Anomaly Detection and Analysis for Reliability Management in Clustered Container Architectures

Areeg Samir, Nabil El Ioini, Ilenia Fronza, Hamid R. Barzegar, Van Thanh Le and Claus Pahl

Faculty of Computer Science
Free University of Bozen-Bolzano
39100 Bolzano, Italy
Email: firstname.surname@unibz.it

***Abstract*—Virtualised environments such as cloud and edge computing architectures allow software to be deployed and managed through third-party provided services. Here virtualised resources available can be adjusted, even dynamically to changing needs. However, the problem is often the boundary between the service provider and the service consumer. Often there is no direct access to execution parameters at resource level on the provider's side. Generally, only some quality factors can be directly observed while others remain hidden from the consumer. We propose an architecture for autonomous anomaly analysis for clustered cloud or edge resources. The key contribution is that the architecture determines possible causes of consumer-observed anomalies in an underlying provider-controlled infrastructure. We use Hidden Hierarchical 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. We illustrate use cases in the context of container technologies to show the utility of the proposed architecture.**

***Index Terms*—Cloud Computing; Edge Computing; Container Technology; Cluster Architectures; Markov Model; Anomaly Detection; Performance; Workload.**

I. Introduction

As a consequence of the dynamic nature of cloud and edge computing environments, users 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, for example

- change in a cluster node workload demand or configuration updates may cause dynamic changes,
- reallocation or removal of resources may affect the workload of system components.

In principle, application deployments can be adjusted, even dynamically to changing conditions. A problem, however, emerges. Cloud and edge computing allow applications to be deployed in remote environments, but these are managed by third parties based on provided virtualised resources [1], [2], [3], [4] which often hides the underlying causes from the consumers of these services.

In virtualised environments, 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). Thus, the reasons for these anomalies remain unclear. Recent works on anomaly detection [5], [6], [7] have looked at resource usage, rejuvenation or analysing the correlation between resource consumption and abnormal behaviour of applications. However, more work is needed on identifying the reason behind observed resource performance degradations.

We here investigate the possible root causes of performance anomalies in an underlying provider-controlled cloud infrastructure. We propose an anomaly detection and analysis architecture for clustered cloud and edge environments that aims at automatically detecting possibly workload-induced performance fluctuations, thus improving the reliability of these architectures. 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) [10] 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.

We focus on a clustered, hierarchically organised environment with containers as loads on the individual nodes, similar to container cluster solutions like Docker Swarm or Kubernetes [36]. We use a detailed use case discussion in container technologies to illustrate the applicability of the proposed solution.

In order to broaden the discussion, we also expand our anomaly notion. In addition to performance and workload

anomalies, we introduce trust anomalies and discuss the transferability of concepts to this trust concern.

This paper is structured as follows. Section II discusses the state of the art. Section III introduces our wider anomaly management architecture. Section IV details the anomaly detection and fault analysis. Section V evaluates the proposed architecture. This is followed by an extended use case discussion in Section VI that shows the applicability of the results. Section VII discusses the transferability of concerns to a trust anomaly context. Section VIII ends the paper with some conclusions and possible future work.

II. Related Work

This section explores the detection, identification, and recovery of anomaly in literature. Moreover, it sheds light on the literatures that use the Hidden Markov Model to mitigate the anomalous behavior.

A. Anomaly Detection and Identification

Several studies [11] and [7] have addressed workload analysis in dynamic environments. Sorkunlu et al. [12] identify system performance anomalies through analysing the correlations in the resource usage data. Peiris et al. [13] analyse the root causes of performance anomalies by combining the correlation and comparative analysis techniques in distributed environments.

Dullmann et al. [14] provide an online performance anomaly detection approach that detects anomalies in performance data based on a discrete time series analysis. Wang et al. [7] model the correlation between workload and the resource utilization of applications to characterize the system status. However, the authors work neither classifies the different types of workloads, or recovers the anomalous behaviour.

In [26] the author detects the anomalous behaviours (CPU overload and Denial of Service Attack), and provides an adaptation policy using a multi-dimensional utility-based model and algorithms. The author gives a score and likelihood for the anomaly detected to select an adaptation policy to be able to scale compute resources. The author work specifies a node leader for each microservice cluster. Each node maintains the cluster state and preserves the cluster logs. The leader also votes on the adaptation policy action. However, the author work handles two types of anomalies, and it is limited to the horizontal and vertical auto-scaling actions to mitigate the anomalous behaviour. Further, the work does not predict the future workload.

The work in [46] detects the anomalous behaviour in performance using a forecasting model to estimate the bandwidth, detect performance changes and to decompose time series into components. However, the authors use a hard threshold in all the dataset which may not reflect the actual workloads in system. In addition, they only detect anomalies without analysing them, and they use labelled-time which is not good enough to detect all anomalies as some anomalies could not be discovered during the detection process and time complexity in terms of data size may occur.

In [38] the authors focus on detecting anomalous behaviour of services deployed on VM in a cloud environment. Like our architecture, different anomaly injection scenarios are created and a workload is generated to test the impact of anomaly on the cloud services. The authors emulated different anomalies with the CPU, memory, disk, and network. However, their work does not track the cause of anomalous behaviour in a containerized cluster environment, and it neglects the dependency between nodes.

The work in [50] implements a probabilistic prediction model based on a supervised learning method. The model aims at detecting anomalous behaviour in cloud infrastructure through analysing correlation between different metrics (CPU, memory, disk, and network) to find the essential metrics that can characterize the correlation between cloud performance and anomaly event. The work uses a directed acyclic graph to analyse the correlation of various performance metrics with failure events in a virtual and physical machines. The author computes the conditional probability of every metric on anomaly occurrences, and selects those metrics whose conditional probabilities are greater than a predefined threshold. Nevertheless, the results show that the model suffers from poor prediction efficiency when it is used to predict cloud anomalies.

The work in [54] presents a general purpose prediction model to prevent anomalies in cloud environment. The author uses a supervised learning-based model that combines two dependent Markov chain models with the tree augmented Bayesian networks. The work applies statistical learning algorithms over system level metrics (CPU, memory, network I/O statistics) to predict the anomalous behaviour. However, the author does not provide information about the prediction efficiency.

The work in [61] predicts the impact of processor cache interference among consolidated workloads at application level. To predict the performance degradation of consolidated applications, the prediction technique is only linear to the number of cores sharing the last-level cache. However, the author limits its discussion to cache contention issues, ignoring other resource types.

The work in [47] develops a description language ”Performance Problem Diagnostics Description Model” to specify the information required for conducting an automatic performance problem diagnostics. The work analyses the workload to detect and categorize the faults into three layers namely. (1) Symptoms, externally visible indicators of a performance problem, (2) manifestation, internal performance indicators or evidences, and (3) root-causes, physical factors whose removal eliminates the manifestations and symptoms of a performance incident. However, the approach neither considers the dependency between faults nor avoids human interaction (i.e., performance experts should provide heuristics to be able to detect performance problems). The approach is designed to

apply for a specific domain, it does not provide a recovery mechanism to the detected faults neither discovers the dependency between anomalies. Further the approach is based on predefined heuristics (rules) to detect performance problems. Consequently, applying the approach on a different domain or changing the fault model requires heuristics update.

The work in [70] proposes an approach for localizing anomalies at operation time of a target system using the Kieker monitoring approach. For the localization of anomalies, the author calculates an anomaly score for an operation through specifying a threshold. The author specifies a set of rules to detect performance anomaly. The rules are continuously evaluated based on the anomaly score through using forecasting techniques to predict future values in a time series. The author evaluates the observed measurement values, (i.e. response times) with the forecasted values to detect anomalies. However, the work ignores the type of the performance anomaly and anomaly dependency.

The work in [39] localizes faulty resources in cloud environments through modelling correlations among anomalous resources. The author uses the graph theory to locate the correlation between pairs of resources. The author focuses on analysing the amount of occupied memory in a physical server, the CPU consumption of a virtual machine, and the number of connections accepted by an application. However, the author work does not target anomaly in microservice or container.

In [30] the author studies the performance of several machine learning models to predict attacks on the IoT systems accurately. The results show that the random forest model achieves a promising anomaly prediction comparing to the other machine learning models. Nevertheless, the work only concentrates on predicting the network anomaly.

In [34] the author proposes an approach to estimate the capacity of a microservice by measuring the maximal number of successfully processed user requests per second for a given service such that no Service Level Objective SLO is violated. The author conducts a limited set of load tests followed by fitting an appropriate regression model to the acquired performance data. The author work examines the impact of workload on the CPU and memory usage. The author mentions that changing the number of requests affects the number of virtual CPU cores but it does not affect the memory utilization significantly. However, the work does not predict the future workload. Also, the work neglects the dependency between the nodes and services.

The work in [42] investigates the network performance impact of containers deployed on virtual machines. The author does several experiments to analyse the network performance of containers considering the horizontal scaling and network data transfer rate. Nevertheless, the work concentrates only on network and its impact on container performance.

In [43] the work explores the affect of microservices on each other on the same host. The author measures the CPU, memory and network usage metrics of the containers and nodes. However, the work is limited to evaluate the current failure prediction methods in Microservice environment. Moreover, the work does not locate or detect anomalous behaviour, and it focuses is CPU-bound workload.

B. Hidden Markov Model

Many literatures use the HMM, and its derivations to detect anomaly. In [17], the author proposes various techniques implemented for the detection of anomalies and intrusions in the network using the HMM.

Ge et al. [19] detect faults in real-time embedded systems. The authors use the HMM to describe the healthy and faulty states of a systems hardware components. In [22] the HMM is used to find which anomaly is part of the same anomaly injection scenarios.

C. Anomaly Recovery

In [28] the author provides a fault tolerance management mechanism at the Physical Machines and Virtual Machines levels. the work uses Redundant Array of Independent Disks (RAID-6) to optimize the space storage and to recover data in case of machine failure. The author divides a set of VM and PM into sub-sets of the same size. The author uses two services to gather information about a resource status and to manage resources through adding and deleting resources to mitigate resource failure. Nevertheless, the author only focuses on two aspects of recovery: handle the storage disk crash, and deal with the operating system crash.

Maurya and Ahmad [16] propose an algorithm that dynamically estimates the load of each node and migrates the task on the basis of predefined constraint [31]. 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.

In [77] the author presents a control theory-based consolidation approach that mitigates the effects of the cache, memory and hardware contention of coexisting workloads. The approach manages interference among consolidated VMs by dynamically allocating the resources to applications based on the workload SLAs. But, the author focuses is CPU-bound workload and compute-intensive applications.

III. Self-Adaptive Fault Management

Our ultimate goal is a self-adaptive fault management architecture [9], [8], [23] 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 architecture 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 architecture. The focus in this paper is on the Fault management model.

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 [15] such as the Docker Swarm or Kubernetes, to which we aim to apply our architecture ultimately.

We align the architecture with the Monitor, Analysis, Plan, Execute based on the anomaly detection Knowledge (MAPE-K) feedback control loop. The Monitor, collects data regarding the performance of the system as the observable state of each resource [18]. This can later be used to compare the detected status with the currently observed one. Each anomalous state has a weight (probability of occurrence). The 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. The 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 the observed response time failures. Corresponding rules can be updated with the results from the recovery stage. This update aids in learning our models and enhancing the future detection.

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 the system resources (i.e., CPU) to address unreliable behaviour. We can also classify the resource workload into normal load (NL), overload (OL), and underload (UL) categories to capture the workload fluctuations.

B. Anomaly Detection and Identification

Anomaly detection, the Monitoring stage in the 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 the 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 the Hierarchical Hidden Markov models (HHMM) [10], a doubly stochastic model for modeling hierarchical structures of data, to identify the source of anomalies.

Based on the response time emissions, we trace the path of the observed states in each observation window. Once we diagnose anomalous behaviour, the affected nodes are annotated with a weight, which is a probability of fault occurrence for an observed performance anomaly. Nodes are addressed based on a first-detected-first-healed basis.

In order to illustrate the usefulness of this analysis, we 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 the MAPE-K, is applied to carry out the load balancing or the 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 the 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. Once a state has recovered, it is removed from an anomaly queue, stored it in the recovered list flagged as 'anomaly free', and the rules to enhance the future prediction of the model are updated. If the recovery process does not succeed, a new weight is assigned.

We specify the recovery mechanism using the following aspects: **Analysis**: relies on the current or 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 apply this two sample strategies below.

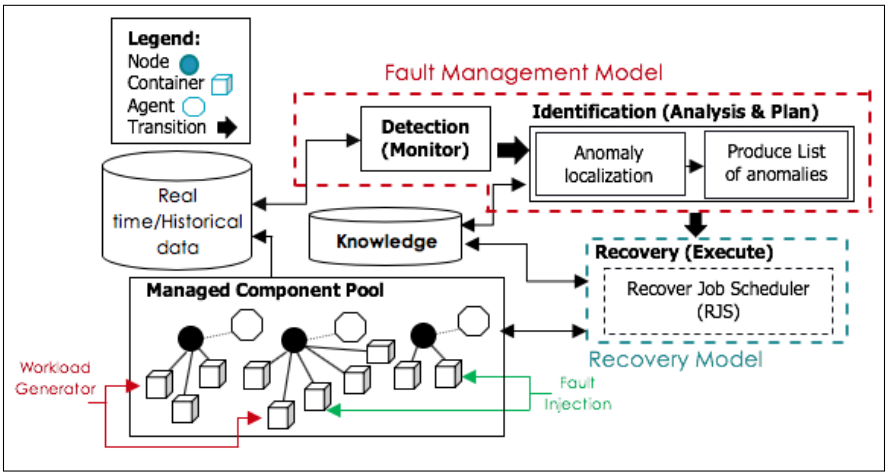

FIGURE 1. THE PROPOSED FAULT MANAGEMENT FRAMEWORK.

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): this happens when a container c_3 in node 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 a low response time observed from those containers.
Analysis: based on the current/historic observations, hidden states
Observation: low response time at the connected containers (overall failure to meet 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 communicate with 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 a 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 [24]. We assume that a failure is an undesired response time observed during a 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 resources capacity that is in use. It helps us in knowing the resource workload, and helps us in reducing the amount of jobs from the overloaded resources, e.g., a resource is saturated when its usage is 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 to $200ms$, which can be used to transfer the workload from the heavy loaded resources to the underloaded resources.

As a result, the recovery strategy differs based on the type of observed failure and the 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.

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 composed 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 identify different cases that may occur at container, node or cluster levels as illustrated in Figure 3. These detected cases 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.

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 goes 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 when there is a 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 a 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 slows 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

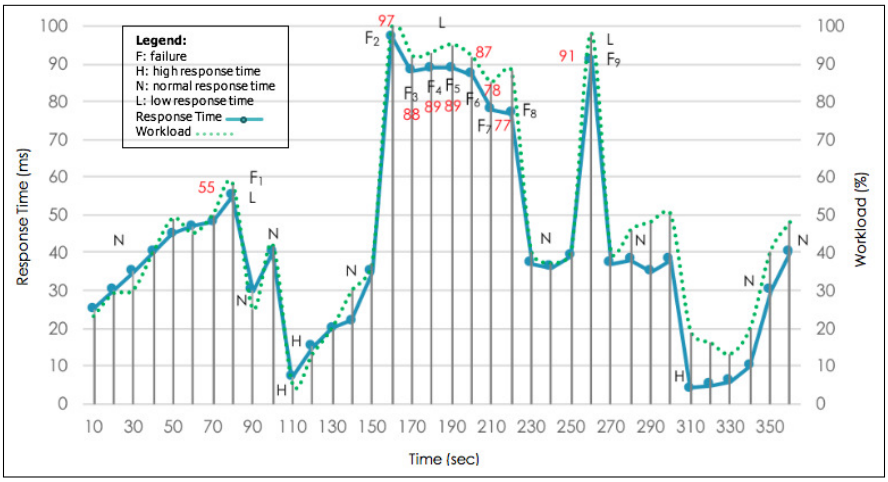

FIGURE 2. RESPONSE TIME AND WORKLOAD FLUCTUATIONS.

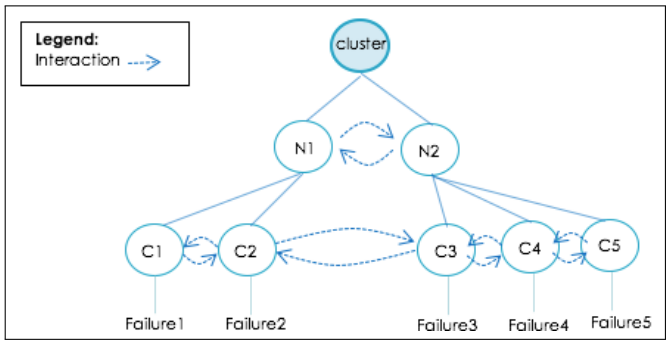

FIGURE 3. THE INTERACTION BETWEEN CLUSTER, NODES AND CONTAINER.

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 causes 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, the HHMMs are used to reflect the system topology.

C. Hierarchical Hidden Markov Model

The 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). The 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 the HHMM are hidden from the observer, and only the observation space is visible (response time). The states of the 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 is 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.

The HHMM is identified by $HHMM =< \lambda, \theta, \pi >$. The λ is a set of parameters consisting 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.

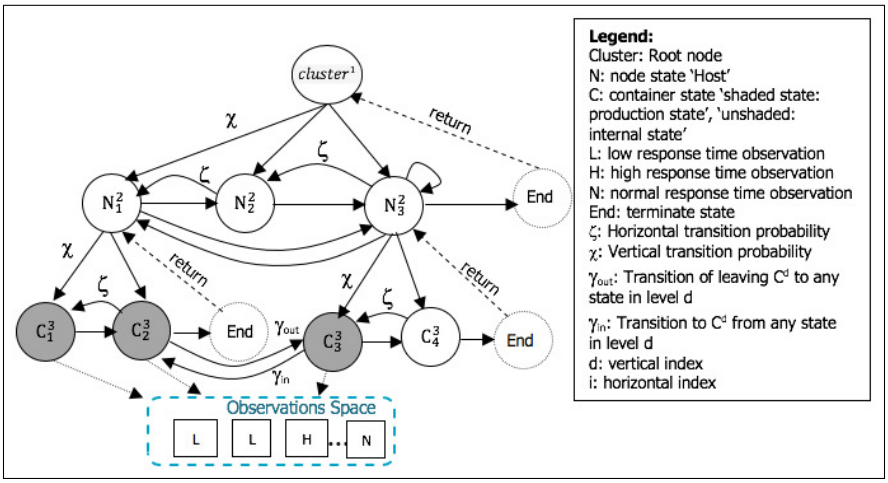

FIGURE 4. HHMM FOR WORKLOAD.

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 the CPU, and it is associated with the response time observations that are emitted from a 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 $(f_1, f_2, ..., f_{n-1})$ is generated before the activation of its parent state N. A state ends its operation at time t if the F_t is the last observation generated by any of the production states C reached from N, and the control returns 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 the normal load (NL) or underload (UL). This fluctuation in workload is associated with a probability that reflects the state transition status from the 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 the 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 use the generalized Baum-Welch algorithm [10] 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 $state_i^d$ are 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.

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

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

The output of the algorithm is used to train the 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 is 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 is 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.

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

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

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

Once we have trained the model, we compare the detected hierarchy against the observed one to detect and identify the type of workload. If the observed hierarchy and detected one is similar, and within the specified threshold, then the status of the observed component is declared as 'Anomaly Free', and the architecture returns to gather more data for further investigation. Otherwise, the hierarchy with the lowest probabilities is considered anomaly. Once we detect and identify 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 its activation, its activated substates, and the time at which the control returns to the calling state. This helps us in the recovery procedure as the anomalous state is recovered first come-first heal.

E. Workload and Resource Utilization Correlation

To check if the occurrence of an anomaly at cluster, node, container resource due to a workload, we calculate 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 the 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 add a unique workload identifier to the group of workloads in the same period to achieve traceability through the entire system. We utilize the probabilities of states transitions that we obtain from the HHMM to describe workload during T period. We transform the obtained probabilities to get a workload behaviour vector ω to characterize user transactions behaviours as in "(8)".

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

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

A. Environment Set-Up

To evaluate the effectiveness of the proposed architecture, the experiment environment consists of three VMs. Each VM is equipped with Linux OS, 3 VCPU, 2 GB 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 deploy two containers, and we run into them the TPC-W benchmark.

The TPC-W benchmark is used for resource provisioning, scalability, and capacity planning for e-commerce websites. The TPC-W emulates an online bookstore that consists of 3 tiers: client application, web server, and database. Each tier is installed on VM. We do not consider the database tier in the anomaly detection and identification, as a powerful VM should be dedicated to the database. The CPU and Memory utilization

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

[2] https://xenproject.org/

are gathered from the web server, while the Response time is measured from clients end. We run the TPC-W for 300 min. The number of records that we obtained from the TPC-W is 2000.

We use the docker $stats$ command to obtain a live data stream for running containers. The SignalFX Smart Agent[3] monitoring tool is used and configured to observe the runtime performance of components and their resources. We also use the Heapster[4] to group the collected data, and store them in a time series database using the 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 lasts 150 minutes.

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 the 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 behaviour, we consider the container is gradually loaded 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), the HTTPing[8] is installed on each node. Also, the 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 are different 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 add a new container C_5^3 in N_1^2, and we inject it by one anomaly at a time. For the CPU Hog, the anomaly is injected at 910 sec. It takes 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 happens at 1135 sec, and the model detects and identifies the anomaly at 1145 and 1163 sec respectively as shown in Table I.

TABLE I. CONTAINER OVERLOAD SELF-DEPENDENCY ANOMALY SCENARIO.

Container overload				
Anomaly				
Injection		Detection	Localization	Type
C^3_5	910	940	955	CPU hog
	1135	1145	1163	Network

Case 1.2. Container sibling overloaded (internal container dependency): in this case, the injection occurs at C_3^3 which in relation with C_4^3. The CPU injection begin at 700 sec for C_3^3, the model detects the anomalous behaviour at 710 sec and localizes it at 725 sec. For Network packet loss/latency, the injection of anomaly occurs at 905 sec. The model needs 46 sec for the detection and 19 sec for the identification. For the C_4^3 the detection happens 34 sec later the detection of C_3^3 for the CPU Hog and the anomaly is identified at 754 sec. For the Network, the detection and identification occur at 903 and 990 sec respectively as shown in Table II.

TABLE II. CONTAINER OVERLOAD INTERNAL-DEPENDENCY ANOMALY SCENARIO.

Container overload				
Anomaly				
Injection		Detection	Localization	Type
C^3_3	700	710	725	CPU hog
	905	951	970	Network
C^3_4		744	754	CPU hog
		903	990	Network

Case 1.3. Container neighbour overload (external container dependency): at this case, a CPU Hog is injected at C_1^3 which in relation with C_3^3. The injection begin at 210 sec. After training the HHMM, the model detects and localizes the anomalous behaviour for C_1^3 at 225 and 230 sec. For Network fault, the injection occurs at 415 sec for C_1^3. The model takes 429 sec for the detection and 450 sec for the identification. While for C_3^3, the CPU and Network faults are detected at 215/423 sec and identified at 240/429 sec as shown in Table III.

b) Low Response Time Observed at Node Level: Case 2.1. Node overload (self-dependency): at this case we create a new node N_4^2 with small application and we inject the node by one anomaly at a time. For the CPU Hog, the anomaly is injected at N_4^2. The injection begins at 413 sec. After training the HHMM, the model detects the anomalous behaviour at

[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://alexei-led.github.io/post/pumba_docker_netem/
[8]https://www.vanheusden.com/httping/
[9]https://aws.amazon.com/xray/

TABLE III. CONTAINER OVERLOAD EXTERNAL-DEPENDENCY ANOMALY SCENARIO.

Container overload				
Anomaly				
Injection		Detection	Localization	Type
C^3_1	210	225	230	CPU hog
	415	429	450	Network
C^3_3		215	240	CPU hog
		423	429	Network

443 sec and localizes it at 461 sec. For the Network packet loss/latency, the injection of anomaly happens at 1210 sec, and the model detects and identifies anomaly at 1260 and 1275 sec respectively as shown in Table IV.

TABLE IV. NODE OVERLOAD SELF-DEPENDENCY ANOMALY SCENARIO.

Node overload				
Anomaly				
Injection		Detection	Localization	Type
N^2_4	413	443	461	CPU hog
	1210	1260	1275	Network

Case 2.2. External node dependency: at such situation, a CPU Hog anomaly is injected at N^2_1. The injection begins at 813 sec. After training the HHMM, the model detects the anomalous behaviour at 846 sec and localizes it at 862 sec. For Network packet loss/latency, the injection of anomaly occurs at 1024 sec. The model needs 1084 sec for the detection and 1115 sec for the identification as shown in Table V.

TABLE V. NODE OVERLOAD EXTERNAL-DEPENDENCY ANOMALY SCENARIO.

Node overload				
Anomaly				
Injection		Detection	Localization	Type
N^2_1	813	846	862	CPU hog
	1024	1084	1115	Network

c) Low Response Time Observed at Cluster Level (Cluster Dependency): Case 3.1. Communication disconnection: at this case, we terminate the containers in N^2_3, and we send a request to the TPC-W server (N^2_3). The detection and identification for each network fault are 585 sec for the detection and 610 sec for the identification as shown in Table VI.

TABLE VI. CLUSTER OVERLOAD COMMUNICATION DISCONNECTION ANOMALY SCENARIO.

Cluster overload		
Anomaly		
	Detection	Localization
N^2_3	585	610

Case 3.2. Resource limitation: at this case, we inject N^2_1, and N^2_3 at the same time with the CPU Hog fault to exhaustive the nodes capacity. The injection, detection, and identification are 1120, 1181, and 1192 sec. For the Network fault, the injection happens at 1372 sec, and the detection, and identification are at 1387, and 1392 sec as shown in Table VII.

TABLE VII. CLUSTER OVERLOAD RESOURCE LIMITATION ANOMALY SCENARIO.

Cluster overload				
Anomaly				
Injection		Detection	Localization	Type
N	1120	1181	1192	CPU hog
	1372	1387	1392	Network

D. Detection and Identification of Workload Contention

For the workload, to show the influence of workload on CPU utilization monitored metric, we measure 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 generate gradual requests/sec at the container level. The number of user 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 reaches 220 request/sec. This means that when the number of user requests is reached 220 request/sec, the utilization of CPU reaches 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 the dynamic workloads have a noticeable impact on the container metrics as the monitored containers are unable to process more than those requests. We also notice that there is a linear relationship between the number of concurrent users and the CPU utilization before resource contention in each user transaction behaviour pattern. We calculate the correlation between the monitored metric, and the number of user requests. We obtain a strong correlation between the two measured variables reaches 0.25775 for two variables. The result concludes that the number of requests influences the performance of the monitored metrics.

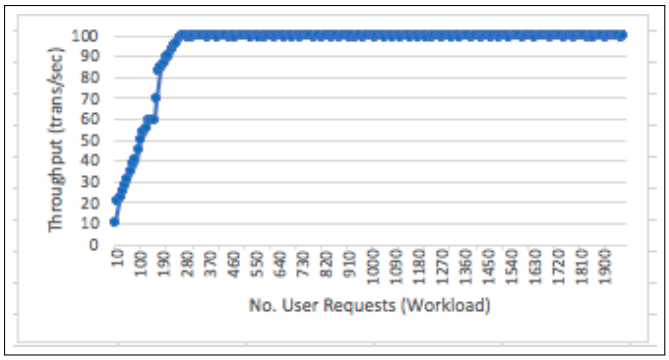

FIGURE 5. WORKLOAD - THROUGHPUT AND NUMBER OF USER REQUESTS.

E. Assessment of Detection and Identification

The model performance is compared with other techniques such as the Dynamic Bayesian Network (DBN) and the Hierarchical Temporal Memory (HTM). To evaluate the effectiveness

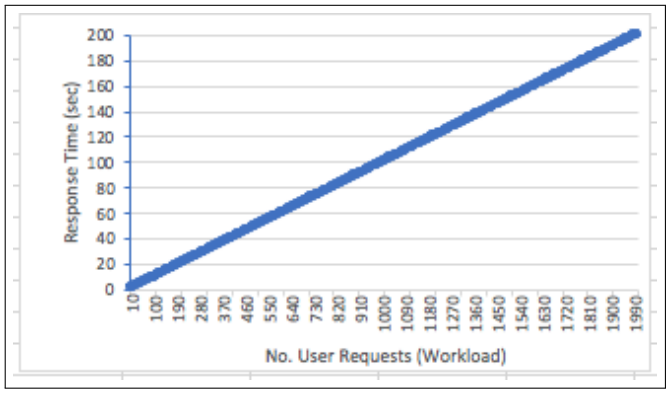

FIGURE 6. WORKLOAD - RESPONSE TIME AND NUMBER OF USER REQUESTS.

of anomaly detection, common measures [25] 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 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) is the number of correct identified anomalies (NCIA) out of the total set of identification, which is the number of correct identification (NCIA) + the number of incorrect identifications (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 is 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 VIII. 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 VIII depict that both the HHMM and HTM achieve good results for the detection and identification. While the results of the DBN a little bit decayed for the CDA with approximately 5% than the HHMM and 3% than the HTM. The three algorithms can detect obvious anomalies in the datasets. Both the HHMM and HTM show 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. USE CASE DISCUSSION

In order to illustrate the architecture, we discuss here two use cases. The first addresses a widely used cloud setting, where clusters of containers are managed by an orchestration solution such as the Kubernetes. The second looks at an edge cloud scenario, where a cluster of constrained hardware devices hosts container clusters.

A. Use Case: Cloud Container Management

Containers have grown in popularity in recent years and are now widely used as the unit of software deployment, also in cloud environments. Many cloud infrastructure (IaaS) and platform service (PaaS) providers offer container deployment options. In many cases, an orchestration tool like the Kubernetes[10], see Figure 7, that supports automated deployment, scaling and management of containerized applications are used by the providers, see Figures 8 and 7. These are typically homogeneous cloud container cluster in terms of the underlying infrastructure.

A problem that becomes apparent here is that a service consumer have access to monitoring data at the service level, but not necessarily at the underlying (physical) infrastructure level [40]. Nonetheless, service consumer are often given access to controllers that can for instance auto-scale the application deployed.

In this case, the user can be provided with a trained HMM that reflects possible faults for the observed failures.

B. Use Case: Edge Cloud Orchestration

Containers as a more lightweight form of virtualisation compared to virtual machines (VMs) consume less resources. They compare favourably to VMs in terms of startup time to

[10]https://kubernetes.io/

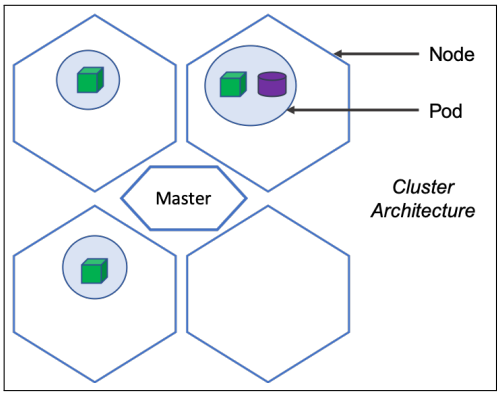

FIGURE 7. A CLUSTER ARCHITECTURE BASED ON KUBERNETES ARCHITECTURAL CONCEPTS.

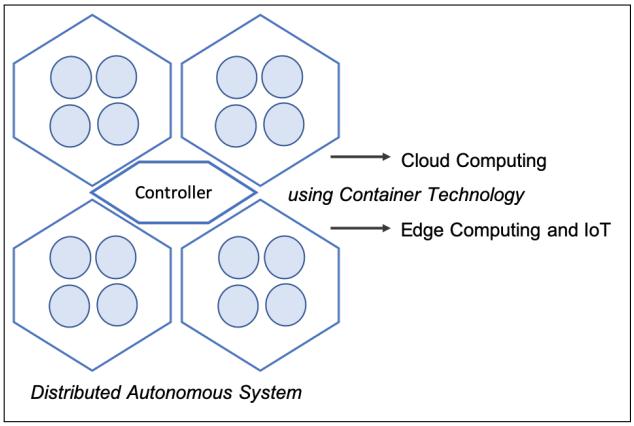

FIGURE 8. A DISTRIBUTED SYSTEM FOR CLOUD AND EDGE COMPUTING BASED ON CONTAINERS.

memory/storage needs. This makes containers more suitable to be utilised outside the classical centralised cloud environment. Here, edge cloud infrastructures that provide computational capabilities for IoT or other remote application can benefit from the containers' lightweightness. This is in particular useful if the edge infrastructure is limited in terms of its capabilities.

For the latter situation, we consider here a cluster on single-board devices as the physical infrastructure to host the

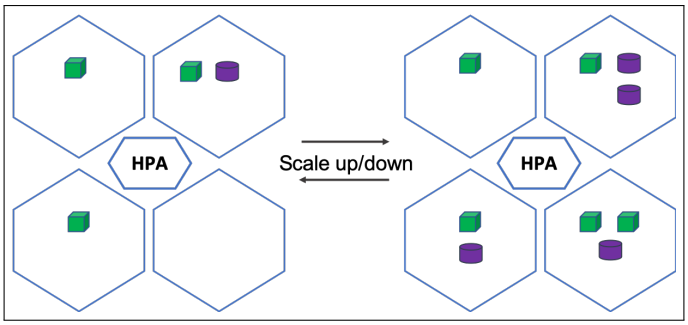

FIGURE 9. KUBERNETES AUTO-SCALING BASED ON THE HPA HORIZONTAL POD AUTOSCALER.

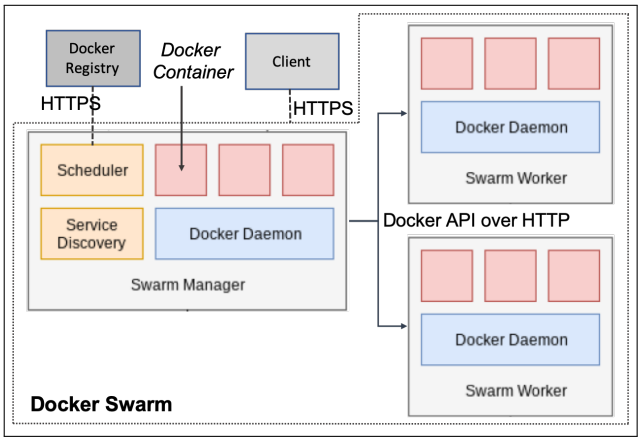

FIGURE 10. A DOCKER SWARM MANAGED ARCHITECTURE FOR CONTAINER ORCHESTRATION.

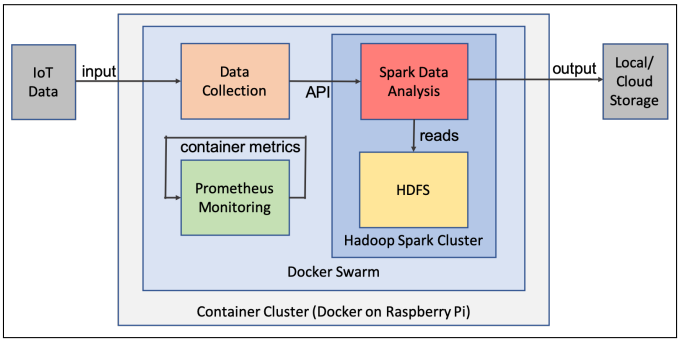

FIGURE 11. A DOCKER CONTAINER ARCHITECTURE FOR DATA STREAM PROCESSING WITH MONITORING SUPPORT.

container cluster platform. Specifically, we use Raspberry Pi[11] devices in this use case. In our experiments, we use the Docker Swarm[12] as the container orchestration tool, see Figure 10.

C. Use Case Scenario: Smart Farming

We categories the fault/failure cases, in which observable failures (to meet QoS requirements) are mapped to their root causes, i.e., the faults that have caused them. Examples are an overloaded container itself or a neighbouring container on which a container depends (e.g., is waiting for an answer) [23], [44]. We use the Markov models to reflect the possibility of several causes and the likelihood of each of these. Typical fault types are the CPU hog, network latency or workload contention.

For each of these mapping cases, we associate suitable remedial actions, such as workload distribution, container migration or resource rescaling.

These can be illustrated in a smart farming scenario. We assume here three central services: an animal stable in which air conditioning and feeding are automated, an outdoor irrigation system and support for tractor and machinery positioning in

[11]https://www.raspberrypi.org/
[12]https://docs.docker.com/engine/swarm/

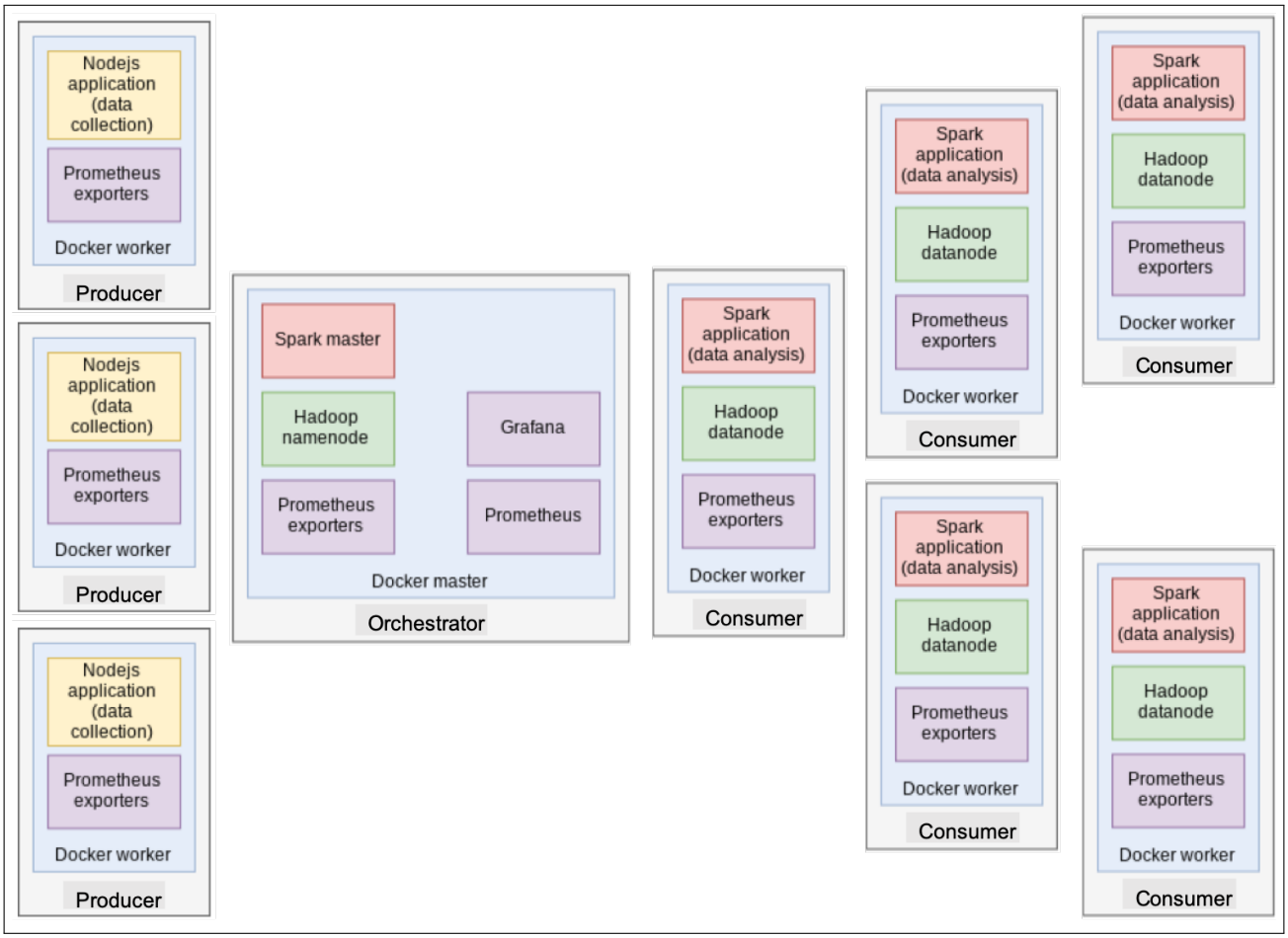

FIGURE 12. A DOCKER CONTAINER DISTRIBUTION FOR A RASPBERRY PI HOSTED CLUSTER.

remote fields. In particular, the outdoor services rely on low-power infrastructure to allow battery and/or solar panel driven energy supply. The indoor service requires reliability based on robust, but also redundant device infrastructures that can work in challenging conditions (e.g., dirt). The following problem situations are possible, and can be supported by our solution:

- Configuration and Testing: during installation or maintenance, increased demands on particular devices (e.g., CPU hogs) can emerge if data-rich test programs are run. Here, migrating containers (i.e., repurposing devices outside the actual service domain) can help.
- Increased Mobility: a higher number of vehicles on the fields might need to be coordinated, for instance during harvest time. This increase the latency problems in the network for both the coordination between vehicles and also recording of data on a central server. Here, moving containers for data preprocessing close to the vehicles can help to reduce the data volume on the network.

Our work in [20], [45] demonstrates how a container cluster solution implemented on Raspberry Pis can support this type of scenario. There, the Docker Swarm based management supports containers for data stream processing (the Apache Park), supported by the Prometheus as a monitoring tool, the Grafana for analyse/visualised data and databases like the InfluxDB to store data, see Figures 11 and 12.

The HMM identifies different anomaly states [1]. These are dependent on the monitored performance and workload/utilisation metrics. In other works [33], [35], we have used fuzzy logic to map monitored data to so-called membership functions that represent these different states. We refer the reader to these works for more detail. Here we focus on the anomaly processing.

VII. DISCUSSION – TRUST ANOMALIES

Anomaly detection and analysis techniques normally address performance and resource management in the context of software systems management. Another quality concern that is different from performance and resource consumption is the context of security and trust. Any open software system has a range of security vulnerabilities. Thus, checking continuously for anomalies in order to find unusual behaviour that might indicate attacks or the loss of information in some form is consequently also a relevant anomaly detection concern.

The concept of trust is here a related aspect that covers security but also the trust into the measurement and handling

of performance and other technical factors. A trust problem occurs if providers and consumers of services meet in an environment where no prior trust relationship exists. A trust anomaly here is a situation in which the delivery of a previously guaranteed service (or the promise of its delivered quality) is in doubt. An anomaly detection solution as the one presented here can help to proactively invoke a remedial action or to record more detailed information (in a tamper-proof way to avoid trust issues to arise). This would allow the analysis and resolution of disputes at a later stage.

The management of trust regarding the Quality-of-Service (QoS) compliance using a trust anomaly management solution shall now be discussed. If a-priori trust does not exist, it is crucial to capture, collect and store necessary information in a tamper-proof way that neither party can interfere with. Distributed ledger technology in the form of blockchains as mechanism to manage anomalies is a possible solution. A blockchain is a distributed data store for digital transactions, ressembling a ledger [55], [48], [48]. The blockchains are used for various applications [59], [63], [58], [56], [57]. These blocks are connected and secured using cryptographic mechanisms. Each of the blocks contains a cryptographic hash of the previous block, and also a timestamp and transaction data. Thus, a blockchain is inherently tamper-proof by design, which means it is resistant to modification of stored data.

This blockchain idea applies in case a consumer requires trustworthy documentation for instance in failure cases, but these blockchains maybe also always be used if a provider need assurance about having provided as planned or promised in a contract. More concretely, an anomaly detection mechanism as we introduced above can now, if the QoS compliance is for example under threat, switch on blockchain storage [55], [41], [37]. This could be as remedial supportive action for later analysis that can provide the required tamper-proof information for the recovery or dispute solving. This solution remains a part of the future work on our anomaly management architecture. However, this short discussion shows that the architecture presented is not limited to performance concerns and immediate remedial actions only, but that other quality concerns can be considered and long-term disputes over the origin and responsibilities can be solved.

VIII. Conclusion and Future Work

Cloud environments cause separation between providers and consumers. The virtualisation in these contexts does anyway separate the physical view from the logical perspective. Furthermore, only providers have access to the infrastructure, which means that consumers cannot always accurately interpret observed anomalies in application and service behaviour. We have introduced a architecture for the detection and identification of anomalies. The key objective is 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.

As the formal model, so-called the Hidden Hierarchical Markov Models (HHMM) are used to represent 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 the proposed architecture, the HHMM is compared with other machine learning algorithms such as the Dynamic Bayesian Network (DBN), and the Hierarchical Temporal Memory (HTM). The results show that the proposed architecture is able to detect and identify anomalous behaviour with more than 96%, which demonstrates the suitability of the solution.

We have been focusing specifically on clustered cloud environments as the architectural setting [52], [49], [69], [68], ultimately aiming at self-adaptation in the recovery process [66], [51]. In addition, we have selected the now widely used container technology as the deployment solution [40], [78]. The use cases that we have discussed here reflect this setting and show the suitability of the proposed architecture in this context.

As part of our future work, we are planning to fully implement the architecture. Also, carrying out further experiments is expected to fully confirm these given conclusions here is a wider range of application settings. Furthermore, another aim is to provide a self-healing mechanism to recover the localized anomalies detected.

On a more practical side, we want to follow the focus on containers further and aim to explore concerns from microservice architectures [21], [65], [67] and containers [32], [20] as their deployment technology in future investigations.

Anomalies are generally considered to be situations that impact on clearly specified system requirements, like performance. These might in turn impact on the user to fulfill her/his objectives with the system in question. An interesting possible investigation in the future could approach this more clearly from the user perspective. Providing a semantic context of activities would here be a first step [62], [53]. As a concrete application area where the user objectives are complex is educational technology systems [71], [72], [73], [74], where learning activities as cognitive processes need to be facilitated [75], [76]. This shall be looked at as well.

References

[1] A. Samir and C. Pahl, “Anomaly Detection and Analysis for Clustered Cloud Computing Reliability,” in *The Tenth International Conference on Cloud Computing, GRIDs, and Virtualization*, 110–119. 2019.

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

[3] C. Pahl, I. Fronza, N. El Ioini, and H. Barzegar, “A Review of Architectural Principles and Patterns for Distributed Mobile Information Systems,” in *14th Intl Conf on Web Information Systems and Technologies*. 2019.

[4] D. von Leon, L. Miori, J. Sanin, N. El Ioini, S. Helmer, and C. Pahl, “A Lightweight Container Middleware for

Edge Cloud Architectures," in *Fog and Edge Computing: Principles and Paradigms*, 145–170. 2019.

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

[6] 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.

[7] T. Wang, J. Xu, W. Zhang, Z. Gu, and H. Zhong, "Self-adaptive cloud monitoring with online anomaly detection," in *Fut Gen Computer Systems*, 80:89-101. 2018.

[8] P. Jamshidi, A. Sharifloo, C. Pahl, H. Arabnejad, A. Metzger, and G. Estrada, "Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures," in *12th International ACM SIGSOFT Conference on Quality of Software Architectures, QoSA*, 70–79. 2016.

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

[10] S. Fine, Y. Singer, and N. Tishby, "The hierarchical hidden markov model: analysis and applications," in *Machine Learning*, vol. 32, no. 1, 41–62. 1998.

[11] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload Prediction Using ARIMA Model and Its Impact on Cloud Applications," in *IEEE Transactions on Cloud Computing*, 3(4):449–458. 2015.

[12] N. Sorkunlu, V. Chandola, and A. Patra, "Tracking system behaviour from resource usage data," in *Intl Conference on Cluster Computing*, 410–418. 2017.

[13] M. Peiris, J. H. Hill, J. Thelin, S. Bykov, G. Kliot, and C. Konig, "PAD: Performance anomaly detection in multi-server distributed systems," in *International Conf on Cloud Computing, CLOUD*, June, 769–776. 2014.

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

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

[16] S. Maurya and K. Ahmad, "Load Balancing in Distributed System using Genetic Algorithm," in *Intl Journal of Engineering and Technology*, 5(2):139–142. 2013.

[17] H. Sukhwani, "A survey of anomaly detection techniques and hidden markov model," in *International Journal of Computer Applications*, vol. 93, no. 18, 26–31. 2014.

[18] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger, "Performance engineering for microservices: research challenges and directions," in *ACM/SPEC International Conference on Performance Engineering Companion*, 223–226. 2017.

[19] N. Ge, S. Nakajima, and M. Pantel, "Online diagnosis of accidental faults for real-time embedded systems using a hidden Markov model," in *Simulation* 91(19):851-868. 2016.

[20] R. Scolati, I. Fronza, N. El Ioini, A. Samir, and C. Pahl, "A Containerized Big Data Streaming Architecture for Edge Cloud Computing on Clustered Single-Board Devices," in *10th International Conference on Cloud Computing and Services Science*, 68-80. 2019.

[21] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural Patterns for Microservices: A Systematic Mapping Study," in *Proceedings CLOSER Conference*, 221–232. 2018.

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

[23] A. Samir and C. Pahl, "A Controller Architecture for Anomaly Detection, Root Cause Analysis and Self-Adaptation for Cluster Architectures," in *The Eleventh International Conference on Adaptive and Self-Adaptive Systems and Applications*, 75–83. 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.

[26] B. Magableh and M. Almiani, "A Self Healing Microservices Architecture: A Case Study in Docker Swarm Cluster," in *Intl Conference on Advanced Information Networking and Applications*, 846–858. 2019.

[27] C. Pahl, "An ontology for software component matching," in *International Conference on Fundamental Approaches to Software Engineering*, 6–21. 2003.

[28] A. Khiat, "Cloud-RAIR: A Cloud Redundant Array of Independent Resources," in *Intl Conference on Cloud Computing, Grids, and Virtualization*, 133–137. 2019.

[29] C. Pahl, N. El Ioini, and S. Helmer, "A Decision Framework for Blockchain Platforms for IoT and Edge Computing," in *3rd International Conference on Internet of Things, Big Data and Security*, 105-113. 2018.

[30] M. Hasan, M. Milon Islam, I. Islam, and M. Hashem, "Attack and Anomaly Detection in IoT Sensors in IoT Sites Using Machine Learning Approaches," in *Internet of Things*, vol. 7, 1–14. 2019.

[31] P. Jamshidi, C. Pahl, and N. C. Mendonca, "Pattern-based multi-cloud architecture migration," in *Software: Practice and Experience*, 47 (9), 1159-1184. 2017.

[32] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead," in *IEEE Software*, 35 (3), 24-35. 2018.

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

[34] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modelling for Cloud Microservice Applications," in *ACM/SPEC International Conference on Performance Engineering*, 25–32. 2019.

[35] P. Jamshidi, C. Pahl, and N. C. Mendonca, "Managing uncertainty in autonomic cloud elasticity controllers," in *IEEE Cloud Computing*, 50–60. 2016.

[36] C. Pahl and B. Lee, "Containers and clusters for edge cloud architectures - A technology review," in *IEEE International Conference on Future Internet of Things and Cloud*, August, 379–386. 2015.

[37] C. Pahl., N. El Ioini, S. Helmer, and B. Lee, "An architecture pattern for trusted orchestration in IoT edge clouds," in *The Third International Conference on Fog and Mobile Edge Computing, FMEC*, April, 63–70. 2018.

[38] C. Sauvanaud, M. Kaâniche, K. Kanoun, K. Lazri, and G. Da Silva Silvestre, "Anomaly detection and diagnosis for cloud services: Practical experiments and lessons learned," in *Jrnl of Systems and Software* 139:84-106. 2018.

[39] L. Mariani, C. Monni, M. Pezze, O. Riganelli, and R. Xin, "Localizing Faults in Cloud Systems," in *IEEE 11th International Conference on Software Testing, Verification and Validation*, 262–273. 2018.

[40] F. Ghirardini, A. Samir, I. Fronza, and C. Pahl, "Performance Engineering for Cloud Cluster Architectures using Model-Driven Simulation," in *ESOCC Workshops - CloudWays'18*. 2019.

[41] C. Pahl, N. El Ioini, S. Helmer, and B. Lee, "A Semantic Pattern for Trusted Orchestration in IoT Edge Clouds," in *Internet Technology Letters*. 2019.

[42] N. Kratzke, "About Microservices, Containers and their Underestimated Impact on Network Performance," in *CoRR*, vol. abs/1710.0. 2017.

[43] T. Zwietasch, "Online Failure Prediction for Microservice Architectures," Master Thesis, U Stuttgart. 2017.

[44] A. Samir and C. Pahl, "Detecting and Predicting Anomalies for Edge Cluster Environments using Hidden Markov Models," in *IEEE International Conference on Fog and Mobile Edge Computing*, 21–28. 2019.

[45] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," in *IEEE Cloud Computing*, 4 (5), 22-32. 2017.

[46] O. Ibidunmoye, T. Metsch, and E. Elmroth, "Real-time detection of performance anomalies for cloud services," in *IEEE/ACM Intl Symposium on Quality of Service*. 2016.

[47] A. Wert, "Performance problem diagnostics by systematic experimentation," PhD, KIT. 2015.

[48] N. El Ioini, C. Pahl, and S. Helmer, "A decision framework for blockchain platforms for IoT and edge computing," in *Proceedings of the 3rd International Conference on Internet of Things, Big Data and Security*. 2018.

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

[50] Q. Guan, C. C. Chiu, and S. Fu, "CDA: A cloud dependability analysis framework for characterizing system dependability in cloud computing infrastructures," in *Pacific Rim Intl Symp on Dependable Computing,* 11–20. 2012.

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

[52] P. Jamshidi, C. Pahl, S. Chinenyeze, and X. Liu, "Cloud migration patterns: a multi-cloud service architecture perspective," in *Service-Oriented Computing ICSOC2014 Workshops*, 6-19. 2015.

[53] D. Fang, X. Liu, I. Romdhani, P. Jamshidi, and C. Pahl, "An agility-oriented and fuzziness-embedded semantic model for collaborative cloud service search, retrieval and recommendation," in *Future Generation Computer Systems* 56, 11-26. 2016.

[54] Y. Tan, H. Nguyen, Z. Shen, and X. Gu, "PREPARE : Predictive Performance Anomaly Prevention for Virtualized Cloud Systems," in *Intl Conference on Distributed Computing Systems*, 285–294. 2012.

[55] N. El Ioini and C. Pahl, "Trustworthy Orchestration of Container Based Edge Computing Using Permissioned Blockchain," in *Intl Conference on Internet of Things: Systems, Management and Security*, 147-154. 2018.

[56] C. A. Ardagna, R. Asal, E. Damiani, N. El Ioini, and C. Pahl, "Trustworthy IoT: An Evidence Collection Approach Based on Smart Contracts," in *2019 IEEE International Conference on Services Computing (SCC)*, 46–50. 2019.

[57] V. T. Le, C. Pahl, and N. El Ioini, "Blockchain Based Service Continuity in Mobile Edge Computing," in *6th International Conference on Internet of Things: Systems, Management and Security*, 2019.

[58] C. A. Ardagna, R. Asal, E. Damiani, T. Dimitrakos, N. El Ioini, and C. Pahl, "Certification-based cloud adaptation," in *IEEE Transactions on Services Computing*. 2018.

[59] G. D'Atri, V.T. Le, C. Pahl, and N. El Ioini, "Towards Trustworthy Financial Reports Using Blockchain," in *Proceedings Tenth International Conference on Cloud Computing, GRIDs, and Virtualization*. 2019.

[60] N. El Ioini and C. Pahl, "A Review of Distributed Ledger Technologies," in *On the Move to Meaningful Internet Systems. OTM 2018 Conferences*, 227-288. 2018.

[61] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, "Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines," in *ACM Symp on Cloud Computing*1–14. 2011.

[62] M. Javed, Y. M. Abgaz, and C. Pahl, "Ontology change management and identification of change patterns," in *Journal on Data Semantics* 2(2-3), 119-143. 2013.

[63] S. Helmer, M. Roggia, N. El Ioini, and C. Pahl, "EthernityDB - Integrating Database Functionality into a Blockchain," in *European Conference on Advances in Databases and Information Systems*, 37–44. 2019.

[64] S. Helmer, C. Pahl, J. Sanin, L. Miori, S. Brocanelli, F. Cardano, D. Gadler, D. Morandini, A. Piccoli, S. Salam,

A. M. Sharear, A. Ventura, P. Abrahamsson, and D. T. Oyetoyan, "Bringing the cloud to rural and remote areas via cloudlets," in *Proceedings of the 7th Annual Symposium on Computing for Development*, 14. 2016.

[65] D. Taibi, V. Lenarduzzi, C. Pahl, and A. Janes, "Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages," in *XP2017 Scientific Workshops*, 2017.

[66] N. C. Mendonca, P. Jamshidi, D. Garlan, and C. Pahl, "Developing Self-Adaptive Microservice Systems: Challenges and Directions," in *IEEE Software*. 2020.

[67] D. Taibi, V. Lenarduzzi, and C. Pahl, "Microservices Anti-Patterns: A Taxonomy," in *Microservices - Science and Engineering*, Springer. 2019.

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

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

[70] J. Ehlers, A. van Hoorn, J. Waller, and W. Hasselbring, "Self-adaptive software system monitoring for performance anomaly localization," in *IEEE International Conference on Autonomic Computing, ICAC*, 197–200, 2011.

[71] C. Pahl, "Layered ontological modelling for web service-oriented model-driven architecture," in *European Conference on Model Driven Architecture-Foundations and Applications*. 2005.

[72] S. Murray, J. Ryan, and C. Pahl, "A tool-mediated cognitive apprenticeship approach for a computer engineering course," in *Proceedings 3rd IEEE International Conference on Advanced Technologies*, 2-6. 2003.

[73] C. Pahl, R. Barrett, and C. Kenny, "Supporting active database learning and training through interactive multimedia," in *ACM SIGCSE Bulletin* 36 (3), 27-31. 2004.

[74] C. Kenny and C. Pahl, "Automated tutoring for a database skills training environment," in *ACM SIGCSE Symposium 2005*, 58-64. 2003.

[75] X. Lei, C. Pahl, and D. Donnellan, "An evaluation technique for content interaction in web-based teaching and learning environments," in *IEEE International Conference on Advanced Technologies*, 294-295. 2003.

[76] M. Melia and C. Pahl, "Constraint-based validation of adaptive e-learning courseware," in *IEEE Transactions on Learning Technologies* 2(1), 37-49. 2009.

[77] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for QoS-aware clouds," in *European Conference on Computer Systems*, 237–250. 2010.

[78] F. Ghirardini, A. Samir, I. Fronza, and C. Pahl, "Model-Driven Simulation for Performance Engineering of Kubernetes-style Cloud Cluster Architectures," in *ES-OCC 2018 Workshops, PhD Symposium, EU-Projects*, 2019.