Adaptive Resource Allocation to Enhance the Kubernetes Performance for Large-Scale Clusters

Jiayin Luo
fro_l@zju.edu.cn
Zhejiang University
Hangzhou, Zhejiang, China

Xinkui Zhao*
zhaoxinkui@zju.edu.cn
Zhejiang University
Hangzhou, Zhejiang, China

Yuxin Ma
yuxinma@zju.edu.cn
Zhejiang University
Hangzhou, Zhejiang, China

Shengye Pang
pangsy@zju.edu.cn
Zhejiang University
Hangzhou, Zhejiang, China

Shuiguang Deng
dengsg@zju.edu.cn
Zhejiang University
Hangzhou, Zhejiang, China

Jianwei Yin
zjuyjw@zju.edu.cn
Zhejiang University
Hangzhou, Zhejiang, China

Abstract

The advent of cloud computing has led to a dramatic increase in the deployment of hyper-scale, diverse workloads in containerized form on cloud infrastructures. This expansion necessitates the management of numerous large-scale clusters. However, Kubernetes (k8s), the industry standard for container orchestration, faces challenges with low scheduling throughput and high request latency in such environments. We identify resource contention among components co-located on the same master node as a primary bottleneck. To address this, we introduce a lightweight framework designed to enhance the performance and scalability of Kubernetes clusters. Our approach adaptively allocates resources among co-located components, improving their overall performance. Implemented as a non-intrusive solution, our evaluations across various cluster scales show significant improvements, with a *7.3* × increase in cluster scheduling throughput and a *37.3%* reduction in request latency, surpassing the performance of vanilla Kubernetes and baseline resource allocation strategies.

1 Introduction

With the rapid development of cloud computing, the growth of diverse workloads, including microservices [2, 13], batch processing jobs [4, 10], and Function as a Service (FaaS) [14, 17], has resulted in a substantial increase in the scale of nodes and containers on cloud platforms. This expansion exerts considerable pressure on the underlying infrastructure.

Kubernetes (k8s), a key component of cloud infrastructure [8], is recognized for its role in container orchestration but shows limitations in scaling within large-scale clusters. For instance, operational constraints limit clusters to 5,000 nodes and cap the pod count per node at 110 [9]. In large-scale clusters, Kubernetes performance degrades severely when handling a surge of requests. Initially, request latency increases significantly. The API server experiences delayed response times for routine requests [3, 12], and the scheduler exhibits a decrease in throughput during pod scheduling operations[1] [3, 18]. Furthermore, cluster availability is at risk. The influx of requests may overwhelm the cluster's capacity, leading to repeated crashes and restarts due to OOM (Out-Of-Memory) issues, ultimately resulting in cluster inaccessibility.

To overcome Kubernetes' scalability limitations and facilitate the unified deployment and management of extensive workloads, there are numerous works in both industry and academia focusing on enhancing a cluster's capacity by optimizing control plane components [1, 5, 18], mechanisms [3, 19], fault tolerance [6, 15], and resource utilization [16, 20]. However, most of these efforts are designed for specific scenarios and require modifications to the Kubernetes code, leading to complex deployment.

This paper presents a lightweight framework to improve the scalability of Kubernetes clusters. Driven by the prevalent resource contention among components co-located on the master node, our method dynamically adjusts the resource allocation and traffic control parameters of control plane components to improve the cluster's overall efficiency.

We have seamlessly integrated our approach as a non-intrusive plugin, requiring no changes to Kubernetes code. We have conducted comprehensive evaluation across a range of cluster scales and request intensities. The results highlight significant performance improvements. Specifically, it outperforms vanilla Kubernetes and alternative resource allocation strategies by increasing scheduling throughput by up to *7.3* × and reducing request latency by up to *37.3%*.

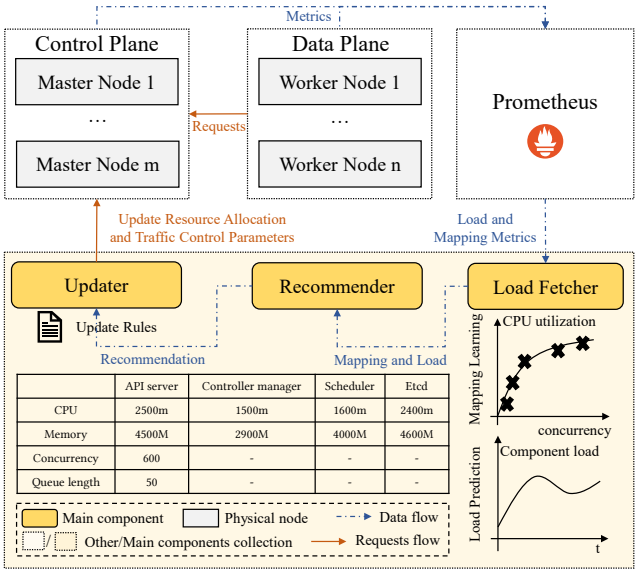
The contributions of this paper are summarized as follows:

- We develop a queuing model for request processing of Kubernetes and devise an algorithm for the dynamic

*Corresponding author

The 2nd Workshop on Hot Topics in System Infrastructure (HotInfra'24), co-located with SOSP'24, November 3, 2024, Austin, TX, USA

[1] In our test, as the cluster size grows from 1,000 to 5,000 nodes, the average latency has increased by 3.03×, and the scheduling throughput has decreased by 38.5%.



	API server	Controller manager	Scheduler	Etcd
CPU	2500m	1500m	1600m	2400m
Memory	4500M	2900M	4000M	4600M
Concurrency	600	-	-	-
Queue length	50	-	-	-

Figure 1. Architecture of the system highlighting the main components in yellow. The data flow is centered on optimizing resource allocation and traffic control parameters, while the request flow involves recommendation implementation.

optimization of resource allocation and traffic control parameters to enhance overall performance.

- We implement our approach in a non-intrusive manner, with experiments validating its superiority against vanilla Kubernetes and baseline resource allocation methods. The code is open-sourced on GitHub[2].

2 Motivation and Methodology

2.1 Resource Contention among Co-located Components

On Kubernetes master nodes, the concurrent operation of multiple control plane components can lead to resource contention, especially under heavy load. As the cluster scales, resource contention becomes more severe. CPU contention can cause performance degradation and latency. Memory contention can result in OOM errors and component crashes, affecting the cluster's availability. The default Kubernetes configuration does not consider master instance placement and resource allocation, exacerbating contention. In a high-availability setup, the arbitrary placement of master components can concentrate them on a single node, increasing resource contention and performance decline.

2.2 System Overview

Motivated by the resource contention problem, we aim to design a system capable of adaptive resource allocation and traffic control parameter adjustment. Fig. 1 illustrates the system's architecture. During operation, the **load fetcher** retrieves concurrency and resource utilization metrics from Prometheus to determine the load of each component and adjust its CPU-concurrency mapping. The **recommender** creates periodic recommendations for CPU allocation and traffic control parameters based on current load and mapping data, which are then passed to the updater. The **updater** translates valid recommendations into executable requests sent to the master node, and implements master instance placement (place master instances of the scheduler and controller manager on separate nodes to mitigate resource contention), triggering the eventual updates.

[2]https://github.com/FROOOOOOO/ark-hotinfra

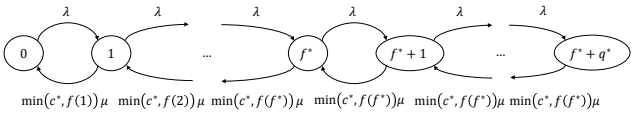


Figure 2. Rate transition diagram for the birth-death process.

2.3 Recommendation Algorithm

For each component, the optimal CPU allocation c^* is calculated by the recommender. And for the API server whose mapping metrics and traffic control parameters are available, the recommendation also includes the maximum concurrency f^* and queue length q^*.

We model the request processing by control plane components as queuing systems. Instead of using basic queuing models such as M/M/1 and M/M/c, we construct a model characterized by a birth-death process as shown in Fig. 2, because we find that the CPU-concurrency mappings $f(x)$ are nonlinear and dynamic, deviating significantly from those of basic queuing models. Control plane components are categorized into three groups based on their optimization objectives (*Group A: the API server and etcd*, minimize latency W; *Group B: the scheduler*, maximize throughput S; *Group C: the controller manager*, maintain stability, i.e., the maximum service rate μ exceeds the arrival rate λ.).

The recommender follows the workflow depicted in Fig. 3. In a global steady state when all busy components on the master node are steady, the objective is to minimize the average latency $\bar{W}$ while maximizing the scheduling throughput, and the optimization problem is formulated as follows:

$$\begin{aligned} \min \quad & w_A \bar{W} + w_B \frac{1}{S+1}, \\ \text{s.t.} \quad & \min\left(c_i^*, f(f_i^*)\right) > c_i', C. \end{aligned} \tag{1}$$

Here, w_A and w_B are the weights assigned to balance the optimization objectives for different components. In a global unsteady state where the total CPU load of synchronous components exceeds the master node's capacity, the objective is to minimize the remaining CPU load l while balancing the

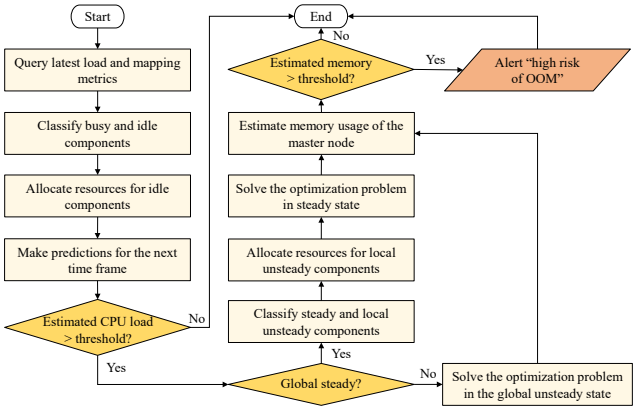


Figure 3. Flow chart of the recommendation algorithm.

time t to process it:

$$\begin{aligned} \min \quad & \sum_i l_i, \\ \text{s.t.} \quad & t = l_i / c_i^*, C. \end{aligned} \tag{2}$$

For (1) and (2), C denotes constraints, including the total allocated CPU within the node's capacity, and the cluster's availability not violating the service level objective (SLO).

3 Implementation and Evaluation

The system is implemented in 4,500 lines of Python code. Main components operate as standalone processes, interfacing with Kubernetes clusters via the Kubernetes API. Such flexible deployment requires only accessible master node IPs and a prepared kubeconfig file.

We employ a linear interpolation method for mapping learning and an ARMA model [11] for load metric prediction. We develop a customized evolutionary algorithm to efficiently solve (1) and (2). Specifically, the CPU granularity is proportional to the remaining CPU[3], controlling the size of solution space and the solution time within the threshold. The updater achieves resource allocation through vertical pod scaling without pod restart, traffic control parameters tuning through APF [7] mechanism, master instance placement through pod redeployment under low pressure.

We evaluate the performance of our approach through evaluations in terms of request latency and scheduling performance using KWOK to simulate large-scale nodes.

As depicted in Fig. 4 and 5, our approach delivers substantial performance improvements, reducing average latency by 9.58% to 37.27% and P99 read latency by 11.85% to 58.48%. Its scheduling throughput is 1.78 to 7.3 times higher when managing large workloads, and startup latency for urgent pods is reduced by up to 80.51%. In summary, the recommendation algorithm and master instance placement strategy effectively

[3]remain CPU = total CPU - total CPU load / time.

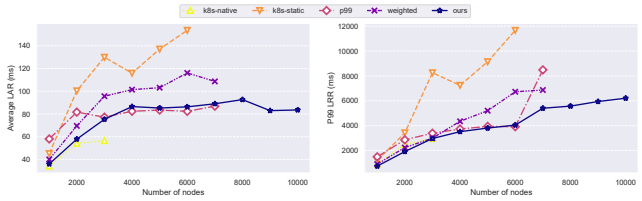


Figure 4. Request latency metrics (LAR: latency of all requests; LRR: latency of read requests).

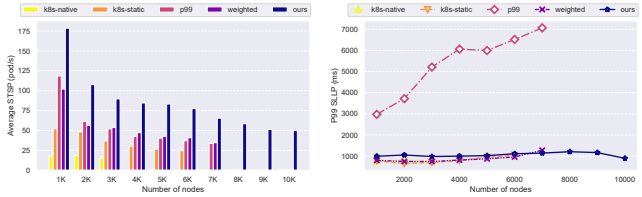


Figure 5. Scheduling performance (STLW: scheduling throughput of large workloads; SLSW: startup latency of small workloads).

Table 1. Resource and time cost.

Resource cost		Time cost	
Average/P99 CPU usage (core)	Average/P99 memory usage (MB)	Average/P99 update time (s)	Average/P99 recommendation time (s)
0.04/0.85	179.53/180.96	0.24/0.41	3.75/8.28

mitigate resource contention, accelerating both request processing and pod scheduling. Our approach demonstrates superior performance compared to all baselines[4].

To assess the overhead associated with our approach, we have monitored its resource and time expenditures, as detailed in Table 1. The CPU and memory consumption are both within acceptable limits. The average time required for updates is less than half a second. Regarding the recommendation time, we set a solution time threshold of 10 seconds. The actual time expenditure falls short of this threshold, thereby highlighting the efficiency of our algorithm.

4 Conclusion and Future Works

This paper presents a lightweight framework developed to enhance the Kubernetes performance in large-scale clusters. To address resource contention among components, we introduce a recommendation algorithm to optimize resource allocation, as well as a master instance placement mechanism to mitigate resource contention. Evaluations demonstrate the efficiency of our approach. The framework significantly optimizes both request latency and scheduling throughput, showcasing its ability to bolster Kubernetes cluster performance. In future endeavors, we aim to further optimize the

[4]`k8s-native`: native k8s; `k8s-static`: k8s with static arguments optimization; `p99`: take the 99th percentage usage as recommendation; `weighted`: allocate resources weighted by load.

master instance placement strategy, ensuring a more seamless and intelligent distribution of master instances.

Acknowledgments

We thank reviewers for their valuable comments. This work was supported in part by the Key Research and Development Program of China under Grant 2022YFF0902702, and in part by the Major Program of National Natural Science Foundation of Zhejiang (LD24F020014), and in part by the Zhejiang Pioneer (Jianbing) Project (2024-C01032), and in part by the Key R&D Program of Ningbo (2023Z235), and in part by the Ningbo Yongjiang Talent Programme (2023A-198-G).

References

[1] Xingyu Chen. 2019. Performance optimization of etcd in web scale data scenario. https://www.cncf.io/blog/2019/05/09/performance-optimization-of-etcd-in-web-scale-data-scenario

[2] Ka-Ho Chow, Umesh Deshpande, Veera Deenadayalan, Sangeetha Seshadri, and Ling Liu. 2024. Atlas: Hybrid Cloud Migration Advisor for Interactive Microservices. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 870–887.

[3] Alibaba Cloud Native Community. 2019. How Does Alibaba Ensure the Performance of System Components in a 10,000-node Kubernetes Cluster? https://www.alibabacloud.com/blog/how-does-alibaba-ensure-the-performance-of-system-components-in-a-10000-node-kubernetes-cluster_595469

[4] Jiangfei Duan, Ziang Song, Xupeng Miao, Xiaoli Xi, Dahua Lin, Harry Xu, Minjia Zhang, and Zhihao Jia. 2024. Parcae: Proactive, Liveput-Optimized DNN Training on Preemptible Instances. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1121–1139.

[5] Yihui Feng, Zhi Liu, Yunjian Zhao, Tatiana Jin, Yidi Wu, Yang Zhang, James Cheng, Chao Li, and Tao Guan. 2021. Scaling Large Production Clusters with Partitioned Synchronization. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 81–97.

[6] Jiawei Tyler Gu, Xudong Sun, Wentao Zhang, Yuxuan Jiang, Chen Wang, Mandana Vaziri, Owolabi Legunsen, and Tianyin Xu. 2023. Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 96–112.

[7] Kubernetes. 2024. API Priority and Fairness. https://kubernetes.io/docs/concepts/cluster-administration/flow-control/

[8] Kubernetes. 2024. Kubernetes – Production-Grade Container Orchestration. https://kubernetes.io/

[9] Kubernetes. 2024. Kubernetes Scalability thresholds. https://github.com/kubernetes/community/blob/master/sig-scalability/configs-and-limits/thresholds.md

[10] Baolin Li, Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, Karen Gettings, and Devesh Tiwari. 2021. RIBBON: cost-effective and qos-aware deep learning model inference using a diverse pool of cloud computing instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 24, 13 pages.

[11] Spyros Makridakis and Michele Hibon. 1997. ARMA models and the Box–Jenkins methodology. *Journal of forecasting* 16, 3 (1997), 147–163.

[12] OpenAI. 2021. Scaling Kubernetes to 7,500 nodes. https://openai.com/research/scaling-kubernetes-to-7500-nodes

[13] Vighnesh Sachidananda and Anirudh Sivaraman. 2024. Erlang: Application-Aware Autoscaling for Cloud Microservices. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 888–923.

[14] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, Andrii Golovei, Pradeep Venkat, Andrew Mcfague, Dimitrios Skarlatos, Vipul Patel, Ravinder Thind, Ernesto Gonzalez, Yun Jin, and Chunqiang Tang. 2023. XFaaS: Hyperscale and Low Cost Serverless Functions at Meta. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 231–246.

[15] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. 2022. Automatic Reliability Testing For Cluster Management Controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 143–159.

[16] Ziliang Wang, Shiyi Zhu, Jianguo Li, Wei Jiang, K. K. Ramakrishnan, Yangfei Zheng, Meng Yan, Xiaohong Zhang, and Alex X. Liu. 2022. DeepScaling: microservices autoscaling for stable CPU utilization in large scale cloud systems. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 16–30.

[17] Zhaojie Wen, Yishuo Wang, and Fangming Liu. 2022. StepConf: SLO-Aware Dynamic Resource Configuration for Serverless Function Workflows. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*. 1868–1877.

[18] Wu Xiang, Yakun Li, Yuquan Ren, Fan Jiang, Chaohui Xin, Varun Gupta, Chao Xiang, Xinyi Song, Meng Liu, Bing Li, Kaiyang Shao, Chen Xu, Wei Shao, Yuqi Fu, Wilson Wang, Cong Xu, Wei Xu, Caixue Lin, Rui Shi, and Yuming Liang. 2023. Gödel: Unified Large-Scale Resource Management and Scheduling at ByteDance. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 308–323.

[19] Jie Zhang, Chen Jin, YuQi Huang, Li Yi, Yu Ding, and Fei Guo. 2022. KOLE: breaking the scalability barrier for managing far edge nodes in cloud. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 196–209.

[20] Zhiqiang Zhou, Chaoli Zhang, Lingna Ma, Jing Gu, Huajie Qian, Qingsong Wen, Liang Sun, Peng Li, and Zhimin Tang. 2023. AHPA: Adaptive Horizontal Pod Autoscaling Systems on Alibaba Cloud Container Service for Kubernetes. *Proceedings of the AAAI Conference on Artificial Intelligence* 37, 13 (Sep. 2023), 15621–15629.