DLA: Detecting and Localizing Anomalies in Containerized Microservice Architectures Using Markov Models

Areeg Samir
Free University of Bozen-Bolzano, Bolzano, Italy

Claus Pahl
Free University of Bozen-Bolzano, Bolzano, Italy

***Abstract*—Container-based microservice architectures are emerging as a new approach for building distributed applications as a collection of independent services that works together. As a result, with microservices, we are able to scale and update their applications based on the load attributed to each service. Monitoring and managing the load in a distributed system is a complex task as the degradation of performance within a single service will cascade reducing the performance of other dependent services. Such performance degradations may result in anomalous behaviour observed for instance for the response time of a service. This paper presents a Detection and Localization system for Anomalies (DLA) that monitors and analyzes performance-related anomalies in container-based microservice architectures. To evaluate the DLA, an experiment is done using R, Docker and Kubernetes, and different performance metrics are considered. The results show that DLA is able to accurately detect and localize anomalous behaviour.**

***Index Terms*—Cloud, Anomaly Detection, Performance, Container, Microservice, HHMM, Correlation Analysis.**

I. Introduction

Microservice architectures structure an application as a collection of services to keep it scalable and reliable [1, 6]. Each service can be considered as an application of its own. Services can communicate with each other by exposing their endpoints for the same service contract in order to provide functionality over different protocols. Microservices introduce multiple benefits like high availability, better flexibility, fast responsiveness, and scalability. To support microservices, it is beneficial having a light-weight deployment mechanism using small independently deployable services. Those requirements can be achieved by utilizing the container technology.

Containers [3] isolate microservices into smaller instances that utilize only the required operating system and platform components, thus making them lightweight [15, 22]. However, microservices may face several challenges. One of these challenges is changing a service workload in a composed microservice may increase the system load as changes in one service may influence the workload of other dependent services, and may result in response time degradation that would be considered as anomalous. These variations may lead to violations of Service Level Agreements (SLAs) between providers and users.

In such a case, anomaly detection and localization can help in capturing and tracking the anomalous behaviour that deviates from the normal behaviour [11]. To be able to detect and localize the anomalous behaviour, there are plentiful metrics at node, container, service, and application levels.

In this paper, we propose a Detection and Localization systems for Anomalies (DLA) that detects and locates the anomalous behaviour of microservices based on the observed response time. The DLA tries to detect the variation in response time, and correlates it to the respective component (microservice, container, service), which shows anomalous behaviour. To validate the DLA, evaluation based on analyzing a dataset extracted from microservice based container is discussed in Section VI. The experiment shows that DLA can detect and locate the anomalous behaviour with an accuracy percentage of more than 97%.

The paper is organized as follows: a background is given in Section II. An overview of the literature studies is explained in Section III. Adjusting our model to reflect our system topology is addressed in Section IV. An overview of the proposed work (DLA) is presented in Section V. An evaluation is presented in Section VI. Conclusions are made in Section VII.

II. Background

Containers are isolated workload environments in a virtualized operating system such as cloud. They are used to allocate computing resources, help to organize, migrate and develop microservices [8, 9, 12].

Microservice architectures are an approach to develop a single application that is composed of small services, each of which is in a container and runs its own process. Microservices are communicating with lightweight mechanisms, often an HTTP resource API. They enable continuous delivery/deployment of large complex applications as they are deployed as individual containers within a containerized architecture so they can be scaled up or down by adding or removing instances. However, there are some characteristics [2] of microservices, which complicate its performance monitoring. One of such characteristics is *monitoring and deployment*. To guarantee the performance of container-based microservice application, a coherent analysis of the performance metrics across microservices, containers, and nodes monitored resources is required. However, variations and uncertainties of performance metrics across different microservices, containers and nodes resources complicate the assessment of microservice performance. If the anomalous measurements are

978-1-7281-2888-7/19/$31.00 ©2019 IEEE
DOI 10.1109/FiCloud.2019.00036

not localized to the causing component, then anomalies will remain untreated.

In this paper, we adopt Hierarchical Hidden Markov Models (HHMM) [5] to learn the model based on different monitored metrics such as CPU, Memory, and Network to detect and locate the anomalous behaviour. The aim is to model the relation between the monitored metrics of container, node and service, and the variation in response time under different load scenarios. HHMM [5] is a generalization of the statistical machine learning technique Hidden Markov Model (HMM). It is designed to model domains with hierarchical structure (e.g., speech recognition, plan recognition, intrusion detection). In most of these applications the model has a well performance to model hierarchical structure.

HHMM does not emit observable symbols directly as it is consisted of one root state that is composed of a sequence of sub-states. Each sub-state may compose of another sub-states and generates sequences by recursive activation. The process of recursive activations stops when it reaches a production state. The production state(s) is the only state that emits output symbols like HMM. The other states that emit sub-states instead of output symbols are called "internal states" or "abstract states". The process that an internal state activates a sub-state is termed "vertical transition", while a state transition at the same level is called "horizontal transition". When a vertical transition is completed, the state which originated the recursive action will get the control and then performed a horizontal transition. Each level, except the root, has an end state, which ends the horizontal transition at this level, and returns the control to the root state of this whole hierarchy.

HHMM is identified by $HHMM =< \lambda, \chi, \theta >$. The λ is a set of parameters consisted of horizontal ξ and vertical χ transitions between states q^d, d specifies the number of vertical levels in the hierarchy structure; state transition probability A; observation probability distribution B; initial transition π; state space SP at each level and the hierarchical parent-child relationship q_i^d, q_i^{d+1}. The i specifies the index of horizontal levels in the hierarchy structure. The Σ consists of all possible observations O. The states in HHMM are hidden from the observer and only the observation space is visible.

We expect the model to be able to: (1) efficiently analyze large amounts of data to detect potential anomalous observation in container based microservice architecture; (2) track the cause of the detected anomaly.

III. Related Work

In [7], the authors evaluate the performance of container-based microservices considering two models: master-slave (one container is the master of other containers), nested-container (children containers run application, and they are limited by their parent's boundaries). The authors evaluate the models by focusing on the CPU and Network performance data. In [2], the author propose an approach to detect and locate anomaly in container-based microservice using PAD, RanCorr and EAR. The authors focus on reducing the false alarm observed at response time. The paper in [10] presents an anomaly detection system for container-based microservices using different machine learning techniques. The authors evaluate the performance of the system through focusing on the CPU, Memory and Network. In [13], the authors investigate the behaviour of RPCA algorithm and HTM neural network on detecting anomalies in microservice architecture. In [14], an approach is proposed to detect anomalies that are related to the behaviour of services in VMs using machine learning techniques. The authors evaluate their approach by focusing on CPU, Memory, Disk and Network performance data.

Many literatures used HMM and its derivations to detect anomaly. In [17], the author proposes various techniques implemented for the detection of anomalies and intrusions in network using HMM. In [20] the author detects faults in real-time embedded systems using HMM through describing healthy and faulty states of a system's hardware components. In [23], HMM is used to find which anomaly is part of the same anomaly injection scenarios.

The objective of this paper is to detect and locate (DLA) the anomalous behaviour for container-based microservices using HMMs. The proposed DLA consists of: (1) *Monitoring* that collects the performance data of (services, containers, nodes 'VM') such as CPU, memory, and network metrics; (2) *Detection* that detects anomalous behaviour which is observed in response time of a component; (3) *Anomaly injection* which simulates different anomalies and gathers dataset of performance data representing normal and abnormal conditions.

IV. HHMM Customization

In this paper, we use HHMM: (1) to correlate the hidden component metrics with the observed response time to find the root cause of the anomaly. (2) We also use HHMM to detect components that are affected by the anomaly. We consider that our System Under Test (SUT) has hierarchical structure. The microservice architecture MS (root state) consists of some nodes VMs (internal states) that consists of one or more containers C (substates), and each container has one service S (production state) that runs on it. Containers may have communication at the same node or at external node. A microservice can be deployed in several containers at the same time, and a container is defined as a group of one or more containers constituting one microservice. These characters substituted a foundation to apply HHMM to describe the behaviours of a system in hierarchical structure.

To meaningfully represent our system, Fig. 1, we modify the HHMM annotations. So, the $q^{d=1}$ root state is mapped to $MS^{j=1}$. The internal state $q_i^{d=2}$ represents virtual machine (node), and it is mapped to VM_i^j. The i is the state index at horizontal level, and j is the hierarchy index at vertical level d. The substate q_i^{d+1} is mapped to C_i^{j+1} to represent containers (e.g., C_1^3 at vertical level 3 and horizontal level 1), and the production state q_i^{d+2} is mapped to S_i^j to represent services that emit observations $O_n = \{o_1, \cdots, o_n\}$. The set of observations O_n is associated to the response time fluctuation RT_n that are emitted from services in containers. The State Space SP is mapped to Microservice Space MSS, which consists of one or more Microservice(s) MS. Each MS consists of a set of VMs, containers C, and services S. Example, $VM_1^2 = \{C_1^3, C_2^3\}$, VM_2^2, $VM_3^2 = \{C_3^3, C_4^3\}$; $C_1^3 = \{S_1^4\}$, $C_2^3 = \{S_2^4\}$, $C_3^3 = \{S_3^4\}$, $C_4^3 = \{S_4^4\}$. The HHMM vertically calls one of VM_i^j with vertical transition χ. Since VM_i^j is internal state, it enters its child HMM substates C_i^j, which call the production state(s) S_i^j. Since S_i^j is a production state, it emits observations, and may make horizontal transition ξ from C_1^3 to C_2^3 as the service is deployed inside a container. Once there is no another transition, C_2^3 transits to the end state End, which ends the transition for this substate, to return the control to the calling state VM_1^2. Once the control returns to the state VM_1^2, it makes a horizontal transition (if exist) to state VM_2^2, which horizontally transits to state VM_3^2. When all the VM states finish all their substates, the last VM state returns the control to the MS to obtain the whole transitions and observations. The edge direction (horizontal/vertical transition) between the states represents the interaction between the states in our SUT. The horizontal transition between containers reflect the request/reply between the client/server and the vertical transition refers to parent-child dependency between microservice/VMs/containers/services. This type of hierarchical presentation aids in detecting and tracking the anomaly.

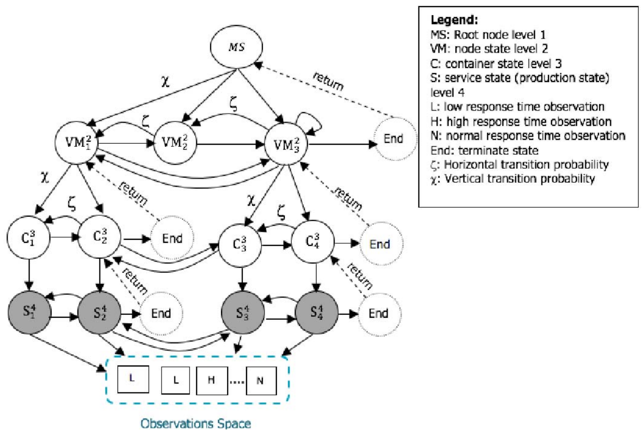

Legend:
MS: Root node level 1
VM: node state level 2
C: container state level 3
S: service state (production state) level 4
L: low response time observation
H: high response time observation
N: normal response time observation
End: terminate state
ζ: Horizontal transition probability
χ: Vertical transition probability
MS
χ
ζ
return
VM$_1^2$
VM$_2^2$
VM$_3^2$
End
C$_1^3$
C$_2^3$
C$_3^3$
C$_4^3$
S$_1^4$
S$_2^4$
S$_3^4$
S$_4^4$
L
L
H
N
Observations Space

Fig. 1. System Structure Using HHMM.

V. Detection and Localization of Anomaly (DLA)

A. Resource Performance Monitoring

To be able to accurately capture anomaly behaviour, performance data at different system levels are being collected from "Managed Component Pool" to study system resource status to be able to assess the overall performance of microservice.

Each VM in the pool has an installed agent that is responsible for collecting metrics from the pool, and exposing log files of containers and nodes to Real-Time/Historical Data storage. The collector clears outliers from the collected data. Then, those data will be monitored by the resource performance monitoring as will be shown later in this section.

As there are a huge number of metrics at different system levels, and a microservice may be deployed across many

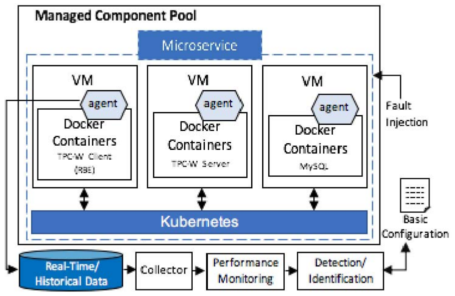


Fig. 2. Detection and Localization of Anomaly system (DLA).

servers, identifying the key metrics of microservices are necessary for describing its behaviour sufficiently. Thus, we explore the Key Performance Indicators (KPIs) metrics at the host-infrastructure and microservice levels, and we select the most common performance metrics that will aid us in monitoring the anomalous behaviour [4, 24, 25]. Table I shows the metrics that are collected at different system levels. Further, several monitoring tools are used to collect and store performance metrics of the target system such as SignalFX Smart Agent,docker $stats$ command to obtain a live data stream for running containers, $mpstat$ to know the percentage of CPU utilization, and $vmstat$ to collect information about the memory usage. The collected performance data are grouped using Heapster, and stored in a time series database called InfluxDB (Real-Time/Historical Data) to determine the time that the data collected belong. The data collected in the storage will be accompanied with the user transaction, which is a key created to identify the requests in a given time for that transaction. The key and the monitored metric parameter values will be used to know how much the resource vary based on the user transactions, and to compare it with the response time in the same given time of transaction. The user transactions refer to the request rate, which is the number of requests per second.

TABLE I. Monitoring Metrics

Metric Name	Description
CPU Usage	CPU utilization on all cores
CPU Request	CPU request in millicores
CPU Limit	CPU limit in millicores
CPU Rate	CPU usage on all cores in millicores
Memory Usage	Memory utilization
Memory Request	Memory request in bytes
Memory Limit	Memory limit in bytes
Memory Cache	Cache memory usage
Memory Page-Faults	Number of page faults
Memory Page-Faults-Rate	Number of page faults per second
Network-rx	Cumulative no. bytes received over netw
Network-rx-rate	No. of bytes received over network per sec
Network-rx-errors-rate	No. of errors while receiving per sec
Network-tx	Cumulative no. of bytes sent over network
Network-tx-rate	No. of bytes sent over network per sec
Network-tx-errors-rate	No. of errors while sending over network

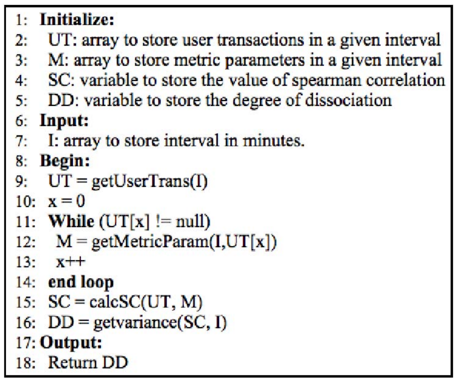
To check if there is performance variation at service or container level, we verify if the captured variation in a metric is a symptom of anomaly. Thus, we use spearman's rank correlation coefficient to estimate the dissociation between different number of user transactions, and the collected parameters from the monitored metrics. The dissociation refers to anomalous behaviour occurred at one or more component(s) at a given time that deviates from normal behaviours. The dissociation tells us if the value of the metric parameter increases or decreases according to the number of requests. If there is a decrease in the dissociation degree, then the metric is not associated with the increasing number of requests, which means the observed degradation in performance is not an anomaly. In case the degree of dissociation increases, this refers to the existence of anomaly. In such case, the anomaly detected as the impact of dissociation between the number of user transaction, and the monitored metric exceeds a certain value. To achieve that we generate general thresholds to highlight the occurrence of anomaly at different monitored metrics.

In Fig. 3, Lines 1-5 represent the creation of different parameters to be used during the algorithm. Lines 6-7 capture the number of user transactions during t-th time interval. Lines 8-14 loop to retrieve users transactions in a specific interval, and accumulate all the transactions to be used by the correlation coefficient. We retrieve and accumulate the monitored metric parameters for all containers that run the same service in the same specified interval for the users transactions. Line 15 calculates the spearman correlation for the users transactions' and the measured metric. The correlation is computed for each monitored metric, and the number of user transactions processed in a specific t time interval. The result of the correlation will tell the strength of the relationship between the two measured variables, which means if the number of user transactions increases, then the cumulative value of metric parameter should also increase. If the cumulative value of a parameter increases, and the number of requests does not increase, then the result of correlation indicates that the degradation in the performance metric may relate to workload contention or resource exhaustive. Line 16 calculates both the variances for the current time interval, and the highest old one for both monitored metric and user transactions. Then the product between the estimated variances will give one threshold. Such threshold is estimated for each monitored metric. Line 17 returns the threshold (degree of dissociation 'DD') used to detect the occurrence of anomaly behaviour for each monitored metric. The degree of dissociation (DD = 45 for CPU Utilization; DD = 20 for Memory Utilization; 15 for Network) can be used as an indicator for performance degradation. Once the anomaly behaviour is captured, we build HHMM to detect and locate the anomaly.

B. Detection and Localization of Anomaly (DLA)

Since anomalies always cause metrics to fluctuate, locating anomalous metrics help to narrow down the root causes. Thus, we need to detect and localize the metric, which most probably causes the container, node, or service status to change to anomalous. Here, we use the generalized Baum-Welch algorithm to train the HHMM to estimate model parameters.

We assume that each service, node, or container has workload, linked to CPU, Memory or Network metrics. Such



```
1:  Initialize:
2:    UT: array to store user transactions in a given interval
3:    M: array to store metric parameters in a given interval
4:    SC: variable to store the value of spearman correlation
5:    DD: variable to store the degree of dissociation
6:  Input:
7:    I: array to store interval in minutes.
8:  Begin:
9:    UT = getUserTrans(I)
10:   x = 0
11:   While (UT[x] != null)
12:     M = getMetricParam(I,UT[x])
13:     x++
14:   end loop
15:   SC = calcSC(UT, M)
16:   DD = getvariance(SC, I)
17: Output:
18:   Return DD
```

Fig. 3. Check Performance Degradation.

workload is hidden from the observer, and it can be detected through observing the variations in response time behaviour, which is emitted by the production service states S_i^j. To obtain the total performance data of a specific microservice, the performance data should be collected from various layers of the service that emitted the observation (node, container, service). The data, which are collected, depict whether a microservice is anomalous. We focus on performance detection and identification at container, node and microservice level.

a) Calculate The Likelihood of Observing Sequence: since each of the internal state in our HHMM can be viewed as HMM model that consists of containers substates and services production states, we calculate the likelihood of generating anomalous response time observations given the state sequence $P(RT|\lambda,\ MS_i^j)$, $P(RT|\lambda,\ VM_i^j)$, $P(RT|\lambda,\ C_i^j)$, $P(RT|\lambda,\ S_i^j)$. Let us assume that the last state ended End at the third level C^3 to return the control to VM_i^j. The observation sequences RT_n emitted at hierarchical length T, which is the length of the observation sequence ($t = 1,\ 2,\ \cdots,\ T$). The $i = (i_1, \cdots, i_n)$ is a set of horizontal states index that were visited during the generation of RT at the same level of states. $j = (j_1, \cdots,\ j_n)$ is a set of vertical state index. To calculate the probability that the observation sequence RT_n was generated by C, we calculate the probability that partial RT_n was generated by C^{j-1} (at that case: VM_i^2), and C^j was the last state activated by C^{j-1} as shown in "(1)". The $\alpha(t,\ t+k,\ C_i^j,\ C^{j-1})$ is the probability that the partial observation sequence $RT_n = \{rt_t, \cdots, rt_{t+k}\}$ was generated by state C_i^j at time t and end at time $t+k$, and C_i^j was the last state activated by C^{j-1} during the generation of the partial observation sequence. Each rt_t may generate observation sequence $rt_t = \{high\ response\ time,\ normal\ response\ time,$ $high\ response\ time\}$. To calculate α, we sum all possible states at level j ending at C_{End}^{j-1}.

$$\alpha(t,\ t+k,\ C_i^j,\ C^{j-1}) = P(rt_t, \cdots,\ rt_{t+k},\ C_i^j,\ t+k \mid C^{j-1},\ t) \tag{1}$$

To calculate the probability that the sequence $RT_n = \{rt_t, \cdots, rt_{t+k}\}$ was generated by C^{j-1}, we sum all the possible states at level j ending at C_{End}^{j-1} as shown in "(2)".

$$P(rt_t, \cdots,\ rt_{t+k}|C^{j-1}) = \sum_{i=1}^{C^{j-1}} \alpha(t,\ t+k,\ C_i^j,\ C^{j-1})\, a_{i\ End}^{C^{j-1}} \tag{2}$$

Then, we calculate the likelihood for the entire observation sequence through accumulating all the possible starting states (C and VM) called by the root state MS that began at time $t = 1$ and ends at T as in "(3)". The equation is applied also for each substate called by internal state, and for each production state called by substate by subtracting from the vertical level. For example, if we are at the container level $C^{j=3}$, then C^{j-1} leads to node level $VM^{j=2}$, and C^{j-2} from the container level leads to root state $MS^{j=1}$. We follow that way in the rest of the paper for simplicity. The same mechanism is also applied for the states at production state level $S^{j=4}$. In that case, we accumulate all the states from production states to the root state considering hierarchy structure has 4 levels.

$$\begin{aligned} P(RT_n|\lambda) &= \sum_{i=1}^{C^{j-2}} \alpha(1,\ T, C^j,\ C^{j-1},\ C^{j-2}) \\ P(RT_n|\lambda) &= \sum_{i=1}^{S^{j-3}} \alpha(1,\ T, S^j,\ S^{j-1},\ S^{j-2},\ S^{j-3}) \end{aligned} \tag{3}$$

Once we calculate the forward variable for all the states in the hierarchy, we calculate the backward variable as in (4). The backward variable $\beta(t,\ t+k,\ S_i^j,\ S^{j-1},\ S^{j-2},\ S^{j-3})$ is the probability that the observation sequence $(rt_t, \cdots, rt_{t+k})$ was generated by S_i^j at time t, and S_i^j was the last state activated by S^{j-3} in the hierarchy and finished at $(t+k)$ during the generation of the partial observation sequence.

$$\begin{aligned} &\beta(t,\ t+k,\ S_i^j,\ S^{j-1},\ S^{j-2},\ S^{j-3}) = \\ &P(rt_t, \cdots,\ rt_{t+k}|\ S_i^j,\ t,\ S^{j-3},\ t+k) \end{aligned} \tag{4}$$

Those steps will be applied to each state in the hierarchical starting from bottom to up.

b) Find The Most State Sequence (Detect Anomalous Path): to find the hidden hierarchical states sequence, which is the cause of anomaly, and to correlate the observed anomalous RT_n behaviour to the path that has anomalous component, we use the generalized Viterbi algorithm, and customize it to reflect our system. We start from the production states, and calculate the likelihood of the most probable hierarchical state sequence generating the observation sequence $(rt_t,\ \cdots,\ rt_{t+k})$ that S^{j-3} was entered at time t, and its substates S^{j-2}, S^{j-1} are the last states to be activated by S^{j-3}, and the control returns to S^{j-3} at time end $(t+k)$. First, we initialize the production state, and then we calculate the probabilities from bottom-up: (1) $\delta(t,\ t+k,\ S_i^j,\ S^{j-1},\ S^{j-2},\ S^{j-3})$, which is the likelihood of the most probable state sequence generating $(rt_t, \cdots,\ rt_{t+k})$ by a recursive activation. Such activation starts from S^{j-3}, and ended at S_i^j at time t, then it returns to S^{j-3} at time end $(t+k)$ as shown in "(5)". (2) We calculate the state list $\psi(t,\ t+k,\ S_i^j,\ S^{j-1}, \cdots,\ S^{j-3})$, which is the index of the most probable production states to be activated by S^{j-3} before activating S_i^j. (3) We calculate $\tau(t,\ t+k,\ S_i^j,\ S^{j-1}, \cdots,\ S^{j-3})$, which is the transition time at which S_i^j was called by S^{j-1} at time t and ended at time $(t+k)$ considering the hierarchy until S^{j-3} as show in "(6)". If S_i^j generates the entire subsequence, we set $\tau(t,\ t+k,\ S_i^j,\ S^{j-1}) = t$, where t refers to the started time of the generated sequence. Given (τ, ψ), we can obtain the most probable hierarchical state sequence from the production states to the root state listed by their activation time. From (δ, ψ, τ), for each level, we compute the recursion from the production until the root state. Equation "(7)" shows the computation of recursion from production state to S_i^j to root state S^{j-3} considering the probabilities of observations and states transitions.

$$\delta(t,\ t,\ S_i^j,\ S^{j-1}, \cdots,\ S^{j-3}) = \pi^{S^{j-3}}\ (S_i^j)\ b^{S_i^j}(rt_t) \tag{5}$$

$$\tau(t, t+k, S_i^j, S^{j-1}, \cdots,\ S^{j-3}) = t+k \tag{6}$$

$$\begin{aligned} &(\delta(t,\ t+k,\ S_i^j, \cdots,\ S^{j-3}),\ \psi(t,\ t+k,\ S_i^j, \cdots,\ S^{j-3})) = \\ &\max_{1 \le y \le S^{j-3}} \begin{Bmatrix} \delta(t,\ t+k-1, S_y^j, \cdots,\ S^{j-3}) \\ a_{i.y}^{S^{j-3}}\ b^{S_i^j}(rt_{t+k}) \end{Bmatrix} \end{aligned} \tag{7}$$

To find the most probable hierarchical state sequence in a set of hierarchical state sequences, we calculate δ of a state sequence at time set $\bar{t} = t+1, \cdots,\ t+k$. Then, we recursively calculate ω that represents ψ and $\Delta(\bar{t})$, where $\bar{t}$ is the time when $S_r^{j=4}$ was activated by $S_r^{j-1}, S_r^{j-2},\ S_i^{j-3}$ (e.g., $C,\ VM,\ MS$) at start time t and end time $(t+k)$. Then we take the maximum as depicted in "(8)", and "(9)". The r and y represent horizontal indexes. Then, we calculate transition probabilities (a) from production states until root state.

$$L = \max_{(1 \le r \le S_i^{j-1})} \begin{Bmatrix} \delta(\bar{t},\ t+k,\ S_r^j,\ S_r^{j-1}) \\ a_{r\ End}^{S_i^{j-1}} \end{Bmatrix} \tag{8}$$

$$\omega = \max_{(1 \le y \le S^{j-3})} \begin{Bmatrix} \delta(t, \bar{t}-1,\ S^{j-1},\ S^{j-2},\ S^{j-3}) \\ a_{yi\ End}^{S^{j-3}}\ L \end{Bmatrix} \tag{9}$$

Once all the recursive transitions are finished and returned to root state (MS), we get the probability of most hierarchy state sequence starting from (MS) following by (VM) internal state(s) to obtain all its substates in the hierarchy through scanning: the sate list ψ, states likelihood δ, and transition time τ as shown in "(10)".

$$stateSeq = \max_{VM_i^2} \begin{Bmatrix} \delta(1, T, VM_i^2, MS), \\ \tau(1, T, VM_{End}^2, MS), \\ \psi(1, T, VM_{End}^2, MS) \end{Bmatrix} \tag{10}$$

c) Estimate The Model Parameters (Locate the Anomalous Component): the HHMM is trained on response time over the hidden metrics. The model is learned using the generalized Bauch-Welch algorithm to obtain the transition probabilities of states, observations, and state transitions. Once we have a learned model, this output will be used to feed the Viterbi algorithm to obtain the hierarchy path of the anomalous state. To achieve that, we train the model to get the probabilities over all the states. So, four variables will be calculated: forward transition α , backward transition β, vertical transition χ, and horizontal transition ξ as shown in "(11)-(15)". At equation "(11)", we start with calculating the transition and observation probabilities at service level (S) that is happened at time t and ended at time $(t+k)$.

$$\alpha(t,\ t+k,\ S_i^j,\ S^{j-1}) = \left[\sum_{u=1}^{S^{j-1}}\ \alpha(t,\ t+k-1,\ S_u^j,\ S^{j-1})\ a_{ui}^{S^{j-1}}\right] b^{S_i^{j-1}}\ (rt_{t+k}) \tag{11}$$

Then as in "(12)", we calculate the transition probabilities at the container (C_i^j) and VM (C^{j-1}) levels that happened at time t and ended at time $(t+k)$ considering the observations of substate that are generated from the production state.

$$\begin{gathered}\alpha(t,\ t+k,\ C_i^j,\ C^{j-1}) = \\ \sum_{l=0}^{k-1}\left[\sum_{y=1}^{C^{j-1}}\ \alpha(t,\ t+l,\ C_y^j,\ C^{j-1})\ a_{i.y}^{C^{j-1}}\right] \\ \left[\sum_{g=1}^{C_i^j}\ \alpha(t+l+1,\ t+k,\ C_g^{j+1},\ C_i^j)\ a_{g\ End}^{C_i^j}\right] + \\ \pi^{C^{j-1}}(C_i^j)\left[\sum_{g=1}^{C_i^j}\ \alpha(t,\ t+k,\ C_g^{j+1},\ C_i^j)\ a_{g\ End}^{C_i^j}\right]\end{gathered} \tag{12}$$

Then as in "(13)", we calculate the backward probability β. The β is the probability of seeing the observations from time $(t+1)$ to time end $(t+k)$ at the production state level.

$$\beta(t,\ t+k,\ S_i^j,\ S^{j-1}) = b^{S_i^{j-1}}\ (rt_t)\left[\sum_{u\neq End}^{S^{j-1}}\ a_{ui}^{S^{j-1}}\ \beta(t+1,\ t+k,\ S_u^j,\ S^{j-1})\right] \tag{13}$$

After that as in "(14)", we calculate the backward transition probabilities at the container (C_i^j) and VM (C^{j-1}) levels considering the observations of substate that are generated from the production state (C^{j+1}).

$$\begin{gathered}\beta(t,\ t+k,\ C_i^j,\ C^{j-1}) = \\ \sum_{l=0}^{k-1}\left[\sum_{s=1}^{C_i^j}\pi^{C_i^j}(C_s^{j+1})\ \beta(t,\ t+l,\ C_s^{j+1},\ C_i^j)\right] \\ \left[\sum_{u=1}^{C^{j-1}}\ a_{i.u}^{Cj-1}\ \beta(t+l+1,\ t+k,\ C_u^j,\ C^{j-1})\right] + \\ \left[\sum_{s=1}^{C_i^j}\pi^{C_i^j}(C_s^{j+1})\ \beta(t,\ t+k,\ C_s^{j+1},\ C_i^j)\right] a_{i\ End}^{C^{j-1}}\end{gathered} \tag{14}$$

To obtain the number of horizontal transitions from a state C_i^j to another C_{End}^j both are substates of C_l^{j-1} , we calculate ξ as in "(15)". The η_{in} refers to the probability of performing vertical transition from substate C_l^{j-1} to state C^{j-2} that the RT_n is started to be emitted for a state at starting time t. The vertical transition happened, once all the horizontal transitions are done. η_{in} is calculated using ξ by summing all substates C_l^{j-1} which can perform transition to C^{j-2}. The η_{out} refers to the vertical state transition probability from C_l^{j-1} to state C^{j-2} in which RT_n is emitted and finished at T.

$$\begin{gathered}\xi(t, C_i^j,\ C_{End}^j,\ C_l^{j-1}) = \frac{1}{P(RT|\lambda)} \\ \left[\sum_{t=1}^{T}\ \eta_{in}(t,\ C_l^{j-1},\ C^{j-2})\ \alpha(t,\ T,\ C_i^j,\ C_l^{j-1})\right] \\ a_{i\ End}^{C_l^{j-1}}\ \eta_{out}(T,\ C_l^{j-1},\ C^{j-2})\end{gathered} \tag{15}$$

Then at "(16)", we calculate χ to obtain the probability of vertical transition from C_l^{j-1} to C_i^j in which C_l^{j-1} is entered at time t before emitting response time observation RT_t to activate state C_i^j. Here, η_{in}, we consider the vertical transition from C^{j-1} (VM) that starts at time t to C^{j-2} root state (MS). We also consider the backward probability β, which is the transition from C_i^j to C^{j-1}. The η_{out} calculates the transition from C^{j-1} to C^{j-2} once C^{j-1} finishes at time e with observations $(rt_{t+1}, \cdots, rt_T|\lambda)$.

$$\begin{gathered}\chi(t,\ C_i^j,\ C_l^{j-1}) = \frac{\eta_{in}(t,\ C_l^{j-1},\ C^{j-2})\ \pi^{C_l^{j-1}}(C_i^j)}{P(RT|\lambda)} \\ \left[\sum_{e=t}^{T}\ \beta(t,\ e,\ C_i^j,\ C_l^{j-1})\ \eta_{out}(e,\ C_l^{j-1},\ C^{j-2})\right]\end{gathered} \tag{16}$$

After that, we calculate Production State Vertical transition (PSV), to obtain the number of vertical transitions χ from container substate S^{j-1} to the production service state S_i^j after finishing all the horizontal transitions between states S^j that are substate(s) from S^{j-1} at the same level as in "(17)".

$$\begin{gathered}\sum_{t=1}^{T-1}\ PSV\ (t, S_i^j,\ S^{j-1}) = \\ \left[\sum_{t=1}^{T}\ \chi(t,\ S_i^j,\ S^{j-1})\ +\ \sum_{t=2}^{T}\ Q(t,\ S_i^j,\ S^{j-1})\right]\end{gathered} \tag{17}$$

To learn the model, and to enhance the detection and identification, we train the model to obtain a new set of parameters for the state transitions probability $\hat{A}$, observation transition probability $\hat{B}$, and state transition hierarchy sequence. The training of HHMM is stored in the "Basic Configuration" file to be used in the future. The model is trained on observations until parameter convergence become fixed. To train the HHMM parameters on continuously updated response time, we use the sliding technique [28] to read the observations through specifying window size = 10-200 observations, and feed them to the HHMM model.

We compare the obtained sequence against the observed one to detect anomalous behaviour, and to obtain the anomalous path. If the observed sequence and detected sequence are similar, we can conclude that we have a model learned on normal behaviour, and an alarm will be issued for further investigation. Learning the model on normal behaviour aids in estimating the probability of each observed sequence. We call the sequence with the lowest estimated probabilities anomalous. The model tracks the detected anomaly to locate its root-cause. The located component(s) (service, container, VM, or microservice), will be stored in a storage with its probability, time of its activation, and time of detection.

VI. Evaluation

To assess the DLA, we investigate different anomaly scenarios, which are injected into TPC-W benchmark.

A. Setup Settings

TPC-W[1] benchmark is used for resource provisioning, scalability, and capacity planning for e-commerce. TPC-W

[1]http://www.tpc.org/tpcw/

emulates an online bookstore that consists of 3 tiers: client application, web server, and database. Each tier is installed on VM, and deployed on Kubernetes. For simplicity, we assume no database anomalies. To obtain historical data, TPC-W is run for 300 minutes, and workload is generated at different times. The workload of each container should be similar. The number of records that we obtain from the TPC-W was 2000 records, 50% is used for training, and the rest is used to validate the model. The model training lasted 150 minutes.

The experimental environment is consisted of four VMs (nodes) in two servers. Each server is equipped with Ubuntu 18.10, Xen 4.11[2], 2.7 GHz CPU and 8 GB RAM. The first server hosts 3 VMs (TPC-W benchmark), and the second one hosts 1 VM (Fault Injection). Each VM is equipped with LinuxOS (Ubuntu 18.10 version), one VCPU, 2GB of VRAM. The VMs are connected through a 100 Mbps network.

An agent is deployed on each VM to collect monitoring data from the components in SUT (e.g., host metrics, container, performance metrics, and workloads), and send them to the storage to be processed by the performance monitoring. We focus on CPU, Memory, Network, and throughput metrics, which are gathered from the web server, while the response time is measured from client's end. We further use docker $stats$ command to obtain live data stream for running containers, $mpstat$ to know the percentage of CPU utilization, and $vmstat$ to collect information about the memory usage. We use Heapster[3] to group the collected data, and store them in a time series database using InfluxDB[4]. The gathered data from the monitoring tool, and from datasets are stored in the Real-Time/Historical Data storage to enhance the future anomaly detection and identification.

The SUT services are running in containers, and the number of the replica of component is set to two. It means there will be two containers running the same service. The performance data of a service is the sum of all the containers running this service. Thus, the anomaly of a service is often caused by the anomalous behaviours of one or more containers belong to this service. By collecting the performance data of all the related containers, we can obtain the total performance data of a specific microservice.

We consider 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 resources (i.e., CPU time) at runtime. Also, we assume that containers are not resource bound to be able to show the overload at SUT. Because the actual amount of resource (i.e., CPU cycles) allocated to each container instance will differ based on the number of containers running on the same node, and the relative settings (i.e., CPU-share) assigned to containers.

[2]https://xenproject.org/

[3]https://github.com/kubernetes-retired/heapster

[4]https://www.influxdata.com/

B. Anomaly Scenarios

In the beginning of the experiment, the microservice operations get assigned initial delays, which are intended to resemble usual response times, and are not treated as actual injections. To simulate real anomalies of the system, script is written to inject different types of anomalies into the SUT (nodes and containers).

a) Resource Exhaustion: CPU Hog: such anomaly is injected to consume all CPU cycles by employing infinite loops. *Memory Leak*: an anomaly that exhausts a component memory. *Network Congestion*: components are injected with anomalies to send or accept a large amount of requests in network. Pumba[5] is used to cause network latency and package loss; to create CPU Hog and Memory leak, stress[6] is used.

b) Workload Contention: The TPC-W web server is emulated using client application, which generates workload (using Remote Browser Emulator) by simulating a number of user requests that is increased iteratively. Since the workload is always described by the access behaviour, we consider the container is gradually workloaded within [30-100000] emulated users requests, and the number of requests is changed periodically. The client application reports response time metric, and the web server reports throughput and resource utilization. We create a workload that runs for 2940 seconds. To measure the number of requests and response (latency), HTTPing[7] is installed on each node. Also ZipKin[8] is used to trace the request through the system.

C. Anomaly Detection and Localization

To represent the anomaly detection/identification of the anomaly scenarios for the SUT, different anomalies were injected at each component in the system one at a time. The components are chronologically ordered based on the period consumed from the beginning of injection until detection. Following shows different anomalies injection occurred for each component at different times.

a) Resource Exhaustion: Tables II, III, IV depict the anomalies injection, detection, and localization time for each component with injection time 300 sec for each component, and pause time 120 sec. The start and end times of each anomaly are logged. As shown in the tables, $VM_{2.1}$ and $C_{3.2}$ were the first components detected by the model for the CPU Hog and Memory leak respectively, while for Network congestion, $C_{3.1}$ was the first component captured by the DLA. Hence, we inject one component at a time, the reason behind the variation of the detection time returns to the variation in the injection time of anomalies. The localization of the components differed from one to another, however, the maximum time that the DLA took to locate the detected anomalous component was: 10 sec for the CPU Hog injection, 60 sec for the Memory leak, and 40 sec for Network Congestion.

[5]https://alexei-led.github.io/post/pumba_docker_netem/

[6]https://linux.die.net/man/1/stress

[7]https://www.vanheusden.com/httping/

[8]https://zipkin.io/pages/quickstart

TABLE II. CPU Hog Detection and Localization using HHMM

Components	Injection Time (start-end sec)	Detection Time (sec)	Localization Time (sec)
VM2.1	100-400	300	305
VM2.2	630-930	837	847
C3.1	1050-1350	1241	1246
C3.2	1470-1770	1713	1719
C3.3	1890-2190	2008	2013
C3.4	2310-2610	2563	2573

TABLE III. Memory Leak Detection and Localization using HHMM

Components	Injection Time (start-end sec)	Detection Time (sec)	Localization Time (sec)
C3.2	250-550	270	330
C3.4	670-970	680	740
VM2.2	1090-1390	1351	1404
VM2.1	1510-1810	1540	1585
C3.1	1930-2230	2140	2175
C3.3	2350-2650	2502	2532

b) Workload Contention: For the workload, we generate gradual user requests/sec at the container level (that contains TPC-W server). The number of users requests increases from 30 to 1000 with a pace of 100 requests/second incrementally, then we increase the requests from 1000 to 100,000 with a pace of 500 requests/second. Fig. 4 shows the effect of the workload (i.e., number of user requests) on the monitored metrics. The monitored metrics utilizations increase when the number of requests increases, then it remains constant. This means that the utilizations of CPU, memory, and network almost reached maximum capacity at 90%, 89%, and 25% respectively with the increasing number of user requests that reached 4000, 11500, 10000 requests/sec for CPU, memory and network. The result demonstrated that dynamic generated workloads have strong impact on the containers' metrics as the monitored containers were unable to process more than those requests. At that point the containers become overloaded and the monitored metrics are severely affected.

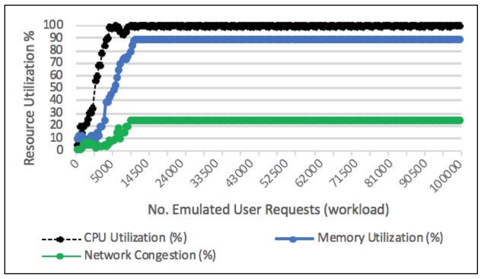


Fig. 4. Workload Contention - Metrics and Emulated User Requests.

TABLE IV. Network Congestion Detection and Localization using HHMM

Components	Injection Time (start-end sec)	Detection Time (sec)	Localization Time (sec)
C3.1	500-800	560	593
C3.2	920-1220	940	963
VM2.1	1340-1460	1350	1380
VM2.2	1580-880	1643	1678
C3.3	1700-2000	1710	1750
C3.4	1930-2230	2140	2160

D. Results and Discussion

The accuracy of results is compared with Dynamic Bayesian Network (DBN), and Hierarchical Temporal Memory (HTM). It is further evaluated based on different metrics such as: Root Mean Square Error (RMSE), Number of Correctly Detected Anomaly (CDA), Number of Correctly Identified Anomaly (CIA), Number of Incorrectly Detected Anomaly (IDA), and Number of Incorrectly Identified Anomaly (IIA). As show in Table V, the DLA using HHMM achieved good performance with 87.73% and 75.03% for CDA and CIA respectively. Also, the HTM relatively performed better than DBN with 79.54% for CDA, and 57.81% for CIA, and it gave good results for both CDA and CIA. However, in the testing phase in Table VI, the results of the HHMM, DBN and HTM for CDA and CIA increased comparing to the training phase. This may return to the configurations of the models.

TABLE V. Results Summary for Training Phase

Metrics	Results		
	HHMM	**DBN**	**HTM**
RMSE	0.2291	0.2593	0.2482
CDA	87.73%	73.16%	79.54%
CIA	75.03%	53.83%	57.81%
IDA	12.27%	26.84%	20.46%
IIA	24.97%	46.17%	40.19%

TABLE VI. Results Summary for Testing Phase

Metrics	Results		
	HHMM	**DBN**	**HTM**
RMSE	0.1521	0.2601	0.1593
CDA	97.73%	83.16%	91.89%
CIA	98.43%	94.01%	96.99%
IDA	2.27%	16.84%	8.11%
IIA	1.57%	5.99%	3.01%

We compare the efficiency of each algorithm in identifying correctly the root-cause with the number of samples (the number of anomalies in a dataset). Fig. 5 represents the accuracy of each algorithm in the anomaly identification. We define the Accuracy of Root-Cause Identification (ARCI) to be the number of anomalies successfully localized out of the total number of anomalies presented to the system to be localized. The higher the value, the more correct the identification made by the model. As presented by the figure, The root-cause is identified correctly by HHMM with an accuracy reached 97%. HHMM gives satisfying results on different samples. While the DBN achieved fair results with a moderate number of samples, its accuracy reduced once it reached 90 samples to be 93%. For the HTM, its accuracy fluctuated with the increasing number of samples to reach at 94% with 100 samples. It should be noted that HHMM achieved optimal results with 90 and 100 samples, while, DBN, and HTM achieved optimal results with (70-90), and (90, 100) samples respectively.

As shown in the previous experiments, the workload can significantly change depending on various events in runtime environment. Thus, we model our system hierarchically to detect and locate anomalous behaviour across different system levels through utilizing HHMMs. The mechanism of HHMMs is accompanied with our algorithms to detect and

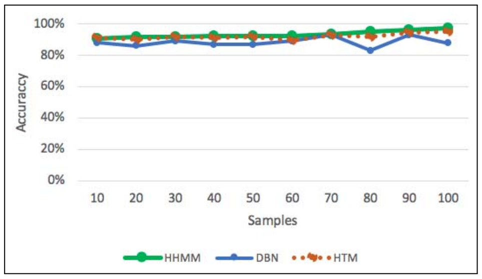


Fig. 5. The Accuracy of Anomaly Root-Cause Identification

identify anomalous behaviour. We conduct performance evaluation over small-scale computing clusters. We inject different anomalies, and we generate various workloads to measure the effects on the container-based microservice at different system settings. This allows us to find the relationships among system components at different levels. At the end, the results demonstrate that the DLA can detect and identify anomalous behaviour with high accuracy.

VII. Conclusions and Future Work

Monitoring and analyzing container-based microservice performance creates challenges in the area of resource performance management at run-time [26, 27]. The complexity of the distribution may for instance lead to performance variations (i.e., anomalous behaviour) observed at the response time metric. To mitigate and reduce the occurrence of anomalies, this paper proposes a Detection and Localization system for Anomalies (DLA) that analyzes (i.e., detects and identifies anomalies) based on the monitored performance of container-based microservices using Hierarchical HMM and Correlation Analysis. To assess the DLA performance, different anomalies and workload patterns are generated at different points in time to gather performance data in various conditions. The DLA is compared with different machine learning algorithms. The results demonstrate the ability of the DLA to detect and localize the anomalous behaviours accurately.

In the future, more anomaly scenarios, metrics, and more detailed experiments will be conducted to fully confirm these conclusions. Further, we will provide a self-healing mechanism [16, 18, 19, 21] to recover the localized anomaly.

References

[1] C. Pahl, P. Jamshidi, O. Zimmermann, "Architectural principles for cloud software," *ACM Trans on Internet Technology*, 2018.
[2] T. F. Düllmann, "Performance anomaly detection in microservice architectures under continuous change," Master, University of Stuttgart, 2017.
[3] C. Pahl, A. Brogi, J. Soldani, P. Jamshidi, "Cloud container technologies: a state-of-the-art review," *IEEE Transactions on Cloud Computing*, 2017.
[4] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L.E. Lwakatare, C. Pahl, S. Schulte, J. Wettinger, "Performance engineering for microservices: research challenges and directions," *ACM*, 2017.
[5] S. Fine, Y. Singer, and N. Tishby, "The hierarchical hidden markov model: analysis and applications," *Machine Learning*, vol. 32, no. 1, pp. 41–62, 1998.
[6] D. Taibi, V. Lenarduzzi, C. Pahl, "Architectural Patterns for Microservices: A Systematic Mapping Study," *CLOSER*, 2018.
[7] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *Intl Symposium on Network Computing and Applications*. 2016, pp. 27–34.
[8] C. Pahl, "An ontology for software component matching," *FASE Conference*, 2003.
[9] P. Jamshidi, C. Pahl, S. Chinenyeze, X. Liu, "Cloud migration patterns: a multi-cloud service architecture perspective," *ICSOC 2014 Workshops*, 2014.
[10] Q. Du, T. Xie, and Y. He, "Anomaly detection and diagnosis for container-based microservices with performance," in *Intl Conf on Algorithms and Architectures for Parallel Processing*. 2018.
[11] 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.
[12] P. Jamshidi, C. Pahl, N.C. Mendonca, "Pattern-based multi-cloud architecture migration," *Software: Practice & Experience*, 2017.
[13] J. Ohlsson, "Anomaly detection in microservice infrastructures," Master, School of Computer Science and Communication, 2018.
[14] C. Sauvanaud, M. Kaâniche, K. Kanoun, K. Lazri, and G. Da Silva Silvestre, "Anomaly detection and diagnosis for cloud services: Practical experiments and lessons learned," *Journal of Systems and Software*, vol. 139, pp. 84–106, 2018.
[15] 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.
[16] P. Jamshidi, A.M. Sharifloo, C. Pahl, A. Metzger, G. Estrada, "Self-learning cloud controllers: Fuzzy q-learning for knowledge evolution," *ICCAC*, 2015.
[17] H. Sukhwani, "A survey of anomaly detection techniques and hidden markov model," *International Journal of Computer Applications*, vol. 93, no. 18, pp. 975–8887, 2014.
[18] H. Arabnejad, C. Pahl, P. Jamshidi, G. Estrada, "A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling," *Intl Symp Cluster, Cloud and Grid Computing*, 2017.
[19] P. Jamshidi, C. Pahl, N.C. Mendonca, "Managing uncertainty in autonomic cloud elasticity controllers," *IEEE Cloud*, 2016.
[20] N. Ge, S. Nakajima, and M. Pantel, "Online diagnosis of accidental faults for real-time embedded systems using a hidden Markov model," *Simulation*, vol. 91, no. 19, pp. 851–868, 2016.
[21] 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.
[22] R. Scolati, I. Fronza, N. El Ioini, A. Samir, C. Pahl, "A Containerized Big Data Streaming Architecture for Edge Cloud Computing on Clustered Single-Board Devices,", *Closer*, 2019.
[23] G. Brogi, "Real-time detection of advanced persistent threats using information flow tracking and hidden markov," Doctoral dissertation, 2018.
[24] P. Jamshidi, C. Pahl, N.C. Mendonca, J. Lewis, S. Tilkov, *Microservices: The journey so far and challenges ahead*, IEEE Software 35 (3), 24-35, 2018.
[25] T. Taylor, "6 Container performance KPIs you should be tracking to ensure DevOps success," 2018.
[26] A. Samir and C. Pahl, "Anomaly Detection and Analysis for Clustered Cloud Computing Reliability,", *Intl Conf on Cloud Computing, GRIDs, and Virtualization*, 2019.
[27] A. Samir and C. Pahl, "Self-Adaptive Healing for Containerized Cluster Architectures with Hidden Markov Models,", *Intl Conf Fog & Mobile Edge Comp*, 2019.
[28] T. Chis, "Sliding hidden markov model for evaluating discrete data," 2014.