Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

<Initial cap, lower case>[†]

Author One*[1] | Author Two[2,3] | Author Three[3] | <Author name><corresponding author*>[5] | Author Name[5,5]

[1]Org Division, Org name, State name, Country name
[2]Org Division, Org name, State name, Country name
[3]Org Division, Org name, State name, Country name
[4]<Org Division>, <Org name>, <State name>, <Country name>
[5]<Org Division>, <Org name>, <State name>, <Country name>

Correspondence
<corresponding author link*> <author name, address. Email: <authorone@email.com>

Present Address
<Present address>

<Abstract heading>

Services running in Microservices cluster could be scaled in/out based on the demand issued by end-users, orchestration algorithm, or load balancer running on the cluster leader. The behaviour of Microservices architecture is continuously changing overtime, which makes it a challenging task to use a statistical model to identify normal and abnormal behaviour of the running services. The performance of the cluster nodes could fluctuate around the demand to accommodate scalability, orchestration and load balancing demands. This requires a model that is able to detect anomalies in real-time and generate a high rate of accuracy in detecting any anomalies and a low rate of false alarms. At the same time, it requires dynamic policy configuration that can be used to adapt the recent changes in the operational environment. This paper focuses on proposing a self-healing Microservices architecture, that is able to continuously monitor the operational environment, detects and observes anomalous behaviour, and provides a reasonable adaptation policy using multidimensional utility-based model. We argue in this paper that such model could offer self-scaling of Microservices architecture and adapt the sudden changes in the operational environment. The self-healing property is achieved by means of parameter tuning and dynamic adjustment of the cluster configuration. We believe integrating utility theory in the dynamic decision-making process improves the effectiveness of the adaptation and reduces the adaptation risk including resources over-provisioning and thrashing. Also, it preserving the cluster state by preventing multiple adaptation to take place at the same time.

KEYWORDS:
Self Healing, MicoroServices Architecture, Anomaly detection, Run- time configuration

1 | INTRODUCTION

A Microservices architecture could be defined in the context of a service-oriented architecture as a composition of tiny fine-grained distributed loosely coupled building blocks of software components[1]. Microservices improve software modularity and make the application easy to develop and maintain. However, With the rapid development of cloud infrastructures and virtualisation techniques, a high demand for building Microservices architectures in a complete virtualised environment has emerged.

[0]**<abbreviation head:>** <abbreviations> ..
[†]<title footnote.>

This need was met by introducing containers engine like Docker [1] as well as cluster management framework such as Docker swarm [2]. The performance of Microservices running in cluster mode could fluctuate around the demand to accommodate scalability, orchestration and load balancing offered by the cluster leader[1]. In their daily base tasks, many Dev-Ops face an issue of defining and identifying a threshold which can be used to identify a network, system or user activity as normal behaviour. Another challenge that exist in Microservices clusters is the ability to dynamically scale horizontally, (i.e. adjusting the number of nodes participating in the cluster), or vertically (i.e. adjusting the computational resources available for the services). In addition to this, it is not possible to configure auto scaling policy that could be used by the cluster leader to perform resilient and autonomous reasoning to achieve the desired QoS of the architecture.

Nowadays, cluster management technology does not embed a component that can guarantees continuous monitoring and adaptation of the operation environment and that can offer the architecture dynamic and self-adaptive capability to perform changes at run-time. To achieve such a high level of scalability, a swarm cluster, for instance, should have a component for continuously monitoring the cluster and a component for adaptation that can implement a reasonable reaction/scaling policy to accommodate the changes in the operating environment. This presents a challenge to build a self-healing microservices architecture that can dynamically adjust its own behaviour and heal itself against anomalous behaviour detected at real-time. Self-healing refers to a property of a self-adaptive software to have the capability of discovering, diagnosing and reacting to disruptions. It can also anticipate potential problems and, accordingly, take proper actions to prevent a failure[2]. To achieve a high level of self-healing it is necessary to have four major functionalities at run-time: 1) Monitoring and detecting events (contextual changes), 2) Context reasoning (dynamic decision making), 3) Adaptation strategies, 4) validation and verification of the adaptation action[3].

The proposed model in this paper offers Microservices architecture a self-healing property by providing a mechanism for continuous monitoring, context detecting of anomalous behaviour using real-time unsupervised anomaly detection algorithm, dynamic decision making using a multidimensional utility based model, enabling dynamic adaptation horizontally or vertically based on the demand and the changes of the operational environment, and runtime verification and validation of the fitness of proposed adaptation strategy. We argue that the self-healing attribute of Microservices architecture is achieved by leveraging parameter tuning and dynamic decision-making supported by an accurate anomaly detection of Microservices architecture.

This paper is structured as follows: Section 2 provides an overview of self-healing architectures and surveys the approaches for anomaly detection, and run-time configurations. Section 3 presents a model that can continuously observe Microservice architectures with Self-healing capabilities. Section 4 proposes a strategy for analysing and evaluating the capability of the model to detect anomalous behaviours and to trigger suitable adaptation actions. Section 5 is focused on results found followed a by a critical discussion of the effectiveness of this model. Section 6 summarises this research, highlighting its contribution and setting future work.

2 | RELATED WORK

Self-adaptive software is characterised by a number of properties best referred to as autonomic[4]. These the 'self-* properties' include Self-organisation, Self-healing, Self-optimisation and Self-protection[5]. Self-healing architecture refers to the capability of discovering, diagnosing and reacting to disruptions. It can also anticipate potential problems and, accordingly, take suitable actions to prevent a failure[5]. Self-healing aspects of Microservices architectures requires a decision-making strategy that can work in real-time. This is essential for Microservice to reason about its own state and its surrounding environment in a closed control loop model and act accordingly[6]. Typically, a self-healing system should implements the closed control loop stages including: Gathering of data related to the surrounding context (Context Sensing); Context observation and detection; Dynamic decision making; Adaptation execution to achieve the adaptation objectives defined as QoS; Verification and validation of the applied adaptation actions in terms of its ability to meet the adaptation objectives and meet the desired QoS.

This section focuses on discussing the related work of anomaly detection and dynamic decision making. However, there is many approaches are used for achieving high level of self-adaptability though Context sensing involving context collection, observation and detection of contextual changes in the operational environment[7]. Also, the ability of the system to dynamically adjust its behaviour can be achieved using parameter-tuning[8], component-based composition[9], or Middleware-based

[1]https://www.docker.com
[2]https://docs.docker.com/engine/swarm/

approaches[10]. Another important aspect of self-adaptive system is related to its ability to validate and verify the adaptation action at runtime based on Game theory[11], Utility theory as in[12,13], or Model driven approach as in[14].

Context information (1) refers to any information that is computationally accessible and upon which behavioural variations depend[15]. Context observation and detection approaches (2) are used to detect abnormal behaviour within the microservices architecture at run-time. Related work in context modelling, context detection and engineering self-adaptive software system are discussed in[2,6,16,7]. In dynamic decision making and context reasoning (3), the architecture should be able to monitor and detect normal/abnormal behaviour by continuously monitoring the contextual information found in the Microservices cluster.

There are two phases for detecting anomalies in a software system: a training phase which involves profiling the normal behaviour of the system; a second phase aimed at testing the learned profile of the system with new data and employing it to detect normal/abnormal behaviours[17].

Three major techniques for anomaly detection have emerged from the literature: statistical anomaly detection, data-mining and machine-learning based techniques.

Within the statistical methods, a system observes the activity of the system and generates profiles of system metrics to represent its behaviour. The system profile includes performance measures of the system resources such as CPU and Memory. For each measure, a separate profile is stored. Then, the current readings of the system are profiled and compared against the memorised past profile to calculate an anomaly score. This score is calculated by comparing all measures within the profile against a threshold specified by the developer. Once the system detects that the current readings of the system are higher than this threshold, then these will be automatically categorised as intrusions thus triggering an alert[18].

Various statistical anomaly detection systems have been proposed and they have some advantages[19,20]. One of this is that they can detect an anomaly without prior knowledge of the system. This can mitigate the common problem of a cold start found in machine learning techniques. Additionally, statistical anomaly detection provides accurate notifications of malicious attacks that occurred over long periods of times and it performs better in detecting denial-of-service attacks[17]. However, a disadvantage is that a skilled attacker might train a statistical anomaly detection system to accept the abnormal behaviour as normal. It is difficult to determine the thresholds that make a balance between the likelihood of a false negative (the system fails to identify an activity as an abnormal behaviour) and the likelihood of a false positive (false alarms). Statistical methods need an accurate model with a precise distribution of all measures. In practice, the behaviour of virtual machines/computers cannot be entirely be modelled using solely statistical methods.

With regard to data-mining approaches, data-mining is about finding insights which are statistically reliable, unknown previously, and actionable from data[21]. The dataset must be available, relevant, adequate, and clean. The data mining process involves discovering a novel, distinguished and useful data pattern in large datasets to extract hidden relationships and information about the data. In general, there are two issues involved in the use of data mining in an anomaly detection system. First, there is a lack of a large dataset to be used by the algorithm containing lots of information about the architecture. Second, few approaches were targeting the anomaly detection system in Microservices architecture[21]. Data mining based anomaly detection systems have three major difficulties which prevent them from being widely adopted in Microservices architecture[17]. Firstly, the low accuracy of detecting anomalous behaviour[22,17], as the data mining process would require large dataset with longer time interval to be able to improve the accuracy of detection. Most data mining techniques are heavily on computational resources, this negatively influences their adoption in a Microservices architecture[17]. Additionally, usually a data mining method used to classify an attack within a specific system cannot be successfully employed within another system for the same purpose. This because the process of training, testing the model and performing classification of anomalies needs to be repeated with different data or architecture[23].

Machine learning, in the context of anomaly detection, can allow the creation of software system able to learn and improve its detection accuracy over time[24]. Machine learning-based anomaly detection models aims to detect anomalies similar to statistical and data mining approaches. However, unlike them latter which tend to focus on understanding the process that generated the data, the former are data-driven and are mainly focus on training a model based exclusively on past data[17]. This means that, when additional and new data is provided they can intrinsically change their detection strategy and classify significant deviations from the normal behaviour of an underlying software program. An application of Machine Learning which enables the Microservices cluster to distinguish between normal and abnormal behaviour in the data can be found in[23]. In general, anomaly detection systems uses a combination of clustering and classification algorithms to detect anomalies. The clustering algorithm is used to cluster the dataset and label them. Then, a decision tree algorithm can be used to distinguish between normal and abnormal behaviour. Golmah[25] suggested the use of an effective classification model to identify normal and abnormal behaviour in network-based anomaly detection. The usage of Machine Mearning algorithm in this context can be found in[25,26,23]. Due to the opening deployment and limited resources found in a Microservices cluster, it is very important to use a lightweight approach of data clustering and classification. Due to this issue, this research focuses on proposing an anomaly detection mechanism that is more suitable for the Microservices architecture and can be easily deployed with less footprints on the limited resources found in the tiny containers running in Microservices cluster .

Numenta Platform for Intelligent Computing (NUPIC) is based on the Hierarchical Temporal Memory (HTM) model proposed in[27]. HTM has been experimentally applied in real-time anomaly detection of streaming data in[28,29]. The proposed system based on the HTM model claimed to be efficient and tolerant to noisy data. Most importantly it offers continuous monitoring of real-time data and adapts the changes of the data statistics. It also detects very subtle anomalies with a very minimum rate of false positives. In a recent study, Ahmad et al.[30] proposed an updated version of the anomaly detection algorithm with the introduction of the anomaly likelihood concept. The anomaly score calculated by the NUPIC anomaly detection algorithm represents an immediate calculation of the predictability of the current input stream. This approach works very well with predictable scenarios in many practical applications. As there is no noisy and unpredictable data found, the raw anomaly score gives an accurate prediction of false negatives. However, the changes in predictions would lead to revealing anomalies in the system's behaviour. Instead of using the raw anomaly score, Ahmad et al.[30] proposed a method for calculating the anomaly likelihood by modelling the distribution of anomaly scores and using the distribution to check the likelihood of the current state of the system to identify anomalous behaviour. The anomaly likelihood refers to a metric which defines how anomalous the current state is based on the prediction history calculated by the HTM model. So, the anomaly likelihood is calculated by maintaining a window of the last raw anomaly scores and then calculating the normal distribution over the last obtained/trained values, then the most recent average of anomalies is calculated using the Gaussian tail probability function (Q-function)[31].

In Microservices cluster, the performance of the cluster nodes could fluctuate around the demand to accommodate scalability, orchestration and load balancing issued by cluster leader. This requires a model that is able to detect anomalies in real-time and generate a high rate of accuracy in detecting any anomalies and a low rate of false alarms. In addition, there will be a set of variations that can be used by the system to adapt the changes in its operational environment. This requires a dynamic decision making that can calculate the utility of all possible adaptation actions based on the architecture constraints (i.e. number of replicas, number of nodes, desired objectives, metrics thresholds), anomaly score of the detected conditions (CPU, Memory, DISK I/O, Network I/O), and the confidence and accuracy of the anomaly score of the detected abnormal behaviour, and the desired/predicted cluster state. Then, the adaptation manager will execute the adaptation action and verifies its successfulness over the cluster architecture. Also, the adaptation manager will be able to self-tune and self-adjust the architecture parameters to meet high/low demand for services. Finally, the architecture will preserve the cluster state through the adaptation cycle (monitoring, observing, detecting, reacting, and verifying). This research focuses on finding a method to continuously observe and monitor the swarm cluster and be able to detect anomalous behaviour with a high accuracy and generate a low rate of false alarms. Then provide the architecture with adaptation strategies with high utility to reason about the detected anomalies and be able to self-adjust the architecture parameters and verifies the adaptation actions at runtime without human intervention.

3 | DESIGN AND METHODOLOGY

One important aspect of a self-healing Microservices architecture is the ability to continuously monitor the operational environment, detect and observe anomalous behaviour, and provide a reasonable policy for self-scaling, self-healing, and self-tuning the computational resources to adapt a sudden changes in its operational environment dynamically at rune-time.

To validate the ideas presented in this paper, we design and develop a working prototype of Microservice architecture in Docker swarm [3] as shown in Figure 1. The cluster consisted of many managers and many workers. To meet scalability and availability, the cluster manager distributed the work load between the workers based on Raft Consensus Algorithm[32]. This means that each service could be executed by assigning multiple containers across the cluster.

The main services implemented in this architecture are: Time series metrics database for context collection, Nodes metrics used to collect metrics from all nodes in the cluster , Alert and notification manager used to notify the adaptation manager about contextual changes offered by Prometheus framework [4]. Docker containers metrics collector for collecting fine-grained metrics

[3] https://docs.docker.com/engine/swarm/
[4] https://prometheus.io

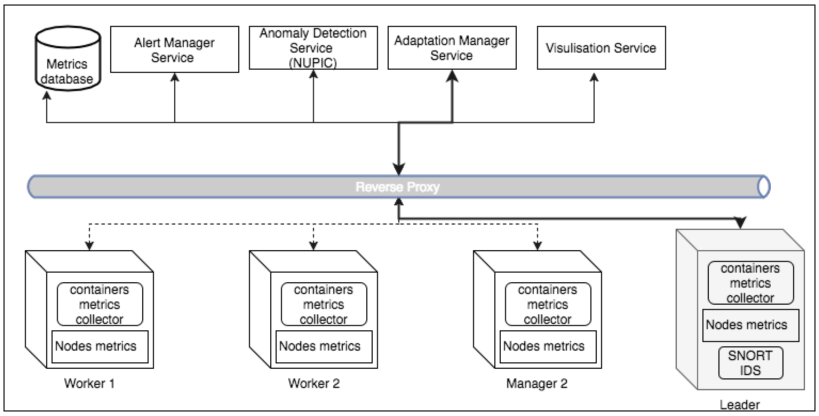

FIGURE 1 Microservices Architecture implemented in Docker Swarm

about all running containers in all nodes [5]. Reverse proxy for routing traffic between all services in the cluster [6]. Unsupervised Real-time Anomaly Detection based on NUPIC [7], Adaptation manager for executing, validating the adaptation actions developed as a prototype of this research. Time series analytic and visualisation dashboard for observing the behaviour of the Microservices cluster [8].

The prototype offering the Microservices architecture with the following functionalities:

Metric collection: Continuous collection of fine-grained metrics about cluster nodes, services, and containers including (CPU usage, Memory, Disk Reads Bytes/sec, Network Read/s, network write/s and Disk Writes Bytes/sec). The data are streamed into anomaly detection service at real-time.

Model Training: The NUPIC anomaly detection algorithm is continuously running over the streamed data collected in Step 1, which enables the generation of the training model for the collected metric.

Anomaly Detection: The collected real-time data is feed on the fly to NUPIC machine learning algorithm[30], which provides two features: First, continuous detection of anomalous behaviour with high accuracy. Second, it also provides predictions about the architecture performance based on the collected historic data. This can alerts the architecture about incoming spike on resources demand which can be used by the adaptation manager to schedule a proactive adaptation strategy ahead of time.

Adaptation Election: Once their is an anomalous behaviour detected with high anomaly score and likelihood. The Alert manager notifies the adaptation manager about the contextual changes. The adaptation manager selects the adaptation action(s) after calculating the utility value for all actions as explained below in equ. 1. Then, the adaptation manager uses the input of the anomaly likelihood, architecture constraints (specified by the DevOp during deployment), and desired/predicted QoS to calculate the best variation of the adaptation that has the highest utility as in equ. 2.

In different words, the Dev-Ops define the policy parameters that suits the Microservices architecture, but the adaptation manager at runtime adjusts those polices based on the demand and the runtime requirements. The adjustment of the adaptation policy is a major challenge as there might a finite set of configurations and parameters tuning over unlimited number of contextual changes. For this aim, the adaptation manager implements a multidimensional utility to elect the best adaptation strategy. From utility theory, a von Neumann-Morgenstern utility function $U_i : X_i \rightarrow \mathbb{R}$ assigns a real number to each quality dimension i, which we can normalize to the range [0, 1][33]. Across multiple dimensions of contextual changes, we can attribute a percentage weight to each dimension to account for its relative importance compared to other dimensions. These weights form the utility preferences. The overall utility is then given by the utility preference function calculated using equ. 1. For example, if three objectives, u1, u2, u3, are given decreasing importance as follows: the first is twice as important as the second, and the second is three times as important as the third. Then the utility fitness would be quantified as $[w1 : 0.6, w2 : 0.3, w3 : 0.1]$[8]. As soon as there is contextual changes detected in the architecture, the adaptation manager computes the utility of all variants related to the operational conditions and decides if an adaptation is required or not. This is achieved by checking if the current variant is still the one offering the highest utility or the predicted/desired variant of adaptation would offer higher utility. So, the utility function

[5]https://github.com/google/cadvisor
[6]https://caddyserver.com/docs/proxy
[7]http://nupic.docs.numenta.org/stable/index.html
[8]https://grafana.com

is calculated as in equ. 2 proposed in[13]. In this case, the adaptation strategy refers to any parameter-based or compositional-based configuration of the architecture, maintaining its original functional properties[13]. The adaptation manager uses a utility function to calculate and priorities the requirements that need to be meet in the adaptation action. Once an alert is triggered, the adaptation manager calculates the utility preference for all metrics collected from the cluster's operational environment.

$$fitness_i(V_j, C_m) = \sum_i^k W_i U_i \tag{1}$$

Konstantinos et al.[13] refer to the weight W_i in equ. 2 as the user preferences. In this paper we use the anomaly likelihood as the value of W_i to indicates the relative importance of the utility dimension compared to other dimensions. We argue that the use of anomaly likelihood to weight the collected metrics provides an accurate calculation of the utility dimension, as the anomaly likelihood is accurately defining how anomalous the current state comparing to the distribution of the trained values in the trained model. This enables the adaptation manager to scale the weight of each context change (utility dimension) over the distribution value calculated and aggregated in the anomaly likelihood value. The anomaly likelihood is a scaler value between 0 to 1, meaning if the w_{cpu} is 1 and the CPU usage value is 70% then this will give this metric high utility dimension so it will to be considered in the next adaptation action. In another scenario, if the anomaly likelihood is 0, this would give the metric low utility dimension so it will not be considered in the next adaptation action by the adaptation manager.

In addition to the above, it is important to have a method that can accurately calculate the cost of adaptation of the utility that have the highest dimension. For example, if the $Utility(_{cpu})$ has the highest dimension, this will triggers an adaptation action to reason about the high demand of CPU usage, so the adaptation manager needs to add additional nodes to join the cluster. Such process needs to be controlled to avoid resources overallocation.

For this aim, an accurate calculation is needed, that calculates the required number of nodes needed to meet the demand of the adaptation action and at the same time maintain the resource provisioning within the allocated budget by the Dev-Ops. The utility cost of provisioning a new node/container in the architecture is calculated based on the equ. 3, the $Current(u_i)$ is the current value of the utility dimension (context changes). The $Predicted(u_i)$ refers to the Predicted value of the utility dimension. The $AnomScore(u_i)$ is the Anomaly Score of the utility dimension at time t_i, and W_i is the anomaly likelihood value of utility dimension. The $Predicted(u_i)$, $AnomScore(u_i)$, and $W(u_i)$ are provided by anomaly detection service implemented based on NUPIC. The $UsageTime(u_i)$ refers to the total number of hours the node is expected to be used per/day, this value is calculated based on the rate of changes calculated based on equ. 4. The $Cost(instanceType)$ is the cost in \$ for provisioning an instance per/day, normally this is a constant value specified by the cloud infrastructure provider based on the instance type. Finally, the value of $Cost(u_i)$ is calculated against the constraint of $budget$ as $Cost(u_i) \leq budget$, The $budget$ is assigned by the Dev-Op to reflect the value of the available budget, so the adaptation manager will not exceed this value at any case. A negative value returned by $Cost(u_i)$ function means the number of nodes/replicas in the cluster should be reduced by the adaptation action.

$$Utility(V_j, C_m) \equiv \frac{\sum_{i=1}^k (W_i \cdot fitness_i(V_j, C_m))}{\sum_{i=1}^k W_i} \tag{2}$$

$$Cost(u_i) = \frac{(Current(u_i) - Predicted(u_i)) \cdot (AnomScore(u_i) \cdot W(u_i))}{UsageTime(u_i) * Cost(instanceType)} \tag{3}$$

$$\Delta U_i = \sum_i^k (U_i \cdot W_i) - (U_{i-1} \cdot W_{i-1}) \tag{4}$$

Adaptation Execution: The adaptation manager executes the strategies based on the aggregated value of the utility dimensions calculated in equ. 2. Once the adaptation action is completed by the adaptation manager, a set of adaptation actions are deployed in the architecture. To avoid, conflicts between multiple adaptation variations, the adapter allow the adaptation actions to be fully completed and verified by the cluster leader, then it will put a cool off timer before initiating new adaptation actions. This technique is used to avoid resources thrashing and preserving the cluster state for auto-recovery. The adaptation manager sends the cluster leader a set of instructions that might involve tuning of cluster parameters, (horizontal scaling), adding/removing nodes, or vertical scaling of microservice's containers like scaling a service in/out.

Adaptation Verification: The cluster leader and all managers in the cluster will vote on the adaptation action based on the consensus algorithm[32]. The vote results is used to validate and verifies the adaptation action. If the adaptation action won the votes, the adaptation action will be executed by the cluster leader, the adaptation manager records the adaptation attempt as successful. If the adaptation action lost the voting process, then the adaptation manager keeps the current state of the cluster and

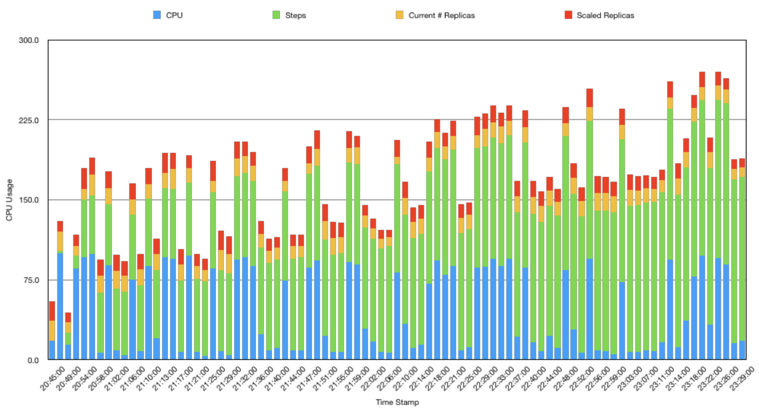

FIGURE 2 Dynamic Scaling of Web Service

records the adaptation attempt as failed. In both cases, the adaptation manger records the number of attempts used to complete the adaptation actions.

The evaluation of this model will come in two folds: First, evaluating the consistent behaviour of the cluster by evaluating the state of the swarm after executing a set of adaptation actions. Second, evaluating the accuracy of the anomaly detection algorithms using confusion matrix[34], as described in the following section.

4 | RESULTS

The Microservices cluster implemented with many services including the adaptation manager, which contains the NUPIC machine learning and the adaptation executer. This live snapshot [9] provides a full virtualisation of all services running in the cluster. The evaluation of the effectiveness of this model will be based on calculating a utility function for all metrics monitored, then it will calculates the number of adaptation attempts, successful convergence of services/nodes, or errors which leads to unstable state of the cluster.

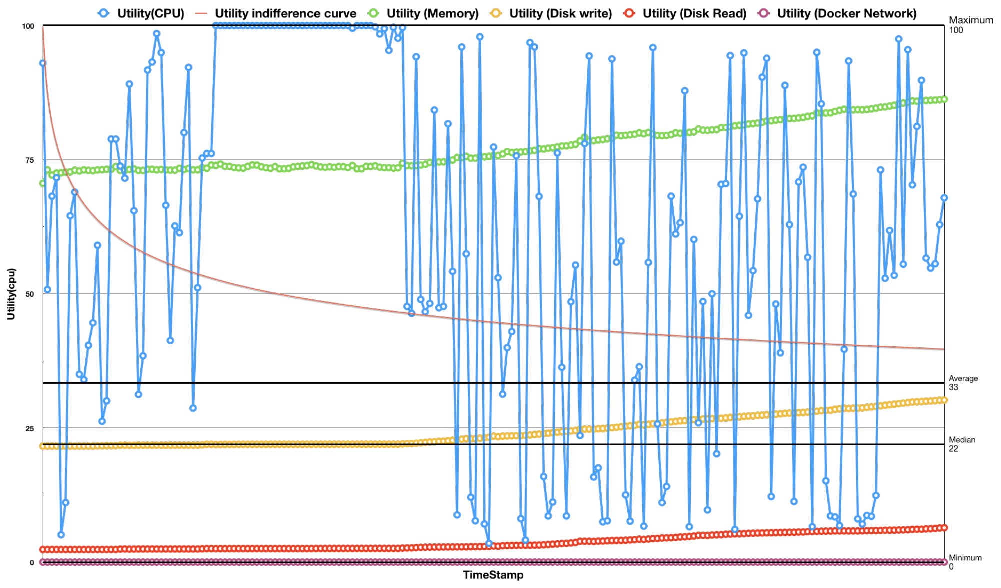

FIGURE 3 Dimensional analysis of variations of utility functions

[9]https://goo.gl/LJPrFy

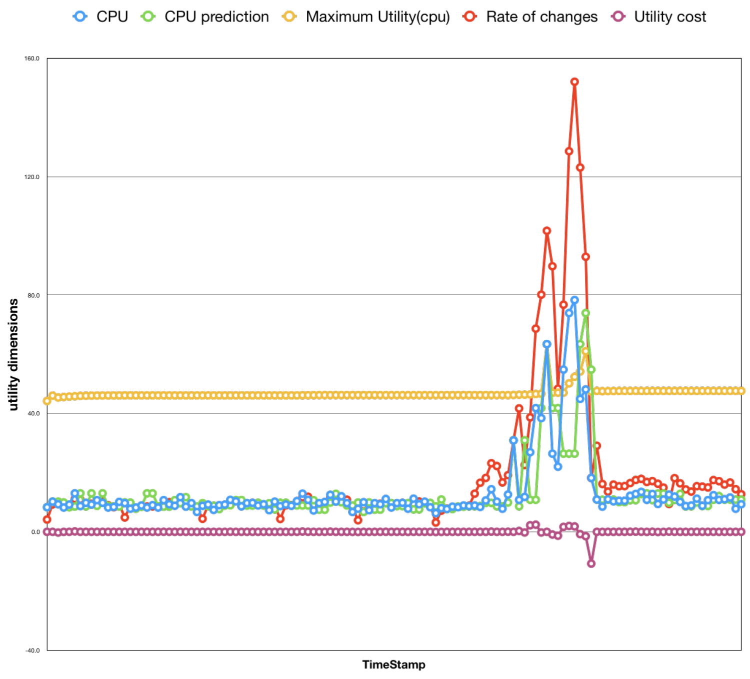

FIGURE 4 Utility(cpu) rate of changes and Cost calculated based in equ. 3 and 4

For this aim, first we run a stress test in the cluster manager until its CPU usage reaches 70%, which triggers an alert to the adaptation manager. The adaptation manager collects the current reading of the metrics, anomaly score, and anomaly likelihood (see figure 1). Then, the adaptation manager calculates the rate of changes in each metric to elect the metric that has the highest utility dimension. As example, Figure 3 is showing the utility of CPU usage, Memory usage, Disk reads (bytes/s), Disk writes (bytes/s), Docker Network (sent/received bytes/s) . The CPU usage has the highest utility as confirmed by the Utility indifference indicator shown in Figure 3. Also, the memory usage of the service shows slow rate of changes over time, which make it optional to be considered in the adaptation action. With regard to the utility of Disk Read/Write, the Figure 3 shows no divergent above the moving average (i.e. utility indifference curve) so it will not be considered in the next adaptation action. The docker network shows no changes over the time of the experiment as the load balancer and the reverse proxy manage to divert the traffic to many containers distributed in the cluster.

As the $Utility_(CPU)$ has the highest value of changes as shown in Figure 3. This will triggers an adaptation action to reason about the high demand of CPU usage, so the adaptation manager will create additional nodes and add them to the swarm cluster automatically. The number of nodes is equals to the utility cost calculated as in equ. 4. This results in adding new nodes to the swarm as shown in the snapshot [10] (A full visualised and analytics dashboard of the swarm after the adaptation). Once the CPU demand is reduced, the adaptation manager will calculate the variations of the utility and remove number of nodes equals to value returned by the utility cost function in equ. 4. A snapshot [11] of the system after executing the adaptation action to reason about the low level of the cpu usage.

In another scenario, we simulated Distributed Denial of Service attack to a web service running in the swarm, to verify that the adaptation manager will be able to accommodate the DDOS attack by adding more replicas to the service. In this case, we wish to verify the ability of the proposed model to dynamical adjust the number of service's replicas against the variations of the CPU usage and to maintain an acceptable response time of the web service. At the same time, it is very important that the adaptation action would not scale the service endlessly. So the utility cost is calculated to count the number of replicas needed. The outcome of this experiments is shown in Figure 2. The figure show how the number of scaled replicas are tuned linearly against the CPU usage. Also, it shows the number of steps taken by the adaptation manager to execute the scaling policy in/out. The adaptation manager counts the utility of the CPU metric and the utility cost to define the number of added/removed replicas. The adaptation manager waits for receiving a high utility value by sending heartbeat signal to get the latest value of utility dimension and cost

[10] goo.gl/pGVmwZ
[11] https://goo.gl/H5js2s

every 20s for a window of 300s. Once the utility of the CPU get the highest utility dimension, the adaptation manager calculates the number of replicas to be added/removed to the service. Also, the number of steps needed to achieve the desired state are counted as shown in Figure 2. The number of steps needed to perform the adaptation varies based on the severity and variation of the utility cost over time. Once the adaptation is applied and verified by winning the consensus algorithm votes, the service will be scaled and the adaptation manager puts a cool off timer of 300s before initiating new adaptation action. Also, it reset the steps timer.

The accuracy of the utility cost, rate of changes, and the maximum utility dimension are vital for the success of the adaptation process. So, Figure 4 depicts the calculation of the rate of changes and the utility cost to reach the desired number of nodes/replicas needed. We find the calculation accurately satisfies the adaptation objectives and provides the architecture with accurate calculation of the needed number of nodes/replicas. As shown in Figure 4, the number of nodes increases at the right time when the CPU demand spikes, then the number of nodes/replicas reduced just before the CPU demand is declined significantly as shown in Figure 4. The rate of changes in CPU usage declined so the Utility cost return negative value for the required number of nodes/replicas as long they are above the minimum number of nodes/replicas specified by the Dev-Ops. Also, as shown in Figure 4 The utility cost normalizes and tunes the CPU demand. This provides a great evidence that the employment of the utility dimension provides the adaptation cycle a dynamic variability over the needed/allocated resources. Instead off scaling the architecture in/out according to a static threshold.

The accuracy of the anomaly score and likelihood plays an important role for the successes the proposed model. So, in the following we will evaluate the accuracy of the anomaly detection algorithm implemented in this model. A famous technique for evaluating anomaly detection algorithm is the use of a confusion matrix[34]. The confusion matrix represents the total number of records inspected by algorithm as shown in Table 1. The matrix represents the relationship between the actual levels of normal behaviour and the detected level of normal behaviour by the algorithm. Also, it represents the relationship between the actual anomaly recorded in the system and the predicted/identified anomaly that the algorithm identified as anomalous behaviour.

5 | DISCUSSION

In case of our proposed model, If NUPIC anomaly detection considered a specific record as anomalous and it was an actual anomaly then this attempt is classified as a **True Positive**. If NUPIC classifies the data as normal behaviour and it was normal data then this attempt is classified as **True Negative**. If NUPIC classifies an anomalous behaviour as normal behaviour this means that NUPIC failed to detect anomaly. In the case, this attempt is classified as **False Negative**. If NUPIC classifies the data as anomalous behaviour but the data was normal behaviour then this attempt is considered a False Alarm/**False Positive**. Both True Positive and False Positive are important benchmark to measure the accuracy of intrusion detection systems. Table 1 summarizes the values used for evaluating the anomaly detection algorithm we used in this paper. The confusion matrix will be used to calculate the model detection rate, false positive, and accuracy. Those results will be compared against SNORT [12] [20]. SNORT is implemented as a container in the leader node of the cluster (see Figure 1). Based on the calculations of NUPIC anomaly detection shown in Table 1 and Snort in Table 2, it was found that our proposed model of using NUPIC performs significantly better in terms of the detection rate, false alarms, and accuracy. The false alarm rate in the model (0.0264) is lower than SNORT (1,12). In addition, the model provides a high rate of true alarms and anomaly detection as summarised in Table 3. The equations used in this evaluation are listed in 3.

TABLE 1 Results of the proposed anomalies detection model on confusion matrix

X= 1528	Predicted anomalies	Predicted normality
Actual anomalies (TP + FN) (55)	TP = 49	FN = 6
Actual normalies (FP + TN) (1473)	FP/ False Alarm = 38	TN = 1435

[12]https://snort.org

TABLE 2 Results of the SNORT anomaly detection on confusion matrix

X= 1528	Predicted anomalies	Predicted normality
Actual anomalies (TP + FN) (55)	TP = 32	FN = 23
Actual normalies (FP + TN) (1473)	FP/ False Alarm = 62	TN = 1411

TABLE 3 Comparison between SNORT Anomaly Detection and the Proposed Model

Rate	NUPIC	SNORT	Result
Accuracy	0.97	0.944	Implementation of NUPIC provides better accuracy $Accuracy = (TP + TN)/x$, over the SNORT anomaly detection.
Misclassification	0.0287	0.0554	The rate of false alarms is lower in the proposed model compared with SNORT. The Misclassification is calculated using $Misclassification = (FP + FN)/x$
True Positive Rate (TP)	0.89	0.58	$TP\ rate = TP/(Actual\ anomalies\ in\ x)$. The rate of true alarms in the model is significantly better than SNORT.
False Positive (FP)/ False Alarm	0.0264	1.12	The proposed model has a lower rate of incorrect classification of attacks calculated using $FP\ rate = FP/(Actual\ in\ x)$, but SNORT has a higher rate of not detecting attacks and considering them as normal behaviour
True Negative (TN	0.998	0.981	Both models- SNORT and the proposed model- have a good rate of detecting the normal behaviour of the system as calculated by $TN\ rate = TN/(Actual\ in\ x)$. However, the proposed model provides better results.
Precision	0.89	0.58	The precision $Precision = TP/(Predicted\ anomalies\ in\ x)$, of the proposed model is far better than SNORT. The gap between the two models is clear.
Prevalence	0.03599	0.03599	Both models have the same prevalence rate calculated by $Prevalence = (Actual\ anomalies)/x$ as the DDOS was simulated as part of the experiment.

The values in Table 3 clearly show that the proposed model performs better in terms of false alarms, detection rate, and accuracy. A further calculation, as shown below, could provide a better idea of the performance of the proposed model. The detection rate of SNORT is $32/((32 + 62)) = 0.340$ and this indicates that the proposed model has a better detection rate, as the proposed model used a machine-learning algorithm to build a model about the data and provides a prediction for each value. Those factors improve the detection rate compared with SNORT, which used a rule-based engine to match the incoming traffic and pre-configured rules.

The False Alarm of the proposed model is 0.0257. This indicates that the proposed model has a very low rate of providing a false alarm or identifying normal behaviour as anomalous. In the contrast, the false alarm for SNORT is $62/((62 + 1411)) = 0.042$. This value is a clear indication that SNORT would provide more false alarms about normal traffic in the system. This is due to the fact that SNORT considers one factor of identifying an anomaly which is the network traffic. In contrast, the proposed model continuously observed and learned the behaviour of so many performance criteria about the cluster in addition to the network traffic.

The overall accuracy of the proposed model is 0.97. The accuracy of SNORT is 0.944, due to the fact that SNORT is continuously monitoring the system network, but it has no means of understanding the behaviour of the system. The proposed model builds an accurate model of the system behaviour so it has better insight into any fluctuations or spikes in the data. This feature is considered to be very important for the success of the adaptation strategy and the self-healing property of the Microservices architecture.

However, we believe that this model manages to offer the Microservices architecture with continuous monitoring, continuous detection of anomalous behaviour, and provides the architecture with dynamic decision making based on the employment of multidimensional utility-based model. The results in above, shows high accuracy in detecting the anomaly and an accurate calculation of variant adaptation actions. It Also shows high success rate in performing horizontal and vertical autoscaling in response to contextual changes.

6 | CONCLUSIONS AND FUTURE WORK

Anomaly Detection using statistical approaches are not suitable for Microservices Architecture. It is difficult to determine the thresholds of a virtualised resources, which can be used to perform scaling or configuration policies. Also, it is difficult to have an accurate profile of Microservices cluster that has an accurate distribution among all measures found in the cluster. Real-time data streams introduces big challenges for machine learning algorithms as data streams are often found with massive volumes and high velocities.

Finally, this paper provides an architecture model that guarantees a continuous monitoring and observing of Microservices architecture. Also, it employs a mechanism for real-time unsupervised anomaly detection algorithm, which offers high accuracy of detecting normal and abnormal behaviour of the architecture. The uses of multidimensional utility-based model enables the architecture to dynamically elect a reasoning approach based on the highest utility dimension of context changes in the operational environment. The self-healing property is achieved by parameter tuning of the running services and dynamic adjustment of the swarm cluster. We believe integrating utility theory in the decision making process improves the effectiveness of the adaptation and reduces the adaptation risk including resources over-provisioning and thrashing. Also, it preserving the cluster state by preventing multiple adaptation to take place at the same time. Enabling a prediction over the metrics of the operational resources enables the architecture to schedule a proactive adaptation actions.

This model requires further improvement with regard to services compositions, and decentralised decision making. A complete decentralisation of the adaptation manager, as it is currently implemented as a central component in the leader node. Also, this model requires an enhanced mechanism of run-time configurations, as it supports currently parameter tuning of the architecture configurations. Also, the current implementation of NUPIC requires multiple implementations for each contextual changes. NUPIC has no support for swarming (model training) over multiple variables data model. Finally, we believe the ability of the Microservices to self-heal itself is achievable, but a major question stands related to how we could evaluate the self-healing property of Microservice architecture. This question require further investigation to find applicable runtime evaluation mechanisms.

References

1. Stubbs J, Moreira W, Dooley R. Distributed systems of microservices using docker and serfnode. In: IEEE; 2015: 34–39.
2. Salehie M, Tahvildari L. Self-adaptive software: Landscape and research challenges. *Transactions on Autonomous and Adaptive Systems (TAAS* 2009; 4(2).
3. Kapitsaki GM, Prezerakos GN, Tselikas ND, Venieris IS. Context-aware service engineering: A survey. *Journal of Systems and Software* 2009; 82(8): 1285–1297.
4. Jelasity OBM, Fetzer AMC, Moorsel vSLA, Steen vM. Self-star Properties in Complex Information Systems. 2005.
5. Horn P. Autonomic computing: IBM's Perspective on the State of Information Technology. tech. rep., 2001.
6. Cheng B, Lemos dR, Giese H, et al. Software Engineering for Self-Adaptive Systems: A Research Road Map (Draft Version). *Dagstuhl Seminar Proc. 08031* 2008.
7. Strang T, Linnhoff-Popien C. A context modeling survey. In: . 4. ; 2004: 34–41.

8. Cheng SW, Garlan D, Schmerl B. Evaluating the effectiveness of the rainbow self-adaptive system. In: IEEE. ; 2009: 132–141.

9. Mikalsen M, Paspallis N, Floch J, Stav E, Papadopoulos GA, Chimaris A. Distributed context management in a mobility and adaptation enabling middleware (madam). In: ACM; 2006: 733–734.

10. Cheung-Foo-Wo D, Tigli JY, Lavirotte S, Riveill M. Self-adaptation of event-driven component-oriented middleware using aspects of assembly. *Proceedings of the 5th international workshop on Middleware for pervasive and ad-hoc computing: held at the ACM/IFIP/USENIX 8th International Middleware Conference* 2007: 31–36.

11. Wei W, Fan X, Song H, Fan X, Yang J. Imperfect Information Dynamic Stackelberg Game Based Resource Allocation Using Hidden Markov for Cloud Computing. *IEEE Transactions on Services Computing* 2016; 11(1): 78–89.

12. Menascé DA, Dubey V. Utility-based QoS brokering in service oriented architectures. In: IEEE; 2007: 422–430.

13. Kakousis K, Paspallis N, Papadopoulos GA. Optimizing the utility function-based self-adaptive behavior of context-aware systems using user feedback. In: Springer; 2008: 657–674.

14. Sama M, Rosenblum DS, Wang Z, Elbaum S. Model-based fault detection in context-aware adaptive applications. In: ACM; 2008: 261–271.

15. Hirschfeld R, Costanza P, Nierstrasz OM. Context-oriented programming. *Journal of Object technology* 2008; 7(3): 125–151.

16. De Lemos R, Giese H, Müller HA, et al. Software engineering for self-adaptive systems: A second research roadmap. In: Springer. 2013 (pp. 1–32).

17. Patcha A, Park JM. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *COMPUTER NETWORKS* 2007; 51(12): 3448–3470.

18. Kruegel C, Vigna G. Anomaly detection of web-based attacks. In: ACM; 2003: 251–261.

19. Anderson D, Frivold T, Valdes A. Next-generation intrusion detection expert system (NIDES): A summary. 1995.

20. Roesch M. Snort: Lightweight intrusion detection for networks. In: ; 1999: 229–238.

21. Phua C, Lee V, Smith K, Gayler R. A comprehensive survey of data mining-based fraud detection research. *arXiv preprint arXiv:1009.6119* 2010.

22. Gupta D, Singhal S, Malik S, Singh A. Network intrusion detection system using various data mining techniques. In: IEEE. ; 2016: 1–6.

23. Buczak AL, Guven E. A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection. *IEEE Communications Surveys & Tutorials* 2016; 18(2): 1153–1176.

24. Bujlow T, Riaz T, Pedersen JM. A method for classification of network traffic based on C5. 0 Machine Learning Algorithm. In: IEEE; 2012: 237–241.

25. Golmah V. An efficient hybrid intrusion detection system based on C5. 0 and SVM. *International Journal of Database Theory and Application* 2014; 7(2): 59–70.

26. Haq NF, Onik AR, Hridoy MAK, Rafni M, Shah FM, Farid DM. Application of machine learning approaches in intrusion detection system: a survey. *IJARAI-International Journal of Advanced Research in Artificial Intelligence* 2015; 4(3): 9–18.

27. Hawkins J, Blakeslee S. *On Intelligence*. Macmillan . 2007.

28. Ahmad S, Purdy S. Real-Time Anomaly Detection for Streaming Analytics. *arXiv preprint arXiv:1607.02480* 2016; abs/1607.02480.

29. Lavin A, Ahmad S. Evaluating Real-time Anomaly Detection Algorithms - the Numenta Anomaly Benchmark. *CoRR* 2015; abs/1510.03336: 38–44.

30. Ahmad S, Lavin A, Purdy S, Agha Z. Unsupervised real-time anomaly detection for streaming data. *Neurocomputing* 2017; 262(Supplement C): 134–147.

31. Craig JW. A new, simple and exact result for calculating the probability of error for two-dimensional signal constellations. In: IEEE; 1991: 571–575.

32. Ongaro D, Ousterhout JK. In search of an understandable consensus algorithm.. In: ; 2014: 305–319.

33. Fishburn PC, Kochenberger GA. Two-piece von Neumann-Morgenstern utility functions. *Decision Sciences* 1979; 10(4): 503–518.

34. Kohavi R, Provost F. Confusion matrix. *Machine learning* 1998; 30(2-3): 271–274.

How to cite this article: <aurhor name>, <aurhor name>, <aurhor name>, <aurhor name>, and <aurhor name> (<year>), <journal title>, <*journal name*> <year> <vol> Page <xxx>-<xxx>