# Self Healing Microservices Architecture: A case study in Docker Swarm Cluster

Basel Magableh and Luca Longo

**Abstract** 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.

Basel Magableh
Dublin Institute of Technology, Kevien Street, Dublin, Ireland, e-mail: basel.magableh@dit.ie

Luca Longo
Dublin Institute of Technology, Kevien Street, Dublin, Ireland e-mail: luca.longo@dit.ie

## 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. 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 runtime: 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

[1] https://www.docker.com

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

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 5 provides an overview of self-healing architectures and surveys the approaches for anomaly detection, and run-time configurations. Section 2 presents a model that can continuously observe Microservice architectures with Self-healing capabilities. Section 3 proposes a strategy for analysing and evaluating the capability of the model to detect anomalous behaviours and to trigger suitable adaptation actions. Section 4 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 Design and methodology

### *2.1 Self-healing Microservices Architecture*

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.

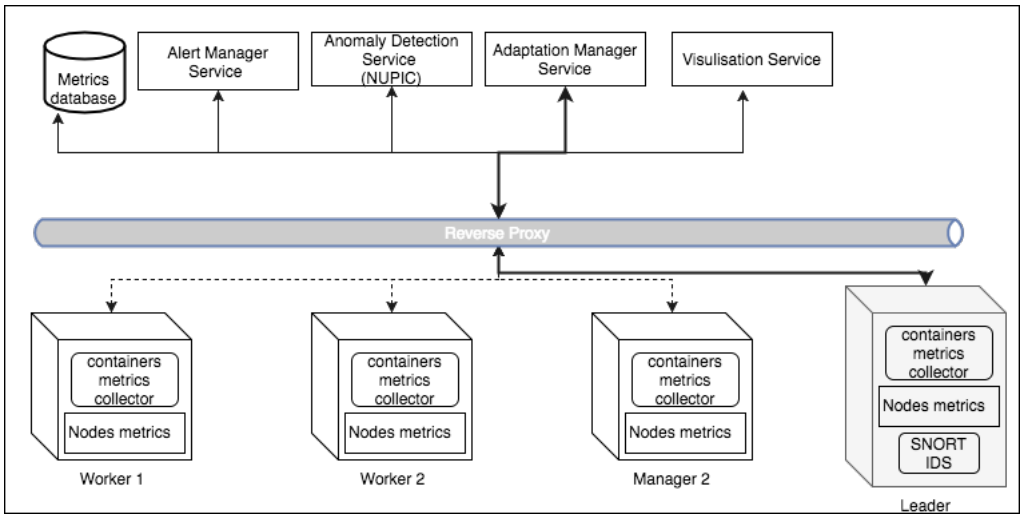


**Fig. 1** Microservices Architecture implemented in Docker Swarm

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 workers. Each cluster has one leader. To meet scalability and availability, the cluster leader distributed the work load between the workers based on Raft Consensus Algorithm [4]. This means that each service could be executed by assigning multiple containers across the cluster's nodes.

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 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].

This model 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 service [5] 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 anomaly detection service [5], 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. In addittion, the anomaly detection service is able to detect anomalies as early as possible before the anomalous behaviour is interrupting the functionality of the running services in the cluster as demonstrated by Ahmad et al. [5].

**Adaptation Election**: Once their is an anomalous behaviour detected with high anomaly score and likelihood, both values are calculated by the Anomaly Detection Service as shown in Figure 1. The Alert manager notifies the adaptation manager about the anomalous detected. The adaptation manager selects the adaptation

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

[4] https://prometheus.io

[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

action(s) after calculating the utility value for all actions as explained in the following Section 2.2. 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.

**Adaptation Execution**: The adaptation manager executes the strategies based on the aggregated value of the utility returned by the algorithm. 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 polices, the adapter allow the adaptation actions to be fully completed and verified by the cluster leader according to the consensus performed by RAFT, 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 [4]. 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 records the adaptation attempt as failed. In both cases, the adaptation manger records the number of attempts used to complete the adaptation actions.

## *2.2 Adaptation Election*

This research focuses on defining a model 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.

For this aim, This research focuses on proposing a model that can continuously observe and monitor the Microservices architecture and be able to detect anomalous behaviour with high accuracy and generate low rate of false alarms. At the same time, the architecture should be able to respond to True positive alarms by suggesting a set of adaptation policies (adaptation strategy), that can be deployed in the cluster to achieve high level of self-healing in response to changes in its operating environment.

To provide this model with dynamic policy election that guarantees high accuracy of selecting the best adaptation action that fits in the current execution context. The adaptation manager is implemented using a deep reinforcement algorithm (Deep

Q Learning) [**?**]. The problem of self-healing architecture requires an algorithm to learn how to choose an adaptation action from an infinitely large action space and to optimise the reward function to guarantees that the architecture will reach the adaptation objectives [**?**].

A deep-Q network (DQN) is a multi-layered neural network that for a given state $s$ outputs a vector of action values. Knowing the optimal state value is very useful in identify the best adaptation policy. Bellman found an algorithm to estimate the optimal state-action values called Q-values. The Q-value of a state-action pair is noted by $Q(s,\alpha)$. The $Q(s,\alpha)$ refers to he sum of discounted future rewards that the adaptation action expect to reach in a state $s$ after selecting the adaptation action $\alpha$.

The main problem with Q-learning with polices exploration is that it does not scale well in environment that has large set of states and actions. This make the number of possible state to be explored by the algorithm large to be applicable in Microservices architecture. The alternative for this is to use a utility function that can be approximately used to estimate the Q-Value function in deep neural network [**?**]. To efficiently employ the algorithm of Deep Q-learning in the adaptation manager we need to define the possible actions to be taken by the manager and the reward function. The reward function will be used to calculate the probability of the architecture to move from one state to another state. This where the anomaly detection algorithm plays a significant role. The anomaly likelihood will be used to calculate the reward function at each pairs of state and action. So the value of $Q(s,\alpha)$ is calculated based on the equation in 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 outcome of the reward function. 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 and to calculate the reward function.

$$fitness_i(s,\alpha) = \sum_{i}^{k} W_i ContxtObservationU(s) = \tag{1}$$

From utility theory, a von Neumann-Morgenstern utility function $U_i : X_i \rightarrow R$ assigns a real number to each quality dimension i, which we can normalize to the range [0, 1] [6]. 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 $[w1 : 0.6, w2 : 0.3, w3 : 0.1]$ [7]. 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 based on the outcome of the reward function implemented in the Deep Q-learning algorithm.

This is achieved by checking if the current variant is still the one offering the highest reward. So, the utility function is calculated as in equ. 2 proposed in [8]. In this case, the adaptation strategy refers to any parameter-based or compositional-based configuration of the architecture, maintaining its original functional properties [8]. 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.

Konstantinos et al. [8] 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, which will be used to calculate the reward function. We argue that the use of anomaly likelihood to weight the collected metrics provides an accurate calculation of the reward function and provides the model will better estimation of the adaptation action. 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 might give this metric high utility dimension so it will to be considered in the next adaptation action and it will get a high value of reward. 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 as it will return a low reward to the algorithm.

$$Q(S,\alpha) \equiv \frac{\sum_{i=1}^{k}(W \cdot fitness_i(S,\alpha))}{\sum_{i=1}^{k} W_i} \tag{2}$$

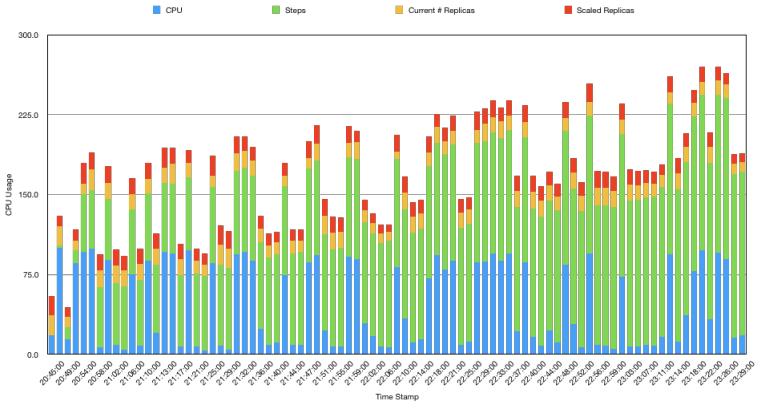


**Fig. 2** Dynamic Scaling of Web Service

## 3 Related work

Self-adaptive software is characterised by a number of properties best referred to as autonomic [11]. These the 'self-* properties' include Self-organisation, Self-healing, Self-optimisation and Self-protection [12]. 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 [12]. 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 [13]. 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 [14]. Also, the ability of the system to dynamically adjust its behaviour can be achieved using parameter-tuning [7], component-based composition [15], or Middleware-based approaches [16]. 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 [17], Utility theory as in [18, 8], or Model driven approach as in [19].

Context information (1) refers to any information that is computationally accessible and upon which behavioural variations depend [20]. 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, 13, 21, 14]. 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 [22].

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 [23].

Various statistical anomaly detection systems have been proposed and they have some advantages [24, 10]. 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 [22]. 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 [25]. 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 [25]. Data mining based anomaly detection systems have three major difficulties which prevent them from being widely adopted in Microservices architecture [22]. Firstly, the low accuracy of detecting anomalous behaviour [26, 22], 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 [22]. 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 [27].

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 [28]. 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 [22]. 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 [27]. 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 [29] 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 [29, 30, 27]. 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 [31]. HTM has been experimentally applied in real-time anomaly detection of streaming data in [32, 33]. 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. [5] 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. [5] 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) [34].

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.

## References


1. J. Stubbs, W. Moreira, R. Dooley, in *Science Gateways (IWSG), 2015 7th International Workshop on* (IEEE, 2015), pp. 34–39
2. M. Salehie, L. Tahvildari, Transactions on Autonomous and Adaptive Systems (TAAS **4**(2) (2009)
3. G.M. Kapitsaki, G.N. Prezerakos, N.D. Tselikas, I.S. Venieris, Journal of Systems and Software **82**(8), 1285 (2009)
4. D. Ongaro, J.K. Ousterhout, in *USENIX Annual Technical Conference* (2014), pp. 305–319
5. S. Ahmad, A. Lavin, S. Purdy, Z. Agha, Neurocomputing **262**(Supplement C), 134 (2017)
6. P.C. Fishburn, G.A. Kochenberger, Decision Sciences **10**(4), 503 (1979)
7. S.W. Cheng, D. Garlan, B. Schmerl, in *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS'09. ICSE Workshop on* (IEEE, 2009), pp. 132–141
8. K. Kakousis, N. Paspallis, G.A. Papadopoulos, in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"* (Springer, 2008), pp. 657–674
9. R. Kohavi, F. Provost, Machine learning **30**(2-3), 271 (1998)
10. M. Roesch, in *Lisa* (1999), pp. 229–238
11. O.B.M. Jelasity, A.M.C. Fetzer, S.L.A. van Moorsel, M. van Steen, (2005)
12. P. Horn, Autonomic computing: IBM's Perspective on the State of Information Technology. Tech. rep. (2001)
13. B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, R.M. Malek, H. Müller, S. Park, M. Shaw, M. Tichy, Dagstuhl Seminar Proc. 08031 (2008)
14. T. Strang, C. Linnhoff-Popien, in *Workshop on advanced context modelling, reasoning and management, UbiComp*, vol. 4 (2004), vol. 4, pp. 34–41
15. M. Mikalsen, N. Paspallis, J. Floch, E. Stav, G.A. Papadopoulos, A. Chimaris, in *Proceedings of the 2006 ACM symposium on Applied computing* (ACM, 2006), pp. 733–734
16. D. Cheung-Foo-Wo, J.Y. Tigli, S. Lavirotte, M. Riveill, Proceedings of the 5th international workshop on Middleware for pervasive and ad-hoc computing: held at the ACM/IFIP/USENIX 8th International Middleware Conference pp. 31–36 (2007)
17. W. Wei, X. Fan, H. Song, X. Fan, J. Yang, IEEE Transactions on Services Computing **11**(1), 78 (2016)
18. D.A. Menascé, V. Dubey, in *Web Services, 2007. ICWS 2007. IEEE International Conference on* (IEEE, 2007), pp. 422–430

19. M. Sama, D.S. Rosenblum, Z. Wang, S. Elbaum, in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering* (ACM, 2008), pp. 261–271
20. R. Hirschfeld, P. Costanza, O.M. Nierstrasz, Journal of Object technology **7**(3), 125 (2008)
21. R. De Lemos, H. Giese, H.A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N.M. Villegas, T. Vogel, et al., in *Software Engineering for Self-Adaptive Systems II* (Springer, 2013), pp. 1–32
22. A. Patcha, J.M. Park, COMPUTER NETWORKS **51**(12), 3448 (2007)
23. C. Kruegel, G. Vigna, in *Proceedings of the 10th ACM conference on Computer and communications security* (ACM, 2003), pp. 251–261
24. D. Anderson, T. Frivold, A. Valdes, (1995)
25. C. Phua, V. Lee, K. Smith, R. Gayler, arXiv preprint arXiv:1009.6119 (2010)
26. D. Gupta, S. Singhal, S. Malik, A. Singh, in *Research Advances in Integrated Navigation Systems (RAINS), International Conference on* (IEEE, 2016), pp. 1–6
27. A.L. Buczak, E. Guven, IEEE Communications Surveys & Tutorials **18**(2), 1153 (2016)
28. T. Bujlow, T. Riaz, J.M. Pedersen, in *Computing, Networking and Communications (ICNC), 2012 International Conference on* (IEEE, 2012), pp. 237–241
29. V. Golmah, International Journal of Database Theory and Application **7**(2), 59 (2014)
30. N.F. Haq, A.R. Onik, M.A.K. Hridoy, M. Rafni, F.M. Shah, D.M. Farid, IJARAI-International Journal of Advanced Research in Artificial Intelligence **4**(3), 9 (2015)
31. J. Hawkins, S. Blakeslee, *On Intelligence* (Macmillan, 2007)
32. S. Ahmad, S. Purdy, arXiv preprint arXiv:1607.02480 **abs/1607.02480** (2016)
33. A. Lavin, S. Ahmad, CoRR **abs/1510.03336**, 38 (2015)
34. J.W. Craig, in *Military Communications Conference, 1991. MILCOM'91, Conference Record, Military Communications in a Changing World., IEEE* (IEEE, 1991), pp. 571–575