

## RESEARCH ARTICLE

# A Dynamic Interval Auto-Scaling Optimization Method Based on Informer Time Series Prediction

YU DING<sup>1</sup>, CHENHAO LI<sup>1</sup>, ZHENGONG CAI<sup>1</sup>, XINGHAO WANG<sup>2</sup>, AND BOWEI YANG<sup>3</sup><sup>1</sup>College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang 310027, China<sup>2</sup>School of Software Technology, Zhejiang University, Ningbo, Zhejiang 315048, China<sup>3</sup>School of Aeronautics and Astronautics, Zhejiang University, Hangzhou, Zhejiang 310012, China

Corresponding author: Zhengong Cai (cstcaizg@zju.edu.cn)

**ABSTRACT** With the rapid development and application of container cloud computing-related technologies, more and more applications are being deployed to container cloud clusters. As an essential feature of container cloud platforms and cloud-native architecture, auto-scaling aims to automatically and quickly adjust the allocation of cloud resources according to the resource requirements of applications. Currently, widely used responsive auto-scaling methods, such as Kubernetes HPA, exhibit certain lags due to the startup time costs of containers and Pods. This lag makes it difficult to guarantee the service quality of applications when there is a sudden increase in online application load. This paper proposes a dynamic interval auto-scaling optimization method based on Informer time series prediction. By predicting online application load and dynamically determining the auto-scaling interval, sufficient resources are allocated to the application in advance. In the experiments conducted on the official World Cup forum load and Alibaba cluster CPU load, the Informer time series prediction algorithm demonstrated better long-sequence time series prediction capabilities compared to algorithms such as LSTM and RNN. In elastic scaling experiments, compared to Kubernetes HPA, the method proposed in this paper reduces the average application response time from 0.821 seconds to 0.692 seconds, and the SLA violation rate decreases from 18.277% to 9.157%. This indicates a significant improvement in the service quality metrics of online applications. Furthermore, the proposed method effectively maintains a balance between high CPU resource utilization and low application response time and SLA violation rate, which is something RNN-based elastic scaling method cannot achieve, as it can only reduce application response time and SLA violation rate by sacrificing CPU resource utilization.

**INDEX TERMS** Auto-scaling, container cloud, dynamic interval, informer, time series prediction.


## I. INTRODUCTION

### A. CONTAINER CLOUD

On August 9, 2006, Google CEO Eric Schmidt introduced the Cloud Computing concept at the Search Engine Strategies (SES San Jose 2006) conference. However, the idea of virtualization, the core of cloud computing, was formally proposed in a paper published by Christopher Strachey in 1965. This marked the earliest origin of cloud computing. Since then, cloud computing technologies have

developed rapidly. In 2013, Matt Stine of Pivotal first proposed the concept of cloud-native. Cloud computing has shifted towards containerization, automated management, and microservices-oriented architecture [1].

Containerization is a type of virtualization technology that is more lightweight and portable than traditional virtual machines. The open-source application container engine Docker has been widely adopted [2]. As containerization technology has advanced, container orchestration technology has become a key research focus for major cloud platform providers and the academic community. This technology enables efficient management operations for large-scale

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

clusters, including container creation, deletion, and scheduling. Docker Inc. proposed Docker Swarm, which is highly compatible with Docker [3]. Mesosphere's Marathon container orchestration system was also a strong competitor to Swarm [4]. However, the container orchestration system that ultimately gained widespread adoption in industry and academia is Google's open-source Kubernetes [5]. The prototype of Kubernetes is derived from Google's internal large-scale cluster management system, Borg. Container cloud technology, based on containerization technology and container orchestration engines, has gradually become mainstream in cloud computing.

## B. AUTO-SCALING

Unlike traditional virtual machines, containerization technology based on container engines like Docker does not require installing a new operating system. It only needs to include the necessary environment dependencies. This lightweight characteristic effectively saves resources, and Docker containers can rapidly deploy within seconds, offering better portability.

Auto-scaling is an essential feature in container cloud platforms and cloud-native architectures. It refers to the automatic and rapid adjustment of resources based on the resource demands and monitoring metrics of deployed services. When application resource demands increase, cloud computing resources are scaled up, and when resource demands decrease, excess cluster resources are reclaimed. Container cloud platforms leverage modules like Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA). These modules are used in container orchestration engines such as Kubernetes. They help achieve elastic scaling of containers, meeting the resource needs of various applications.

The elastic scaling of containers can be abstracted as a MAPE loop in automatic control problems, which stands for Monitoring, Analysis, Planning, and Execution [6]. The monitoring phase involves collecting relevant metrics to make scaling decisions. During the analysis phase, algorithms determine or predict current or future resource usage to make scaling decisions. The planning phase calculates and determines the specific resources needed for scaling up or down. The execution phase carries out the scaling operations based on the results of the analysis and planning phases.

The industry's most widely used scaling method is rule-based reactive auto-scaling. This method analyzes and calculates the required resources for scaling up or down only after actual resource changes, leading to latency issues. Due to the startup and shutdown costs associated with Pods or containers, scaling operations executed after a sudden increase in load may fail. These operations might not meet the suddenly increased resource demands of applications. This can potentially lead to a decline in service quality.

Additionally, frequent elastic scaling operations can cause window jitter, which can affect the overall stability of the cloud computing system. For example, the most widely used HPA in Kubernetes makes scaling decisions based on rules

and employs sliding time windows and tolerance strategies to prevent window jitter. Setting thresholds in this method requires extensive professional knowledge. It also demands experience with the deployed applications and workload trends. This makes the method overly dependent on human expertise.

In recent years, proactive auto-scaling methods have gradually gained attention. Proactive scaling methods focus on predicting future workload demands and expanding resources in advance to mitigate performance issues before they occur. For example, traditional machine learning techniques, such as Auto-Regressive Integrated Moving Average (ARIMA) [7], can be used for time-series prediction tasks with relatively simple and linear workload demands. However, when faced with dynamic and complex time-series prediction tasks, deep learning models based on RNNs [8] (such as LSTM [9] and Bi-LSTM [10]) can be employed. These models can learn long-term dependencies and selectively retain or forget information as needed. Compared to reactive auto-scaling methods like Kubernetes HPA, the aforementioned proactive auto-scaling approaches enable containerized applications to optimize resource utilization, enhance performance, and reduce costs. Although this method inevitably introduces issues such as lower timeliness, there has been considerable academic work around it in recent times. For example, in a specific study [11], the authors proposed a proactive auto-scaling method named HANSEL, which utilizes a bidirectional long short-term memory (Bi-LSTM) model with an attention mechanism to accurately predict the load of microservices and employs reinforcement learning to achieve proactive elastic scaling. HANSEL significantly improved SLA in edge computing scenarios, increasing system resource utilization by approximately 20%.

The subsequent organization of this paper is as follows: Chapter II introduces related work on elastic scaling optimization; Chapter III presents the proposed dynamic interval elastic scaling optimization method based on Informer time series prediction; Chapter IV covers experimental validation; and Chapter V provides the conclusion.

## C. CONTRIBUTIONS

This paper proposes an optimized method for elastic scaling based on time prediction and dynamic scaling intervals. The technique utilizes an improved Informer [12] algorithm based on the Transformer [13] architecture for time series prediction of container cloud application loads. This approach addresses the latency issues of traditional rule-based reactive auto-scaling by making scaling decisions before changes in application loads and resource demands occur. It prevents a decline in application service quality caused by the inability to meet sudden increases in resource demands.

The method also introduces an optimization approach for dynamic scaling intervals by leveraging the Informer algorithm's superior predictive capability for long-time series. This effectively reduces the frequency of scaling

operations, avoiding the window jitter problem caused by frequent scaling and ensuring the stability of the container cloud cluster. Furthermore, the method employs an “all-at-once” approach for scaling up. It uses a stepped approach for scaling down. This effectively meets application resource demands. It also avoids instability in application service quality due to rapid downscaling. This aligns with an excellent scaling system’s “scale fast, scale slow” strategy.

Through experiments, we have demonstrated that the Informer time series prediction algorithm outperforms LSTM and RNN, showcasing its superior performance in long-sequence time series forecasting. Additionally, in the elastic scaling experiments, compared to Kubernetes HPA, the proposed method reduced the average application response time from 0.821 seconds to 0.692 seconds, and the SLA violation rate decreased from 18.277% to 9.157%. These results indicate that the proposed method significantly improves the service quality of online applications. At the same time, the method effectively balances high CPU resource utilization, low application response time, and a low SLA violation rate, whereas the RNN-based elastic scaling method can only reduce response time and SLA violation rate at the expense of CPU resource utilization, failing to achieve such a balance.

## II. RELATED WORK

Elastic scaling algorithms can be divided into two categories based on the execution time: reactive and predictive scaling. Rule-based auto-scaling is currently the most widely used method in the industry due to its low computational cost, simplicity, and strong reliability. For instance, the HPA in Kubernetes is a rule-based reactive method. The HPA controller in Kubernetes uses metrics data collected by the Metrics Server from Pod resource objects to make scaling decisions. It calculates the desired number of Pod replicas based on the metrics data. Suppose the current number of Pod replicas does not match the calculated desired number. In that case, it recalculates the number of replicas based on the ratio of the current metrics value to the target metrics value.

In addition to Kubernetes HPA, many other rule-based scaling methods have been proposed in the industry. For example, RightScale adopts an autonomous voting process where scaling actions are only executed when most virtual machines decide to do so [14]. After each scaling operation, a cooldown period follows. Taherizadeh and Stankovski implemented a metrics monitoring system called SWITCH. They proposed new rules for automatic virtual machine scaling. This achieved better results in terms of service quality assurance and cost control [15]. Beloglazov and Buyya proposed a method for dynamically consolidating virtual machines based on adaptive threshold utilization and determined the probability distribution of CPU utilization in cloud platforms [16].

However, rule-based reactive elastic scaling methods respond slowly to sudden increases or decreases in resource demands for online applications. This can lead to a decline

in application service quality. It can also result in resource wastage. Setting thresholds in rule-based methods often requires extensive prior knowledge of the deployed applications. This includes understanding resource demands and load change trends. As a result, these methods are overly dependent on manual operations and business experience.

Relative to reactive scaling methods, predictive scaling methods based on time series prediction offer an advantage. They can perform scaling operations before actual load changes occur. This helps address the latency issue. Time series prediction algorithms can be divided into statistical and machine learning-based categories. Statistical time series prediction algorithms include moving average algorithms, autoregressive moving average (ARMA), and autoregressive integrated moving average (ARIMA) models. Machine learning-based time series prediction algorithms include ensemble learning models like XGBoost [17] and neural network models such as Recurrent Neural Networks (RNN) and Long Short-Term Memory (LSTM) networks.

Zhang et al. combined the ARIMA time series prediction algorithm with the reinforcement learning algorithm A-SARSA for container elasticity optimization. They categorized CPU utilization and response time into different levels. They only increased the number of actions when the response time violated the SLA, and the action in the previous period was already at its maximum. Compared to other reinforcement learning algorithms, their experimental results showed reduced SLA violation rates. However, CPU utilization also decreased correspondingly [18]. Qazi et al. proposed a neural network based on Cartesian genetic programming for resource estimation and a rule-based auto-scaling system for IaaS cloud servers. The resource monitor obtains resource utilization and provides it to the evaluator for practical resource evaluation. However, the final scaling is still based on set rules [19].

Imdough et al. calculated the maximum load a single container could bear. They used LSTM neural networks for load prediction. They also introduced a cooldown timer (CDT) to mitigate oscillation. Nonetheless, the resource estimation phase still adopted a rule-based approach. It calculated the ratio of the predicted value to the container’s maximum load for scaling decisions. This made it challenging to meet sudden resource demand spikes [20]. Ali Yadav Nikraves et al. proposed a cost-driven auto-scaling method considering cloud customers’ cost preferences and used genetic algorithms to configure rule-based systems to minimize scaling costs. However, they still employed rule-based methods like cooldown times and threshold settings [21].

Tang et al. constructed the Fisher load prediction model using bidirectional LSTM (Bi-LSTM) to predict workloads and application throughput, significantly improving accuracy compared to ARIMA and LSTM models [22]. Kumar et al. designed a hybrid associative learning architecture. They combined LSTM and Bi-LSTM models. They also employed a GRU-based time series prediction model. This was done to optimize resource load prediction accuracy and training time.

Their method showed improved computational efficiency and predictive accuracy compared to other models [23].

Taha et al. proposed a proactive auto-scaling method based on deep learning, specifically designed for Service Function Chains (SFC) in cloud computing [24]. This method employs a hybrid model that combines a Multi-Layer Perceptron (MLP) with a Long Short-Term Memory network (LSTM). The model is capable of capturing the long-term dependencies and complex temporal patterns of Virtual Network Functions (VNF) resource demands. Upon receiving the predicted resource demand results, the system dynamically adjusts the resource allocation for VNF (including CPU, memory, and bandwidth) according to the needs of the Service Function Chain, thereby achieving proactive resource scaling, optimizing resource utilization, and enhancing Quality of Service (QoS). However, the proposed method may perform poorly when encountering sudden events or previously unseen load patterns. In such cases, the system might not be able to adjust in time, leading to service interruptions or performance degradation.

Jeong et al. proposed a proactive resource auto-scaling scheme based on SCINet, aimed at providing stable and elastic container resource management for high-performance cloud computing [25]. This method predicts future workloads by applying the Sample Convolution and Interaction Network (SCINet) model with Reversible Instance Normalization (RevIN), generating elastic resource requests. These requests are then efficiently adjusted by HiPerRM's VPA (Vertical Pod Autoscaling) and HPA (Horizontal Pod Autoscaling) to manage Pod resource requests and replica counts. However, the SCINet model and its integrated system possess high complexity, particularly when handling large-scale and diverse workloads. The complexity of the model can lead to higher computational resource consumption, and in scenarios where real-time responsiveness is critical, it may affect the speed of scaling decisions.

The related work mentioned above mostly employs traditional machine learning methods such as LSTM, ARIMA, and XGBoost, or deep learning methods for load prediction, and still adopts fixed interval or cooldown-based scaling strategies. These approaches fail to effectively address the issue of window jitter caused by frequent scaling. Additionally, some studies may reduce response time and SLA violation rates by sacrificing resource utilization, failing to effectively balance the relationship between the two.

In contrast, this paper uses the Informer time series prediction algorithm, which is an improved algorithm based on the Transformer architecture, particularly suitable for long-sequence time series prediction. Moreover, this paper proposes a dynamic interval scaling method, which dynamically adjusts the scaling operation intervals by predicting future load changes, avoiding the drawbacks of fixed interval scaling methods. The scaling operation adopts a "one-time" scaling strategy, i.e., allocating sufficient resources in one step when an increase in resource demand is predicted, and using a stepwise reduction strategy when

the demand decreases to avoid service quality degradation. Finally, the paper experimentally validates the superiority of the Informer algorithm in long-sequence time series prediction and conducts elastic scaling experiments in the simulation environment. The results show that the proposed method achieves a balance between resource utilization and service quality by significantly reducing application response time and SLA violation rates while maintaining high CPU resource utilization.

### III. METHOD

#### A. INFORMER TIME SERIES PREDICTION ALGORITHM

Time series prediction has a wide range of application scenarios in the industry, such as predicting the access volume of cloud services and the resource usage of online applications. By predicting future resource usage, resources can be allocated and deployed in advance. With the rapid development of machine learning technologies, including deep learning, more machine learning-based time series prediction algorithms have been proposed and applied. These include Recurrent Neural Networks (RNN) and Long Short-Term Memory (LSTM).

As the length of the prediction sequence increases, it becomes increasingly challenging to ensure the accuracy of the prediction algorithms. However, predicting more extended time series can guarantee more precise resource allocation and more accurate scaling operations. This helps avoid the window jitter problem caused by frequent scaling. Long-range dependency challenges must be addressed for long sequence time series prediction tasks.

Time series prediction algorithms based on the Transformer architecture have been proposed recently. The Transformer model is a deep learning model based on the self-attention mechanism. It adopts an Encoder-Decoder architecture. After the input data is embedded, it is input into the encoder. After passing through six encoder blocks, an encoding information matrix is obtained. This matrix is then input into the decoder module for self-attention score calculation, followed by the feed-forward neural network module for model training [13].

The self-attention mechanism is a crucial part of the Transformer model. The computation of self-attention requires three matrices: Query ( $Q$ ), Key ( $K$ ), and Value ( $V$ ). These matrices are derived from the input vector  $X$  or the output of the previous encoder block through linear transformation matrices  $W_Q$ ,  $W_K$  and  $W_V$  [26]. The calculation method for self-attention is shown in Equation (1).

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V \quad (1)$$

Among these,  $d$  is the number of columns of the  $Q$  and  $K$  matrices, i.e., the vector dimension. *Softmax* is a normalization function used to convert input scores into a probability distribution. Specifically, the *Softmax* function normalizes the matrix obtained after the  $QK^T$  product, transforming it into a probability distribution where the sum



of all output values equals (1). This probability distribution highlights queries with higher relevance while suppressing those with lower relevance.

Applying the Transformer model to long sequence time series prediction tasks presents several challenges. In the traditional Transformer model, the time and space complexity of the standard dot-product computation in self-attention are both  $O(L^2)$  where  $L$  is the sequence length. This results in slow computation times and excessively high memory usage, affecting model efficiency. Additionally, the step-by-step decoding approach used in the Transformer model's Encoder-Decoder structure leads to longer sequences' prediction times.

The Informer time series prediction algorithm improves upon the Transformer model by introducing the ProbSparse Self-Attention mechanism [12]. This sparse self-attention mechanism only considers dot products that significantly contribute to attention calculations. It ignores other dot products. This reduces the time and space complexity of self-attention calculations to  $O(L \log L)$ . The Informer model uses KL divergence to measure the sparsity of the Query matrix, with the specific calculation formula given in equation (2).

$$M(q_i, K) = \ln \sum_{j=1}^{L_K} e^{\frac{q_i k_j^T}{\sqrt{d}}} - \frac{1}{L_K} \sum_{j=1}^{L_K} \frac{q_i k_j^T}{\sqrt{d}} \quad (2)$$

$q_i$  is the  $i$ -th query matrix,  $K$  is the key matrix, and  $d$  is the vector dimension. Sparsity is measured by subtracting the arithmetic mean from all keys' Log-Sum-Exp (LSE). This sparsity calculation results in the ProbSparse self-attention computation method, as shown in equation (3). What is additionally required is  $\bar{Q}$  refers to the top- $u$  queries sorted based on the sparsity evaluation  $M(q_i, K)$ .

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{\bar{Q}K^T}{\sqrt{d}}\right)V \quad (3)$$

Additionally, the Informer model introduces a generative decoder to produce long sequence outputs. This requires only one forward step. Thus, it avoids error accumulation. Combining these advantages, the Informer model improves the prediction capability for long sequences. This paper innovatively applies the Informer model to optimize elastic scaling methods in container cloud platforms. It leverages the Informer model's higher prediction accuracy for long sequences. The paper proposes a dynamic interval-based elastic scaling optimization method based on Informer time series prediction.

## B. DYNAMIC INTERVAL ELASTIC SCALING METHOD

The rule-based responsive scaling method widely used in the industry currently employs fixed interval scaling, such as the HPA module in Kubernetes. This module uses fixed interval scaling. It collects real-time metric values via the metrics API. It compares these values with the desired metric values to determine whether scaling operations are needed. This

process is configured through the horizontal-pod-autoscaler-sync-period, which defaults to 15 seconds.

This fixed interval scaling method has some issues. For online business deployments on container cloud platforms, there may be continuous load growth during traffic peaks. The startup of Pods and containers requires a certain amount of time and resource consumption. Repeated scaling during traffic peaks may fail to meet rapidly growing resource demands of online businesses. This can lead to a decline in service quality metrics. Additionally, widespread scaling operations can affect the stability of the container cloud platform. The dynamic interval elastic scaling method based on Informer time series prediction proposed in this paper is as follows.

Collect and record historical load time series data of the application through the metrics API, denoted as  $Load_{old} = \{L_1, L_2, L_3, \dots, L_n\}$ . Input the historical load data into the Informer time series prediction algorithm for load prediction. Set the prediction interval length  $d_{pred}$ . Obtain the predicted load data, denoted as  $Load_{pred} = \{L_{n+1}, L_{n+2}, L_{n+3}, \dots, L_{n+d_{pred}}\}$ .

Calculate the threshold for the resource change amount for scaling operations: the threshold for scaling up  $Threshold_{up}$  and the threshold for scaling down  $Threshold_{down}$ . The method for calculating the threshold for scaling up resources,  $Threshold_{up}$ , is shown in Equation (4).

$$Threshold_{up} = \alpha \times Request \times PodNum_{now} \quad (4)$$

$Request$  is the resource quota requested by the application Pod.  $PodNum_{now}$  is the current number of Pods and application replicas. The product of these two values represents the total resources allocated by the container cloud cluster for the application.  $\alpha$  is the proportion of the resource increase relative to the total allocated resources, with a default value of 0.05.

Similarly, the method for calculating the threshold for resource change during scaling down,  $Threshold_{down}$ , is shown in Equation (5).

$$Threshold_{down} = \beta \times Request \times PodNum_{now} \quad (5)$$

$Request$  is the resource quota requested by the application Pod. And  $PodNum_{now}$  is the current number of Pods.  $\beta$  is the proportion of the resource decrease relative to the total allocated resources, with a default value of 0.10. The reason for the different default values of  $\alpha$  and  $\beta$  is caution. For container cloud platforms hosting online applications, scaling down operations should be handled more cautiously than scaling up operations. This is to prevent a reduction in allocated resources from leading to a rise in application load, which could result in a decline in service quality.

Starting from the initial point  $T_{start}$  of the predicted load sequence  $Load_{pred}$ , define the application load at this time as  $L_{start}$ . When the resource increase is more significant than or equal to  $Threshold_{up}$ , it is determined that a scaling-up operation is needed. Continue to search for the predicted load-interval that produces the maximum resource increase,

and define the time point that makes the maximum resource increase as  $T_{end}$ . The application load at this time is defined as  $L_{end}$ . The interval  $[T_{start}, T_{end}]$  is the determined scaling interval. The desired number of Pod replicas can be calculated at this point, as shown in Equation (6).

$$PodNum_{desired} = \left\lceil \frac{L_{end}}{Request \times Target} \right\rceil \quad (6)$$

$PodNum_{desired}$  is the calculated desired number of Pod replicas.  $L_{end}$  is the load value at the time point producing the maximum resource increase.  $Request$  is the resource quota requested by the application.  $Target$  is the target resource utilization rate of the deployed application.

At this point, the final number of Pod replicas needed for scaling up can be determined. This is done by taking the minimum value between  $PodNum_{desired}$  and the maximum number of replicas set for the application,  $maxReplicas$ , as shown in Equation (7).

$$PodNum_{new} = \min \{PodNum_{desired}, maxReplicas\} \quad (7)$$

Once a scaling interval  $[T_{start}, T_{end}]$  has been determined and a scaling operation has been executed, the load data from the current cycle can be added to the historical load data for the next scaling cycle prediction. The updated historical load data will be  $Load_{old} = \{L_1, L_2, L_3, \dots, L_{start}, L_{start+1}, \dots, L_{end}\}$ . The point  $T_{end}$  will serve as the starting point for the next prediction cycle, where a new scaling interval  $[T_{start'}, T_{end'}]$  and the required scaling operations will be determined. If no scaling is needed, no action will be taken, and this process will be repeated.

Similarly, starting from the initial point  $T_{start}$  of the predicted load sequence  $Load_{pred}$ , if the resource decrease is greater than or equal to  $Threshold_{down}$ . Then the algorithm will search for the time point that produces the maximum resource decrease. Determine the scaling-down interval  $[T_{start}, T_{end}]$ . The desired number of Pods,  $PodNum_{desired}$ , can be calculated. Compare this with the minimum number of Pod replicas set for the application. This will give you the required number of replicas for scaling down. However, unlike the scaling-up operation, executing the scaling-down operation in one step might be problematic. It could lead to an inability to meet the application's resource demands. This could result in a decline in service quality. Therefore, online applications' elastic scaling system must handle scaling-down operations more cautiously. This method employs a gradual scaling-down strategy for scaling-down operations.

For the gradual scaling-down strategy, the number of Pod replicas that need to be scaled down,  $PodNum_{scaledown}$ , is first calculated. The calculation method is shown in Equation (8).

$$PodNum_{scaledown} = \left\lceil \frac{L_{start} - L_{end}}{Request \times Target} \right\rceil \quad (8)$$

First, determine the scaling interval  $[T_{start}, T_{end}]$ . Then, determine the number of Pod replicas that need to be scaled down for the current scaling operation. Set the

number of Pod replicas reduced in each step of the gradual scaling-down operation to  $n$ . Then, calculate the time interval for each scaling-down operation,  $\Delta T_{scaledown}$ , as shown in Equation (9).

$$\Delta T_{scaledown} = \frac{T_{end} - T_{start}}{PodNum_{scaledown}} \times n \quad (9)$$

$\Delta T_{scaledown}$  is the time interval for each scaling-down operation.  $n$  is the number of Pod replicas reduced in each scaling-down step, defaulting to 1. And  $PodNum_{scaledown}$  is the number of Pod replicas to be reduced as previously calculated.  $T_{start}$  and  $T_{end}$  are the time points defining the scaling interval. Thus, for every  $\Delta T_{scaledown}$  interval,  $n$  Pod replicas are scaled down.

The overall dynamic interval elastic scaling optimization method based on Informer time series prediction is described in Algorithm 1.

---

**Algorithm 1** Algorithm for Dynamic Interval Scaling Based on Informer Time Series Prediction

---

**Require:** Application historical load  $Load_{old}$ , maximum number of replicas  $maxReplicas$ , minimum number of replicas  $minReplicas$ , target utilization rate  $Target$ ,  $PodRequest$ .

**Ensure:** Scaling decision

- 1: Obtain the historical application load data  $Load_{old} = \{L_1, L_2, L_3, \dots, L_n\}$  monitored by the Metrics Server
- 2: Input the application historical load data into the Informer time series prediction algorithm  $Load_{pred} = \text{Informer}(Load_{old})$ . Obtain the predicted load data  $Load_{pred} = \{L_{n+1}, L_{n+2}, L_{n+3}, \dots, L_{n+d_{pred}}\}$ .
- 3: Calculate the thresholds for resource changes as  $Threshold_{up} = \alpha \times Request \times PodNum_{now}$  for scaling up and  $Threshold_{down} = \beta \times Request \times PodNum_{now}$  for scaling down.
- 4: Iterate through the predicted load data  $Load_{pred}$ . Calculate the resource change amount for each load  $L_i$  relative to the initial load  $L_{start}$  at time  $T_{start}$ . Record and update the maximum resource  $L_{max}$  and minimum resource  $L_{min}$  along with their corresponding time points.
- 5: **IF**  $L_i - L_{start} \geq Threshold_{up}$  **THEN** decide to execute a scaling-up operation, i.e., set  $scale\_decision = 1$ , **ELSE IF**  $L_i - L_{start} \leq -Threshold_{down}$  **THEN** decide to execute a scaling-down operation, i.e., set  $scale\_decision = -1$ .
- 6: **IF**  $scale\_decision = 1$ , **THEN** continue to iterate through the predicted load data  $Load_{pred}$ , obtain the load maximum  $L_{end}$  (i.e., the resource amount that produces the maximum resource increase) and the corresponding time  $T_{end}$ , determine the scaling interval as  $[T_{start}, T_{end}]$ , calculate the desired number of Pod replicas  $PodNum_{desired} = \left\lceil \frac{L_{end}}{Request \times Target} \right\rceil$ , take the minimum value between this and the maximum number of replicas  $maxReplicas$  to obtain the new number of Pod replicas  $PodNum_{new}$ , execute the scaling-up operation,

add the new load data to  $Load_{old}$ , and **GOTO 2** to enter the next cycle.

- 7: **IF**  $scale\_decision = -1$ , **THEN** continue to iterate through the predicted load data  $Load_{pred}$ , obtain the load minimum  $L_{end}$  (i.e., the resource amount that produces the maximum resource decrease) and the corresponding time  $T_{end}$ , determine the scaling interval as  $[T_{start}, T_{end}]$ , calculate the number of Pod replicas to be scaled down  $PodNum_{scaledown} = \left\lfloor \frac{L_{start} - L_{end}}{Request \times Target} \right\rfloor$ , calculate the time interval for each scaling-down operation  $\Delta T_{scaledown} = \frac{T_{end} - T_{start}}{PodNum_{scaledown}} \times n$ , scale down  $n$  Pod replicas every  $\Delta T_{scaledown}$  interval, and if the number of replicas reaches the minimum number of Pod replicas  $minReplicas$ , then stop scaling down early. Add the new load data to  $Load_{old}$  and **GOTO 2** to enter the next cycle.
- 8: **IF**  $scale\_decision = 0$ , **THEN** do not perform any scaling operation, add the new load data to  $Load_{old}$ , and **GOTO 2** to enter the next cycle.

## IV. EXPERIMENT

### A. TIME SERIES PREDICTION

This paper conducts an experiment on time series prediction of the load for online applications on container cloud platforms. The experiment uses the access load data of the official forum server from the 1998 FIFA World Cup (WorldCup98). This load time series data includes one load value per minute and possesses the universal load characteristics for online applications on container cloud platforms. The line chart of the preprocessed load data from the official World Cup forum is shown in Figure 1. Specifically, the vertical axis represents the load values of the CPU cores, while the horizontal axis represents the corresponding time points under that load.

In addition, to thoroughly validate the superiority of the method proposed in this paper, we also used the cluster load data open-sourced by Alibaba Group, which can be obtained at <https://github.com/alibaba/clusterdata>. This dataset contains the load information of various Pods over a period of time, including CPU usage, memory usage, and more. The experiment primarily focuses on the variation of CPU usage over time. As shown in the Figure 2, the horizontal axis represents time, while the vertical axis indicates the corresponding CPU utilization percentage at each time point.

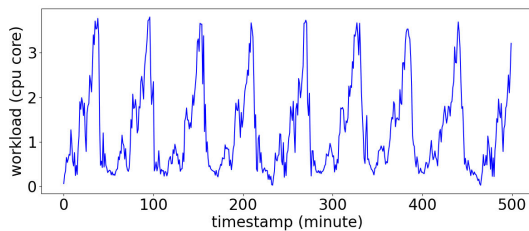


FIGURE 1. Worldcup forum load data line chart.

The load exhibits a periodic pattern with distinct peaks and troughs within each cycle. In the middle to later stages of a cycle, there is a significant load spike. This is followed by

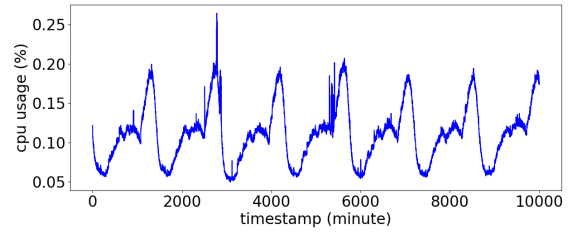


FIGURE 2. Alibaba cluster CPU load data line chart.

a sharp decrease. This pattern aligns with the load patterns of most online applications on container cloud platforms. Many online applications experience a peak load period, making this dataset suitable for experiments with the scaling optimization method.

The time series prediction experiment part compares the widely used time series prediction algorithms ARIMA, RNN, LSTM, and Bi-LSTM with the Informer algorithm, using the access load data of the WorldCup official forum server. The length of the predicted load time series is set to 72. The evaluation metrics used in the time series prediction comparison experiment,  $MAE$ ,  $MSE$ , and  $R^2$ , are calculated as shown in Equations (10), (11), and (12), respectively.

$$MAE = \frac{1}{n} \sum_{i=1}^n |\hat{y}_i - y_i| \quad (10)$$

$$MSE = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2 \quad (11)$$

$$R^2 = 1 - \frac{\sum_{i=1}^n (\hat{y}_i - y_i)^2}{\sum_{i=1}^n (\bar{y}_i - y_i)^2} \quad (12)$$

When performing the time series forecasting task using Informer, the attention mechanism is employed. In our designed experiment, we defined the prediction length as 72 (i.e.,  $L = 72$ ). Since the time series information only includes two components: time and the predicted load indicator,  $d$  is set to 2. For  $Q$ ,  $K$ , and  $V$ , as Informer uses the self-attention mechanism, they are all derived from the linear transformation of the past time series data  $X$ . Assuming we use the past 216 minutes to predict the data for the next 72 minutes,  $X$  would be the matrix composed of the time series data from those 216 minutes.

The time series prediction results of the Informer algorithm compared to ARIMA, RNN, LSTM, and Bi-LSTM algorithms for the WorldCup official forum access load data are shown in Table 1. The Informer algorithm achieves the best prediction performance, with evaluation metrics  $MAE$ ,  $MSE$ , and  $R^2$  being 0.230, 0.104, and 0.883, respectively. Compared to the second-best performing Bi-LSTM, these metrics improve by 11.5%, 31.6%, and 5.4%, respectively. For long sequence load time series prediction, the Informer algorithm demonstrates superior prediction capability. It outperforms LSTM, Bi-LSTM, and other algorithms. This makes it more suitable for scaling optimization.

**TABLE 1.** World-cup official forum load time series prediction results.

Model	MAE	MSE	$R^2$
Informer	0.230	0.104	0.883
Bi-LSTM	0.260	0.152	0.829
LSTM	0.291	0.191	0.785
RNN	0.372	0.266	0.702
ARRIMA	0.482	0.408	0.542

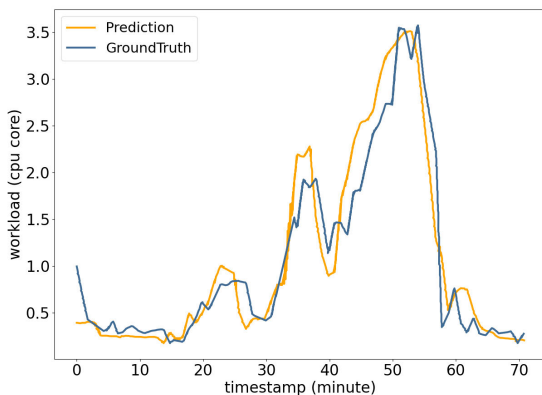
Table 2 shows the time series prediction results of the same algorithms on the Alibaba cluster load dataset. Similarly, thanks to the self-attention mechanism of the Informer algorithm, it still achieves the best prediction performance. Since the Alibaba cluster load dataset is much larger than the WorldCup dataset, this further demonstrates the superiority of the Informer algorithm in extracting global temporal features for long-term sequence prediction.

**TABLE 2.** Alibaba cluster CPU load time series prediction results.

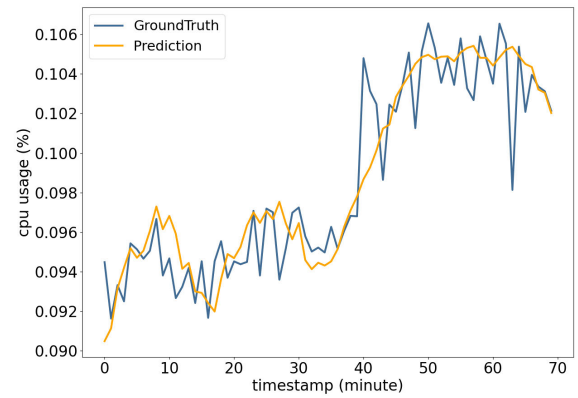
Model	MAE	MSE	$R^2$
Informer	0.011	0.009	0.794
Bi-LSTM	0.018	0.012	0.703
LSTM	0.026	0.020	0.685
RNN	0.039	0.031	0.641
ARRIMA	0.045	0.039	0.578

The time series prediction performance of the Informer algorithm is shown in Figure 3. The blue line represents the actual load values and the orange line represents the predicted load values. It can be seen that the prediction results closely fit the actual load changes. The predicted load increase slightly precedes the actual values, which can benefit elastic scaling optimization.

Similarly, the Informer algorithm's prediction of the Alibaba cluster load data is shown in Figure 4. Due to the significant fluctuations in the real CPU load of the cluster, the predicted curve does not fit the actual load changes very well. However, it still accurately predicts the overall trend of the CPU load changes.

**FIGURE 3.** Worldcup: Informer load time series prediction performance chart.

In Figures 3 and 4, the vertical axis represents the CPU core load values, and the horizontal axis represents the time

**FIGURE 4.** Alibaba: Informer load time series prediction performance chart.

points, with the unit being minutes. In the experiment we designed, the horizontal axis of Figure 3 actually corresponds to the last time segment of the horizontal axis in Figure 1, specifically time points 410 to 480, corresponding to the period from 1998-08-03 02:57:00 to 1998-08-03 04:07:00. Similarly, Figure 4 shows one segment of the CPU load variation from Figure 2, specifically time points 2600 to 2670.

Currently, the mainstream approach in the industry for online applications is to allocate sufficient resources. This creates a redundancy pool to cope with sudden load spikes. However, during scaling operations, the startup time and cost of containers and Pods can still lead to declining business QoS (quality of service) metrics during sudden load surges. This anticipatory scaling maximizes the application's service quality while ensuring that the increase in resource usage remains acceptable. This demonstrates that the Informer time series prediction method is highly suitable for optimizing elastic scaling.

## B. ELASTIC SCALING OPTIMIZATION

The elastic scaling optimization part uses the widely used cloud computing simulation environment CloudSim [27] and its extension, AutoScaleSim [28]. CloudSim is a cloud computing simulation software introduced on April 8, 2009, by the Gridbus Project and the CLOUDS Laboratory at the University of Melbourne. After years of improvement and development, it now supports the modeling and simulation of large-scale cloud computing infrastructures. It provides a virtualization engine. It also offers resource management and scheduling capabilities for cloud computing. CloudSim supports the modeling and simulation of virtualized data centers and application containers. This makes it the most widely used cloud computing simulation environment. It is suitable for this elastic scaling optimization experiment.

AutoScaleSim is an automatic elastic scaling simulation environment extended from CloudSim. It supports the complete MAPE (Monitor, Analyze, Plan, Execute) process for automatic elastic scaling simulation and has been validated in a real-world cloud computing environment



using OpenStack. This validation has demonstrated that its metrics are consistent with those of real cloud computing environments. AutoScaleSim is suitable for verifying the elastic scaling optimization method proposed in this paper.

In this paper, the interface of AutoScaleSim was extended to implement the proposed dynamic interval elastic scaling optimization method based on Informer time series prediction. Comparative experiments were conducted between this method and the HPA method. The results showed significant improvements in SLA violation rates and average application response times. Acceptable resource utilization metrics declined. This led to enhanced service quality indicators for applications. The overall architecture of AutoScaleSim is shown in Figure 5.

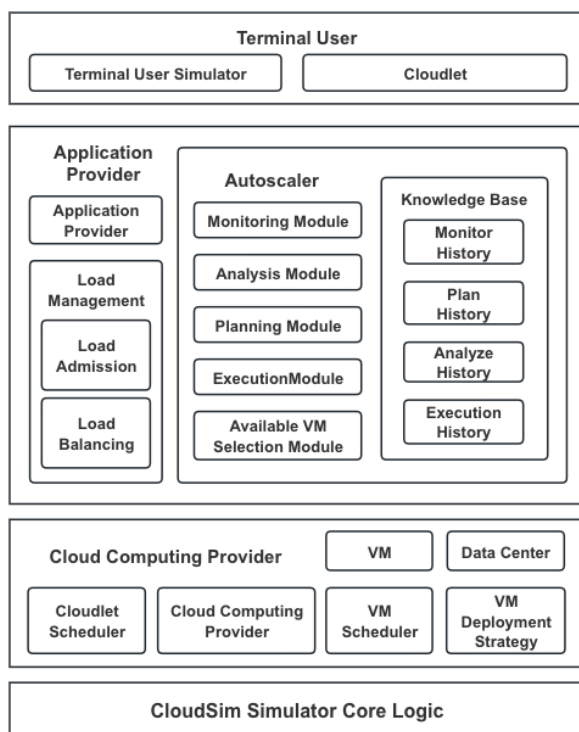


FIGURE 5. Autoscalesim overall architecture diagram.

Three methods were compared in the elastic optimization experiment. Firstly, It compared the dynamic interval elastic scaling method based on Informer time series prediction with the Kubernetes HPA method. The comparison was done on the last WorldCup official forum access load cycle and the Alibaba cluster CPU load data corresponding to time points 2600 to 2670. The scaling was based on CPU resources, with other resources being handled similarly. The Pod Request was set to 300m Core, and the target utilization (Target) was set to 50%. Additionally, to validate the advantages of Informer in elastic scaling scenarios, we introduced an elastic scaling method based on RNN time series prediction and

conducted comparative experiments using AutoScaleSim as the simulation environment.

The comparison experiment results are shown in Table 3 and Table 4. Since the conclusions drawn from the data in both tables are consistent, only the analysis of Table 3 is provided here.

It is evident that due to the significant load spikes, the Kubernetes HPA method results in higher application response times and SLA violation rates. In contrast, using the method proposed in this paper, the average application response time decreased from 0.821s to 0.692s. The SLA violation rate decreased from 18.277% to 9.157%. CPU resource utilization only slightly decreased from 47.6% to 40.0%. Additionally, the total number of scaling-down events remained largely unchanged. The cumulative number of scaled-up Pod replicas and scaled-down Pod replicas also remained largely unchanged. However, the total number of scaling-up events decreased from 12 to 5 using the proposed method. Fewer scaling-up events can improve the stability of the container cloud platform.

In addition, both Table 3 and Table 4 show that the dynamic interval elastic scaling method based on Informer time series prediction effectively balances CPU resource utilization, application response time, and SLA violation rate. It ensures that, even with significant load spikes, CPU resource utilization only slightly decreases while significantly reducing application response time and SLA violation rate. In contrast, the elastic scaling method based on RNN cannot achieve this balance, as it can only reduce application response time and SLA violation rate by sacrificing CPU resource utilization. Therefore, our proposed method can more flexibly and rapidly adjust the allocated cloud resources according to the application's resource demands.

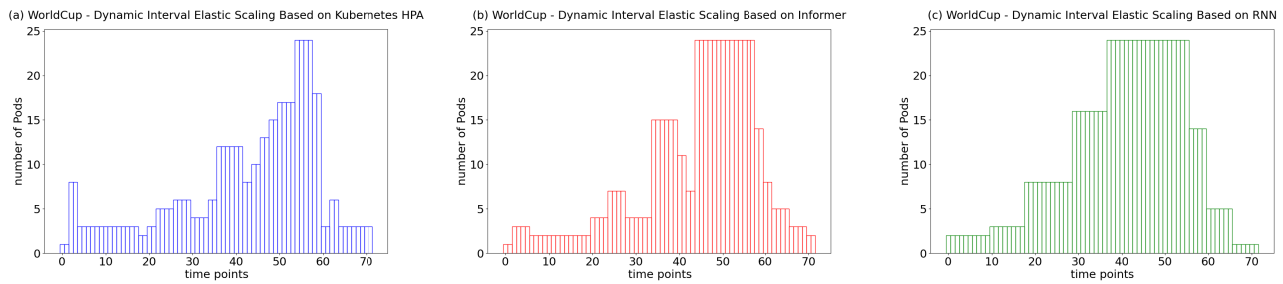
TABLE 3. Worldcup: Auto-scaling optimization comparison experiment results.

Method	K8S HPA	RNN Auto-Scaling	Ours
CPU Resource Utilization	47.6%	31.4%	40.0%
Average Response Time	0.821s	0.617s	0.629s
SLA Violation Rate	18.277%	9.134%	9.157%
Total Number of Scaling-ups	12	4	5
Cumulative Scaled-up Pods	35	22	36
Total Number of Scaling-downs	7	3	9
Cumulative Scaled-down Pods	36	23	34

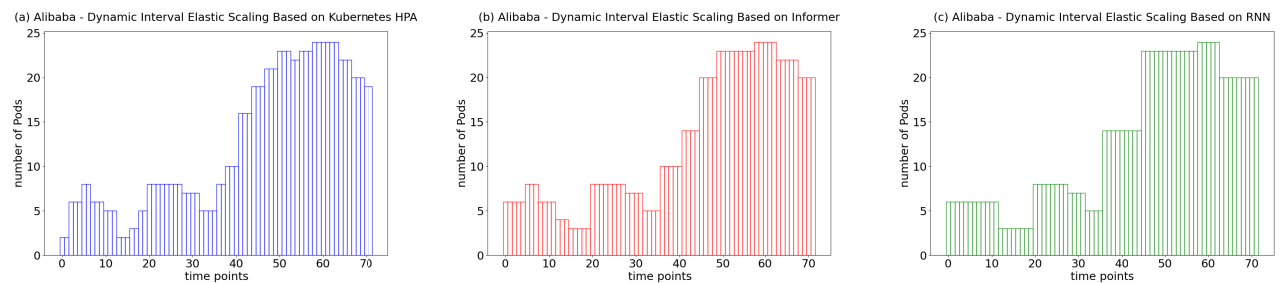
TABLE 4. Alibaba: Auto-scaling optimization comparison experiment results.

Method	K8S HPA	RNN Auto-Scaling	Ours
CPU Resource Utilization	40.1%	25.7%	36.9%
Average Response Time	1.197s	0.924s	0.909s
SLA Violation Rate	25.819%	12.594%	12.992%
Total Number of Scaling-ups	14	3	6
Cumulative Scaled-up Pods	31	24	26
Total Number of Scaling-downs	9	4	7
Cumulative Scaled-down Pods	15	10	12

Figures 6 and 7 show the simulation results of different elastic scaling methods in the AutoScaleSim environment



**FIGURE 6. Worldcup: Pod replica count variation chart.**



**FIGURE 7. Alibaba: Pod replica count variation chart.**

(Subfigure a represents Kubernetes HPA, Subfigure b represents Informer-based scaling, and Subfigure c represents RNN-based scaling). The horizontal axis represents the corresponding time points in minutes, and the vertical axis represents the number of Pods at those time points. The changes in the number of Pods reflect the corresponding scaling actions. Similarly, since the conclusions drawn from the data in both figures are consistent, only the analysis of Figure 6 is provided here.

Subfigures a, b, and c in Figure 6 respectively illustrate the changes in the number of application Pod replicas under the Kubernetes HPA horizontal scaling method, the dynamic interval elastic scaling method based on Informer time series prediction proposed in this paper, and the elastic scaling method based on RNN. It can be observed that the method proposed in this paper allocates sufficient resources to the application in advance when there is a sudden increase in load, reduces unnecessary scaling down, and improves the overall stability of the scaling process. Furthermore, this method avoids excessive allocation of redundant resources, thereby achieving high CPU resource utilization while maintaining low application response time and a reduced SLA violation rate.

Considering that the mainstream approach in the industry is to provide additional resource redundancy to cope with sudden traffic and load increases in online applications, which results in the average resource utilization rate of most clusters being below 35%, the 7.6% reduction in CPU resource utilization caused by the method proposed in this paper is entirely acceptable compared to the 16.2% reduction seen with the RNN-based elastic scaling method. This also

demonstrates that our method can significantly improve the quality of service for online applications in container cloud platforms while keeping the additional resource consumption within an acceptable range.

### C. LIMITATIONS

An obvious fact is that the accuracy of model predictions may decrease as the scale of the container cloud platform increases, because scaling introduces more uncertainties, and such volatility can reduce the predictive accuracy of the model, leading to over-provisioning or under-provisioning of resources, which negatively impacts the cost-effectiveness and performance of the system. Due to limitations in experimental resources, we were unable to conduct validation at a larger scale; thus, the method we proposed currently only considers scenarios involving small-scale container cloud platforms. In the experiments we designed, the container cloud platform managed on the order of  $10^3$  instances. At this scale, our proposed method demonstrated good elastic scaling performance. However, if the number of instances were to be further increased to the order of  $10^4$  or even  $10^5$ , a decline in performance may occur. Additionally, the exploration focused solely on elastic scaling for stateless services, as stateful applications require extra consideration for data synchronization and migration, rendering them unsuitable for the proposed method.

It should also be noted that during practical implementation, this method may result in longer prediction times and higher resource overhead due to its computational complexity. Informer relies on the self-attention mechanism to perform time series prediction, which brings a time

complexity of  $O(n^2)$ . This can cause performance bottlenecks in elastic scaling systems with high real-time requirements, potentially necessitating an additional reactive scaling mechanism to ensure that the system can respond promptly.

## V. CONCLUSION

This paper proposes a dynamic interval elastic scaling optimization method based on Informer time series prediction. The method uses the Informer time series prediction algorithm. This algorithm is an improvement based on the Transformer architecture. It predicts application resource loads. Taking advantage of its superior long sequence prediction capabilities compared to algorithms like LSTM, RNN, and ARIMA, the proposed method optimizes dynamic scaling intervals. By allocating sufficient resources to applications in advance of actual resource demand growth, the method ensures timely scaling operations.

Experimental validation shows that with only a reduction in CPU resource utilization from 47.6% to 40.0%, the average application response time decreased from 0.821s to 0.692s, and the SLA violation rate dropped from 18.277% to 9.157%. These results demonstrate that the proposed method significantly improves the service quality of online applications in container cloud platforms, with acceptable additional resource consumption.

Current research represents just a phase of progress. Future work aims to explore the scalability of this method on large-scale container cloud platforms, investigate how to maintain predictive accuracy and efficiency as the system scales, and extend this method to stateful services by addressing challenges related to data synchronization, state management, and consistency during scaling operations. Additionally, while the existing method has alleviated the issue of service quality degradation caused by sudden load spikes to some extent, predictions may still lag in extreme cases, leading to insufficient resource allocation. Breakthroughs in rapid response mechanisms for sudden load spikes are anticipated, such as incorporating more sensitive anomaly detection algorithms that can immediately trigger additional resource allocation when an abnormal surge in load is detected.

## REFERENCES

- [1] G. J. Choi, H. Kang, B. G. Kim, Y. S. Choi, J. Y. Kim, and S. Lee, "Pain after single-incision versus conventional laparoscopic appendectomy: A propensity-matched analysis," *J. Surgical Res.*, vol. 212, pp. 122–129, May 2017, doi: [10.1016/j.jss.2017.01.023](https://doi.org/10.1016/j.jss.2017.01.023).
- [2] A. Bhardwaj and C. R. Krishna, "Virtualization in cloud computing: Moving from hypervisor to containerization—A survey," *Arabian J. Sci. Eng.*, vol. 46, no. 9, pp. 8585–8601, 2021, doi: [10.1007/s13369-021-05553-3](https://doi.org/10.1007/s13369-021-05553-3).
- [3] *Swarm Mode Overview, Docker Documentation*. Accessed: Sep. 16, 2024. [Online]. Available: <https://docs.docker.com/engine/swarm/>
- [4] *Marathon, Mesosphere Documentation*. Accessed: Sep. 16, 2024. [Online]. Available: <https://mesosphere.github.io/marathon/>
- [5] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Commun. ACM*, vol. 59, no. 5, pp. 50–57, Apr. 2016, doi: [10.1145/2890784](https://doi.org/10.1145/2890784).
- [6] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003, doi: [10.1109/mc.2003.1160055](https://doi.org/10.1109/mc.2003.1160055).
- [7] C. Huang and A. Petukhina, "ARMA and ARIMA modeling and forecasting," in *Applied Time Series Analysis and Forecasting With Python*. Cham, Switzerland: Springer, 2022, pp. 107–142, doi: [10.1007/978-3-031-13584-2\\_4](https://doi.org/10.1007/978-3-031-13584-2_4).
- [8] M. Georgiopoulos, C. Li, and T. Kocak, "Learning in the feed-forward random neural network: A critical review," *Perform. Eval.*, vol. 68, no. 4, pp. 361–384, 2011, doi: [10.1016/j.peva.2010.11.002](https://doi.org/10.1016/j.peva.2010.11.002).
- [9] Y. Yu, X. Si, C. Hu, and J. Zhang, "A review of recurrent neural networks: LSTM cells and network architectures," *Neural Comput.*, vol. 31, no. 7, pp. 1235–1270, Jul. 2019.
- [10] S. Wang, X. Wang, S. Wang, and D. Wang, "Bi-directional long short-term memory method based on attention mechanism and rolling update for short-term load forecasting," *Int. J. Electr. Power Energy Syst.*, vol. 109, pp. 470–479, Jul. 2019.
- [11] M. Yan, X. Liang, Z. Lu, J. Wu, and W. Zhang, "HANSEL: Adaptive horizontal scaling of microservices using bi-LSTM," *Appl. Soft Comput.*, vol. 105, Jul. 2021, Art. no. 107216, doi: [10.1016/j.asoc.2021.107216](https://doi.org/10.1016/j.asoc.2021.107216).
- [12] H. Zhou, S. Zhang, and J. Peng, "Informer: Beyond efficient transformer for long sequence time-series forecasting," in *Proc. 35th AAAI Conf. Artif. Intell.*, 2021, pp. 1–9, doi: [10.1609/aaai.v35i12.17325](https://doi.org/10.1609/aaai.v35i12.17325).
- [13] A. Vaswani, N. Shazeer, and N. Parmar, "Attention is all you need," *Comput. Lang.*, vol. 14, no. 4, pp. 1–10, 2017.
- [14] RightScale. *Set Up Autoscaling Using Voting Tags*. Accessed: Sep. 16, 2024. [Online]. Available: [http://support.rightscale.com/03-Tutorials/02-AWS/02-Website\\_Edition/Set\\_up\\_Autoscaling\\_using\\_Voting\\_Tags](http://support.rightscale.com/03-Tutorials/02-AWS/02-Website_Edition/Set_up_Autoscaling_using_Voting_Tags)
- [15] S. Taherizadeh, V. Stankovski, and J.-H. Cho, "Dynamic multi-level auto-scaling rules for containerized applications," *Comput. J.*, vol. 62, no. 2, pp. 174–197, Feb. 2019, doi: [10.1093/comjnl/bxy043](https://doi.org/10.1093/comjnl/bxy043).
- [16] B. Malet and P. Pietzuch, "Resource allocation across multiple cloud data centres," in *Proc. 8th Int. Workshop Middleware Grids, Clouds e-Sci.*, Nov. 2010, pp. 1–6, doi: [10.1145/1890799.1890804](https://doi.org/10.1145/1890799.1890804).
- [17] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2016, pp. 785–794, doi: [10.1145/2939672.2939785](https://doi.org/10.1145/2939672.2939785).
- [18] S. Zhang, T. Wu, M. Pan, C. Zhang, and Y. Yu, "A-SARSA: A predictive container auto-scaling algorithm based on reinforcement learning," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, Oct. 2020, pp. 489–497, doi: [10.1109/ICWS49710.2020.00072](https://doi.org/10.1109/ICWS49710.2020.00072).
- [19] Q. Z. Ullah, G. M. Khan, and S. Hassan, "Cloud infrastructure estimation and auto-scaling using recurrent Cartesian genetic programming-based ANN," *IEEE Access*, vol. 8, pp. 17965–17985, 2020, doi: [10.1109/ACCESS.2020.2966678](https://doi.org/10.1109/ACCESS.2020.2966678).
- [20] M. Imdoukh, I. Ahmad, and M. G. Alfaiilakawi, "Machine learning-based auto-scaling for containerized applications," *Neural Comput. Appl.*, vol. 32, no. 13, pp. 9745–9760, Jul. 2020, doi: [10.1007/s00521-019-04507-z](https://doi.org/10.1007/s00521-019-04507-z).
- [21] A. Y. Nikraves, S. A. Ajila, and C.-H. Lung, "Using genetic algorithms to find optimal solution in a search space for a cloud predictive cost-driven decision maker," *J. Cloud Comput.*, vol. 20, pp. 1–13, Dec. 2018, doi: [10.1186/s13677-018-0122-7](https://doi.org/10.1186/s13677-018-0122-7).
- [22] X. Tang, Q. Liu, Y. Dong, J. Han, and Z. Zhang, "Fisher: An efficient container load prediction model with deep neural network in clouds," in *Proc. IEEE Int. Conf. Parallel Distrib. Process. With Appl., Ubiquitous Comput. Commun., Big Data Cloud Comput., Social Comput. Netw., Sustain. Comput. Commun. (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, Dec. 2018, pp. 199–206, doi: [10.1109/BDCloud.2018.00041](https://doi.org/10.1109/BDCloud.2018.00041).
- [23] S. Kumar, N. Muthiyar, S. Gupta, A. D. Dileep, and A. Nigam, "Association learning based hybrid model for cloud workload prediction," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2018, pp. 1–8, doi: [10.1109/IJCNN.2018.8488996](https://doi.org/10.1109/IJCNN.2018.8488996).
- [24] M. B. Taha, Y. Sanjalawe, A. Al-Daraiseh, S. Fraihat, and S. R. Al-E'mari, "Proactive auto-scaling for service function chains in cloud computing based on deep learning," *IEEE Access*, vol. 12, pp. 38575–38593, 2024, doi: [10.1109/access.2024.3375772](https://doi.org/10.1109/access.2024.3375772).
- [25] B. Jeong, J. Jeon, and Y.-S. Jeong, "Proactive resource autoscaling scheme based on SCINet for high-performance cloud computing," *IEEE Trans. Cloud Comput.*, vol. 11, no. 4, pp. 3497–3509, Oct. 2023, doi: [10.1109/TCC.2023.3292378](https://doi.org/10.1109/TCC.2023.3292378).
- [26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. NIPS*, 2017, pp. 6000–6010, doi: [10.48550/arXiv.1706.03762](https://doi.org/10.48550/arXiv.1706.03762).

- [27] Cloudbus Project. *CloudSim: A Framework For Modeling and Simulation of Cloud Computing Infrastructures and Services*. Accessed: Sep. 16, 2024. [Online]. Available: <http://cloudbus.org/cloudsim/>
- [28] M. S. Aslanpour, A. N. Toosi, J. Taheri, and R. Gaire, "AutoScaleSim: A simulation toolkit for auto-scaling Web applications in clouds," *Simul. Model. Pract. Theory*, vol. 108, pp. 1–17, Apr. 2021, doi: [10.1016/j.simpat.2020.102245](https://doi.org/10.1016/j.simpat.2020.102245).



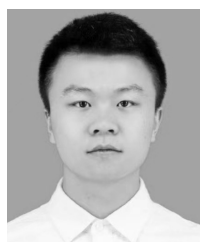
**ZHENGONG CAI** received the B.S. and Ph.D. degrees from the School of Computer Science and Engineering, Zhejiang University, China, in 2006 and 2011, respectively. He is currently with the School of Software Technology, Zhejiang University, in 2011. His research interests include software engineering, data mining, and cloud computing.



**YU DING** received the B.S. degree in computer science and technology from Jilin University, in 2005. He is currently pursuing the Ph.D. degree with the School of Computer Science, Zhejiang University. He is a Researcher at the Alibaba Cloud Intelligence Group. His main research interests include containerization, unified resources scheduling, co-location for containers, enterprise cloud architecture, and AI engineering and technology.



**XINGHAO WANG** received the B.S. degree from the School of Computer Science and Engineering, Central South University, China, in 2023. He is currently pursuing the M.S. degree with the School of Software Technology, Zhejiang University, China. His main research interests include cloud computing and software engineering.



**CHENHAO LI** received the B.S. degree from the College of Computer Science and Technology, Zhejiang University, China, in 2024. His main research interests include cloud computing and software engineering.



**BOWEI YANG** received the Ph.D. degree in computer science and technology from Zhejiang University, China, in 2011. He is currently an Associate Professor with the School of Aeronautics and Astronautics, Zhejiang University, in 2016. His main research interests include software-defined satellite networking, UAV ad hoc networks, AI-assisted wireless communications technology, and AI-big data joint optimization in cellular networks and D2D communications.

...