Hindawi
International Journal of Intelligent Systems
Volume 2023, Article ID 5959223, 25 pages
https://doi.org/10.1155/2023/5959223

Research Article

An Efficient Load Prediction-Driven Scheduling Strategy Model in Container Cloud

Lu Wang, Shuaidong Guo, Pengli Zhang, Haodong Yue, Yaxiao Li, Chenyi Wang, Zhuang Cao, and Di Cui

School of Computer Science and Technology, Xidian University, Xi'an 710071, China

Correspondence should be addressed to Lu Wang; wanglu@xidian.edu.cn

Received 4 May 2023; Revised 30 June 2023; Accepted 1 July 2023; Published 17 July 2023

Academic Editor: Mohammad R. Khosravi

The rise of containerization has led to the development of container cloud technology, which offers container deployment and management services. However, scheduling a large number of containers efficiently remains a significant challenge for container cloud service platforms. Traditional load prediction methods and scheduling algorithms do not fully consider interdependencies between containers or fine-grained resource scheduling, leading to poor resource utilization and scheduling efficiency. To address these challenges, this paper proposes a new load prediction model CNN-BiGRU-Attention and a container scheduling strategy based on load prediction. The prediction model CNN and BiGRU focus on the local features of load data and long sequence dependencies, respectively, as well as introduce the attention mechanism to make the model more easily capture the features of long distance dependencies in the sequence. A container scheduling strategy based on load prediction is also designed, which first uses the load prediction model to predict the load state and then generates a scheduling strategy based on the load prediction value to determine the change of the number of container replicas in a fine-grained manner based on the load prediction value in the next time window, while the established domain-based container selection method is employed to facilitate the coarse-grained online migration of containers. Experiments conducted using public datasets and open-source simulation platforms demonstrate that the proposed approach achieves a 37.4% improvement in container load prediction accuracy and a 21.7% improvement in container scheduling efficiency compared to traditional methods. These results highlight the effectiveness of the proposed approach in addressing the challenges faced by container cloud service platforms.

1. Introduction

A container is a form of operating system-level virtualization, allowing people to use containers to run everything from small-scale microservices to large-scale applications, affording systematic construction on agility and high compatibility [1]. Large containers must centralize the deployment and management of forms, with a container cloud involving a cloud computing technology that provides container deployment and management services. Currently, the container cloud has entered a stage of rapid development in the market. In the public cloud market, containers have widely covered 20%–35% of virtualization applications. According to iResearch, this number will grow to 50%–75% in 2025. Furthermore, by 2025, it will exceed 6 billion yuan, and the container cloud market will maintain high growth [2].

In the container cloud's scheduling process, we first conduct preselection to traverse all nodes and filter out the ones that do not meet the conditions. All nodes that meet the output of the requirement at this stage will be recorded and used as the input for the second stage. If all nodes do not meet the conditions, the Pod will be pending until the notes meet conditions and the scheduler will retry exploiting them. After filtering, if multiple nodes meet the conditions, the system will enable them according to their priorities and finally select the node with the highest priority to deploy the Pod application.

The container cloud platform currently faces several challenges. Firstly, predicting container load conditions is a complex and variable process that becomes increasingly challenging as time progresses. Traditional prediction methods only consider one factor, which oversimplifies the

issue. Secondly, a linear prediction model requires stable and continuous time series data, which poses significant difficulties for accurately predicting load conditions. The container scheduling strategy also presents several challenges, such as larger container scales and complex dependency relationships, resulting in increased server resource consumption and reduced scheduling efficiency. Fine-grained resource scheduling is not considered, and there is a lack of distinction between coarse-grained and fine-grained resources. Additionally, maintaining high scheduling frequency may lead to low resource utilization rates. These issues must be addressed to improve the performance and reliability of the container cloud platform.

Spurred by the current deficiencies, this paper conducts reasonable modeling and accurate prediction of many heterogeneous resources to ensure the normal operation, safety, and reliability of the container cloud environment that stably responds to emergencies. The contributions of this work are as follows:

(1) The proposed solution utilizes the CNN-BiGRU-Attention model to develop a load prediction method that addresses traditional model issues and copes with complex load situations in container cloud resources. Compared with other load prediction models such as ARIMA, DBN, and CNN-LSTM, CNN-BiGRU-Attention prediction is better because (1) for time series data, convolutional neural networks train models better than ARIMA and DBN; (2) BiGRU adopts the idea of bidirectional mechanism to build a network structure with bidirectional gated cyclic units, which has a more concise network structure with faster model training compared to LSTM, and the GRU model only extracts features from a single direction for the input sequence, while the BiGRU model is designed to process the load sequence with a pair of opposite direction GRU models and merge the results of two GRU model operations along opposite directions at the output, while a two-dimensional vector transformed by the Gram's corner field is used to preserve the dependence on time; thus, capturing features that GRU may ignored; and (3) the attention mechanism can capture the long time series data dependencies that may be missed by BiGRU and improve the accuracy of the model.

(2) The container scheduling strategy based on a load prediction model, CSSLPM, reduces the frequency of cross-server call with the binding of coarse-grained node-based and fine-grained container-based scheduling strategy on the base of load prediction, solves the fragmentation problem of server resources, and provides load prediction and resource optimization for cloud container environment.

We experimentally evaluate and improve the proposed load prediction method and container scheduling strategy utilizing cluster-trace-v2018 (a public dataset), CloudSim (open-source cloud computing simulation platform software) [3], and TrainTicket (an open-source train booking benchmark system). The corresponding results demonstrate that compared with the traditional method, the container load prediction accuracy of the proposed method is improved by 37.4%, and the container scheduling efficiency is increased by 21.7%.

The rest of the paper is organized as follows. Section 2 explains related works, while Section 3 describes the CNN-BiGRU-Attention load prediction model and the CSSLPM container scheduling strategy in detail. Section 4 presents experimental verification of the model's accuracy and the strategy's effectiveness. Finally, Section 5 summarizes the paper and suggests future research directions.

2. Related Work

With the development of container technology, scheduling-based containers have been widely researched, and appealing results have been achieved.

In industry, widely used container orchestration systems are Mesos [4], Kubernetes [5], and Swarmkit [6], developed by different companies. Mesos provides an interface for developers to service, where the developers should overwrite the original scheduling method and customize the plan. It has a higher threshold for developers. Kubernetes is an open-source project from Google. Compared with other orchestration tools, it has the most features, including automated software deployment and cluster-level scaling. However, it has many disadvantages, such as complex architecture, involving a large system, and difficulty in modifying and operating [5]. Swarmkit is an open-source project developed by Docker in 2016. Because it belongs to the same company as Docker, it is compatible with Docker container technology. Its architecture is easy to read, and the tools are simple to learn. Nevertheless, its scheduling method is too simple to satisfy container scheduling. However, when the number of containers increases, the existing scheduling strategies have limited applicability [6].

Various techniques are widely used for scheduling problems, including mathematical modeling, heuristics, meta-heuristics, and machine learning. Mathematical modeling involves optimizing a linear function using a set of linear constraints, with integer linear programming being a common technique. Zhang et al. [7] proposed a linear model for the deployment of containers to servers. The model considered two optimization objects, energy consumption and network cost. However, due to the complexity of the ILP formulation, it is not suitable for solving large-scale problems. Heuristics are often used to quickly realize solutions, with DRAPS [8] being a proposed algorithm for container deployment in Docker Swarm. This algorithm selects the node for container deployment based on available resources and service demand, resulting in a more efficient and balanced usage of resources compared to Swarmkit. However, this approach may lead to high network workloads. Inspired by intelligent processes and behaviors in nature, meta-heuristic algorithms have two important characteristics: selecting the fittest and adapting to the environment. Ant colony optimization [9] is an example of meta-heuristics used for container scheduling, which can

enhance resource utilization using proper load balancing. However, this algorithm only considers a few optimization objectives. For example, Li from Southwest Jiaotong University proposed a load balancing algorithm for the Kubernetes cluster [10] that considers the cluster node load information, including CPU, RAM, and disk usage, but does not consider the fine granularity of resources. Machine learning-based solutions offer more intelligent scheduling decisions to improve solution accuracy and effectiveness than other heuristic algorithms. Nanda and Hacker [11] proposed a deep reinforcement learning technique for container scheduling, which outperformed the shortest job first and random placement algorithms. However, all methods do not consider container dependency so that the dependent container is distributed to different server causing cross-server call and increasing server load. Besides, these methods ignore the grain of server resource and waste some of server fragment resources. In this academic paper, we present CSSLPM—a solution that offers load balancing and resource optimization capabilities for container cloud environments. Our approach is based on training models to anticipate the state of container cloud loads and leveraging a combination of coarse-grained node-based and fine-grained container scheduling. Through this method, we are able to decrease the frequency of cross-server scheduling of service invocations and effectively address the issue of server resource fragmentation.

For container scheduling technology, its accuracy and timeliness depend not only on whether the scheduling algorithms are superior but also, to some extent, on whether the analysis of historical load data is comprehensive and whether the prediction methods are highly efficient. At present, academics have widely conducted research on load prediction and proposed many effective methods, such as prediction algorithms based on probability and statistics, big data, and neural networks.

The classical methods include the ARMA-based time series model, AR model, and ES model for the prediction algorithms based on probability and statistics. However, these methods suffer from limited expressiveness, so some scholars proposed improvement schemes. For instance, Xue et al. [12] fixed the regression model and added particle filtering. However, this method requires a large number of samples. Wei et al. proposed the dynamic feedback load balancing algorithm of load weight by comprehensively considering the nodes' real-time load Information and their performance [13]. Nevertheless, two aspects are considered to ensure that the nodes with better performance can bear relatively more loads so that the ratio of computing resources and loads between nodes in the cluster will be more balanced. However, the problem that complex dependency occupies too many server resources is not considered. In addition, Monfared et al. [14] and Calheiros et al. [15] also improved the prediction algorithm based on the ES and ARMA models, respectively. Specifically, these methods automatically adjust the parameters according to the actual situation, but they show shortcomings when facing the complex and variable load in a real situation. For example, the prediction accuracy is limited. Chen and Fang [16] proposed a load forecasting algorithm based on the analytic integrated model to achieve load forecasting based on big data analysis for medium- and long-term load time series data. Wang et al. [17] introduced a load forecasting method based on K-mode clustering to achieve load curve forecasting. However, such big data-based load prediction methods are based on massive data, and the technical difficulty is greater than general methods. So, the span of historical load time series data cannot be too large. Neural network-based load forecasting methods can solve the drawbacks of big data-based methods, and the forecasting process can be achieved with a smaller amount of data. For example, Islam et al. [18] implemented load prediction by combining NNs and AR models, and Qiu et al. [19] proposed a load prediction method based on RBM and DBN to achieve workload prediction for virtual machines in cloud environments. Ashraf [20] used an LSTM-RNN network in automatic scaling for load prediction of virtual resources, and Guo et al. [21] developed a type-aware-based prediction method that determines the current load type based on the dynamic change of the load to switch the prediction method accordingly.

We analyzed the computational complexity of the above-mentioned prediction models and finally selected BiGRU to extract the load data dependencies. The ARMA model is a linear model with relatively low model computational complexity due to the small number of parameter models. The computational complexity of the DBN model mainly depends on the number and size of its hidden layers and the number of neurons in each layer. The computational complexity of the LSTM model depends mainly on the length of the input sequence and the size of the hidden layers. Under the same parameter scale, the AMRA model has the lowest computational complexity but poor prediction for non-smooth sequences. BiGRU has lower computational complexity than DBN and LSTM due to its simple structure and has better prediction performance than DBN and LSTM for shorter length input sequences, while LSTM may outperform BiGRU in some long-term dependency tasks. The differences among different scheduling strategies and load prediction methods are shown in Tables 1 and 2, respectively.

In summary, the existing studies of neural network-based load prediction methods using simple NNs models fail to capture the complex features in load timing data, lack consideration of the data context, and underperform in training efficiency. When the service invocation, load balancing, and resource optimization are too complex in the cloud environment, numerous defects are revealed, and the desired prediction accuracy cannot be achieved. Most currently used scheduling strategies suffer from additional resource overhead, resulting in resource waste or scheduling strategy being too simple to meet the scheduling objectives. Therefore, this paper proposes a resource scheduling technique based on load prediction and describes and models the container resources in cloud environments in a more reasonable and diversified way with real-time, accuracy, and scalability goals.

3. Load Prediction Algorithm and Container Scheduling Strategy

This section describes the proposed load prediction algorithm and container scheduling strategy in a container cloud environment, with the corresponding workflow illustrated in Figure 1.

Table 1: Scheduling comparison.

	Large-scale problem	Low network load	Sufficient optimization objective	Dependencies	Performance	CPU utilization
Mathematical model [7]	×	✓	✓	×	+	+
Heuristics [8]	✓	×	✓	×	+	++
Meta-heuristics [9, 10]	✓	✓	×	×	++	++
Machine learning-based model [11]	✓	✓	✓	×	+	+
CSSLPM	✓	✓	✓	✓	+++	+++

Large-scale problem: ability to work with large-scale scheduling problems. Low network load: ability to work under low network load. Sufficient optimization objective: ability to optimize the scheduling process in a multi-objective manner. Dependencies: ability to consider container dependencies. Performance: the less the resources and time consumed by scheduling, the better the performance; more "+" means better performance. CPU utilization: ratio of CPU consumed by the algorithm in scheduling only to the overall CPU consumed by the algorithm; more "+" means better CPU utilization.

Table 2: Prediction comparison.

	Large dataset requirement	Completion easiness	Length dependent	Accuracy	Training easiness
ARMA model [12, 14, 15]	✓	++	×	+	++
K-modes [17]	×	+++	×	+	++
RBM and DBN [19]	✓	++	×	+	++
RNN and variants [20]	✓	+	✓	++	+++
CNN-BiGRU-Attention	✓	+	✓	+++	+

Large dataset requirement: performance depends on large dataset. Completion easiness: ease of building model; more "+" means easier building model. Length dependent: ability to identify long time series dependencies. Accuracy: accuracy of load prediction for container cloud environments; more "+" means better accuracy. Training easiness: ease of training model; more "+" means easier training model.

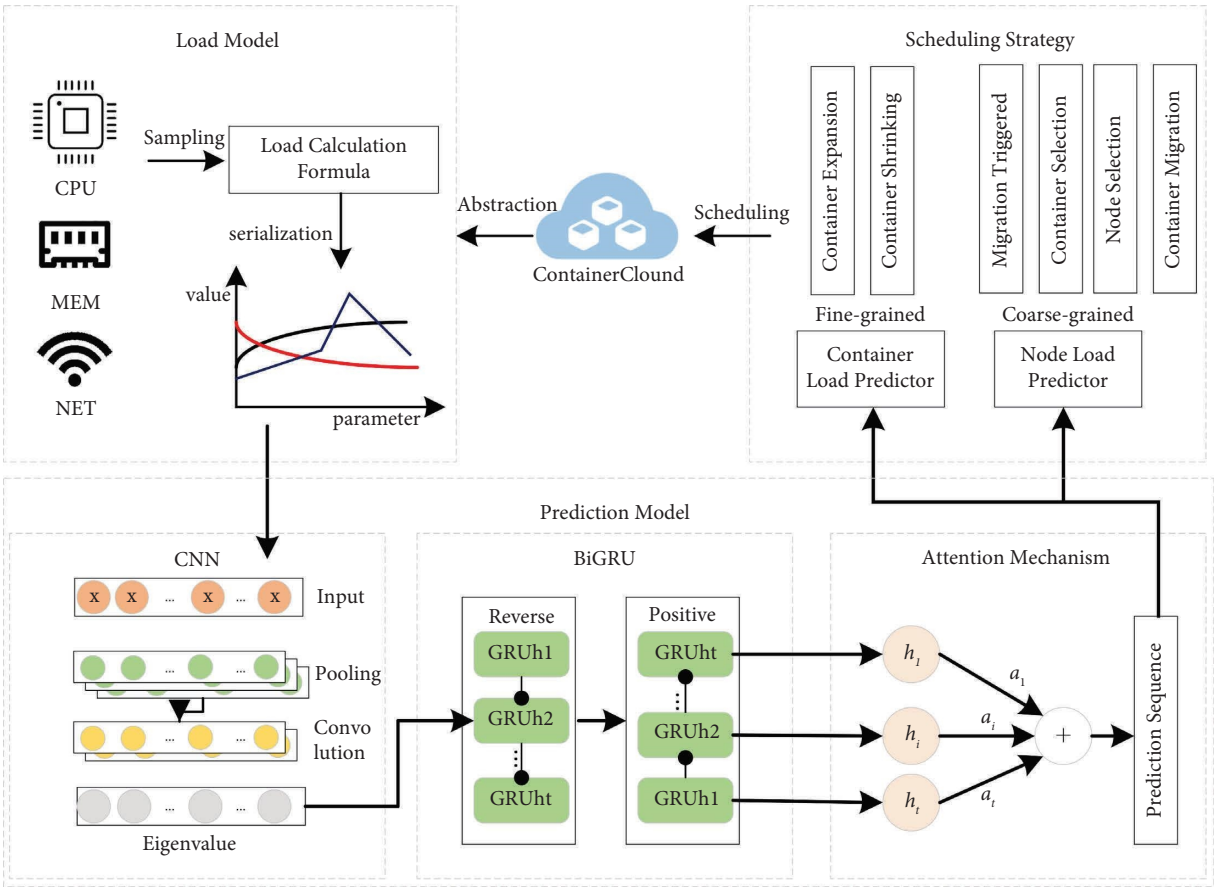

Figure 1: Workflow of the proposed container scheduling strategy based on load prediction.

3.1. Load Characteristics and Load Model. This subsection builds a load model to quantitatively describe the resource load of the container clouds by extracting characteristic quantities such as CPU, memory utilization, and network latency from the container and node perspectives and temporalizing the container cloud load data by the corresponding load calculation formula. To optimize load prediction, it is necessary to quantify the resource load conditions, i.e., based on comprehensive and in-depth analysis of load characteristics, load factors that can be quantified are extracted from resource monitoring information and combined into a load model consisting of multi-dimensional factors. Regarding the load characteristics, Dinda from Carnegie Mellon University, USA, conducted experiments and studies for load variation and obtained a large amount of load information during long-term tracking sampling. Moreover, the author conducted the statistical analysis and organized and summarized the load characteristics, with the results reported in [22], some of which are summarized in Table 3.

According to the load characteristics in Table 3 and considering that a container is the smallest service unit in a container cloud environment and a node is the smallest service unit of the container carrier, the resource load situation of the container cloud comprises container and node loads. The latter two components' load values are modeled from a coarse and fine granularity perspective. The time series correlation and high self-similarity in load

characteristics should detect not only the instantaneous load values but also the load values within a time sequence. Indeed, a time series data model should be established to predict the future load value sequence based on the historical load value sequence. Hence, this paper models the container load, node load, and timing data in three aspects.

3.1.1. Container Load Model. The load of a container is dependent on the management kernel services of the node host operating system, as noted in [23]. Despite load average being a measure of each service's load, there exists no direct measurement method for the container load. However, given that CPU and memory are vital computing resources for containers when providing their services and that these parameters are easily measurable, their average utilization value in unit time t can approximate the container's load value. It is critical to consider real-time load as an essential factor in predicting accuracy. A smaller time unit t results in a more instantaneous load value that can be calculated using the following equation:

$$\text{load}_i = \omega_c \times \text{avg}(\text{used}_c) + \omega_m \times \text{avg}(\text{used}_m), \quad (1)$$

where $\text{load}_i i$ denotes the container load value in the i-th unit time, $\text{avg}(*)$ is the mean value operator, used_c and used_m are the CPU and memory utilization sampled values of the node host in a given segment t, respectively, and ω_c and ω_m denote the mean CPU and memory utilization parameters in the load model.

3.1.2. Node Load Model. Shang et al. [24] studied the node load by proposing an improved dynamic load model that selected some characteristic static physical influences. However, the non-signature and uncertainty of these metrics prohibit reflecting the actual load of the resources in a container cloud environment. Therefore, we combine the characteristic dynamic and static factors to model the node hosts' state and actual load availability. The dynamic factors comprise resource availability time h, resource request r, and resource service intensity q [25, 26]. Combining those two factors ensures that the node load values can be obtained quickly and enhances the node's modeling accuracy. Assuming that the container cloud environment is a cluster of n nodes, expressed as $\text{node}_1, \text{node}_2, \ldots, \text{node}_n$, and each node has m resources, then for node i $(1 < i < n)$,

(1) Resource request r is that sum the average number of service requests received by all node. Assume node_i received r_i requests in per unit time, the resource request of cluster R_t is expressed as follows:

$$R_t = \sum_{i=1}^{n} r_i. \quad (2)$$

(2) Resource service intensity q is the ratio of the average time time_a of node_i to complete a service request to the average time interval time_i of the service request. p_i is the parallel service capability of node i, and is expressed as equations (3) and (4):

$$q_i = \frac{\text{time}_a}{\text{time}_i}, \quad (3)$$

$$\text{time}_a = \frac{r}{p_i}, \quad (4)$$

and the resource service intensity Q of the node cluster in unit time t expressed as

$$Q_t = \sum_{i=1}^{n} q_i. \quad (5)$$

Based on the above, we also consider the static factors. Thus, this paper mainly considers the static factors regarding the average CPU utilization, memory, disk I/O, and network bandwidth. Hence, a dynamic weighting algorithm [27] is used to describe the nodes' load state, with the load of node_i per unit time t expressed as

$$\text{load}_i = \begin{cases} \omega_1 c_{\text{u}}^t + \omega_2 m_{\text{u}}^t + \omega_3 d_{\text{u}}^t + \omega_4 n_{\text{u}}^t + 1 & r_t = R_t \text{ or } q_t = Q_t \\ \omega_1 c_{\text{u}}^t + \omega_2 m_{\text{u}}^t + \omega_3 d_{\text{u}}^t + \omega_4 n_{\text{u}}^t + \omega_5 \dfrac{r_t}{R_t} + \omega_6 \dfrac{q_t}{Q_t} & \text{others,} \end{cases}, \quad (6)$$

where c, m, d, and n denote the CPU, memory, disk I/O, and network bandwidth utilization of node_i per unit time t, respectively. ω_1, ω_2, ω_3, ω_4, ω_5, and ω_6 represent the proportion of each influencing factor in the load model, with the parameter values determined by the service type provided by the container.

3.1.3. Temporal Data Model. Since the loads are correlated and highly self-similar time series, we combine the modeling of a container and the node loads presented in previous subsections, with the load timing data over time expressed as

$$L_n = \{(t_1, \text{load}_1), (t_2, \text{load}_2), \ldots, (t_n, \text{load}_n)\}|_{i=1}^{n}, \quad (7)$$

where L_n denotes n sequence periods of container cloud environment load data, n is the sequence length, and load_i is the load value of the container at the i-th unit time t.

3.2. Load Prediction Model and Algorithm. In this section, we detail the load prediction model. Specifically, we leverage the temporal load data obtained in the previous section as input to extract sequence features through a convolutional neural network (CNN) prediction model. We further design

Table 3: Load characteristics.

Load characteristics	Description	Container status or trigger conditions
Randomness	The load curve shows a low average but a very high standard deviation and maximum value. This means that there is enough cycle for the entire cluster to support the external provision of services.	Dynamic and smooth
Volatility	The load on a cluster can remain relatively stable over multiple time windows, and sudden changes occur at the boundaries of that time and cause large fluctuations in subsequent time windows.	Random events are triggered
Time series affinity	Historical time series load values have a large impact on future load values, so forecasting based on historical load values is feasible.	Continuously occupied/continuously idle
High self-similarity	The load curve is self-similar when the Hurst exponent index is high, i.e., similar load curves may occur in different time windows.	Periodic event triggers

a bidirectional gated recurrent network layer BiGRU to continuously capture long-range features. After passing through the BiGRU layer, an attention mechanism layer is implemented to capture interdependencies between long-range features. Finally, a prediction processing algorithm generates load prediction data. We rely on a generic CNN model as the basis of our proposed model framework. However, since the CNN cannot capture long-range features, it is supplemented with a BiGRU recurrent neural network to capture long-range feature sequences. Additionally, we introduce an attention weight layer to enhance the influence of important information. The proposed deep learning network preserves long-range feature sequences during the model training process and enables accurate feature capture from the input sequence. The proposed load prediction model comprises five parts: input layer, CNN layer, BiGRU layer, attention layer, and output layer. Figure 2 illustrates this architecture.

Regarding the model's workflow, first, the load timing data of the container cloud environment are used as the input. Then the CNN's convolution and pooling operations extract local features. After that, the BiGRU layer and the attention layer learn the changing pattern of the load timing data from the local features extracted by the CNN layer and predict the changing trend of the future load conditions. Finally, the output layer provides the prediction results.

3.2.1. CNN Layer. The CNN layer comprises two convolutional layers, two pooling layers, and one fully connected layer, with both convolutional layers being one-dimensional, and the non-linearReLU function is selected as the activation function. Due to the large volatility of the container cloud load time series data and to reduce the model parameters and the risk of overfitting, the maximum pooling method is chosen for both pooling layers. After the above two convolution and pooling operations, the original load timing data will be mapped to the feature space of the hidden layer, which will be transformed by adding a fully connected layer to extract the feature vectors. The Sigmoid function is selected as the activation function in the fully connected layer. Therefore, the output feature vector H_c of the CNN layer is expressed as equations (8)–(12):

$$C_1 = f\left(X \otimes W_1 + b_1\right) = \mathrm{ReLU}\left(X \otimes W_1 + b_1\right), \tag{8}$$

$$P_1 = \max\left(C_1\right) + b_2, \tag{9}$$

$$C_2 = f\left(P_1 \otimes W_2 + b_3\right) = \mathrm{ReLU}\left(P_1 \otimes W_2 + b_3\right), \tag{10}$$

$$P_2 = \max\left(C_2\right) + b_4, \tag{11}$$

$$H_c = f\left(P_2 \times W_3 + b_5\right) = \mathrm{Sigmoid}\left(P_2 \times W_3 + b_5\right), \tag{12}$$

where C_1 and C_2 are the outputs of convolutional layers 1 and 2, respectively, and this output result vector is expressed as $C_i = [C_1, C_2, \ldots, C_n], i = 1, 2$. P_1 and P_2 are the outputs of pooling layers 1 and 2, expressed as $P_i = [C_1, C_2, \ldots, C_m], i = 1, 2$. W_1, W_2, and W_3 are the weight matrices, b_1, b_2, b_3, b_4, and b_5 are the deviations, O denotes the convolution operation, $\max(*)$ is the maximum pooling function, and the CNN output layer is n. Finally, $H_c = [h_{c_1}, h_{c_2}, \ldots, h_{c_2}]$.

3.2.2. BiGRU Layer. The prediction model must capture the data features over a time sequence, and the CNN convolutional neural network has certain defects, i.e., it cannot capture long-range features well. Recurrent Neural Networks combine the input of the current moment with the hidden state of the previous moment by introducing a recurrent connection between the input and the hidden layer, allowing the hidden state to be transmitted continuously in time, which can effectively capture long sequence dependencies [28]. Therefore, we introduce the BiGRU layer (an RNN variant) in the prediction model, which extracts long sequence dependencies from the local feature vectors output from the CNN layer.

Figure 3 illustrates the BiGRU layer comprising the forward and inverse GRU networks. The GRU network mainly comprises update and reset gates, with the former gates controlling the degree of remembering the hidden state and the latter controlling the degree of the hidden state-like information acting on the candidate set. These states combined determine the degree of the hidden state acting on the current hidden state of the previous layer, and when both states are zero, it is only related to the input of the current layer.

For the GRU network, assuming that the current moment is t, the model is expressed as equations (13)–(16):

$$r_t = \sigma\left(\omega_r \cdot \left[h_{t-1}, W_t\right] + b_r\right), \tag{13}$$

$$z_t = \sigma\left(\omega_z \cdot \left[h_{t-1}, W_t\right] + b_z\right), \tag{14}$$

$$\widetilde{h}_t = \tanh\left(\omega_h \cdot \left[r_t * h_{t-1}, W_t\right] + b_h\right), \tag{15}$$

$$h_t = z_t * h_{t-1} + \left(1 - z_t\right) * \widetilde{h}_t \quad t \in [1, m], \tag{16}$$

where r_t is the reset gate at time t, σ is the Sigmoid activation function, ω_r, ω_z, and ω_h denote the weight matrices, $\cdot$ is the dot product, b_r, b_z, and b_h are the bias values, W_t is the local feature vector of the input at time t, and z_t is the update gate at time t. $\widetilde{h}_t$ is the candidate set of the current state, and h_t is the hidden state at time t, which is also the final output vector. The BiGRU model comprising the GRU network is expressed as equations (17)–(19):

$$\overrightarrow{h_t} = \mathrm{GRU}\left(W_t, \overrightarrow{h_{t-1}}\right), \tag{17}$$

$$\overleftarrow{h_t} = \mathrm{GRU}\left(W_t, \overleftarrow{h_{t-1}}\right), \tag{18}$$

$$h_t = \omega_t \overrightarrow{h_t} + \nu_t \overleftarrow{h_t} + b_t \quad t \in [1, m], \tag{19}$$

where $\overrightarrow{h_t}$ and $\overleftarrow{h_t}$ are the hidden states of the forward and inverse GRU model at time t, respectively, the $\mathrm{GRU}(*)$function is the non-linear transformation of the GRU model, ω_t and ν_t are the weights corresponding to the forward and inverse GRU hidden layer states $\overrightarrow{h_t}$ and $\overleftarrow{h_t}$ of

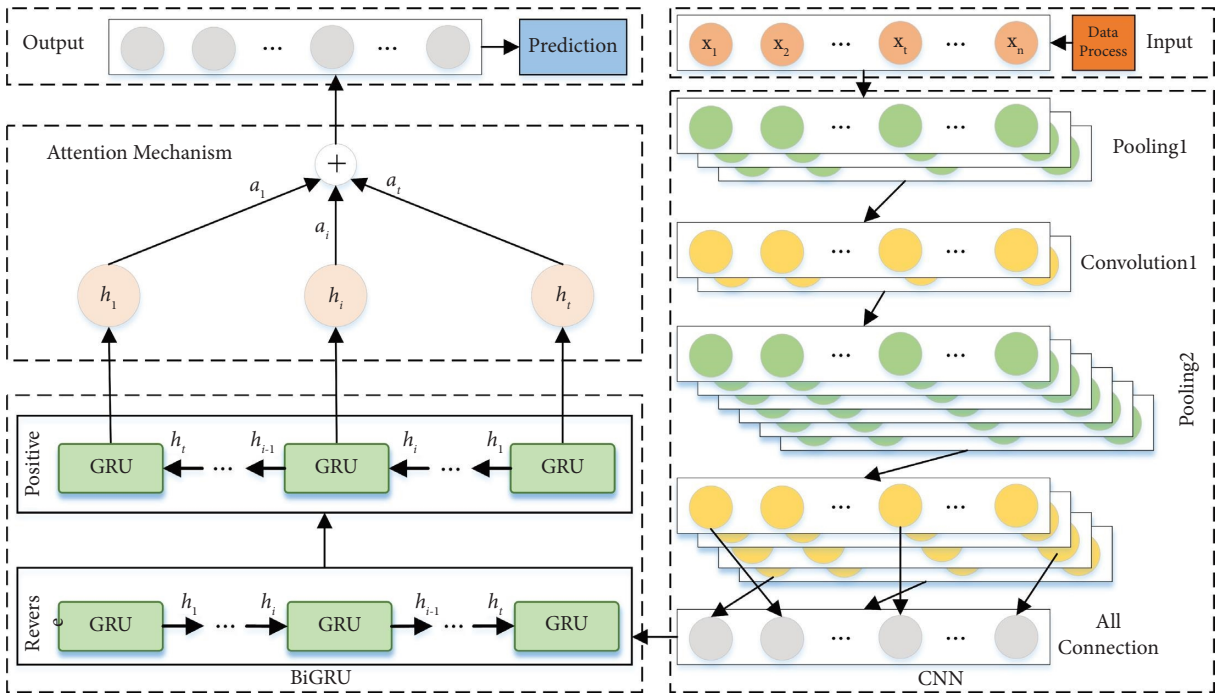

Figure 2: CNN-BiGRU-Attention prediction model architecture.

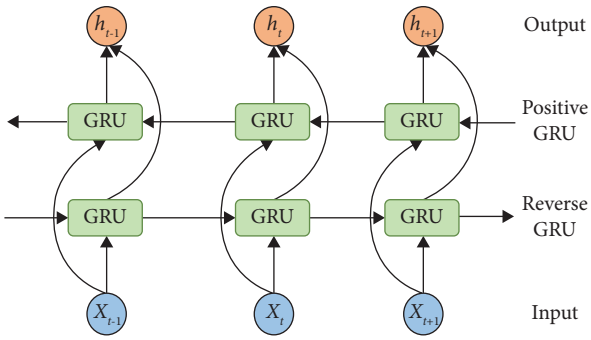

Figure 3: BiGRU model architecture.

BiGRU at time t, and b_t is the bias value corresponding to the hidden layer state at time t. The output of the final BiGRU layer is expressed as $H_t = [h_1, h_2, \ldots, h_t]$.

3.2.3. Attention Layer. The attention mechanism assigns different weights to model input features, enhancing the impact of important information while avoiding information loss for long sequences. It also allows the model to capture long-range interdependent features in the sequence. In this work, the attention mechanism layer learns features and patterns of BiGRU output data by combining multiple structures and iteratively estimating the optimum weight parameter matrix using the weight assignment principle to calculate probabilities of different feature vectors.

Figure 4 illustrates that the input of the attention mechanism layer is the output vector H_t that the BiGRU layer has activated. Moreover, the probabilities corresponding to different feature vectors are computed according to the weight assignment principle, assisting the iteration process to optimize the weight parameter matrix. Assuming that the current moment is t, the weight coefficients of the attention mechanism layer are expressed as equations (20)–(22):

$$e_t = \nu \tanh(\omega h_t + b), \tag{20}$$

$$\alpha_t = \frac{\exp(e_t)}{\sum_{j=1}^{t} \exp(e_j)}, \tag{21}$$

$$s_t = \sum_{t=1}^{i} \alpha_t \cdot h_t, \tag{22}$$

where e_t denotes the value of the attention probability distribution determined by the output vector h_t of the BiGRU layer at moment t. Additionally, ν and ω are the weight coefficients, b is the bias value, and s_t denotes the output of the attention layer at moment t.

3.2.4. Load Prediction Processing Algorithm. The CNN-BiGRU-Attention model's load prediction algorithm predicts future load data in a container cloud environment through multi-step prediction. The algorithm is divided into four parts: initializing flag bits for load overflow and rise, calculating load variation sequence, determining load rise degree, and giving final prediction based on load rise and fall values. Algorithm1presents the proposed algorithm's pseudocode.

3.3. Container Scheduling Strategy. In this section, we describe the scheduling strategy and design the container scheduling strategy CSSLPM (container scheduling strategy

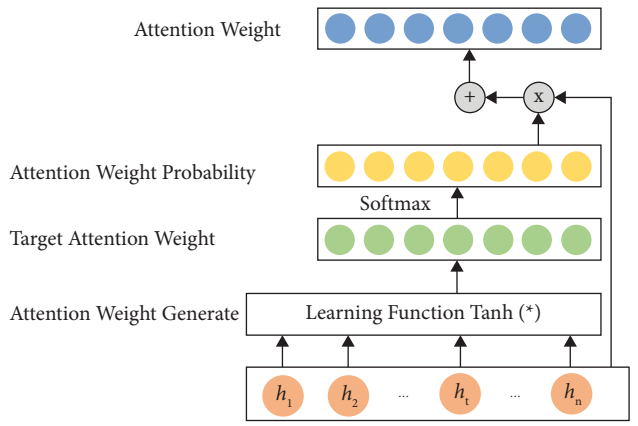

Figure 4: Feedforward attention mechanism architecture.

Input: Prediction list_{pre}, and Load threshold $\langle \text{threshold}_{\text{low}}, \text{threshold}_{\text{high}} \rangle$
Output: Predict the outcome of the processing $\langle \text{list}_{\text{pre}}[0], \text{isExceede d}, \text{isRose}, \text{riseDegree} \rangle$
(1) Initialize isExceeded = false, isRose = false, riseDefree = 0, riseArea = 0, downArea = 0
(2) l length $(\text{list}_{\text{pre}})$
(3) for each i in l
(4) check if $i + 1$ exceeds bounds
(5) $\langle \text{da ta}_a, \text{da ta}_b > < \text{list}_{\text{pre}}[i], \text{list}_{\text{pre}}[i+1] \rangle$
(6) if da ta_{a} grater than da ta_{b}
(7) toAdd (downArea, $\text{data}_a - \text{data}_b$)
(8) else
(9) toAdd (riseArea, $\text{da ta}_b - \text{da ta}_a$)
(10) end if
(11) end for
(12) riseDegree (riseArea – downArea)/$(\text{list}_{\text{pre}}[0] * \text{l})$
(13) checkIsExceeded ($\langle \text{threshold}_{\text{low}}, \text{threshold}_{\text{high}} \rangle$, riseDegree)
(14) isRose riseArea > downArea ? true: false

Algorithm 1: Pseudocode of predictive processing algorithm.

based on load prediction model) involving coarse-grained (node) and fine-grained (container) scheduling, respectively. CSSLPM is based on the container and node predicted load values, where container-level scheduling includes the container expansion or shrinkage calculation, and node-level scheduling includes migration triggering, container selection, node selection, and container migration.

Based on the CNN-BiGRU-Attention prediction model, we design the CSSLPM architecture illustrated in Figure 5, which is based on our proposed load prediction method and utilizes the output scheduling demand as the input of the container scheduling strategy. Depending on the output, CSSLPM first determines whether to select a fine-grained or coarse-grained scheduling strategy, then generates scheduling triggers in the strategy group, and finally the triggers actually do the scheduling work for the container cloud environment. The more accurate the load prediction, the more CSSLPM can solve the problems faced by the container cloud environment in terms of load balancing and resource optimization.

3.3.1. Scalable Flexible Scheduling Method. The scalable flexible scheduling method is a powerful tool for load balancing in container cloud environments. During heavy traffic periods, such as holidays, it can effectively balance the load at the node level based on predicted load values. Elastic scaling ensures that the container cloud remains responsive under sudden increases in load, while also maintaining overall load balance. The scale-out elastic scheduling method consists of three steps: triggering expansion/downsizing based on current and predicted load values; calculating the appropriate replica count using an aggressive but dynamic weighted scheme to prevent overloading; and scheduling with a cooling period to avoid constant scaling operations triggered by predicted values. Using this method, containers can avoid overload and scaling operation problems while maintaining rapid response times during peak load periods.

Hence, we consider the load forecast value, the current resource load value, and the resource threshold, and the dynamic weighting is expressed as

$$f_t = \omega_1 \text{load}_p + \omega_2 \text{load}_c, \tag{23}$$

where load_p is the load prediction value, load_c is the current load value, ω_1 and ω_2 are weights, and f_t is the marker value obtained by dynamically weighting the predicted and the current load values. Meanwhile, the trigger strategy of resilient scheduling includes two trigger cases:

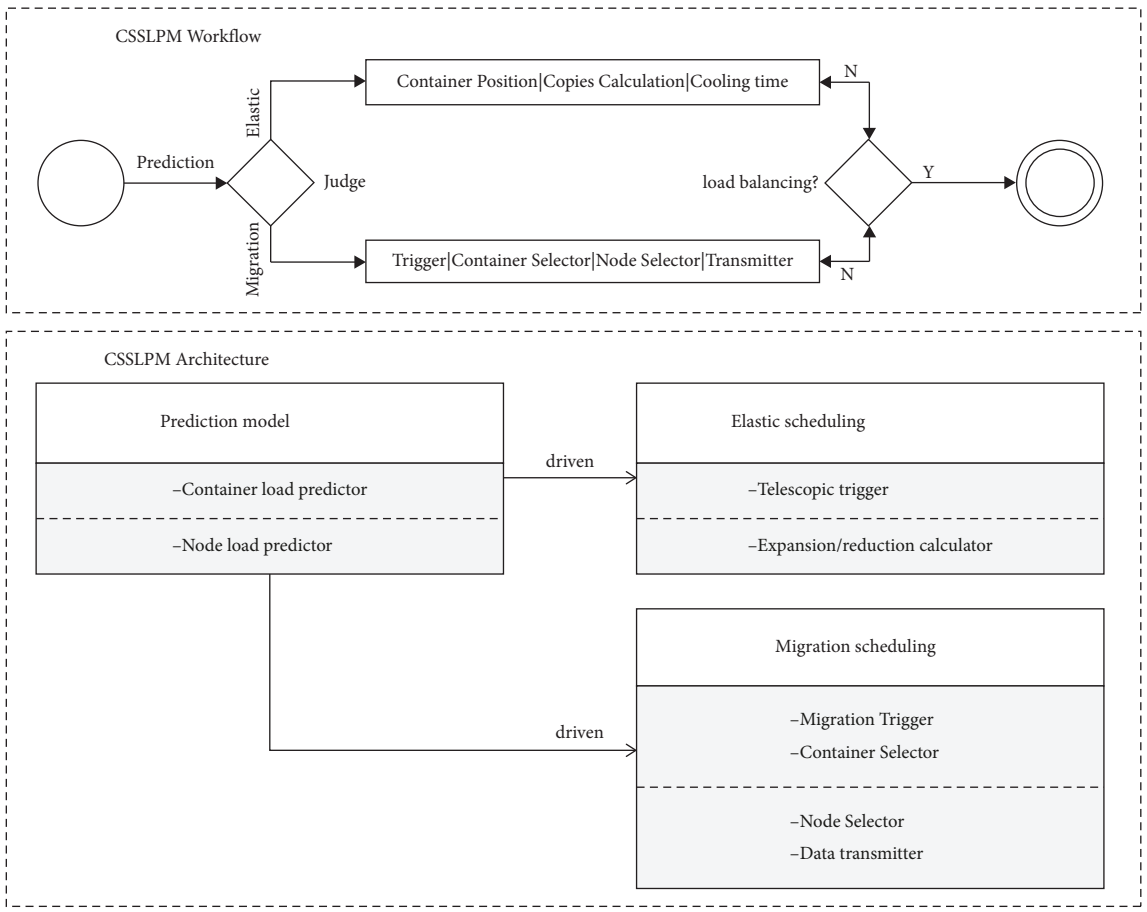

Figure 5: Container scheduling strategy architecture.

(i) *Expansion Trigger*. If the dynamically weighted tag value exceeds the pre-defined expansion threshold, it suggests that the container replica set is likely to face excessive load in the near future, and expansion should be considered to allocate additional resources.

(ii) *Reduction Trigger*. If the dynamic weighted marker value is lower than the pre-defined shrinkage threshold, it suggests that the container replica set is likely to have excess resources in the near future, and a shrinkage operation should be considered to optimize resource allocation.

To avoid triggering multiple expansion or contraction operations quickly, a resilient scheduling cooling phase is designed to judge whether it is in the cooling phase after each trigger to reduce the load volatility impact. The corresponding workflow is illustrated in Figure 6.

The key step in the entire scaling scheduling process is to conduct the replica count calculation process of the containers based on the current and the predicted resource value. Next, we introduce the expansion and scaling replica count calculation schemes to provide accurate data support for flexible scheduling.

(1) Expansion Replica Count Calculation. To avoid under-expansion, we adopt a slightly aggressive strategy in the expansion process, i.e., we select larger predicted and load values for the expansion replica number calculation to reserve sufficient resources for the service. Specifically, we use the mathematical expectation method to calculate the number of replicas which is expressed as

$$R_{\text{exp}} = \text{ceil}\left(\max\left(\frac{\text{load}_{\text{pro}}, \text{load}_{\text{cur}}}{\text{load}_{\text{exp}}}\right)\right) * R_{\text{cur}}, \tag{24}$$

where R_{cur} and R_{exp} denote the number of current and desired replicas, load_{pre} is the load forecast value, load_{cur} is the current load value, and load_{exp} denotes the desired load value.

(2) Reduction Replica Count Calculation. The scaling operation is affected by load volatility, and to reduce the frequent scaling phenomenon, we adopt a dynamic weighting method to determine the number of replicas [29], i.e., the distance weighting formula dynamically adjusts the weights occupied by the load and the predicted values in calculating the desired number of replicas, minimizing the impact of load volatility. The distance weighting is expressed as equations (25) and (26):

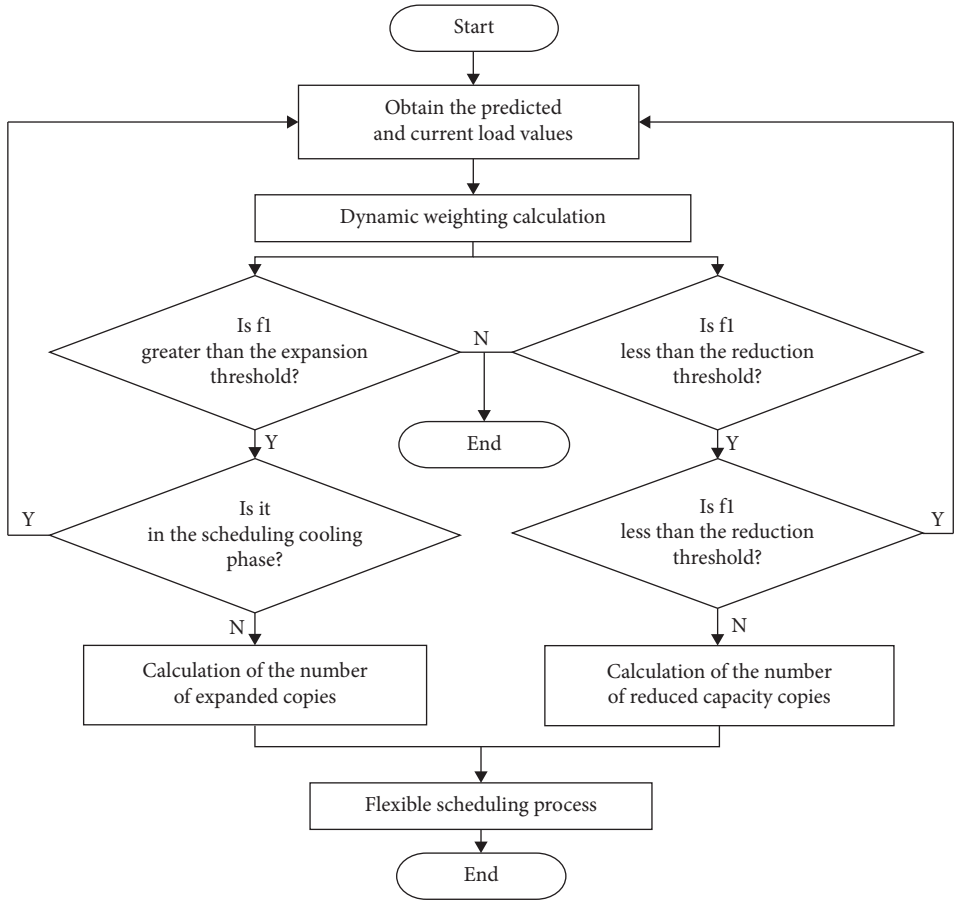

Figure 6: Flexible scheduling trigger determination workflow.

$$W_{\text{pre}} = \frac{\left(\text{load}_{\text{exp}} - \text{load}_{\text{cur}}\right)}{\left(\text{load}_{\text{exp}} - \text{load}_{\text{cur}}\right) + \left(\text{load}_{\text{exp}} - \text{load}_{\text{pro}}\right)}, \quad (25)$$

$$W_{\text{cur}} = \frac{\left(\text{load}_{\text{exp}} - \text{load}_{\text{pro}}\right)}{\left(\text{load}_{\text{exp}} - \text{load}_{\text{cur}}\right) + \left(\text{load}_{\text{exp}} - \text{load}_{\text{pro}}\right)}, \quad (26)$$

where W_{pre} and W_{cur} are the prediction weight and load weight, and load_{pro}, load_{cur}, and load_{exp} satisfy the conditions $\text{load}_{\text{exp}} > \text{load}_{\text{cur}}$ and $\text{load}_{\text{exp}} > \text{load}_{\text{pro}}$. Then, the formula for the weighted resource values is expressed as

$$\text{load}_W = W_{\text{pre}} * \text{load}_{\text{pro}} + W_{\text{cur}} * \text{load}_{\text{cur}}. \quad (27)$$

Thus, the expected replica number $R_e$xp is expressed as

$$R_{\text{exp}} = \text{ceil}\left(\frac{\text{load}_w}{\text{load}_{\text{exp}}}\right) * R_{\text{cur}}. \quad (28)$$

3.3.2. Integrated Migration Scheduling Method. We propose an integrated migration scheduling method for maintaining balanced load in container cloud environments with high-intensity services. Coarse-grained elastic scaling ensures load balance among nodes under continuous high-load pressure, and it continuously adjusts the load of each node.

The suggested method has four phases: container migration triggering, container selection, target node selection, and container migration. In the first phase, we trigger container migration based on the load values of nodes and predicted load values. In the second phase, containers are migrated to achieve dormancy of underloaded and load reduction of overloaded nodes. The third phase uses a load correlation algorithm to select an optimal set of nodes to prevent redundant work. Finally, in the container migration phase, an online pre-merge-based algorithm improves migration efficiency and guarantees success rate.

This integrated method maintains load balance during long periods of high load while ensuring efficient container migration.

(1) Container Migration Triggering. Various resource usages, such as CPU and memory, affect a node's load and fluctuate due to the containers' diversity and dynamics. Therefore, the deployed nodes and containers must meet the following requirements in determining the load balancing conditions.

The containers must be unique, i.e., each container can and will only be deployed on top of a particular node in the container cloud environment which is expressed as

$$\sum_n D_n^m = 1, \forall m \in N, D_n^m \in 0, 1, \quad (29)$$

where $D_n^m = 1$ means container n is deployed to node m. Otherwise, $D_n^m = 0$.

Resource finiteness, which refers to the fact that when multiple containers are deployed on the same node, the total amount of resources required by all containers must not exceed the total amount of resources that the node has, is expressed as

$$\sum_n r_n^R \times D_n^m \le R_m^R, R \in \text{CPU, MEM, Net, Disk}, \tag{30}$$

where r_n^{CPU}, r_n^{MEM}, r_n^{Net}, and r_n^{disk} denote the amount of CPU, memory, network bandwidth, and disk IO resources required by container n and R_m^{CPU}, R_m^{MEM}, R_m^{Net}, and R_m^{Disk} denote the total amount of CPU, memory, network bandwidth, and disk IO resources owned by node m.

Meanwhile, we add the predicted load data to the load balancing model proposed in [30] to evaluate all nodes in the container cloud environment, i.e., from the evaluation results of all nodes, the load and resource of the container cloud can be expressed as equations (31) and (32):

$$U_{\text{avg}}^R = \frac{\sum_{j=1}^{l} U_j^R}{l}, \tag{31}$$

$$F = \frac{\sum_R \sum_{j=1}^{l} \left| U_j^R - U_{\text{avg}}^R \right|}{l}, \tag{32}$$

where U_j^R denotes utilizing resource R in node j, l is the total number of nodes in the container cloud environment, U_{avg}^R is the average resource utilization, and F is the load balance of all nodes in the container cloud environment, ranging from (0, 1) to (0, 1). The larger the value of F, the more unbalanced a load of resources in the container cloud environment, and vice versa.

In the trigger conditions of the container migration, the joint action of various resource conditions causes a node load problem, with the migration triggering factors described as follows. First, the load prediction result determines whether the node exceeds the threshold set. The node will be directly added to the overloaded node collection if it exceeds the threshold. Otherwise, according to the current calculation of the container cloud environment load, if the load balance is less than the set threshold, the underload sorting of nodes is performed, and the corresponding nodes are added to the underload node set. If the load balance exceeds the threshold set, the overload sorting of the nodes is performed, and the corresponding nodes are added to the overload node set. Finally, when both the underload and the overload node sets are not empty, the container migration schedule is triggered, and vice versa. The corresponding process is illustrated in Figure 7.

(2) Container Selection. We describe a method for selecting the container to be migrated. During this process, a major concern is selecting the object to be migrated [31]. Thus, after obtaining the set of nodes to be migrated, the corresponding container must be selected from each node for migration. However, the service container state and the dependencies between the containers are also important

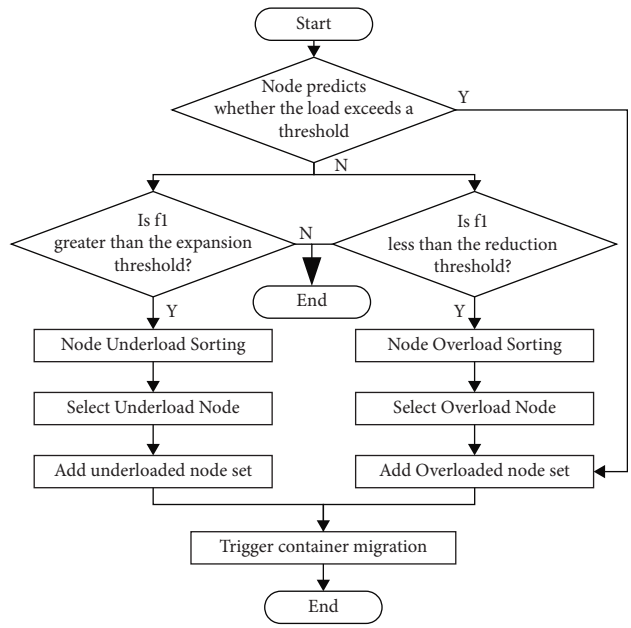

Figure 7: Migration triggering workflow.

factors to be considered in the container scheduling process, which effectively reduces the time consumption of the containers by calling each other. Therefore, we adopt a container selection strategy based on the container dependency model [17].

The container dependency model uses a weighted directed graph to represent the dependency invocation relationship between the containers on a node belonging to the set to be migrated and is expressed as equations (33) and (34):

$$G(V, E) = \left(\omega_{\text{ij}}\right)_{n \times n}, \tag{33}$$

$$\omega_{ij} = \begin{cases} 0, & \text{no call from container } i \text{ to container } j, \\ k, & k \text{ calls from container } i \text{ to container } j, \end{cases} \tag{34}$$

where w_{ij} denotes the set of all containers on the node, w_{ij} denotes the dependency relationships between all containers on the node, and w_{ij} is the number of invocations from container i to container j. Thus, after the containers' dependency partitioning, we obtain k disjoint dependency neighborhoods, represented by V_i $(0 < i <= k)$. Then, the synthesis of all container invocations in neighborhood V_i is expressed as

$$\omega\left(V_i\right) = \sum_{V \in V_i} \omega(V). \tag{35}$$

Similarly, the sum of the number of calls between all containers in neighborhood V_i and all containers in the neighborhood V_k is expressed as

$$\text{gain}\left(V_i, V_k\right) = \sum_{i=1}^{m} \omega\left(V_j\right) \otimes \omega\left(V_k\right). \tag{36}$$

The container selection strategy is shown as follows:

(i) *Node Underload*. For underloaded nodes with small containers, their resource utilization is low. To optimize resource allocation, these containers should be migrated separately to other nodes that are not overloaded or underloaded. This will facilitate subsequent migration and reallocation of resources, ultimately allowing underloaded hosts to hibernate, resulting in container consolidation, improved resource utilization, and reduced energy consumption.

(ii) *Node Overload*. To reduce node load on overloaded nodes, containers must be filtered and selected for container scheduling. This involves dividing each overloaded node into dependency neighborhoods using the container dependency model. Next, several dependency neighborhoods with the smallest sum gain(V_i, V_k) are chosen from each node's dependency neighborhoods. Finally, the containers from these chosen neighborhoods are added to the set of containers to be migrated in the next step of container migration scheduling.

(3) Target Node Selection. The traditional target node selection algorithm simply considers the relationship between containers in the process of container migration, and there is a risk that container migration causes the target node to be overloaded to trigger the migration work several times. Therefore, we propose a target node selection algorithm based on load correlation (NSALC), in which the Pearson correlation coefficient method is introduced as the main method to calculate the load correlation between containers and nodes. In the process of container migration target node selection, the constraint of load correlation is added, which can effectively solve the problem of migration leading to node overload and thus frequent triggering of migration. Therefore, the resource is expressed as

$$\mathrm{Cor}^{\mathrm{res}}_{C_a,N_k} = \frac{\sum_{m=1}^{n}\left(\mathrm{res}^{m}_{C_a} - \overline{\mathrm{res}_{C_a}}\right)\left(\mathrm{res}^{m}_{N_k} - \overline{\mathrm{res}_{N_k}}\right)}{\sqrt{\sum_{m=1}^{n}\left(\mathrm{res}^{m}_{C_a} - \overline{\mathrm{res}_{C_a}}\right)^2 \sum_{m=1}^{n}\left(\mathrm{res}^{m}_{N_k} - \overline{\mathrm{res}_{N_k}}\right)^2}}, \quad (37)$$

where $\mathrm{res}^{m}_{C_a}|m = 1, 2, 3, \ldots, n$ denotes the load history timing data of container C_a in n time windows, $\overline{\mathrm{res}_{C_a}}$ is the average value of load history timing data of container C_a, $\mathrm{res}^{m}_{N_k}|m = 1, 2, 3, \ldots, n$ denotes the load history timing data of node N_k in n time windows, and $\overline{\mathrm{res}_{N_k}}$ denotes the average value of load history timing data of container N_k.

The load correlation between container C_a and node N_k is expressed as

$$\mathrm{Cor}_{(C_a,N_k)} = \omega\left|\mathrm{Cor}^{\mathrm{CPU}}_{(C_a,N_k)}\right| + \omega\left|\mathrm{Cor}^{\mathrm{MEM}}_{(C_a,N_k)}\right|, \quad (38)$$

where ω is the weight, which is used to set the node selection threshold, and $\mathrm{Cor}_{(C_a,N_k)} \in (0, 1)$.The load correlation Col_r between the set of containers to be migrated and node N_k for a set of n containers is expressed as

$$\mathrm{Cor}_{(\mathrm{Col}_r,N_k)} = \frac{\sum_{i=1}^{N}\mathrm{Cor}_{(C_i,N_k)}}{n}. \quad (39)$$

A node is selected when the load correlation between the set of containers to be migrated and the node exceeds $\mathrm{Cor}_{(\mathrm{Col}_r,N_k)}$. In summary, the pseudocode of the load correlation-based node selection algorithm (NSALC) is presented in Algorithm 2.

(4) Container Migration Phase. Next, we describe the working process of container migration, comprising three parts. First, a full check of the containers to be migrated on the source node is performed before container migration, and the memory image file of the containers before migration is obtained. Then, incremental checks are continuously performed on the containers to be migrated during the container migration process, and the memory image file of the container at migration time is obtained. Finally, the two memory image files are pre-merged and transferred to the target node, and the containers to be migrated to the source node are stopped by restoring the pre-migration state of the containers according to the memory image files.

The main factors affecting a container's online migration downtime include the checkpointing, transfer, and recovery phases. Thus, reducing the time consumed by these three phases is the key to reducing downtime. Hence, this paper adds a fast memory synchronization optimization design to the traditional migration method. Before restoring the container to normal operation in the recovery phase, the memory image files are pre-merged, i.e., the memory image files received on the target node are pre-merged, and the final memory image files are generated. In short, in the subsequent recovery phase, the recovery process of the container can be completed only by obtaining the final generated merged image file. Additionally, the process of merging memory image files and reducing downtime can be conducted through the transmission process.

For the containers, the generated memory image files can be roughly divided into two categories: pages.img and pagemap.img files. The former file type is mainly used to store the specific content of memory pages, and the latter type is mainly used to store the memory mapping relationship. pagemap1.img file in Figure 8 indicates that the first four memory pages are read from pages1.img file and placed at address 0x1000000. When an incremental checkpoint is executed, a flag bit in_parent is added to the pagemap.img file which indicates that the identified memory page was read from the previous checkpoint.

The role of the state check is to dump the process state information of the container to be migrated into an image file, either fully or incrementally. This operation requires using a call function to the ptrace interface to inject parasite code into the container's processes and enable the collection of memory data for the container's processes. Additionally, this operation relies on the /proc file system, with the image file mainly including file description information, process parameter information, and memory mapping information. The specific flow of the container state detection point is as follows:

(i) Recursively traversing /proc/$ pid/task/ and /proc/$ pid/task/$ tid/children based on the container

Input: Container collection to be relocated Col_r Node Collection Col^N
Output: Target Node Collection Col_r^N
(1) for each Col_r^i in Col_r
(2) for each N_k in Col^N
(3) for each C_a in Col_r^i
(4) $\mathrm{Cor}_{(\mathrm{Col}_r,N_k)}\mathrm{Cor}_{(\mathrm{Col}_r,N_k)}$ + calculate $\mathrm{Cor}_{(C_a,N_k)}$
(5) end for
(6) $\mathrm{Cor}_{(\mathrm{Col}_r,N_k)}\mathrm{Cor}_{(\mathrm{Col}_r,N_k)}/n$
(7) if $\mathrm{Cor}_{(\mathrm{Col}_r,N_k)} > 0.6$
(8) $\mathrm{Col}_r^{N_r} N_k$
(9) end if
(10) sort $\mathrm{Col}_r^{N_k}$ by $\mathrm{Cor}_{(\mathrm{Col}_r,N_k)}$
(11) Col_r^N max $(\mathrm{Col}_r^{N_r})$
(12) end for
(13) end for

Algorithm 2: Pseudocode of NSALC.

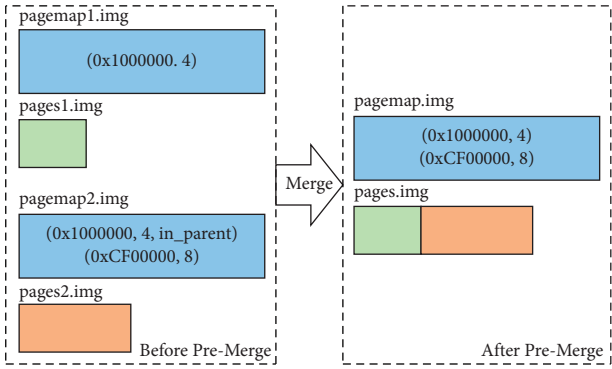

Figure 8: Memory pre-merge workflow.

process pid to collect information about the process tree constructed from the container process and its child processes.

(ii) Freezing the process tree in step 1 by calling the PTRACE_SEIZE command of the ptrace interface.

(iii) Collecting file descriptions, process parameters, memory maps, and other information about the container process and its child processes and writing them to the corresponding memory image files.

(iv) Dynamically injecting the parasite code in PIE format into the container and child processes. When the mmap operation is invoked by the container process and its child processes, the parasite code will be invoked into the corresponding memory address space and the memory changes will be recorded.

(v) Executing the rt_sigreturn() system call by calling the ptrace interface to clean up the parasite code injected in the appeal step.

The pseudocode of the pre-copy-based container online migration algorithm is presented in Algorithm 3.

4. Experimental Analysis

This section presents the experimental results evaluating a container scheduling strategy based on the CNN-BiGRU-Attention model and using the CSSLPM policy. First, we describe the experimental setup, present the actual results, and then analytically discuss the findings.

4.1. Experimental Design

4.1.1. Benchmarking Methods and Choice of Comparison Algorithm. The prediction effectiveness of the CNN-BiGRU-Attention model is evaluated based on MAPE (mean absolute percentage error), MAE (mean absolute error), MSE (mean square error), and RMSE (root mean square error). Moreover, based on these evaluation methods, the ARIMA model, the LSTM-RNN model, and the CNN-LSTM model are selected as the CNN-BiGRU-Attention model control group. These models are chosen because they are good predictors and can be used in various applications, which are expressed as equations (40)–(43).

$$\mathrm{MAPE} = \frac{100\%}{n}\sum_{i=1}^{n}\left|\frac{\mathrm{load}_i - \widehat{\mathrm{load}_i}}{\mathrm{load}_i}\right|, \tag{40}$$

$$\mathrm{MAE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\mathrm{load}_i - \widehat{\mathrm{load}_i}\right)^2}, \tag{41}$$

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(\mathrm{load}_i - \widehat{\mathrm{load}_i}\right)^2, \tag{42}$$

$$\mathrm{RMSE} = \frac{1}{n}\sum_{i=1}^{n}\left|\mathrm{load}_i - \widehat{\mathrm{load}_i}\right|. \tag{43}$$

We utilize CloudSim, a popular simulation software program for evaluating virtual technology-based scheduling algorithms, to validate the effectiveness and efficiency of the

Input: to be migrated node Pod_x, Target node Pod_y, Global Checkpoint Ω_c, Incremental checkpoints Δ_c, Iteration number N, Threshold L
Output: Target node Pod_y
(1) ImitializenCheak Ω_c = Image (Pod_x and container)
(2) Transfer (Pod_y, Ω_c)
(3) for each i in N
(4) Δ_c = Image (Pod_x and runningcontainer)
(5) if ($\Delta_c > L$)
(6) break/n
(7) Transfer (Pod_y, Δ_c)
(8) end for
(9) Stop (Pod_x and container)
(10) Transfer (Pod_y, Δ_c)
(11) Merge ($\Omega_c, \sum \Delta_c$)
(12) Run (Pod_y and container)

Algorithm 3: Pseudocode of container online migration algorithm.

container scheduling strategy (CSSLPM) that is based on a load prediction model. In addition, we compare CSSLPM with classic container scheduling strategies such as FFHS [32], MOPSO [33], and Spread [34], which are used as a control group.

4.1.2. Research Questions. The following research questions are answered to verify the accuracy of the CNN-BiGRU-Attention model and the effectiveness of the CSSLPM.

RQ1. Is the CNN-BiGRU-Attention model significantly superior to other load prediction models, and is there hyperparameterization in the CNN-BiGRU-Attention model?

RQ2. Is CSSLPM efficient in scheduling containers based on load prediction? Does it ensure better container cloud resource utilization than other scheduling strategies?

RQ3. Does the container scheduling strategy based on the CNN-BiGRU-Attention prediction model still have better prediction results on real-running software systems? Is it able to maintain container cloud load balancing?

4.1.3. Experimental Datasets. We employ the Alibaba public dataset cluster-trace-v2018, with further information listed in Table4. Specifically, the cluster-trace-v2018 dataset was produced by tracking operational data from approximately 4000 machines over eight days. It has a larger scaler than the previous v2017 version and contains DAG information for production batch workloads.

Moreover, we use TrainTicket, a benchmark system developed by Professor Xin's team at Fudan University, to validate our scheduling strategy. This microservice-based system reflects a real train ticket ordering system in a production environment and consists of 41 microservices, including 24 business logic services and 17 infrastructure services. The system can be run in a small container cloud environment. In addition, 22 typical industrial microservice failure cases are replicated in the TrainTicket system and can be verified directly without the need for injection in this paper.

Table 4: Cluster-trace-v2018 datasets.

Name	Description
machine_meta	Meta and event information for the machine
machine_usage	Resource usage per machine
container_meta	The container's meta and event information
container_usage	Resource usage per container
batch_instance	Instance information in batch workloads
batch_task	Instance information in batch workloads

4.1.4. Operating Environment Configuration. The corresponding training and simulation experiments were conducted for the designed load prediction and container scheduling techniques. In particular, the hardware device details are listed in Table 5, and the utilized software and version information is presented in Table 6.

4.2. Algorithm Validity Testing

4.2.1. CNN-BiGRU-Attention Model Prediction Accuracy Test. This section aims to validate the prediction accuracy of the CNN-BiGRU-Attention model and compare it with similar models using a cross-validation process. The process includes optimizing the model hyperparameters using a multi-layer iterative search network implementation. The optimized model is then used for comparative experiments against competitor models under the same load prediction conditions. The results are analyzed to draw clear conclusions and answer RQ1.

(1) Analysis of Model Hyperparameter Selection. Several key parameters in neural network-based prediction algorithms can significantly impact prediction accuracy. These parameters generally include the length of each input load, the number of hidden layers, and the number of neurons per

Table 5: Hardware configuration.

Class	Parameter information
CPU	Intel (R) Xeon (R) Silver 4210R 2.4 GHz
Threads	20
CPU cores	10
Memory	128 GB
GPU	GeForce RTX 3080
GPU's memory	10 GB

Table 6: Software configuration.

Class	Parameter information
Programming language	Python (3.6.8), Java(1.8.0_191)
Compiler	PyCharm 2021.2, IntelliJ IDEA 2021.3
Toolkit	TensorFlow-gpu (2.1.4), Keras (2.3.1), Numpy (1.22.3), pandas (1.1.5), CUDA (11.4.141)

hidden layer. In this experiment, the length of each input load data and the number of hidden layers are set. Additionally, the number of neurons in each hidden layer was preserved to reduce the difficulty of this experiment. Finally, the maximum number of iterations during training was set to 240, the tensor dimension in the attention mechanism layer was set to 64, and the error was appropriately set due to hardware considerations, affecting the prediction accuracy.

Therefore, we use a cross-validation method to obtain the optimum parameters based on a multi-layer iterative search network implementation. The best parameters are also based on the model parameters, prediction accuracy, and model training time. In this paper, the best parameter configurations are obtained by ranking parameter setups in descending order according to their prediction accuracy. The experimental results are illustrated in Figures 9 and 10, where the horizontal coordinates indicate the parameter combinations. For example, 32-2-50 means that the length of each input load datais 32 bytes, the number of hidden layers is 2, and the number of neurons in each hidden layer is 50.

From the above experimental results, we observe that the training time used by the CNN-BiGRU-Attention model is longer when the data length of each input load is 16, the number of hidden layers is 3, and the number of neurons in each hidden layer is 40. However, in this case, the pre-error is relatively low. In this paper, we consider that this is the best-performing experimental model. Hence, the load versus the predicted values using this parameter combination are depicted in Figure 11, where the solid red curve indicates the actual load of the node, and the blue dotted curve indicates the predicted load value of the CNN-BiGRU-Attention model. The results infer that the CNN-BiGRU-Attention model suffers from hyperparameterization, answering RQ1. The test results show differences between the predicted and load value curves, which do not fit well, but the curve trends and magnitudes are the same, and a better load prediction is possible.

(2) Comparison of Experimental Results. This study compares the load prediction error of the CNN-BiGRU-Attention model to that of ARIMA, DBN, and CNN-LSTM models under the same experimental conditions. Load series of 5, 10, 15, and 20 minutes are used as input data, and the MAE, MAPE, and RMSE metrics are

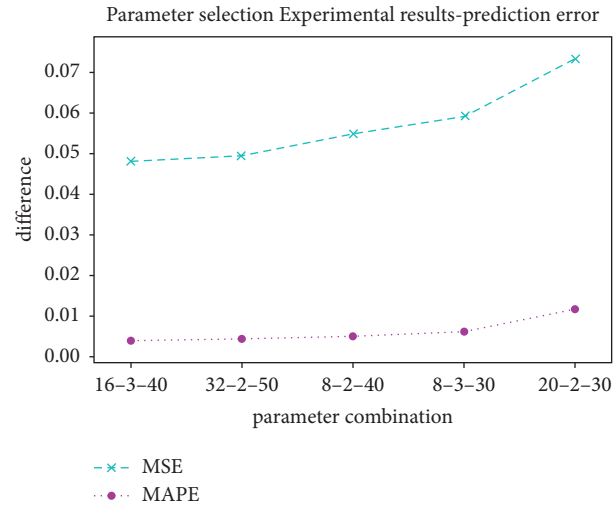

Figure 9: Prediction error.

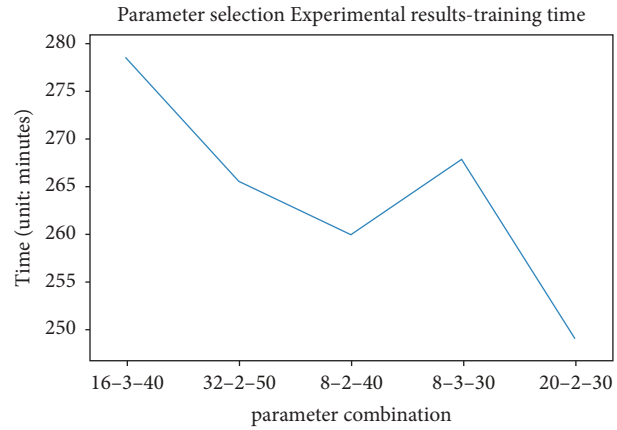

Figure 10: Training time.

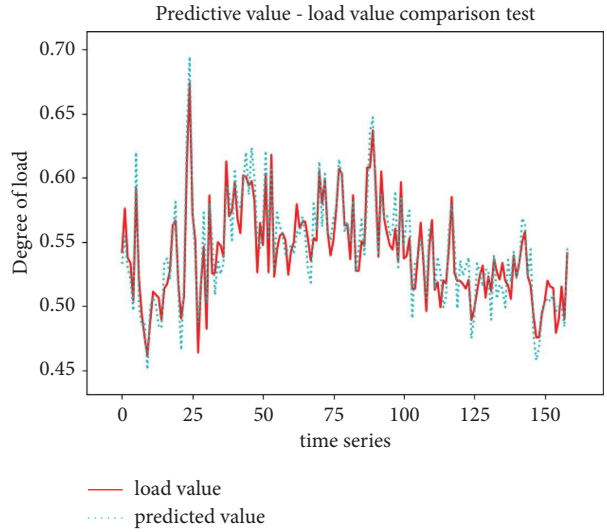

Figure 11: Predictive value-load value comparison test.

employed to measure prediction error. The results are presented in Figures 12–14.

Figure 12 illustrates the experimental results using MAE, where the horizontal coordinate is the length of the time series, and the vertical coordinate is the absolute mean error value of the load prediction values. Figure 12 highlights that the average absolute prediction error of the container and node loads using the competitor models increases the length of the load time series. However, the load prediction algorithms based on the CNN-BiGRU-Attention model have smaller average absolute errors for each length of the load sequence, indicating that the overall difference between the predicted and load values is small.

The evaluation results of this experiment using MAPE are shown in Figure 13, where the horizontal coordinate is the length of the time series, and the vertical coordinate is the average absolute percentage error value of the load prediction values. The figure reveals that the average absolute percentage prediction error of the container and node loads using the competitor models increases as the length of the load time series increases. Nevertheless, the load prediction algorithms based on the CNN-BiGRU-Attention model have smaller average absolute percentage errors for all lengths of load sequences, suggesting that the error values have a smaller ratio to the load values and the errors are relatively low.

The results of the evaluation of this experiment using RMSE are shown in Figure 14, where the horizontal coordinate is the length of the time series, and the vertical coordinate is the root mean square error value of the load predictions. As seen from the figure, the RMSE of the predictions of both container load and node load by various models tends to increase as the length of the load time series increases. In comparison, the load prediction algorithms based on the CNN-BiGRU-Attention model have smaller root mean square errors for each length of the load sequence, which indicates that the overall difference between the error values and the load values is small.

In summary, the errors of all competitor models tend to increase as the time length of the load sequence increases. However, the prediction error growth of the CNN-BiGRU-Attention model gradually slows down. Moreover, the prediction accuracy of the load value within 20 minutes for the CNN-BiGRU-Attention model improves by about 23.1%–93.2% compared to ARIMA, CNN-LSTM, and DBN under the MAE, MAPE, and RMSE metrics. This indicates that the model is better in load prediction in the container-based cloud environment comparison experiments and demonstrates better prediction accuracy for both the container and node load prediction. The CNN-BiGRU-Attention model performs better in similar model comparison experiments, answering RQ1.

4.2.2. CSSLPM Container Scheduling Strategy Validation. In this section, CloudSim, a cloud computing simulation software program widely used in academia to test and evaluate scheduling algorithms based on virtual technologies, is used to answer RQ2, i.e., validate the effectiveness and scheduling efficiency of the container scheduling strategy (CSSLPM) based on the load prediction model. First, this paper extends various simulation interfaces and builds a container model, enabling its cloud data center to support the container-based scheduling simulation process. Second, to simulate the heterogeneous nature of cloud data centers, this paper sets up various server types for conducting this experiment. The start/stop latency of the containers and virtual machines is also set at the second and minute levels. Finally, the current scheduling strategies commonly used in industry and academia are compared with the CSSLPM scheduling strategy. Additionally, five different types of multiple servers are set up to simulate the heterogeneity of cloud servers to run container clusters. The configuration information of each server is listed in Table 7.

First, based on the cluster size, this experiment sets up three different sizes of container clusters for testing, i.e., ten-

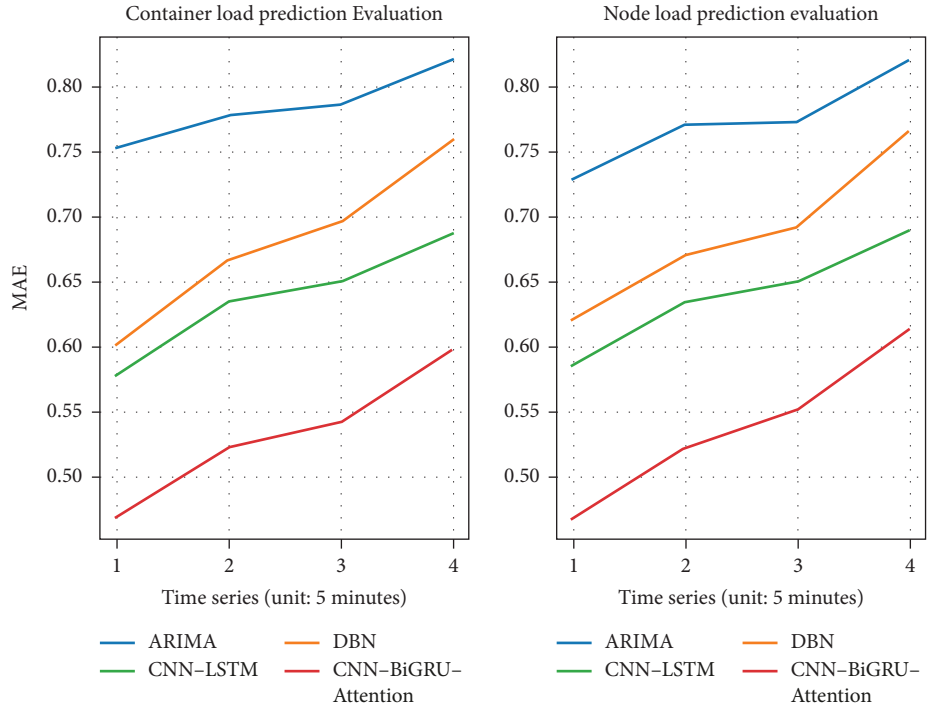

Figure 12: Results of MAE.

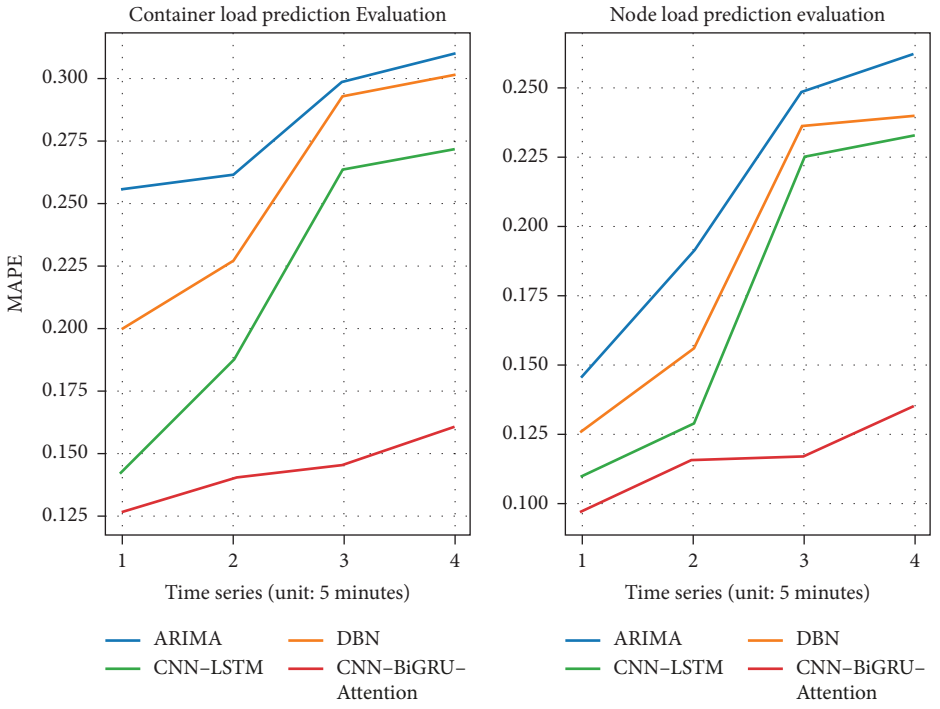

Figure 13: Results of MAPE.

volume, hundred-volume, and thousand-volume covering small-, medium-, and large-scale application scenarios. The container cluster configuration information is presented in Table 8.

Second, for the virtual machines (nodes) and containers used in this experiment, four different types of virtual machines (nodes) and containers are set up, with the specific configuration information reported in Tables 9 and 10, respectively.

The First Fit Host Selection Algorithm (FFHS), Multi-Objective Particle Swarm Optimization (MOPSO), and Spread are selected as the CSSLPM control group. In the

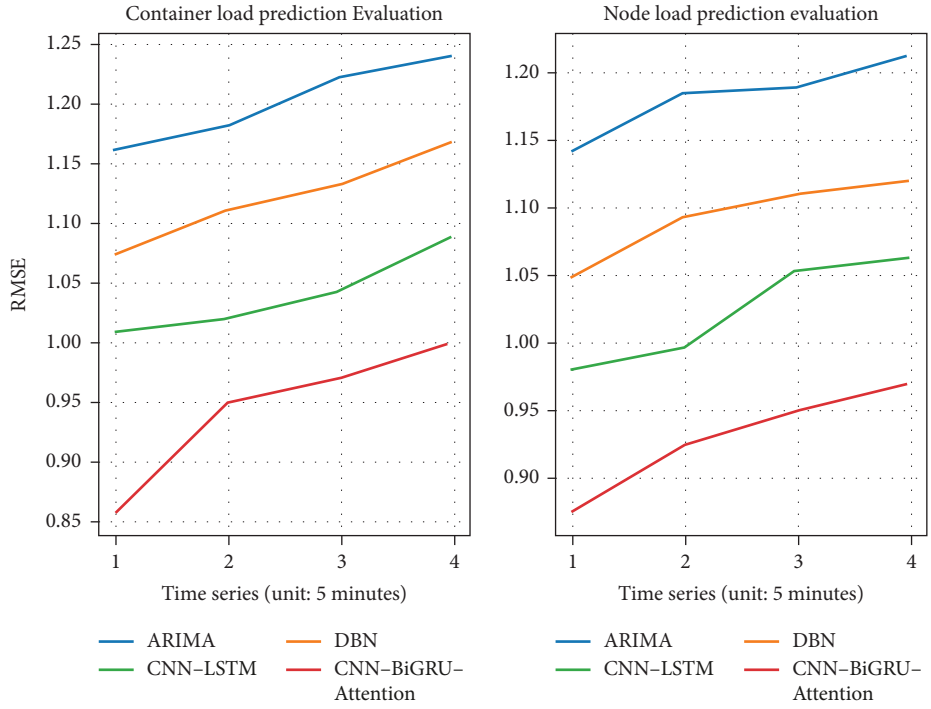

Figure 14: Results of RMSE.

same experimental environment, all three sets of experiments included in this experiment are executed 30 times, and the model uses the optimal combination of parameters. The experimental results include resource usage in the cluster, the average number of containers migrated in 5 minutes, and the average number of nodes created.

(1) Use of Resources. Resource usage's main concern for the same container cluster is achieving relative load balancing on heterogeneous servers. In this experiment, multiple servers of five different types are set up. Therefore, the resources of the same type of servers are calculated together to observe the execution of each scheduling strategy. The results are illustrated in Figures 15(a)–15(c).

The test results reveal that the total server load in the FFHS, Spread, and MOPSO scheduling strategy environments is significantly higher than the total load in the CSSLPM environment for each level of container cluster of this experiment.

(2) Container Scheduling Situation. For the same container cluster, container scheduling focuses on the number of containers scheduled and the number of nodes created within the cluster during the observed time. It further considers the efficiency of the scheduling strategy in achieving relative load balancing of the cluster during the observed time. The results are shown in Figures 16(a) and 16(b).

From the side-by-side comparison of the above test results, it can be seen that CSSLPM is 4.6%–25.8% more efficient than FFHS, Spread, and MOPSO in terms of the number of containers scheduled and nodes created, respectively, in the same cluster environment. Therefore, compared with FFHS, Spread, and MOPSO scheduling strategies, the CSSLPM designed in this paper has better performance regarding load resource usage and container scheduling. This also shows resource fragmentation and uneven load across servers in the FFHS, Spread, and MOPSO scheduling environments. There are two main reasons for this: first, the three scheduling strategies do not consider the interdependencies between the containers. This may cause containers with interdependencies to be migrated to different servers, resulting in cross-server scheduling of service requests and increasing the load on the servers. Second, these three scheduling strategies do not consider the fine-grained scheduling of resources, resulting in fragmented server resources not being utilized effectively. In contrast, CSSLPM combines coarse and fine granularity to achieve container scheduling. On the one hand, container migration-based scheduling uses nodes as the scheduling unit, solving the load imbalance problem from a coarse-grained perspective, and, during the process, aims to migrate containers with interdependent relationships to the same server, thus reducing the frequency of cross-server scheduling of service calls. On the other hand, the scheduling based on container elasticity scaling uses containers as the scheduling unit. This solves the problem of server resource fragmentation from a fine-grained perspective and further improves the utilization of cluster resources.

4.2.3. Case System Validates Algorithm Effectiveness. In this section, the open-source case system TrainTicket is used to answer the research question RQ3. The trained load

Table 7: Server configuration.

Number	Cores	MIPS/core	Memory (GB)	Bandwidth (Gbps)	Storage (TB)
T1	44	2200	128	10	2
T2	16	2200	64	10	2
T3	36	2300	24	10	2
T4	36	2300	64	10	2
T5	16	2200	24	10	2

Table 8: Container cluster configuration.

Number	Containers	Nodes
Cluster 1	64	5
Cluster 2	256	15
Cluster 3	1024	45

Table 9: Node configuration.

Number	Cores	MIPS/core	Memory (GB)	Bandwidth (Gbps)	Storage (GB)
VM1	2	1500	1	10	20
VM2	4	1500	2	10	20
VM3	1	1500	4	10	20
VM4	8	1500	1	10	20

Table 10: Container configuration.

Number	Cores	MIPS/core	Memory (GB)	Bandwidth (Gbps)	Storage (GB)
Container 1	1	100	1	10	5
Container 2	1	200	2	10	5
Container 3	1	400	4	10	5
Container 4	1	600	2	10	5

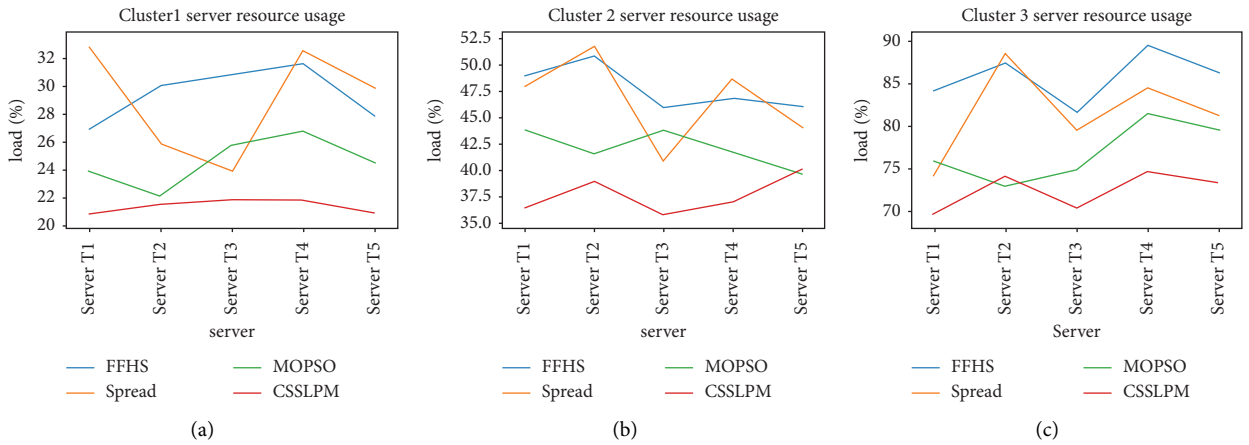

Figure 15: Result of resource usage. (a) Resource usage in cluster 1. (b) Resource usage in cluster 2. (c) Resource usage in cluster 3.

prediction model and the developed and implemented container scheduling strategy are embedded in the system and compared with the model and policy selected as the control group in experiments.

We first conduct a runtime load monitoring process based on the container cluster supporting the TrainTicket ticketing system. Furthermore, we obtain the load data of each node from this process (part of the load data is reported in Table 11), which is used to validate the load prediction algorithm and container scheduling strategy.

Based on the above load data, these data are input into the CNN-BiGRU-Attention model and the control group model for load prediction. The parameters of the CNN-BiGRU-Attention model are set as follows: the length of each input load data is 16, the number of hidden layers is 3, and the number of neurons in each hidden layer is 40. Figure 17

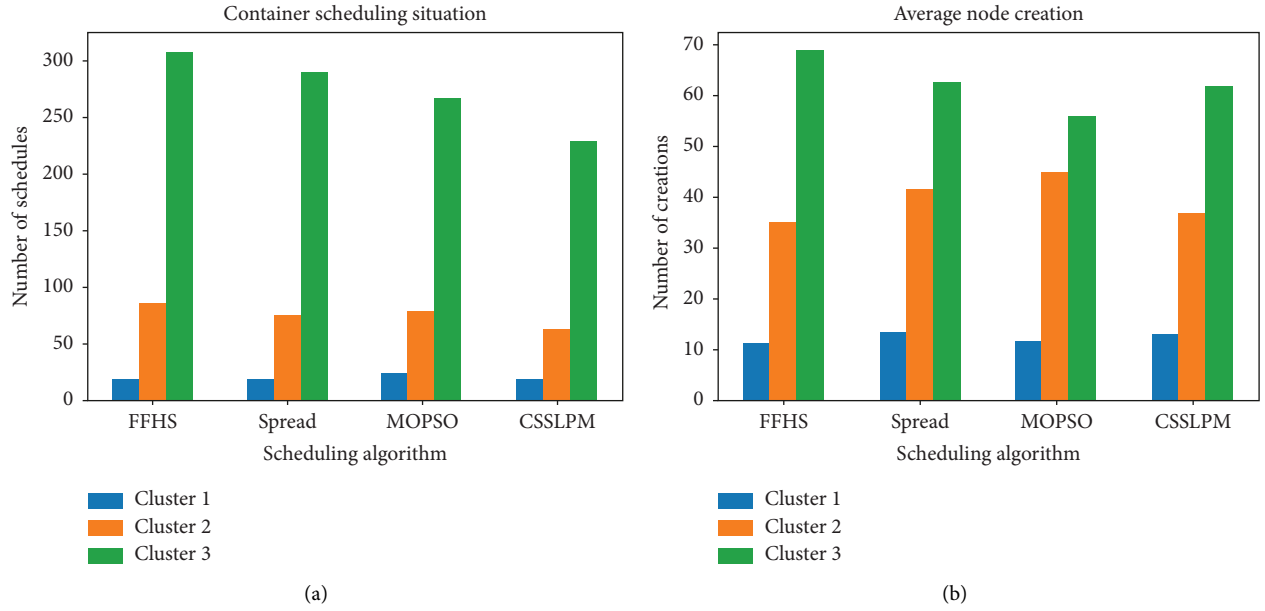

Figure 16: Result of container scheduling. (a) The average number of containers migrated in 5 minutes. (b) The average number of times the container scales in 5 minutes.

Table 11: TrainTicket ticket system load data within 20 seconds.

node_id	time_stamp	cpu_util_percent	mem_gps	disk_usage_percent	disk_io_percent
n_1932	17:23:51	32.25	91.17	53.27	67.84
n_1932	17:23:54	32.64	92.65	57.12	73.31
n_1932	17:23:57	35.38	90.81	54.77	74.67
n_1932	17:23:00	34.64	89.76	55.47	77.91
n_1932	17:24:03	36.29	89.38	57.82	63.93
n_1932	17:24:06	34.83	90.15	59.77	61.57
n_1932	17:24:09	38.68	92.46	49.42	59.38

shows the prediction results of some of the intercepted load data. The solid black line represents the real load data, the red dashed line is the prediction value of the method designed in this paper, and the rest are the prediction values of the control model.

As seen in Figure 17, both the CNN-BiGRU-Attention model and the control model can predict the load in the setting of this experimental setup, and the trend of each predicted value is the same. However, the CNN-BiGRU-Attention model affords the best fit, although it does not achieve a perfect fit in terms of prediction results. In order to verify the accuracy of each load prediction model more intuitively, we employ the MAE, MAPE, and RMSE prediction error metrics for each prediction model and node load. The corresponding results are reported in Table 12.

In addition, based on the above prediction results, the effectiveness of the CSSLPM policy is further validated. A five-virtual machine environment is created in the server, and a multi-node container cluster is built in this environment to support the operation of the TrainTicket passenger ticket system. To analyze the experimental results, we utilize the results from the experiment (2). In particular, Figure 18 shows the resource usage of each VM, and Figure 19 compares the container scheduling results.

For the same experimental setup environment, Figure 18 highlights that the resource load profile of each VM under the designed CSSLPM policy is better than the load profile under the control group, where the VMs have the same attributes, and there is no resource heterogeneity. As seen in Figure 19, the number of scaling and migrations is lower for the designed CSSLPM policy compared to the control group, and therefore, the proposed CSSLPM achieves better load balancing and resource optimization with less scaling and fewer migrations.

4.3. Analysis of Validity Threats

4.3.1. Internal Threats. The internal threat refers to the internal threat from the proposed method that limits the effectiveness of the proposed method. The threat to the effectiveness of the CSSLPM scheduling policy comes from the accuracy of the CNN-BiGRU-Attention load prediction model, which is limited by two main factors. The first factor is that the load model established in this paper does not match the load of real scenarios totally, i.e., there is a problem of container cloud load that cannot be described, and this paper adopts the idea of approximation instead of

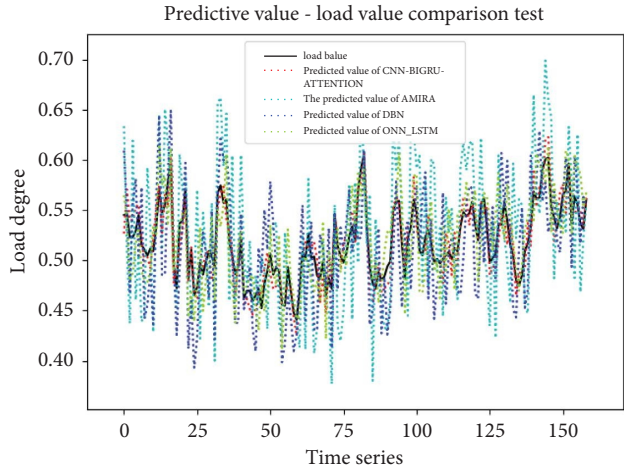

Figure 17: Comparison of load prediction results for each model.

Table 12: Performance comparison of each forecast model.

Model	MAE	MAPE	RMSE
ARIMA	0.7895	0.1673	1.2181
DBN	0.6534	0.1361	1.0618
CNN-LSTM	0.5957	0.11093	0.9839
CNN-BiGRU-Attention	**0.4463**	**0.0973**	**0.8572**

First, MAE, MAPE, and RMSE are the error assessment methods mentioned in the paper, and the smaller their values are, the better the prediction model is. Bold values highlight the evaluation methods on which the model outperforms other models.

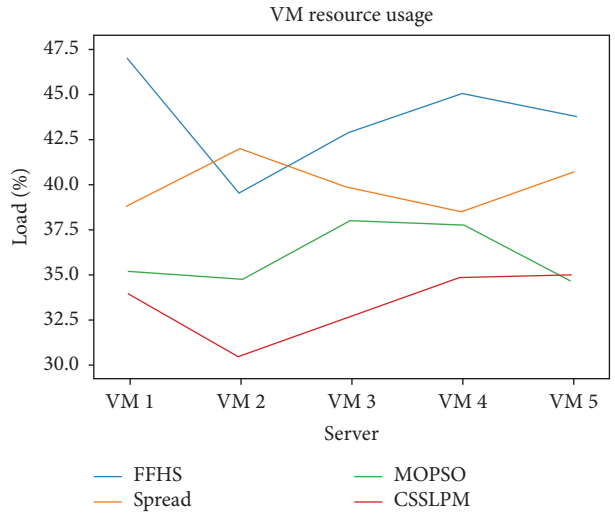

Figure 18: Comparison of resource usage for each model.

load modeling, i.e., the average value of CPU and memory utilization per unit time is used to approximate the load value of containers, which can effectively reduce the real-time load data processing delay but inevitably leads to the degradation of prediction accuracy. The second factor is the range of model hyperparameters. The second factor is that the range of model hyperparameters does not completely cover the whole parameter domain. Since it is a time-consuming task to compare the model training effect of each parameter combination, the hyperparameters of the model are selected from a limited range of parameter combinations, and the existence of globally optimal hyperparameters outside this range may improve the accuracy of the prediction model.

4.3.2. External Threats. The external threat refers to the difference between the experimental environment and the real scenario, thus limiting the effectiveness of the proposed method. In this paper, the proposed CSSLPM is experimentally validated under the open-source cloud computing simulation platform software CloudSim and the open-source benchmark

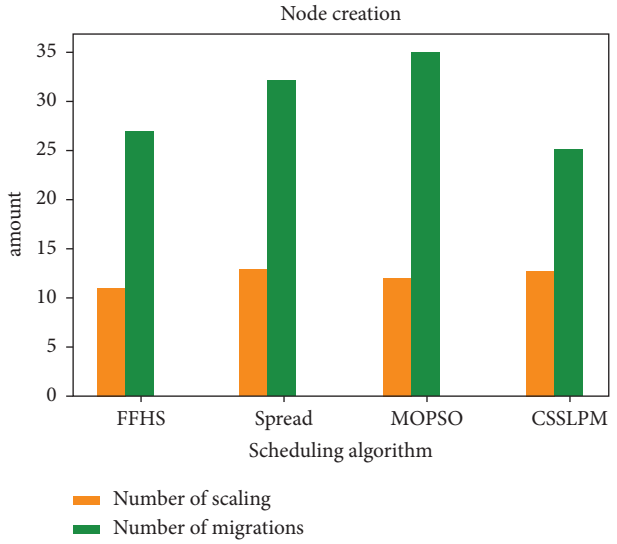

Figure 19: Comparison of number of schedules for each model.

system TrainTicket. Although our work has a more comprehensive experimental design, it lacks consideration of the real industrial environment such as the heterogeneity in the container cloud environment is not only composed of several servers and the user access pressure of the real ticketing system is not really simulated. We must admit that these differences between the experimental environment and the real scenario do threaten the effectiveness of the CSSLPM.

5. Conclusion

This paper proposes a load prediction method and container scheduling strategy for container cloud environments. The proposed method uses a CNN-BiGRU-Attention model to improve the accuracy of load prediction and achieve load balancing and resource optimization. However, there are some limitations, such as inaccurate load modeling and insufficient accuracy in flexible scheduling, which need to be addressed in future research. Additionally, real industrial environment validation is needed since the proposed method has only been experimentally validated in simulation platforms.

Data Availability

The data used to support the findings of this study have been deposited in the Alibaba Cluster Data repository (https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2018/trace_2018.md).

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (grant nos. U21B2015, 61972300, and 62202357) and Young Talent Fund of Association for Science and Technology in Shaanxi (grant no. 20220113).

References

[1] Canonical Ltd, "Linux containers," 2023, https://linuxcontainers.org/.

[2] H. Chen, "Cloud computing ecosystem report," pp. 1–244, 2022, https://comptiacdn.azureedge.net/webcontent/docs/default-source/research-reports/research-brief-comptia-cloud-ecosystem.pdf?sfvrsn=2495b5a7_2.

[3] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.

[4] D. Kakadia, *Apache Mesos Essentials*, Packt Publishing, Birmingham, UK, 2015.

[5] D. Vohra and D. Vohra, *Kubernetes on google cloud platform*, pp. 49–87, Kubernetes Management Design Patterns: With Docker, CoreOS Linux, and Other Platforms, Apress, Berkeley, CA, USA, 2017.

[6] K. Ye, Y. Kou, C. Lu, Y. Wang, and C.-Z. Xu, "Modeling application performance in docker containers using machine learning techniques," in *Proceedings of the 2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 1–6, IEEE, Singapore, December 2018.

[7] D. Zhang, B.-H. Yan, Z. Feng, C. Zhang, and Y.-X. Wang, "Container oriented job scheduling using linear programming model," in *Proceedings of the 2017 3rd International Conference on Information Management (ICIM)*, pp. 174–180, IEEE, Chengdu, China, April 2017.

[8] Y. Mao, J. Oak, P. Anthony, B. Daniel, H. Tao, and H. Peizhao, "Draps: dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster," in *Proceedings of the 2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pp. 1–8, IEEE, San Diego, CA, USA, December 2017.

[9] C. Kaewkasi and K. Chuenmuneewong, "Improvement of container scheduling for docker using ant colony optimization," in *Proceedings of the 2017 9th international conference on knowledge and smart technology (KST)*, pp. 254–259, IEEE, Chonburi, Thailand, March 2017.

[10] L. Tan and H. Tao, "An improved kubernetes scheduling algorithm based on load balancing," *Journal of Chengdu University of Information Technology*, vol. 34, no. 3, pp. 228–231, 2019.

[11] S. Nanda and T. J. Hacker, "Racc: resource-aware container consolidation using a deep learning approach," in *Proceedings of the First Workshop on Machine Learning for Computing Systems*, pp. 1–5, West Lafayette, IN, USA, May 2018.

[12] D. Xue, F. Long, and G. Chen, "Fault prediction algorithm based on particle filter and linear autoregressive models," *Computer technology and development*, vol. 21, no. 11, pp. 133–136, 2011.

[13] Q. Wei, F. Zhang, and Y. Zhao, "Load balancing algorithm based on load weights," *Computer Application Research*, vol. 29, no. 12, pp. 4711–4713, 2012.

[14] M. A. S. Monfared, R. Ghandali, and M. Esmaeili, "A new adaptive exponential smoothing method for non-stationary time series with level shifts," *Journal of industrial engineering international*, vol. 10, no. 4, pp. 209–216, 2014.

[15] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload prediction using arima model and its impact on cloud applications' qos," *IEEE transactions on cloud computing*, vol. 3, no. 4, pp. 449–458, 2015.

[16] Q. Chen and H. Fang, "A prediction method for mid-long term load forecasting using big data technology," *Journal of Wuhan University (Natural Science Edition)*, vol. 50, no. 2, pp. 239–244, 2017.

[17] Y. Wang, Y. Li, and Y. Liao, "Dailyload curve forecastingbyusingk-modes clustering algorithm under the framework of mapreduce," *Computer and Digital Engineering*, vol. 44, no. 2, pp. 230–232, 2016.

[18] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 155–162, 2012.

[19] F. Qiu, B. Zhang, and J. Guo, "A deep learning approach for vm workload prediction in the cloud," in *Proceedings of the 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/ Distributed Computing (SNPD)*, pp. 319–324, IEEE, Shanghai, China, May 2016.

[20] A. S. Ashraf, "Automatic cloud resource scaling algorithm based on long short-term memory recurrent neural network," 2017, https://arxiv.org/ftp/arxiv/papers/1701/1701.03295.pdf.

[21] J. Guo, J. Wu, J. Na, and B. Zhang, "A type-aware workload prediction strategy for non-stationary cloud service," in *Proceedings of the 2017 IEEE 10th Conference on Service-Oriented Computing and Applications (SOCA)*, pp. 98–103, IEEE, Kanazawa, Japan, November 2017.

[22] A. Peter, "Dinda. "The statistical properties of host load"," in *Languages, Compilers, and Run-Time Systems for Scalable Computers*, D. R. O'Hallaron, Ed., pp. 319–334, Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[23] T. Inagaki, Y. Ueda, and M. Ohara, "Container management as emerging workload for operating systems," in *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–10, Providence, RI, USA, September 2016.

[24] W. Shang, D. Liu, L. Zhu, and D. Feng, "An improved dynamic load-balancing model," in *Proceedings of the 2016 4th Intl Conf on Applied Computing and Information Technology/ 3rd Intl Conf on Computational Science/Intelligence and Applied Informatics/1st Intl Conf on Big Data, Cloud Computing, Data Science & Engineering (ACIT-CSII-BCD)*, pp. 337–341, Las Vegas, NV, USA, December 2016.

[25] Z.-J. Hu, "A qos-oriented resource availability evaluation model in computational grids," 2010, https://www.sciencedirect.com/science/article/abs/pii/S0164121215001715.

[26] L. Y. Zuo, Z. B. Cao, and S. B. Dong, "Virtual resource evaluation model based on entropy optimized and dynamic weighted in cloud computing," *Journal of Software*, vol. 24, no. 8, pp. 1937–1946, 2014.

[27] M. Tao, S. Dong, and L. Zhang, "A multi-strategy collaborative prediction model for the runtime of online tasks in computing cluster/grid," *Cluster Computing*, vol. 14, no. 2, pp. 199–210, 2011.

[28] K. Wang, C. Wu, Y. Yao et al., "Association between socio-economic factors and the risk of overweight and obesity among Chinese adults: a retrospective cross-sectional study from the China Health and Nutrition Survey," *Global health research and policy*, vol. 7, no. 1, pp. 41–49, 2022.

[29] L. Luo, "Research on container elastic scaling technology based on load prediction," M.Sc. thesis, Wuhan Textile University, Wuhan, China, 2021.

[30] Y. Guo, *Research and implementation of docker container scheduling strategy in microservice environment*, Ph.D. thesis, Beijing University of Posts and Telecommunications, Beijing, China, 2018.

[31] S. Wu, "Research on docker container scheduling optimization method," M.Sc. thesis, Zhengzhou University, Zhengzhou, China, 2019.

[32] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[33] Y. Sun, B. Jacobus van Wyk, and Z. Wang, "A new multi-swarm multi-objective particle swarm optimization based on pareto front set," in *Proceedings of the Advanced Intelligent Computing Theories and Applications. With Aspects of Artificial Intelligence: 7th International Conference, ICIC 2011*, vol. 7, pp. 203–210, Springer, Zhengzhou, China, August 2011.

[34] S.-H. Kim, G. Lee, I. Hong, Y.-J. Kim, and D. Kim, "New potential functions for multi robot path planning: swarm or spread," in *Proceedings of the 2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, vol. 2, pp. 557–561, IEEE, Singapore, April 2010.