
Basel Magableh

Technological University Dublin, 453543@dit.ie

Follow this and additional works at: https://arrow.dit.ie/scschcomart

Part of the Computational Engineering Commons

Recommended Citation
Deep Q Learning for Self Adaptive Distributed Microservices Architecture

BASEL MAGABLEH1, AND MUDER ALMIANI23
1Technological University, Dublin, Ireland (e-mail: basel.magableh@dit.ie)
2Al-Hussein Bin Tala University, Ma’an, Jordan (e-mail: malmiiani@my.bridgeport.edu)
Corresponding author: Magableh B. (e-mail: basel.magableh@dit.ie).

ABSTRACT One desired aspect of a self-adapting microservices architecture is the ability to continuously monitor the operational environment, detect and observe anomalous behavior, and provide a reasonable policy for self-scaling, self-healing, and self-tuning the computational resources in order to dynamically respond to a sudden change in its operational environment. The behavior of a microservices architecture is continuously changing over time, which makes it a challenging task to use a statistical model to identify both the normal and abnormal behavior of the services running. The performance of the microservices cluster could fluctuate around the demand to accommodate scalability, orchestration and load balancing demands. To achieve the desired high levels of self-adaptability, this research implements microservices architectures model following the MAPE-K model. Our proposed architecture employs Markov decision process (MDP) to identify the transition from one cluster state to another. Our proposed architecture employs a deep Q-learning network (DQN) for dynamically selecting the adaptation action that yields the highest reward. This paper evaluates the effectiveness of using DQN and MDP agent to achieve high level of self-adaptability of microservice architecture. We argue in this paper that such integration between DQN and MDP in MAPE-K model offers microservice architecture with self-adaptability against the contextual changes in the operational environment. The self-adaptation property is achieved by allowing the MDP agent to explore the observation space and lets the DQN to select the adaptation policy with the highest reward, then the MDP agent executes the adaptation action and observes the changes. We believe integrating DQN into the adaptation action selection process improves the effectiveness of the adaptation and reduces the adaptation risk including resources over-provisioning and thrashing. The proposed model preserves the cluster state and prevents multiple actions to take place at the same time. Our model also guarantees that the executed adaptation action fits the current execution context and achieves the adaptation goals.

INDEX TERMS Anomaly detection, Microservices architecture, Q learning, runtime configuration, self-healing, reinforcement learning, policy approximation.

I. INTRODUCTION

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. With the rapid development of cloud infrastructures and virtualisation techniques, a high demand for building Microservices architectures in a complete virtualised environment has emerged. 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]. It is essential for Microservice architecture to be able to reason about its own state and its surrounding environment in a closed control loop and act dynamically at runtime to achieve high level of adaptability [2]. Such level of self-adaptation requires the Microservices architecture to be able to observe its current state and provide a suitable adaptation action so it can adjust itself to reason about various contextual changes.

1https://www.docker.com
2https://docs.docker.com/engine/swarm/
Nowadays, Microservices architecture does not have components that can guarantee continuous monitoring and adaptation of the operational environment. Also Microservices architecture cannot offer the architecture with dynamic capability to reason about context changes at run-time.

To achieve such a high level of adaptability, a microservices cluster should have i) a component for continuously monitoring the cluster and ii) a component for adaptation that can implement a reasonable reaction/scaling policy to accommodate the changes in the operating environment. Finding the policy that yields the highest reward presents a challenge to build a self-adapting microservices architecture that can dynamically adjust its own behaviour and heal itself against anomalous behaviour detected in real-time.

The proposed model in this paper follows the MAPE-K (Monitor, Analyse-Plan, Execute over a shared Knowledge) model in the design of the microservices architecture. Our model provides a mechanism for: a) continuous monitoring, b) context detecting of anomalous behaviour, c) dynamic decision making using reinforcement learning, d) enabling dynamic adaptation, based on the demand and the changes of the operational environment, and e) runtime verification and validation of the fitness of the selected adaptation strategy.

To achieve high levels of self-adaptability, this research employs Markov Decision Process in identifying the transition from one cluster state to another. Also, it employs a deep Q-learning network (DQN) that is able to select the optimal adaptation action that returns the highest reward gained from executing those actions. At the same time, the use of deep Q-learning guarantees that the knowledge from each pair of action-states is used in selecting future adaptation action to avoid adaptation failure and provide the maximum level of availability, reliability and scalability.

This paper contributes to the domain knowledge of self-adaptive microservices architecture by: i) presenting a working prototype of microservices architecture design according to the MAPE-K model. ii) adaptation agent implemented as MDP, which observes the microservices architecture and executes the adaptation actions iii) a deep Q-network used for selecting a sequence of adaptation actions that yield the highest reward and preserves the microservices cluster. iv) an open model that can be extended by other researchers to test various types of adaptation planning and execution algorithms and dynamic decision mechanism. Also, this model provides a framework to evaluate different types of reinforcement learning algorithms in continuous state and action spaces such as service oriented architecture and software system.

This paper is structured as follows: Section II provides an overview of self-healing architectures and surveys the approaches being employed for self-adaptation, anomaly detection, and reinforcement learning. Section III presents a model that can continuously observe Microservice architectures with Self-healing capabilities. Adaptation planning and execution is discussed in Section III-B. Section IV proposes a strategy for analysing and evaluating the capability of the model to detect anomalous behaviours and to trigger suitable adaptation actions. The implementation of this model is discussed in Section IV-A. Section IV-B is focused on the results; followed by a critical discussion of the effectiveness of this model. Section V summarises this research, highlighting its contribution and setting out future suggestions for future investigation.

II. RELATED WORK

A. SELF-ADAPTIVE SOFTWARE

Self-adaptive software is characterised by a number of properties best referred to as autonomic [3]. These the ‘self-* properties’ include Self-organisation, Self-healing, Self-optimisation and Self-protection [4]. Self-healing architecture refers to the capability of software system discovering, diagnosing and reacting to disruptions. Such architecture can also anticipate potential problems and, accordingly, take suitable actions to prevent a failure [4]. Self-adaptation aspects of microservices architectures require decision-making strategies, that can work in real-time. This is essential for a microservices architecture to reason about its own state and its surrounding environment in a closed control loop and to then act accordingly [2].

Typically, a self-adapting system should implements MAPE-K model including: i) Gathering of data related to the surrounding context (Context Sensing); ii) Context observation and detection; ii) dynamic decision making; iv) adaptation execution to achieve the adaptation objectives defined as QoS; v) verification and validation of the applied adaptation actions in terms of its ability to meet the adaptation objectives and to meet the desired QoS.

B. CONTEXT SENSING

However, there are many approaches are used for achieving high levels of self-adaptability though context sensing involving context collection, observation and detection of contextual changes in the operational environment [5]. Also, the ability of a system to dynamically adjust its behaviour can be achieved using parameter-tuning [6], component-based composition [7], or middleware-based approaches [8]. Another important aspect of a self-adaptive system is related to its ability to validate and verify the adaptation action at runtime based on game theory [9], utility theory as in [10], [11], or a model driven approach as in [12].

Context information (1) refers to any information that is computationally accessible and upon which behavioural variations depend [13]. 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], [5], [14], [15]. 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.
C. ADAPTATION PLANING AND EXECUTION

In microservices cluster, the performance of the cluster nodes could fluctuate around the demand to accommodate scalability, orchestration and load balancing issued by the cluster leader. This requires a model that: i) is able to detect anomalies in real-time, ii) can generate a high rate of accuracy in detecting any anomalies and iii) produces a low rate of false alarms. In addition, there will be a set of variations that can be used by the system to adapt to the changes in its operational environment. This adaptability requires a dynamic decision making that can calculate the weight of all possible adaptation actions based on the architecture constraint, anomaly score, and the confidence and accuracy of the anomaly score of the detected abnormal behaviour, and the desired/predicted cluster state. Then, the adaptation agent will execute the adaptation action and verify its success over the cluster architecture. Also, the adaptation agent will be able to self-tune and self-adjust the architecture’s parameters to meet high/low demand levels for services. Finally, the architecture will preserve the cluster state through the adaptation cycle involving: i) monitoring, ii) observing, iii) detecting, iv) reacting, and v) 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 a low rate of false alarms. The ideal method will then equip the architecture with adaptation strategies with high utility to reason about the detected anomalies and be able to self-adjust the architectural parameters and verify the adaptational actions at runtime without human intervention.

D. ANOMALY DETECTION

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 [16].

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

Within the statistical methods, the anomaly detection algorithm observes the activity of the software 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 the 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 high readings will be automatically categorised as intrusions thus triggering an alert [17].

Various statistical anomaly detection systems have been proposed and they have some advantages [18], [19]. One of this is that they can detect an anomaly without prior knowledge of the system itself. This facility 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 occur over long periods of time and the model performs better than other systems in detecting denial-of-service attacks [16].

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 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 [20]. 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 [20]. Data mining based anomaly detection systems have three major difficulties which prevent them from being widely adopted for use in microservices architecture [16]. Firstly, the low accuracy of detecting anomalous behaviour [16], [21], as the data mining process would require a large dataset with longer time intervals to be able to improve the accuracy of detection. Most data mining techniques make heavy demands on computational resources [22], a characteristic which negatively influences their adoption for use in microservice architecture [16]. 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 limitation is because the process of training, testing the model and performing the 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 that is able to both learn and improve its detection accuracy over time [24]. Machine learning-based anomaly detection models aim 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 focused on training a model based exclusively on past data [16]. 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 programme. 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 use 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 learning algorithm in this context can be found in [23], [25], [26]. Due to the opening deployment and limited resources found in a microservices cluster, it is very important to use a lightweight approach to data clustering and classification. Due to this issue, this research focuses on proposing an anomaly detection mechanism that is more suitable for use with microservices architecture and can be easily deployed with fewer and smaller footprints on the limited resources found in the tiny containers running in microservices cluster.

The Numenta Platform for Intelligent Computing (NUPIC) is based on the hierarchical temporal memory (HTM) model proposed in [27]. The HTM has been experimentally applied in real-time anomaly detection of streaming data in [28], [29]. It is important to note that 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 to the changes of the data statistics. It also detects extremely subtle anomalies with a very minimal 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. The most recent average of anomalies is then calculated using the Gaussian tail probability function (Q-function) [31].

E. REINFORCEMENT LEARNING

The goal of reinforcement learning (RL) is to provide the software agent with a possibility to learn a specific policy that can be used to take a decision among a set of actions by maximising the cumulative rewards [32]. Several efforts were made to employ Neural Networks in the implementation of RL algorithms as in [33]–[35]. The idea is to use the neural networks to identify the mathematical relationships between the input data, as well as to identify the maximum reward function of finding the output. Such effort can be found in [36] where RL and NN were used to play Atari games or Go games as in [37]. There are two popular approaches in deep RL algorithms: a) Deep Q Networks (DQN) and b) policy gradients. DQN is a form of Q-learning with function approximation using deep neural networks. The goal of DQN is to learn a Q-value from state-action pair, which is given by the deep networks, by minimizing temporal-difference errors [36]. Based on the DQN algorithm, various network architectures such as Double DQN [35] and DDQN [34] were proposed to improve performance and keep stability. Policy gradient methods directly learn the policy by optimising the deep policy networks with respect to the expected future reward using gradient descent. Williams et al. [38] proposed REINFORCE algorithm simply using the immediate reward to estimate the value of the policy. Silver et al. [39] proposed a deterministic algorithm to improve the performance and effectiveness of the policy gradient in high-dimensional action space. In the work of Silver et al. [39], it is shown that pre-training the policy networks with supervised learning before employing policy gradient can improve the performance of RL algorithms so it reach a state of convergence quickly.

III. DESIGN AND METHODOLOGY

A. SELF-HEALING MICROSERVICES ARCHITECTURE

One important aspect of a self-adapting Microservice architecture is its ability to continuously monitor the operational environment, detect and observe anomalous behaviour. By so monitoring it will also detect and provide a reasonable policy for self-scaling, self-healing, and self-tuning the computational resources so that those resources can dynamically be adapted to any sudden changes in its operational environment.

To validate the ideas presented in this paper, a working prototype of microservice architecture in Docker swarm, as shown in Figure 1, was designed and then developed. The cluster consisted of one leader and many manager and worker nodes. To meet scalability and availability, the cluster leader distributed the work load between the workers based on a Raft Consensus Algorithm [40]; as a result each service could be executed by assigning multiple containers across the cluster.

The microservices’ Architecture is shown in Figure III. The architecture was designed according to the MAPE-K (Monitor, Analyse, Plan, Execute over a shared Knowledge)
FIGURE 1. Distributed Microservices architecture implemented in Docker swarm

The architecture implements the following services:

1) **Monitoring Service**: Provides the continuous collection of fine-grained metrics about cluster nodes, services, and containers including: a) CPU usage, b) Memory, c) Disk ReadsBytes/sec, d) Network Read/s, e) network write/s and Disk Writes Bytes/sec.

2) **Analysing Service**: Responsible for reading the collected observations and calculating how anomalous the current observation is compared to the architecture’s historical behaviour. This comparison is achieved by implementing an anomaly detection service based on a NUPIC framework [30]. The NUPIC anomaly detection service [30] is continuously running over the streamed matrices collected in the matrices database, which enables the generation of the training model for the collected metric in 1). The collected real-time data is fed, on-the-fly, to NUPIC anomaly detection service [30], which provides two features: i) continuous detection of anomalous behaviour with high accuracy; ii) predictions about the architecture’s performance based on the collected historic data. In addition, the anomaly detection service is able to detect events as early as possible before the anomalous behaviour interrupts the functionality of the running services in the cluster [30].

3) **Adaptation Planning**: Once there is an anomalous behaviour detected which has both a high anomaly score and high likelihood, both values are calculated by the anomaly detection service as shown in Figure 1. What happens next is: a) the alert manager notifies the adaptation manager about the anomaly that has been detected; b) the adaptation manager selects the adaptation action(s) after calculating the Q value for all actions as explained in Section III-B; then, c) the adaptation manager uses the input of the metric values, anomaly scores, anomaly likelihoods, architecture constraint, as specified by the DevOp during deployment, and the desired/predicted QoS to d) calculate the best variation of the adaptation that has the highest reward using a deep reinforcement learning algorithm as explained in Section III-B.

4) **Adaptation Election**: The adaptation manager executes the action based on the aggregated value of the Q-value returned by the DQN. 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 policies, the adapter allow the adaptation actions to be fully completed and verified by the cluster leader according to the consensus performed by RAFT, then the adaptation manager will put a cool off timer before initiating new adaptation actions. This technique is used to a) avoid resources thrashing and b) to preserve the cluster state for auto-recovery. Finally, the adaptation manager sends the cluster leader a set of adaptation instructions.
that might involve tuning, configuration, or scaling.

5) Adaptation Verification: The cluster leader and all managers in the cluster will vote on the adaptation action based on the consensus algorithm [40]. The vote results are used to validate and verify the possibility of deploying the adaptation action. If the adaptation action won the voting, the adaptation action will be executed by the cluster leader, the adaptation manager recording the adaptation attempt as successful. If the adaptation action lost the voting process, then the adaptation manager maintains the current state of the cluster and records the adaptation attempt as failed. In both cases, the adaptation manager records the number of attempts used to complete the adaptation actions.

B. ADAPTATION STRATEGY

This research focuses on proposing a model that can continuously observe and monitor the microservices architecture and be able to provide an adaptation action that can maintain the cluster state with high availability. At the same time, the architecture should be able to respond to true positive alarms by suggesting a set of adaptation actions (adaptation strategy), that can be deployed in the cluster to achieve high levels of self-adaptation in response to changes in its operating environment. One of the main problems of self-adapting architecture is that it requires an algorithm: i) to learn how to choose an adaptation action from discrete action space and ii) to optimise the adaptation action to guarantee that the architecture will reach the adaptation objective [33]. This objective can be achieved using a reinforcement learning algorithm, which can guarantee high accuracy for selecting the best adaptation action that will fit in the current execution context.

Our adaptation agent follows the Markov decision process (MDP). MDP defined as a set of states \( s \in S \) and actions \( a \in A \). The transition model from state \( s \) to state \( \tilde{s} \) is defined as a function \( T(s, a, \tilde{s}) \) and the reward of this action in the new state \( \tilde{s} \) is defined by \( R(s, a, \tilde{s}) \), which returns a real value every time the system moves from one state to another. In microservices architecture identification of the set of possible adaptation actions is based on the observation of the current state of the cluster. Adaptation execution requires identifying a set of sequential actions to adapt the contextual changes by executing an adaptation action to reach the adaptation objectives or quality of services. Action selection provides the adaptation manager with positive reward once it reaches the adaptation objectives (i.e. service convergence) or negative reward for every failed adaptation.

However, the adaptation manager, with discrete adaptation actions, has no idea what the transition probabilities are! It does not know \( T(s, a, \tilde{s}) \), and it does not know what the rewards are going to be either (it does not know \( R(s, a, \tilde{s}) \)) once it moves from one state to another. It must experience each state and each transition at least once to know the rewards, and it must experience them multiple times if it is to acquire a reasonable estimate of the transition probabilities. Knowing the optimal state value is very useful in identifying the best adaptation actions. Bellman [42] found an algorithm to estimate the optimal state-action values called Q-values. The Q-value of a state-action pair is noted by \( Q(s, a) \). The \( Q(s, a) \) refers to the sum of discounted future rewards that the adaptation action expects to reach in a state \( s \) after selecting the adaptation action \( a \). Q-value estimation is not applicable in an environment with large sets of states and actions. Alternatively, neural networks could be used to estimate the Q-value by defining an approximation function and training the model in deep Q-network, this approach is called deep approximate learning (deep Q-learning). Deep Q-learning is a multi-layered neural network that for a given state \( s \) outputs a vector of action values using Markov Decision Process [42], using an approximation function to estimate the Q-value \( Q(s, a) \).

So, our adaptation manager will be using a deep Q-learning approach for identifying the best adaptation action that can return the highest reward once it reaches the desired adaptation objective. For this aim, we need to define the reward function i.e. Q-value \( Q(s, a) \) using a function approximation technique. To achieve this, we need to look back at the state \( s \) of the microservices architecture (see Figure 2). At each state \( s \) in Figure 2 there is a set of context values \( c \in C \) measuring the matrices of the operating environment such as: CPU, memory, disk I/O and Network as shown in Figure 2. The anomaly detection service provides us with useful information about how anomalous the current cluster start is by comparing it to the distribution of the previous learned state of the microservices architecture (see Figure 2). Our target is to provide the deep Q-learning algorithm with a scalable value that can be used to assign a weight \( W(s, c) \) for all context values \( c \) found in state \( s \), this value is calculated using equ. 1.

\[
W(al_m, C_m) = \sum_{i=1}^{m} a_i \cdot c_i \cdot as_i
\]  
\[
R(s, \alpha, \tilde{s}) = 1 - softmax(W(al_m, C_m))
\]

At each state \( s \) in Figure 2, the anomaly detection service calculates the Anomaly Score \( (as) \) and anomaly Likelihood \( (al) \) of all current context values \( c \). The anomaly likelihood accurately defines how anomalous the current context value is by comparing that value to the distribution of previously learned values about that specific context \( c \). This process enables the adaptation manager to scale the weight of each metric value over the distribution value calculated and then aggregated in the anomalous likelihood value. The anomaly likelihood is a scalar value between 0 to 1, meaning if the context \( c1 \) is referring to the value of CPU usage of 70% and the anomaly likelihood \( al(cpu) \) value is 1, then this might gives the Q-learning algorithm higher probability in selecting an adaptation action that would add new node to the cluster to reduce the CPU load and keeping the cluster in the desired state. In this case the reward will be probability
y = 1 − W(al, c) and it is calculated using equ. 2, which take the Softmax4 of all values calculated for all metrics. This process returns the cumulative reward from executing an action until the cluster reaches an optimal state; where 'optimal state' means the state that would return the highest reward to the DQN.

In this paper it is assumed that both the anomaly detection and deep Q-learning algorithms are fully trained at the initial state $s_0$, so at each new state (see Figure 2), the adaptation manager performs the following functions:

1) gets the current observation from the metric database service $GetObservation(s, c)$, which returns all metric values from the metrics database as in Figure 2.
2) gets the current anomaly score, anomaly likelihood from the anomaly detection service $GetAnomalyScore(s, c)$ as in Figure 2. This function call the anomaly detection service to return the values of anomaly score and anomaly likelihood for each context value $c \in C$, this will return a vector of the calculated values.

$4$The softmax function takes an un-normalized vector, and normalizes it into a probability distribution. That is, prior to applying softmax, some vector elements could be negative, or greater than one; and might not sum to 1; but after applying softmax, each element $x_i$ is in the interval $[0,1]$, and $\sum_i x_i = 1$

3) Get the possible adaptation action $a \in A$ defined in the action space in Figure 2. Those actions could be: a) adding/removing nodes, b) scale services in/out, c) trigger auto recovery and roll-back to previous state or d) stay at the current state.
4) finally, the DQN will run the adaptation policy for several times and at each step it will calculate the reward from executing the chosen action but it will not yet apply the action. After running several episodes, it will compute each action reward, which is the action that returns the least anomalous likelihood. This procedure will allow the DQN to balance the adaptation action by calculating the highest probability (as in equ. 2) that achieve the lowest value of the Anomaly likelihood.

In this paper it argue that the use of the concept ’anomaly likelihood’ to weigh the collected metrics provides an accurate calculation of the weight of metrics, as well as providing the model with better estimates of the adaptation action. So to achieve highly levels of self adaptability, this research implements a MAPE-K model that employs MDP agents to observe microservices architecture. Also, the adaptation manager employs a deep Q-learning algorithm that is able to select the optimal adaptation action that returns the highest reward from adaptation execution. At the same time, the use of reinforcement learning guarantees that the learned
knowledge from each pair of action-states is learned by the deep Q-learning, which will prevent the adaptation manager from executing any adaptation action that would fail and return negative reward. In the following section, the experimental setup and the tools used to construct the microservices architecture are describe, as is the structure of the neural network used in this experiment.

IV. RESULTS
A. EXPERIMENT SETUP

To validate the ideas presented in this paper, we designed and developed a working microservice architecture prototype in a Docker swarm 5 shown in Figure 1. The Docker swarm enables the architecture to add both manager and worker nodes. Each cluster has one leader, which maintains the cluster state and preserves the cluster logs. Also, the leader node initialises the vote of Raft consensus algorithm [40] to agree/disagree on a specific value based on the consensus by all nodes in the cluster. Only the leader node is allowed to commit and maintain the cluster state, as well as and initiating load balancing and orchestration. The leader node distributed the work load between the workers based on Raft consensus algorithm [40].

The main services implemented in this architecture are:
- Nodes metrics used to collect metrics from all nodes in the cluster.
- Docker containers metrics collector for collecting fine-grained metrics about all running containers in all nodes 7.
- Alert and notification manager used to notify the adaptation manager about contextual changes 8.
- Reverse proxy for routing traffic between all services in the cluster 9.
- Unsupervised Real-time anomaly detection based on NUPIC 10.
- Time series analytic and visualisation dashboard for observing the behaviour of the Microservices cluster 11.
- MDP adaptation agent that can observes the Microservices architecture and executes adaptation actions.
- Adaptation planning and execution is implemented via deep Q-learning network (DQN) using the Keras framework 12 and Tensorflow framework 13. The DQN collects the observation of the Microservices architecture via MDP agent. The DQN process the observation and propose an adaptation action to be executed by the MDP agent. The agent run the action and observes the environment and returns to the DQN the new observation and the reward value.

In this experiment, policy exploration using Boltzmann’s Q policy [43] is implemented. The Boltzmann Q policy is a stochastic exploration policy, where the probability of performing an action is related to the distribution of the associated Q-values. It is worth mentioning that the reward from executing a specific action is calculated using equ. 2. It was assumed that the anomaly likelihood provides a good measurement of how anomalous is the condition of the the current state when compared to the distribution of the previous state. This data would provide the DQN with an accurate measurement of the probability of a movement from state $s$ to $s'$. Therefore, calculating how anomalous the action is in the new state after transition will be used by the DQN to decide whether to stay in this state or move to another state.

$$\text{Cost}(s,c) = \frac{(\text{Current}(c_m) - \text{Predicted}(c_m)) \cdot \text{al}(c_m)}{\text{UsageTime} \cdot \text{Cost}(\text{instanceType})}$$  

Finally, to provide the DQN with proper regularisation about how many nodes/containers to add/remove at a certain state, it is important to provide an architecture constraint that can be used to prevent over provisioning, allocating or thrashing of the computational resources. The DQN uses equ. 3 to calculate the required number of node/replicas based on the current demand in the current state.

In equ. 3, the $\text{Current}(c_m)$ is the current value of the metric value. The $\text{Predicted}(c_m)$ refers to the predicted value of the metric value. The $\text{al}(c_m)$ is the anomaly likelihood value, calculated using an anomaly detection service (see Figure 2). The $\text{UsageTime}$ refers to the total number of hours the node is expected to be used per/day, this value is the mean of action duration returned from the DQN. The $\text{Cost}(\text{instanceType})$ 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 $\text{Cost}(s,c)$ is calculated against the constraint of $\text{Budget}$ as $\text{Cost}(u_m) \leq \text{Budget}$. The $\text{Budget}$ is assigned by the Dev-Op to reflect the value of the available budget, so the adaptation manager will not exceed this value in any case. A negative value returned by the $\text{Cost}(s,c)$ function means the number of nodes/replicas in the cluster should be reduced by the adaptation actions.

The $\text{Cost}(s,c)$ value is used to dynamically adjust the required number of nodes/replicas to reach an optimal state, which guarantees a high level of availability. Once there is a change in the cluster state the DQN repeats the processes of: i) collecting the observation, ii) proposing actions, iii) calculating the highest reward until it reaches a terminal/optimal state.

The neural network architecture shown in Figure 3 consists of ten layers. The input layer is the size of the observation space. In this experiment, the MDP agent collects the CPU

---

5 https://docs.docker.com/engine/swarm/
6 https://prometheus.io
7 https://github.com/google/cadvisor
8 https://prometheus.io/docs/alerting/alertmanager/
9 https://caddyserver.com/docs/proxy
11 https://grafana.com
12 https://keras.io
13 https://www.tensorflow.org
usage, memory usage, disk space, network I/O. for each metric the MDP agent gets the value from Prometheus and feeds it to the anomaly detection service which will calculate the prediction, anomaly score, anomaly likelihood. The collected observation by the agent at each state is used as an input for the DQN first layer (see Figure 3). The second layer is a dense hidden layer, comprising of 20 units. The RELU activation function is used in the remaining layers. DQN is a regression problem so we have used mean squared error (mse) loss function as cost function and we minimize that loss during the training between the Q-value. The last layer in the DQN as in Figure 3 is the output layer of the action space of total of 10 actions. The action space contains the following:

- **Action 0**: Create a leader node and initiate the swarm cluster
- **Action 1**: Add new node
- **Action 2**: Join a node to the cluster
- **Action 3**: Remove a node form the cluster
- **Action 4**: Add a manager node to the cluster.
- **Action 5**: Scale a service vertically (add/remove replicas).
- **Action 6**: Free disk space by deleting unneeded docker images and volumes.
- **Action 7**: Free allocated memory by removing dangling docker containers.
- **Action 8**: Create a new cluster and delete excising one, then all available nodes to the new cluster.
- **Action 9**: Maintain current state as long as it returns the highest reward.

The output layer uses a linear activation function, which outputs an action number from the above action space, which will be passed to the MDP agent to execute it. Finally, the DQN is implemented using an Adam Optimizer, which is used to calculate the Q-value at each state action pair and to then return the highest reward. A full code of the adaptation manager and services stack used in this experiment can be found in [14]. This live snapshot [44] provides a full virtualisation of all services running in the cluster.

The evaluation of the effectiveness of this model will be based on calculating the reward at each state action pair and the adaptation time needed to execute an action. Also we will calculate the number of adaptation attempts, successful convergence of services/nodes, or errors which leads to the creation of an unstable state of the cluster.

The evaluation of this model will come in three stages: i) evaluating the consistent behaviour of the cluster by evaluating the state of the swarm after allowing the DQN to play and run the cluster. The idea is to start with no nodes and the DQN should be able to create a new cluster and add the required number of nodes/replicas until the cluster reaches an optimal state. The decision will be left for the DQN to scale the cluster horizontally or vertically until the cluster reach a stable state as shown in the following section IV-B. ii) evaluating the accuracy of the model in electing the correct adaptation action by identifying the highest metric value that need to be consider in the adaptation and the cumulated reward function. iii) evaluating the mean of Q-value, loss function and adaptation duration. So, the evaluation objectives are:

1) **Obj. 1**: For the DQN to have the ability to manage a microservices cluster and to scale it horizontally or vertically until it reaches an optimal state.
2) **Obj. 2**: For the DQN to have the ability to handle dynamic changes in the cluster and to dynamically adapt to sudden changes, such as simulated stress test or Distributed Denial of Service attack (DDOS).
3) **Obj. 3**: for the architecture to have the ability to meet demands dynamically and maintain the cluster state.

### B. DISCUSSION

In the first experiment, the DQN is executed until it manages to find the state of creating a leader node and initiating the cluster with the all services mentioned above by deploying Docker stack of service via docker compose file, (i.e. docker compose is an architecture description language written in [14]https://github.com/baselm/mgr-selfhealing.git

![Figure 3. Deep Neural Network](image-url)
YAML for service configuration, compositions and deployment), as shown in Figure 4.

Then, the DQN will start to observe and listen for the observations collected by MDP agent and try to explore the optimal policy using the Boltzmann Q Policy. After running the DQN for 5000 episodes, we collect the mean of the Q value per episode was established, as shown in Figure 5. The figure shows that the DQN is indeed reaching a highest reward by trying different types of action until it reaches an optimal value. It is a good indicator that the DQN needs to run for many episodes before it can bring the architecture to an optimal state. Also, the adaptation time of performing each action was evaluated as this could indicate approximately how long it takes the DQN to bring the architecture to its optimal state.

Figure 6 shows the action duration in seconds and clearly indicates that DQN needs about 600 seconds to reach an optimal state as presented in Figure 7 and Figure 8. This result is confirmed by measuring the mean Q value against time as shown in Figure 9. After finishing the training, a test of the DQN was run for 10 episodes. The result of this test is the mean absolute error as shown in Figure 9. Also, the loss of the model is shown in Figure 9. The final architecture of the cluster is shown in Figure 4, demonstrating achievement of the first objective of the evaluation (obj. 1)

Second, we run a stress test in the cluster manager until its CPU usage reached 70%, which triggered an alert to the adaptation MDP agent. The agent collects the current observations, after which the anomaly service calculates the anomaly score, together with the anomaly likelihood, of the current state (see Figure 1). The DQN then calculates the Q-value for the possible action to take then select an action to add new nodes to the cluster. As example, Figure 10 is showing the CPU usage, memory usage, disk reads (bytes/s), disk writes (bytes/s), Docker network (sent/received bytes/s). The CPU usage has the maximum weight, according to equ. 2, as confirmed by the indifference indicator shown in Figure 10. It is worth mentioning that the Maximum weight is calculated by taking the Softmax of all matrices weight as in equ. 2. Also, the memory usage of the service shows slow rates of change over time, which makes the memory weight optional to be considered in the adaptation action by the DQN. With regard to the weight of disk read/write, the Figure 10 shows no divergence above the moving average (i.e. the 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 both managed to divert the traffic to many containers distributed across the cluster, which achieved the second objective of the evaluation (obj. 2).

The $W(CPU)$ has the highest value of changes as shown in Figure 10. This will trigger an adaptation action to reason about the causes behind the high level of CPU usage, so the DQN selects an action that will return the highest reward from decreasing the CPU value by adding additional nodes to the cluster. In this case, the number of nodes is equals to the cost calculated as in equ. 3. This results in adding new nodes to the swarm as shown in the snapshot [45] (A full visualised and analysis dashboard of the swarm after the adaptation) as confirmed in Figure 8. Once the CPU demand is reduced, the DQN will calculate the variations of the weight and remove numbers of nodes equal to the CPU usage can be found in [46], which achieved the second objective of the evaluation (obj. 2).

In another scenario, a distributed denial of service attack (DDOS) was simulated against a web service running in the cluster, to verify that the DQN will be able to accommodate the DDOS attack by adding more replicas to the service. In this case, we wished to verify the ability of the proposed model to dynamically adjust the number of service replicas against the variations of the network traffic and to maintain an acceptable response time for the web service. At the same time, it is very important that the adaptation action would not scale the service endlessly. So the cost is calculated to count the number of replicas needed. The outcome of this experiments is shown in Figure 12. The Figure indicates how the number of scaled replicas are tuned linearly against the CPU usage. Also, Figure 12 shows the number of steps taken by the adaptation agent to execute the scaling policy in/out. The adaptation agent calculates the weight of the CPU metric and the cost to define the number of added/removed replicas. The adaptation agent works to receive a high value by sending heartbeat signal to obtain the latest value of the observation and cost every 20s for a window of 300s. Once the weight of the CPU reaches its highest value, the adaptation agent calculates the number of replicas to be added to/removed from the service. Also, the number of steps needed to achieve the desired state are counted as shown in Figure 12. The number of steps needed to perform the adaptation varies based on the severity and variation of the cost over time. Once the adaptation is applied and verified by winning the consensus algorithm votes, the service will be scaled, and the adaptation agent puts on a cool-off timer of 300s before initiating any new adaptation action. Also, adaptation agent resets the steps timer, which achieves the second objective of the evaluation (obj. 2).
**FIGURE 5.** Reward value per episode

**FIGURE 6.** Action duration time
The accuracy of the cost, rate of changes, and the maximum weight are vital for the success of the adaptation process. So, Figure 11 depicts the calculation of the rates of change and the cost to reach the desired numbers of nodes and/or replicas needed. We find the calculation accurately satisfies the adaptation objectives and provides the architecture with accurate calculation of the needed numbers of nodes and/or replicas. As shown in Figure 11, the numbers of nodes increases at the right time when the CPU demand spikes, then the number of nodes/replicas reduces just before the CPU demand is declined significantly as shown in Figure 11. The rate of changes in CPU usage declined so the cost returned a 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 11, the cost normalizes and tunes the CPU demand. This provides a firm evidence that the employment of the weight as a reward function provides the DQN with dynamic variability over the needed/allocated resources, which achieves the second objective of the evaluation (obj. 3).

V. CONCLUSIONS AND FUTURE WORK

This paper presents microservices architecture model that has continuous monitoring, continuous detection of anomalous behaviour, and provides the architecture with dynamic decision making based on the employment of deep Q-learning. The results in above, shows high success rate in performing horizontal and vertical adaptations in response to various contextual changes. The uses of DQN enable the architecture to dynamically elect a reasoning approach based on the highest reward gained from each action state pair. The self-healing property is achieved by parameter tuning of the running services and dynamic adjustment of the swarm cluster. We believe integrating reinforcement learning in the decision making process improves the effectiveness of the adaptation and reduces the adaptation risk including the possibility of resources over-provisioning and thrashing. Also, our model preserves the cluster state by preventing multiple adaptations to take place at the same time, as well as eliminates the actions that would return the lowest reward. Currently, this model can be extended by adding new actions to the action space implemented in MDB agent, which will allows other researchers to run different types of experiments over this model.

Currently, a Docker swarm enables the cluster to have one leader, which prevents us from testing this model in multi agents/leaders environment. This enforces us to implement the adaptation agent as a central component in the leader node. Also, the current implementation of NUPIC anomaly detection requires multiple implementations for each contextual change. NUPIC has no support for training its model over multiple variables. Finally, we believe the ability of the microservices to self-adapt is a challenge that is achievable by integrating MDP and DQN in MAPE-K architecture.

REFERENCES

FIGURE 9. Adaptation Time Seconds

FIGURE 10. Dimensional analysis of the variations of the observation space as in 2


FIGURE 12. Dynamic Scaling of Web Service


BASEL MAGABLEH Dr. Basel Magableh is Lecturer in the School of Computer Science, Technological University Dublin. Basel has a BSc from the University of Yarmouk, Jordan, MSc from New York Institute of technology, USA and a PhD from Trinity College Dublin, Ireland. Basel has previously worked as a Chief Scientist in Mobile/Terminal Context Awareness in Huawei Technologies, Finland. He has also completed two post doctorate positions in UCD, and the Knowledge and Data Engineering Group (KDEG), Trinity College.

MUDER ALMANI Dr. Muder Almani is a Chair, Computer Information Systems Department / College of Information Technology Chair, Management Information Systems Department / College of Business Administration and Economics, Al-Hussein Bin Talal University, Ma’an - Jordan. Muder is actively researching in document handling, modems, peer-to-peer computing, power amplifiers, redundancy, security of data, storage management