*Advances in Artificial Intelligence, Big Data and Algorithms*
*G. Grigoras and P. Lorenz (Eds.)*


*doi:10.3233/FAIA230814*

# Automatic Resource Scaling Management System on Electric Power Supercomputing Clouds

Rundong GAN[a1], Xun LI[a], Wei WEI[a], Haibin SU[b,c], Zhu ZHAN[b,c]
[a] *Guizhou Power Grid Co., Ltd. Information Center, Guizhou Guiyang, China*
[b] *Fangxin Technology Co., Ltd., Hunan Changsha, China*
[c] *Hunan Electric Power Computing Power Integration Engineering Technology Research Center, Hunan Changsha, China*

**Abstract.** With the advent of the big data era, the explosive growth of data volume has put enormous computing and storage pressure on electric power companies. As a powerful technical means, supercomputing clouds are widely used in data processing, storage, and online services. However, critical services in supercomputing clouds are often deployed with overprovisioned resources to ensure the quality of service for users, resulting in significant energy consumption and additional costs. At the same time, insufficient resources for service provisioning can lead to performance degradation and service violations. To this end, this paper proposes an automatic resource scaling management system on electric power supercomputing clouds. Specifically, the proposed system is based on Transformer's long-sequence prediction model to predict the future load intensity of the service and calculate the number of instances required by the service in the future through the runtime service requirement estimation component, thus automatically scaling resources and minimizing resource costs. Experimental results show that our system achieves the best scaling behavior based on performance metrics and the lowest cost overhead compared to strong baselines.

**Keywords.** Resource Scaling, Electric Power, Supercomputing Clouds

## 1. Introduction

Supercomputing clouds provide great convenience and cost-effectiveness for users [1]. More and more electric power companies rely on cloud resources to build information support platforms, thus integrating resources internally and improving service quality. With the popularity of the Internet and the development of big data, the cloud-based business of electric power systems often faces massive data flow and high concurrency. To ensure quality of service for users, critical services in the clouds are usually deployed with overprovisioned resources, resulting in huge energy consumption and additional costs. Statistically, supercomputing cloud centers have become one of the world's largest sources of carbon emissions. Rising energy costs, regulatory requirements, and societal concerns about greenhouse gas emissions make reducing energy consumption critical for supercomputing clouds [2-3]. However, this is all for naught if supercomputing clouds fail to meet pre-defined service level agreements or quality of service (QoS) targets. This

[1] Rundong GAN, Guizhou Power Grid Co., Ltd. Information Center; e-mail: ganrundong2022@163.com

is because excessive processing latency or even communication blockages are unacceptable to users [4]. As a result, how to reduce energy consumption and meet user service level agreements has become a major challenge for today's cloud computing platforms.

Attracted by the emerging cloud computing paradigm, automatic resource scaling has become an important part of the supercomputing clouds. Currently, most cloud platforms provide a reactive threshold-based approach to help users automate the scaling of resources [5-8]. A typical example is to add an instance when the monitoring system detects that the CPU utilization of a service instance exceeds 70%. The "70%" is a threshold value that is manually specified by the user. In theory, the simple threshold-based approach does not involve an accurate resource estimation, only an empirical estimation, which is hard-coded in the operational part of the rule, such as adding or removing a certain number or percentage of instances. Obviously, specifying appropriate thresholds is not always straightforward for users, especially in functionally complex business scenarios and with diverse resource monitoring metrics [9-11]. At the same time, additional service instances may consume time measured in minutes from startup to service provisioning, making it difficult to apply to bursty network loads.

Ideally, cloud platforms can strike a balance between satisfying user requirements and resource costs. To this end, this study proposes an automatic resource scaling management system for power supercomputing clouds. The system uses the Transformer-based long series prediction model to predict the future load intensity of the service, which can adaptively adjust the model parameters according to different services and application scenarios, thus improving the adaptability and flexibility of the system. The number of instances required by each service in the future is calculated by the runtime service demand estimation component. In this way, the different services can rely on automatic scalers to obtain computing resources on demand.

The contributions of this paper can be summarized as follows:

- We incorporate a Transformer-based long series prediction model into an automated scaling management system that effectively captures long-range dependencies in time series.
- We design an intelligent resource auto-scaling management system. Services under this system can access resources automatically and on-demand without manual customization and preparation.
- Experimental results show that our proposed system achieves the best scaling behavior and the lowest cost overhead compared to robust baselines.

## 2. Model Design

In this section, we detail the process of building the management system for automatic resource scaling.

### *2.1. Overview*

We propose an intelligent management system for automatically resource scaling on electric power supercomputing clouds. The study is based on the real distributed architecture Kubernetes [12], which is an efficient container orchestration system for automating the management of deployment, scaling, and maintenance of containerized

services. In the paper, each instance of an application service is composed of a container, and each service can have multiple functionally consistent instances to meet the load requirements of users. With Kubernetes' API Server directive, we can dynamically control the number of instances of each service in the distributed architecture.

As shown in Figure 1, the resource scaling management system proposed in this paper mainly consists of the following components: (1) management center, (2) Prometheus performance monitoring module [13], (3) demand analysis, (4) capacity planning, (5) Traefik load balancer [14]. The management center is the hub of the system, responsible for coordinating the work of the various components and specifying service level agreements (SLA) in the form of Cartesian Modeling Language (DML). The Prometheus performance monitoring module is used to collect, process, and aggregate real-time quantitative data about the services, such as the number and type of requests. We focus on latency, traffic, request rate and saturation, which are important indicators of service performance and the basis for resource scaling. The demand analysis module analyzes the load of the service in future periods based on Transformer's long series forecasting model. The capacity planning module is designed to calculate the number of instances required for a service, thereby optimizing the allocation of resources and minimizing waste of resources. The Traefik load balancer is utilized to send user requests to each instance of the service in an even manner.

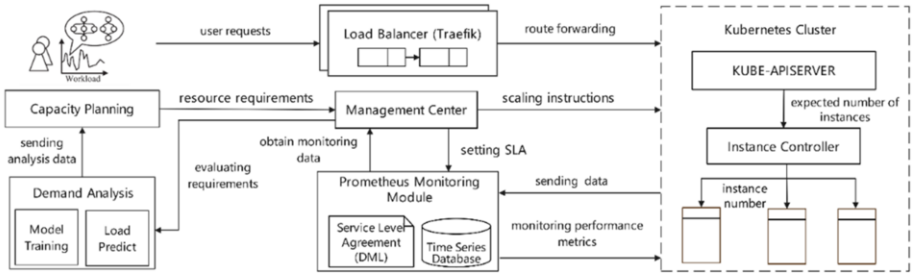



**Figure 1.** The overview of our proposed automatic resource scaling management system.

In the process of resource scaling, the management system has four main tasks: at first, the management center is responsible for periodically retrieving current service status information from the Prometheus timing database and sending the data to the demand analysis module. Then, after obtaining the time series data, the demand analysis module removes the abnormal data such as the long tail and analyzes the load of the service in the future period based on Transformer's long series prediction model. Next, with information on the future load of the service, the capacity planning module optimizes the resource allocation of the service through the SLA metrics and the resource estimator. Finally, the management center sends the command for resource scaling to the cluster management component KUBE-APISERVER of Kubernetes.

### *2.2. Demand Analysis*

In contrast to traditional forecasting methods, the proposed forecasting model converts the load information of historical time periods into vectors by means of a mapping function. Similar to BERT's input, we sum multiple types of vectors to obtain the initial feature input of the model at time $t$:

$$x_t = e_l + e_p + e_w + e_h \tag{1}$$

where $e_l, e_p, e_w, e_h \in \mathrm{R}^{1\times d_x}$ denote the embeddings of load, position, time and holiday respectively. By fusing multiple categories of encoding, it allows the prediction model to explore the periodicity and stochasticity of the application's workload over time.

Next, we define the input sequence of the model. Let the historical rolling window of the model be $L_x$. At time $t$, the model input sequence can be expressed as $X^t = \{x_1^t, x_2^t ..., x_{L_x}^t\}$. Accordingly, the purpose of the model is to predict the load sequence at future moments based on the inputs $Y^t = \{y_1^t, y_2^t, \ldots, y_{L_y}^t\}$, where $L_y$ denotes the predicted scrolling window size.

The prediction model is based on the Transformer architecture. The multi-head self-attention is the key to the prediction model, which can capture long-range dependencies on time series and effectively represent the importance and relationships in the sequence context, regardless of location. The formula of multi-head self-attention is calculated as:

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head}_h)W_m \tag{2}$$

$$\mathrm{head}_i = \mathrm{SelfAttention}(QW_i^Q, KW_i^K, VW_i^V) \tag{3}$$

$$\mathrm{SelfAttention}(X) = \mathrm{Softmax}(\frac{XW_q(XW_k)^T}{\sqrt{d_k}})XW_v \tag{4}$$

where $Q$, $K$, $V$ denote the query matrix, key matrix, and value matrix, respectively. $W_i$ refers to the learnable parameter matrix. Concat(·) represents the aggregation operation of the features. The processing of multi-headed self-attention can compute the feature vectors at each position in parallel, thus providing high characterization capability and operational efficiency.

The native Transformer architecture can only output one prediction per encoding. To solve the problem of too slow prediction for long sequence encoding, the model outputs multiple values at a time as predictions in a generative inference manner. Specifically, in the decoding phase, the model intercepts a short fixed-length sequence from the input sequence as a start token instead of a flag. Next, the model fills the predicted multiple time slots with scalars that are 0 and inputs them to the encoder with the start token. Finally, the encoder output acting on the position of 0 is considered to be the predicted value of the corresponding time period load.

### *2.3. Capacity Planning*

The capacity planning approach proposed in this paper obtains load forecasts through the demand analysis module, then maps resources to the required capacity and deploys an appropriate number of instances to ultimately achieve cost reduction.

For the same service, there is a simple linear relationship between the number of requests and the system load. Once the predicted value of the load that the service will bear at the future moment and the processing capacity of the service instances are available, the required number of instances can be obtained by derivation. We derive the number of instances needed for the next period:

$$k_{reqd} = \text{Ceil}\left[ k_{curr} \times \text{f}\left( \frac{p_{sys}}{k_{curr} p_{ref}}, \lambda \right) \right] \tag{5}$$

$$\text{f}(x, \lambda) = \begin{cases} x, & if \left| x - 1 \right| \geq \lambda \\ 1, & if \left| x - 1 \right| < \lambda \end{cases} \tag{6}$$

where Ceil(·) is the upward rounding function, $p_{sys}$ denotes the system load at the next period, $p_{ref}$ represents the maximum load that a service instance can handle and $k_{curr}$ denotes the number of instances at the current period. f(·) is the tolerance function, and $\lambda$ denotes the tolerance level. $P_{sys}$ / $k_{curr}$ represents the workload borne by a single instance under the system load at the next period. If $p_{sys} / (k_{cuur} p_{ref})$ is less than the tolerance $\lambda$, the scaling operation is abandoned. The capacity planning module optimizes the resource allocation scheme to meet the user's needs and minimize the waste of resources based on the user's behavior and the characteristics of the service.

## 3. Experiment

### *3.1. Testbed and Workload*

Our proposed automatic resource scaling management system (ARSMS) is deployed in a real distributed Kubernetes environment. The Kubernetes cluster consists of five real physical machines with 12-16 CPUs and 32-64G of memory. One physical machine is the master node, and the other four physical machines are the worker nodes. Multiple instances of each service consist of functionally identical containers. We design a matrix multiplication service as a test application that randomly generates two matrices of 100-200 dimensions and returns the result of the multiplication.

In our experiments, we validate our model using two network loads. The Wikipedia trace contains page requests for all languages during the period between September 19th 2007 and January 2nd 2008. In addition, we collect a web load flow from a news media application (NewsFlow) that shows significant periodicity. Here, we evaluate the performance and stability of the service by using two-day traces and simulating user interactions with the application using the load testing tool JMeter.

### *3.2. Experimental Setup*

For the sake of experimental fairness, we choose two typical resource scaling methods as benchmarks:

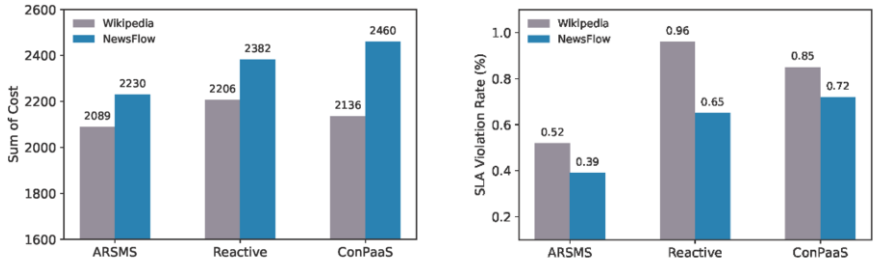
(1) **Reactive** [15]. This is the classic threshold-based elastic scaling approach for handling constantly changing workloads. This method allows users to set a number of scaling trigger indicators, such as CPU and memory usage.

(2) **ConPaas** [16]. Similar to the steps in our work, this method consists of a process of detection, prediction, decision, and execution. The method dynamically adjusts the number of resources served based on changes in load at fixed intervals of 10 minutes.

### *3.3. Experiment Results and Analysis*

We demonstrate the superiority of the flexible scaling management system proposed in this paper in terms of cost consumption and SLA violation rate. Cost consumption is used to measure the sum of the number of container resources occupied by a service over a period of time. It is expressed in container hours and represents the average number of containers occupied by a service in an hour. SLA violation rate is the percentage of vendors that fail to deliver services to customers in accordance with the service at the agreed service level, calculated as (number of failures / total number of service requests) * 100%.

As shown in Figure 2(a), the proposed ARSMS achieves the optimal resource consumption compared to Reactive and ConPaas under two different network loads. Reactive is a threshold-based elastic scaler. However, if the threshold is not set correctly, it will perform frequent scaling operations, wasting more resources. ConPaas relies mainly on machine learning methods to predict future load and thus resource scaling. However, these prediction methods are only suitable for smooth time series, and the fixed prediction interval of 10 minutes is hardly applicable to sudden changes in load. Our proposed method ARSMS can optimize service resources in a timely and on-demand manner through the deep learning-based demand analysis and resource planning modules.




(a) Performance of resource consumption (b) Performance of SLA violations

**Figure 2**. Performance of the automatic resource scaling management system On Wikipedia and NewsFlow.

As illustrated in Figure 2(b), the proposed ARSMS obtains the lowest SLA violation rate under two different network loads. The threshold-based elastic scaler Reactive is difficult to be applied to complex application scenarios. If the load of the system fluctuates a lot, the method may not be able to optimize the allocation of resources effectively. ConPaas is difficult to predict the future data at a long distance and is prone to large prediction errors. Therefore the model has a high SLA violation rate. ARSMS adopts the Transformer-based long series prediction model, which can have a better prediction effect for long-range loads and is more robust to abnormal data and noise.

## 4. Conclusion

In this paper, we propose an automatic resource scaling management system on electric power supercomputing clouds. The system can effectively predict the future resource demand of the service and optimize the resource allocation scheme according to the characteristics of the application, thus satisfying the users' needs and minimizing the waste of resources. Experiments show that our proposed system achieves optimal results. In future research, we will focus on the collaboration and management of resources in hybrid cloud environments to achieve more granular resource allocation.

## References

[1] G. Sriram, "Edge computing vs. cloud computing: an overview of big data challenges and opportunities for large enterprises," International Research Journal of Modernization in Engineering Technology and Science, vol. 4, no. 1, pp. 1331–1337, 2022.
[2] S. Bharany, S. Sharma, O. I. Khalaf, G. M. Abdulsahib, A. S. Al Humaimeedy, T. H. Aldhyani, M. Maashi, and H. Alkahtani, "A systematic survey on energy-efficient techniques in sustainable cloud computing," Sustainability, vol. 14, no. 10, p. 6256, 2022.
[3] M. Jangjou and M. K. Sohrabi, "A comprehensive survey on security challenges in different network layers in cloud computing," Archives of Computational Methods in Engineering, vol. 29, no. 6, pp. 3587–3608, 2022.
[4] G. L. Stavrinides and H. D. Karatza, "An energy-efficient, qos-aware and cost-effective scheduling approach for real-time workflow applications in cloud computing systems utilizing dvfs and approximate computations," Future Generation Computer Systems, vol. 96, pp. 216–226, 2019.
[5] P. Singh, P. Gupta, K. Jyoti, and A. Nayyar, "Research on auto-scaling of web applications in cloud: survey, trends and future directions," Scalable Computing: Practice and Experience, vol. 20, no. 2, pp. 399–432, 2019.
[6] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," ACM Computing Surveys (CSUR), vol. 51, no. 4, pp. 1–33, 2018.
[7] M. Song, C. Zhang, and E. Haihong, "An auto scaling system for api gateway based on kubernetes," in 2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS).
[8] S. Taherizadeh and M. Grobelnik, "Key influencing factors of the kubernetes auto-scaler for computing-intensive microservice-native cloudbased applications," Advances in Engineering Software, vol. 140, p.
[9] T. Hu and Y. Wang, "A kubernetes autoscaler based on pod replicas prediction," in 2021 Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS). IEEE, 2021, pp. 238–241.
[10] F. Rossi, "Auto-scaling policies to adapt the application deployment in kubernetes." in ZEUS, 2020, pp. 30–38.
[11] D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of kubernetes pods," in NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium. IEEE, 2020, pp. 1–5.
[12] Burns B, Beda J, Hightower K, et al. Kubernetes: up and running[M]. " O'Reilly Media, Inc.", 2022.
[13] Padgham L, Winikoff M. Prometheus: A methodology for developing intelligent agents[C]//Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1. 2002: 37-38.
[14] Sharma R, Mathur A, Sharma R, et al. Traefik as Kubernetes Ingress[J]. Traefik API Gateway for Microservices: With Java and Python Microservices Deployed in Kubernetes, 2021: 191-245.
[15] T. C. Chieu, A. Mohindra, et al., "Dynamic scaling of web applications in a virtualized cloud computing environment," in 2009 IEEE International Conference on e-Business Engineering. IEEE, 2009, pp. 281–286.
[16] H. Fernandez, G. Pierre, et al., "Autoscaling web applications in heterogeneous cloud infrastructures," in 2014 IEEE international conference on cloud engineering. pp. 195–204.