Performance Analysis of Anomaly Detection Methods for Application System on Kubernetes with Auto-scaling and Self-healing

Yoichi Matsuo, Daisuke Ikegami
NTT Network Service System Laboratories, NTT Corporation
3-9-11 Midori-cho, Musashino-shi,
Tokyo 180-8585 Japan
Email:{yoichi.matsuo.ex, daisuke.ikegami}@hco.ntt.co.jp

***Abstract*—Kubernetes (K8s) is promising software for application systems since it makes application systems more flexible and robust by auto-scaling, which automatically scales up the application system resources when the application system is overloaded, and self-healing, which automatically recovers the application system from a failure. However, auto-scaling and self-healing make system operators' tasks complex. First, there is a delay, which is the time difference between executing auto-scaling or self-healing and recovering degraded application performance metrics such as response time. Second, the delay depends on types of abnormalities (i.e., overloads and failures). Moreover, the auto-scaling and self-healing cannot always recover the abnormality. Therefore, system operators need to understand the degree of abnormality (i.e., how much the application performance is degraded and how long the delay is). Although many anomaly detection methods have been developed, they have not considered auto-scaling or self-healing when the abnormality occurs. In this paper, we analyze the performance of anomaly detection methods with auto-scaling and self-healing in K8s by implementing anomaly detection methods, and deploying a web application system on K8s. Specifically, first, we verified that there is a delay that depends on types of abnormality by injecting anomalies into the web application system. Then, we evaluated the anomaly detection accuracy of each method by using the data collected from the web application. Finally, a clustering approach is used for anomaly scores, which are the outputs of these methods, to investigate whether anomaly detection methods can provide the degree of abnormality. The evaluations show that our analysis provides useful information for operators to manage the K8s with auto-scaling and self-healing.**

***Index Terms*—Kubernetes, Auto-scale, Self-healing, Anomaly detection**

I. Introduction

Kubernetes (K8s) [1] have been widely used for deploying application systems such as in Box [2], Spotify [3] and Booking.com [4]. K8s is container managing software, which mainly consists of nodes and pods, where a pod is the smallest unit in K8s, in which multiple containers are deployed, and nodes contain multiple pods. An application system is deployed by combining multiple containers with K8s and system operators monitor the metrics of the host servers, nodes, pods, and containers of K8s using anomaly detection methods so that system operators can detect an abnormal status of the application system.

K8s supports two functions, auto-scaling and self-healing, which make application systems more resilient against abnormal statuses. Auto-scaling automatically generates new pods to scale out computation resources when an application system overloaded, while self-healing regenerates new pods when pods or containers have failed in order to ensure that an application system continuously maintain normal status.

However, the auto-scaling and self-healing make system operators' task complex. First, there is a *delay*, which is the time difference between executing auto-scaling or self-healing and recovering degraded application performance metrics such as response time. Second, the delay depends on types of abnormalities (i.e., overloads and failures). Third, these functions may or may not recover an application system from an abnormal status. For instance, if the number of requests is temporally high, auto-scaling will recover the abnormal status once after pods are generated, initialized, and configured. However, if the number of requests to the application system is more than the number of requests that can be handled by a scaled out application system due to the system resource limitation, or the cause of the abnormal status is another part such as in a network between pods in K8s, auto-scaling will not recover the abnormal status [5]. Thus, system operations also need to monitor the application system metrics to see whether the degradation of application system will be recovered or not. Moreover, as a result, the application system will go down during execution of auto-scaling and investigation of the anomaly cause. Since the application system going down will lead users to stop using it, the downtime should be minimized. Therefore, system operators need to understand the *degree of abnormality* (i.e., how much the application performance is degraded, how long the delay is, and whether the auto-scaling or self-healing execution will recover performance degradation).

Anomaly detection methods are the key to providing information regarding abnormalities. Many anomaly detection methods have been developed for application systems with

This work is part of joint collaboration research project between Orange S.A. and NTT Corporation.

cloud computing platforms such as K8s or OpenStack [6] to detect abnormalities in application systems when an overload or failure occurs. However, these methods have not considered auto-scaling or self-healing when an abnormality occurs. Thus, it is not clear whether anomaly detection methods can detect abnormalities and the degree of abnormality under auto-scaling and self-healing.

In this paper, we analyze the performance of anomaly detection methods with auto-scaling and self-healing in K8s by implementing machine learning-based and deep learning-based anomaly detection methods and deploying a web application system on K8s. More specifically, first, we verified that there is a delay that depends on types of abnormalities by injecting anomalies into the web application system. Then, we evaluated the anomaly detection accuracy of each method using the data collected from the web application. Finally, the clustering approach is used for anomaly scores, which are the outputs of these methods, to investigate whether anomaly detection methods can provide the degree of abnormality.

The evaluations show that there is more than 1-minute delay and that delay is different for each among type of abnormality. The anomaly detection accuracy evaluation shows that Long Short-Term Memory (LSTM) is the best method for our dataset. Furthermore, we found out that anomaly scores of LSTM can potentially represent the degree of abnormality.

Contributions of this paper are as follows.

- First, this paper points out the lack of consideration regarding auto-scaling and self-healing complicating system operators' tasks. This paper deals with the effect of auto-scaling and self-healing on system operators' tasks, which has not been considered in previous studies of anomaly detection methods and application system management, and clarifies that it clear that how existing anomaly detection methods can provide useful information for operators to manage K8s with auto-scaling and self-healing.
- Second, comprehensive experiments are conducted by developing a web application service using K8s and implementing many anomaly detection methods. Nine types of abnormality are injected to degrade the web application performance and execute auto-scaling and self-healing.
- Third, by using collected metrics, we verified the delay and analyzed the performance of anomaly detection methods (i.e., evaluating accuracy of anomaly detection and investigating whether anomaly detection methods can provide the degree of abnormality) with auto-scaling and self-healing. We found out that anomaly scores of LSTM can possibly represent the degree of abnormality.

The rest of this paper is organized as follows. We explain related work in Section II. Then, the experimental settings and environment for our paper are described in Section III. The abnormality-injection experiments and evaluations are shown in Sections IV and V. Finally, we conclude this paper in Section VI.

II. Related Work

Various anomaly detection methods have been investigated for managing legacy application systems [7]–[10]. Since in the legacy application system, anomalies should be detected as soon as possible so that the system operators recover the system immediately, these papers try to detect anomalies using metrics such as central processing unit (CPU) usage, memory usage, traffic volume, and/or logs in the system. However, since the application systems in these papers do not use the cloud computing platform, performances of anomaly detection methods with auto-scaling and self-healing are not investigated.

For monitoring application systems with a cloud computing platform, anomaly detection methods have been developed [11]–[16]. Since the cloud computing platform increases the number of monitoring components and metrics and the relationship between the host server, cloud computing platform, and applications is complex, application performance degradation is challenging to detect. Thus, these methods collect metrics such as CPU usage, memory usage, and network traffic of not only host servers but also in the cloud computing platform, and extract hidden relationships between anomalies and metrics by using One-Class Support Vector Machine (OCSVM) [17], AutoEncoder [8] or LSTM [18]. Moreover, system operators have difficulty determining which layers (i.e., application layer, cloud computing platform, and host servers) cause the application performance degradation. Therefore, root cause analysis methods have also been studied [11], [16]. Jayathilaka et al. [11] used kernel density estimation to bind the application performance degradation to metrics in the cloud computing platform and host servers. Wu et al. [16], constructed a causality graph to identify failed components.

However, these methods do not consider execution of auto-scaling and self-healing. Thus, it is not clear how accurate anomaly detection is under auto-scaling and self-healing or what kind of information regarding degree of abnormality these methods can provide. Thus, anomaly detection methods for monitoring the application system on a cloud computing platform with auto-scaling and self-healing should be comprehensively analyzed.

III. Performance Analysis of Anomaly Detection

A. Motivation

In short, since it is not clear whether anomaly detection methods can detect abnormalities and the degree of abnormality under auto-scaling and self-healing, we verify the delay and analyze the performance of anomaly detection methods (i.e., evaluating accuracy of anomaly detection and investigating whether anomaly detection methods can provide the degree of abnormality) with auto-scaling and self-healing. Therefore, we deploy a web application and implement anomaly detection methods. Figure 1 shows an example of metrics in the applications system when the application status becomes abnormal. At around the 40 slot, due to increasing access to the application, the average response time starts to increase,

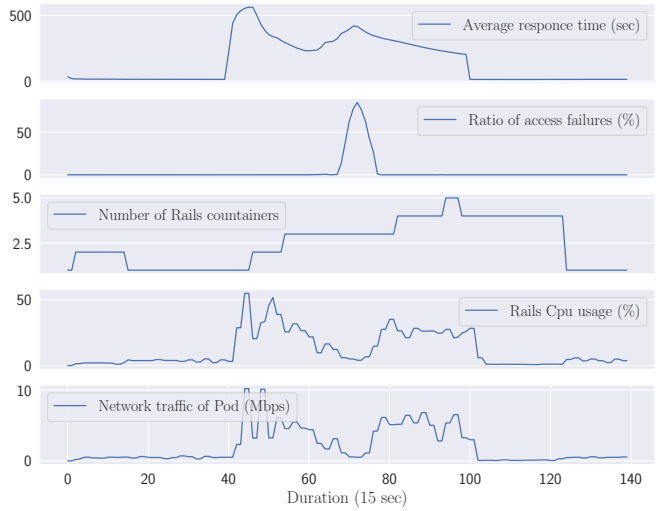


Fig. 1: Example of metrics in web application on K8s

i.e., the application performance starts to degrade. Since the auto-scaling is activated at around the 45 slot and the number of containers starts to increase, the CPU usage of containers will decrease. However, the average response time is not still recovered immediately. The average response time starts to improve at around the 50 slot and needs more time to be fully recovered. Moreover, access failures to the web application occurs at around the 70 slot. Thus, the system operators need to detect an anomaly and monitor the metrics until the application performance is recovered, even when the auto-scaling or self-healing are executed. For these reasons, the delay (which is the time difference between auto-scaling or self-healing and recovery of application performance) needs to be analyzed by injecting anomalies to the web application system, evaluating the anomaly detection accuracy of each method, and investigating information regarding the degree of abnormality.

B. Application system on Kubernetes and Experimental Environment

For our analysis, we deploy a web application using K8s and collect metrics of the web application, K8s, and host servers. The constructed web application is shown in Figure 2. As shown in the figure, a three-tier web application is implemented using Nginx, Rails, and MySQL containers, and Prometheus and Grafana [19], [20] are used for collecting and visualizing metrics, respectively. LOCUST [21], which is load testing software, is used to imitate user requests for the web application by sending HTTP requests to the web application. The average response time to the web application is also collected by Prometheus. For performance analysis, we use LOCUST to inject abnormal statues by setting a high number of requests to imitate an abnormality that occurs in the real web application. Detailed settings of LOCUST are described Section IV-A. We also use PUMBA [22], which is software to crash containers or pods, to inject container

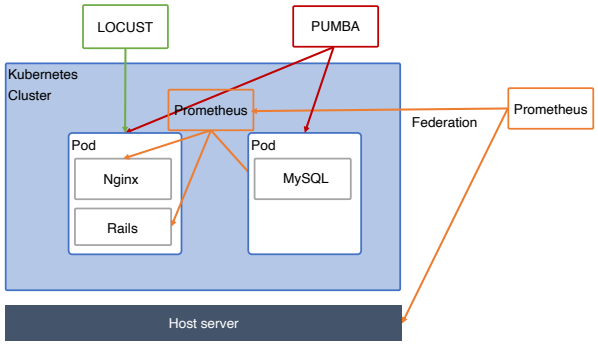


Fig. 2: Constructed web application and experimental environment

failures or pod failures. The detailed settings of PUMBA also are described in Section IV-B.

When the constructed web system status becomes overload, Horizontal Pod Autoscaler, which is the name of auto-scaling in K8s, or self-healing will work depending on the situation of the web application. Horizontal Pod Autoscaler controls the number of pods and has two parameters: 1) the threshold set by operators to determine whether the number of pod is increased, decreased, or kept and 2) the maximum number of pods. Threshold metrics can be set to CPU usage, memory usage, or other metrics, and Horizontal Pod Autoscaler controls the number of pods by the following equation.

$$\text{Threshold} < \frac{\text{Metric}_{\text{threshold}}}{\#\text{pods}} \quad (\#\text{pods} \leq \text{pods}_{\text{max}}), \qquad (1)$$

where $\text{Metric}_{\text{threshold}}$ is the metric for threshold, $\#\text{pods}$ is the number of running pods, and pods_{max} is the parameter for the maximum number of running pods in K8s. On the other hand, when a certain pod fails in the application system, self-healing regenerates new pods so that the number of running pods equal the number of running pods before the failure occurred.

We collect the metrics of host server, K8s, and the web application in normal status and abnormal status caused by LOCUST and PUMBA using Prometheus every 15 seconds. Regarding settings of our web application, we set the number of containers of Nginx and Rails to 2 and containers of MySQL to 1. For the setting of auto-scaling, we select CPU usage and set $\text{Metric}_{\text{threshold}}$ as 35 % and pods_{max} as 5.

C. Implementation of Anomaly Detection Method

For analyzing the performance of anomaly detection methods, density-based, machine learning (ML) based, and deep learning (DL) based methods are implemented, which are explained in Section II. We explain each method briefly in this section.

Let T be the final time slot of collected metrics, where time t is slotted as $t = 1, 2, \ldots, T$ with window size ω. Let $\boldsymbol{x}_t$ be a l-dimensional vector whose elements are metrics at time t , where l is the number of kinds of metrics collected from the application system and y_t is a label of the

application system at time t, which takes 1 if the application system is normal and -1 otherwise. Since we investigate the performance of machine learning-based and deep learning-based anomaly detection methods, a training dataset and a test dataset are prepared. Let the d-th training data be a set of tuples, $\{(\boldsymbol{x}_t, y_t)\}_{t=1}^{T_d}$, and $\mathcal{D}_{\text{train}}$ be a set of the d-th training data, $\{\{(\boldsymbol{x}_t, y_t)\}_{t=1}^{T_d}\}_{d=1}^{|\mathcal{D}_{\text{train}}|}$. Similarly, let $\mathcal{D}_{\text{test}}$ be a set of d-th test data, $\{\{(\acute{\boldsymbol{x}}_t, \acute{y}_t)\}_{t=1}^{\acute{T}_d}\}_{d=1}^{|\mathcal{D}_{\text{test}}|}$. Here, $|\cdot|$ denotes the cardinality. In the training phase, weights of anomaly detection methods are trained using a training dataset $\mathcal{D}_{\text{train}}$, and in the test phase, the anomaly score of $\acute{x}_t$ is calculated to estimate whether $\acute{y}_t$ is a normal or abnormal status. Here $\acute{}$ denotes data in the test dataset, y_t in the training dataset $\mathcal{D}_{\text{train}}$ always takes 1, and $\acute{y}_t$ in the training dataset $\mathcal{D}_{\text{test}}$ takes 1 or -1 depending on the application system status. For anomaly detection methods that do not need a training phase, only test data is used.

1) LOF: Local Outlier Factor (LOF) is a density-based method, in which the anomaly score of a certain data point is calculated using the average distance of the k-nearest neighbor. Let $N_k(x_t)$ be a set of data in k-nearest neighbor of $\acute{x}_t$, and $\text{reachability} - \text{distance}_k(\acute{x}_t, B)$ be $max\{k - \text{distance}(B), d(\acute{x}_t, B)\}$. Then, the local reachability distance of $\acute{x}_t$ is calculated as follows.

$$\text{lrd}_k(\acute{x}_t) = \frac{|N_k(\acute{x}_t)|}{\sum_{B \in N_k(\acute{x}_t)} \text{reachability} - \text{distance}_k(\acute{x}_t, B)} \quad (2)$$

The anomaly score $\text{LOF}_k(\acute{x}_t)$ for the test data is calculated by the following equation.

$$\text{LOF}_k(\acute{x}_t) = \frac{\sum_{B \in N_k(\acute{x}_t)} \text{lrd}_k(B)}{|N_k(\acute{x}_t)| \text{lrd}_k(\acute{x}_t)}. \quad (3)$$

More details can be found in Breunig et al. [7].

2) OCSVM: One-Class Support Vector Machine (OCSVM) is an unsupervised machine learning trained by normal status data so that x_t in normal status data satisfies the following equation.

$$w_{\text{OCSVM}}^T x_t - b \geq 1, \quad (4)$$

where w_{OCSVM} is the weight vector of OCSVM. Then, test data $\acute{x}_t$ is calculated to determine whether the status is normal or abnormal by using the trained OCSVM. More details can be found in Tax and Duin. [17].

3) AE: AutoEncoder (AE) is AN unsupervised encoder-decoder based method in which parameters are trained by a normal status dataset so that input data and output data by AE are the same. In AE, for the encoder, weight parameters W_{AE}^j in the j-th layer are trained by the following equation.

$$z^j = \sigma(W_{\text{AE}}^j z^{j-1} + b^j), \quad (5)$$

where z^0 is x_t, andσ is activation function, and b^j are bias functions. For the decoder, weight parameters $\tilde{W}_{\text{AE}}^j$ in the j-th layer are trained by the following equation.

$$\tilde{z}^j = \sigma(\tilde{W}_{\text{AE}}^j \tilde{z}^{j-1} + \tilde{b}^j), \quad (6)$$

where $\tilde{z}^0$ is z^J, and J is the number of AE layers. Then, AE is trained so that the reconstructed vector $\hat{x}_t$ is equals to x_t as follows.

$$L(x_t) = \|x_t - \hat{x}_t\|_2. \quad (7)$$

where $\hat{x}_t$ is calculated by encoding x_t and decoding z^J. To calculate the anomaly score for test data $\acute{x}_t$, the difference between $\tilde{x}_t$ and predicted $\hat{x}_t$ is calculated by using Equation (7). More details can be found in Sakurada and Yairi. [8].

4) LSTM: Long Short-Term Memory (LSTM) [18], which is a type of recurrent neural network, predicts the data x_{t+1} from time-series $x_0, \ldots, x_t$. In LSTM, x_{t+1} is predicted as follows.

$$\begin{aligned} f_t &= \sigma(W_f x_t + U_f x_{t-1} + b_f) \\ i_t &= \sigma(W_i x_t + U_i x_{t-1} + b_i) \\ o_t &= \sigma(W_o x_t + U_o x_{t-1} + b_o) \\ c_t &= f_t \circ \sigma(W_c x_t + U_c x_{t-1} + b_c) \\ \hat{x}_{t+1} &= o_t \circ \sigma(c_t), \end{aligned} \quad (8)$$

where f_t is the forget gate at time t, i_t is the input gate at time t, o_t is the output gate at time t, c_t is the cell state vector at time t, $W_f, W_i, W_o, W_c, U_f, U_i, U_o$ are weight matrices, and b_f, b_i, b_o, b_c are bias. To calculate the anomaly score for the test data $\acute{x}_t$, the difference between $\acute{x}_t$ and predicted $\hat{x}_t$ is calculated by using Equation (8). More details can be found in Jozefowicz et al. [18].

5) LSTM-AE: LSTM-AE is an unsupervised encoder-decoder based method with LSTM. In LSTM-AE, first, x_t is input to LSTM and h_t is output by replacing x_{t-1} and $\hat{x}_{t+1}$ with h_{t-1} and h_t in Equation (8), respectively. Then, h_t is input to AE, and $\hat{x}_t$ is output using Equation (6). To calculate the anomaly score for the test data $\acute{x}_t$, the difference between $\acute{x}_t$ and predicted $\hat{x}_t$ is calculated by using Equation (7). More details can be found in Diamanti et al. [12].

IV. Experiments

In this section, we describe how to inject abnormalities to the constructed web application and the settings of LOCUST, PUMBA. Metrics during experiments are collected by using Prometheus, and collected data is used for evaluating delay of auto-scaling and self-healing and anomaly detection methods.

In the following subsection, we first explain the LOCUST setting to inject overload status for executing auto-scaling. Then, we show the PUMBA settings to inject failures for executing self-healing.

A. Overload Cases

To imitate overload of the web application and execute the auto-scaling, we send a specified number of HTTP requests using LOCUST. LOCUST has three parameters: the maximum number of requests per second, number of pitches (which represents the increase in requests per second until the maximum number of requests is reached), and duration. Thus, to extensively analyze the effect of auto-scaling, we imitate various overload situations that might occur in real web

TABLE I: Settings of LOCUST

Experiment	Maximum request per second	Pitch per second	Duration (min)
Normal status	10	5	180
Experiment 1	50	5	20
Experiment 2	100	5	10
Experiment 3	100	5	20
Experiment 4	100	50	10
Experiment 5	300	50	10
Experiment 6	500	50	10

applications, by preparing six types of LOCUST setting (which are combinations of maximum number of requests per second, the number of pitches, and duration) and collect metrics of the web application.

Experiments 1, 2 and 3 are ramp-up types that gradually increase the number of requests. An example of these cases is when a website becomes popular due to gradually becoming widely known by the users. Experiments 4, 5 and 6 are burst types that drastically increase the number of requests. An example of these cases is when a website appears in a TV show and many people try to access it. By changing the maximum number of requests per second, the number of pitches, and duration, we prepared six types of experiments. The settings of LOCUST of each experiment are summarized in Table I.

Additionally, to imitate the normal status of the web application, we conducted the experiment by setting the maximum number of requests to 10, the number of pitches per second to 5, and duration to 180 min. Metrics during normal status are used as training data, and metrics during each experiment are used as test data for anomaly detection methods.

B. Failure Cases

To fail the web application and execute self-healing, we inject failures to pods by using PUMBA. Note that PUMBA can kill containers and pods, so we killed pods in our experiments. Since PUMBA randomly sends the kill signal among specified pods with specified intervals, we imitated three types of failures that might occur in real web applications and configured the settings of PUMBA so that it randomly kills pods including Nginx, Rails, and MySQL.

The first type is killing a pod, which is Experiment 7. The example of this case is an accidentally failed pod. For this type, we stop the PUMBA once a pod is killed and self-healing automatically regenerates the killed pod. The second type is killing several pods which is Experiment 8. The example of this case is a compound failure in which multiple pods simultaneously fail. The third type is killing the same pod many times, which is Experiment 9. The example of this case is a software failure in which the same pod repeatedly fails due to software bugs. The settings of PUMBA are summarized in Table II. During Experiments 7, 8, and 9, we use LOCUST to imitate the user access to the web page. The settings of LOCUST are the same as those of the normal status described in Section IV-A.

We experimentally killed the pod including Nginx and Rails, and the pod including MySQL in each Experiment 7 and 9.

TABLE II: Settings of PUMBA

Experiment	Failed pod	The number of failures
Experiment 7	One pod	one time
Experiment 8	Multiple pods	between one time and 10 times
Experiment 9	One pod	more than 10 times

Since each experiment described above is executed three times, the number of test data for overload cases and failure cases are 18 and 15 (3 for Experiment 8 and 6 for Experiment 7 and 9), respectively. Thus, $|\mathcal{D}_{\text{test}}|$ is 32. For training data, $|\mathcal{D}_{\text{train}}|$ is 1.

V. Results and Evaluation

In this section, we verify the delay (which is the time difference between auto-scaling or self-healing and recovery of application performance by injecting anomalies to the web application system), evaluate the anomaly detection accuracy of each method using the data collected from the web application, and investigate information regarding the degree of abnormality by each anomaly detection method by using metrics described in Section IV.

A. Delay of Auto-scaling and Self-healing

To verify the delay, we measured the time from the start of the response time degradation.

Table III summarizes the mean time and the standard deviation of duration until response time starts to be improved by auto-scaling, duration until scaling out, and duration until Rails CPU usage starts to improve from when response time degraded. In the experiments, the minimum and maximum delays are 1 minute in Experiment 1 and 9.5 minutes in Experiment 6, respectively. For Experiments 5 and 6, appropriately 9 minutes elapsed before the response time started to improve. Thus, the delay depended on types of abnormality. Furthermore, it took more time for the response time to fully recover.

Figure 1 shows sequences of metrics in Experiment 1. Here, the interval of each data point is 15 seconds. As described in Section III, average response time took appropriately 90 to start to improve and appropriately 15 minutes to fully recover.

Comparing the duration until scaling out with duration until Rails CPU usage starts to improve, by taking into account the standard deviation, once auto-scaling increases the system resources, CPU usage tends to improve except in Experiment 1.

Table IV also shows the delay for failure cases. In the failure cases, since self-healing regenerates the failed pods within 15 seconds once the pods are killed by PUMBA, we may not be able to collect the duration from executing self-healing to regenerating the pod. However, initialized processes are needed in the regenerated pods and containers in the regenerated pods, so there is a period in which the web application is not accessible. Thus, after regenerating pods and containers, CPU usage starts to improve after appropriately

TABLE III: The delay, time difference between auto-scaling, and recovery of application performance

Experiment	Duration until response time starts to recover (sec)	Duration until scaled out Rails containers (sec)	Duration until CPU usage of Rails starts to improve (sec)
Experiment 1	95.0 (±8.7)	35.0 (±31.2)	110.0 (±67.6)
Experiment 2	190.0 (±8.7)	95.0 (±52.7)	105.0 (±26.0)
Experiment 3	185.0 (±8.7)	115.0 (±22.9)	100.0 (±31.2)
Experiment 4	195.0 (±0.0)	130.0 (±8.7)	120.0 (±15.0)
Experiment 5	600.0 (±0.0)	145.0 (±8.7)	120.0 (±15.0)
Experiment 6	735.0 (±77.9)	175.0 (±95.3)	135.0 (±77.9)

TABLE IV: The delay, time difference between self-healing and recovery of application performance

Experiment	Duration until failures starts to reduce (sec)	Duration until the container regenerated (sec)	Duration until CPU usage of each container starts to improve (sec)
Experiment 7	40.0 (±39.9)	-	30.0 (±19.0)
Experiment 8	45.0 (±30.0)	-	10.0 (±17.3)
Experiment 9	40.0 (±24.5)	-	20.0 (±24.5)

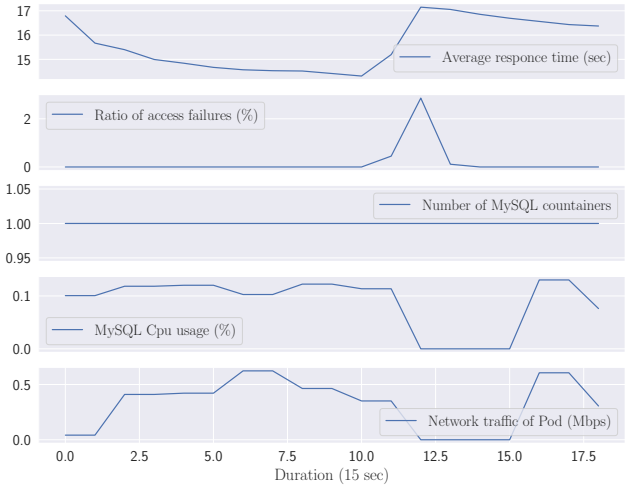


Fig. 3: Metrics in Experiment 7

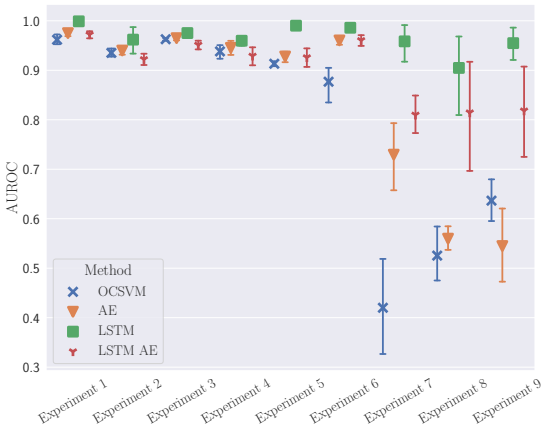


Fig. 4: AUROCs of all methods

30 seconds on average. Then, the performance of the web application finally starts to improve after 40 seconds, and the standard deviation of delay is different each experiment.

Figure 3 shows the sequence of metrics in Experiment 7. Here, although it seems that a pod and a container in that pod was always running, a container was in fact killed once. This is because since Prometheus collects metrics at 15-second intervals, we could not detect the period in which containers failed in this experiment. Since in this case, the pod which includes a MySQL container was killed by PUMBA around between 12 and 15, the values were missing for CPU usage of the MySQL container and network traffic of the pod in which the MySQL container is placed. Thus, the access failures occurred during these times. Note that since the ratio of access failures is calculated by dividing the number of access failures at a certain time slot by the maximum number of requests, the ratio of access failures can be greater than 1.

Owing to this analysis, we verified that there is the delay that depends on types of abnormality.

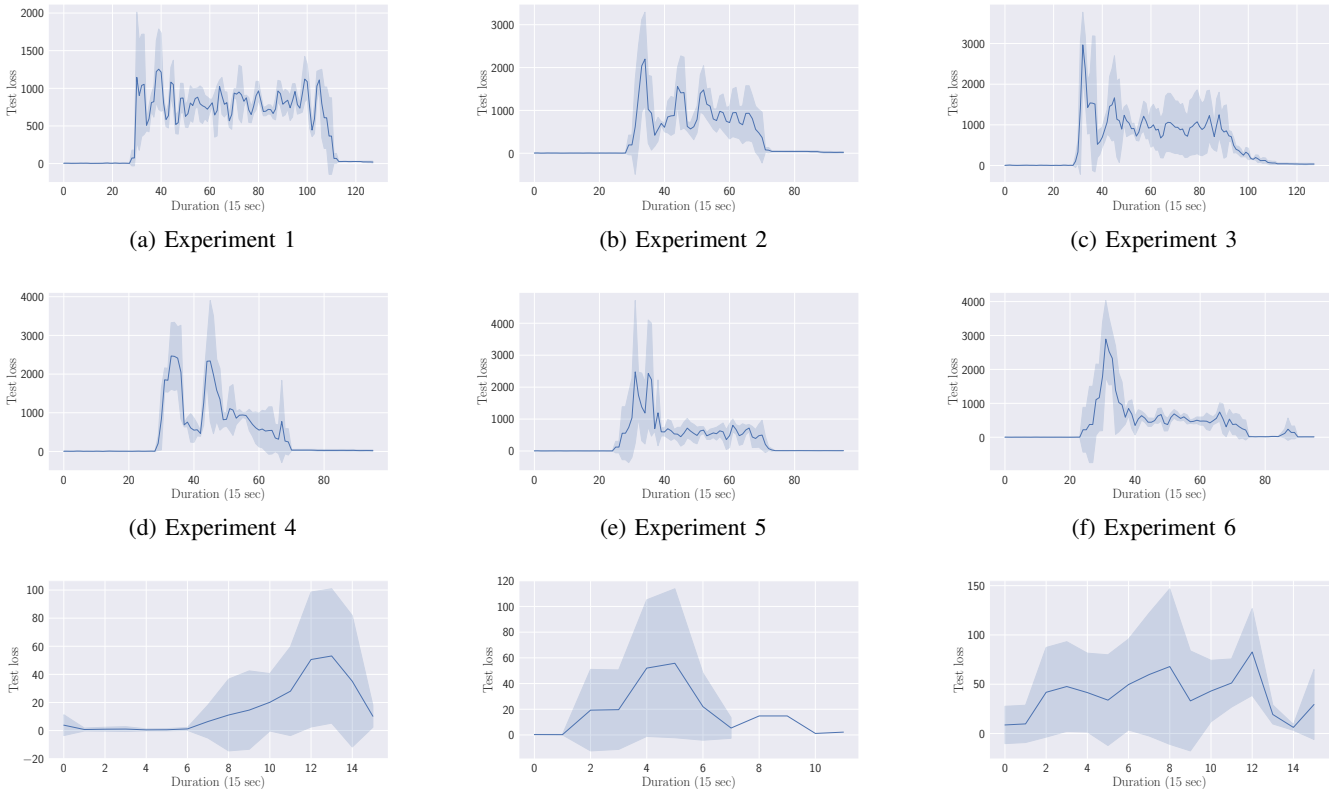
B. Anomaly Detection Accuracy

We evaluated the anomaly detection accuracy for the web application on K8s with auto-scaling and self-healing. For training data, we used three hours of normal status data that includes 19 kinds of metrics (i.e., $l = 19$) on host servers, pods, and containers, such as CPU usage, memory usage, disk read and write, network traffic, and the number of containers. For test data, data during the experiments described in Section IV is used. To label test data, we calculated the average and standard deviation of average response time and ratio of access failures during normal status. If the average response time or ratio of access failures at a certain data point in the test data exceeds a certain value (which is the average response time plus three sigma, which is the standard deviation in normal data), we label that data as abnormal, and otherwise normal. As an evaluation metric of anomaly detection methods, the area under the receiver operating characteristic (AUROC) is selected. LOF and OCSVM were implemented using Scikit Learn [23], and AE, LSTM, and LSTM-AE were implemented using Pytorch [24]. Each method was evaluated three times.

For the hyper-parameters of each anomaly detection

TABLE V: AUROCs of anomaly detection results

Experiment	LOF	OCSVM	AE	LSTM	LSTM AE
Experiment 1	0.035 (±0.016)	0.962 (±0.016)	0.974 (±0.011)	0.999 (±0.001)	0.972 (±0.012)
Experiment 2	0.070 (±0.014)	0.935 (±0.014)	0.939 (±0.016)	0.961 (±0.041)	0.922 (±0.022)
Experiment 3	0.057 (±0.012)	0.938 (±0.023)	0.946 (±0.022)	0.958 (±0.015)	0.929 (±0.028)
Experiment 4	0.087 (±0.017)	0.913 (±0.007)	0.928 (±0.017)	0.986 (±0.011)	0.925 (±0.030)
Experiment 5	0.101 (±0.021)	0.877 (±0.063)	0.960 (±0.014)	0.987 (±0.010)	0.960 (±0.017)
Experiment 6	0.039 (±0.005)	0.963 (±0.004)	0.963 (±0.007)	0.974 (±0.012)	0.952 (±0.013)
Experiment 7	0.369 (±0.164)	0.420 (±0.218)	0.730 (±0.145)	0.958 (±0.083)	0.810 (±0.089)
Experiment 8	0.461 (±0.108)	0.525 (±0.087)	0.560 (±0.037)	0.905 (±0.143)	0.814 (±0.184)
Experiment 9	0.529 (±0.217)	0.637 (±0.093)	0.545 (±0.159)	0.955 (±0.072)	0.818 (±0.206)
Average	0.241 (±0.232)	0.748 (±0.237)	0.802 (±0.198)	0.964 (±0.067)	0.885 (±0.127)



(a) Experiment 1 (b) Experiment 2 (c) Experiment 3

(d) Experiment 4 (e) Experiment 5 (f) Experiment 6

(g) Experiment 7 (h) Experiment 8 (i) Experiment 9

Fig. 5: Test loss of LSTM for each experiment

method, we prepared several patterns and evaluated the anomaly detection performance for each hyper-parameter. For LOF, we prepared 1, 10, 20, 30, and 40 as the numbers of neighbor points. For OCSVM, we prepared LINEAR kernel and RBF kernel, and kernel hyper-parameter of each kernel is set to $10^{-3}, 10^{-2}$ and 10^{-1}. For AE, three layers are prepared, and the final hidden dimension is set to 2, 4, and 6. For LSTM, two-layered LSTM was used and the final hidden dimension is set to 2, 4, and 6. For LSTM-AE, similar to the LSTM, two-layered LSTM was used and final hidden dimension is set to 2, 4, and 6. We also set the learning rate of AE, LSTM, and LSTM-AE as $10^{-3}, 10^{-2}$ and 10^{-1}.

The AUROCs and standard deviation of each method for each experiment are summarized in Table III and Figure 4, which show the results of the best pattern of hyper-parameters for each anomaly detection method. In the anomaly detection method, LSTM is the best for each experiment. This is because LSTM can take into account time dependency, which is important for anomaly detection since used data such as CPU usage, memory usage, and network traffic has time dependency. Although LSTM-AE has a LSTM structure, it does not directly take into account time dependency during the decoding. Thus, LSTM-AE and other methods that also cannot capture time dependency have lower AUROCs than LSTM. LOF, a density-based method, is the worst at detecting anomalies in the web application system.

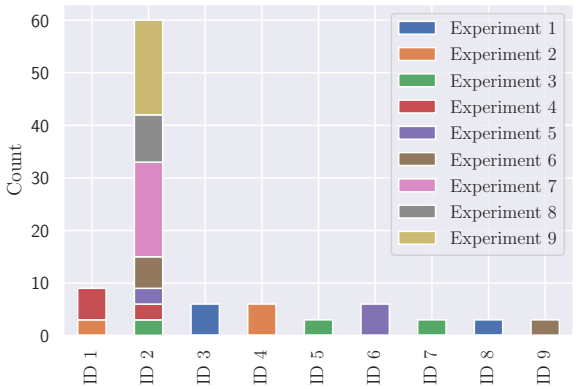


Fig. 6: Clustering result of OCSVM for each experiment

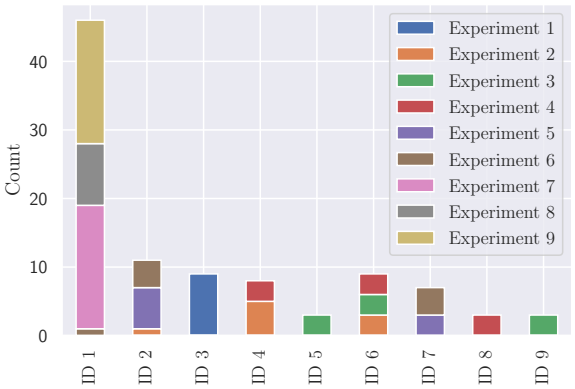


Fig. 8: Clustering result of LSTM for each experiment

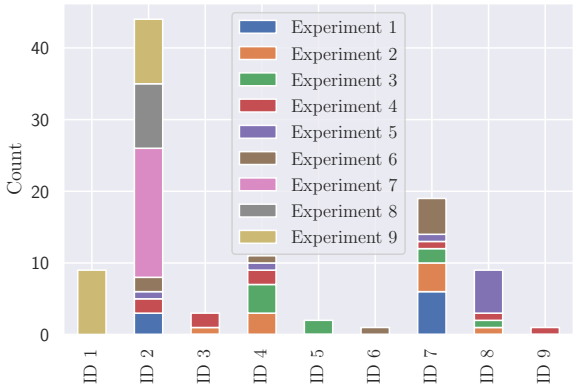


Fig. 7: Clustering result of AE for each experiment

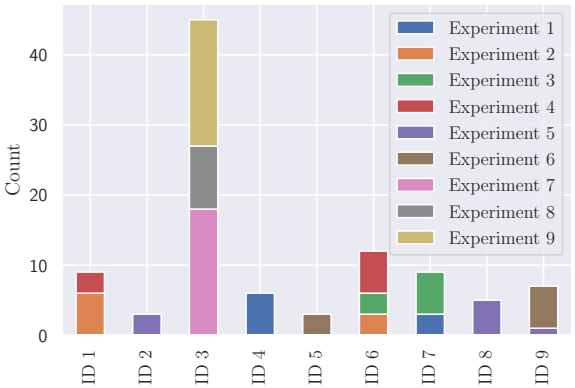


Fig. 9: Clustering result of LSTM AE for each experiment

C. Investigating Degree of Abnormality

We investigated whether anomaly detection methods can provide information of the degree of abnormality in each experiment by visualizing and clustering the anomaly scores. Since how much the application system performance degrades and duration of each experiment are different, the anomaly score that represents the distance from the normal state may differ among experiments. Therefore, we tried to extract that information by clustering those anomaly scores and evaluated the clustering results.

Figure 5 shows the sequence of test loss (i.e., anomaly scores) of LSTM for each experiment. For each experiment in overload cases, the maximum scores are different, and the higher the number of maximum requests, the higher the maximum score tends to be. This is because, since metrics such as CPU usage are affected by the number of maximum requests, the difference between predicted data by LSTM and test data also depends on the number of maximum requests. Moreover, comparing anomaly scores of LSTM of each experiment, the anomaly scores of ramp-up types (i.e., Experiments 1, 2 and 3) tend to be flatter than those of burst types (i.e., Experiments 4, 5, and 6). Thus, LSTM has the potential to extract the characteristics and degree of abnormality in each experiment. For each experiment in failure cases, the anomaly scores are lower than those in overload cases. This is because, since the self-healing regenerates pods quickly, the metrics recovered sooner than in overload cases as shown in Tables III and IV.

To investigate how each anomaly detection method extracts the degree of abnormality, we clustered the anomaly scores of each anomaly detection method except LOF since the AUROC of LOF was very low. By treating each sequence of anomaly scores as a vector, we used the K-means algorithm [25] for clustering and set the number of centroids as 9. Figures 6, 7, 8, and 9 summarize clustering results of each method for each experiment. ID and count represent centroid id and the number of test data classified into that id, respectively. For each method, the anomaly scores of failure cases (i.e., Experiments 7, 8, and 9) are classified into the same ID. This is because the anomaly scores of overload cases and failure cases have different characteristics. For the case of LSTM in Figure 8, all anomaly scores of Experiment 1 labeled as ID 3, and ID 5 and 9 have only the results of anomaly scores of Experiment 3. ID 2 and ID 7 represent anomaly scores Experiments 5 and 6, which are the burst types, except a small number

of anomaly scores of Experiment 2 in ID 2. However, the K-means algorithm was not able to classify anomaly scores of Experiments 7, 8, and 9 and Experiments 2, 3, and 4 whose maximum numbers of requests are the same. Thus, the clustering results of anomaly scores of LSTM depend on abnormality types (i.e., overload cases or failure cases) and the maximum number of HTTP requests. This means that LSTM can provide information regarding how much the application performance is degraded, which is one part of the degree of abnormality, but cannot provide how long the delay lasts for overload cases, which is another part of the degree of abnormality.

Clustering results of OCSVM are similar to those of LSTM. Figure 6 shows that all IDs except ID 1 and ID 2 have anomaly scores of only one type of experiment. However, ID 2 has not only experiments of failure cases but also all types of experiments of overloaded cases. Moreover, clustering results of the other two methods are worse, e.g., almost all anomaly scores labeled as ID 2, ID 4, ID 7, or ID 8 in the AE case 7. Thus, the ability to classify the results of anomaly scores is highly dependent on the anomaly detection methods.

VI. Conclusion

In this paper, we analyzed the performance of anomaly detection methods with auto-scaling and self-healing in Kubernetes (K8s). By deploying a web application on K8s and implementing anomaly detection methods, we evaluated the anomaly detection accuracy of each method using the data collected from the web application. Furthermore, a clustering approach was used for anomaly scores to investigate information regarding the degree of abnormality. The evaluation results show that our analysis provides useful information for operators to manage K8s with auto-scaling and self-healing since we found that Long Short-Term Memory (LSTM) can extract information regarding how much the application performance is degraded, which is one part of the degree of abnormality.

The following remains for future work. First, we will conduct more evaluations using datasets in a large application system. Although we comprehensively prepared experiment types and injected anomalies, we need to validate that anomaly detection methods can extract characteristics and the degree of abnormality by using a large application system and various types of anomalies. Second, we will investigate the difference in each anomaly detection method regarding clustering results. Third, we need to investigate each anomaly detection method in real-time situations. Although we used full sequences of test data (i.e., data from the start to end of an anomaly) in clustering, system operators need to know the types of abnormality as soon as possible. Thus, clustering method needs to be developed that use only several points of data from the start of an anomaly.

References

[1] "Kubernetes," accessed on 19th, April, 2021. [Online]. Available: https://kubernetes.io/

[2] "Box," accessed on 19th, April, 2021. [Online]. Available: https://www.box.com/home

[3] "Spotify," accessed on 19th, April, 2021. [Online]. Available: https://www.spotify.com/us/

[4] "Booking.com," accessed on 19th, April, 2021. [Online]. Available: https://www.booking.com/

[5] "slack," accessed on 19th, April, 2021. [Online]. Available: https://slack.engineering/slacks-outage-on-january-4th-2021/

[6] "openstack," accessed on 19th, April, 2021. [Online]. Available: https://www.openstack.org/

[7] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "Lof: Identifying density-based local outliers," *SIGMOD Rec.*, vol. 29, no. 2, p. 93–104, 2000.

[8] M. Sakurada and T. Yairi, "Anomaly detection using autoencoders with nonlinear dimensionality reduction," in *Proc. MLSDA*, ser. MLSDA'14, 2014, p. 4–11.

[9] A. Mahimkar, Z. Ge, J. Wang, J. Yates, Y. Zhang, J. Emmons, B. Huntley, and M. Stockert, "Rapid detection of maintenance induced changes in service performance," in *Proc. the 7th CoNEXT*, 2011.

[10] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs," in *Proc. 28th IJCAI*, 2019, pp. 4739–4745.

[11] H. Jayathilaka, C. Krintz, and R. Wolski, "Detecting performance anomalies in cloud platform applications," *IEEE Trans. on Cloud Comput.*, vol. 8, no. 3, pp. 764–777, 2020.

[12] A. Diamanti, J. M. S. Vilchez, and S. Secci, "Lstm-based radiography for anomaly detection in softwarized infrastructures," in *32nd ITC*, 2020, pp. 28–36.

[13] Y. Li, Z. M. J. Jiang, H. Li, A. E. Hassan, C. He, R. Huang, Z. Zeng, M. Wang, and P. Chen, "Predicting node failures in an ultra-large-scale cloud computing platform: An aiops solution," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 2, 2020.

[14] J. Wahba, H. Soliman, H. Bannazadeh, and A. Leon-Garcia, "Graph-based diagnosis in software-defined infrastructure," in *12th Int. Conf. CNSM*, 2016, pp. 243–247.

[15] Y. Zuo, Y. Wu, G. Min, C. Huang, and K. Pei, "An intelligent anomaly detection scheme for micro-services architectures with temporal and spatial data analysis," *IEEE Trans. Cogn. Commun. Netw.*, vol. 6, no. 2, pp. 548–561, 2020.

[16] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "Microrca: Root cause localization of performance issues in microservices," in *NOMS*, 2020, pp. 1–9.

[17] D. M. J. Tax and R. P. W. Duin, "Support vector data description," *Mach. Learn.*, vol. 54, no. 1, p. 45–66, 2004.

[18] R. Jozefowicz, W. Zaremba, and I. Sutskever, "An empirical exploration of recurrent network architectures," in *Proc. 32nd ICML*, ser. ICML'15, 2015, p. 2342–2350.

[19] "Prometheus," accessed on 19th, April, 2021. [Online]. Available: https://prometheus.io/

[20] "Grafana," accessed on 19th, April, 2021. [Online]. Available: https://grafana.com/

[21] "Locust," accessed on 19th, April, 2021. [Online]. Available: https://locust.io/

[22] "Pumba," accessed on 19th, April, 2021. [Online]. Available: https://github.com/alexei-led/pumba

[23] "Scikit learn," accessed on 19th, April, 2021. [Online]. Available: https://scikit-learn.org/stable/

[24] "Pytorch," accessed on 19th, April, 2021. [Online]. Available: https://pytorch.org/

[25] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proc. the 5th Berkeley Symposium on Math. Stat and Prob*, vol. 1, no. 14, 1967, pp. 281–297.