Received 7 May 2024, accepted 15 May 2024, date of publication 20 May 2024, date of current version 29 May 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3403451

RESEARCH ARTICLE

A Trend Detection-Based Auto-Scaling Method for Containers in High-Concurrency Scenarios

HAIPENG LIU[1], WENHAO ZHU[1], SIYI FU[1], AND YONGJUN LU[2]
[1]School of Computer Science, Shanghai University, Shanghai 200444, China
[2]Shanghai Center for Student Affairs, Shanghai 200235, China

Corresponding author: Yongjun Lu (luyongjun@shec.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61572434, and in part by Shanghai Science and Technology Committee under Grant 19DZ2204800.

ABSTRACT Cloud computing technology is widely embraced due to its ability to meet diverse computing resource demands from users. As the user base grows, the challenge for cloud service providers to dynamically allocate resources to applications according to real-time customer needs becomes more daunting. Existing auto-scaling mechanisms, such as Horizontal Pod Autoscaler (HPA), lack flexibility in handling transient and irregular fluctuations in resource demands, making it difficult to respond quickly. This research focuses on auto-scaling solutions for applications in high-concurrency scenarios. The study enhances the MAPE structure of traditional auto-scaling models and introduces a trend detection module in a proactive auto-scaling engine based on predictive algorithms. This module detects trends during transient irregular request volume fluctuations, corrects prediction algorithm results inconsistent with the current trend, and provides more reasonable resource allocation for programs. Experimental results, using both real access peak data and simulated data in a Kubernetes environment, indicate that the proposed auto-scaler effectively prevents resource shortages in high-concurrency network environments compared to other auto-scaling mechanisms. It ensures the performance and availability of applications while reducing resource wastage.

INDEX TERMS Cloud computing, auto-scaler, prediction, proactive controller, high-concurrency, trend detection.

I. INTRODUCTION

With a strong demand for efficient deployment and management of applications, cloud computing technology has witnessed rapid development in recent years. Cloud computing can satisfy the resource requirements of applications in various environments and architectures. Additionally, it provides elastic resource allocation policies, allowing real-time adjustments to program resource utilization. In high-load scenarios, extra resources can be allocated to ensure Quality of Service (QoS), while in low-load conditions, the cloud computing platform reduces the allocated resources to programs, enhancing resource efficiency and reducing costs [2]. Typically, users negotiate Service Level Agreements (SLAs) with cloud service providers, and cloud computing makes scalability decisions for program resource usage within the constraints of SLAs to achieve load balancing.

Containers are currently the mainstream deployment method for cloud applications [4]. Unlike virtual machines, containers are a lightweight virtualization solution that shares the host's kernel while encapsulating the application's code, dependencies, and runtime environment, making them more portable [3]. Due to the elastic nature of the cloud, containers, being easily portable and scalable [5], are more suitable as the smallest unit of scaling. Various automatic scaling methods are implementations of the elasticity feature of the cloud. These automatic scaling methods require different scaling strategies depending on their applicability. For containers built for individual programs, autoscalers can change the resource configuration of the target program

The associate editor coordinating the review of this manuscript and approving it for publication was Alberto Cano.

through horizontal scaling and vertical scaling. Horizontal scaling is achieved by adding or removing container instances associated with the application, and HPA [1] is an autoscaling method that uses horizontal scaling. Vertical scaling involves changing the resource specifications allocated to containers, and methods like [12] and [13] adopt vertical scaling. For popular microservices architecture applications, where each service in the system is deployed as an independent container, the response time of requests is affected by end-to-end latency between services [6]. Therefore, autoscaling methods for microservices need to consider dependencies between services when changing resource configurations and cannot simply scale one service. For edge clusters, autoscaling methods need to consider situations where cluster resources are limited [7]. Autoscaling methods for scalable distributed AI frameworks need to consider optimizing the performance and cost-effectiveness of AI workloads in the cloud [9]. The autoscaler proposed in this paper aims to reduce resource shortages in high-concurrency scenarios.

An ideal autoscaler should rapidly respond to load changes and make resource adjustments, making the choice of scaling timing critical. Currently, there are three options for scaling timing: proactive scaling, reactive scaling, and a hybrid approach combining proactive and reactive scaling. Proactive scaling methods typically model historical data, predict future loads, and make resource allocation decisions in advance [15]. Reactive scaling methods typically collect resource utilization metrics at each monitoring cycle and compare the collected metrics with initially set thresholds. If the current metric values exceed or fall below the thresholds, the application's resource configuration is changed [10]. Currently, proactive scaling methods based on prediction algorithms are more popular. However, web applications in large-scale internet environments with a large user base often face sudden spikes in users, such as promotional events or social media trends [19]. In such cases, proactive scaling strategies may not adjust in time, and passive methods lacking historical data support may also struggle to adapt to the situation. Existing scalability strategies face challenges in effectively addressing instantaneous and irregular load fluctuations in high-concurrency environments.

Our work proposes a horizontally automated scaling engine based on a predictive approach. It proactively predicts and configures resources based on historical data and performs trend detection for instantaneous, irregular load fluctuations in high-concurrency environments before resource allocation. We have improved the MAPE structure of traditional automatic scaling models, employing predictive algorithms to calculate the future pod count n time steps ahead. Before actual scaling execution, we have incorporated a trend detection module to identify potential instantaneous, irregular load fluctuations and allocate resources based on the fluctuation situation. We conducted tests on a single pod in a Kubernetes environment under different workload levels, recommending the most suitable workload for the pod under SLA constraints. We trained multiple prediction models, including LSTM, ARIMA, and the Informer [18] model, using data from university campus network authentication interfaces, compared their prediction results, and selected the model most suitable for our auto-scaler. We validated our auto-scaler's ability to ensure application performance and availability under real and simulated peak request data.

II. RELATED WORK

Due to the urgent demand from cloud service users for elastic resource provisioning, the configuration methods of elastic resources have been a focal point of research. Al-Dhuraibi et al. [8] provided a definition of elasticity in cloud computing and comprehensively classified elastic mechanisms. According to Tran et al.'s [10] comprehensive survey on auto-scalers on Kubernetes, current auto-scaling methods are categorized into different classes based on various aspects such as the target application architecture, scaling methods, scaling triggers, and monitoring metrics.

The most common classification method has traditionally been based on scaling methods, which are divided into horizontal scaling methods and vertical scaling methods. ELASTICDOCKER proposed by Al-Dhuraibi et al. [12] and RUBAS proposed by Rattihalli et al. [13] are both examples of vertical scaling methods. They adjust container memory and virtual CPU core resources based on the current workload and achieve vertical elasticity through real-time container migration. The advantage of vertical scaling methods is that they do not lead to resource wastage, but the downside is that container migration interrupts may affect program service quality. The most prevalent horizontal scaling method is the Horizontal Pod Autoscaler (HPA) mechanism [1]. It serves as Kubernetes' default auto-scaling solution, monitoring pod resource usage, comparing it with user-defined resource usage thresholds, and determining whether to increase or decrease the number of pods.

Another classification method is based on scaling triggers, mainly divided into proactive scaling methods and reactive scaling methods. References [12] and [13] are examples of reactive scaling methods, where the auto-scaler adjusts container resources when the load exceeds a threshold. Rossi et al. proposed a RL-based dynamic multi-metric threshold strategy [27] for dynamically adjusting the scaling thresholds for CPU and memory usage, addressing the issue that static threshold-based scaling strategies for single metrics may not adapt well to heterogeneous applications. Imdoukh et al. [16] proposed an auto-scaling framework as a proactive scaling method, leveraging machine learning techniques to optimize automatic scaling strategies for containerized applications. Their approach utilizes Long Short-Term Memory (LSTM) neural networks to forecast future HTTP workloads and adjusts the number of container replicas before the actual workload arrives, thus alleviating delays caused by starting or stopping containers. Toka et al. introduced HPA+ [17], which is also a proactive scaling method, predicting the number of network requests in the future using a combination of LSTM, Autoregressive (AR), and Hierarchical Temporal Memory

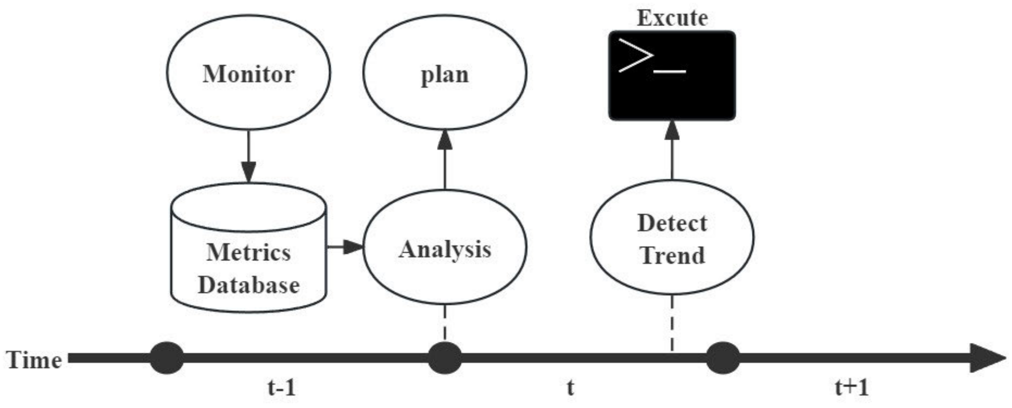


FIGURE 1. Custom auto-scaler architecture.

(HTM) models. Currently, there are also many hybrid methods that incorporate both scaling triggers. Yadav et al. [23] proposed a hybrid method combining reactive and predictive approaches, where the reactive approach primarily scales running containers when the workload increases, while the proactive approach is used to scale containers when the workload decreases.

As microservices architecture gradually becomes the mainstream approach for enterprises to build scalable and efficient applications, research on auto-scaling methods is primarily classified based on the architecture of the target application. In a microservices architecture, response time is influenced by multiple microservices [11], making it challenging for traditional auto-scaling techniques to ensure overall system performance. Unlike auto-scaling methods designed for monolithic applications, auto-scaling methods for microservices must consider the relationships between microservices. Goli et al. proposed Waterfall [14], designed specifically for microservices applications. Waterfall considers the dependencies between microservices in the microservices architecture. It abstracts the target microservices application into a directed graph called the microservices graph. In this graph, vertices represent services, and edges represent dependencies between services. The direction of the edge determines the direction of request transmission. Waterfall initially monitors CPU usage and scales microservices exceeding the threshold. Then, based on the microservices graph and machine learning models, it evaluates and predicts the potential impact of scaling one microservice on other microservices to find the optimal scaling configuration for the entire system. Ahmad et al. proposed Smart HPA, also designed for microservices architecture [19]. Smart HPA combines centralized and decentralized architectural styles, using a hierarchical architecture that enables flexible decentralized management when resources are abundant and effective resource allocation through centralized modules when resources are constrained. The core of Smart HPA lies in its resource-efficient heuristic algorithm, allowing resource exchange between microservices to adapt to changing workloads. Al Qassem et al. proposed a prediction-based auto-scaler using Random Forest (RF) for microservice resource allocation [20]. This method predicts the future CPU and memory utilization of microservices workloads using machine learning models to guide vertical and horizontal scaling. Khaleq and Ra proposed an intelligent auto-scaling system for microservices in the cloud [21]. The system consists of two main modules: the first module utilizes a general auto-scaling algorithm on Google Kubernetes Engine (GKE) to identify the resource requirements of microservices, while the second module uses a Reinforcement Learning (RL) agent to learn and determine auto-scaling thresholds to meet resource demands and QoS constraints. Santos et al. introduced the gym-hpa framework [22], another reinforcement learning-based auto-scaling framework specifically designed for complex microservices applications running on the Kubernetes platform. RL agents determine the scaling triggers based on real-time monitoring data and select the most appropriate scaling actions by analyzing current resource usage, service request volume, and other relevant metrics.

Most automatic scaling methods perform well in configuring resources for applications based on real-time performance data and prediction models. However, during periods of peak request volumes, it's challenging to predict the exact number of requests, making it difficult for most automatic scaling methods to rapidly allocate sufficient resources to applications. References [21] and [22] propose using reinforcement learning methods to address unpredictable request changes, but reinforcement learning requires a large amount of sample data, and currently, there is limited public data on peak requests. Reference [28] divides applications into multiple layers, such as the web server layer, application server layer, and database server layer, to ensure the quality of service by considering both hardware resources and software resources (such as server threads and database connections).

This paper proposes a trend-based automatic scaling method to address unpredictable request changes during

TABLE 1. A comparison of related work on autoscaling methods.

Work	Monitored indicators	Estimation technique	Timing	Scaling Methods	Scope
[12], [13]	CPU, memory	Threshold-based	Reactive	Vertical	Monolithic
[27]	CPU, response time	Threshold-based, RL	Reactive	-	Microservices
[16]	Requests, CPU, memory	LSTM	Proactive	Horizontal	Monolithic
[17]	Requests	AR, HTM, LSTM	Proactive	Horizontal	Monolithic
[23]	CPU, memory	ARIMA, SARIMA, SVR	Hybrid	Horizontal	Monolithic
[14]	CPU	LR, RF, SVM	Proactive	Horizontal	Microservices
[19]	CPU, replicas	Resource Exchange Algorithm	Reactive	Horizontal	Microservices
[20]	CPU, memory	RF	Hybrid	Horizontal and Vertical	Microservices
[21]	CPU, memory, queue, response time	RL	Reactive	Horizontal	Microservices
[22]	CPU, memory, response time	RL	Reactive	Horizontal	Microservices
[28]	Requests	Concurrency-aware autoScaling	Reactive	Horizontal	Mon. & Micro.
Our study	Requests	Trend detection, LSTM	Proactive	Horizontal	Monolithic

peak periods. This method first uses prediction algorithms to calculate the future number of containers n time steps in advance. Before actually scaling, a trend detection module identifies potential transient and irregular request fluctuations and allocates resources based on the fluctuation situation.

III. AUTO-SCALER ARCHITECTURE

Our custom auto-scaler has made improvements to the MAPE loop followed by traditional scalers, and Fig. 1 illustrates the structure of our custom auto-scaler. The main improvement is the addition of a trend detection module before the actual execution steps. The primary function of this module is to detect changes in the trend of network requests in the recent time period and correct the predictions made by the predictive algorithm. Below, we will introduce each module of the auto-scaler.

A. MONITOR MODULE

The monitor continuously collects various relevant metrics and stores the data in a time-series format in the database. In our proposed auto-scaler, the collected data includes resource usage information for the program instances managed by the auto-scaler (such as containers), statistical information on HTTP requests received per second by the application, and runtime log information for the application. The logs are primarily used to calculate their actual throughput per second.

B. ANALYSIS MODULE

The analysis module periodically retrieves HTTP request data (requests per second) from the database. It then predicts the number of HTTP requests for the subsequent two moments. For example, at time t−1, it predicts the request data for times t and t+1.

C. PLANNING MODULE

After computing the predicted workload of the application from the analysis module, The planning module calculates the resources needed by the application. Since we are evaluating our automatic scaler in a Kubernetes environment, we use the number of pods to represent the required resources. The planning module determines the pod quantity at time t+1 using (1). $request_{pod}$ represents the optimal number of requests that a single pod can handle simultaneously, and $Request_{t+1}$ represents the predicted value for time t+1 at time t−1.

$$pod_{t+1} = \frac{Request_{t+1}}{request_{pod}} \tag{1}$$

D. TREND DETECTION MODULE

Before entering the next time period, the trend detection module runs periodically. This is done because pod startup takes some time, and we want to ensure that there are enough pods working when entering the next time period. The trend detection algorithm is outlined below.

The primary function of the trend detection module is to analyze HTTP request data and web application throughput data within the current time period, determine the real-time development trend of network requests, and then decide whether to adjust the number of pods in the prediction queue for the next time step based on this trend. We are not concerned with fluctuations in network requests throughout all time periods; our focus is on whether the predictive algorithm can accurately forecast future HTTP request data in situations of resource scarcity or excess. The trend detection algorithm primarily aims to identify trends in changes in network requests that might occur in such situations.

$$under_t = \sum_{i=1}^{T} (throughput_i \ll requests_i) \tag{2}$$

$$over_t = \sum_{i=1}^{T} (requests_i \ll request_{pod}) \tag{3}$$

The trend detection algorithm requires multiple input parameters. Among them, pred[] contains the predictions of the prediction algorithm for the current time step and the next time step, based on the previous time step. $mean_t$ represents the average number of HTTP requests received per second in the current time period, while std_t denotes the standard deviation of HTTP requests received per second, indicating the magnitude of request changes within the time period. $under_t$ represents the number of time points in the current time period where the throughput is significantly lower

Algorithm 1 Detect Trend Algorithm

```
Require: $pred[], mean_t, std_t, under_t, over_t, CDT, FLAG$
Ensure: $pod_{t+1}$
 1: if $CDT > 0$ then
 2:   $CDT --$
 3:   $pod_{t+1} \leftarrow pod_t$
 4: else
 5:   if $under_t > n_{under}$ then
 6:     if $mean_t < k_{under} \times pred[0]$ and $std_t > k_{std}$ then
 7:       $FLAG \leftarrow FLAG + 1$
 8:     else
 9:       if $mean_t > k_{under} \times pred[0]$ and $std_t > k_{std}$ then
10:         if LinearRegression(data) is up then
11:           $pod_{t+1} \leftarrow \frac{mean_t+std_t}{request_{pod}}$
12:         else
13:           $FLAG \leftarrow FLAG + 1$
14:         end if
15:       else
16:         if $mean_t > k_{under} \times pred[0]$ and $std_t < k_{std}$
            then
17:           $CDT \leftarrow 4$
18:           $FLAG \leftarrow 0$
19:           $pod_{t+1} \leftarrow \frac{mean_t}{request_{pod}}$
20:         end if
21:       end if
22:     end if
23:     if $FLAG$ is 2 then
24:       $pod_{t+1} \leftarrow \frac{mean_t+std_t}{request_{pod}}$
25:       $CDT \leftarrow 4$
26:       $FLAG \leftarrow 0$
27:     end if
28:   else
29:     $FLAG \leftarrow 0$
30:   end if
31:   if $over_t > n_{over}$ then
32:     if $mean_t < k_{over} \times pred[0]$ and $std_t > k_{std}$ then
33:       if LinearRegression(data) is down then
34:         $CDT \leftarrow 4$
35:         $pod_{t+1} \leftarrow \frac{mean_t+std_t}{request_{pod}}$
36:       end if
37:     else
38:       if $mean_t < k_{over} \times pred[0]$ and $std_t < k_{std}$ then
39:         $CDT \leftarrow 4$
40:         $pod_{t+1} \leftarrow \frac{mean_t}{request_{pod}}$
41:       end if
42:     end if
43:   end if
44: end if
```

than the actual requests received. Equation 2 describes the calculation formula for $under_t$, where throughput$_i$ represents the system throughput at a specific time point and requests$_i$ represents the number of requests received by the system at that time point. $over_t$ represents the number of time points where the throughput significantly exceeds the actual throughput that all pods can handle. request$_{pod}$ represents the maximum number of requests that the allocated resources to the system can handle.

n_{under} represents the acceptable number of such time points. When $under_t$ exceeds n_{under}, it indicates a potential resource shortage. Similarly, when $over_t$ exceeds n_{over}, it indicates an oversupply of resources. CDT represents the cooling down time, initially set to 0. If the trend detection algorithm detects an abnormal trend and scales the Pods accordingly, CDT is set to 4, indicating that no scaling will be performed for the next 4 time periods after scaling to avoid container oscillation. FLAG is used as a counter; when an abnormal trend such as the number of requests in several time points within the current period being significantly higher than the average occurs, it may not affect the current application's service instructions but needs to be recorded. If the abnormal request variation persists for several consecutive periods, the trend detection algorithm will also identify it as an abnormal trend and correct the prediction algorithm's forecast results.

In addition to the algorithm's input parameters, there are also many built-in parameters within the algorithm. *kunder* represents the tolerance parameter for resource shortages. The comparison result between $mean_t$ and $pred[0]$ multiplied by this parameter indicates the existence of shortages. k_{std} represents the tolerable request fluctuation magnitude within a time period. The final output of the algorithm, pod_{t+1}, is the number of pods determined by the algorithm for the t+1 time period.

This algorithm evaluates the accuracy of the prediction algorithm and the real-time trends of the network. In case of resource shortage, the algorithm evaluates whether the prediction result (average requests per second) for the current time period in the multi-step prediction is within an acceptable range. If they are tolerable, the auto-scaler continues with the previous prediction results. However, it observes the fluctuations in requests. If data shows significant fluctuations over consecutive time periods, intervention is considered for the next scalability decision to ensure service quality.

In case the prediction result is intolerable, the algorithm examines the request fluctuations within the current time period. If the fluctuations are small, indicating a stable user count for web services, future resource needs can be estimated based on the current user request situation. If fluctuations are significant, linear regression is used to fit the current request changes. If the trend shows a clear upward trajectory, the peak is considered as the future average requests per second, and the prediction algorithm's results are adjusted accordingly. If the trend is chaotic, further observation is postponed until the next time period.

Similarly, in cases of oversupply of resources, the algorithm evaluates the tolerance of the prediction results and the request fluctuations. Unlike in resource scarcity situations, the algorithm is less concerned about whether the prediction results are tolerable, as it does not affect

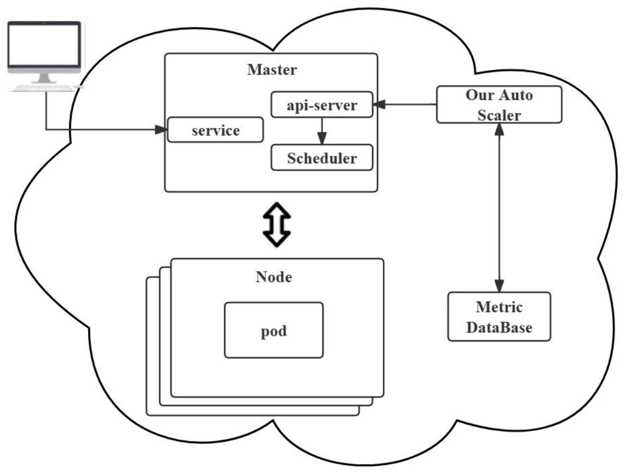


FIGURE 2. The working process of the auto-scaler in kubernetes.

the service quality of web services. A significant reduction in resources may yield adverse results. Resource reduction is only considered when there is a significant and stable downward trend in user requests for web services.

IV. EXPERIMENTS AND EVALUATION

This section primarily evaluates the performance of our custom automatic scaler in high-concurrency network environments through experiments. We test the response time of web applications under different concurrency levels on a Kubernetes cluster and discuss the tolerance level in the algorithm. We assess three popular time-series data prediction models to select the most suitable predictive model for the target scenario. Finally, we test the performance of our automatic scaler in a Kubernetes cluster using real access peak data and simulated access peak data.

A. ASSESSMENT ENVIRONMENT

We conducted experiments and evaluations of our auto-scaler in a Kubernetes v1.23.1 environment. This choice was made because Kubernetes, as a powerful container orchestration platform, is widely used in the field of cloud computing [24]. Our auto-scaler is designed to manage the scaling of applications in complex network environments, and experimenting on Kubernetes simplifies the scaling operations on containers. Figure 2 illustrates the workflow of our auto-scaler on the Kubernetes cluster. The scaler determines the number of pods for the next time step based on the monitoring data collected in the database. Subsequently, it invokes the API to execute the scaling command, and finally, the Kubernetes scheduler instructs the respective worker nodes to start the pods.

B. EVALUATION METRICS

To evaluate the performance of a predictive model, we use four key metrics: mean square error (MSE), root mean square error (RMSE), mean absolute error (MAE), and R-squared. These metrics help evaluate the difference between the predictions generated by the model and the actual observations. MSE measures the mean square error between predicted values and actual observed values. Equation 4 is its calculation formula. Among them y_i represents the observed value, $\hat{y}_i$ represents the predicted value, and n is the number of samples. RMSE is the square root of MSE and provides an estimate of the standard deviation of the forecast error. Equation 5 shows how it is calculated. where samples is the number of samples. MAE measures the mean absolute difference between predicted values and actual observations. Equation 6 explains how RMSE is calculated. where samples is the number of samples. R-squared is a statistical measure that indicates the proportion of variance in the dependent variable that is explained by the independent variables in a regression model. Equation 7 is its calculation formula. Among them, y_i represents the observed value, $\hat{y}_i$ represents the predicted value, $\bar{y}$ represents the average value of the observed value, and n is the sample number.

$$MSE(y,\hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2 \tag{4}$$

$$RMSE(y,\hat{y}) = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1}(y_i-\hat{y}_i)^2} \tag{5}$$

$$MAE(y,\hat{y}) = \frac{1}{n}\sum_{i=0}^{n-1}|y_i-\hat{y}_i| \tag{6}$$

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i-\hat{y}_i)^2}{\sum_{i=1}^{n}(y_i-\bar{y})^2} \tag{7}$$

The accuracy of resource allocation is a key metric of concern. It evaluates the alignment between the resources allocated by the autoscaler and the current workload of the program, while considering potential shortages and surpluses. We calculate this accuracy by comparing the actual number of pods required by the program with the number allocated by each autoscaler. To assess the effectiveness of pod changes under different autoscalers, we use equation (8) to compute the underallocation indicator (Θ_U) and equation (9) to compute the overallocation indicator (Θ_O) [25]. In equations (8) and (9), T represents the total number of time periods, demand$_t$ represents the actual resource demand, and supply$_t$ represents the resources allocated by the autoscaler.

$$\Theta_U[\%] = \frac{100}{T}\sum_{t=1}^{T}\frac{\max(demand_t - supply_t, 0)}{demand_t} \tag{8}$$

$$\Theta_O[\%] = \frac{100}{T}\sum_{t=1}^{T}\frac{\max(supply_t - demand_t, 0)}{demand_t} \tag{9}$$

C. BENCHMARK TESTING

We deployed the web application onto a Kubernetes working node and conducted stress testing on the web service. JMeter [26] is a powerful open-source performance testing

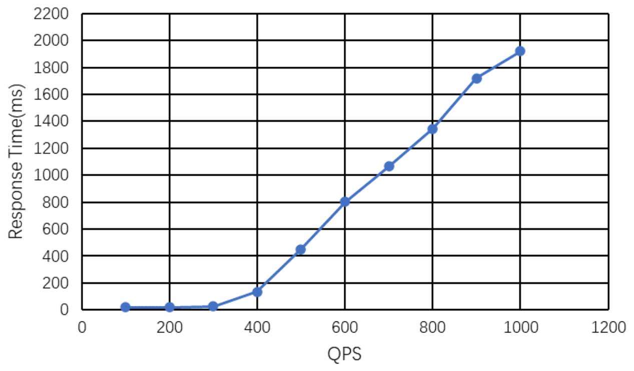


FIGURE 3. Response time of programs under different levels of concurrency.

tool. We used it to conduct the stress tests. Initially, we sent 100 requests per second to the web application for a duration of 1 minute. Subsequently, we increased the number of requests per second in increments of 100 for each test. After each test, we recorded the average response time for the requests within that minute. Through stress testing, we obtained insights into the performance of a single pod under varying levels of concurrency. Fig. 3 illustrates our test results.

We observed that when the Queries Per Second (QPS) is below 300, the program's response time remains normal. However, when the QPS exceeds 300, the response time experiences a sharp increase. This indicates that at around 300 QPS, the program's throughput is roughly in balance with the number of requests, signifying a equilibrium between pod load and resource supply. Beyond 300 QPS, we start to witness insufficient resource supply.

To optimize resource utilization while ensuring service quality, we set this balance point as $request_{pod}$. Additionally, we observed that within the QPS range of 300 to 400, the increase in response time is relatively small, making it imperceptible to users in the short term. This implies that we can tolerate slightly higher pod loads for a short duration, but if the pod load becomes excessively high, the program's response time also increases significantly. In other words, we can tolerate insufficient resource supply when QPS is higher than 300 but not exceeding 450. The parameter k_{under} represents the range beyond which we do not want QPS to exceed, and we set it to 1.5. Similarly, k_{over} represents the range above which we want QPS to consistently stay, and we set it to 0.6.

D. DATASET

The real data used for testing the trend algorithm comes from the authentication log data of the university campus network authentication system in early January 2024. We aggregated the log data by minute. Fig. 5 shows the processed dataset, where the values represent the number of authentication requests per minute for the network system. The dataset directly indicates the real-time growth trend of users on the university campus network. We aim to select the best prediction algorithm directly from this dataset for various network request growth scenarios.

By carefully analyzing Fig. 5, we can observe that although the overall change in the number of network authentication requests follows a sinusoidal trend each day, the time periods of each day's peak and the amplitude of data change vary. Traditional automatic scalers can handle regular continuous growth well, but they struggle with this scenario. We will test our trend detection algorithm under various changes during peak periods. We extracted the data during the peak period on the evening of January 5, 2024, from the dataset shown in Fig. 5 as experimental data. Due to the small scale of the actual dataset, in real experiments, we magnify all the data in this interval by 60 times. Fig. 7 shows the trend of network request changes per second after processing the peak period data. The blue line in Fig. 8 represents the average value per minute after data processing. In Section IV-G, we discuss the performance of our trend detection algorithm compared to other automatic scalers on this data.

The dataset shown in Fig. 7 still does not fully represent the possible request variations during peak periods. In real-world events like shopping festivals, the number of requests may experience significant growth within seconds or even milliseconds. We need to evaluate our automatic scaler in such scenarios. We need to simulate a period of HTTP requests that grow rapidly in a short time and then decline slowly. The blue part in Fig. 12 illustrates the trend of the average QPS simulated data per minute. We simulated a peak period lasting about 20 minutes, with the network requests entering the peak in the second minute, rapidly rising to the maximum value within two minutes, and then slowly decreasing after the peak. In the first two minutes, we used a logarithmic growth model to simulate the increase in the number of network requests. The logarithmic growth model can better capture the rapid growth trend of network load in the initial stage, while in the subsequent decline of requests, we used an exponential decay model to simulate the slow decline in the number of network requests. Exponential decay models are commonly used to simulate the decline in popularity in scenarios such as computing news relevance, and similarly, they can express the decline in network traffic after hot events effectively. We initially simulated the average QPS value per minute during peak periods using these two models. To better approximate reality, we added random errors to each value. Then, we used a normal distribution model to simulate the number of requests per second for each minute. Fig. 11 shows our simulation of the number of requests per second during peak periods.

E. PARAMETER SELECTION

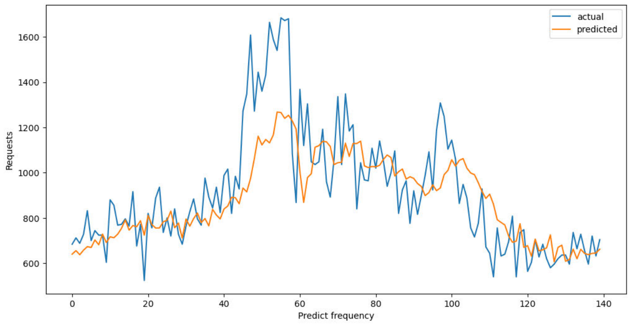
An ideal autoscaler should respond to changes in application load and resource utilization as quickly as possible. However, considering the time required for program startup, for reaction-based autoscalers, the resource adjustment cycle must cover the program's startup time; otherwise, the

TABLE 2. Prediction accuracy under different prediction Interval.

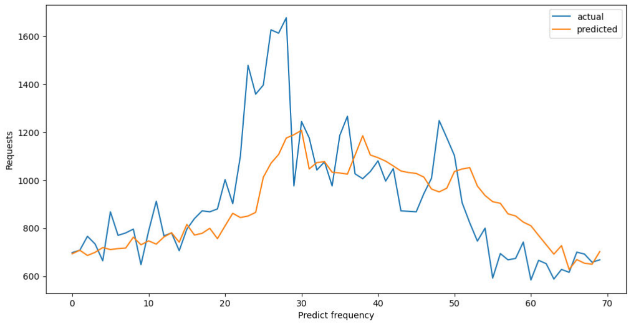
Interval	15s	30s	60s	120s
MSE	**30785.8643**	38586.3142	54327.2857	75225.7222
RMSE	**175.4590**	196.4340	233.0821	274.2730
MAE	**126.85**	138.2571	165.2285	205.0556
R^2	**0.5706**	0.4191	0.0844	-0.3602

adjustment would be meaningless. For example, Kubernetes' HPA mechanism suggests a minimum scaling period of 30 seconds. For autoscalers using prediction algorithms, we adopt a method of multi-step prediction. In this case, the autoscaler can avoid the impact of delayed resource configuration caused by program startup by starting the program in advance. Similarly, considering program startup time, we believe the prediction cycle should be at least 15 seconds. We will evaluate LSTM's multi-step prediction performance on the dataset shown in Fig. 4 under prediction cycles of 15s, 30s, 60s, and 120s in experiments.

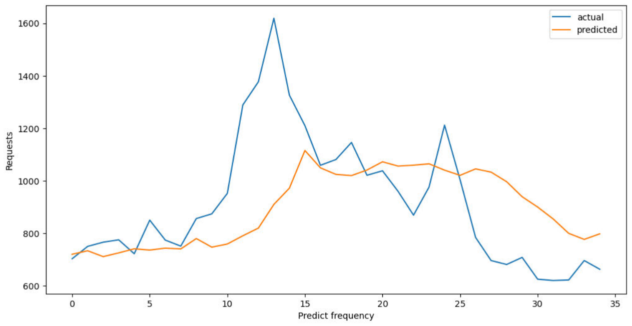
Table 2 shows that the shorter the prediction cycle, the smaller the error between the predicted and actual values, and the higher the accuracy of the LSTM model's prediction. However, this does not necessarily mean that a shorter prediction cycle is more suitable for the autoscaler. The comparison in Fig. 4(a), Fig. 4(b), Fig. 4(c), and Fig. 4(d) illustrates that as the prediction cycle shortens, the prediction model can detect peak periods more quickly, and the prediction of peak values becomes more accurate. However, the shorter prediction cycle makes the prediction model more sensitive to fluctuations in data within a unit time, resulting in more frequent fluctuations in prediction results. This may cause containers to be frequently scaled within a short period. To avoid this container oscillation, many autoscalers use a cooldown mechanism, which means stopping container scaling for a period after significant container scaling. In cases of shorter prediction cycles, the cooldown mechanism may prevent containers from scaling up to the expected scale. Due to the delay in prediction results caused by program startup time, regardless of the prediction cycle, the model's prediction of peak values always has a significant error compared to the actual peak values. Moreover, shorter prediction cycles are not compatible with trend detection algorithms. In our approach, the prediction algorithm forecasts the number of requests two time steps ahead. After obtaining the prediction results, there is a period of time before the actual scaling takes place, which is essentially equal to the scaling period minus the time required for program startup. Trend detection algorithms precisely detect the trend of request changes during this period. With shorter scaling periods, the trend detection algorithm collects fewer samples, leading to lower confidence in trend detection. Therefore, in this paper, we choose a prediction cycle of 60 seconds, which can avoid unnecessary container scaling, leaving the handling of peak periods to the trend detection algorithm. However, our autoscaler and trend detection algorithm also support modification of the prediction cycle.



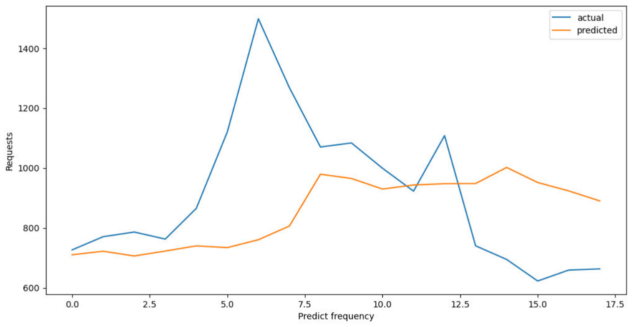
(a) Predicted results for a prediction cycle of 15 seconds.



(b) Predicted results for a prediction cycle of 30 seconds.



(c) Predicted results for a prediction cycle of 60 seconds.



(d) Predicted results for a prediction cycle of 120 seconds.

FIGURE 4. Comparison of predicted results versus actual results under different prediction cycles.

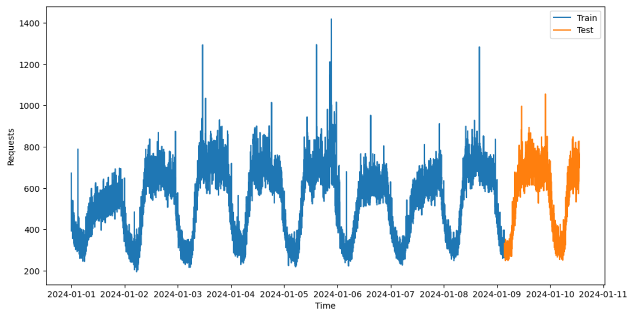


FIGURE 5. Campus network authentication interface dataset.

TABLE 3. Prediction accuracy of ARIMA, LSTM, and Informer models on dataset.

Metric	ARIMA	LSTM	Informer
MSE	2544.4921	2559.6882	**2521.4544**
RMSE	50.4429	50.5933	**50.2141**
MAE	39.0721	39.2950	**38.6366**
R^2	0.9079	0.9074	**0.9088**

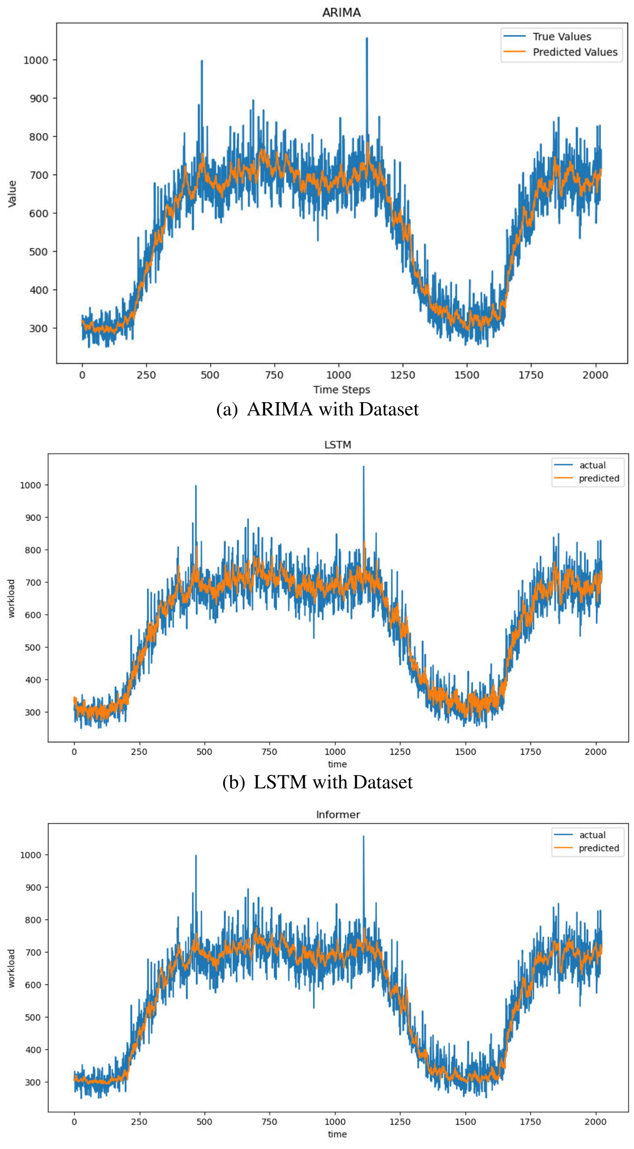
F. EVALUATION OF PREDICTIVE ALGORITHMS

Our autoscaler requires the support of prediction algorithms. To select the appropriate prediction algorithm, we tested three different models: ARIMA, LSTM, and Informer.

In our experiments, the results of the three prediction models are based on a forecast of two steps ahead. We aim for the prediction model in our auto-scaler to accurately forecast values for multiple future time steps. This not only aids in assessing data trends but also allows the auto-scaler to initiate pod scaling in advance, reducing or even eliminating the waiting time for pods due to Readiness probe. The sliding window size for both LSTM and Informer models in the experiment is set to 10. Figure 6 illustrates the comparison between the predicted results of ARIMA, LSTM, and Informer models and the actual data.

The models are trained and predicted using normalized and standardized data. Since the actual values needed are the de-normalized data, we calculate metrics using the de-normalized results. As shown in Table 3, the RMSE and MAE of the three models are very close, indicating that the accuracy of the three models is essentially the same on this dataset. Due to the close performance of the models and considering the training time and prediction speed, we choose LSTM as the predictive model inside the auto-scaler.

The R^2 values of these models are approximately 0.90, indicating that the prediction models can effectively capture features such as seasonality and trends in the data. This experiment reveals that for network request data with evident seasonality and trends, all three prediction models perform well in fitting and forecasting future data. However, real-world network conditions are dynamic and may involve sudden, irregular network fluctuations. From the above graphs, it is evident that the predictive results of the three models are not satisfactory under such conditions. This suggests that in real-world scenarios, where high concurrency situations frequently occur, conventional auto-scalers based on predictive algorithms may struggle to cope with such unpredictable fluctuations.



(a) ARIMA with Dataset

(b) LSTM with Dataset

(c) Informer with Dataset

FIGURE 6. Comparison of actual data with predictions from ARIMA(a), LSTM(b), and Informer(c) models.

G. SIMULATION EXPERIMENTS ON KUBERNETES

1) EXPERIMENT ON REAL DATA

In subSection IV-F, we validated the performance of the prediction models in multi-step forecasting on a dataset. To evaluate the performance of our autoscaler in real-world high-concurrency network environments, we conducted experiments using the peak period dataset represented in Fig. 7. Fig. 8 reflects the approximate trend of this dataset, with the number of requests rapidly increasing and then slowly decreasing. However, this trend is quite one-sided.

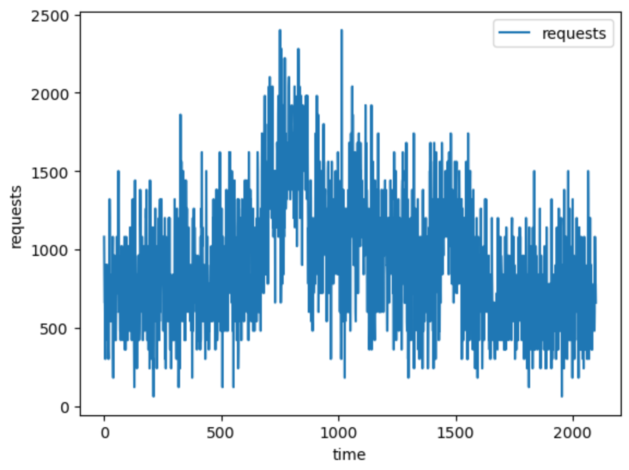


FIGURE 7. Amplified university campus network authentication request data during a 60x traffic peak.

TABLE 4. The Prediction accuracy of ARIMA, LSTM models, and our auto-scaler on real data.

Metric	ARIMA	LSTM	Trend Detection
MSE	63777.5143	**46886.3428**	56883.8
RMSE	252.5421	**216.5325**	238.5032
MAE	185.2857	**152.5143**	180.4286
R2	-0.0700	**0.2134**	0.0457

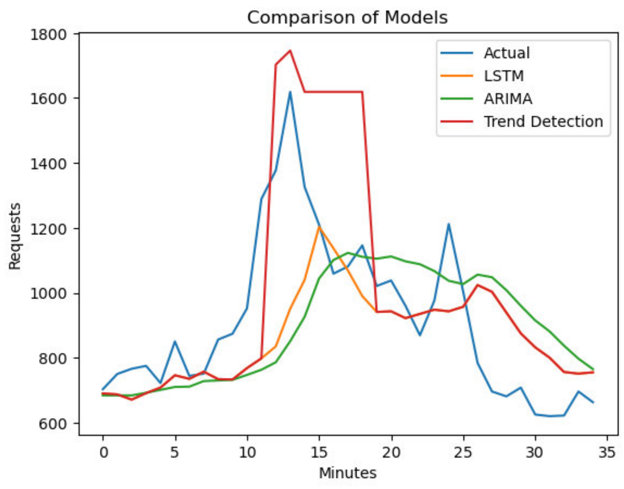


FIGURE 8. Comparison of ARIMA, LSTM, and our auto-scaler's prediction results with actual data.

TABLE 5. Accurate assessment of resource configuration for ARIMA, LSTM, and our auto-scaler during peak access periods.

Metric	LSTM with HPA	ARIMA	Trend Detection
$\Theta_U[\%]$	3.2381	5.4286	**1.7143**
$\Theta_O[\%]$	6.4286	**5.7143**	9.8095

We can observe in Fig. 8 that there is no significant change in the average value per minute during the initial growth of requests. Yet, in Fig. 7, the number of requests per second fluctuates significantly, with many instances where the number of requests exceeds the average value per minute by a considerable margin. Our trend detection algorithm aims to extract as much information as possible from these special moments.

We compared our autoscaler with traditional proactive and hybrid autoscalers in experimental setups. The proactive autoscaler scales based on ARIMA model predictions of future request counts, while the hybrid autoscaler combines LSTM with HPA. This hybrid approach uses LSTM to forecast requests, calculates the required workload, and scales accordingly. Simultaneously, it monitors the resource utilization of these containers every 30 seconds, scaling when the utilization exceeds a threshold.

All three scalers predict the average HTTP request count per second for the next minute at the end of each minute. Thus, we first compare the three scaling methods' prediction results for requests. Figure 8 illustrates the comparison between the predictions of our method, trained LSTM, and ARIMA models with actual data in subsection IV-B. Table 4 provides performance evaluations of their predictions. Obviously, these algorithms show poorer prediction accuracy during peak access periods. The RMSE, MSE, and MAE results for LSTM and ARIMA are significantly higher than those obtained on regular datasets. While these metrics suggest LSTM performs better, Figure 8 shows our autoscaler's predicted trend aligns more consistently with actual data trends. Furthermore, in real-world scenarios, prediction accuracy is not the sole determinant of autoscaler performance.

The evaluation of the auto-scaler's performance is mainly based on the comprehensive assessment of configuration accuracy and service quality. This requires us to examine the auto-scaler in a real-world environment. We deploy the web application on a Kubernetes cluster, and a stress testing program sends HTTP requests to it, with the request rate per second matching the data in the dataset. Figure 9 illustrates the comparison between the pod changes using different scalers and the actual required pod quantity in real-world scenarios. Table 5 presents the configuration accuracy results calculated according to the Equation (8) and (9). We can observe that the proactive autoscaler has the lowest Θ_O but the highest Θ_U. On the other hand, the autoscaler combining proactive and reactive methods has relatively low values for both Θ_U and Θ_O. Meanwhile, the trend-based autoscaler has the lowest Θ_U value. This indicates that in high-concurrency scenarios, our autoscaler is better at avoiding resource shortages and does not lead to excessive resource waste.

In high-concurrency scenarios, our primary concern is whether the service quality of the web application is ensured during peak periods. Given the relatively small concurrency levels, we focus on the program's performance, specifically the average response time for all requests within one minute.

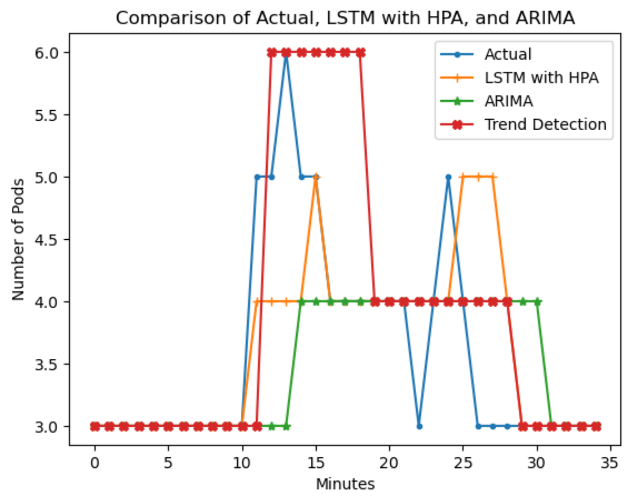


FIGURE 9. Scaling processes of pods controlled by ARIMA, LSTM, and our auto-scaler during peak access periods.

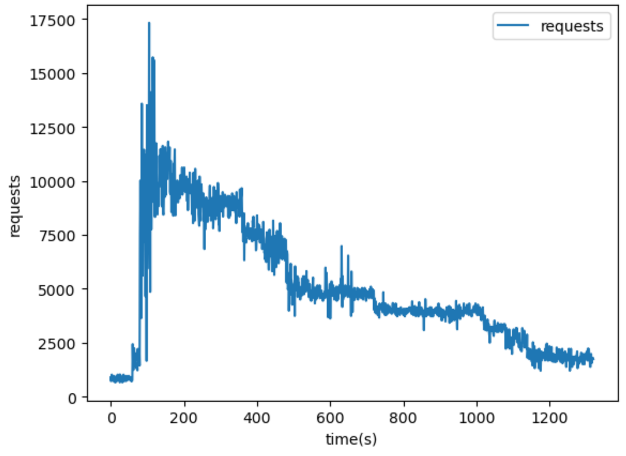


FIGURE 11. Simulated data for requests per second during peak periods.

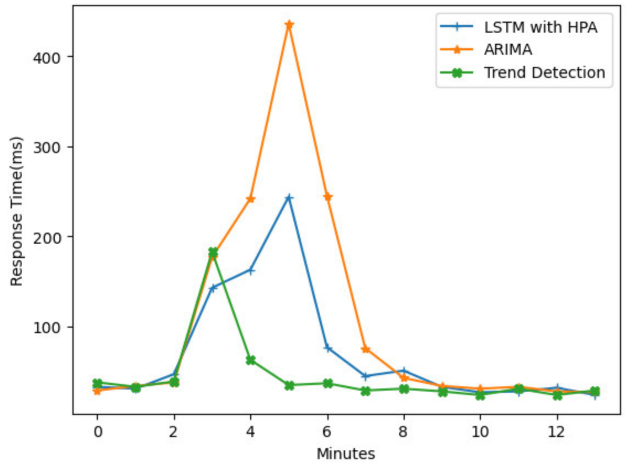


FIGURE 10. Response time of pod requests during peak usage: ARIMA, LSTM, HPA, and our auto-scaler.

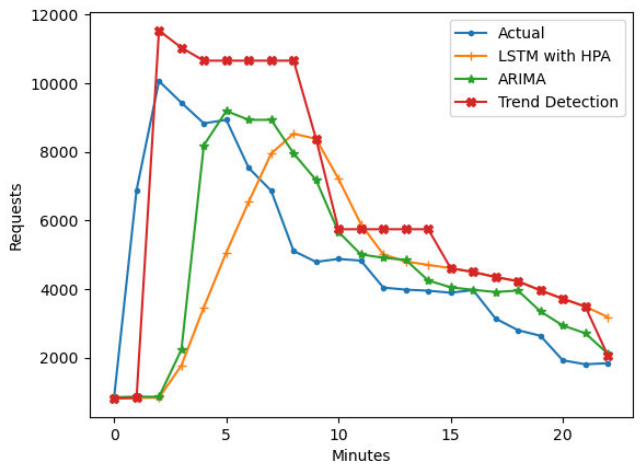


FIGURE 12. Comparison of ARIMA, LSTM, and our auto-scaler prediction results with actual data during simulated 10-fold peak access.

Figure 10 shows the trend in the average response time for the web application using different auto-scalers from the eighth minute to the twenty-first minute of the experiment. When the number of requests increases significantly, the response time of the web service also increases. Among these auto-scalers, only our auto-scaler identifies the rising trend at the moment of a sudden increase in requests and proactively initiates pods, thereby maintaining normal web service response times during subsequent peak periods. In contrast, other auto-scalers do not allocate sufficient resources to the application during this period, resulting in poorer performance.

2) EXPERIMENT ON SIMULATION DATA

Due to the relatively small user base of the school, our collection of peak data is limited. In the real world, external factors such as promotions and hot events may lead to a significant increase in the number of users of a network application within a short period. We need to evaluate whether our autoscaler can still work effectively in such scenarios. We simulated data where the number of requests rises rapidly during peak periods and then declines slowly as experimental data. In the section IV-D, we describe how the data was simulated.

We continue to conduct experiments using the three autoscalers mentioned in subsection IV-G1. We sends requests to the web application based on this simulated data. Fig. 12 shows the per-minute predicted data from different auto-scalers and the simulated request data. Under a tenfold increase in request data, LSTM, which performed well in subsection IV-G1, exhibited the poorest performance here. When the request count rapidly increased, LSTM took the longest time to achieve predictions equal to or greater than the actual values. On the other hand, despite the larger error in our model, its predicted trend closely mirrored the actual trend.

TABLE 6. Prediction accuracy of ARIMA, LSTM, and our auto-scaler under 10 times the peak access scenario.

Metric	ARIMA	LSTM	Trend Detection
MSE	8698396.8696	11904574.2609	**6025037.6957**
RMSE	3450.3006	2949.3045	**2454.5952**
MAE	**1740.9565**	2489.6521	1944.9130
R2	-0.2901	-0.7657	**0.1637**

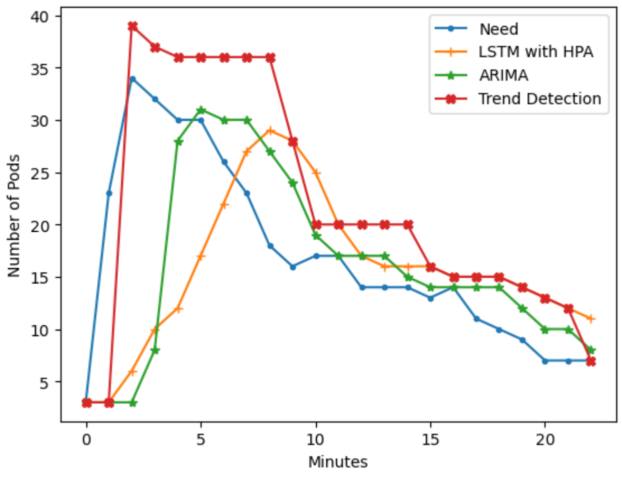


FIGURE 13. Scaling processes of pods controlled by ARIMA, LSTM, and our auto-scaler under 10 times the peak access scenario.

TABLE 7. Accurate assessment of resource configuration for ARIMA, LSTM, and our auto-scaler under 10 times the peak access scenario.

Metric	LSTM with HPA	ARIMA	Trend Detection
$\Theta_U[\%]$	15.5120	11.2956	**3.7807**
$\Theta_O[\%]$	28.4623	**18.2267**	36.2374

Table 6 displays the performance metric evaluations for the predictions made by these auto-scalers in the experiment. The errors for all three auto-scalers are significant. Our model, however, achieved lower values in terms of MSE, RMSE, and MAE, indicating that our model has relatively smaller prediction errors compared to the actual data.

Similar to subsection IV-G1, we conducted container scaling experiments on Kubernetes using three types of autoscalers. Fig. 13 depicts the pod scaling process managed by different auto-scalers during the experiment. When the number of requests increases significantly, requiring a significant increase in the number of pods, only the trend-based autoscaler and the autoscaler based on the hybrid method can respond quickly to the changes in the number of requests. However, in the hybrid method, the reactive method based on thresholds expands the number of pods less and slower due to the limitations of the thresholds. It cannot meet the user demands during peak request periods and still leads to resource shortages.

Table 7 displays the situations of insufficient and excess resource supply for each auto-scaler in the experiment. Although our auto-scaler provides additional resources during peak periods, it effectively avoids situations of insufficient supply. Fig. 14 indicates that under a tenfold increase in request volume, autoscalers based on the hybrid method and ARIMA do not rapidly increase the number of pods. This leads to average response times of over 4000ms during peak request periods for applications managed by these autoscalers. In contrast, the autoscaler based on our trend detection algorithm quickly increases the number of pods after detecting abnormal changes in network request volume. So applications managed by our autoscaler achieve average response times during peak periods that are close to normal response times. Our trend detection algorithm brings stronger performance to applications during peak request periods.

Both Fig. 14 and table 7 demonstrate that the autoscaler based on trend detection algorithms can reduce the time of resource undersupply during peak periods. Resource undersupply not only affects program performance but also reduces program availability. During peak request periods, if insufficient resources are allocated to the program, leading to request processing delays, users who do not receive timely responses may initiate multiple repeated requests. This can result in a backlog of requests within the containers or servers running the program, intense resource competition, and may cause some containers or servers to become unavailable or crash. To evaluate the impact of such scenarios on the availability of the web application, we integrated an Actuator program into the test program and activated the Readiness probe provided by Kubernetes. Through this probe, we monitor whether pods are running normally by requesting interfaces provided by the Actuator program. During peak request periods, when the probe does not receive a response or receives an abnormal response from a pod, that pod is considered unavailable. We adopt a restart strategy for unavailable pods, and any requests that the pod was processing or about to process will return with exceptional responses. By monitoring pod restarts, we can directly assess the impact of sudden increases in requests on program availability.

Fig. 15 illustrates the response variation of the web application managed by different scalers after enabling the Readiness probe. Comparing Fig. 14 and Fig. 15, it can be concluded that the peak response time in Fig. 15 is higher than that shown in Fig. 14, because the crash of pods leads to a reduction in the number of normally functioning pods. The scaler also fails to timely expand pods. In reality, this situation may result in users repeatedly initiating requests, potentially causing a complete service crash.

We still use the data from Fig. 11 to evaluate program availability. Fig. 15 shows the change in response time of web applications managed by different autoscalers after enabling readiness probes. Compared to Fig. 14, at the beginning of the peak request period, the average response times of web applications managed by each autoscaler in Fig. 15 are higher than those at the same time in Fig. 14. This indicates

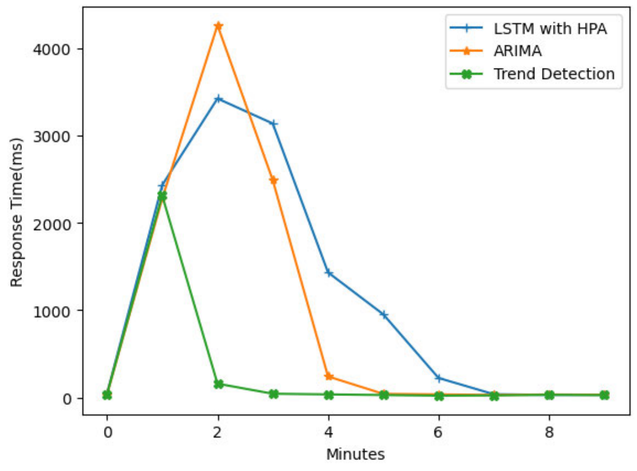


FIGURE 14. Response times of web applications controlled by ARIMA, LSTM, and our auto-scaler under 10 times the peak access scenario.

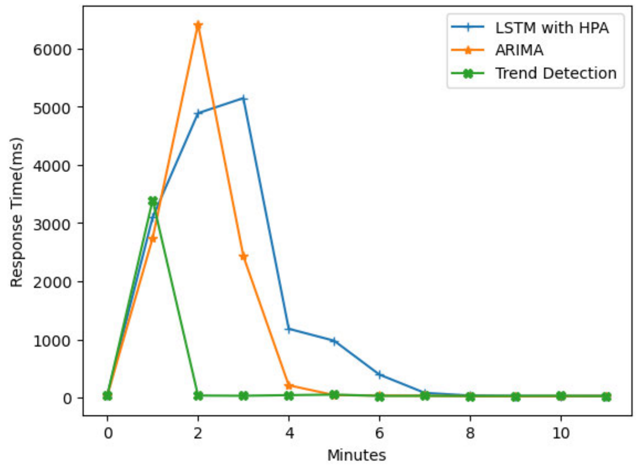


FIGURE 15. Program response time controlled by ARIMA, LSTM, and our auto-scaler in the event of program crashes and restarts.

that pods managed by all autoscalers in this experiment experienced varying degrees of service abnormalities and restarts. When a pod is deemed unavailable, it will be restarted, and all requests being processed or waiting to be processed by the pod will receive abnormal response content. We can determine the occurrence of pod service anomalies at a certain time by monitoring the time of occurrence of abnormal responses and the number of abnormal response occurrences. Fig. 16 shows the percentage of abnormal responses per minute in the web application during this experiment. Programs managed by the trend-based autoscaler only experience a small number of abnormal responses at the beginning of the peak request period, while programs managed by other autoscalers experience more occurrences of abnormal response moments and more abnormal responses than those managed by the trend-based autoscaler. This demonstrates that it can bring stronger availability to programs during peak request periods.

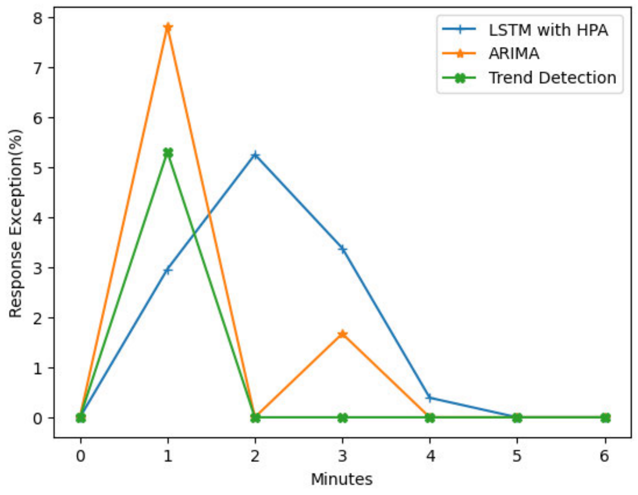


FIGURE 16. The abnormal response ratios of pods controlled by ARIMA, LSTM, and our auto-scaler in extreme scenarios.

V. CONCLUSION

Currently, many cloud service providers like Amazon and Alibaba are exploring better scaling methods for users. However, in high concurrency and frequently fluctuating load scenarios, these methods struggle to quickly respond to changes in user traffic. Experiments show that our proactive auto-scaling engine, combined with a trend detection module, exhibits higher flexibility and efficiency in cases of rapid load increases, reducing the risk of resource shortages and program crashes. However, the resource allocation strategy of our auto-scaler carries the risk of resource over-provisioning, and it also lacks consideration for downward load peaks. In the future, we need to improve the trend detection algorithm to more quickly and accurately identify a wider variety of trend changes. Finally, our auto-scaling algorithm lacks optimization for the currently widespread microservice applications. It lacks unified management of microservice systems, which will be the focus of our future work.

ACKNOWLEDGMENT

This work is supported by National Natural Science Foundation of China (No. 61572434) and Shanghai Science and Technology Committee (No. 19DZ2204800).

REFERENCES

[1] *Horizontal Pod Autoscaling Kubernetes*. Accessed: Dec. 17, 2023. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[2] J. Dobies and J. Wood, *Kubernetes Operators: Automating the Container Orchestration Platform*. Sebastopol, CA, USA: O'Reilly Media, 2020.

[3] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2015, pp. 171–172.

[4] N. Zhou, H. Zhou, and D. Hoppe, "Containerization for high performance computing systems: Survey and prospects," *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 2722–2740, Apr. 2023.

[5] S. Risco, G. Moltó, D. M. Naranjo, and I. Blanquer, "Serverless workflows for containerised applications in the cloud continuum," *J. Grid Comput.*, vol. 19, no. 3, p. 118, Sep. 2021.

[6] N. Marie-Magdelaine and T. Ahmed, "Proactive autoscaling for cloud-native applications using machine learning," in *Proc. GLOBECOM IEEE Global Commun. Conf.*, Dec. 2020, pp. 1–7.

[7] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Machine learning-based scaling management for kubernetes edge clusters," *IEEE Trans. Netw. Service Manage.*, vol. 18, no. 1, pp. 958–972, Mar. 2021.

[8] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in cloud computing: State of the art and research challenges," *IEEE Trans. Services Comput.*, vol. 11, no. 2, pp. 430–447, Mar. 2018.

[9] N. Mungoli, "Scalable, distributed AI frameworks: Leveraging cloud computing for enhanced deep learning performance and efficiency," 2023, *arXiv:2304.13738*.

[10] M.-N. Tran, D.-D. Vu, and Y. Kim, "A survey of autoscaling in kubernetes," in *Proc. 13th Int. Conf. Ubiquitous Future Netw. (ICUFN)*, Jul. 2022, pp. 263–265.

[11] M. Filho, E. Pimentel, W. Pereira, P. H. M. Maia, and M. I. Cortés, "Self-adaptive microservice-based systems–landscape and research opportunities," in *Proc. Int. Symp. Softw. Eng. Adapt. Self-Managing Syst. (SEAMS)*, May 2021, pp. 167–178.

[12] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of Docker containers with ELASTICDOCKER," in *Proc. IEEE 10th Int. Conf. Cloud Comput. (CLOUD)*, Jun. 2017, pp. 472–479.

[13] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes," in *Proc. IEEE 12th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2019, pp. 33–40.

[14] A. Goli, N. Mahmoudi, H. Khazaei, and O. Ardakanian, "A holistic machine learning-based autoscaling approach for microservice applications," in *Proc. 11th Int. Conf. Cloud Comput. Services Sci.*, 2021, pp. 190–198.

[15] W. Iqbal, A. Erradi, and A. Mahmood, "Dynamic workload patterns prediction for proactive auto-scaling of web applications," *J. Netw. Comput. Appl.*, vol. 124, pp. 94–107, Dec. 2018.

[16] M. Imdoukh, I. Ahmad, and M. G. Alfailakawi, "Machine learning-based auto-scaling for containerized applications," *Neural Comput. Appl.*, vol. 32, no. 13, pp. 9745–9760, Jul. 2020.

[17] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Adaptive AI-based auto-scaling for kubernetes," in *Proc. 20th IEEE/ACM Int. Symp. Cluster, Cloud Internet Comput. (CCGRID)*, May 2020, pp. 599–608, doi: 10.1109/CCGrid49817.2020.00-33.

[18] H. Zhou, S. Zhang, J. Peng, S. Zhang, J. Li, H. Xiong, and W. Zhang, "InFormer: Beyond efficient transformer for long sequence time-series forecasting," in *Proc. AAAI*, 2021, vol. 35, no. 12, pp. 11106–11115.

[19] H. Ahmad, "Smart HPA: A resource-efficient horizontal pod auto-scaler for microservice architectures," 2024, *arXiv:2403.07909*.

[20] L. M. Al Qassem, T. Stouraitis, E. Damiani, and I. A. M. Elfadel, "Proactive random-forest autoscaler for microservice resource allocation," *IEEE Access*, vol. 11, pp. 2570–2585, 2023.

[21] A. A. Khaleq and I. Ra, "Intelligent autoscaling of microservices in the cloud for real-time applications," *IEEE Access*, vol. 9, pp. 35464–35476, 2021.

[22] J. Santos, T. Wauters, B. Volckaert, and F. D. Turck, "Gym-hpa: Efficient auto-scaling via reinforcement learning for complex microservice-based applications in Kubernetes," in *Proc. NOMS IEEE/IFIP Netw. Oper. Manage. Symp.*, May 2023, pp. 1–9.

[23] M. P. Yadav, G. Raj, H. A. Akarte, and D. K. Yadav, "Horizontal scaling for containerized application using hybrid approach," *Ingénieri des Systèmes d'Information*, vol. 25, no. 6, pp. 1–10, Dec. 2020.

[24] E. Truyen, D. V. Landuyt, D. Preuveneers, B. Lagaisse, and W. Joosen, "A comprehensive feature comparison study of open-source container orchestration frameworks," *Appl. Sci.*, vol. 9, no. 5, p. 931, Mar. 2019, doi: 10.3390/app9050931.

[25] A. Bauer, J. Grohmann, and N. Herbst, "On the value of service demand estimation for auto-scaling," in *Proc. Int. Conf. Meas., Modelling Eval. Comput. Syst.*, Erlangen, Germany. Cham, Switzerland: Springer, Feb. 2018, pp. 142–156.

[26] N. Jha and R. Popli, "Comparative analysis of web applications using JMeter," *Int. J. Adv. Res. Comput. Sci.*, vol. 8, no. 3, p. 774, 2017.

[27] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Dynamic multi metric thresholds for scaling applications using reinforcement learning," *IEEE Trans. Cloud Comput.*, vol. 11, no. 2, pp. 1807–1821, Apr./Jun. 2023, doi: 10.1109/TCC2022.3163357.

[28] J. Liu, S. Zhang, Q. Wang, and J. Wei, "Coordinating fast concurrency adapting with autoscaling for SLO-oriented Web applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 3349–3362, Dec. 2022.

HAIPENG LIU was born in Hunan, China, in 2000. He is currently pursuing the master's degree with Shanghai University. His main research interests include cloud computing, distributed systems, and elastic computing.

WENHAO ZHU was born in 1979. He received the bachelor's, master's, and Ph.D. degrees from Zhejiang University, in 2002, 2006, and 2009, respectively. From 2012 to 2013, he was a Visiting Scholar with the Computer Laboratory, University of Cambridge, for one year. He is currently a Professor with the School of Computer Engineering and Science, Shanghai University, China. His research interests include text representation, information extraction, and web data mining.

SIYI FU is currently pursuing the master's degree with Shanghai University. His main research interests include blockchain technology, high concurrency, and high throughput research.

YONGJUN LU received the master's degree from Tongji University. He is currently an Engineer, focusing primarily on research in information management.