A Controller Architecture for Anomaly Detection, Root Cause Analysis and Self-Adaptation for Cluster Architectures

Areeg Samir

Faculty of Computer Science
Free University of Bozen-Bolzano
Bolzano, Italy

Claus Pahl

Faculty of Computer Science
Free University of Bozen-Bolzano
Bolzano, Italy

***Abstract*—Service-based cloud computing allows applications to be deployed and managed through third-party provided services, making typically virtualised resources available. However, often there is no direct access to platform-level execution parameters of a provided service, and only some quality properties can be directly observed while others remain hidden from the service consumer. We introduce a controller architecture for autonomous, self-adaptive anomaly remediation in this semi-hidden setting. The controller determines the possible causes of consumer-observed anomalies in an underlying provider-controlled infrastructure. We use Hidden Markov Models to map observed performance anomalies into hidden resources, and to identify the root causes of the observed anomalies. We apply the model to a clustered computing resource environment that is based on three layers of aggregated resources.**

***Index Terms*—Cloud Computing; Container Clusters; Hidden Markov Model; Workload; Anomaly; Performance.**

I. INTRODUCTION

Cloud and Edge computing are examples of services being provided to allow applications to be deployed and managed by third-party providers that make shared virtualised resources accompanied by dynamic management facilities available [2],[3]. Due to the dynamic nature of loads in a distributed cloud and edge computing setting, consumers may experience anomalies (e.g., in our case variation in a resource performance) due to distribution, heterogeneity, or scale of computing that may lead to performance degradation and potential application failures. Furthermore, loads might vary over time: (i) changes of the load on individual resources, (ii) changing workload demand and prioritisation, (iii) reallocation or removal of resources in dynamic environments. These may affect the workload of current system components (container, node, cluster), and may require rebalancing their workloads. Recent studies [1],[4],[5] have looked at resource usage, rejuvenation, or analyzing the correlation between resource consumption and abnormal behavior of applications. Less attention has been given to the possibly hidden reason behind the occurrence of an observable performance degradation (root cause)[23], and how to deal with the degradation in a hierarchically organised cluster setting.

To handle these challenges in a shared virtualised environment, third party providers provide some factors that can be directly observed (e.g., the response time of service activations) while others remain hidden from the consumer (e.g., the reason behind the workload, the possibility to predict the future load behaviour, the dependency between the affected nodes and their loads in a cluster).

We differentiate between two types of observation in relation to workload and response time fluctuations: **System states (anomaly/fault)** that refer to anomalous or faulty behavior, which is hidden from the consumer. This indicates that the behavior of a system resource is significantly different from normal behavior. An anomaly in our case may point to an undesirable behavior of a resource such as overload, or to a desirable behavior like underload of a system resource, which can be used as a solution to reduce the load at overloaded resources. **Emission or Observation (observed failure from these states)**, which indicates the occurrence of failure resulting from a hidden state.

To address this problem, we use Hierarchical Hidden Markov Models (HHMMs) [8] as a stochastic model to map the observed failure behavior of a system resource to its hidden anomaly causes (e.g., overload) through tracking the detected anomaly to locate its root cause. We implement the proposed controller for a clustered computing resource environment. The contribution of this paper is a controller [7],[6] that automatically detects the anomalous behavior within a cluster of containers running on cluster nodes, where a sequence of observations is emitted by the system resource. The controller remedies the detected anomalies that occur at the container, node or cluster level. To achieve that this paper: (i) analysed the possible causes of observable anomalies in an underlying provider-controlled infrastructure; (ii) defined an anomaly detection, and analysis controller for a self-adaptive cluster environment, that automatically manages the resource workload fluctuations. The paper objective is to introduce the controller in terms of its architecture and processing activities.

The paper is organized as follows. Section II reviews related work. Section III explains the motivation behind our work. Section IV gives an overview of HHMM. Section V explains the mapping of failure and fault. Section VI explains the controller architecture. Section VII evaluates it.

 ISBN: 978-1-61208-706-1

II. Related Work

There are a number of studies that have addressed workload analysis in dynamic environments [1],[4],[5]. They proposed various methods for analyzing and modeling workload.

Dullmann [13] provide an online performance anomaly detection approach that detects anomalies in performance data based on discrete time series analysis.

Peiris et al. [14] analyze root causes of performance anomalies by combining the correlation and comparative analysis techniques in distributed environments. Sorkunlu et al. [15] identify system performance anomalies through analyzing the correlations in the resource usage data. Wang et al. [5] propose to model the correlation between workload and the resource utilization of applications to characterize the system status.

Maurya and Ahmad [16] propose an algorithm that dynamically estimates the load of each node and migrates the task if necessary. The algorithm migrates the jobs from overloaded nodes to underloaded ones 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.

Moreover, many literatures used HMM and its derivations to detect anomaly. In [17], the author proposed various techniques implemented for the detection of anomalies and intrusions in the network using HMM. In [19] the author detected faults in real-time embedded systems using HMM through describing the healthy and faulty states of a system's hardware components. In [21], 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 the anomalous behaviour in containerized cluster environment [20] through considering the influence of dynamic workloads on their anomaly detection solutions. The proposed controller 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) *Identification*, which tracks the cause of the detected anomaly. (4) *Recovery*, that heals the identified anomalous components. (5) *Anomaly injection*, which simulates different anomalies, and gathers dataset of performance data representing normal and abnormal conditions.

III. Motivating Example

A failure is the inability of a component to perform its functions with respect to a specified (e.g., performance) requirements [18]. Faults (also called anomalies) are system properties that describe an exceptional condition occurring in the system operation that may cause one or more failures [24].

We assume that a failure is a kind of unexpected response time observed during system component runtime (i.e., observation), while fluctuations occurring during a resource execution of a component are considered as faults or anomalies (state of a hidden component). For example, fluctuations in workload such as overload faults may cause delay in a system response time (observed failure).

Generally, the observed metrics do not provide enough information to identify the cause of an observed failure. For example, the CPU utilization of a containerized application is about 30% with 400 users, and it increases to about 70% with 800 users in the normal situation. Obviously, the system is normal with 800 users. But probably the system shows anomalous behaviour with 400 users, when the CPU utilization is about 70%. Thus, it is hard to identify whether the system is normal or anomaly just based on the CPU utilization. Thus, specifying a threshold for the utilization of resource without considering the number of users, will raise anomalous behaviours. Consequently, it is important to integrate the data of workload into anomaly detection and identification solutions. Once provided with a link between faults (workloads) and failures (response time) emitted from components, we can also apply a suitable recovery strategy depending on the type of identified fault.

Thus, a self-adaptation controller will be introduced later in this paper to automatically manage faults through identifying the degradation of performance, determining the dependency between faults and failures, and applying recovery strategies. We can align the steps of the fault management with the Monitoring, Analysis, Planning, Execution, and Knowledge (MAPE-K) control loop as a conceptual framework.

IV. Hierarchical Hidden Markov Model (HHMM)

Hierarchical Hidden Markov Model (HHMM) [8] 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 1. The edge direction indicates the dependency between states.

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 the impact of workloads on the resources, (3) track the anomaly cause, and (4) represent the response time variations that emit from nodes or containers.

 ISBN: 978-1-61208-706-1

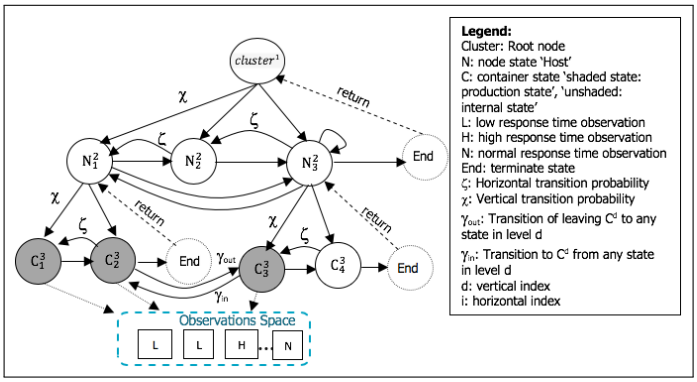

FIGURE 1. THE HHMM FOR WORKLOAD.

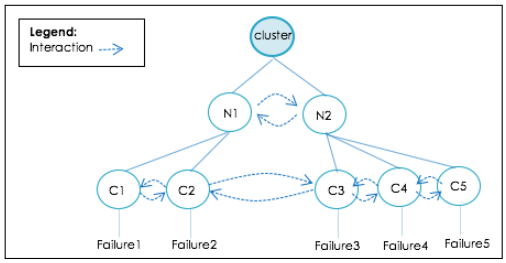

FIGURE 2. DEPENDENCIES BETWEEN CLUSTER, NODES AND CONTAINERS.

V. FAILURE-TO-FAULT MAPPING

Based on analyzing the log file and monitored metrics from existing systems, we can obtain knowledge regarding (1) the dependencies between containers, nodes and clusters; (2) response time fluctuations emitted from containers or nodes; (3) workload fluctuations that cause changes in response time. We need a mechanism that automatically maps a type of anomaly to its causes. We can identify different failure-fault cases that may occur at container, node or cluster level as illustrated in Figure 2. We focused on addressing the correlation between workload (overload) and the response time at container, node, and cluster.

A. 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 all resources. The container has a communication with c_1. At such situation, when c_2 is overloaded, c_1 will go into underload. The reason is that 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.

B. 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.

C. Low Response Time Observed at Cluster Level

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.

The mapping between faults and failures needs to be formalised in a model that distinguishes observations and hidden states. Thus, HHMM is used to reflect the system topology.

VI. SELF-ADAPTIVE CONTROLLER ARCHITECTURE

This section explains the controller architecture (Figure 3).

A. Managed Component Pool

The system under observation consists of a cluster that is composed of a set of nodes that host containers as the application components. A node could be a virtual machine that has a given capacity. The main job of the node is to assign requests to its containers. Containers are stand-alone,

 ISBN: 978-1-61208-706-1

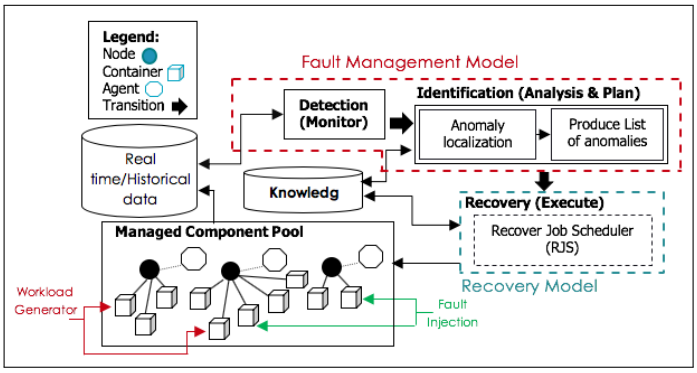

FIGURE 3. THE CONTROLLER ARCHITECTURE.

executable packages of software. Multiple containers can run on the same node, and share the operating environment with other containers. Each component either cluster, node, or container may emit observations. Observations are emissions of failure from a component resource.

We installed an agent on each node to collect metrics from the pool, and to expose log files of containers and nodes to Real-Time/Historical Data storage. The agent adds data interval function to determine the time interval at which the data collected belongs. The data interval function specifies the lower and upper limits for the data arrivals. The response time, and the state of the component are assigned to each interval. Moreover, the agent gathers data regarding the workload (i.e., no. of requests issued to component), and monitored metrics (i.e., CPU, Memory) to characterize the workload of components processed in an interval. The agent push the data to be stored in the Real time/Historical storage to be used by the Fault Management Model.

B. Fault Management Model

The model is based on the history of the overall system performance. This can be used to compare the predicted status with the currently observed one to detect anomalous behaviour. The fault management model consists of:

a) Detection (Monitor): To detect anomaly, the monitor collects system data from the Real time/Historical storage. Then, it checks if there is anomalous behaviour at the managed components through utilizing spearman's rank correlation coefficient to estimate the dissociation between the response time and the number of requests (workload). If there is a decrease in the correlation degree, then the metric is not associated with the increasing workload, which means the observed variation in performance isn't an anomaly. In case the correlation degree increase, this refers to the existence of anomaly occurred as the impact of dissociation between the workload and the response time exceeds a certain value. To achieve that we wrote an algorithm to be used as a general threshold to highlight the occurrence of anomaly in the managed pool under different workloads. We added a unique workload identifier to the group of workloads in the same period to achieve traceability through the entire system. We specified that the degree of dissociation (DD = 15) can be used as an indicator for performance degradation considering different response time, and different workloads. The value of DD will be compared against the monitored metrics (i.e., CPU, Memory utilization) to detect anomalous behaviour within the system. In case an anomaly is detected, the controller will move to the fault management to track the cause of anomaly in the system.

b) Identification (Analysis and Plan): Once there is appearance of anomaly, we built HHMMs to identify anomalies in system components as shown in Figure 1.

The HHMM vertically calls one of its substates $N_1^2 = \{C_1^3, C_2^3\}$, N_2^2, $N_3^2 = \{C_3^3, C_4^3\}$ with "vertical transition χ" and d index (superscript), where $d = \{1, 2, 3\}$. Since N_1^2 is abstract state, it enters its child HMM substates C_1^3 and C_2^3. Since C_2^3 is a production state, it emits observations, and may make horizontal transition γ, with i horizontal index (subscript), where $i = \{1, 2, 3, 4\}$, from C_1^3 to C_4^3. 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 N_1^2. Once the control returns to the state N_1^2, it makes a horizontal transition (if exist) to state N_2^2, which horizontally transits to state N_3^2. State N_3^2 has substates C_3^3 that transits to C_4^3 which may transit back to C_3^3 or transit to the End state. Once all the transitions under this node are achieved, the control returns to N_3^2. State N_3^2 may loop around, transit back to N_2^2, or enters its End state, which ends the whole process and returns control to the cluster. The model can't horizontally do transition unless it vertically transited. Further, the internal sates don't need to have the same number of substates. It can be seen that N_1^2 calls containers C_1^3 and C_2^3, while N_2^2 has no substates. The horizontal transition between containers reflect the request/reply between the client/server in our system under test, and the vertical transition refers to child/parent relationship between containers/node.

The observation O is denoted by $F_i = \{f_1, f_2, ..., f_n\}$ to refer to the response time observations sequence (failures). An observed low response time might reflect workload fluctuation. 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.

We used the generalized Baum-Welch algorithm [8] to train the model by calculating the probabilities of the model parameters: (1) the horizontal transitions from a state to another. (2) probability that the O is started to be emitted for $state_i^d$ at t. $state_i^d$ refers to container, node, or cluster. (3) the O of $state_i^d$ were emitted and finished at t. (4) the probability that $state^{d-1}$ is entered at t before O_t to activate state $state_i^d$. (5) the forward and backward transition from bottom-up.

The output of algorithm will be used to train Viterbi algorithm to find the anomalous hierarchy of the detected anomalous states. As shown in "(1)-(3)", 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

 ISBN: 978-1-61208-706-1

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.

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

$$\Im = \max_{(1 \le y \le N^{j-1})} \left\{ \delta(t, \bar{t}-1, N_i^d, N^{d-1}) a_{End}^{N^{d-1}}\, L \right\} \quad (2)$$

$$stSeq = \max_{cluster} \left\{ \delta(T, cluster), \tau(T, cluster), \psi(T, cluster) \right\} \quad (3)$$

Once we have a trained the model, we compare the detected hierarchies against the observed one to identify the type of workload. The hierarchies with the lowest probabilities will be considered anomaly. Once we detected and identified the workload type (e.g., OL), a hierarchy of faulty states (e.g., $cluster$, N_1^2, C_1^3 and C_2^3) that are affected by the anomalous component (C_1^3) is obtained that reflects observed anomalous behaviour. 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 sub-states, and the time at which the control returns to the calling state. The result of the fault management model (anomalous components) is stored in Knowledge storage. This aid us in the recovery procedure as the anomalous state will be recovered first come-first heal.

C. Fault-Failure Recovery Cases

Based on the fault type, we apply a recovery mechanism that considers the dependencies between components, and the current component status. The recovery mechanism is specified based on historic and current observations of a response time for a container or node and the hidden states (containers or nodes). The following steps and concerns are considered by the recovery mechanism:

- Analysis: relies on current and historic observation.
- Observation (failure): indicates the type of observed failure (e.g., low response time).
- Anomaly (fault): reflects the fault type (e.g., overload).
- Reason: explains the causes of the problem.
- Remedial Action: explains different solutions that can be applied to solve the problem.
- Requirements: constraint that might apply.

We look at two anomaly cases and suitable recovery strategies, which exemplify recovery strategies for the fault-failure mapping cases 1.3 and 2.1. These strategies can be applied based on the observed response time (current and historic observations) and related faults (hidden states).

1) Container neighbour overload (external container dependency) **Analysis**: current/historic observations, hidden states
Observation (failure): low response time at the anomalous container and the dependent one.
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 (MM) node within the node with anomalous container behaviour. Next, direct the communication of current containers to (MM). 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, this means that the observed failure relates to regular system maintenance or update happened to the system. Thus, no recovery option will be applied.
Requirements: need to consider node capacity.

2) Node overload (self-dependency) **Analysis**: current and historic observations
Observation (failure): low response time at node level.
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).

D. Recovery Model

The recovery model (Execute stage in MAPE-K) receives an ordered list of faulty states from the identification step. It applies a recovery mechanism considering the type of the identified anomaly and the resource capacity. We have configured the fault management model to have a specific number of nodes and containers because increasing the number of nodes and containers will lead to a large amount of different recovery actions (Load balancing rules), which reduces model performance. We are mainly concerned with two workload anomalies: (1) overload as it reflects anomalous behavior,

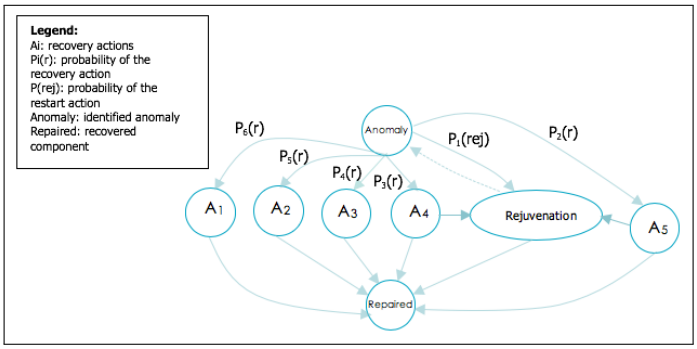

FIGURE 4. HMM FOR RECOVERY ACTIONS.

(2) underload category, as it is considered anomaly but it represents a solution to migrate load from heavy loaded component. We define different recovery actions for each fault-failure case. Consequently, for an identified anomaly case, we need to select the most appropriate action from the time and cost perspectives.

The Recover Job Scheduler (RJS) heals the identified anomaly based on first identified-first heal. It mitigates the anomalous state, by distributing the load to the underloaded components considering their status. The recovery actions are stored in the Knowledge storage to keep track of the number of applied actions to the identified anomalous component. Before applying any of the recovery option, "Restart" option will be applied to save the cost of trying multiple recovery options if the component doesn't reach its restart action number limit. In case a restart option doesn't enhance the situation, RJS checks the existence of underloaded component identified by the fault management model and stored in the knowledge storage. If there is underloaded component, the HMM is trained using the Forward-Backward algorithm to select the most probable action for the anomalous component as shown in Figure 4. The states A_i in the model refer to the hidden recovery actions. The $rejuventation$ hidden state refers to the restart action, and $P_i(r)$, is the probability of the recovery actions. We estimated $P_i(r)$ based on computing the maximum likelihood. The result of the HMM will be, for instance, the most probable action for anomalous state C_1^3 is 'distribute load'. The RJS apply the selected action to the fault component in the "Managed Component Pool". In case, RJS couldn't find underloaded components, the "pause" action will be applied. If the number of applied recovery actions for the anomalous component exceeds a predefined threshold, terminate action will be applied after backing up the component. For each component, we further keep a profile of the type of applied action to enhance the recovery procedure in the future.

a) Metrics for Recovery Plan Determination: In order to better capture the accuracy of the proposed fault identification, we estimated the Fault Rate to capture (1) the number of fault during system execution $\Re(FN)$, and (2) the overall length of failure occurrence $\Re(FL)$ as depicted in "(4)" and "(5)". This aids us later in reducing the fault/failure occurrence through providing the best suited recovery mechanism, for instance for frequent or long-lasting failures. The observed behaviour will be analysed in terms of failure rates for each state – e.g., low response times may result from overload states or normal load states – in order to determine the number of failures observed for each state and to estimate the total failure numbers for all the states. We define $\Re$ as follows:

$$\Re(FN) = \frac{No\ of\ Detected\ Faults}{Total\ No\ of\ Faults\ of\ Resource} \quad (4)$$

$$\Re(FL) = \frac{Total\ Time\ of\ Observed\ Failures}{Total\ Time\ of\ Execution\ of\ Resource} \quad (5)$$

The Average Failure Length (AFL), as in "(6)", might also be relevant to judge the relative urgency of recovery. Other relevant metrics that impact on the decision which strategy to use, but which we do not detail here, are resilience metrics addressing recovery times.

$$AFL = \frac{\sum Time\ of\ Failure\ Occurrence}{Number\ of\ Observed\ Failures} \quad (6)$$

VII. 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 of 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 Real-Time/Historical storage to be processed by the Monitor. 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 minutes. The number of records that we obtained from the TPC-W was 2000 records.

[1]http://www.tpc.org/tpcw/
[2]https://xenproject.org/

 ISBN: 978-1-61208-706-1

We further 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 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 gathered dataset is classified into training and testing datasets 50% for each. The model training last 150 minutes.

To simulate real anomaly scenarios, script is written to inject different types of anomalies. The anomaly injection for each component last 5 minutes. The anomaly scenarios are: (1) CPU Hog, consume all CPU cycles by employing infinite loops. (2) Memory Leak, exhausts the component memory. The stress[6] tool is used to create pressure on CPU and Memory.

Further, workload contention is generated to test the controller under different workloads. To generate workload, 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 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[7] is installed on each node. Also AWS X-Ray[8] is used to trace of the request through the system.

B. The Detection Assessment

The detection model is evaluated by Root Mean Square Error (RMSE), Mean Absolute Percentage Error (MAPE), and False Alarm Rate (FAR), which are the commonly used metrics [25] for evaluating the quality of detection. We further measured the Number of Correctly Detected Anomaly (CDA) and Accuracy of Detection (AD).

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

b) Mean Absolute Percentage Error (MAPE): It measures the detection accuracy of a model. Both RMSE and MAPE are negatively-oriented scores, which means lower values are better.

[3]https://www.signalfx.com/
[4]https://github.com/kubernetes-retired/heapster
[5]https://www.influxdata.com/
[6]https://linux.die.net/man/1/stress
[7]https://www.vanheusden.com/httping/
[8]https://aws.amazon.com/xray/

TABLE I. DETECTION EVALUATION.

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%
AC	0.94	0.84	0.91
FAR	0.27	0.46	0.31

c) 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.

d) Accuracy of Detection (AD): It measures the completeness of the correctly detected anomalies to the total number of anomalies in a given dataset. Higher AD means that fewer anomaly cases are undetected.

e) False Alarm Rate (FAR): The number of the normal detected component, which has been misclassified as anomalous by the model.

The efficiency of the model is compared with a Dynamic Bayesian network (DBN), see Table I. The results show that the HHMM and HTM model detects anomalous behaviour with promised results comparing to DBN.

C. The Identification Assessment

The accuracy of the results is compared with Dynamic Bayesian Network (DBN), and Hierarchical Temporal Memory (HTM), and it is evaluated based on different metrics such as: Accuracy of Identification (AI), Number of Correctly Identified Anomaly (CIA), Number of Incorrectly Identified Anomaly (IIA), and FAR.

a) Accuracy of Identification (AI): It measures the completeness of the correctly identified anomalies to the total number of anomalies in a given dataset. Higher AI means that fewer anomaly cases are un-identified.

b) Number of Correctly Identified Anomaly (CIA): It 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.

$$CIA = \frac{NCIA}{NCIA + NICI} \tag{7}$$

c) Number of Incorrectly Identified Anomaly (IIA): IIA is the number of the identified component, which represents an anomaly but misidentified as normal by the model. The lower value indicates that the model correctly identified anomaly component.

$$IIA = \frac{FN}{FN + TP} \tag{8}$$

TABLE II. ASSESSMENT OF IDENTIFICATION.

Metrics	HHMM	DBN	HTM
AI	0.94	0.84	0.94
CIA	94.73%	87.67%	93.94%
IIA	4.56%	12.33%	6.07%
FAR	0.12	0.26	0.17

TABLE III. RECOVERY EVALUATION.

Evaluation Metrics	*Results*
RA	99%
MTTR	60 seconds
OA	97%

d) False Alarm Rate (FAR): The number of the normal identified component, which has been misclassified as anomalous by the model.

$$FAR = \frac{FP}{TN + FP} \tag{9}$$

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.

As shown in Table II, HHMM and HTM achieved promising results for the identification of anomaly. While the results of the DBN a little bit decayed for the CIA with approximately 7% than HHMM, and 6% than HTM. Both HHMM and HTM showed higher identification accuracy as they are able to identify temporal anomalies in the dataset. The result interferes that the HHMM is able to link the observed failure to its hidden workload.

D. The Recovery Assessment

To assess the recovery decisions of the model, we measure: (1) the Recovery Accuracy (RA) to be the number of successfully recovered anomalies to the total number of identified anomalies, (2) Mean Time to Recovery (MTTR), the average time that the approach takes to recover starting from the anomaly injection until recovering it. (3) Over all Accuracy (OA) to be the number of correct recovered anomalies to the total number of anomalies. The results in Table III show that once HMM model is configured properly, it can efficiently recover the anomalies with an accuracy of 99%.

VIII. CONCLUSIONS

This paper presented a controller architecture for the detection, and recovery of anomalies in hierarchically organised clustered computing environments, that reflects recent container cluster orchestration tools like Kubernetes or Docker Swarm. The key objective was to provide an analysis feature, that maps observable quality concerns onto hidden resources in a hierarchical clustered environment, and their operation in order to identify the reason for performance degradations and other anomalies. From this, a recovery strategy that removes the workload anomaly, thus removing the observed performance failure is the second step.

We have proposed to use Hidden Markov Models (HMMs) to reflect the hierarchical nature of the unobservable resources, and to support the detection, identification, and recovery of anomalous behaviours. We have further analysed mappings between observations and resource usage based on a clustered container scenario.

The objective of this paper was to introduce the complete controller architecture with its key processing steps. In the future, we will complete the current controller prototype, carry out further experimental evaluations, and also address practical concerns such as the relevance for microservice architectures [10].

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," Doctoral 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," 12th International ACM SIGSOFT Conference 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. 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.

[10] D. Taibi, V. Lenarduzzi, C. Pahl, "Architecture Patterns for a Microservice Architectural Style," Springer, 2019.

[11] H. Arabnejad, C. Pahl, G. Estrada, A. Samir, F. Fowley, "A fuzzy load balancer for adaptive fault tolerance management in cloud platforms," European Conference on Service-Oriented and Cloud Computing, 109-124, 2017.

 ISBN: 978-1-61208-706-1

[12] H. Arabnejad, C. Pahl, P. Jamshidi, G. Estrada, "A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling," 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2017.

[13] T. F. Düllmann, "Performance anomaly detection in microservice architectures under continuous change," Master, University of Stuttgart, 2016.

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

[15] N. Sorkunlu, V. Chandola, and A. Patra, "Tracking system behavior from resource usage data," in *International Conference on Cluster Computing, ICCC*, 2017.

[16] S. Maurya and K. Ahmad, "Load Balancing in Distributed System using Genetic Algorithm," *International Journal of Engineering and Technology (IJET)*, vol. 5, no. 2, pp. 139–142, 2013.

[17] H. Sukhwani, "A survey of anomaly detection techniques and hidden markov model," *Intl Journal of Computer Applications*, vol. 93, no. 18, 2014.

[18] 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.

[19] 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.

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

[21] G. Brogi, "Real-time detection of advanced persistent threats using information flow tracking and hidden markov," Doctoral dissertation, 2018.

[22] D. von Leon, L. Miori, J. Sanin, N. El Ioini, S. Helmer, C. Pahl, "A performance exploration of architectural options for a middleware for decentralised lightweight edge cloud architectures," CLOSER, 2018.

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

[24] IEEE, "IEEE standard classification for software anomalies (IEEE 1044 - 2009)," pp. 1–4, 2009.

[25] K. Markham, "Simple guide to confusion matrix terminology," 2014.

 ISBN: 978-1-61208-706-1