Anomaly Detection and Analysis for Clustered Cloud Computing Reliability

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

Claus Pahl
Faculty of Computer Science
Free University of Bozen-Bolzano
Bolzano, Italy
Email: Claus.Pahl@unibz.it

***Abstract*—Cloud and edge computing allow applications to be deployed and managed through third-party provided services that typically make virtualised resources available. However, often there is no direct insight into execution parameters at resource level, and only some quality factors can be directly observed while others remain hidden from the consumer. We investigate a framework for autonomous anomaly analysis for clustered cloud or edge resources. The framework determines possible causes of consumer-observed anomalies in an underlying provider-controlled infrastructure. We use Hidden Markov Models to map observed performance anomalies to hidden resources, and to identify the root causes of the observed anomalies in order to improve reliability. We apply the model to clustered hierarchically organised cloud computing resources.**

***Index Terms*—Cloud Computing; Edge Computing; Container Cluster; Hidden Markov Model; Anomaly; Performance.**

I. Introduction

Cloud and edge computing allow applications to be deployed and managed by third parties based on provided virtualised resources [2],[3]. Due to the dynamicity of computation in cloud and edge computing, consumers may experience anomalies in performance caused by the distributed nature of clusters, heterogeneity, or scale of computation on underlying resources that may lead to performance degradation and application failure: (1) change in cluster node workload demand or configuration updates may cause dynamic changes, (2) reallocation or removal of resources may affect the workload of system components. Recent works on anomaly detection [1],[4],[5] have looked at resource usage, rejuvenation or analyzing the correlation between resource consumption and abnormal behaviour of applications. However, more work is needed on identifying the reason behind observed resource performance degradations.

In a shared virtualised environment, some factors can be directly observed (e.g., application performance) while others remain hidden from the consumer (e.g., reason behind the workload changes, the possibility of predicting the future load, dependencies between affected nodes and their load). In this paper, we investigate the possible causes of performance anomalies in an underlying provider-controlled cloud infrastructure. We propose an anomaly detection and analysis framework for clustered cloud and edge environments that aims at automatically detecting possibly workload-induced performance fluctuations, thus improving the reliability of these architectures. We assume a clustered, hierarchically organised environment with containers as loads on the individual nodes, similar to container cluster solutions like Docker Swarm or Kubernetes.

System workload states that might be hidden from the consumer may represent anomalous or faulty behaviour that occurs at a point in time or lasts for a period of time. An anomaly may represent undesired behaviour such as overload or also appreciated positive behaviour like underload (the latter can be used to reduce the load from overloaded resources in the cluster). Emissions from those states (i.e., observations) indicate the possible occurrence of failure resulting from a hidden anomalous state (e.g., high response time). In order to link observations and the hidden states, we use Hierarchical Hidden Markov Models (HHMMs) [8] to map the observed failure behaviour of a system resource to its hidden anomaly causes (e.g., overload) in a hierarchically organised clustered resource configuration. Hierarchies emerge as a consequence of a layered cluster architecture that we assume based on a clustered cloud computing environment. We aim to investigate, how to analyse anomalous resource behaviour in clusters consisting of nodes with application containers as their load from a sequence of observations emitted by the resource.

This paper is organized as follows. Section II provided the related work. Section III explores our wider anomaly management framework. Section IV details the anomaly detection and fault analysis. Section V discusses evaluation concerns, followed by conclusions an future work.

II. Related Work

Several studies [9] and [5] have addressed workload analysis in dynamic environments. Sorkunlu et al. [10] identified system performance anomalies through analyzing the correlations in the resource usage data. Peiris et al. [11] analyzed the root causes of performance anomalies by combining the correlation and comparative analysis techniques in distributed environments. Dullmann et al. [12] provided an online performance anomaly detection approach that detects anomalies in performance data based on discrete time series analysis. Wang et al. [5] proposed to model the correlation between workload and the resource utilization of applications to characterize the system status. However, the technique neither classifies the different types of workloads, or recovers the anomalous behaviour. Maurya and Ahmad [14] proposed an algorithm that dynamically estimates the load of each node and migrates the task on the basis of predefined constraint. However, the algorithm migrates the jobs from the overloaded nodes to the underloaded one through working on pair of nodes, it uses a server node as a hub to transfer the load information in the network which may result in overhead at the node.

Many literatures used HMM, and its derivations to detect anomaly. In [15], the author proposed various techniques implemented for the detection of anomalies and intrusions in the network using HMM. Ge et al. [17] detected faults in real-time embedded systems using HMM through describe the healthy and faulty states of a system's hardware components. In [20] HMM is used to find which anomaly is part of the same anomaly injection scenarios.

III. Self-Adaptive Fault Management

Our ultimate goal is a self-adaptive fault management framework [7],[6],[21] for cloud and edge computing that automatically identifies anomalies by locating the reasons for degradations of performance, and making explicit the dependency between observed failures and possible faults cause by the underlying cloud resources.

A. The Fault Management Framework

Our complete framework consists of two models: (1) Fault management model that detects and identifies anomalies within the cloud system. (2) Recovery model that applies a recovery mechanism considering the type of the detected anomaly and the resource capacity. Figure 1 presents the overall approach. The focus in this paper is on the Fault management model.

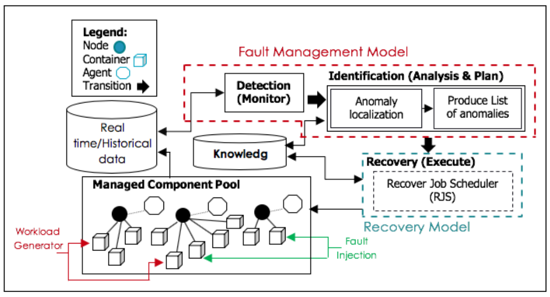

FIGURE 1. THE PROPOSED FAULT MANAGEMENT FRAMEWORK.

The cloud resources consist of a cluster, which composed of a set of nodes that host application containers as loads deployed on them. Each node has an agent that can deploy containers and discover container properties. We use the container notion to embody some basic principles of container cluster solutions [13] such as Docker Swarm or Kubernetes, to which we aim to apply our framework ultimately.

We align the framework with the Monitor Analysis, Plan, Execute based on the anomaly detection Knowledge (MAPE-K) feedback control loop. Monitor, collects data regarding the performance of the system as the observable state of each resource [16]. This can later be used to compare the detected status with the currently observed one. Each anomalous state has a weight (probability of occurrence). An identification step is followed by the detection to locate the root cause of anomaly (Analysis and Plan). The identified anomalous state is added to a queue that is ordered based on its assigned weight to signify urgency of healing. Knowledge about anomalous states are kept on record. Different recovery strategies (Execute) can mitigate the detected anomalies. Different pre-defined thresholds for recovery activities are assigned to each anomaly category based on observed response time failures.

The detection of an anomaly is based on using historical performance data to determine probabilities. We classify system data into two categories. The first one reflects observed system failures (essentially regarding permitted response time), and the second one indicates the (hidden) system faults related to workload fluctuations (e.g., by containers consuming a resource). We further annotate each behavioural category to reflect the severity of anomalous behaviour within the system, and the probability of its occurrence. The response time behaviour captures the amount of time taken from sending a request until receiving the response (e.g., creating container(s) within a node). For example, observed response time can fluctuate. The classified response time should be linked to the failure behaviour within system resources (i.e., CPU) to address unreliable behaviour. We can also classify the resource workload into normal (NL), overload (OL), underload (UL) categories to capture workload fluctuations.

B. Anomaly Detection and Identification

Anomaly detection, the Monitoring stage in MAPE-K, collects and classifies system data. It compares new collected data with previous observations based on the specified rules in the Knowledge component. Fault identification, the Analysis and Plan stages in MAPE-K, identifies the fault type and its root cause to explain the anomalous behaviour. The main aim of this step is specifying the dependency between faults (the proliferation of an anomaly within the managed resources), e.g., an inactive container can cause another container to wait for input. We use Hierarchical Hidden Markov models (HHMM) [8], a doubly stochastic model for hierarchical structures of data to identify the source of anomalies.

Based on the response time emissions, we track the path of the observed states in each observation window. Once we diagnose anomalous behaviour, the affected nodes will be annotated with a weight, which is a probability of fault occurrence for an observed performance anomaly. Nodes that have a high workload will be prioritised in the later fault handling based on the assigned weight. Nodes with the same weight can be addressed based on a first-detected-first-healed basis. In order to illustrate the usefulness of this analysis, we will also discuss the fault handling and recovery in the next subsection. Afterwards, we define the HHMM model structure and the analysis process in detail.

C. Fault Handling and Recovery

After detecting and identifying faults, a recovery mechanism, the Execute stage in MAPE-K, is applied to carry out load balancing or other suitable remedial actions, aiming to improve resource utilization. Based on the type of the fault, we apply a recovery mechanism that considers the dependency between nodes and containers. The recovery mechanism is based on current and historic observations of response time for a container as well as knowledge about hidden states (containers or nodes) that might have been learned. The objective of this step is to self-heal the affected resource. The recovery step receives an ordered weighted list of faulty states. The assigned probability of each state based on a predefined threshold is used to identify the right healing mechanism, e.g., to achieve fair workload distribution.

We specify the recovery mechanism using the following aspects: **Analysis**: relies on e.g., current observation, historic observation. **Observation**: indicates the type of observed failure (e.g., low response time). **Anomaly**: reflects the kind of fault (e.g., overload). **Reason**: explains the root causes of the problem. **Remedial Action**: explains the solution that can be applied to solve the problem. **Requirements**: steps and constraints that should be considered to apply the action(s). We will apply this to two sample strategies below.

D. Motivating Failure/Fault Cases and Recovery Strategies

In the following, we present two samples failure-fault situations, and suitable recovery strategies. The recovery strategies are applied based on the observed response time (current and historic observations), and its related hidden fault states. We illustrate two sample cases–overloaded neighbouring container and node overload.

1) Container Neighbour Overload (external dependency)

Analysis: based on current/historic observations, hidden states
Observation: low response time at the connected containers (overall failure to need performance targets).
Anomaly: overload in one or more containers results in underload for another container at different node.
Reason: heavily loaded container with external dependent one (communication)
Remedial Actions: *Option 1*: Separate the overloaded container and the external one depending on it from their nodes. Then, create a new node containing the separated containers considering the cluster capacity. Redirect other containers that in communication to these 2 containers in the new node. Connect current nodes with the new one, and calculate the probability of the whole model to know the number of transitions (to avoid the occurrence of overload), and to predict the future behaviour. *Option 2*: For the anomalous container, add a new one to the node that has the anomalous container to provide fair workload distribution among containers considering the node resource limits. Or, if the node does not yet reach the resource limits available, move the overloaded container to another node with free resource limits. At the end, update the node. *Option 3*: create another node within the node with anomalous container behaviour. Next, direct the communication of current containers to this node. We need to redetermine the probability of the whole model to redistribute the load between containers. Finally, update the cluster and the nodes. *Option 4*: distribute load. *Option 5*: rescale node. *Option 6*: do nothing, if the observed failure relates to regular system maintenance/update, then no recovery is applied.
Requirements: need to consider node capacity.

2) Node overload (self-dependency)

Analysis: current and historic observations
Observation: low response time at node level (a failure).
Anomaly: overloaded node.
Reason: limited node capacity.
Remedial Actions: *Option 1*: distribute load. *Option 2*: rescale node. *Option 3*: do nothing.
Requirements: collect information regarding containers and nodes, consider node capacity and rescale node(s).

IV. ANOMALY DETECTION AND ANALYSIS

A failure is the inability of a system to perform its required functions within specified performance requirements. Faults (or anomalies) describe an exceptional condition occurring in the system operation that may cause one or more failures. It is a manifestation of an error in system [22]. We assume that a failure is an undesired response time observed during system component runtime (i.e., observation). For example, fluctuations in workload are faults that may cause a slowdown in system response time (observed failure).

A. Motivation

As an example, Figure 2 shows several observed failures and related resource faults in a test environment. These failures occurred either at a specific time (e.g., F_1, F_9) or over a period of time (e.g., $F_2 - F_8$). These failures result from fluctuations in resource utilization (e.g., CPU). Utilization measures a resource's capacity that is in use. It aids us in knowing the resource workload, and aid us in reducing the amount of jobs from the overloaded resources, e.g., a resource is saturated when its usage is at over 50% of its maximum capacity.

The response time varies between high, low and normal categories. It is associated with (or caused by) resource workload fluctuations (e.g., overload, underload or normal load). The fluctuations in workload shall be categorised into states that reflect faults. The anomalous response time is the observed failure that we use initially to identify the type of workload that causes the anomalies. In more concrete terms, we can classify the response time by the severity of a usage anomaly on a resource: low response time (L) varies from $501 - 1000ms$, normal response time (N) reflects the normal operation time of a resource and varies from $201 - 500ms$, and high response time (H) occurs when a response time is less than or equal $200ms$, which can be used to transfer the workload from the heavy loaded resources to the underloaded resources.

As a result, the recovery strategy will differ based on the type of observed failure and hidden fault. The period of recovery, which is the amount of time taken to recover, differs based on: (1) the number of observed failures, (2) the volume of transferred data (nodes with many tasks require longer recovery time), and (3) network capacity.

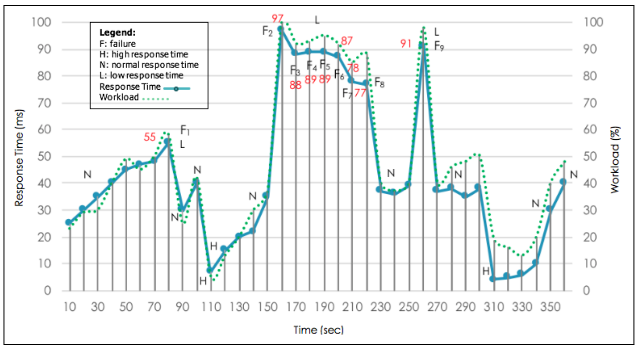

FIGURE 2. RESPONSE TIME AND WORKLOAD FLUCTUATIONS.

B. Observed Failure to Fault Mapping

The first problem is the association of underlying hidden faults to the observed failures. For the chosen metrics (e.g., resource utilization, response time), we can assume prior knowledge regarding (1) the dependency between containers, nodes and clusters; (2) past response time fluctuations for the executable containers; and (3) workload fluctuations that cause changes in response time. These can help us in identifying the mapping between anomalies and failures. An additional difficulty is the hierarchical organisation of clusters consisting of nodes, which themselves consist of containers. We associate an observed container response time to its cause at container, node, or cluster level, where for instance also a neighbouring container can cause a container to slow down. We define a mapping based on an analysis of possible scenarios.

The interaction between the cluster, node and container components in our architecture is based on the following assumptions. A cluster, which is the root node, is consisted of multiple nodes, and it is responsible for managing the nodes. A node, which is a virtual machine, has a capacity (e.g., resources available on the node such as memory or CPU). The main job of the node is to submit requests to its underlying substates (containers). Containers are self-contained, executable software packages. Multiple containers can run on the same node, and share the operating environment with other containers. Observations include the emission of failure from a state (e.g., high, low, or normal response time may emit from one or more states). Observation probabilities express the probability of an observation being generated from a resource state. We need to estimate the observation probabilities in order to know under which workloads large response time fluctuations occur and therefore to efficiently utilize a system resource while achieving good performance.

We need a mechanism that dynamically detects the type of anomaly and identifies its causes using this mapping. We identified different cases that may occur at container, node or cluster levels as illustrated in Figure 3. These detected cases will serve as a mapping between observable and hidden states, each annotated with a probability of occurrence that can be learned from a running system as a cause will often not be identifiable with certainty.

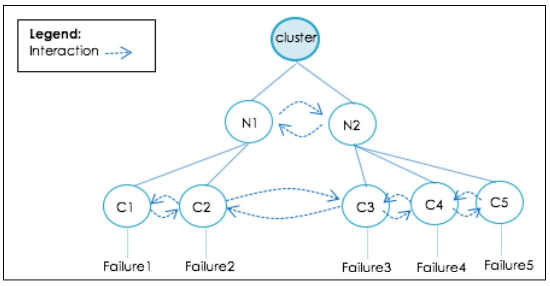

FIGURE 3. THE INTERACTION BETWEEN CLUSTER, NODES AND CONTAINER.

1) Low Response Time Observed at Container Level: There are different reasons that may cause this:

- *Case 1.1. Container overload (self-dependency)*: means that a container is busy, causing low response times, e.g., c_1 in N_1 has entered into load loop as it tries to execute its processes while N_1 keeps sending requests to it, ignoring its limited capacity.
- *Case 1.2. Container sibling overloaded (internal container dependency)*: this indicates another container c_2 in N_1 is overloaded. This overloaded container indirectly affects the other container c_1 as there is a communication between them. For example, c_2 has an application that almost consumes its whole resource operation. The container has a communication with c_1. At such situation, when c_2 is overloaded, c_1 will go into underload, because c_2 and c_1 share the resources of the same node.
- *Case 1.3. Container neighbour overload (external container dependency)*: this happens when a container c_3 in N_2 is linked to another container c_2 in another node N_1. In another case, some containers c_3, and c_4 in N_2 dependent on each other and container c_2 in N_1 depends on c_3. In both cases c_2 in N_1 is badly affected once c_3 or c_4 in N_2 are heavily loaded. This results in low response time observed from those containers.

2) Low Response Time Observed at Node Level: There are different reasons that cause such observations:

- *Case 2.1. Node overload (self-dependency)*: generally node overload happens when a node has low capacity, many jobs waited to be processed, or problem in network. Example, N_2 has entered into self load due to its limited capacity, which causes an overload at the container level as well c_3 and c_4.

- *Case 2.2. External node dependency*: occurs when low response time is observed at node neighbour level, e.g., when N_2 is overloaded due to low capacity or network problem, and N_1 depends on N_2. Such overload may cause low response time observed at the node level, which slow the whole operation of a cluster because of the communication between the two nodes. The reason behind that is N_1 and N_2 share the resources of the same cluster. Thus, when N_1 shows a heavier load, it would affect the performance of N_2.

3) Low Response Time Observed at Cluster Level (Cluster Dependency): If a cluster coordinates between all nodes and containers, we may observe low response time at container and node levels that cause difficulty at the whole cluster level, e.g., nodes disconnected or insufficient resources.

- *Case 3.1. Communication disconnection* may happen due to problem in the node configuration, e.g., when a node in the cluster is stopped or disconnected due to failure or a user disconnect.
- *Case 3.2. Resource limitation* happens if we create a cluster with too low capacity which causing low response time observed at the system level.

This mapping between anomalies and failures across the three hierarchy layers of the architecture needs to be formalised in a model that distinguishes observations and hidden states, and that allows weight to be attached. Thus, HHMMs are used to reflect the system topology.

C. Hierarchical Hidden Markov Model

Hierarchical Hidden Markov Model (HHMM) is a generalization of the Hidden Markov Model (HMM) that is used to model domains with hierarchical structure (e.g., intrusion detection, plan recognition, visual action recognition). HHMM can characterize the dependency of the workload (e.g., when at least one of the states is heavy loaded). The states (cluster, node, container) in HHMM are hidden from the observer, and only the observation space is visible (response time). The states of HHMM emit sequences rather than a single observation by a recursive activation of one of the substates (nodes) of a state (cluster). This substate might also be hierarchically composed of substates (containers). Each container has an application that runs on it. In case a node or a container emit observation, it will be considered a production state. The states that do not emit observations directly are called internal states. The activation of a substate by an internal state is a vertical transition that reflects the dependency between states. The states at the same level have horizontal transitions. Once the transition reaches to the End state, the control returns to the root state of the chain as shown in Figure 4. The edge direction indicates the dependency between states.

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

We choose HHMM as every state can be represented as a multi-levels HMM in order to:

1) show communication between nodes and containers,
2) demonstrate impact of workloads on the resources,
3) track the anomaly cause,
4) represent the response time variations that emit from nodes and containers.

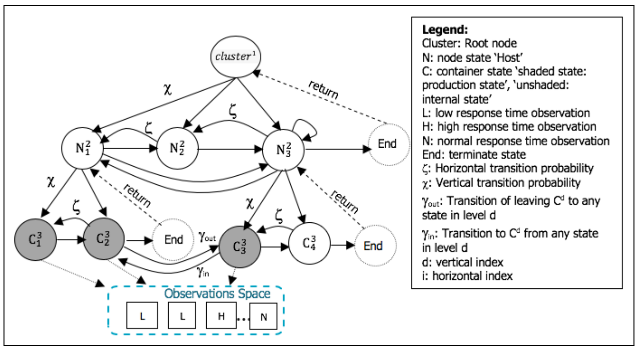
cluster¹
χ
return
ζ
ζ
N_1^2
N_2^2
N_3^2
End
χ
return
ζ
χ
ζ
return
C_1^3
C_2^3
End
γ_{out}
C_3^3
C_4^3
End
γ_{in}
Observations Space
L
L
H
N
Legend:
Cluster: Root node
N: node state 'Host'
C: container state 'shaded state: production state', 'unshaded: internal state'
L: low response time observation
H: high response time observation
N: normal response time observation
End: terminate state
ζ: Horizontal transition probability
χ: Vertical transition probability
γ_{out}: Transition of leaving C^d to any state in level d
γ_{in}: Transition to C^d from any state in level d
d: vertical index
i: horizontal index

FIGURE 4. HHMM FOR WORKLOAD.

D. Detection and Root Cause Identification using HHMM

Each state may show an overload, underload or normal load state. Each workload is correlated to the resource utilization such as CPU, and it is associated with response time observations that are emitted from container or node through the above case mapping. The existence of anomalous workload in one state not only affects the current state, but it may also affect the other states in the same level or across the levels. The vertical transitions in Figure 4 trace the fault and identify the fault-failures relation. The horizontal transitions show the request/reply transfered between states.

The observation O is denoted by $F_i = \{f_1, f_2, ..., f_n\}$ to refer to the response time observations sequence (failures). The substate and production states are denoted by N and C respectively. A node space SP containing a set of containers, $N_1^2 = \{C_1^3, C_2^3\}$, $N_3^2 = \{C_3^3, C_4^3\}$. Each container produces an observation that reflects the response time fluctuation, $C_1^3 = \{f_1\}$, $C_2^3 = \{f_1\}$, $C_3^3 = \{f_2\}$. A state C starts operation at time t if the observation sequence $(f_1, f_2, ..., f_{n-1})$ was generated before the activation of its parent state N. A state ends its operation at time t if the F_t was the last observation generated by any of the production states C reached from N, and the control returned to N from C_{end}. The state transition probability $A_{ij}^{N_i^d} = (a_{ij}^{N^d})$, $a_{ij}^{N^d} = P(N_j^{d+1} | N_i^{d+1})$ indicates the probability of making a horizontal transition from N_i^d to N_j^d. Both states are substates of $cluster^1$.

An observed low response time might reflect some overload (OL). This overload can occur for a period of time or at a specific time before the state might return to normal load (NL) or underload (UL). This fluctuation in workload is associated with a probability that reflects the state transition status from OL to NL ($PF_{OL \to NL}$) at a failure rate $\Re$, which indicates the number of failures for a N, C or $cluster$ over a period of time. Sometimes, a system resource remains OL/UL without returning to its NL. We reflect this type of fault as a self-transition overload/underload with probability PF_{OL} (PF_{UL}). Further, a self-transition is applied on normal load PF_{NL} to refer to continuous normal behaviour. In order to address the reliability of the proposed fault analysis, we define a fault rate based on the number of faults occurring during system execution $\Re(FN)$ and the length of failure occurrences $\Re(FL)$ as depicted in "(1)" and "(2)"

$$\Re(FN) = \frac{No\ of\ Detected\ Faults}{Total\ No\ of\ Faults\ of\ Resource} \quad (1)$$

$$\Re(FL) = \frac{Total\ Time\ of\ Observed\ Failures}{Total\ Time\ of\ Execution\ of\ Resource} \quad (2)$$

As failure varies over different periods of time, we can also determine the $Average\ Failure\ Length$ (AFL). These metrics feed later into a proactive recovery mechanism. Possible observable events can be linked to each state (e.g., low response time may occur for an overload state or normal load) to determine the likely number of failures observed for each state, and to estimate the total failures numbers for all the states. To estimate the probability of a sequence of failures (e.g., probability of observing low response time for a given state). Its sum is based on the probabilities of all failure sequences that generated by (q^{d-1}), and where (q_i^d) is the last node activated by (q^{d-1}) and ending at End state. This is done by moving vertically and horizontally through the model to detect faulty states. Once the model reaches the end state, it has recursively moved upward until it reaches the state that triggered the substates. Then, we sum all possible starting states called by the $cluster$ and estimate the probability.

We used the generalized Baum-Welch algorithm [8] to train the model by calculating the probabilities of the model parameters. As shown in "(3)" and "(4)", first, we calculate the number of horizontal transitions from a state to another, which are substates from q^{d-1}, using ξ as depicted in "(3)". The γ_{in} refers to the probability that the O is started to be emitted for $state_i^d$ at t. $state_i^d$ refers to container, node, or cluster. The γ_{out} refers to the O of $state_i^d$ were emitted and finished at t. Second, as in "(4)", $\chi(t, C_i^d, N_l)$ is calculated to obtain the probability that $state^{d-1}$ is entered at t before O_t to activate state $state_i^d$. The α, and β denote the forward and backward transition from bottom-up.

$$\xi(t, C_i^d, C_{End}^d, N_l) = \frac{1}{P(O|\lambda)} \left[\sum_{s=1}^{t} \gamma_{in}(N_l, cluster)\ \alpha(t, C_i^d, N_l)\right] a_{End}^{C_l} \gamma_{out}(t, C_l, cluster) \quad (3)$$

$$\chi(t, C_i^d, N_l) = \frac{\gamma_{in}(t, N_l, cluster)\pi^{N_l}(C_i^d)}{P(O|\lambda)} \left[\sum_{e=t}^{T} \beta(t, e, C_i^d, N_l)\gamma_{out}(e, N_l, cluster)\right] \quad (4)$$

The output of algorithm will be used to train Viterbi algorithm to find the anomalous hierarchy of the detected anomalous states. As shown in "(5)-(7)", we recursively calculate $\Im$ which is the ψ for a time set ($\bar{t} = \psi(t, t+k, C_i^d, C^{d-1})$), where ψ is a state list, which is the index of the most probable production state to be activated by C^{d-1} before activating C_i^d. $\bar{t}$ is the time when C_i^d was activated by C^{d-1}. The δ is the likelihood of the most probable state sequence generating $(O_t, \cdots, O_{(t+k)})$ by a recursive activation. The τ is the transition time at which C_i^d was called by C^{d-1}. Once all the recursive transitions are finished and returned to $cluster$, we get the most probable hierarchies starting from $cluster$ to the production states at T period through scanning the sate list ψ, the states likelihood δ, and transition time τ.

$$L = \max_{(1 \le r \le N_i^d)} \left\{ \delta(\bar{t}, t+k, N_r^{d+1}, N_i^d)\, a_{End}^{N_i^d} \right\} \quad (5)$$

$$\Im = \max_{(1 \le y \le N^{j-1})} \left\{ \delta(t, \bar{t}-1, N_i^d, N^{d-1}) a_{End}^{N^{d-1}}\ L \right\} \quad (6)$$

$$stSeq = \max_{cluster} \left\{ \delta(T, cluster), \tau(T, cluster), \psi(T, cluster) \right\} \quad (7)$$

Once we have trained the model, we compare the detected hierarchies against the observed one to detect and identify the type of workload. If the observed hierarchies and detected one are similar, and within the specified threshold, then the status of the observed component will be declared as 'Anomaly Free', and the framework will return to gather more data for further investigation. Otherwise, the hierarchies with the lowest probabilities will be considered anomaly. Once we detected and identified the workload type (e.g., OL), a path of faulty states (e.g., $cluster$, N_1^2, C_2^3 and C_3^3) is obtained that reflects observed failures. We repeat these steps until the probability of the model states become fixed. Each state is correlated with time that indicates: the time of it's activation, it's activated substates, and the time at which the control returns to the calling state. This aid us in the recovery procedure as the anomalous state will be recovered first come-first heal.

E. Workload and Resource Utilization Correlation

To check if the anomaly at cluster, node, container resource due to workload, we calculated the correlation between the workload (user transactions), and resource utilization to specify thresholds for each resource. The user transactions refer to the request rate per second. Thus, we used spearman's rank correlation coefficient to generate threshold to indicate the occurrence of fault at the monitored metric in multiple layers.

Our target is to group similar workload for all containers that run the same application in the same period. So that the workloads in the same period have the similar user transactions and resource demand. We added a unique workload identifier to the group of workloads in the same period to achieve traceability through the entire system. We utilized the probabilities of states transitions that we obtained from the HHMM to describes workload during T period. We transformed the obtained probabilities to get a workload behavior vector ω to characterize user transactions behaviors as in "(8)".

$$\omega = \{C_{i=1}^{d=3}, \cdots, C_{j=m}^{d=n}, \cdots, N_{i=1}^{d=2}, \cdots, N_{j=m}^{d=n}, \cdots, cluster\} \quad (8)$$

The correlation between the workload and resource utilization metric is calculated in the normal load behaviour to be a baseline. In case the correlation breaks down, then this refers to the existence of anomalous behaviour (e.g., OL).

V. Evaluation

The proposed framework is run on Kubernetes and docker containers. We deployed TPC-W[1] benchmark on the containers to validate the framework. We focused on three types of faults CPU hog, Network packet loss/latency, and performance anomaly caused by workload congestion.

A. Environment Set-Up

To evaluate the effectiveness of the proposed framework, the experiment environment consists three VMs. Each VM is equipped with LinuxOS, 3VCPU, 2GB VRAM, Xen 4.11 [2], and an agent. Agents are installed on each VM to collect the monitoring data from the system (e.g., host metrics, container, performance metrics, and workloads), and send them to the storage to be processed. The VMs are connected through a 100 Mbps network. For each VM, we deployed two containers, and we run into them TPC-W benchmark.

TPC-W benchmark is used for resource provisioning, scalability, and capacity planning for e-commerce websites. TPC-W emulates an online bookstore that consists of 3 tiers: client application, web server, and database. Each tier is installed on VM. We didn't considered the database tier in the anomaly detection and identification, as a powerful VM should be dedicated to the database. The CPU and Memory utilization are gathered from the web server, while the Response time is measured from client's end. We ran TPC-W for 300 min. The number of records that we obtained from TPC-W was 2000.

We used docker $stats$ command to obtain a live data stream for running containers. SignalFX Smart Agent[3] monitoring tool is used and configured to observe the runtime performance of components and their resources. We also used Heapster[4] to group the collected data, and store them in a time series database using InfluxDB[5]. The data from the monitoring and from datasets are stored in the Real-Time/Historical Data storage to enhance the future anomaly detection. The gathered datasets are classified into training and testing datasets 50% for each. The model training lasted 150 minutes.

[1]http://www.tpc.org/tpcw/
[2]https://xenproject.org/
[3]https://www.signalfx.com/
[4]https://github.com/kubernetes-retired/heapster
[5]https://www.influxdata.com/

B. Fault Scenarios

To simulate real anomalies of the system, script is written to inject different types of anomalies into nodes and containers. The anomaly injection for each component last 5 minutes to be in total 30 minutes for all the system components. The starting and end time of each anomaly is logged.

- CPU Hog: such anomaly is injected to consume all CPU cycles by employing infinite loops. The stress[6] tool is used to create pressure on CPU
- Network packet loss/latency: the components are injected with anomalies to send or accept a large amount of requests in network. Pumba[7] is used to cause network latency and package loss
- Workload contention: 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 behavior, we consider the container is gradually workloaded within [30-2000] emulated users requests, and the number of requests is changed periodically. The client application reports response time metric, and the web server reports CPU and Memory utilization. To measure the number of requests and response (latency), HTTPing[8] is installed on each node. Also AWS X-Ray[9] is used to trace of the request through the system.

C. Fault-Failure Mapping Detection and Identification

To address the fault-failure cases, the fault injection (CPU Hog and Network packet loss/latency) is done at two phases: (1) the system level (nodes), (2) components such as nodes and containers, one component at a time. The detection and identification will be differed as the injection time is varied from one component to another. The injection pause time between each injected fault is 180 sec.

a) Low Response Time Observed at Container Level: Case 1.1. Container overload (self-dependency): here, we added a new container C_5^3 in N_1^2, and we injected it by one anomaly at a time. For the CPU Hog, the anomaly was injected at 910 sec. It took from the model 30 sec to detect the anomaly and 15 sec to localize it. For the Network packet loss/latency, the injection of anomaly happened at 1135 sec, and the model detected and identified anomaly at 1145 and 1163 sec respectively.

Case 1.2. Container sibling overloaded (internal container dependency): in this case, the injection occurred at C_3^3 which in relation with C_4^3. The CPU injection began at 700 sec for C_3^3, the model detected the anomalous behaviour at 710 sec and localized it at 725 sec. For Network packet loss/latency, the injection of anomaly occurred at 905 sec. The model

[6]https://linux.die.net/man/1/stress
[7]https://alexei-led.github.io/post/pumba_docker_netem/
[8]https://www.vanheusden.com/httping/
[9]https://aws.amazon.com/xray/

needed 46 sec for the detection and 19 sec for the identification. For the C_4^3 the detection happened 34 sec later the detection of C_3^3 for the CPU Hog and the anomaly was identified at 754 sec. For the Network, the detection and identification occurred at 903 and 990 sec respectively.

Case 1.3. Container neighbour overload (external container dependency): at this case, a CPU Hog was injected at C_1^3 which in relation with C_3^3. The injection began at 210 sec. After training the HHMM, the model detected and localized the anomalous behaviour for C_1^3 at 225 and 230 sec. For Network fault, the injection occurred at 415 sec for C_1^3. The model took 429 sec for the detection and 450 sec for the identification. While for C_3^3, the CPU and Network faults were detected at 215/423 sec and identified at 240/429 sec.

b) Low Response Time Observed at Node Level: Case 2.1. Node overload (self-dependency): at this case we created a new node N_4^2 with small application and we injected the node by one anomaly at a time. For the CPU Hog, the anomaly was injected at N_4^2. The injection began at 413 sec. After training the HHMM, the model detected the anomalous behaviour at 443 sec and localized it at 461 sec. For the Network packet loss/latency, the injection of anomaly happened at 1210 sec, and the model detected and identified anomaly at 1260 and 1275 sec respectively.

Case 2.2. External node dependency: at such situation, a CPU Hog anomaly was injected at N_1^2. The injection began at 813 sec. After training the HHMM, the model detected the anomalous behaviour at 846 sec and localize it at 862 sec. For Network packet loss/latency, the injection of anomaly occurred at 1024 sec. The model needed 1084 sec for the detection and 1115 sec for the identification.

c) Low Response Time Observed at Cluster Level (Cluster Dependency): Case 3.1. Communication disconnection: at this case, we terminated N_1^2 and N_2^2, and we injected N_3^2 once with CPU Hog at 290 sec, and once with Network fault 525 sec. The detection and identification for each anomaly was: for CPU 335 and 345 sec respectively, and for Network was 585 sec for the detection and 610 sec for the identification.

Case 3.2. Resource limitation: at this case, we injected N_1^2, and N_3^2 at the same time with the CPU Hog fault to exhaustive the nodes capacity. The injection, detection, and identification were 1120, 1181, and 1192 sec. For the Network fault, the injection happened at 1372 sec, and the detection, and identification were at 1387, and 1392 sec.

D. Detection and Identification of Workload Contention

For the workload, to show the Influence of workload on CPU utilization monitored metric, we measured the response time (i.e., the time required to process requests), and throughput (i.e., the number of transactions processed during a period). We first generated gradual requests/sec at the container level. The number of users requests increases from 30 to 2000 with a pace of 10 users incrementally, and each workload lasts for 10 min. As shown in Figure 5, the results show that the throughput increases when the number of requests increases, then it remains constant once the number of requests reached 220 request/sec. This means that when the number of users requests is reached 220 request/sec, the utilization of CPU reached a bottleneck at 90%, and the performance degrades. On the other hand, the response time keep increasing with the increasing number of requests as shown in Figure 6. The result demonstrated that dynamic workloads has a noticeable impact on the container metrics as the monitored container was unable to process more than those requests. We also noticed that there is a linear relationship between the number of concurrent users and CPU utilization before resource contention in each user transaction behavior pattern. We calculated the correlation between the monitored metric, and the number of user requests. We obtained a strong correlation between the two measured variables reached 0.25775 for two variables. The result concludes that the number of requests influences the performance of the monitored metrics.

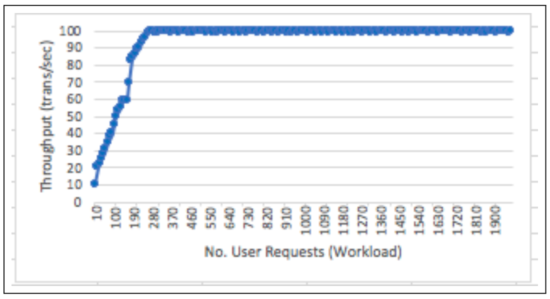

FIGURE 5. WORKLOAD - THROUGHPUT AND NO. OF USER REQUESTS.

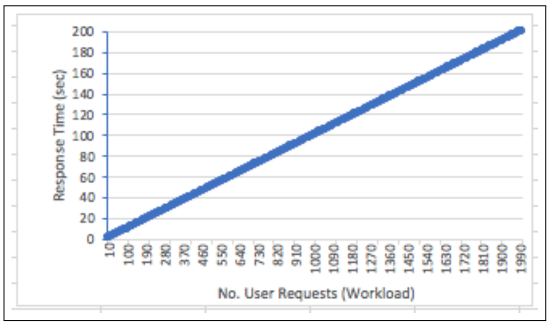

FIGURE 6. WORKLOAD - RESPONSE TIME AND NO. OF USER REQUESTS.

E. Assessment of Detection and Identification

The model performance is compared with other techniques such as Dynamic Bayesian Network (DBN), and Hierarchical Temporal Memory (HTM). To evaluate the effectiveness of anomaly detection, common measures in anomaly detection are used:

Root Mean Square Error (RMSE) measures the differences between detected and observed value by the model. A smaller RMSE value indicates a more effective detection scheme.

Mean Absolute Percentage Error (MAPE) measures the detection accuracy of a model. Both RMSE and MAPE are negatively-oriented scores, i.e., lower values are better.

Number of Correctly Detected Anomaly (CDA) It measures percentage of the correctly detected anomalies to the total number of detected anomalies in a given dataset. High CDA indicates the model is correctly detected anomalous behaviour.

Recall It measures the completeness of the correctly detected anomalies to the total number of anomalies in a given dataset. Higher recall means that fewer anomaly cases are undetected.

Number of Correctly Identified Anomaly (CIA) CIA is the number of correct identified anomaly (NCIA) out of the total set of identification, which is the number of correct Identification (NCIA) + the number of incorrect Identification (NICI)). The higher value indicates the model is correctly identified anomalous component.

$$CIA = \frac{NCIA}{NCIA + NICI} \tag{9}$$

Number of Incorrectly Identified Anomaly (IIA) is the number of identified components which represents an anomaly but misidentified as normal by the model. A lower value indicates that the model correctly identified anomalies.

$$IIA = \frac{FN}{FN + TP} \tag{10}$$

FAR The number of the normal identified component which has been misclassified as anomalous by the model.

$$FAR = \frac{FP}{TN + FP} \tag{11}$$

The false positive (FP) means the detection/identification of anomaly is incorrect as the model detects/identifies the normal behaviour as anomaly. True negative (TN) means the model can correctly detect and identify normal behaviour as normal.

TABLE I. Validation Results.

Metrics	HHMM	DBN	HTM
RMSE	0.23	0.31	0.26
MAPE	0.14	0.27	0.16
CDA	96.12%	91.38%	94.64%
Recall	0.94	0.84	0.91
CIA	94.73%	87.67%	93.94%
IIA	4.56%	12.33%	6.07%
FAR	0.12	0.26	0.17

The results in Table I depicted that both HHMM and HTM achieved good results for the detection and identification. While the results of the DBN a little bit decayed for the CDA with approximately 5% than HHMM and 3% than HTM. The three algorithms can detect obvious anomalies in the datasets. Both HHMM and HTM showed higher detection accuracy as they are able to detect temporal anomalies in the dataset. The result interferes that the HHMM is able to link the observed failure to its hidden workload.

VI. Conclusion and Future Work

This paper presented a framework for the detection and identification of anomalies in clustered computing environments. The key objective was to provide an analysis feature that maps observable quality concerns onto hierarchical hidden resources in a clustered environment and their operation in order to identify the reason for performance degradations and other anomalies. We used hidden hierarchical Markov models (HHMM) to reflect the hierarchical nature of the unobservable resources. We have analysed mappings between observations and resource usage based on a clustered container scenario. To evaluate the performance of proposed framework, HHMM is compared with other machine learning algorithms such as Dynamic Bayesian Network (DBN), and Hierarchical Temporal Memory (HTM). The results show that the proposed framework is able to detect and identify anomalous behavior with more than 96%.

In the future, we aim to fully implement the framework, and carry out further experimental evaluations to fully confirm these conclusions. Further, we will provide a self-healing mechanism to recover the localized anomaly. More practical concerns from microservices and container architectures shall also be investigated [19],[18]

References

[1] X. Chen, C.-D. Lu, and K. Pattabiraman, “Failure Prediction of Jobs in Compute Clouds: A Google Cluster Case Study,” *International Symposium on Software Reliability Engineering, ISSRE*, pp. 167–177, 2014.

[2] C. Pahl, P. Jamshidi, O. Zimmermann, “Architectural principles for cloud software,” ACM Transactions on Internet Technology (TOIT) 18 (2), 17, 2018.

[3] D. von Leon, L. Miori, J. Sanin, N. El Ioini, S. Helmer, C. Pahl, “A Lightweight Container Middleware for Edge Cloud Architectures,” Fog and Edge Computing: Principles and Paradigms, 145-170, 2019.

[4] G. C. Durelli, M. D. Santambrogio, D. Sciuto, and A. Bonarini, “On the Design of Autonomic Techniques for Runtime Resource Management in Heterogeneous Systems,” PhD dissertation, Politecnico di Milano, 2016.

[5] T. Wang, J. Xu, W. Zhang, Z. Gu, and H. Zhong, “Self-adaptive cloud monitoring with online anomaly detection,” *Future Generation Computer Systems*, vol. 80, pp. 89–101, 2018.

[6] 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.

[7] P. Jamshidi, A. Sharifloo, C. Pahl, A. Metzger, G. Estrada, “Self-learning cloud controllers: Fuzzy q-learning for knowledge evolution,” Intl Conference on Cloud and Autonomic Computing, 208-211, 2015.

[8] 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.

[9] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, “Workload Prediction Using ARIMA Model and Its Impact on Cloud Applications,” *IEEE Transactions on Cloud Computing*, vol. 3, no. 4, pp. 449–458, oct 2015.

[10] N. Sorkunlu, V. Chandola, and A. Patra, “Tracking System Behavior from Resource Usage Data,” in *Pro-*

ceedings - IEEE International Conference on Cluster Computing, ICCC, 2017, pp. 410–418.

[11] M. Peiris, J. H. Hill, J. Thelin, S. Bykov, G. Kliot, and C. Konig, “PAD: Performance anomaly detection in multi-server distributed systems,” in *Intl Conference on Cloud Computing, CLOUD*, 2014.

[12] T. F. Düllmann, “Performance Anomaly Detection in Microservice Architectures Under Continuous Change,” Master, University of Stuttgart, 2016.

[13] C. Pahl, A. Brogi, J. Soldani, P. Jamshidi, “Cloud container technologies: a state-of-the-art review,” IEEE Transactions on Cloud Computing, 2018.

[14] S. Maurya and K. Ahmad, “Load Balancing in Distributed System using Genetic Algorithm,” *Intl Jrnl of Engineering and Technology*, vol. 5, no. 2, 2013.

[15] H. Sukhwani, “A Survey of Anomaly Detection Techniques and Hidden Markov Model,” *Intl Jrnl of Computer Applications*, vol. 93, no. 18, pp. 975–8887, 2014.

[16] 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.

[17] N. Ge, S. Nakajima, and M. Pantel, “Online diagnosis of accidental faults for real-time embedded systems using a hidden Markov model,” *SIMULATION*, pp. 0 037 549 715 590 598—-, 2015.

[18] 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.

[19] D. Taibi, V. Lenarduzzi, C. Pahl, “Architecture Patterns for a Microservice Architectural Style,” Springer, 2019.

[20] G. Brogi, “Real-time detection of Advanced Persistent Threats using Information Flow Tracking and Hidden Markov,” Doctoral dissertation, 2018.

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

[22] IEEE, “IEEE Standard Classification for Software Anomalies (IEEE 1044 - 2009),” pp. 1–4, 2009.