', and 'edges' representing transitions between those states. Each state has an initial state distribution and emits the observation probability at which a given symbol is to be observed. The edges in the model represent transition probabilities between states. HMMs have been selected here to better control prediction accuracy.

In order to train a HMM, historical execution data is collected for response time and resource utilization, which we will describe in the implementation section later. The aim from collecting data is capturing the system status to study its resource consumption to assess the overall observable performance of the system for the given resource consumption.

Once training data are collected, we can built HMMs for containers that are deployed at different nodes, as shown in Fig. 1. The HMM is trained using different hidden states. The HMM is trained on CPU and Memory computing resources that are assigned to the containers. Based on resource utilization measurements, the HMM is built as a model of the containers' normal behaviour. Any observed sequence of resource utilization measurements that is unlikely is then considered as an anomaly, as the HMM is trained to detect anomalous behaviour. After the initial training of the model, a test is applied to compare actual observed data with the predicted based on the stored historical data. If there is no observed anomaly, then the approach will declare the status of the system to be anomaly free or not.

Fig. 2 shows the process of the detection step, where for a detected anomaly, a root cause identification is performed to find the underlying anomaly reason.

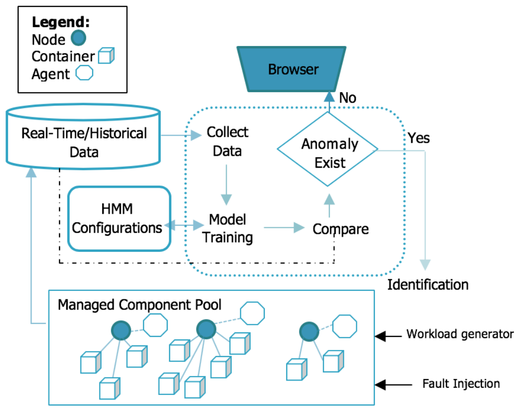

Fig. 2: Anomaly Detection and Prediction Process

C. Model Observations and Probabilities

An executed HMM model will emit sequences of observations based on the defined HMM rules. The observations depict the emitted response time produced by the containers.

Let us consider the observation at time t to be the variable O_t. O_t represents the response times that are emitted by a container. According to the HMM, we assume that the observation at time t was generated by states C_t. These states are hidden from the observer. The C_t refers to the deployed containers. The C_t states are discrete. They can take i values denoted by the integers $1, \ldots, i$. These states C_t are dependent only of the previous state C_{t-1}, and the output O_t only depends on the state C_t. To define the HMM, we need to specify probability distributions over the initial state, state transition, and observation. The HMM model for containers is consisted of a state space (SP_C) that includes several container states $\{C_{3.1}, C_{3.2}, C_{3.3}, \ldots, C_{i.j}\}$.

The model has (1) an initial probability value (π) for each state indicating how likely it is for a new input sequence to start in a given state (i.e., the probabilities for the initial state of the system), (2) a state transition probability matrix (A) indicating the likelihood of transitioning from one state to another. The state transition probability controls the way the hidden state at t is chosen given: the hidden state at time $t-1$, and an output probability distribution of observation matrix (B). The observation matrix indicates how likely it is for a certain measure value to come from a given state.

To calculate the initial transition probability matrix for a container cluster model, a Bayesian rule is used as it obtains simple rules as in "(1)".

$$P(C_{i.j}|C_{i.k}) = \frac{f(C_{i.j}|C_{i.k})}{f(C_{i.k})} \tag{1}$$

The $f(C_{i.j}|C_{i.k})$ is the observed frequency of transition of the $C_{i.j}$ to the $C_{i.k}$ (e.g., $C_{3.1}$ to $C_{3.2}$). $f(C_{i.k})$ is the measured frequency of the $f(C_{i.k})$. Once the initial probability transition matrix is calculated, we apply the Baum-Welch algorithm to train the model parameters to obtain the state transition matrix. The model training terminates once the probabilities of each model do not change anymore. At the end of the training, the resulting model allows us to

1) obtain the probability distributions of the observation sequence and states,
2) determine an anomalous sequence of states and predicting future occurrences.

Once we have a learned probabilistic model, we can compare it against the observed sequence to detect anomalous behaviour. If the observed sequence and predicted sequence are similar, we can conclude that we have a model learned on normal behaviour. In such case, the approach will declare the status of the system 'Anomaly Free' (no observed anomaly), and it will issue an alarm for further investigation. Learning the model on normal behaviour aids in estimating the probability of each observed sequence.

The output of the model is normalized in order to have proper probability distributions as in "(2)".

$$P(O_w|C_{i.j}) = \frac{n(O_w, C_{i.j}) + 1}{n(C_{i.j}) + |O_w|} \tag{2}$$

where $|O_w|$ is the observation number, $n(O_w, C_{i.j})$ is the number of times observations O_w emitted by container state $C_{i.j}$, and $n(C_{i.j})$ is the total number of observations emitted by container state $C_{i.j}$.

D. Metrics

For an observed sequence O_t from the container model, we assume here a sample time period from 0 to 100 sec to reflect common small deployment scenarios. These values represent the response time of container resource utilization. An abstracted state of a container to aid the anomaly detection/prediction can be inferred from the resource utilization. For example, if the CPU utilization is less than the lower threshold (e.g., 30%), the container is underloaded, and if the CPU utilization is greater than the upper threshold (i.e., 80%), the container is overloaded. Otherwise, the container is considered at normal load.

III. Implementation and Evaluation

The implementation of the model focuses on detecting and predicting anomalous workload in containers through considering different performance measurements such as CPU utilization, Memory consumption, and Response Time. *Detection* is assessed in terms of performance, looking at whether the technique can be used at runtime. *Prediction* is assessed in terms of accuracy, looking at the dependability of the results.

A. Experimental Setup

In order to train the HMM, and evaluate the effectiveness of the proposed approach, the TPC-W benchmark is used. TPC-W covers resource provisioning, scalability and capacity planning for e-commerce websites. The benchmark emulates an online bookstore that consists of 3 tiers: client application, web server, and database [17]. Each tier is installed on a single node (i.e., VM). For each VM, we deployed a set of containers, and we run into them TPC-W benchmark.

The experimental environment is consisted of four VMs in two servers. Each server is equipped with Ubuntu 18.10, Xen 4.11, 2.7 GHz CPU and 8 GB RAM. The first server hosts 3 VMs and the second one hosts 1 VM. Each VM is equipped with LinuxOS (Ubuntu 18.10 version), one VCPU, 2GB of VRAM. The VMs are connected through a 100 Mbps network. Table I shows the environment settings under observation.

SignalFX Smart Agent monitoring third-party tool is used and configured to observe the runtime performance of containers and their resources. The agent is deployed on each VM to collect, and store performance metrics of the system under test (e.g., host metrics, container, performance metrics, and workloads). The gathered data are sent to the Real-Time/Historical storage. The experiments focused on CPU utilization, Memory utilization, and Response Time metrics. The CPU and Memory measurements are gathered from the TPC-W web server tier, while the Response time is measured from TPC-W client's end tier.

TABLE I. Configuration of Virtual Environment.

Components	**Settings**
Number of Nodes	4 VMs (3 for benchmark, 1 for fault injection)
Container no.	11
Number of VCPUs	4
CPU MHz	3100
VStorage	512 GB/VM
VRAM	2 GB
Guest OS	Ubuntu_18.10_x64
Speed	100 Mbps

A number of user requests are simulated, and iteratively increased using a client simulator (i.e., TPC-W Remote Browser

Emulator ’RBE’), see Fig. 3. The requests are sent from client to server with different randomly generated configuration parameters. Benchmark data are stored, and the initial database contents are randomly generated. The performance data of the benchmark are gathered from all the containers that run this benchmark. To measure the number of requests and response (latency), HTTPing is installed on each node. We assume that the database tier does not show anomalies.

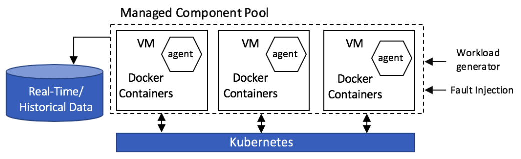

Fig. 3. Experimental Environment Setup

In the experiments, we considered that if multiple containers run on a node, all will get the same proportion of resources (i.e., CPU cycles). Consequently, if resources in one container are idle, other containers are able to use the leftover of resources (i.e., remaining of CPU cycles). There is no guarantee that each container will have a specific amount of resource (i.e., CPU time) at runtime. Because the actual amount of CPU cycles allocated to each container instance will differ based on the number of containers running on the same node, and the relative CPU-share settings assigned to containers.

B. Benchmark-based HMM Model Training

We configured the benchmark to generate different datasets with different configuration parameters. At the end, there are 1841 records in the generated datasets: 410 records in dataset ’X’ from node1, 800 records in dataset ’Y’ from node2, and 631 records in dataset ’Z’ from node3. The gathered datasets are separated into training and evaluation datasets.

To prepare the datasets, CPU utilization, Memory usage, and Response Time were monitored as performance metrics. Response time as an observable metric is measured at the client’s end, while CPU utilization and Memory usage are measured for the server and represent metrics hidden.

To obtain training data, TPC-W is run for 200 minutes. The workload data, which is also used for simulation, is created by the workload generator at different times while the system is under load. The workload of each container is assumed to be similar. The workload is generated by the TPC-W client applications (i.e., RBE). We created a workload that runs for 2940 seconds (49 min). During this period, the type of traffic mix, number of emulated-users and duration of each period changes periodically. We load each container gradually within a range of [30-1500] user requests. We used docker *stats* to obtain a live data stream for running containers. We also used *mpstat* to know the percentage of CPU utilization, and *vmstat* to collect information about the Memory usage. Fig. 4 shows CPU and Memory utilization for sample container $C_{3.4}$ for a long observation period (2 months).

The collected data is used to train the HMM model to detect and predict anomalies in containers. The model training lasted 100 minutes with results varying according to the number of states as shown in Fig. 1. The training of the model terminates once its probabilities do not change anymore. The remaining time is used for model testing.

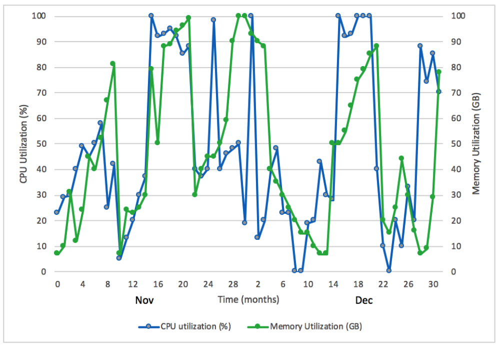

Fig. 4. CPU and Memory Utilization Before Prediction

Once the information is gathered, Heapster is used to group the collected data, and store it in the time series database InfluxDB. The gathered realtime historical data from the monitoring tool and from datasets are stored to improve the anomaly prediction based on the HMM model.

C. Anomaly Scenarios and Fault Injection

Training a prediction model requires suitable training data. However, the availability of such data is limited due to the rarity of anomaly occurrence, and the complexity of a manual training process. Thus, we assume that there is only one anomalous container at the same time.

To provide a more realistic setting for the experiments in addition to the data obtained from the benchmark, different types of anomalies are injected. Anomaly injection is also used to assess the anomaly prediction quality later. Anomalies are injected at the container level for collecting performance data at different system conditions. The fault injection is installed on a separate VM. Different types of faults are simulated based on the resources they impact, such as high CPU utilization and memory leaks. CPU load and memory faults are simulated using Stress-ng tool, which is used as a workload generator to run different load cases. The fault injection varies slightly depending on the anomaly type, but all last between 4 and 8 minutes. For each anomaly, the experiment is repeated from 5 to 10 times to verify the obtained results. Thus, the reported results reflect mean values over the runs for each experiment.

After the injection, datasets with various types of anomalies from different containers are gathered, and are used as anomaly datasets to detect and predict anomalous behaviour.

Table II summarizes the types of fault injections and workload generation in our experiments.

The fault injection types selected here aim at reflecting common edge anomaly situations.

- An anomaly situation arises if connected sensors in an IoT setting produce varying amounts of data, which in high volume cases cause spike demands for data processing and storage. The *CPU hog* reflects this for the processing (computation) aspect. The *memory leak* addresses the data storage problem.

TABLE II. INJECTED FAULTS

Resource	Fault Type	Fault Description	Injection Operation
CPU	CPU hog	Consume all CPU cycles for a container	Employing infinite loops that consumes all CPU cycle
Memory	Memory leak	Exhausts a container memory	Injected anomalies to utilize all memory capacity
Workload	Workload Contention	Generate increasing number of requests that saturate the resources of components	Emulating users requests using Remote Browser Emulators in TPC-W benchmark

- Other anomalies in an edge cloud setting are response time degradations caused by overloaded low-capacity devices. Limits in network bandwidth could also impact on performance. Here the *workload contention* case applies.

We now specify the three fault cases in detail.

1) CPU Hog: CPU hog, also called CPU stress, causes anomalous behaviour in a component, i.e., a component not responding fast enough or hanging under heavy load. A CPU hog can be due to an unplanned increase in user requests. Fig. 5 (top) illustrates a CPU injection that happened at container $C_{3.4}$. The injection lasted 4 minutes, from 650-890 seconds.

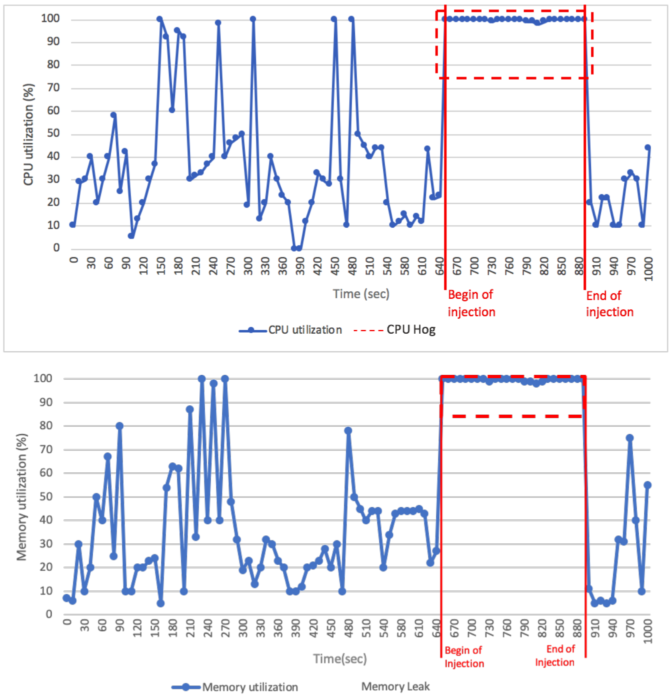

Fig. 5. CPU Injection (top) and Memory Injection (bottom)

2) Memory Leak: In this scenario, we simulated a memory leak. During the experiment, the usage of memory available in the component is significantly increased over a relatively short period of time. If all the memory is consumed, then an out-of-memory error is caused, and the health state of components is declared as anomalous. Fig. 5 (bottom) shows the memory injection that occurred at container $C_{3.4}$ for 4 minutes from 650-890 seconds, reflecting the same situation as the CPU hog.

3) Workload Contention: Workload contention occurs when an application imposes heavy workload on system resources, which result in performance degradation. Workload contention might occur due to limited resources, network latency, increasing the utilization of resource. Thus, workload management also plays important role in the total cost of ownership.

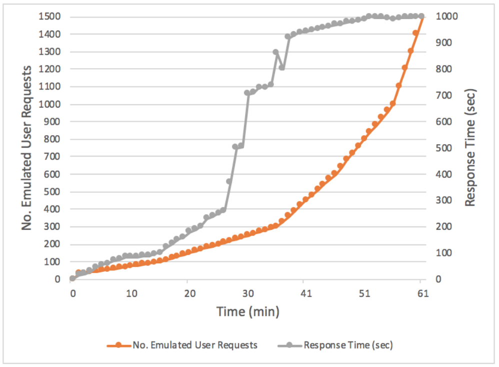

Fig. 6. Workload Contention – Response Time and Emulated User Requests

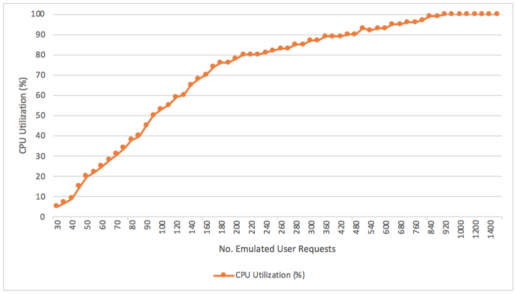

Fig. 7. Workload Contention – CPU Utilization and Emulated User Requests

- In this scenario shown in Fig. 6 increasing the number of emulated user requests has direct gradual influence on the response time of a container. Spearman's Ranked Correlation monotonically depicts a positive relationship between the number of emulated user requests and response time. The calculation of Spearman's correlation for those two variables gives a value of 0.99641, which reflects a strong relationship. It can also be observed from Fig. 7 that the CPU utilization reached a bottleneck started from 60% with increasing numbers of emulated user requests. It shows a strong correlation between the measured variables reaching 0.99219 as increasing the emulated user requests increases the CPU utilization.
- As shown in Fig. 6 and 7, both CPU utilization and response time are impacted by the number of emulated user request. The maximum throughput observed is around 150 request/sec, and it indicates that containers resources have reached 100% utilization as shown in Fig. 8.

D. *Evaluation – Anomaly Detection*

To evaluate the detection of anomalies using our HMM in a realistic setting in terms of detection speed, different containers are injected, one at a time, with different faults. The injection of containers follows the fault categories which are presented in the previous subsection. Fig. 9 shows that an anomaly was injected at 1500 sec (period 300 with 5 sec per

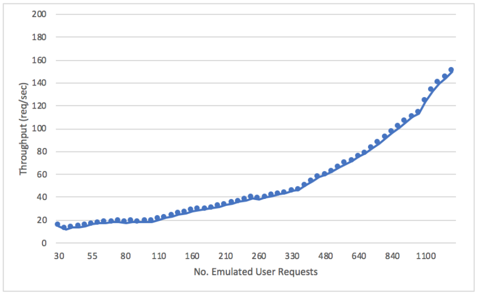

Fig. 8. Workload Contention – Throughput over Load Intensity

period) for all containers. The model took 205 sec to detect the CPU utilization anomaly at $t = 1705sec$ (period 341).

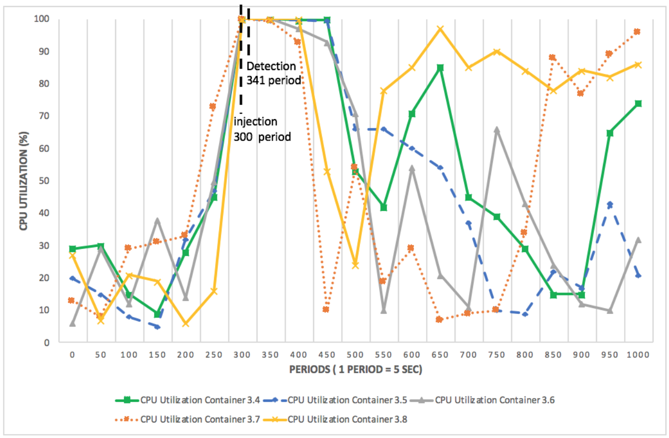

Fig. 9. CPU Detection using HMM

For the memory consumption, the model took 221 sec to detect anomaly, that was injected at $t = 250sec$ (period 50), and detected at $t = 471sec$ (period 73). As shown in the figure, Container $C_{3.4}$ was the first container affected by the CPU fault injection, while container $C_{3.6}$ was the first to be exposed by the memory leak injection.

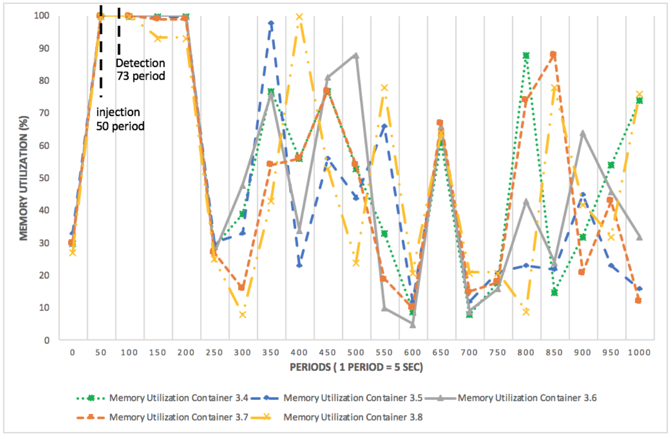

Fig. 10. Memory Detection using HMM

For workload contention, the number of emulated user requests increased gradually to 20, 50, 80, 110 and 140 for containers $C_{3.4}$, $C_{3.5}$, $C_{3.6}$, $C_{3.7}$ and $C_{3.8}$, respectively. Fig. 11 shows injection and detection of each container, with varying detection times. The reason for that points to different capacities of each container, and the correlation between the number of emulated user requests and response time.

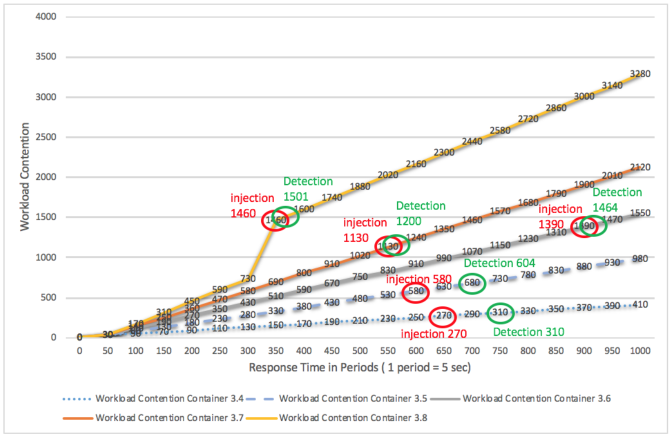

Fig. 11. Workload Contention Detection using HMM

Overall, we can conclude that with an average PC capacity, the detection is not instantaneous. Short spikes in demand causing performance anomalies might not be detected in time for remedial actions, but more gradual changes can be identified and remedied.

E. Evaluation – Anomaly Prediction

To evaluate the prediction accuracy for anomalous behaviour, the prediction was conducted using our HMM and also Dynamic Bayesian Network (DBN) to allow a comparative evaluation. DBN is based on directed acyclic graphs in which nodes represent the model variables, and edges represent the probabilistic relations between variables. A DBN is chosen as a reference since it is considered a common probabilistic graphical model that effectively deals with various uncertainty problems [18].

TABLE III. CPU FORECAST–HMM.

CPU utilization %	Time (months)
95.07	2018 – 11 – 18
98.32	2018 – 11 – 25
100	2018 – 12 – 15
88.10	2018 – 12 – 28
99.67	2019 – 01 – 19
100	2019 – 02 – 12
99.81	2019 – 02 – 13
99.43	2019 – 02 – 15

TABLE IV. CPU FORECAST–DBN.

CPU utilization %	Time (months)
91	2018 – 11 – 18
88.02	2018 – 11 – 25
99	2018 – 12 – 15
72.59	2018 – 12 – 28
98.22	2019 – 01 – 19
98.13	2019 – 02 – 12
98.03	2019 – 02 – 13
97.98	2019 – 02 – 15

We focus on CPU utilization as the most relevant anomaly case. Table III and Table IV show 2 months forward forecasts of CPU utilization after injecting the $C_{3.4}$ container with faults at different times. The result of the prediction are close to the utilization of CPU in the overall 2 months monitoring period. The figure shows the prediction of anomalous behaviour of CPU utilization that occurred in different periods of time.

The values of the CPU utilization here are approximated. The HMM model predicted actual behaviour with an average of 96% accuracy of the anomalous behaviour in CPU utilization, while DBN predicted with 90% accuracy.

The accuracy of the prediction model results was evaluated by Root Mean Square Error (RMSE), Mean Absolute Percentage Error (MAPE) and R^2 Prediction Accuracy, commonly used metrics for accuracy evaluations.

1) RMSE measures the difference between predicted and observed value by a model. As in "(3)", x is the actual output, y is the predicted output, and O_{no} is the number of observations in the dataset. A smaller RMSE value indicates a more effective prediction scheme.
2) MAPE also known as mean absolute percentage deviation (MAPD), is a measure of prediction accuracy of a model. MAPE are negatively-oriented scores, which means smaller values indicate a more effective prediction scheme as in "(4)". MAPE should be multiplied by 100, but this is omitted in this paper for ease of presentation without loss of generality.
3) R^2 measures the fitness of the detection model. R^2 value falls within the range [0,1]. It determines how the fitted model approximates the real data points. R^2 prediction accuracy of 1.0 indicates that the forecasting model is a perfect fit, while the smaller the R^2 value, the weaker is the model. In the equation, x is the actual output, y is the predicted output, O_{no} is the number of observations in the dataset, and $(y_i - z)$ is the variance of the predicted and actual output as in "(5)".

$$RMSE = \sqrt{\sum_{i=1}^{O_{no}} (y_i - x_i)^2 / O_{no}} \tag{3}$$

$$MAPE = 1/O_{no} \sum_{i=1}^{O_{no}} |y_i - x_i| / O_{no} \tag{4}$$

$$R^2 = 1 - \left(\sum_{i=1}^{O_{no}} (y_i - x_i)^2 / \sum_{i=1}^{O_{no}} (y_i - z)\right)$$
$$z = 1/O_{no} * \sum_{i=1}^{O_{no}} x_i \tag{5}$$

Furthermore, to obtain the number of correctly and incorrectly detected anomalies, recall and precision are calculated:

1) Precision, the ratio of the number of correctly detected anomalies to the total number of correctly and incorrectly detected anomalies (also referred to as Positive Predictive Value 'PPV') as in "(6)". High precision indicates the model correctly detects anomalous behaviour.

$$Precision = \frac{TP}{TP + FP} \tag{6}$$

2) Recall ('Sensitivity') in "(7)", also known as Confidence Score or True Positive (TP) rate, is the ratio of correctly detected anomalies to the total number of detected anomalies that are actually defective. Higher recall means that fewer anomaly cases are undetected.

$$Recall = \frac{TP}{TP + FN} \tag{7}$$

True positive (TP) means the anomaly is classified as anomalous behavior. False positive (FP), means that the model incorrectly classified normal data as anomaly. It is also known as a false alarm. False negative (FN), means the model incorrectly classifying anomaly as normal.

Table V presents the accuracy evaluation results for those metrics. The HMM prediction model performs slightly better compared to the DBN, though both models achieved fair results on the whole dataset. Comparing with DBN, the HMM model predicts anomalous behaviour with higher Precision, MAPE, Recall and RMSE.

TABLE V. PREDICTION EVALUATION RESULTS – COMBINED VIEW.

Metrics	HMM	DBN
RMSE	0.420	0.426
MAPE	0.506	0.508
R^2	0.774	0.722
Precision	0.736	0.721
Recall	0.713	0.699

The model was also looked at for each of the considered 3 datasets individually, as shown in Table VI. The results show that both HMM and DBN achieved good results on the moderate dataset 'Z'. The results of the models worsened gradually on dataset 'Y' and showed a dramatic decrease on dataset 'X'. However, DBN gave good results over HMM on dataset X. Comparing the results of Table V and Table VI, HMM is recommended over DBN to be used. Generally, the results are better on moderately sized datasets.

TABLE VI. PREDICTION EVALUATION RESULTS – 3 DATASETS.

Metrics	Datasets					
	X = 410		Y = 800		Z = 631	
	HMM	DBN	HMM	DBN	HMM	DBN
RMSE	0.441	0.436	0.391	0.400	0.310	0.316
MAPE	0.526	0.517	0.410	0.444	0.320	0.336
R^2	0.551	0.697	0.825	0.809	0.837	0.811
Precision	0.708	0.730	0.822	0.818	0.830	0.827
Recall	0.683	0.700	0.724	0.722	0.880	0.874

IV. Related Work

This section presents a review of anomaly detection models in general and HMMs as specific mechanism. Wang et al. [20] propose a technique to model the correlation between workload and the resource utilization metric of application to characterize the system status. However, the technique excludes anomalous behaviour prediction.

In [5], [4] Principal Component Analysis (PCA) is used to detect anomalous behaviour. However, PCA do not provide sufficiently accurate results. Others such as [6] use crosscutting concern, specifically Aspect Oriented Programming (AOP), to detect anomalies in web applications. Du et al. [10] detected anomalous behaviour in containerized environments. However, the authors focused on microservice architectures, which do not reflect the edge setting aimed at here. In [21], [23] HMM is used to detect intrusions within a network flow.

Unlike all these mentioned works, our presented model focuses on detecting and predicting performance anomaly within containers based on analyzing their response time, and mapping it to underlying hidden infrastructure problems. In this work, we used container response time and resource utilization measurements to construct a HMM that reflects the expected variations of container workload.

V. Conclusions and Future Work

Edge cloud architectures are often based on heterogeneous hosting infrastructure, possibly low capacity or mobile. where mapping anomalies observable by a user to underlying infrastructure problems isn't considered a simple task. We explored the use of HMMs model to detect anomalous behaviours found in edge cluster workloads. We focused on measuring the response time, and resource utilization of containers deployed on cluster nodes to detect and predict anomalous behaviours in such environment. As part of the model analysis, we derived workload variations of containers over different time scales captured by HMMs. Then, the Baum-Welch algorithm of HMM theory is used to estimate the model parameters from sets of empirical workload data from several containers.

The conducted assessment has demonstrated that the model is sufficiently accurate in detecting anomalies, and also predicting the future anomalous behaviours of workload within containers over different workload intensity periods. Moreover, it performs sufficiently well in common detection situations to allow it to be used in an online setting.

Currently, we are working on identifying different types of anomalies, and tracking anomaly causes in complex distributed systems. Once the anomaly is detected using the detection model, it will be analyzed through the identification step, that better identifies the actual root cause and can enforce an appropriate recovery action such as start/stop or migration [19]. Furthermore, capturing container properties in a semantic model [22] can be beneficial to improve the analysis accuracy.

References

[1] C. Mahmoudi, F. Mourlin, and A. Battou, "Formal definition of edge computing: An emphasis on mobile cloud and iot composition," in *3rd Intl Conference on Fog and Mobile Edge Computing (FMEC)*, 2018.

[2] D. von Leon, L. Miori, J. Sanin, N. E. Ioini, S. Helmer, and C. Pahl, "A lightweight container middleware for edge cloud architectures," in *Fog and Edge Computing: Principles and Paradigms*, 2019.

[3] A. Samir and C. Pahl, "A Controller Architecture for Anomaly Detection, Root Cause Analysis and Self-Adaptation for Cluster Architectures,", *Intl Conf Adaptive and Self-Adaptive Systems and Applications*, 2019.

[4] B. Agrawal, T. Wiktorski, and C. Rong, "Adaptive Anomaly Detection in Cloud Using Robust and Scalable Principal Component Analysis," *Intl Symposium on Parallel and Distributed Computing (ISPDC).*

[5] J. Alcaraz, M. Kaa, C. Sauvanaud, C. Sauvanaud, A. Detection, and L. Économie, "Anomaly Detection in Cloud Applications," Laboratoire d'Analyse et d'Architecture des Systemes, Toulouse, Tech. Rep., 2016.

[6] J. P. Magalhães, "Self-healing Techniques for Web-based Applications," PhD, University of Coimbra, 2013.

[7] P. Jamshidi, A.M. Sharifloo, C. Pahl, A. Metzger, G. Estrada, "Self-learning cloud controllers: Fuzzy q-learning for knowledge evolution," *International Conference on Cloud and Autonomic Computing*, 2015.

[8] P. Jamshidi, A. Sharifloo, C. Pahl, H. Arabnejad, A. Metzger, G. Estrada, "Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures," *Intl Conf on Quality of Software Architectures*, 2016.

[9] A. S. Shanmugam, "Docker Container Reactive Scalability and Prediction of CPU Utilization Based on Proactive Modelling," MSc Project, National College of Ireland.

[10] Q. Du, T. Xie, and Y. He, "Anomaly Detection and Diagnosis for Container-based Microservices with Performance," p. 14, 2018.

[11] D. Jurafsky and J. H. Martin, *Speech and Language Processing*, Prentice Hall, 2008.

[12] T. Taylor, "6 Container Performance KPIs You Should be Tracking to Ensure DevOps Success - Part 1," 2018.

[13] D. von Leon, L. Miori, J. Sanin, N. E. Ioini, S. Helmer, and C. Pahl, "A performance exploration of architectural options for a middleware for decentralised lightweight edge cloud architectures," in *Intl Conference on Internet of Things, Big Data and Security*, 2018, pp. 73–84.

[14] A. Samir and C. Pahl, "Anomaly Detection and Analysis for Clustered Cloud Computing Reliability,", *Intl Conf on Cloud Computing, GRIDs, and Virtualization*, 2019.

[15] A. Samir and C. Pahl, "Self-Adaptive Healing for Containerized Cluster Architectures with Hidden Markov Models,", *Intl Conf Fog & Mobile Edge Comp*, 2019.

[16] C. Pahl, N. E. Ioini, S. Helmer, and B. Lee, "An architecture pattern for trusted orchestration in iot edge clouds," *Intl Conf on Fog and Mobile Edge Computing*, 2018.

[17] TPC-W. (2012) TPC-W. [Online]. Available: http://pharm.ece.wisc.edu/tpcw.shtml

[18] E. Yu and P. Parekh, "A Bayesian Ensemble for Unsupervised Anomaly Detection," *arXiv preprint arXiv:1610.07677*, 2016.

[19] P. Jamshidi, C. Pahl, S. Chinenyeze, X. Liu, "Cloud migration patterns: a multi-cloud service architecture perspective," *ICSOC 2014 Workshops*, 2019.

[20] T. Wang, W. Zhang, J. Wei, and H. Zhong, "Fault Detection for Cloud Computing Systems with Correlation Analysis," pp. 652–658, 2015.

[21] C. M. Chen, D. J. Guan, Y. Z. Huang, and Y. H. Ou, "Anomaly network intrusion detection using Hidden Markov Model," *Intl Jrnl of Innovative Computing, Information and Control*, vol. 12, no. 2, 2016.

[22] C. Pahl, "An ontology for software component matching," *FASE Conference*, 2003.

[23] S. B. Cho and H. J. Park, "Efficient anomaly detection by modeling privilege flows using hidden Markov model," *Computers and Security*, vol. 22, no. 1, pp. 45–55, 2003.