Dynamic Scaling and Performance Optimization for Microservices Using Kubernetes

ABSTRACT

The article discusses the capabilities of Kubernetes software in the process of managing microservice architecture. The tools of this software used in the scaling process were studied, which, for example, include: Horizontal Pod Autoscaler (HPA), and Cluster Autoscaler. Load balancing mechanisms were also described, including using Ingress controllers and network proxies to optimize traffic between workloads.
The methodology of the article includes an analysis of various approaches to the management of containerized microservices by Kubernetes, for example: automation of scaling processes, monitoring, security, and access control. The results of the research demonstrate that by using Kubernetes, the flexibility of the system increases, stable operation of services is maintained, the risk of failures is reduced, and prompt response to requests is provided with an increase in the number of users. The problems faced by developers when implementing this software were also considered, namely: security settings, optimization of auto-scaling, and configuration scheme of network interaction between components.The article is aimed at developers and architects of software systems that optimize microservice applications. In conclusion, recommendations are given on using Kubernetes to create a flexible, fault-tolerant microservice structure ready for high loads.

Keywords: Kubernetes, microservice architecture, automatic scaling, fault tolerance, load balancing, controllers, security, monitoring.

Introduction

The advancement of information technologies has led to increasing demands for application performance. Microservices architecture is becoming a significant approach in software system design. This method involves dividing software into independent modules—microservices. Each module operates autonomously, simplifying the adaptation of applications to various operational conditions.

As Kubernetes, being a container orchestration platform, provides tools that automate scaling processes, load balancing, and resource distribution, its mechanisms ensure software stability regardless of load levels.

Microservices-based applications impose specific requirements on performance and fault tolerance. Kubernetes offers developers functionality for resource management, module interaction, and application adaptation to dynamic loads. Implementing the platform involves configuring auto-scaling, optimizing network connections, and ensuring secure data transmission [10].

The purpose of this article is to explore the application of Kubernetes in the microservices architecture. The paper analyzes load management mechanisms, resolves bottlenecks, and establishes a robust structure capable of functioning efficiently under significant request volumes.

Materials and Method

Various methods were utilized to analyze the Kubernetes architecture and microservices management mechanisms. System analysis was employed to study the platform's structure and examine its functional features. As part of the research, comparisons were conducted on approaches to microservices orchestration, identifying the distinctive characteristics of Kubernetes and clarifying its differences from alternative solutions. A comprehensive approach ensured an accurate assessment of the platform's capabilities and facilitated the formulation of recommendations for improving the performance of microservices applications.

The following scientific works were considered as sources. The study by Sun Y. et al. [1] focuses on scheduling methods and load balancing in Kubernetes. The authors emphasize the importance of selecting algorithms that consider the dynamic nature of microservices and the ability to flexibly allocate resources to ensure system reliability.

The articles by Rossi F., Cardellini V., and Presti F. L. [2, 7] discuss the hierarchical scaling of microservices in Kubernetes, which enables adaptation to variable load conditions. It is argued that hierarchical scaling methods increase flexibility and enhance efficiency.

The research by Vayghan L. A. et al. [3] describes a Kubernetes controller for state management of microservices. The developed controller improves the system’s fault tolerance, which is essential for applications with specific availability requirements.

Ding Z., Wang S., and Jiang C. [4] study the placement of microservices with consideration of dynamic resource allocation, demonstrating an approach that optimizes resource usage.

Jian Z. et al. [5] proposed a reinforcement learning-based method to enhance the Kubernetes scheduler. This method improves scheduling efficiency, reducing delays under high-load conditions.

The practical research is based on works [6-9]. The research by Nurzhankyzy Asem [6] focuses on best practices for containerization in deploying microservices under heavy load, offering recommendations for container optimization to enhance system performance. Shulyak A. V. [8] provides practical guidelines for scaling resources using Kubernetes. Rathi G. et al. [9] analyze the performance of various Ingress controllers in a Kubernetes cluster, noting that the choice of controller impacts throughput.

The reviewed literature highlights the need for a deliberate selection of tools and methods for microservice management, with an emphasis on scalability, reliability, and efficiency under high-load conditions.

Results and Discussion

The command-line tool `kubectl` is used for managing applications in Kubernetes, enabling the deployment of containers based on prepared OCI images. The system employs pods, which are logical units that group containers within a shared context. Containers with close interdependencies are placed within the same pod, allowing them to share resources, including network namespaces and volumes.

Each pod has its IP address, facilitating communication between containers within the pod as well as with external systems. A volume serves as an immutable data storage shared among all containers in the pod, simplifying information exchange. Pod configurations are defined using YAML files, which specify operational parameters in detail.

The readiness probe procedure determines a pod’s ability to handle incoming traffic. The system monitors container health and restarts them in case of failures to maintain application stability.

In a microservices architecture, each service is isolated in its container. This approach creates independent environments for operation, simplifying component configuration, system updates, and maintenance tasks. Containers also support running multiple versions of services simultaneously, preventing conflicts during updates.

System reliability is ensured through monitoring and backup mechanisms. Data stored in the `etcd` controller is backed up to enable workload recovery. Monitoring tools such as Prometheus and Grafana provide insights into container status, resource utilization, and performance metrics. These tools assist in identifying and resolving issues, optimizing processes, and maintaining application stability without disruptions [6].

Additionally, the tools such as Horizontal Pod Autoscaler (HPA) and Cluster Autoscaler should be noted, as they enable automatic resource adjustment, providing intelligent scaling capabilities for the microservice architecture and increasing its adaptability to fluctuating load conditions. These tools facilitate interaction between application layers, contributing to resilient and efficient task management in a distributed environment.

Horizontal Pod Autoscaler (HPA) is a component focused on adjusting the number of pods based on specific load metrics, such as CPU and memory usage, or application-specific metrics (e.g., request volume per second). A key feature of HPA is its ability to autonomously monitor system status and react to changes, scaling up the number of pods during periods of high traffic and scaling down as the load decreases. While on the other hand, Cluster Autoscaler manages infrastructure at the node level, acting as a balancing mechanism.

The combined operation of these tools forms a multi-level, coordinated scaling system. Such interaction enables the creation of a highly adaptive system capable of effectively handling peak loads, optimizing both available computational resources and overall performance [7].

Security in this software is implemented through two main mechanisms. The first includes RBAC (Role-Based Access Control) and Network Policies, which restrict access at the network level and define the boundaries of service interactions, ensuring network segmentation and data isolation. The second mechanism implements mTLS (mutual Transport Layer Security) authentication, which mitigates the risk of data interception by enforcing mutual authentication between services. The mTLS implementation automates certificate lifecycle management - including creation, rotation, and revocation - which minimizes human error and reduces the likelihood of security-compromising misconfigurations.

Service Mesh tools enable the definition and enforcement of access rules, isolating unauthorized interactions between services and preventing the escalation of attacks within the system. For instance, mesh-level policies restrict access to authorized clients only, even in cases where vulnerabilities exist within the application code itself. This adaptability, combined with centralized control, strengthens the security of the Kubernetes cluster, making it more resilient to threats.

Containerization reduces the number of physical systems, lowering operational costs. The evaluation of microservice architecture efficiency considers parameters such as the number of simultaneous requests and response time for single server and cluster configurations. Testing is conducted in two scenarios: the first involves 10,000 connections with 10,000 requests each, and the second involves 100,000 connections with 1,000 requests per connection. Each scenario is repeated three times to minimize errors. The threshold load for automatic scaling is set at 80% of the echo server load. The results for different scenarios are presented in Tables 1 and 2.

Table 1. Scenario 1: CPU load (in milli-cores) [8].

Stage	Single Server (no scaling)	Multiple Servers (with scaling)
1	619	210
2	591	217
3	609	230
Average	606.34	219.00

Table 2. Scenario 2: CPU load (in milli-cores) [8].

Stage	Single Server (no scaling)	Multiple Servers (with scaling)
1	580	215
2	484	340
3	552	324
Average	555.00	293.00

In addition to CPU load, server response time was also measured. Table 3 presents the average response times.

Table 3. Average response times [8].

Response Time (ms)	Single Server (no scaling)	Multiple Servers (with scaling)
Scenario 1	43 ms	12 ms
Scenario 2	58 ms	32 ms

These data indicate that enabling automatic scaling reduces both CPU load and response time, ensuring high microservice performance even with a significant increase in the number of requests [8].

For secure external access to the workload, ingress controllers support SSL/TLS termination at the controller level, offloading decryption tasks from the services. The controllers also support path-based routing, allowing flexible request distribution to different services based on the URL structure:

```
paths:
  - path: /v1
    pathType: Prefix
    backend:
      service:
        name: v1-service
        port:
          number: 80
```

Kubernetes ingress controllers provide efficient management of external access and load balancing between services within a cluster. Understanding the architecture, functions, and mechanisms of ingress controllers enables the creation of flexible and scalable solutions for managing containerized applications. In-depth analysis of different controller capabilities allows for configuration adaptation based on workload characteristics, ensuring optimal performance [9].

Below, Table 4 describes the advantages and disadvantages of using Kubernetes for managing microservices under high-load conditions.

Table 4. Advantages and disadvantages of using Kubernetes for microservice management under high-load conditions [9].

Category	Advantages	Disadvantages
Scalability	- Automatic scaling efficiently distributes load and optimizes resources based on current demands.	- Complex configurations are needed to ensure proper scaling and operation.
	- Simplifies management of multiple microservices and their interactions under fluctuating traffic.	- High-load conditions may introduce delays during deployment and scaling processes.
	- Enables horizontal scaling to handle increased load, minimizing downtimes.	
Resilience	- Automatic container recovery and traffic redistribution during failures maintain system stability.	- Misconfigurations or resource shortages may hinder recovery efforts, causing delays.
	- Supports load balancing, evenly distributing requests among containers to reduce failure risks.	- Configuration conflicts may result in system downtimes.
Manageability	- Allows setting CPU and memory limits to	- Complex configurations may

	prevent excessive resource usage and maintain stability.	lead to opaque and challenging resource management.
	- Optimizes cluster resource allocation based on container needs.	- Requires meticulous monitoring and alerting to prevent resource overload.
Containerization	- Provides service isolation, reducing the risk of interference between microservices under heavy load.	- High support demands for multiple containers increase infrastructure costs.
	- Simplifies deployment and management using standardized methods.	- Constant monitoring of isolation is necessary to prevent failures.
Updates	- Supports continuous deployment (CI/CD), enabling updates without downtime under high load.	- Transitional states during deployments may cause unexpected downtimes or errors.
	- Rolling Updates and Blue-Green Deployment minimize risks during updates.	- Complex update configurations require additional resources.
Monitoring	- Integration with monitoring and logging systems (e.g., Prometheus, Grafana) provides real-time insights.	- Additional resources are needed for monitoring tools, increasing system load.
	- Delivers detailed metrics essential for performance analysis and troubleshooting.	- High metric granularity complicates analysis under heavy load.
Flexibility	- Supports tools and customization options tailored to specific system needs.	- Custom configurations require extra expenditure.
	- Easily integrates with other systems, making it a versatile solution for various infrastructure tasks.	- Plugins and integrations add complexity to cluster maintenance and management.

The use of Kubernetes for managing microservices under high-load conditions offers prospects for creating scalable systems adapted to evolving requirements. This tool encompasses a broad range of functions, with its effective operation relying on proper configuration. It serves as a foundation for developing architectures that ensure the stability, performance, reliability, and adaptability of applications operating in digital environments.

Conclusion

This study presented approaches to managing microservice architectures based on the Kubernetes platform. Existing methods aimed at maintaining system stability under changing loads and during the allocation of computational resources were also described.

The research examined tools such as Horizontal Pod Autoscaler and Cluster Autoscaler, which primarily focus on resource management. Additionally, network policy configurations and load-balancing mechanisms were explored. This analysis demonstrated that the use of Kubernetes facilitates the creation of a flexible microservices infrastructure that adapts to dynamic conditions, thereby maintaining operational stability.

Disclaimer (Artificial intelligence)

Option 1:

Author(s) hereby declare that NO generative AI technologies such as Large Language Models (ChatGPT, COPILOT, etc.) and text-to-image generators have been used during the writing or editing of this manuscript.

Option 2:

Author(s) hereby declare that generative AI technologies such as Large Language Models, etc. have been used during the writing or editing of manuscripts. This explanation will include the name, version, model, and source of the generative AI technology and as well as all input prompts provided to the generative AI technology

Details of the AI usage are given below:

1.

2.

3.

REFERENCES

1. Sun, Yu. and others (2023). Review of planning and load balancing methods in Kubernetes. In *4th International Conference on Computer Science, Parallel and Distributed Systems (ISPDS)*. IEEE, 284-290.

2. Rossi F., Cardellini, V., Presti, F. L. (2020). Hierarchical scaling of microservices in kubernetes. In *IEEE 2020 International Conference on Autonomous Computing and Self-organizing Systems (ACSOS)*. IEEE, 28-37.

3. Vaigan, L. A. et al. (2021). Kubernetes controller for managing the availability of stateful applications based on elastic microservices. In *Journal of Systems and Software*. 175, 110924.

4. Ding, Z., Wang, S., Jiang, S. (2022). Placement of Kubernetes-oriented microservices with dynamic resource allocation. In *IEEE Transactions on Cloud Computing*, 11 (2). 1777-1793.

5. Jian, Z. (2024). DRS: Advanced Kubernetes Scheduler for deep reinforcement learning for a microservices-based system. In *Software: practice and experience*, 54 (10), 2102-2126.

6. Nurzhankyzy, Asem Best Containerization practices for deploying microservices in high-load systems. [Electronic resource] Access mode: https://na-journal.ru/4-2024-informacionnye-tekhnologii/10854-luchshie-praktiki-kontejnerizacii-dlya-razvertyvaniya-mikroservisov-v-sistemah-vysokoj-nagruzki (accessed 10/27/2024).

7. Rossi, F. (2020). Geo-distributed efficient deployment of containers with Kubernetes. In *Computer Communications*,159, 161-174.

8. Yulyak, A.V. (2022). Introduction of resources using Kubernetes. In *Young Scientist*, 31 (426), pp. 8-13.

9. Rati, G. et al. (2024). Performance analysis of various input controllers in the Kubernetes cluster. In *IEEE 2024 International Conference on Information Technology, Electronics and Intelligent Communication Systems (ICITEICS)*, pp. 1-6.

10. Kurniawan, Dedy. 2022. “Towards Migrating from Monolithic-Based Web Application to Micro Service: A Case Study of EzScrum Product Backlog”. Journal of Engineering Research and Reports 23 (12):252-71. https://doi.org/10.9734/jerr/2022/v23i12782.