OPTIMIZATION OF KUBERNETES: RESOURCE ALLOCATION AND DYNAMIC SCALING

Džemil Sejdija,
[0009-0009-2042-0559]
Aldina Avdić*
[0000-0003-4312-3839]

State University of Novi Pazar,
Novi Pazar, Serbia

Abstract:
Kubernetes, a leading container orchestration platform, has become essential for managing modern cloud-native applications due to its scalability, automation, and resource optimization capabilities. This research focuses on Kubernetes' architecture, resource allocation strategies, and autoscaling mechanisms, highlighting key features such as the Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA). Through an analysis of experimental data and related works, the research underscores the importance of advanced scheduling algorithms, efficient monitoring tools like Prometheus and Grafana, and proactive resource management in improving overall operational efficiency. The findings demonstrate that combining Kubernetes-native features with customized enhancements can significantly reduce latency, resource contention, and operational costs, making Kubernetes a powerful tool for distributed application management.

Keywords:
Kubernetes, Resource allocation strategies, Autoscaling mechanisms, Horizontal Pod Autoscaler (HPA), Vertical Pod Autoscaler (VPA).

INTRODUCTION

Kubernetes (commonly abbreviated as K8s) is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications. Developed originally by Google and later adopted by the Cloud Native Computing Foundation (CNCF), Kubernetes has become a cornerstone of cloud-native architectures. Its robust ecosystem enables developers to focus on building and delivering applications, while Kubernetes handles complex operational tasks such as load balancing, resource allocation, and service discovery.

Kubernetes achieves this through its distributed architecture, where multiple nodes operate as a unified cluster, ensuring high availability and fault tolerance. The control plane manages the cluster's state by monitoring workloads and scheduling tasks across worker nodes, which run the containerized applications. Pods, the smallest deployable units in Kubernetes, encapsulate one or more containers and share network and storage resources within the same namespace. This architecture supports horizontal scaling, where additional pods can be deployed dynamically based on demand, and vertical scaling, where resource allocations for existing pods can be adjusted.

Correspondence:
Aldina Avdić

e-mail:
apljaskovic@np.ac.rs

DOI: 10.15308/Sinteza-2025-550-554

With the rapid growth of cloud-native computing, Kubernetes has become indispensable for organizations seeking to deploy resilient, scalable applications across hybrid, multi-cloud, or edge environments. However, as the scale and complexity of Kubernetes clusters grow, challenges such as resource underutilization, high costs, and performance bottlenecks necessitate adopting advanced optimization techniques. This research delves into Kubernetes' core components and resource management capabilities, focusing on strategies for optimizing autoscaling, dynamic resource allocation, and cost efficiency to enhance system performance and reliability in cloud-based environments. The methodology used in this work is experimental. The paper consists of an overview of similar works, an overview of Kubernetes architecture, an overview of Kubernetes resources and optimization methods, actual experiments for different techniques, and a discussion of results.

2. LITERATURE OVERVIEW

Experimental data from industry examples demonstrate the benefits of optimized Kubernetes configurations.

In the paper [1] key topics are Kubernetes autoscaling mechanisms (HPA, VPA and CA) and performance evaluation of HPA using Prometheus. Experimental insights provided by the paper and practical lessons show how to enhance the efficiency of resource management in the Kubernetes environment.

In the paper [2], key topics are performance bottlenecks (inefficient autoscaling) and experimental results that show performance improvements. The focus on reducing latency and enhancing scheduling strategies achieves efficient dynamic resource allocation.

In the paper [3], the author uses techniques for optimizing resource allocation but also incorporates predictive analytics to anticipate workload demands. The relevance of this work is the combination of Kubernetes-native features with third-party tools. In combination with other works, they give a lot of information regarding: Resource Optimization, Autoscaling Insights, Performance Improvement and Tool integration (Prometheus).

3. ARCHITECTURE OF KUBERNETES

The architecture of Kubernetes (Figure 1) follows a master-worker distributed model, designed to efficiently manage containerized workloads across multiple nodes. This architecture ensures scalability, reliability, and fault tolerance. The primary components are divided into control plane components (master node) and data plane components (worker nodes), with each serving a distinct purpose to maintain the desired state of the cluster [4] [5].

3.1. THE CONTROL PLANE (MASTER NODE)

The control plane is responsible for managing the overall state of the cluster, ensuring that the desired configuration is maintained. The key components are:

- API Server: The communication hub of Kubernetes, handling requests and updates from users and internal components.
- Controller Manager: Ensures the cluster stays in the desired state by managing various controllers (e.g., replication and node health).
- Scheduler: Assigns workloads (pods) to suitable worker nodes based on available resources and constraints.
- etcd: A distributed storage system that holds cluster configuration and state data, ensuring consistency and fault tolerance.

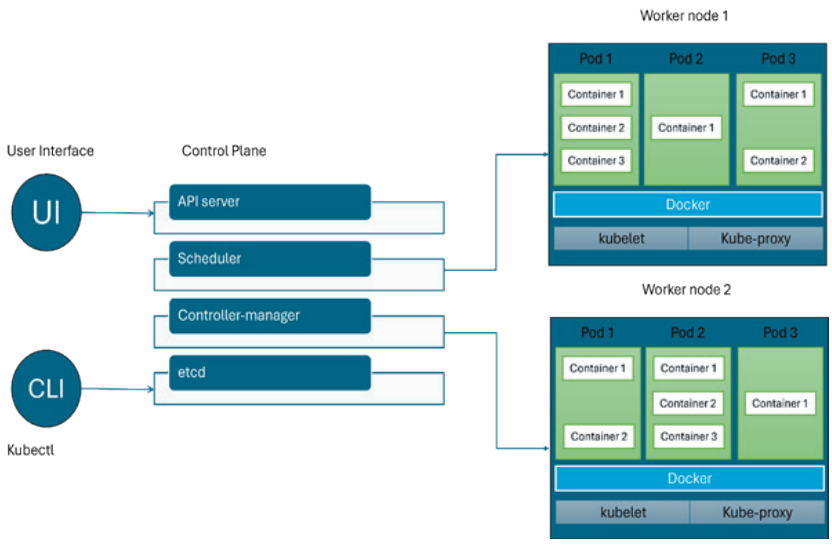

Figure 1. An Architecture of Kubernetes

3.2. THE DATA PLANE (WORKER NODES)

Worker nodes run the applications and provide the necessary computing resources. The key components are:

- kubelet: An agent on each node that ensures the assigned containers are running and healthy.
- kube-proxy: Manages network communication within the cluster and to external clients.
- Container Runtime: Responsible for running containerized applications (e.g., Docker, Containered).

3.3. CORE KUBERNETES OBJECTS

Kubernetes uses objects like Pods, ReplicaSets, Deployments, Services, and Ingress to manage containerized workloads, ensuring scalability, availability, and network accessibility within and outside the cluster. Additionally, features like Namespaces enable logical isolation for multi-tenancy, Horizontal Pod Autoscaler (HPA) scales pods based on resource usage, and Vertical Pod Autoscaler (VPA) optimizes resource allocation dynamically.

Kubernetes facilitates communication within the cluster through service discovery and internal DNS. When a pod is created, it is assigned a unique IP address within the cluster. Services enable other pods or external clients to access these pods via stable DNS names, avoiding direct dependency on pod IPs that may change over time [6] [7].

4. KUBERNETES RESOURCES AND OPTIMIZATION METHODS

Efficient resource management in Kubernetes is crucial for achieving high performance, cost efficiency, and system reliability. Kubernetes provides various mechanisms to allocate and manage CPU, memory, and storage resources across containerized applications. It offers both static and dynamic resource allocation, allowing workloads to scale based on demand.

When it comes to CPU management, Kubernetes ensures that containers receive a guaranteed minimum amount of CPU, while also setting an upper limit on how much they can consume. For example, if a container requests a fraction of a CPU core, it is guaranteed that amount, but it cannot exceed a specified limit.

Memory management works in a similar way, where a container is allocated a minimum amount of memory to ensure stable performance. However, if it exceeds the defined limit, it may be terminated to prevent excessive resource consumption.

In terms of storage, Kubernetes supports different types of storage solutions. Persistent storage can be either pre-provisioned or dynamically created based on application needs, ensuring data remains available even if a pod is restarted. Applications can request specific storage capacity using claims, while ephemeral storage provides temporary space that exists only for the duration of a pod's lifecycle.

By leveraging these resource management features, Kubernetes ensures that applications run efficiently, using resources optimally while maintaining system stability [8] [9].

Kubernetes optimizes resource use through autoscaling, quotas, and monitoring. Autoscaling adjusts resources based on demand—scaling pods horizontally to handle increased load, vertically adjusting resource limits (sometimes requiring restarts), and scaling clusters by adding or removing nodes to control costs.

Resource management is enforced with quotas and limits. Quotas cap total CPU, memory, and storage in a namespace, while limit ranges set default resource allocations to prevent over or under-provisioning.

Workload placement is optimized using affinity rules to assign pods to specific nodes and taints/tolerations to isolate resource-heavy workloads.

Monitoring tools like Prometheus, Grafana, and kube-state-metrics provide real-time insights into resource usage, helping ensure efficient performance and cost management [10] [11] [12].

5. EXPERIMENTS

5.1. EXPERIMENT 1: HPA TESTING FOR CPU UTILIZATION

The objective is to observe HPA behavior in response to increasing CPU utilization and pod scaling. The cluster should have three worker nodes (4 CPUs, 8 GB RAM each). Prometheus should be used for monitoring. The methodology is as follows: deploy nginx (CPU 250m request, 500m limit, memory 256Mi request, 512Mi limit). Configure HPA to scale based on CPU and run a load test with 2000req/sec using "hey". The results are in Table 1.

5.2. EXPERIMENT 2: VPA RESOURCE ADJUSTMENT

The objective is to analyse how VPA adjusts CPU and memory allocations for pods based on usage patterns. The cluster should have three worker nodes (4 CPUs and 16 GB RAM). The methodology is: to deploy a Linux image and make it do a batch processing task every 5 min (it can only generate and discard 128-256 MB of data to simulate CPU and memory load). Set initial CPU request to 100m and memory to 128Mi. Enable VPA and monitor how VPA adjusts resources. The results are in Table 2.

5.3. EXPERIMENT 3: RESOURCE QUOTAS AND COST CONTROL

The objective is to enforce resource quotas and monitor cost savings by restricting excessive resource usage across namespaces. The cluster should have four nodes (4 CPUs and 8 GB RAM) and two namespaces (team-a and team-b). Apply resource quotas to both namespaces. Deploy applications that attempt to request higher resources. Motor allocation is done using Grafana. The results are in Table 3.

6. DISCUSSION

Experiment 1 showcased how the Horizontal Pod Autoscaler (HPA) dynamically adjusted the pod count based on CPU usage. When usage exceeded 50%, the HPA increased the number of pods to handle the load, preventing performance issues. As the load decreased, the HPA scaled down the pods, optimizing resource usage and reducing costs. This demonstrated the HPA's effectiveness in maintaining system stability and ensuring responsiveness during peaks while conserving resources during idle times. Overall, the HPA proved to be a valuable tool for balancing performance and cost efficiency.

Experiment 2 demonstrated how the Vertical Pod Autoscaler (VPA) adjusted CPU and memory based on workload demands. As resource usage increased, VPA allocated more resources to ensure smooth performance. Conversely, when the demand dropped, it scaled down resource allocation to avoid over-provisioning. By dynamically adjusting resources, VPA optimized allocation, preventing inefficiencies, and ensuring the system operated cost-effectively

Experiment 3 demonstrated how resource quotas effectively managed resource usage by enforcing limits. When team-a exceeded their allocated quota, their excessive requests were throttled, ensuring that other teams had fair access to resources.

Table 1. HPA Results

Time	CPU% Utilization	Number of pods
0	20	2
5	55	4
10	70	6
15	80	8
20	48	4

Table 2. VPA Results

Time	CPU Request(m)	Memory Request
0	100	128
5	200	256
10	400	512
15	250	384
20	150	256

Table 3. Resource quota results

Namespace	Requested CPU	Allocated CPU	Status
Team-a	3	2	Limited
Team-b	1.5	1.5	Allowed

By implementing quotas, resource allocation remained balanced and prevented any team from monopolizing resources, promoting fairness across the system.

7. CONCLUSION

Effective resource management in Kubernetes is further supported by proactive monitoring through tools such as Prometheus and Grafana. These monitoring solutions enable real-time visibility into resource consumption, providing valuable insights that allow for timely adjustments and informed decision-making. By continuously tracking key performance metrics, organizations can identify potential bottlenecks before they impact operations, ensuring that resources are utilized efficiently, and applications remain stable under varying workloads. The integration of monitoring with autoscaling capabilities contributes to a more resilient and responsive infrastructure, ultimately improving the reliability and efficiency of cloud-native applications. The combination of HPA and VPA offers a holistic approach to scalability and resource management within Kubernetes environments. While HPA provides the ability to scale horizontally by increasing or decreasing the number of running pods based on load, VPA fine-tunes resource allocations within individual pods to ensure efficient utilization. This synergistic approach enhances the responsiveness of applications to fluctuating workloads and ensures that resources are allocated precisely where they are needed, reducing waste and improving overall performance. Together, these autoscaling mechanisms provide a comprehensive solution to the challenges of managing cloud-native applications in dynamic environments.

REFERENCES

[1] T.-T. Nguyen, A. M. Rahman, Y. H. Tran, Q. M. Tran, and C. H. Choi, "Horizontal pod autoscaling in Kubernetes for elastic container orchestration," *Sensors*, vol. 20, no. 16, p. 4621, 2020. doi: 10.3390/s20164621.

[2] S. K. Mondal, Z. Zheng, and Y. Cheng, "On the optimization of Kubernetes toward the enhancement of cloud computing," *Mathematics*, vol. 12, no. 16, p. 2476, 2024. doi: 10.3390/math12162476.

[3] A. Mustyala, "Dynamic resource allocation in Kubernetes: Optimizing cost and performance," *EPH - International Journal of Science and Engineering*, vol. 7, no. 3, pp. 59–71, 2021.

[4] C. C. Chang, S. R. Yang, E. H. Yeh, P. Lin, and J. Y. Jeng, "A Kubernetes-based monitoring platform for dynamic cloud resource provisioning," in *Proc. IEEE Global Communications Conference (GLOBECOM)*, Singapore, 2017, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/8254046

[5] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring potential for non-disruptive vertical auto scaling and resource estimation in Kubernetes," in *Proc. IEEE Int. Conf. on Cloud Computing (CLOUD)*, Milan, Italy, 2019, pp. 33–40. [Online]. Available: https://ieeexplore.ieee.org/document/8814504

[6] M. Song, C. Zhang, and H. E, "An auto scaling system for API gateway based on Kubernetes," in *Proc. 2018 IEEE 9th Int. Conf. on Software Engineering and Service Science (ICSESS)*, Beijing, China, 2018, pp. 109–112. doi: 10.1109/ICSESS.2018.8663784.

[7] Y. Jin-Gang, Z. Ya-Rong, Y. Bo, and L. Shu, "Research and application of auto-scaling unified communication server based on Docker," in *Proc. 2017 10th Int. Conf. on Intelligent Computation Technology and Automation (ICICTA)*, Changsha, China, 2017, pp. 152–156. [Online]. Available: https://ieeexplore.ieee.org/document/8089924

[8] P. Townend, A. Basu, M. Eisa, and J. Kołodziej, "Improving data center efficiency through holistic scheduling in Kubernetes," in *Proc. 2019 IEEE Int. Conf. on Service-Oriented System Engineering (SOSE)*, Newark, CA, USA, 2019, pp. 156–166. [Online]. Available: https://ieeexplore.ieee.org/document/8705815

[9] D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of Kubernetes pods," in *Proc. IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Budapest, Hungary, 2020, pp. 8–12. [Online]. Available: https://ieeexplore.ieee.org/document/9110428

[10] E. Casalicchio and V. Perciballi, "Auto-scaling of containers: The impact of relative and absolute metrics," in *Proc. 2017 IEEE 2nd Int. Workshops on Foundations and Applications of Self Systems (FASW)*, Tucson, AZ, USA, 2017, pp. 207–214. [Online]. Available: https://ieeexplore.ieee.org/document/8064125

[11] J. Santos, T. Wauters, B. Volckaert, and F. D. Turck, "Resource provisioning in fog computing: From theory to practice," *Sensors*, vol. 19, no. 10, p. 2238, 2019. doi: 10.3390/s19102238.

[12] W. S. Zheng and L. H. Yen, "Auto-scaling in Kubernetes-based Fog Computing platform," in *Proceedings of the International Computer Symposium*, Singapore: Springer, 2018, pp. 338–345. doi: 10.1007/978-981-13-9190-3_35.