Thoth: Automatic Resource Management with Machine Learning for Container-based Cloud Platform

Akkarit Sangpetch, Orathai Sangpetch, Nut Juangmarisakul and Supakorn Warodom
Department of Computer Engineering, Faculty of Engineering, King Mongkut's Institute of Technology Ladkrabang, 1 Chalongkrung Road, Ladkrabang, Bangkok, Thailand

Keywords: Cloud Computing, Scheduling, Container, Platform-as-a-Service.

Abstract: Platform-as-a-Service (PaaS) providers often encounter fluctuation in computing resource usage due to workload changes, resulting in performance degradation. To maintain acceptable service quality, providers may need to manually adjust resource allocation according to workload dynamics. Unfortunately, this approach will not scale well as the number of applications grows. We thus propose Thoth, a dynamic resource management system for PaaS using Docker container technology. Thoth automatically monitors resource usage and dynamically adjusts appropriate amount of resources for each application. To implement the automatic-scaling algorithm, we select three algorithms, namely Neural Network, Q-Learning and our rule-based algorithm, to study and evaluate. The experimental results suggest that Q-Learning can the best adapt to the load changes, followed by a rule-based algorithm and NN. With Q-Learning, Thoth can save computing resources by 28.95% and 21.92%, compared to Neural Network and the rule-based algorithm respectively, without compromising service quality.

1 INTRODUCTION

Computing paradigm has been radically shifted in the past decade: from dedicated, on premise infrastructure to on-demand cloud computing. Many enterprises and organizations turn to cloud for IT services so that they can focus on the core businesses. Cloud providers have been leveraging on virtualization technology to provide elastic and flexible IT infrastructure in a form of virtual machines (VMs). However, using VMs tend to incur inefficient resource usage since, in practice, many VMs run the same operation system and similar dependent software, wasting computing resources on the same content. In order to mitigate this inefficiency problem, the container concept has been introduced. Many containers could share not only the same infrastructure, but also the underlying software and dependencies, such as operating systems and runtime components, increasing the resource usage efficiency.

Additionally, container technology can address fundamental challenges found in software development and deployment – that is the discrepancies in terms of software versions and runtime dependencies in development, testing and production environments, increasing risk of software malfunction. With containers, developers can package all coupled components and dependencies into a pod where all components are co-scheduled and co-located. Hence, developers can compartmentalize micro-services of an application and exercise continuous integration process more easily, allowing applications become more agile.

Container-based cloud, e.g. Docker, has emerging in mainstream in the past few years. Then, major cloud providers, such as Google, Microsoft, Amazon, IBM, have followed. Scaling container service is also available where developers can specify parameters and rules to determine the trigger thresholds for scaling containers. One challenge with the existing approach is how to appropriately configure parameters. Taking too long to adjust container instances could risk performance degradation, leading to poor customer satisfaction. Immediately instantiating instances could also waste resources, increasing the operational cost. To set the parameters, developers need to quantify their workload with observation and analysis on running applications and hope to discover request patterns. This stage could take hours, days or months, depending on applications and usage. The situation

Sangpetch, A., Sangpetch, O., Juangmarisakul, N. and Warodom, S.
Thoth: Automatic Resource Management with Machine Learning for Container-based Cloud Platform.
DOI: 10.5220/0006254601030111
In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, pages 75-83
ISBN: 978-989-758-243-1

becomes even worse when workload is dynamic and agile, like web applications.

Web usage could change rapidly, depending on content and services offered on web applications. Ability to quickly respond to load changes is crucial to prevent degraded service quality. With existing cloud services, this ability heavily relies on how quickly and accurately developers can set parameters for the automatically-scaling service. It is difficult to maintain the desired service quality when tuning parameters manually. To address such challenge, we propose an automatic resource management engine, called Thoth. Thoth can automatically adjust container instances in order to maintain acceptable service quality without excessive provision. For Thoth's decision-making component, we evaluate two machine learning algorithms, namely 1) a reinforcement learning-based approach, called Q-Learning and 2) Neural Network, against our rule-based scaling method. The algorithms consider percentages of CPU and memory utilization, the number of requests, the number of replicas and service time. Service time is the time that containers take to serve requests. The resource utilization of the containers indicates how busy containers are. The number of requests and service time over time should imply the usage and trend. All selected metrics can suggest when container instances should be adjusted so that containers can gracefully handle load changes.

We implement Thoth based on Docker, which provides container-based infrastructure, and Kubernetes as a container orchestrator, similarly to Google App Engine. Unlike existing systems, we also integrate HAProxy to serve as a load balancer as well as a request monitoring endpoint. The target workload for Thoth is web applications. With Thoth, developers no longer need to manually set scaling parameters to meet desired performance. Thoth could help reduce the time that takes to respond load changes. From the experiments, Thoth with Q-Learning appropriately adjust the container instances while achieving the desired request response time and consuming the least resources.

This paper is organized as follows. Section 2 describes related work. We explain the architecture of the implemented system in Section 3. We evaluate the performance of the system in Section 4 and conclude in Section 5.

2 RELATED WORKS

Resource allocation problem in computing infrastructure, especially in a shared environment such as cloud computing, has been studied in previous works. One of important factors is scale-out latency which indicates the amount of time that takes to create a sufficient number of working instances. As discussed in ASAP (Jiang, 2011), scale-out latency on a virtual-machine based platform can significantly impact the ability to adjust to the demand. To reduce scale-out latency, we choose to build the infrastructure based on container technology since each container instance can be created or removed in a swift. A container instance is also its light-weighted and agile. Many previous works (Rao et al., 2009), (Dutreilh et al., 2011) focus on a virtual-machine based infrastructure, while our work situates on the container-based platform.

There are previous studies (Rao et al., 2009), (Dutreilh et al., 2011) that deployed data mining techniques to predict the demand pattern, similarly to our work. The derived patterns were used to predict the upcoming workload in order to prepare an adequate number of virtual machines (VMs). For examples, Rao et al (2009) and Dutreilh et al (2011) selected reinforcement learning algorithms to allocate virtual machine resources based on resource utilization. Jamshidi et al (2016) used self-learning fuzzy logic controller to scale the resource. Although our work also leverages the reinforcement learning algorithms, it considers the service time in addition to the resource utilization. Moreover, the studies (Rao et al., 2009), (Dutreilh et al., 2011) also propose the adjustments to the existing workflow deployed in the cloud platform.

Dawoud et al (2012) suggested a fine-grain resource adjustment in order to avoid the initial boot-up latency. Dynamic optimization method has been used in (Mao et al., 2010) to determine when to increase or decrease virtual machine instances with respect to budget, performance or energy constraints (Dougherty, 2012).

Although there are existing solutions, such as Kubernetes (2017), for providing container-based infrastructure, their resource management capabilities are not sufficient to manage web applications where workload is quite dynamic. Kubernetes is an open-source platform, it was originally created by Google and offered to Cloud Native Computing Foundation (CNCF). Kubernetes and Thoth are similar in a sense that both try to automatically adjust the system resources so that

they meet the incoming demand. However, the Autoscaling feature in Kubernetes version 1.1 (as of this writing) only supports the threshold-based ruling. In other words, users need to specify the target CPU utilization which will be compared with the average CPU utilization. If the target value is met, Kubernetes will automatically spawn one more container to help relieve the load. This current method is simple but still neglect other important factors that can contribute to the quality of web applications, such as network utilization and memory utilization. Deis (2017) builds open-source tools to manage applications on Kubernetes. Thus, Deis inherently shares the same scaling challenge found in Kubernetes. In contrast, Thoth takes other information, such as service time, request rate, memory utilization and CPU utilization, into account in order to make the scaling decision more accurately.

Flynn (2017) is an open-source platform-as-a-service based on container technology, but not utilized Kubernetes. Flynn helps developers in application deployment. Flynn's scale mechanism is still quite static. Users need to specify the number of target instances on a command line in order to scale an application.

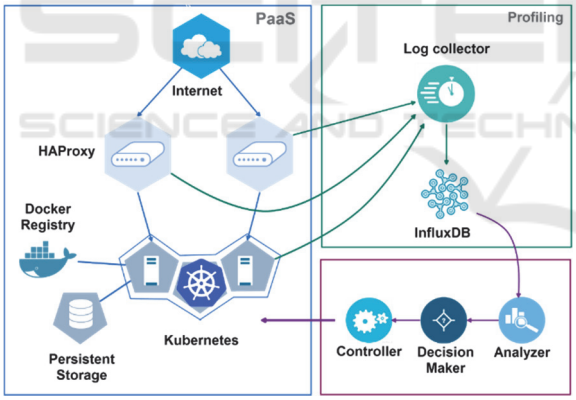

Figure 1: Thoth System Architecture Overview.

3 ARCHITECTURE

We design the Thoth's architecture, as shown in Figure 1. One of our key design rationale is that we try to leverage existing tools as much as possible and provide the necessary modifications to complete our resource management mechanism. The existing tools we utilize are, for examples, Kubernetes, HAProxy and Docker Engine. The system architecture contains three different modules: 1) Platform-as-a-Service or PaaS Module, 2) Profiling Module and 3) Scaling module.

3.1 PaaS Module

The PaaS module is responsible for managing the container and runtime infrastructure for applications. We leverage Docker Engine to execute packaged containers submitted by users. We use Kubernetes to manage a cluster of Docker-enabled servers and perform the container-replica creation and decommission. To run web applications, port-mapping is also one of the important tasks. We also use Kubernetes to oversee port-mapping. At the physical level, we connect each physical server to a shared persistent storage in order to store persistent data, such as database or application configurations.

We assume that each packaged container is stateless and the different instances of the packaged container will be able to serve each incoming request equivalently. This assumption is generally true in practice for recently-created web applications. One of the key design for web applications is reliability. In modern applications, reliability can be achieved through running multiple instances in various datacenters, rather than obtaining very expensive pieces of hardware. Hence, new web applications are normally designed to have many identical workers serve users. These workers are replaceable if any problem occurs, while users do not even notice the problem.

When a user sends a request to the web servers running on Thoth, each incoming request is going through HAProxy servers. HAProxy serves as a load balancer and reverse proxy for services running on our platform. We also use HAProxy to capture request statistics, such as the number of incoming request and the service time of each container. Combining HAProxy and Kubernetes, we have a flexible platform which can manage applications' resource scaling as well as collect the resource usage statistics of all tenants in the system. Our PaaS module is designed similarly to existing PaaS platform, such as Cloud Foundry (originally developed by VMware) and OpenShift by Red Hat.

3.2 Profiling Module

The main task of the profiling module is profiling performance and collecting the resource usage of applications running on our platform. In this module, we implement a centralized log collector to keep two important information: 1) the application-request statistics and the service time, reported from

HAProxy and 2) the CPU and memory utilization of each container and each physical servers. The CPU and memory utilization is monitored by Kubernetes. All statistics are captured as time series and stored into InfluxDB to be used for further analysis and monitoring the status of the system.

3.3 Scaling Module

The scaling module is our primary contribution which automatically analyses the stream of information collected by the profiling module, described in Sub-section 3.2. The goal of the analysis is to determine whether a web application is needed more resources in order to meet the acceptable quality of service. Then, it needs to identify an appropriate course of actions for the system in order to achieve such goal.

The collected time-series data is normalized and reformatted to fit the requirements of the selected machine-learning algorithms. In this paper, we choose to explore Q-Learning, a reinforcement learning-based approach, and Neural Network. The selected algorithms will be compared with our rule-based scaling method, which mimics the existing approaches. These algorithms will serve as a key component in our decision maker sub-module in order to decide whether or not to increase or decrease the instances of application containers. The final decision is forwarded to the controller which interacts with the Kubernetes to scale the container for the application.

We implement our decision maker component as a service module so that it can utilize different plug-ins for various algorithms including Q-learning and artificial neural network. In the future, different applications might be able to use different algorithms as appropriate.

Our scaling module is similar to the Autoscaling feature in Kubernetes version 1.1 (as of this writing) in a way that it tries to automatically adjust the resources in order to satisfy the performance requirement. However, the Autoscaling feature in Kubernetes version 1.1 only considers a threshold-based scaling based on target CPU utilization. In contrast, our scaling module includes various metrics including the application-request-related information and system utilization into the consideration. And our system can be easily extended to additional information in the future.

4 EVALUATION

The main objective of the evaluation is to investigate and assess the resource-control effectiveness of each selected algorithm. We consider how quickly each algorithm can adjust the number of computing containers to cope with changes in workload so that the request response time is maintained at the acceptable level. The best algorithm should be able to achieve desirable response time, while using the computing resources as little as possible. In the cloud regime, the amount of computing resources in use implies the cost of computing resources. Cloud providers may charge users or developers in term of the number of running containers in combination of the time in use. The more running containers developers have, the more expense would incur. In the experiments, all containers are instantiated from the same image, i.e. using the same programs.

We choose three algorithms, namely Q-Learning (Section 4.1.1), Neural Network (Section 4.1.2) and Rule-based algorithm which uses the parameters in Table 2. The rule-based algorithm represents a fixed solution where developers are able to come up with appropriate rules to handle existing workloads. The rules are defined by the scaling threshold for monitored Quality of Service metrics (such as service time and request/response rate), as well as resource utilization metrics (CPU, Memory.) However, if the access pattern changes, the rules may need to be revised and manually changed accordingly. In this case, we assume that developers are knowledgeable to be able to identify the appropriate rules for such workload. The configuration details of each algorithm are described in Section 4.1.

We conduct three experiments for each selected algorithm using the generated workload with the characteristics, elaborated in Section 4.2. While running the experiments, we use HAProxy to collect all performance data, i.e. response time, CPU and memory utilization. The experimental results are demonstrated and discussed in Section 4.3.

4.1 Experimental Settings

As mentioned earlier, we select three algorithms for the evaluation, namely Q-Learning, Neural Network and Rule-based algorithm. The goal of each algorithm is the same that is trying to maintain the response time below 500 milliseconds for each request by increasing the number of containers to keep up with the additional workload or removing the containers to reduce the excess resource. The configurations of each algorithm are described in the following.

4.1.1 Q-Learning

Q-Learning is a reinforcement-learning based approach for machine learning. As shown in Figure 2, Q-Learning system simulates a state diagram with variable link weight. After the system transition from an initial state, the reward of the current state is evaluated. Good state will yield high reward and reinforcement the associated transition. Bad state will yield lower reward and reflect back to ensure that the system is less likely to take the transition. As shown in Figure 2, state A is a current state. If no change in the resource allocation is needed, for example when the service quality is still in the acceptable range, it still stays in state A. If we need to add an additional container instance or a replica in order to handle higher demand, it will transition from state A to state B. If we want to remove a replica due to diminished workload, it can transition from state A to state C.

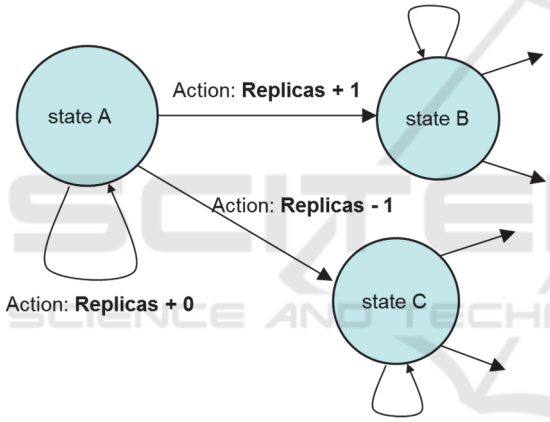

Figure 2: A state diagram showing states and transitions to multiple states generated while performing Q-Learning approach.

Typical Q-Learning implementation uses finite matrix to store its state-space information. However, in our settings, the state-space size could be exceedingly large. There are numerous scenarios in terms of the number of outstanding requests, request response time, CPU and memory utilization. To make the approach practical for actual deployment, we use a sparse learning matrix and store the list of indexed states instead. In practice, the number of explored state is limited and the time required to calculate the state transition is negligible.

In order to cope with the large number of available states, we have implemented a delayed reward calculation in order to properly capture the effect of change in the system. After transition to a new state, the actual reward for such transition is calculated and stored in the database. The reward calculation is presented in Figure 3. The calculation is comprised of the selected factors as follows:

1) The average CPU utilization.
2) The average memory utilization.
3) The service time that takes a server to respond to a request.
4) The ratio of the total number of requests to the maximum number of requests that can be served by a single instance. This ratio suggests the degree of workload quantity.
5) The number of existing replicas running an application.

Table 1 demonstrates how we configure each factor in order to make the calculation shown in Figure 3. After combining all factors or variables, the result determines the current state of an application. Each state is identified as a tuple of the variables <CPULOW, MEMLOW, RTIMELOW, REQFOLD, REPLICAS>. Each state is associated with a Q matrix value and the initial reward matrix value is set to zero.

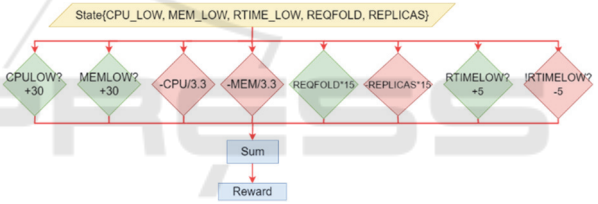

Figure 3: Variables used to identify the Q-Learning state.

Table 1: Q-Learning State Variables.

System Variables	Description	Possible Values
CPULOW	Average CPU utilization is less than 50 %	0, 1
MEMLOW	Average memory utilization is less than 50%	0, 1
RTIMELOW	Response time is less than average response time	0, 1
REQFOLD	The ratio of the total number of requests to the maxinum number of requests that can be served by a single instance	0, 1, 2, ….
REPLICAS	The number of replicas for the application	1, 2, 3, …

4.1.2 Artificial Neural Network (ANN)

Artificial Neural Network, simply Neural Network (NN) is usually utilized as pattern classification. As depicted in Figure 4, NN normally has multiple

inputs, weighed with appropriate values. Then, all weighted inputs are combined using a selected transfer function. The result will go through an activation function which defines the output of such inputs.

In this work, we deploy NN to recognize patterns of application characteristics that require scaling. Similar to Q-Learning, we use the variables identified in Table 1 to construct an artificial neural network with back-propagation. As shown in Figure 5, we represent the application state with a three-layer neural network with one hidden layer. There are 36 nodes in total. The input nodes conform to the same guidelines described in Table 1, similarly to Q-Learning. The task of the network is to classify the combination of the given inputs and then determine an associate action which can be increasing or decreasing the number of existing containers or replicas.

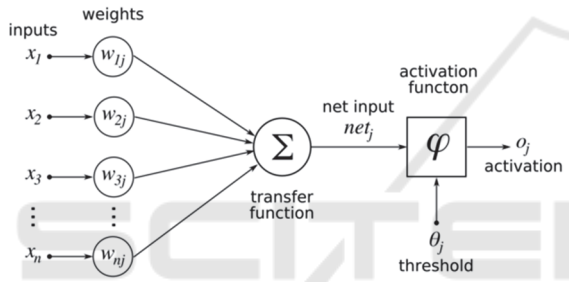

Figure 4: Neural network architecture.

4.1.3 Rule-based Algorithm

The rule-based algorithm (Rule) is created as a representation of a fixed resource control algorithm. Appropriate rules are influenced by experts with relevant knowledge and experiences. The objective of these rules is to achieve acceptable application performance while minimizing the resources utilized by applications. Two common application performance metrics are response time and throughput. The application response time is the amount time that takes from sending an application request until receiving the response. If we exclude the network round-trip time from the response time, it will be the time that the application servers take to complete the request, referred to as service time. In the rule-based approach, we give the service time higher priority than the throughput or request rate because the response time is an important factor to indicate interactivity level with users. We also use the CPU and memory utilization to estimate the resource usage of individual applications. However, the rules that we have constructed are not adaptive to load fluctuation. This approach is similar to what we have in current PaaS infrastructures.

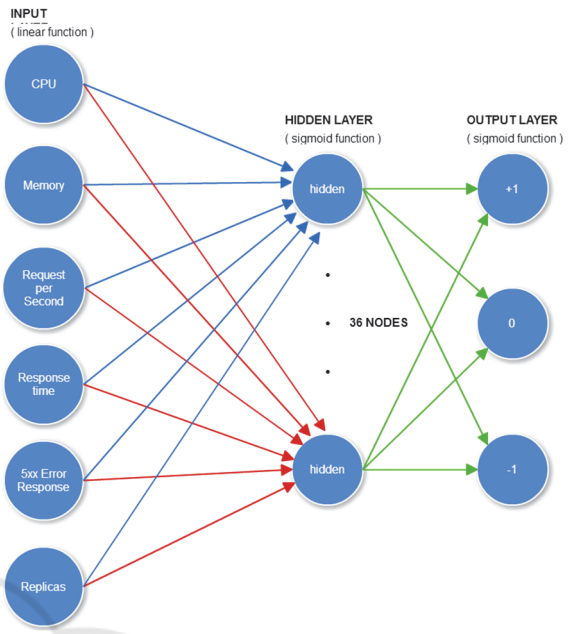

Figure 5: The artificial neural network that is used in our scaling algorithm.

To construct the rules, we select four important metrics, namely 1) service time, which determines the amount time that takes the containers to finish a request, 2) the request rate or the number of requests per second, 3) memory utilization percentage of running containers and 4) CPU utilization percentage of running containers. If the values of the metrics exceed the maximum, the system needs to create an additional container to handle the workload. However, if the number decreases below the minimum setting, the system needs to remove a container in order to release unnecessary resources.

Our rule-based algorithm will consider the parameters according to the order in Table 2 – that is the metrics are sorted by its priority. A higher priority metric will get considered before a lower priority metric. For example, the service time has higher priority than CPU utilization. If the service time exceeds 500 milliseconds but the number of requests per second is below 1,200, the system still needs to create another container. The reason is the service time has a higher priority than a request rate.

Table 2: Rule-based parameters and priorities.

Parameters	Minimum	Maximum
1) Service Time (Milliseconds)	150	500
2) Request Rate (Requests per Second)	1,200	2,500
3) Memory Utilization (%)	50	75
4) CPU Utilization (%)	50	75

4.2 Workloads

We use Locust (locust.io) to generate a synthetic workload to simulate a dynamic usage scenario. The test application is an eight-puzzle solver with randomly-generated settings. The time to solve each setting varies to mimic dynamic applications. The generated workload pattern is shown in Figure 6.

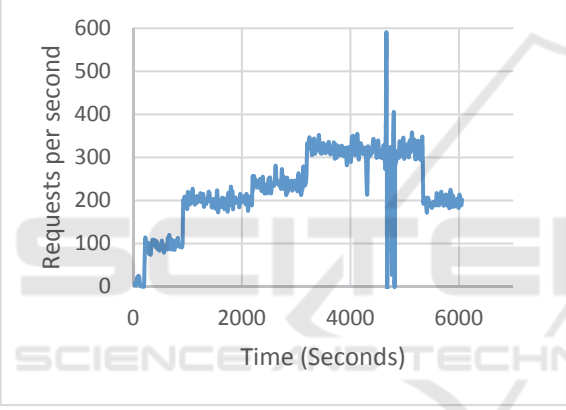

Figure 6: The average number of requests per second for the synthetic workload.

4.3 Experimental Results

From the experiments, both Neural Network algorithm (NN) and Q-Learning algorithm (QL) have very similar cumulative distribution functions as shown in Figure 7A and 7B, while the rule-based algorithm (Rule) has a little higher response time at higher percentile. When we consider the response time at the 90th percentile (Figure 7A), QL has the best response time, followed by NN and Rule – that is 6, 7 and 11 milliseconds respectively. Yet, the CDF graphs of all three algorithms have long tails. The 99th percentile response times of NN, QL and Rule (Figure 7B) are 5,800, 3,782, 3,647 milliseconds respectively. The 99th percentile response times are three orders of magnitude higher than the 90th percentile response times. The average response times of NN, QL and Rule are 169.76, 129.55, and 93.49 milliseconds respectively. NN has the highest average response time, which is 23.68% and 44.93% higher than the average response time of QL and Rule respectively. From the experimental results, NN has the worst capability to maintain the acceptable response time, while QL has a bit better overall response time than Rule.

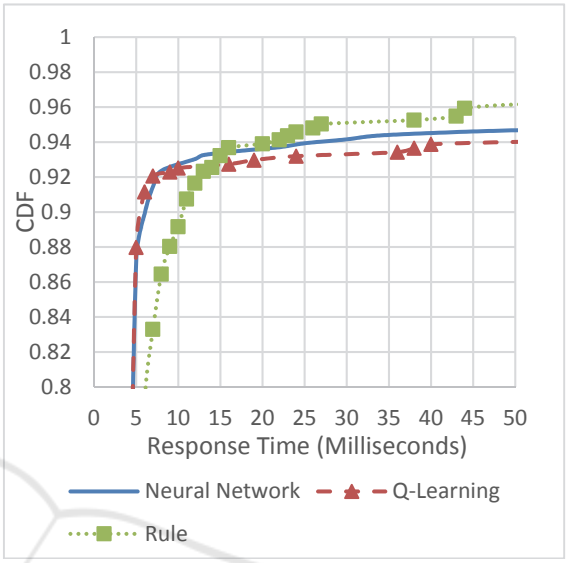

Figure 7A: The zoomed-in response time cumulative distribution function plot of NN, QL, and Rule (shown the 90th percentile).

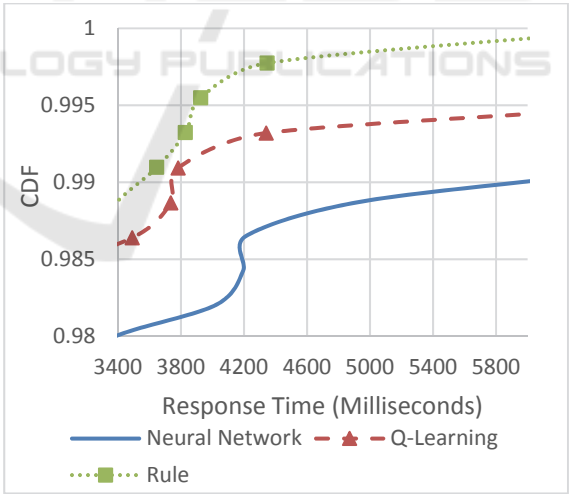

Figure 7B: The zoomed-in response time cumulative distribution function plot of NN, QL, and Rule (shown the 99th percentile).

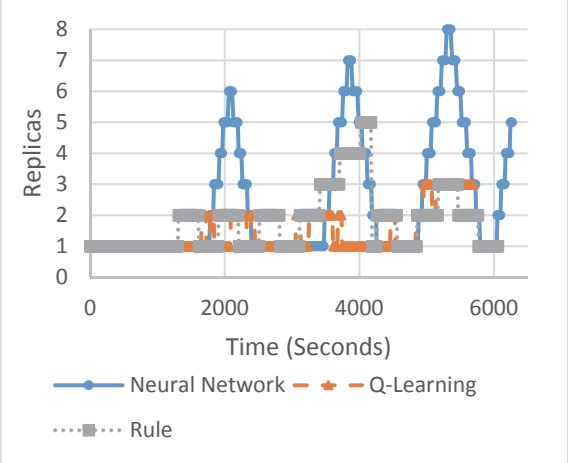

Figure 8: The number of replicas used with each algorithm.

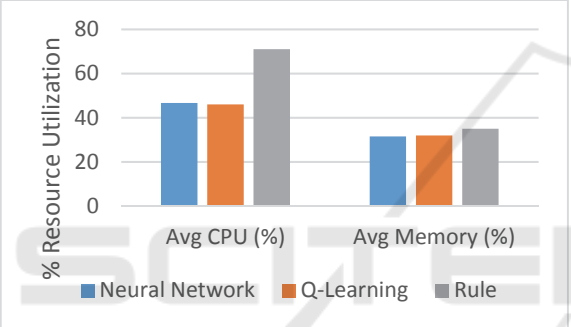

Figure 9: The average CPU and memory utilization for each algorithm.

Consider the number of containers or replicas in use for each experiment, as shown in Figure 8. The number of replicas imply the expense that would incur to developers. NN needs a lot more replicas than QL and Rule in order to handle the same amount of workload. Rule uses the containers a bit more than QL, especially during the period of 3,400 and 4,100 milliseconds. Therefore, on average NN, QL and Rule require 1.94, 1.38 and 1.68 replicas respectively. Hence, QL should yield the least computing cost to developers, followed by Rule and NN.

Consider the CPU and memory utilization of all in-use replicas, as shown in Figure 10 and 11. NN and QL have the similar average utilization. As shown in Figure 8 and 9, NN requires many more replicas to do the same amount of work as QL since the overall computing utilization of both NN and QL is about the same. Rule has 35.16% and 9.59% higher average CPU and memory utilization than the others respectively. As shown in Figure 8, Rule is

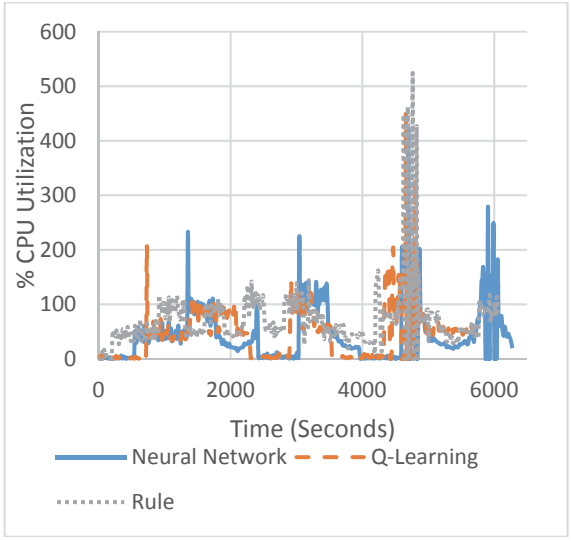

Figure 10: The CPU utilization of the application containers during the experiments with each algorithm.

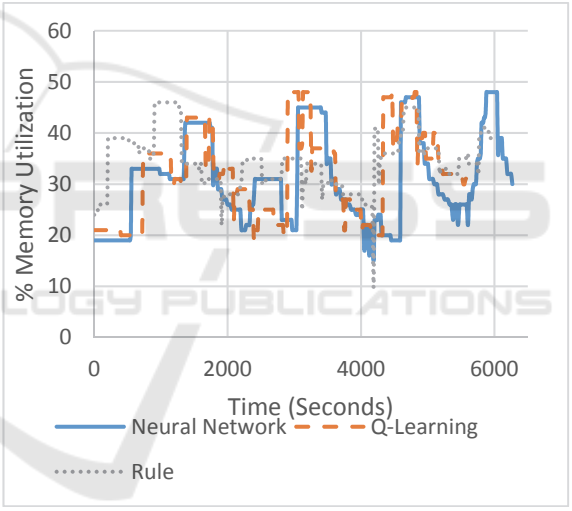

Figure 11: The memory utilization of the application containers during the experiments with each algorithm.

more sensitive to the workload changes than NN and QL. Thus, Rule tends to trigger an additional instance more quickly than QL, resulting in higher CPU and memory resource usage. During the period of 3,400 - 4,100 milliseconds, NN was unable to respond to the load changes in time, which could be the reason that NN has the worst overall response time.

The results from Figure 6-11 suggest that QL is a preferable choice over NN since QL yields better service (i.e. 34.79% better response time at 99th percentile) with less computing cost (i.e. 28.96% less containers in use). When we compare QL and Rule, QL has overall slightly-better service at the

90th percentile and incurs less computing cost (i.e. 21.94% less containers in use). As mentioned earlier, NN requires the most average number of containers in order to achieve the same amount of work. Additionally, the overall response time of NN seems to be the worst. This is because NN tends to have the slowest response to the load changes. In other words, NN cannot spawn a sufficient number of containers in time in order to cope with more workloads and when the load becomes smaller, NN does not reduce the in-use containers as quickly as it should.

5 CONCLUSION

We have proposed Thoth, an automated resource management system for container-based cloud platform, using different learning algorithms to auto-scale computing resources for web applications. The goal is to assist developers so that they do not need to manually adjust computing resources when workload changes to maintain acceptable level of service. Thoth utilized three algorithms as pluggable scaling modules, namely Neural Network, Q-Learning and Rule-based algorithm. These algorithms are studied and evaluated in a container-based platform as a service system. The experimental results suggest that QL can achieve the best quality of service with the least computing cost since QL can adapt to the load changes more quickly and appropriately than the others. Although Rule-based algorithm can yield a similar quality of service to QL, Rule requires 21.94% more computing resources, resulting in more expense. Additionally, the rule-based algorithm requires experts to manually calibrate the rules and it cannot be automatically adjusted to changes in the workload. NN performs the worst in terms of the amount of resources and service quality since it cannot adjust the load changes quickly enough. From the evaluation, QL could help developers maintain acceptable service quality as well as automatically adjust the proper amount of computing resources in order to minimize the computing resource expense.

REFERENCES

Dawoud, W., Takouna, I. and Meinel, C. (2012) Elastic virtual machine for fine-grained cloud resource provisioning. In: *Global Trends in Computing and Communication Systems*, Springer Berline Heidelberg, pp. 11-25.

Deis.io, (2017). *Deis builds powerful, open source tools that make it easy for teams to create and manage applications on Kubernetes*. [online] Available at: https://deis.io.

Dougherty, B., White, J. and Schmidt, D.C. (2012) Model-driven auto-scaling of green cloud computing infrastructure. In: *Future Generation Computer Systems*, 28, no 2., pp.371-378.

Dutreilh, X., Kirgizov, S., Melekhova O., Malenfant, J., Rivierre, N. and Truck, I. (2011) Using Reinforcement Learning for Autonomic Resource Allocation in Clouds: Toward a Fully Automated Workflow. In: *the 7th International Conference on Autonomic and Autonomous Systems*, Venice, Italy: ICAS, pp.67-74.

Flynn.io, (2017). *Throw away the duct tape. Say hello to Flynn*. [online] Avaiable at: https://flynn.io/

Jamshidi, P., Sharifloo, A., Pahl, C., Arabnejad, H., Metzger, A. and Estrada, G. (2016) Fuzzy Self-Learning Controllers for Elasticity Management in Dynamic Cloud Architectures. In: *the 12th International ACM SIGSOFT Conference on Quality of Software Architectures*, Venice: QoSA, pp. 70-79.

Jiang, J., Lu, J., Zhang, G. and Long, G. (2013) Optimal Cloud Resource Auto-Scaling for Web Applications. In: *the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Delft: CCGrid, pp. 58-65.

Jiang, Y., Perng, C.S., Li, T. and Chang, R. (2011) ASAP: A Self-Adaptive Prediction System for Instant Cloud Resource Demand Provisioning. In: *IEEE 11th International Conference on Data Mining*, Vancouver, BC, pp. 1104-1109.

Kubernetes.io, (2017). *Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications*. [online] Available at: https://kubernetes.io/

Mao, M., Li, J. and Humphrey, M. (2010) Cloud auto-scaling with deadline and budget constraints. In: *the 11th IEEE/ACM International Conference on Grid Computing, Brussels*, pp. 41-48.

Rao, J., Bu, X., Xu, C.Z., Wang, L., and Yin, G. (2009). VCONF: a reinforcement learning approach to virtual machine auto-configuration. In: *the 6th International Conference on Autonomic Computing*, Barcelona, Spain: ICAC, pp. 137-146.