Neural Computing and Applications
https://doi.org/10.1007/s00521-019-04507-z

ORIGINAL ARTICLE

Machine learning-based auto-scaling for containerized applications

Mahmoud Imdoukh[1] · Imtiaz Ahmad[1] · Mohammad Gh. Alfailakawi[1]

Received: 30 December 2018 / Accepted: 21 September 2019

Abstract
Containers are shaping the new era of cloud applications due to their key benefits such as lightweight, very quick to launch, consuming minimum resources to run an application which reduces cost, and can be easily and rapidly scaled up/down as per workload requirements. However, container-based cloud applications require sophisticated auto-scaling methods that automatically and in a timely manner provision and de-provision cloud resources without human intervention in response to dynamic fluctuations in workload. To address this challenge, in this paper, we propose a proactive machine learning-based approach to perform auto-scaling of Docker containers in response to dynamic workload changes at runtime. The proposed auto-scaler architecture follows the commonly abstracted four steps: monitor, analyze, plan, and execute the control loop. The monitor component continuously collects different types of data (HTTP request statistics, CPU, and memory utilization) that are needed during the analysis and planning phase to determine proper scaling actions. We employ in analysis phase a concise yet fast, adaptive, and accurate prediction model based on long short-term memory (LSTM) neural network to predict future HTTP workload to determine the number of containers needed to handle requests ahead of time to eliminate delays caused by starting or stopping running containers. Moreover, in the planning phase, the proposed gradually decreasing strategy avoids oscillations which happens when scaling operations are too frequent. Experimental results using realistic workload show that the prediction accuracy of LSTM model is as accurate as auto-regression integrated moving average model but offers 600 times prediction speedup. Moreover, as compared with artificial neural network model, LSTM model performs better in terms of auto-scaler metrics related to provisioning and elastic speedup. In addition, it was observed that when LSTM model is used, the predicted workload helped in using the minimum number of replicas to handle future workload. In the experiments, the use of GDS showed promising results in keeping the desired performance at reduced cost to handle cases with sudden workload increase/decrease.

Keywords Containerization · Auto-scaling · Proactive controller · Prediction · Neural network · Long short-term memory

1 Introduction

Cloud computing has emerged as the backbone of modern economy by offering subscription-based computing resources and services anytime, anywhere following a pay-as-you-go model [1]. A core ingredient in cloud computing is the virtualization technology that allows resources on a single computer to be sliced into multiple isolated virtual resources such as CPU, memory, and storage to enable customers to build computational nodes based on their software requirements and budget [2]. Traditionally, hypervisor-based virtualization has been used in public clouds as a way to create virtual machines (VMs) with specific resources and a guest operating system (OS) [1]. Recently, container-based virtualization has emerged as a lightweight alternative to VMs [3]. Compared to VMs, containers are lightweight, share the same OS of the host, require less memory space, are easy to maintain, and are portable. Further, containers are very quick to launch, consume less resources to run applications, ease the scale-in or scale-out resources as per requirements, and are mostly open source. In addition, it is easy-to-use version control mechanisms with containers [4]. Many container technologies are available such as LXC, Kubernetes, with Docker being the predominant [5].

✉ Mohammad Gh. Alfailakawi
alfailakawi.m@ku.edu.kw

[1] Department of Computer Engineering, College of Engineering and Petroleum, Kuwait University, Kuwait City, State of Kuwait

Published online: 08 October 2019

In recent years, containers have gained widespread popularity and are being used by many renowned organizations. Some of the well-known container-based services and development platforms are Google container engine [6], Amazon ECS (Elastic Container Service), and Azure container service (azure.micorsoft.com). In addition, micro-services are the latest trend gaining momentum in software service design, development, and delivery, wherein software parts became independent unit of development, versioning, deployment, and scaling [7]. Micro-services are being used by numerous organizations such as Amazon, Spotify, Netflix, and Twitter to deliver their software [8]. Containers are considered the standard to deploy micro-services to the cloud [7]. Containers are also being used for different other applications such as IoT, smart cars, fog computing, and service meshes [7, 9–11]. Therefore, the emerging container technology is going to change the way cloud computing platforms are designed and managed [12]. The widespread adoption of containers as base technology for large-scale systems opens many challenges in the area of runtime resource management.

One of the key features of cloud computing is elasticity which allows users to dynamically acquire and release the right amount of computing resources (VMs or container instances) according to their workload needs [13]. This process of dynamically acquiring or releasing resources is called auto-scaling. There are two major categories of auto-scaling: proactive and reactive [14]. In proactive scaling, an algorithm is used to predict the future from historical data, whereas in reactive scaling the algorithm reacts to workload or resource utilization according to a set of pre-defined rules and thresholds in real time [14]. The core principle behind these solutions is to scale resources according to workload demands and targeted latency. The choice of an auto-scaling method may significantly affect important quality parameters (such as response time and resource utilization) due to the fact that over-provisioning will lead to resource wastage and extra monetary cost, while under-provisioning causes performance degradation. Therefore, it is crucial to use an auto-scaling approach that automatically and in a timely manner adjusts provision and de-provision cloud resources without human intervention in response to dynamic fluctuations in workload at runtime. For hypervisor-based virtualization, many auto-scaling solutions exist [14–16]; however, container-based auto-scaling solutions are still an open issue that needs addressing [17, 18].

Container-based cloud applications require sophisticated auto-scaling methods in order to operate under different load conditions. However, designing and implementing an efficient general-purpose auto-scaler for Web applications in containers environment is a challenging task due to various factors such as dynamic workload characteristics, diverse resource requirements, and complex pricing models. As will be explained in Sect. 2, most of the existing auto-scalers use reactive threshold-based approaches since they are simple to implement [19–22]. However, choosing appropriate thresholds is difficult, especially when handling dynamic workloads. In proactive auto-scaling, many discussed applying statistical time series analysis for forecasting [23–28], whereas only few used machine learning algorithms to design a proactive controller [29, 30]. All these approaches are either slow in matching dynamic workload demands with adequate capacity or suffer from over-provision resources.

The recent developments in artificial intelligence (AI) and machine learning promise a bright future to build general-purpose auto-scalers [31]. In this paper, we propose a proactive machine learning-based approach to auto-scale containers in response to dynamic workload changes. Our proposed proactive controller is based on long short-term memory (LSTM) network [32]. LSTM, which is a special type of recurrent neural network (RNN) known to overcome the vanishing gradients drawback of RNN [33] and is concise, adaptive, and powerful. In this work, LSTM network learns from past scaling decisions and workload behavior to generate scaling decisions ahead of time. The contribution of this paper is summarized as follows: First, a high-level architecture of an auto-scaler is proposed which is designed specifically for container-based virtualization. Second, a novel time series prediction model using a special type of neural network, i.e., LSTM, is implemented. Third, gradually decreasing strategy (GDS) is proposed to deal with sudden increase and decrease in HTTP workload. Experimental results using realistic workload show that our proposed LSTM-based method achieves 600 times better performance than other state-of-the-art methods such as auto-regressive integrated moving average (ARIMA) statistical model in predicting future workload. As compared with artificial neural network (ANN) model, LSTM model performs better in terms of auto-scaler metrics related to provisioning and elastic speedup.

The rest of this paper is structured as follows. In Sect. 2, background and related work pertaining to the design and implementation of auto-scalers for containers are discussed. System architecture and the proposed auto-scaler architecture are discussed in Sects. 3 and 4, respectively. In Sect. 5, prediction model used is discussed, while the evaluation of the proposed auto-scaler is given in Sect. 6. Section 7 concludes the paper and highlights future directions.

2 Background and related works

In this section, we provide the background material and reported works related to containers auto-scaling.

2.1 Background

Auto-scaling problem is a classical automatic control problem, which demands a controller that dynamically tunes the types of resources and the amount of resources allocated to reach a certain goal [14, 15]. Specifically, it is commonly abstracted as a MAPE (monitor, analyze, plan, and execute) control loop [17, 18] where the control cycle continuously repeats itself over time. All phases of the MAPE loop impact significantly the auto-scaler efficiency. We briefly explain each phase and summarize key challenges facing auto-scaler designers in each phase.

Monitoring: It is the first phase in the loop where nodes key scaling indicators such as resource utilization, HTTP requests rate, and number of database transactions are gathered to make scaling decisions. These scaling indicators are produced and monitored at different levels of the system hierarchy from low-level metrics at the physical/container level (utilization of CPU, memory, etc.) or high-level metrics at the application level (request rate, average response time, etc.). Some auto-scalers use both high-level and low-level metrics. The challenge at this step is to decide which metrics are relevant to take decisions.

Analysis: All collected data are then processed in the analysis phase to either obtain current state or predict future workload demand in order to determine whether it is necessary to perform scaling actions based on the monitored information. Some of the key decisions made during the analysis phase include the followings:

- *Scaling time* When to start scaling the application is a critical decision. Based on scaling time, the approaches are grouped into reactive which depends on the current status of the application and the workload, and proactive which supports provisioning or de-provisioning of resources considering future needs of the application. Since proactive analysis technique performs prediction, it requires sophisticated forecasting techniques based on either queuing theory, statistical time series analysis, or machine learning.
- *Oscillation mitigation* How the auto-scaler reduces the chance of provision oscillation? One common solution adopted to mitigate oscillation is to conservatively wait a fixed amount of time between each scaling operation.
- *Adaptivity* Whether and how the auto-scaler adapts to changes of workload and application. Non-adaptive approaches make decisions purely based on the current input. Examples are the rule-based approaches employed in industry. They require the user to define a set of scaling-out/scaling-in conditions and actions offline. A self-adaptive mechanism is capable of autonomously tuning its settings and update its decisions according to new incoming information.

Planning: Once current or future workload is known, the planning phase estimates how many resources in total are to be provisioned or de-provisioned in the next scaling action in order to minimize financial cost and meet service-level agreements. Core decisions during the planning phase include:

- *Resource estimation* How the auto-scaler estimates the amount of resources needed to handle the workload. Resource estimation is the core of auto-scaling as it determines the efficiency of resource provisioning. Accurate resource estimation allows the auto-scaler to quickly converge to optimal resource provisioning. On the other hand, estimations errors either result in degraded performance or increased cost. The literature review presents several estimation models, from simple approaches such as rule-based and application profiling to more sophisticated methods such as analytic modeling and machine learning-based approaches.
- *Scaling methods* How the auto-scaler decides the method to provision resources and what combination of resources is provisioned to the application? Depending on the particular environment, scaling can be performed vertically, horizontally, or both. Vertical scaling deals with the adjustment of resources such as CPU, memory, or storage assigned to containers or VMs. On the other side, horizontal scaling deals with manipulating the number of VMs or container instances.

Execution: The actual scaling is done in this phase where the auto-scaler executes the scaling command through cloud provider's APIs.

2.2 Related work

In this subsection, we review recent work related to container auto-scaling.

In [23], Kan proposed DoCloud, a platform based on Docker that includes an auto-scaler to determine the number of container replicas needed for auto-scaling. The auto-scaler controller uses both reactive and proactive

controllers: specifically, a reactive controller for scaling out and proactive one for scaling in. The reactive controller requests resource usage (CPU and memory utilization) of worker nodes from a monitor sub-system and issues a scale-out command if usage exceeds a threshold value. The proactive controller uses auto-regressive moving average (ARMA) statistical model to predict request rate, and then based on the predicted request rate it estimates the number of containers needed to meet future workload demand. To avoid premature scale-in to cause oscillations, DoCloud only issues scale-in command if the number of containers predicted by the proactive model is all below current running containers for continuous k periods. Experimental results indicate that the DoCloud platform maintains stable resource utilization and auto-scales appropriately according to the workload.

Similar to Kan work [23], Li and Xia also used ARMA model as a container resource (CPU) utilization prediction algorithm on Docker orchestration tool [24]. However, results were not compared with any other approach. Ye et al. also used second-order ARMA model as a container resource (CPU) utilization prediction algorithm on Kubernetes orchestration tool [34]. The algorithm exploited both horizontal and vertical scaling methods. Comparison of experimental results with Kubernetes horizontal pod auto-scaler (HPA) [25], which is a typical reactive auto-scaler, showed that their approach adapts better to fluctuating workload. However, oscillation mitigation mechanism was missing from their proposed approach.

Ciptaningtyas et al. extended the work in [23] for a Docker container by using a different statistical model, namely auto-regressive integrated moving average (ARIMA), to predict the number of requests during planning phase [26]. The reason for selecting ARIMA model is its higher accuracy for short-term forecasting for statistically dependent time series. The authors used four different ARIMA (p,d,q) models where lag order value (p) takes on values 1, 2, 3, and 4 while fixing degree differencing (d) to 1 and order of moving average (q) to 0. Results show that ARIMA model with lag order of 4 achieved the lowest average error rate of 7.83% for incoming requests among all tested models.

Meng et al. also used ARIMA model as a container resource (CPU) utilization prediction algorithm and called it (CRUPA) [27]. CRUPA was integrated in a containers' cluster managed by Docker. The three parameters p, d, q of the ARIMA model were configured using Box–Jenkins method [35]. Experimental results showed that CRUPA outputs more accurate resource predictions compared to a threshold-based algorithm (TBA) as the average error of CRUPA with respect to actual resources needed is only 6.5% compared to TBA's 16.9%. However, the identification of ARIMA model parameters is a time-consuming process. Furthermore, CRUPA does not have oscillation mitigation mechanism as well.

Baresi et al. [36] presented an auto-scaling technique that uses an adaptive discrete-time feedback controller to enable a containerized application to dynamically scale resources, both horizontally and vertically. The algorithm selects from both VMs and containers in order to improve the performance of Web applications. Wu et al. used gray prediction model as a container resource (CPU) utilization prediction algorithm on Docker orchestration tool [37]. Gray prediction has the advantages that it needs few discrete historical data values to characterize an unknown system [38]. The algorithm exploited only horizontal scaling method and does not have oscillation mitigation mechanism.

The authors of [19] proposed CAUS, a controller that has two auto-scaling mechanisms and uses MAPE loop for managing elasticity of containerized micro-services. The first mechanism is a reactive approach that uses the maximum capacity of a container to handle workload. The second mechanism manages a buffer of additional containers to handle delay associated with starting of new containers or random unexpected workload spikes. The reflected number of containers for CAUS during experiments shows that it can efficiently deploy container replicas under different workload patterns. However, the rules for scaling are ad hoc in nature and can be tuned better with the presence of more knowledge regarding arrival patterns and workload characteristics.

Al-Dhuraibi et al. presented in [20] rules-based reactive approach to estimate the resources to scale Docker containers vertically. The auto-scaler adjusts memory and virtual CPU cores according to workloads. The upper and lower thresholds limits are set based on experimentation by trying several values and selecting the ones that lead to less response time. Additionally, the ratios to increase or decrease CPU and memory are fixed. For instance, when memory utilization is greater than the upper threshold it adds 256 Mb to the container and when it is below the lower threshold it decreases the memory by 128 Mb. A smaller scaling step is used for de-provisioning to avoid abruptly interrupting the functionality of the application. In addition, a live migration process is used when resizing is no longer possible on the host machine.

Recently, Taherizadeh et al. in [21] presented a reactive auto-scaling technique that uses multi-level, infrastructure (CPU and memory utilization) and application level (response time and throughput), data to dynamically adjust thresholds depending upon workload status at runtime in container-based environment. The results showed that their proposed method has better overall performance under changing workload than other rule-based auto-scaling methods which used fixed thresholds.

A rule-based framework for Docker container auto-scaling, especially for IoT devices, was presented in [22]. The proposed technique is integrated within datacenter operating system (DC/OS). The framework consists of a monitoring mechanism which collects CPU utilization of all containers. It includes a history recorder that stores all scaling decisions along with appropriate time stamps. Scaling decisions are issued once CPU usage exceeds certain threshold and the application is not in a cooldown time and ready for scaling. Experiments show that the auto-scaler performs well, in terms of elasticity, when there is a pattern in the workload as compared to the case when it is random. The study concluded that the cooldown time needed to avoid oscillation must be properly managed in order to balance system stability and platform elasticity.

Unlike the above-mentioned work, the prediction model in [29] is based on supervised machine learning. In [29], the authors built a single-hidden-layer artificial neural network (ANN) to predict task duration and resource utilization. In order to build a training dataset for the ANN, a crawler was implemented to collect file counts and total size of projects repositories from Github and task duration from Travis CI. Then, the ANN model was trained offline. The proposed ANN model was evaluated and compared with a simple linear regression prediction model and was found to provide a 20% reduction in prediction error. However, in their approach, ANN was trained offline and thus impractical for real applications.

In [30], the authors proposed *Thoth*, an automatic resource management that is composed of platform-as-a-service (PaaS), profiling, and scaling modules. The goal of Thoth is to auto-scale container replicas according to multiple metrics including resource utilization, request rate, response time, and number of replicas. The authors use three different algorithms: Q-learning algorithm which is a model-free reinforcement learning approach [39], a basic artificial neural network (NN), and rule-based algorithm to make scaling decisions. Experiments indicate that Q-learning algorithm outperforms rule-based as well as neural network in saving resources by almost 22% and 29%, respectively.

The authors of [28] presented a container traffic analyzer (COTA), called least traffic load balancing (LTLB) algorithm, which aims to balance network traffic across service instances and improve their performance. It uses container's networking data, bandwidth, and traffic to select the optimal number of containers that can serve users. In addition, it has an auto-scaler that uses a moving average (MA) method-based proactive controller to predict network traffic. Experiments on Docker containers show that LTLB leads to better network utilization compared to least connection (LC) algorithm. Compared to threshold-based controller, it reduces both retention time and cost of resource operation.

A comparison of all aforementioned auto-scaling techniques for containers is given in Table 1. Most of these techniques use rule-based reactive models directly or indirectly. In these approaches, the scaling process is triggered according to a predefined set of rules. Even though technically the specification of such rules is easy to implement, it is difficult to define the right boundaries in order to meet the tailored requirements for the application. The second popular group of techniques use statistical models (ARMA, ARIMA, etc.), which are relatively slower in matching dynamic workload demands with adequate capacity and requires more configuration. Recent advances in machine learning have achieved great success in a number of application domains ranging from computer vision to natural language understanding [31] and have led to a new era of machine learning techniques. However, there is a limited use of machine learning techniques in the context of containers auto-scaling. Therefore, inspired by the huge success of machine learning in recent years, we propose a proactive LSTM model-based approach to auto-scale containers in response to dynamic workload changes by exploring the fertile field of machine learning.

3 System architecture

Container orchestration frameworks provide support for deploying and managing a multi-tiered distributed application as a set of containers on a cluster of nodes [40]. One of the most prominent orchestration frameworks is the Docker Swarm. A high-level architecture of a simplified containerized system deployed on a cluster of nodes is shown in Fig. 1. All experiments that will be discussed in Sect. 5 assume such architecture. The system is composed of several connected nodes with different roles orchestrated by the Docker Swarm. Regardless of its role, each node runs a Docker container engine to be able to host containers. In recent years, there has been a strong industry adoption of Docker containers due to its easy-to-use approach that allows to package an application with all of its dependencies into one standardized unit for software deployment.

The first node type, the manager, is dedicated to orchestrate both the cluster and deployed containers by the Docker Swarm. The manager node is responsible to maintain the state of the cluster by continuously checking the nodes or nodes that request to join the cluster. It also handles job scheduling such as deploying/removing/starting/stopping containers as well as changing containers' replicas. Besides the two main tasks performed by manager node, it also hosts a containerized auto-scaler that adds

Table 1 Characteristics of auto-scaling approaches for containers

Work	Platform	Monitored indicators	Analysis phase			Planning phase	
			Timing	Adaptivity	Oscillation mitigation	Estimation technique	Scaling method
[23]	Docker	CPU, Mem.	Hybrid	✓	✓	ARMA	Hor.
[24]	Docker	Req. Rate, CPU, Mem.	Hybrid	✓	✓	ARMA	Hor.
[25]	Kubernetes	CPU	Proactive	✓	×	ARMA	Hor. & Ver.
[26]	Docker	–	Hybrid	✓	×	ARIMA	Hor.
[27]	Docker	CPU	Proactive	✓	×	ARIMA	Hor.
[36]	Docker	CPU, Mem., Appl	Proactive	✓	✓	Control theory	Hor. & Ver.
[37]	Docker	CPU	Proactive	✓	×	Gray prediction	Hor.
[19]	–	Workload Intensity	Reactive	✓	✓	Rule based	Hor.
[20]	Docker	CPU, Mem.	Reactive	✓	✓	Rule based	Ver.
[21]	Kubernetes	CPU, Mem., Appl.	Proactive	✓	×	Rule based	Hor.
[22]	Docker	CPU, Mem.	Reactive	✓	✓	Rule based	Hor.
[29]	–	Task duration	Proactive	✓	×	ANN	Hor.
[30]	Docker, Kubernetes	Req., CPU, Mem. Service Time	Proactive	✓	×	Q-lear., NN, rule based	Hor.
[28]	Docker	Network traffic/bandwidth	Proactive	✓	×	MA	Hor.

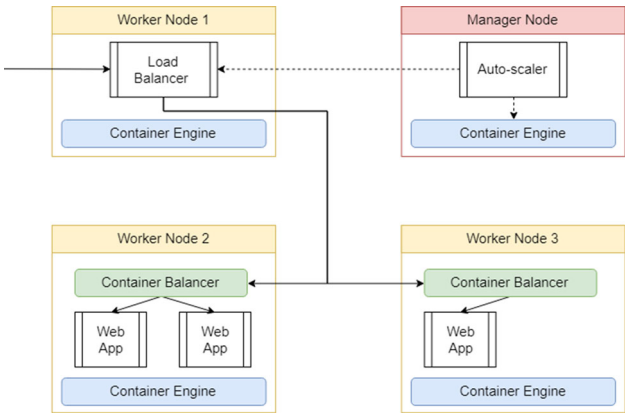

Fig. 1 System architecture

elasticity to the system by adjusting the number of replicas according to predicted future workload. The details of the auto-scaler architecture are described in Sect. 4.

The second type of node is the worker node. Its job is only to host running Docker containers. In this architecture, one worker node will only host a containerized load balancer; therefore, it will act as a system gateway that receives all incoming requests. The load balancer will, then, distribute the workload evenly across other worker nodes (Docker containers that have Web applications installed) and then send back the responses to users. Obviously, load balancer for Web application is very important and thus requires the load balancer to have a very good performance and robustness. HAProxy is an open-source high-performance, high-availability, and reliable proxy server used for TCP/HTTP load balancing [41]. Over the years, it has become the de facto standard open-source load balancer and is now shipped with most mainstream Linux distributions and is often deployed by default in cloud platforms. HAProxy can also be used to capture request statistics such as the number of incoming requests.

Worker nodes 2 and 3 host replicas of containerized Web application. A node may host more than one container and thus need for load balancer to distribute load evenly on worker node. Further, since worker nodes 2 and 3 each have a container load balancer, the workload will be further distributed across container replicas running on that node. The containerized application receives the request from the load balancer and performs some processing that requires resources such as CPU and memory. One of our key design rationales is that we try to leverage existing tools (HAProxy, Docker Engine, Docker Swarm) as much as possible and provide the necessary modifications to complete our auto-scaler approach.

4 Auto-scaler architecture

The proposed auto-scaler follows the commonly abstracted MAPE loop discussed earlier in Sect. 2 [17, 18]; however, implementation details of the various steps are different when compared to earlier approaches. Our primary contributions are in the analysis and planning steps which can be considered as the core of the auto-scaler. Figure 2 shows the architecture of the auto-scaler showing the four main

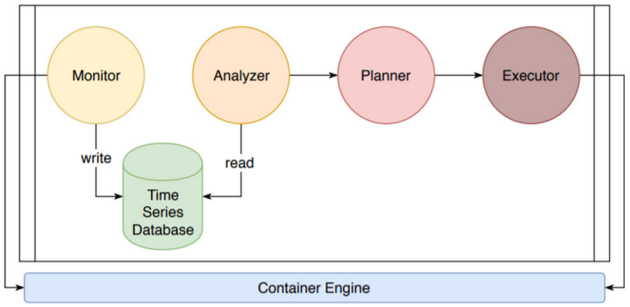

Fig. 2 Auto-scaler architecture

steps in addition to a time series database. The database stores different data items utilized by the monitor and analyzer units. Next, we briefly describe the details of each unit in the auto-scaler.

4.1 Monitor

The monitor continuously collects different types of data needed during analysis and planning phases to determine appropriate scaling actions. In the presented architecture, data with various characteristics are collected from two different sources: (1) networking data such as incoming HTTP request statistics (requests per second) from load balancer, (2) CPU and memory utilization for all containers running on all nodes from the manager node (using Docker remote RESTful API).

4.2 Time series database

The Time series database is a special type of database that is optimized to store data with an associated time stamp. Adding a time series database to the auto-scaler allows to properly maintain all collected data as historical record. These records are used when training the prediction model to enhance its prediction accuracy. This approach leads to building an application-specific prediction model since each application has its own data trends and patterns generated when serving users requests.

4.3 Analyzer

The analyzer periodically and continuously retrieves the latest collected data with predefined window size w from the database. The retrieved data are preprocessed by performing operations such as normalization to transform it to a form suitable to the prediction model used. The prediction model uses time series data d_0, d_1, ... , d_w to predict next data in the sequence d_{w+1}. In our implementation, LSTM neural network is used to predict the future HTTP workload and will be further discussed in Sect. 5. The importance of the analysis phase in MAPE loop is that it assists in forecasting future workloads. Consequently, the auto-scaler can increase or decrease container replicas before the occurrence of actual workload to eliminate delays caused by starting or stopping running containers.

4.4 Planner

Instead of using the current workload, the planner uses analyzer output, i.e., predicted workload, to decide the number of replicas (i.e., increase or decrease). It is designed specifically to horizontally scale up or down container replicas to meet future workload. The decision, i.e., estimated replicas ($R_{\text{estimated}}$), is determined using (1).

$$R_{\text{estimated}} = \left\lceil \frac{W^{\text{total}}_{\text{predicted}}}{W^{\text{max}}_{\text{container}}} \right\rceil \tag{1}$$

where $W^{\text{total}}_{\text{predicted}}$ is total predicted incoming workload to the system, whereas $W^{\text{max}}_{\text{container}}$ is the maximum workload that can be handled by each container per second. The value $W^{\text{total}}_{\text{predicted}}$ is obtained from the analyzer, while the value of $W^{\text{max}}_{\text{container}}$ can be determined from using stress tests during development stage.

Algorithm 1 summarizes the planner functionality. First, the algorithm computes the value $R_{\text{estimated}}$ and then compares it with the number of current replicas (R_{current}) to make a scaling decision (Line 1). If there is no change in the number of containers, then no scaling command is issued (Lines 2–3). If the estimated replicas are greater than R_{current}, the planner will issue a scale-up command to the executor (Lines 4–6). The cooldown timer (CDT) is restarted after any scale-up command, and it is used to allow the system to maintain a stable number of container replicas to handle sudden change in workload. However, if $R_{\text{estimated}}$ are less than the current replicas, a scale-down command will not be passed to the executor unless CDT has timed out (Lines 7–12). Even when sending a scale-down command, the planner will follow a gradually decreasing strategy (GDS) that aims to scale down replicas to a number that is less than estimated replicas by scaling down ratio (SDR) (Lines 9–10). Since the planner follows GDS, only few number of containers will be stopped at a time until the number of replicas reaches the predefined minimum number of replicas R_{min}. The main goal of GDS to avoid oscillation happens when scaling operations happen frequently. In our experiments, CDT is set to 10 s, SDR is set to 0.40 (40%) and $R_{\text{min}} = 5$ replicas.

Algorithm 1 Planner Algorithm

Input: R_{min}, $R_{current}$, $W^{total}_{predicted}$, $W^{max}_{container}$, CDT, SDR
Output: Scaling commands

$R_{estimated} = \left\lceil W^{total}_{predicted} / W^{max}_{container} \right\rceil$
if $R_{estimated} == R_{current}$ **then**
  sendComand($NONE$)
else if $R_{estimated} > R_{current}$ **then**
  sendComand($SCALE\_UP, R_{estimated}$)
  restart CDT
else if CDT timed out **then**
  restart CDT
  $R_{estimated} = \lfloor (R_{current} - R_{estimated}) * (1 - SDR) \rfloor$
  $R_{estimated} = max(R_{current} - R_{estimated}, R_{min})$
  sendCommand($SCALE\_DOWN, R_{estimated}$)
else
  sendComand($NONE$)
end if

4.5 Executor

The executor which is the last phase of MAPE loop is responsible for receiving commands from the planner and changing the actual number of container replicas. It communicates with the container engine (Docker daemon) through its API to execute scaling commands.

5 Prediction model

In this section, the proposed artificial neural network prediction model is discussed.

5.1 LSTM

Long short-term memory (LSTM), a successor for recurrent neural network (RNN), is a special type of recurrent neural network (RNN) known to overcome the vanishing gradients drawback of RNN [33]. This capability makes LSTM very suitable to predict the next sequence in a time series data such as workload over time. The LSTM network is built up with a series of LSTM units/cells connected to each other. The main part of LSTM is the cell state, shown as red dashed line in Fig. 3, which stores data coming from gates. The LSTM unit is composed of three gates: forget, input, and output. The purpose of any gate is to control the amount of data that pass through it. Each gate consists of a sigmoid function and a multiplication operation. Sigmoid function outputs a value between 0 and 1 as determined by the concatenation of $h(t-1)$ and $x(t)$. The sigmoid result is then multiplied by the input to generate the gate's result. For example, if the result of sigmoid is 1, the gate's result will be identical to its input since it is multiplied by 1. For each input vector to LSTM network, the unit processes the input data as follows:

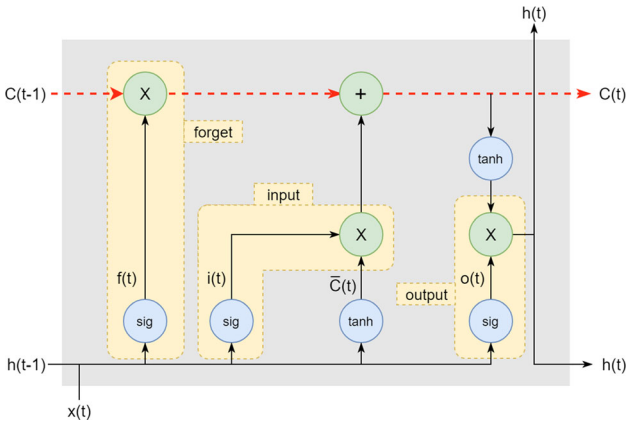

Fig. 3 LSTM architecture

i. Input vector $x(t)$ is concatenated with the previous hidden state vector $h(t-1)$ to produce a new vector that will be the input for the three gates in addition to a tanh function.
ii. The forget gate controls the amount of previous cell state to be maintained within the LSTM unit according to:

$$f(t) = \text{sig}(W_f * [h(t-1), x(t)] + b_f) \quad (2)$$

where W and b are the weight and bias, respectively.
iii. A candidate value for the current cell state $\overline{C}(t)$ is computed by:

$$\overline{C}(t) = \tanh(W_C * [h(t-1), x(t)] + b_C) \quad (3)$$

iv. The input gate decides the amount of $\overline{C}(t)$ that will be added to the current cell state by multiplying it to $i(t)$, which is determined using:

$$i(t) = \text{sig}(W_i * [h(t-1), x(t)] + b_i) \quad (4)$$

v. The final current cell state $C(t)$ is calculated as:

$$C(t) = f(t) * C(t-1) + i(t) * \overline{C}(t) \quad (5)$$

vi. The output gate controls the amount $\overline{C}(t)$ that will be passed to the next cell using (6), whereas the final hidden state $h(t)$ is calculated using (7).

$$o(t) = \text{sig}(W_o * [h(t-1), x(t)] + b_o) \quad (6)$$

$$h(t) = o(t) * \tanh(C(t)) \quad (7)$$

5.2 Neural network architecture

The LSTM ANN is used to build up a neural network to predict future incoming workload. The architecture of prediction neural network is shown in Fig. 4. It includes 10 neural cells input layer, 30 LSTM units hidden layer, and 1 neural cell output layer. The input layer will receive

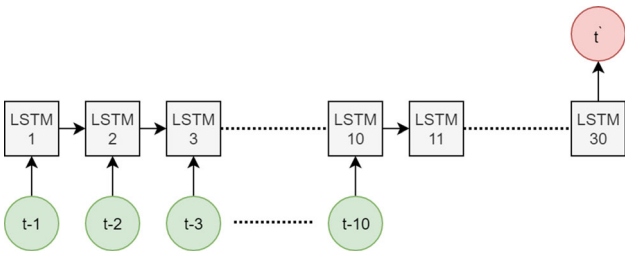

Fig. 4 Neural network architecture

previous workloads that occurred in the last 10 time steps to predict the workload that will happen in the next time step. The simple architecture of LSTM prediction model helps in generalizing the training data and avoid overfitting so that it can be used with different input datasets.

5.3 Single-step and multi-step predictions

The proposed neural network architecture is capable in predicting the next value given a time series of 10 past observations as shown in Fig. 5. However, multi-step prediction, that is predicting multiple future values given a past time- series, is more appropriate for an auto-scaler for better deciding the scaling decisions. There are two approaches for the multi-step prediction: direct and recursive. The direct strategy works by training the model to predict a value that will occur after multiple steps as shown in Fig. 6. The recursive strategy uses historical data to predict multiple next values; thereafter, the predicted next value is recursively used by adding it to historical data to predict the next value. Figure 7 shows an example of predicting four different next values. First, the historical data are used to predict p_0, then, p_0 is added to the history to predict the value of p_1, similarly p_1 is added to the history to predict p_2 and so on until p_3 is predicted. The main drawback of the recursive strategy is that the prediction error gets accumulated with each step. For the purpose of illustration, Figs. 5, 6, and 7 use historical data of size 5, to predict the value after four steps in case of multi-steps.

6 Experiments and evaluation

In this section, we evaluate the system architecture proposed in Sect. 3 in two ways. First, the proposed LSTM prediction model will be evaluated and compared with

Fig. 5 Single-step prediction

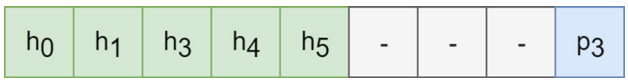

Fig. 6 Direct multi-step prediction

Fig. 7 Recursive multi-step prediction

ANN [29] and auto-regressive integrated moving average (ARIMA) models [26, 27] in terms of prediction accuracy and speed. Second, the complete system architecture will be analyzed in a simulated and real environments to study the responsiveness of the proposed auto-scaler.

6.1 Dataset

Worldcup98 represents the http requests logs of around 1.3 billion total requests of FIFA World Cup Web Site in 1998 between April 30 and July 26 [42]. Each log includes Unix time stamp besides other data such as client ID, method, and status. The dataset has been intensively used in the cloud computing literature to evaluate auto-scalers such as the works in [43–46] and many more.

The first step in preprocessing the dataset was to aggregate all the logs that occurred in the same second, but different millisecond, into one accumulative record. So, the overall dataset is converted such that each record represents the total workload, HTTP requests, per second. The dataset, is then, split into two parts that will be referenced in the remaining section as S1 and S2 as shown in Table 2. S1 represents around 70% of dataset with 5244199 records from 1998-04-30 21:30:00 to 1998-06-30 19:30:59 and the S2 represents about 30% of the dataset with 2255274 records from 1998-06-30 19:31:00 to 1998-07-26 21:59:00.

Table 2 Worldcup98 dataset

Ref.	Dataset	Size	Usage
S1	Workload per second	5,244,199	Evaluation
S2		2,255,274	Evaluation
M1	Workload per minute	87,710	Training
M2		37,590	Evaluation

However, throughout the experiments, it was noticed that S1 and S2 can be simplified and yet maintaining the main pattern in the dataset. The simplification not only helped in speeding up the training process of the prediction model, but also it helped in making the model better generalize and reduced the overfitting. The dataset is simplified by taking the maximum workload for each 60 seconds in the same minute. As a result, the dataset now represents the maximum workload per minute which are referred to as M1 and M2 as shown in Table 2. The size of both S1 and S2 is reduced by 60% as M1 and M2 include 87,710 and 37,590 records, respectively. Figure 9 shows the simplified dataset which maintains the same patterns in the original dataset shown in Fig. 8.

All the prediction models, that are represented, in the following section were trained and evaluated by using M1 dataset after scaling its data between -1 and 1. As for the purpose of validating the model, 10% of M1 was used in the validation process. The remaining datasets, S1, S2, and M2, were used for evaluation as summarized in Table 2.

6.2 Experiment description

Two experiments were conducted to evaluate the proposed prediction model for auto-scaling containers. The purpose of the first experiment is to train and evaluate the prediction model that will be used in the auto-scaler to predict the next coming workload. All prediction models are trained using M1 dataset. The LSTM prediction model explained in Sect. 5 is compared with ANN and ARIMA prediction models. Second, the system architecture and auto-scaler will be experimented in a real and simulated environment to study the responsiveness of the proposed container auto-scaler.

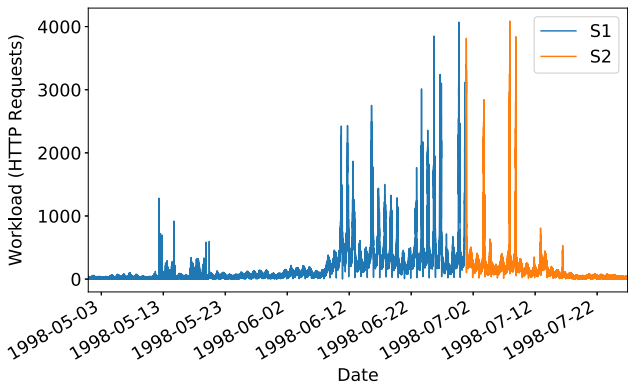

Fig. 8 Worldcup98 dataset represented as workload per second

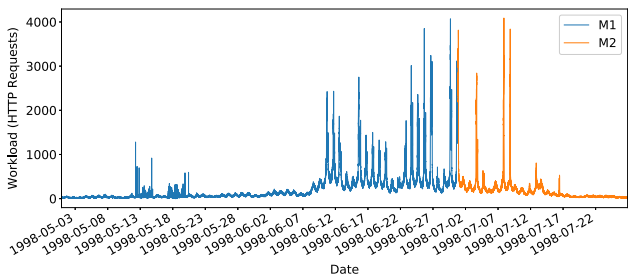

Fig. 9 Simplified Worldcup98 dataset represented as max workload per minute

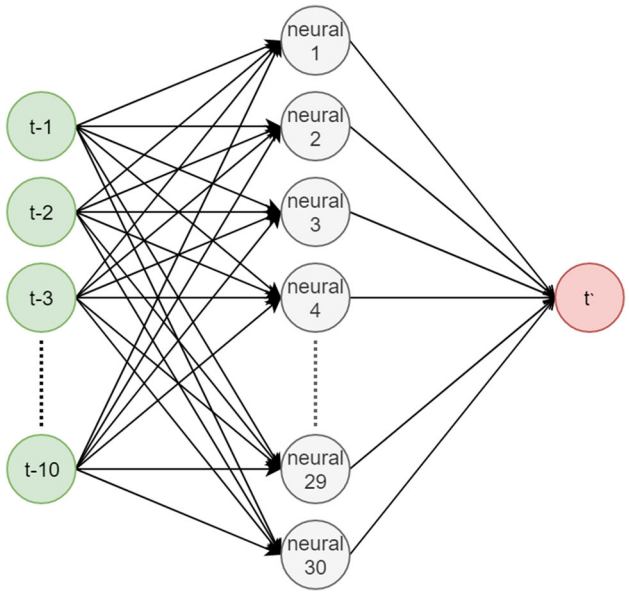

Fig. 10 Architecture of ANN prediction model

The architecture of ANN model includes input layer of size 10 neural cells, one hidden layer of size 30 neural cells, and output layer of size 1 neural cell as shown in Fig. 10 and described in Table 3. Because the model is a nonlinear regression model, Relu activation function is used in the hidden layer. The architecture of LSTM is explained in Sect. 5.2. The complete LSTM configurations are shown in Table 4. Both models use mean square error (MSE) as loss function for the training. To avoid

Table 3 ANN model configurations

Input size	10
Output size	1
Number of hidden layers	1
Number of hidden neural cells	30
Activation function	Relu
Loss function	MSE
Optimizer	adam
Batch size	64
Number of epochs	50
Early stopping patience	2

Table 4 LSTM model configurations

Input size	10
Output size	1
Number of LSTM layers	1
Number of LSTM units	30
Kernel initializer	Lecun uniform
Loss function	MSE
Optimizer	adam
Batch size	64
Number of epochs	50
Early stopping patience	2

overfitting, early stopping is used to stop the training process when the validation loss is no longer get reduced.

There are two types of LSTM and ANN models that were trained using M1 dataset to be used in the experiments. The first type: LSTM (1 step) and ANN (1 step) are models that were trained to predict the max workload in the next minute given the max workloads in the last 10 minutes. The second type: LSTM (5 steps) and ANN (5 steps) are models that were trained, using direct multi-step strategy, to predict the max workload that will occur after 5 minutes. All prediction models were implemented using Keras library powered by TensorFlow backend. The models were trained on tensor processing unit (TPU) provided by Colab. ARIMA model was also implemented in Python using StatsModel library. The ARIMA model was configured by setting lag order (p) to 4, degree differencing (d) to 1 and the order of moving average (q) to 0 as suggested in [26].

In the second experiment, the simulation environment is implemented in Python to simulate the system architecture and auto-scaler MAPE loop. The simulation starts running the analyzer which retrieves and scales a subset from worldcup98 dataset and send it to the planner. The planner runs Algorithm 1 to decide the needed number of containers to serve the workload. The planner sends to the executor the new number of containers. The executor, then, either does nothing, increases or decreases the number of containers. A delay is added when containers are increased to simulate the start-up time of containers. Since there are three worker nodes in the system, increasing three containers is done in parallel as each one will start in a different node.

The real environment is implemented by creating four virtual nodes where each node is running Docker container engine. Docker Swarm is used to orchestrate the cluster which hosts the nodes where one node acts as a manager, while the other three are configured as worker nodes. The manager node runs the auto-scaler container which is composed of two parts: a MAPE loop and a customized time series database. In the MAPE loop, the monitor will continuously request incoming HTTP workload from HAProxy and store it in the database. The analyzer retrieved HTTP workload for the last 5 s from the database to predict the workload for the next second. Additionally, if recursive multi-step prediction is enabled, it will forecast workload that will happen over the next few seconds. The planner will execute the scale planning algorithm. Finally, the executor communicates with Docker engine through Docker-Py API to increase or decrease the number of container replicas. One of the worker nodes is running an HAProxy load balancer container to balance the workload across the other two worker nodes and so it exposes a port to be accessed by end users. The two worker nodes are running Web application containers which process incoming requests. To generate HTTP workload on HAProxy, Apache JMeter is used.

6.3 Evaluation metrics

The prediction model will be evaluated according to mean square error (MSE) shown in Eq. 8 and coefficient of determination (R^2) shown in Eq. 9 where y, $\hat{y}$ and $\bar{y}$ represents actual value, predicted value, and mean values of y, respectively. Additionally, in some cases mean absolute error (MAE) and root-mean-square error (RMSE) are considered for evaluation. Prediction speed will be studied by measuring the average time, of 30 tries, that each model takes for prediction.

$$\mathrm{MSE}(y,\hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2 \tag{8}$$

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y_i})^2}{\sum_{i=1}^{n}(y_i - \bar{y_i})^2} \tag{9}$$

$$\mathrm{MAE}(y,\hat{y}) = \frac{1}{n_{\mathrm{samples}}}\sum_{i=0}^{n_{\mathrm{samples}}-1}|y_i - \hat{y_i}| \tag{10}$$

$$\mathrm{RMSE}(y,\hat{y}) = \sqrt{\frac{1}{n_{\mathrm{samples}}}\sum_{i=0}^{n_{\mathrm{samples}}-1}(y_i - \hat{y_i})^2} \tag{11}$$

To evaluate the auto-scaler, recently proposed metrics related to provisioning and elastic speedup are used [47, 48]. Under-provisioning metric, θ_U, identifies the number of containers that are needed to reach to the demanded number of containers as shown in Eq. 12. Over-provisioning metric, θ_O, identifies the number of supplied containers that exceeds the demanded number as shown in Eq. 13. Under-provisioning and over-provisioning time shares, T_U and T_O, represent the time interval when the auto-scaler is under-provisioned and over-provisioned as

shown in Eqs. 14 and 15, respectively. Elasticity speedup (ϵ_n) shown in Eq. 16 compares the performance for the cases when auto-scaling is used or not where subscript a refers to the case when used and n otherwise. This metric measures the gain of using auto-scaler to the other case when the value is larger than 1 and loss of performance when it is lower than 1.

$$\theta_U[\%] = \frac{100}{T}\sum_{t=1}^{T}\frac{\max(\text{demand}(t) - \text{supply}(t), 0)}{\text{demand}(t)}\Delta t \quad (12)$$

$$\theta_O[\%] = \frac{100}{T}\sum_{t=1}^{T}\frac{\max(\text{supply}(t) - \text{demand}(t), 0)}{\text{demand}(t)}\Delta t \quad (13)$$

$$T_U[\%] = \frac{100}{T}\sum_{t=1}^{T}\max(\text{sgn}(\text{demand}(t) - \text{supply}(t)), 0)\Delta t \quad (14)$$

$$T_O[\%] = \frac{100}{T}\sum_{t=1}^{T}\max(\text{sgn}(\text{supply}(t) - \text{demand}(t)), 0)\Delta t \quad (15)$$

$$\epsilon_n = \left(\frac{\theta_{U,n}}{\theta_{U,a}}\cdot\frac{\theta_{O,n}}{\theta_{O,a}}\cdot\frac{T_{U,n}}{T_{U,a}}\cdot\frac{T_{O,n}}{T_{O,a}}\right)^{\frac{1}{4}} \quad (16)$$

6.4 Experiment results

After running the experiments described in Sect. 6.2, the following results were observed. In the first experiment, initially, LSTM was compared with ARIMA model as shown in Table 5. It is apparent in the table that both models have similar MAE and RMSE values and thus have same accuracy. However, the LSTM model is 600 and 130 times faster than ARIMA model in single- and multi-step predictions, respectively. Thus, because of slow prediction speed of ARIMA model, it is not appropriate for real-time usage for container auto-scaling and will not be considered in the remaining section.

On the other side, when comparing prediction speed of LSTM and ANN, they are relatively fast as shown in Table 6. Although ANN is faster than LSTM to predict incoming workload, LSTM is still fast enough, prediction speed below 2 ms, to be used in real-time situations such as containers auto-scaling. The difference in prediction speed between the two models is due to the complex structure of LSTM cell, that is needed to maintain memory.

In terms of prediction accuracy, both LSTM and ANN scores very close results in most cases. As shown in Table 7, when comparing LSTM and ANN for predicting one step ahead (type 1 step), LSTM achieves around 1.5 to

Table 5 Comparison between LSTM and ARIMA

Model Type	LSTM Single	ARIMA	LSTM Multiple	ARIMA
MAE	0.0141	**0.0136**	0.0186	**0.0181**
RMSE	0.0185	**0.0177**	0.0246	**0.0238**
Speed (ms)	**4.9**	3364.7	**47.5**	6205.3

Bold values indicate the best results

Table 6 ANN and LSTM prediction speed

Model	ANN	LSTM	ANN	LSTM
Type	1 step		5 steps	
Speed (ms)	**0.8**	1.6	**0.8**	1.7

Bold values indicate the best results

Table 7 ANN versus LSTM prediction accuracy using different datasets

Model Type	ANN 1 step	LSTM	ANN 5 steps	LSTM
Ref.	M1			
MSE	0.0017	**0.0006**	**0.0009**	0.0010
R^2	0.9430	**0.9790**	**0.9690**	0.9650
Ref.	M2			
MSE	0.0017	**0.0005**	**0.0018**	0.0020
R^2	0.9360	**0.9810**	**0.9320**	0.9270
Ref.	S1			
MSE	0.0018	**0.0009**	**0.0006**	0.0009
R^2	0.9320	**0.9670**	**0.9760**	0.9660
Ref.	S2			
MSE	0.0019	**0.0014**	**0.0012**	0.0014
R^2	0.9100	**0.9330**	**0.9410**	0.9320

Bold values indicate the best results

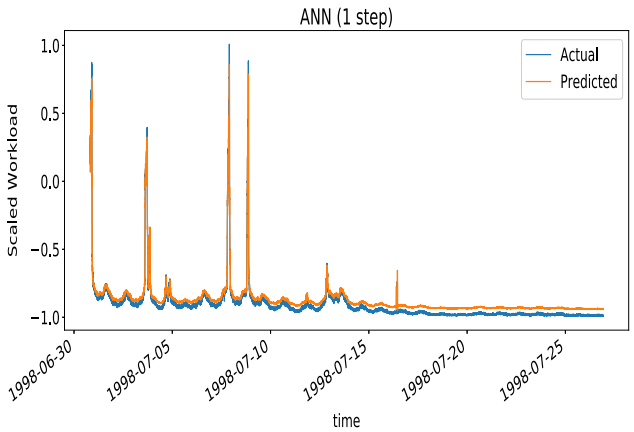

Fig. 11 ANN next-step prediction using M2 dataset

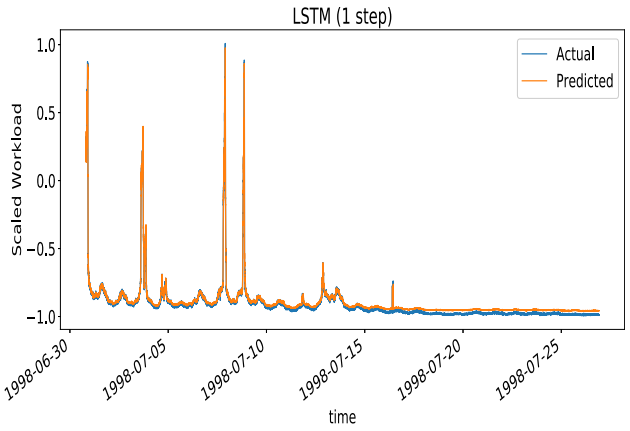

Fig. 12 LSTM next-step prediction using M2 dataset

3.5 times lower MSE than ANN which indicates better prediction accuracy. Figures 11, 12 show that LSTM is more accurate than ANN when using M2 for evaluation. Additionally, in all datasets, both ANN and LSTM score almost the same R^2 value; yet, LSTM is better with minor difference. In contrary, for models of predicting five steps ahead (type 5 steps), although ANN achieves better scores in both MSE and R^2, the difference is too small to be considered. The figures for this case are not shown to keep the discussion concise. The same experiments were carried out for S2 datasets, and similar results were achieved as can be seen in Table 7.

In the second experiment, in the simulation environment, the auto-scaler was evaluated by analyzing its performance in terms of provisioning and elasticity speed. Figures 13 and 14 show the number of containers started for the case of M2 test set for ANN and LSMT auto-scalers, respectively, for the case of one prediction approach. A similar behavior was observed for the case of five-step prediction as well. Comparison of both auto-scalers for all

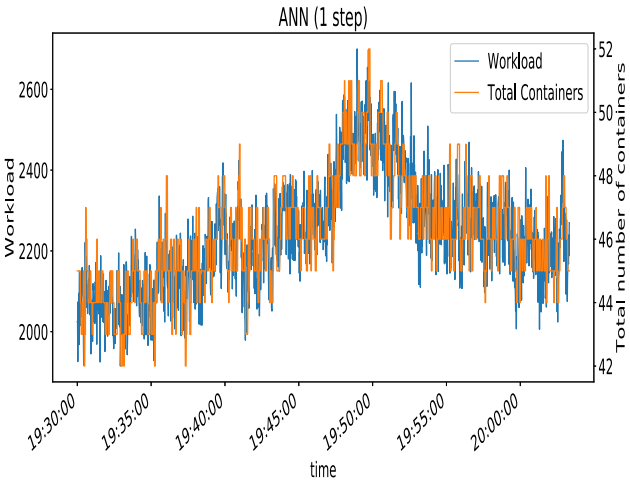

Fig. 13 Number of containers started by ANN auto-scaler

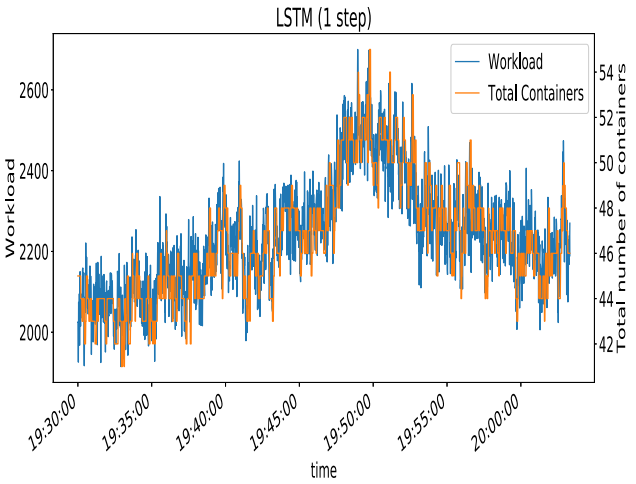

Fig. 14 Number of containers started by LSTM auto-scaler

cases analyzed is summarized in Table 8. The table uses percentages to represent the values for θ and milliseconds for T. It can be observed from the table that the LSTM model achieves better results for θ_U and T_U. That is, it tends not to be under-provisioned and when it does, it occurs for a shorter time interval as compared to ANN. On the contrary, ANN achieves better results in terms of θ_O and T_O and thus it tends not to be over-provision and when it does, only for a shorter time interval as compared to LSTM. Similar characteristics were observed for the five-step models. Nevertheless, LSTM outperforms ANN in general as it achieves better elasticity speed score as it is apparent in Table 8.

The proposed approach was compared to ANN in terms of number of containers supplied to the actual demand. Figure 15 shows the behavior of the ANN model, whereas Fig. 16 shows that for LSMT for the single-step prediction model. It can be seen from comparing the figures that the proposed approach has better match to the demand as compared to the ANN model. Analyzing the case for five-step models, LSMT as compared to ANN performs in a similar fashion.

For the real environment, Fig. 17 shows the number of container replicas that were generated based on the predicted workload compared to the actual workload. The figure illustrates that whenever the workload increases the number of replicas is increased instantaneously to handle the incoming workload. Because of the proposed feature, that is GDS, whenever there is a sudden drop in the workload, the replicas are not decreased immediately. Additionally, even when the workload becomes low and steady, the replicas are decreased gradually to handle any sudden increase in the workload.

Table 8 Auto-scaler evaluation

	no auto-scaling	ANN (1 step)	LSTM (1 step)	ANN (5 steps)	LSTM (5 steps)
θ_U	3.29	0.94	**0.48**	5.07	**1.88**
θ_O	3.06	**3.51**	4.28	**1.42**	2.62
T_U	57.95	26.50	**15.85**	76.50	**42.55**
T_O	57.05	**73.20**	89.10	**32.55**	57.25
ϵ_n	1.00	1.50	**1.84**	1.17	**1.29**

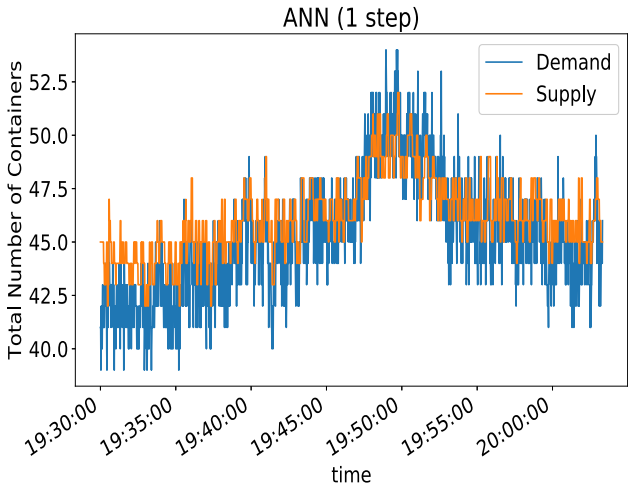

Fig. 15 Supplied versus demanded number of containers by ANN auto-scaler

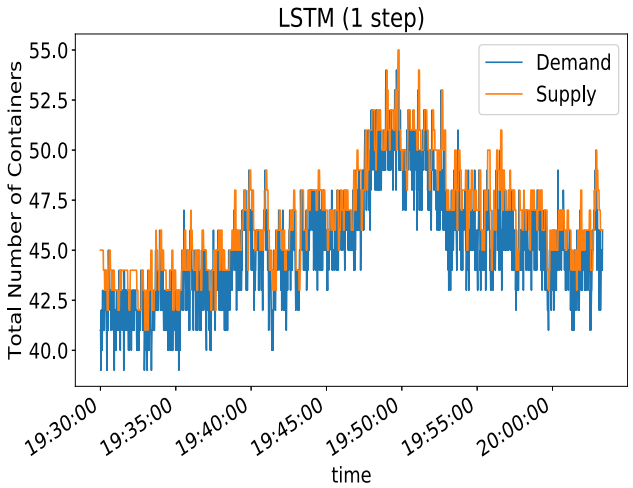

Fig. 16 Supplied versus demanded number of containers by LSTM auto-scaler

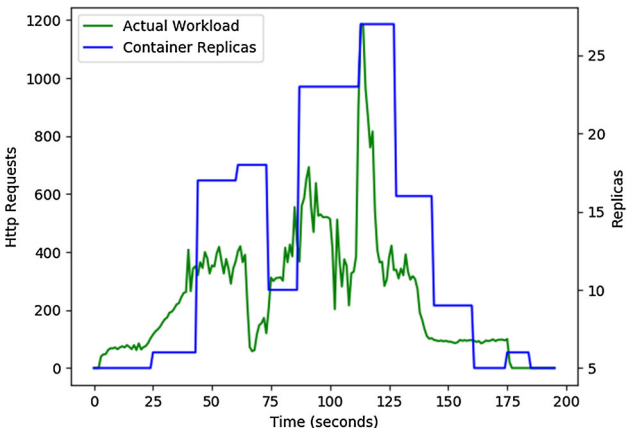

Fig. 17 Number of replicas in response to HTTP workload

7 Conclusion and future works

Containers are the new trend for packaging and deploying micro-services-based cloud applications. The widespread adoption and usage of containers as base technology for cloud systems require proficient auto-scaling methods that automatically and in a timely manner adjust containers provisioning based on the workload demands. In this paper, we proposed a system architecture for Docker containerized application with an auto-scaler based on a machine learning model. Wrapping the auto-scaler into a container makes it easier to deploy and migrate. The auto-scaler details including the monitoring mechanisms, time series database, prediction model, and decision mechanism were introduced. The prediction model was a concise method based on the LSTM model, which learns from historical time series (request statistics) to accurately predict the future workload. In addition, a gradually decreasing strategy (GDS) was proposed to avoid oscillations because of sudden workload spikes. Experimental results demonstrated the efficiency of the proposed architecture as it was able to match the accuracy of systems that are based on ARIMA prediction model but with 600 times speedup in prediction time. As compared with artificial neural network (ANN) model, LSTM model performs better in terms of auto-scaler metrics related to provisioning and elastic speedup. Moreover, the architecture was found to decrease the number of container replicas gradually, thus making it efficient when handling sharp spikes of incoming workloads.

The design of the optimal auto-scaler for Web applications in containers environment is far from trivial. As for future works, we plan to deploy bidirectional LSTM prediction model to achieve more accurate predictions. In our study, we explored only horizontal scaling of containers. Extending this work to vertical scaling (i.e., adding/

removing CPU/memory resources to the same container instance) and/or integrating both horizontal and vertical scaling is another research direction worthy of further investigation.

Acknowledgements The authors would like to thank the anonymous reviewers for their invaluable comments that definitely improved the overall quality of the manuscript.

Compliance with ethical standards

Conflict of Interest The authors declare that they have no conflict of interest.

References

1. Varghese B, Buyya R (2018) Next generation cloud computing: new trends and research directions. Future Gener Comput Syst 79:849–861
2. Alouane M, El Bakkali H (2016) Virtualization in cloud computing: existing solutions and new approach. In: 2016 2nd international conference on cloud computing technologies and applications (CloudTech). IEEE, pp 116–123
3. Pahl C, Brogi A, Soldani J, Jamshidi P (2017) Cloud container technologies: a state-of-the-art review. IEEE Trans Cloud Comput
4. Gupta V, Kaur K, Kaur S (2017) Performance comparison between light weight virtualization using docker and heavy weight virtualization, vol 2, pp 211–216
5. Bernstein D (2014) Containers and cloud: from lxc to docker to kubernetes. IEEE Cloud Comput 1(3):81–84
6. Burns B, Grant B, Oppenheimer D, Brewer E, Wilkes J (2016) Borg, omega, and kubernetes. ACM Queue 14:70–93
7. Jamshidi P, Pahl C, Mendonça NC, Lewis J, Tilkov S (2018) Microservices: the journey so far and challenges ahead. IEEE Softw 35:24–35
8. Soldani J, Tamburri DA, Heuvel W-JVD (2018) The pains and gains of microservices: a systematic grey literature review. J Syst Softw 146:215–232
9. Khazaei H, Bannazadeh H, Leon-Garcia A (2017) Savi-iot: self-managing containerized iot platform. In: 2017 IEEE 5th international conference on future Internet of Things and Cloud (FiCloud), pp 227–234
10. Morabito R, Farris I, Iera A, Taleb T (2017) Evaluating performance of containerized iot services for clustered devices at the network edge. IEEE Internet Things J 4:1019–1030
11. Morabito R, Petrolo R, Loscrì V, Mitton N, Ruggeri G, Molinaro A (2017) Lightweight virtualization as enabling technology for future smart cars. In: 2017 IFIP/IEEE symposium on integrated network and service management (IM), pp 1238–1245
12. Buyya R, Srirama SN, Casale G, Calheiros R, Simmhan Y, Varghese B, Gelenbe E, Javadi B, Vaquero LM, Netto MAS, Toosi AN, Rodriguez MA, Llorente IM, Vimercati SDCD, Samarati P, Milojicic D, Varela C, Bahsoon R, Assuncao MDD, Rana O, Zhou W, Jin H, Gentzsch W, Zomaya AY, Shen H (2018) A manifesto for future generation cloud computing: research directions for the next decade. ACM Comput Surv 51:105:1–105:38
13. Al-Dhuraibi Y, Paraiso F, Djarallah N, Merle P (2018) Elasticity in cloud computing: state of the art and research challenges. IEEE Trans Serv Comput 11:430–447
14. Lorido-Botran T, Miguel-Alonso J, Lozano JA (2014) A review of auto-scaling techniques for elastic applications in cloud environments. J Grid Comput 12:559–592
15. Aslanpour MS, Ghobaei-Arani M, Toosi AN (2017) Auto-scaling web applications in clouds: a cost-aware approach. J Netw Comput Appl 95:26–41
16. Huebscher MC, McCann JA (2008) A survey of autonomic computing-degrees, models, and applications. ACM Comput Surv 40:7:1–7:28
17. Qu C, Calheiros RN, Buyya R (2018) Auto-scaling web applications in clouds: a taxonomy and survey. ACM Comput Surv 51:73:1–73:33
18. Cardenas YMR (2018) Scaling policies derivation for predictive autoscaling of cloud applications. Master's thesis, University of Munich
19. Klinaku F, Frank M, Becker S (2018) Caus: an elasticity controller for a containerized microservice. In: Companion of the 2018 ACM/SPEC international conference on performance engineering, ICPE '18, New York. ACM, pp 93–98
20. Al-Dhuraibi Y, Paraiso F, Djarallah N, Merle P (2017) Autonomic vertical elasticity of docker containers with elasticdocker. In: 2017 IEEE 10th international conference on cloud computing (CLOUD), pp 472–479
21. Taherizadeh S, Stankovski V (2018) Dynamic multi-level auto-scaling rules for containerized applications. Comput J 62:174–197
22. Zhang F, Tang X, Li X, Khan SU, Li Z (2019) Quantifying cloud elasticity with container-based autoscaling. Future Gener Comput Syst 98:672–681
23. Kan C (2016) Docloud: an elastic cloud platform for web applications based on docker. In: 2016 18th international conference on advanced communication technology (ICACT), p 1
24. Li Y, Xia Y (2016) Auto-scaling web applications in hybrid cloud based on docker. In: 2016 5th International conference on computer science and network technology (ICCSNT), pp 75–79
25. Kubernetes horizontal pod auto-scaling.' http://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale. Accessed 19 April 2019
26. Ciptaningtyas HT, Santoso BJ, Razi MF (2017) Resource elasticity controller for docker-based web applications. In: 11th international conference on information & communication technology and system (ICTS), pp 193–196
27. Meng Y, Rao R, Zhang X, Hong P (2016) Crupa: a container resource utilization prediction algorithm for auto-scaling based on time series analysis. In: 2016 International conference on progress in informatics and computing (PIC), pp 468–472
28. Kim W-Y, Lee J-S, Huh E-N (2017) Study on proactive auto scaling for instance through the prediction of network traffic on the container environment. In: Proceedings of the 11th international conference on ubiquitous information management and communication, IMCOM '17, New York, NY, USA. ACM, pp 17:1–17:8
29. Borkowski M, Schulte S, Hochreiner C (2016) Predicting cloud resource utilization. In: 2016 IEEE/ACM 9th international conference on utility and cloud computing (UCC), pp 37–42
30. Sangpetch A, Sangpetch O, Juangmarisakul N, Warodom S (2017) Thoth: automatic resource management with machine learning for container-based cloud platform, pp 103–111
31. Pouyanfar S, Sadiq S, Yan Y, Tian H, Tao Y, Reyes MP, Shyu M-L, Chen S-C, Iyengar S (2018) A survey on deep learning: algorithms, techniques, and applications. ACM Comput Surv (CSUR) 51(5):92
32. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780
33. Pascanu R, Mikolov T, Bengio Y (2013) On the difficulty of training recurrent neural networks. In: Proceedings of the 30th

international conference on international conference on machine learning, vol 28, ICML'13, JMLR.org, pp III-1310–III-1318
34. Ye T, Guangtao X, Shiyou Q, Minglu L (2017) An auto-scaling framework for containerized elastic applications. In: 2017 3rd international conference on big data computing and communications (BIGCOM), pp 422–430
35. Box GEP, Jenkins GM, Reinsel GC, Ljung GM (2015) Time series analysis: forecasting and control, 5th edn. Wiley, Hoboken
36. Baresi L, Guinea S, Leva A, Quattrocchi G (2016) A discrete-time feedback controller for containerized cloud applications. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, FSE 2016, New York, NY, USA. ACM, pp 217–228
37. Wu S, Zhang D, Yan B, Guo F, Sheng D (2018) Auto-scaling web application in docker based on gray prediction. In: 2018 International conference on network, communication, computer engineering (NCCE 2018). Atlantis Press, 2018/05, pp 169–174
38. Chiang JS, Wu PL, Chiang SD, Chang TJ, Chang ST, Wen KL (1998) Introduction of grey system theory. GAO-Li Publication, Taiwan
39. Watkins CJCH, Dayan P (1992) Q-learning. In: Machine learning, pp 279–292
40. Truyen E, Van Landuyt D, Preuveneers D, Lagaisse B, Joosen W (2019) A comprehensive feature comparison study of open-source container orchestration frameworks. Appl Sci 9(5)
41. Haproxy—the reliable, high-performance tcp/http load balancer. http://www.haproxy.org. Accessed 16 April 2019
42. Arlitt M, Jin T (2000) A workload characterization study of the 1998 world cup web site. IEEE Netw 14(3):30–37
43. Bauer A, Herbst N, Spinner S, Ali-Eldin A, Kounev S (2018) Chameleon: a hybrid, proactive auto-scaling mechanism on a level-playing field. IEEE Trans Parallel Distrib Syst 30(4):800–813
44. Messias VR, Estrella JC, Ehlers R, Santana MJ, Santana RC, Reiff-Marganiec S (2016) Combining time series prediction models using genetic algorithm to autoscaling web applications hosted in the cloud infrastructure. Neural Comput Appl 27(8):2383–2406
45. Roy N, Dubey A, Gokhale A (2011) Efficient autoscaling in the cloud using predictive models for workload forecasting. In: 2011 IEEE 4th international conference on cloud computing. IEEE, pp 500–507
46. Moore LR, Bean K, Ellahi T (2013) Transforming reactive auto-scaling into proactive auto-scaling. In: Proceedings of the 3rd international workshop on cloud data and platforms. ACM, pp 7–12
47. Herbst N, Krebs R, Oikonomou G, Kousiouris G, Evangelinou A, Iosup A, Kounev S (2016) Ready for rain? A view from spec research on the future of cloud metrics. arXiv:1604.03470
48. Bauer A, Grohmann J, Herbst N, Kounev S (2018) On the value of service demand estimation for auto-scaling. In: International conference on measurement, modelling and evaluation of computing systems. Springer, New York, pp 142–156