Performance Comparison between Monolith and Microservice Using Docker and Kubernetes

Napawit Toomwong and Waraporn Viyanon

***Abstract*—Nowadays, various technologies play more of a role in our daily lives. Most people can easily access technology via computers or mobile phones. It cannot be denied that technology is involved in all activities in life. On the other hand, in the world of software development, it is considered to be very much affected by the inability of systems to accommodate a huge amount of people who can access the website or application due to design architecture that no longer exists in an age where technology has evolved quickly.**

In this paper, we tested and analyzed the performance comparison between monolith and microservices using Docker and Kubernetes by developing a simulation system based on those concepts. This paper presents a performance comparison of web services using the same scenarios with two different factors: using a monolith and microservices on Docker and Kubernetes.

The results showed that the Monolith and microservices architecture developed with Kubernetes can reduce response time and increase throughput in the system. Moreover, we have explained the factors that make the system work in a more efficient way.

***Index Terms*—Monolith, microservices, docker, kubernetes.**

I. Introduction

In the present day, it cannot be denied that a good architectural design has to think about the efficiency of use. But, the design of a good architectural structure must meet the needs of the volume of users and easily adaptable to rapidly changing evolution of technology. Working with distributed software application is fundamentally different from implementing software on a single application. The main difference is that there are lots of things that have to concern such as availability, resilience, ease of deployment and replaceability. The challenge to transform a single software application to a distributed software application is that it needs to guarantee the quality of high performance and keep the user experience still the same. In this research, we simulated the software application using open-source software to present how these software impact on the application with different factors. The following sections describe the concepts of design architecture and the open-source software available to manage a large number of virtual hosts.

Monolith is a concept that combines services and business needs into a single software application that is suitable for a relatively small system. The advantage of this concept is that it is easy to develop a system does not take a long time, but the disadvantage of this concept is that all modules were tightly coupled inside a single application. Moreover, they often become very complex and difficult to implement and slow down any user who needs to work on the system, further increasing high impact when the project gets larger.

The definition of microservices [1] is an application component and standalone application of their own. Moreover, the independent application component can communicate using Restful API or Message Queue. The design concept of microservices is focused on service and business boundaries, as making it obvious where the code lives for a given piece of functionality.

Containerization [2] is a technique for simulating an environment or a virtual operating system to manage the software, which is similar to a virtual machine. The widely-known software is Docker. Docker is open source software developed to improve performance from traditional virtual machines that used much less resources than a virtual machine. Docker can install more than one container on a single operating system. It is unlike a virtual machine, which must be installed on its own operating system.

Kubernetes [3] is an open source system that helps manage, scale, and manage containers. It is designed to manage the container that run the applications and ensure that it is available twenty-four hours a day. Moreover, it provides a framework that resiliently run a distributed system.

Horizontal pod autoscaler [4] is a key feature of Kubernetes is used to change the shape of the system by automatically increasing or decreasing the number of pods in response to the workload of the CPU or memory consumption in response to the custom metrics reported by Kubernetes. The horizontal pod autoscaler algorithm operates using the ratio between the desired metric value and the current metric is calculated as in (1)

$$n = currentReplicas * \left(\frac{currentMetricValue}{desiredMetricValue}\right) \quad (1)$$

For example, if the current metric value is 200m, and the desired value is 100m, the number of replicas will be doubled, since 200.0 / 100.0 == 2.0. If the current value is 50m instead, the resulting number of replicas, since 50.0 / 100.0 == 0.5. It will skip scaling if the ratio is sufficiently close to 1.0 [5].

Autoscaling [6] is a method used in cloud computing to adjust the resources of the system based on its needs. For example, autoscaling allows the system to scale up or down, and has its own resources. These increased resources increased and were sufficient for the needs of incoming traffic. Furthermore, when no one is using it, the system automatically reduces it to normal size. It could reduce the burden for people in monitoring and letting the system take care of themselves. On the other hand, autoscaling used much

Manuscript received December 29, 2020; revised March 22, 2021.

Napawit Toomwong and Waraporn Viyanon are with the Srinakharinwirot University, Thailand (e-mail: napawit.toomwong@g.swu.ac.th, waraporn@g.swu.ac.th).

more for handling failure or reacting to load conditions. For example, the rule are specified clearly, for there should be at least 5 instances in a group, so if one goes down a new one is automatically launched. It can help to increase efficiency and reduce costs by using a platform that allows it to work optimize the computing resources.

The remainder of this paper described a case study on developing web services based on monolith and microservices using Docker and Kubernetes in Section II. Section III describes the architecture deployment on Amazon web services. Section IV shows the results of tested performances, such as throughput and response time. The last section describes the factor that cause the results.

II. Methodology

In this research, the performance testing is performed by simulating the loyalty system based on monolith and microservices concepts of Amazon web services [7], [8]. The system was developed by using Node.js (8.6.0), MongoDB (3.4.0), Docker (19.0.3) and Kubernetes (1.15.0), which consists of a browsing product list and product redemption. The responsibility of the first service is browsing the product list on the database and the last service uses spending to redeem products. In this experiment the request transaction is 300 transactions per second and 20 threads usage for 10 minutes generated by Apache JMeter.

A. Monolith Architecture Using Docker

The monolith architecture of the loyalty system consists of three parts: (1) service loyalty, (2) database, and (3) load balancer. Each part is installed on the Docker container. This monolith architecture is installed on Amazon EC2 T3.small (Ubuntu 18.0.4, 2 vCPUs, 2.5 GHz, Intel Skylake P-8175, 2 GiB memory). The database consists of two collections: users and products which stored 1,000 documents and 3 documents, respectively. The architecture of monolith is shown in Fig. 1.

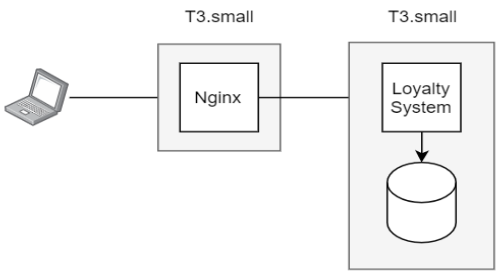

Fig. 1. Monolith architecture using Docker.

When the first service is called through nginx, the loyalty system retrieves information from the database and responds to a client. Points and stock items are charged by updating the database after the client selects a product for redemption. The sequence diagram is shown in Fig. 3.

B. Monolith Architecture Using Kubernetes

In this scenario, the system was installed in a Kubernetes cluster consisting of four nodes, one master and three workers. Master node is a node controlling and managing a set of worker nodes. The worker nodes are a worker machines in Kubernetes, previously known as a minion. The specification of each node is Amazon EC2 T3. small (Ubuntu 18.0.4, 2vCPUs, 2.5 GHz, Intel Skylake P-8175, 2 GiB memory). This cluster automatically increases and decreases the number of pods across their workers to maintain an average CPU utilization, when CPU usage is over than 50 percent. The monolith architecture is similar to the previous one, but in this cluster the software used to manage the request transaction is ingress-nginx, which is also installed on the cluster. The architecture of monolith using Kubernetes is shown in Fig. 2.

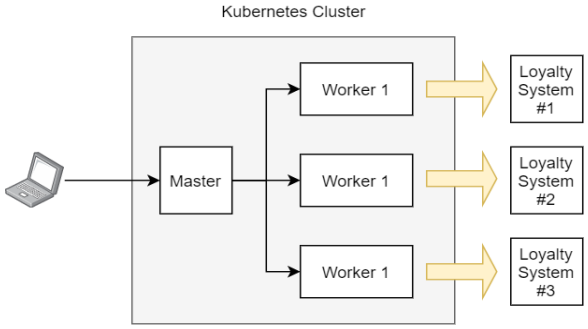

Fig. 2. Monolith architecture using Kubernetes.

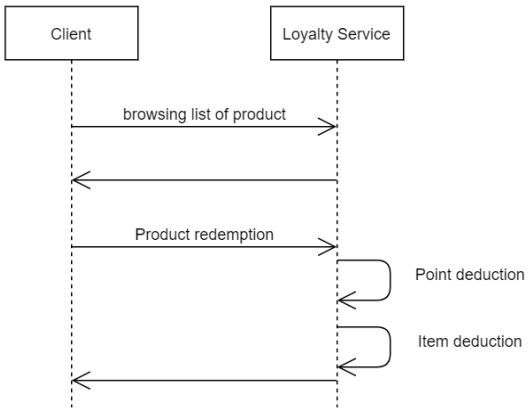

Fig. 3. Sequence diagram of monolith architecture.

C. Microservices Architecture Using Docker

The loyalty system developed with microservices architecture break the monolith architecture into small pieces that work together. In this system, there are 3 domains, including user, catalog, and redemption. The architecture consists of 3 instances which run on Amazon EC2 T3. small (Ubuntu 18.0.4, 2 vCPUs, 2.5 GHz, Intel Skylake P-8175, 2 GiB memory). The responsibility of the first instance is managing the requests from a client using nginx [6]. The redemption and user services were installed in the second instance. The last instance is catalog services, which retrieve the product list from the database. The services communicate in this architecture using HTTP/REST protocols. The services can be developed and deployed independently of one another. The architecture overview is shown in Fig. 4.

The microservices was deployed as shown in Fig. 4. Each service has its own database in order to be decoupled from other services. The catalog database stored the record of products that client can redeem and the user databases collect points and user information. Both of them stored the number of record similar to the monolith architecture.

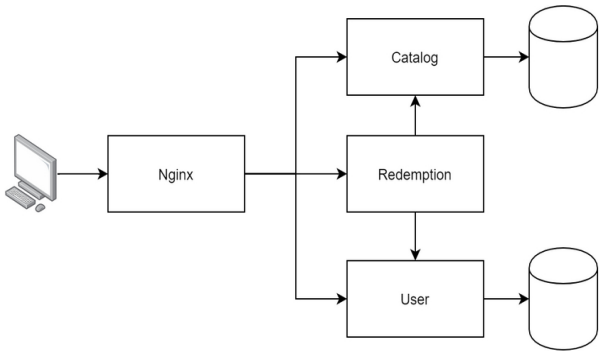

Fig. 4. Microservices architecture using Docker.

D. Microservices Architecture Using Kubernetes

In this approach, microservices using kubernetes are used extensively in the IT industry to handle large workloads and services, which facilitate both declarative configurations and automation. In this section, there were 4 instances, including one master node and three workers. The architecture and resources were similar to the microservices using docker but nginx was replaced by ingress-nginx.

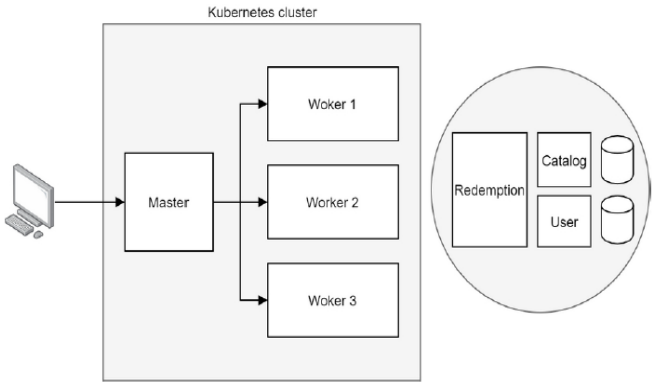

Fig. 5. Microservices architecture using Kubernetes.

The microservices architecture was deployed as it is shown in Fig. 5. This cluster also implemented the horizontal pod autoscaler, which can adjust the number of pods if the CPU utilization more than 50 percent.

III. Experimental Results

Apache JMeter was used in the experiment to push a 300 transactions per second load for 10 minutes. The experiment was performed three times for the performance comparison of the monolith and microservices architecture using Docker and Kubernetes. The variables used to measure the usage efficiency Fig. 6. Sequence diagram of microservices architecture is the average response time throughout. Response time [9] is the actual time to process the request, it includes network delays and queuing delays. The use of Kubernetes can increase the system performance and does not affect the efficiency of the original system, which will explain the results of various experiments in the next section.

A. Results of Monolith Architecture

The simulation experiment of loyalty system was developed using monolith architecture compared with Docker and Kubernetes on Amazon web services which all services were embedded into the single codebase. In the previous experiment, the monolith architecture using Docker was summarized in Table I. The performance result of monolith architecture using Kubernetes are summarized in Table II. It was found that the CPU usage of monolith architecture using Kubernetes has increased more than 50 percent of the CPU limit usage. As a result, the deployment was resized to three replicas, as shown in Table III.

TABLE I: Result of Monolith Using Docker

Services	Average response time	Throughput (t/s)
Catalog	18.28	149.50
Redemption	20.80	149.58
Total	19.54	298.96

TABLE II: Result of Monolith Using Kubernetes

Services	Average response time	Throughput (t/s)
Catalog	5.25	150.01
Redemption	5.71	150.00
Total	5.48	299.99

TABLE III: Number of Pods in Kubernetes Horizontal Pod Autoscaler

Services	Min pods	Max pods	Replicas
Loyalty	1	10	3

B. Results of Microservices Architecture

The microservices performance tests using Docker and Kubernetes, the microservices using Kubernetes has a better performance than Docker. The results of the experiment are summarized in Table IV and Table V. It was found that the microservices architecture using Kubernetes increased more than 50 percent of the CPU limit usage. As a result, the deployment of catalog services and redemption services were resized to three replicas. The results are shown in Table VI.

TABLE IV: Result of Microservices Using Docker

Services	Average response time	Throughput (t/s)
Catalog	46.29	141.27
Redemption	83.17	141.26
Total	64.73	298.96

TABLE V: Result of Microservices Using Kubernetes

Services	Average response time	Throughput (t/s)
Catalog	7.82	149.81
Redemption	27.99	149.81
Total	17.90	299.60

TABLE VI: Number of Pods in Kubernetes Horizontal Pod Autoscaler

Services	Min pods	Max pods	Replicas
Catalog	1	10	3
Redemption	1	10	3
User	1	10	1

C. Comparison of Monolith and Microservices Using Docker and Kubernetes

As shown in the result tables above, when comparing the results of the experiment, it was found that Kubernetes improved system performance. The results of the comparison are summarized in Fig. 6 and Fig. 7.

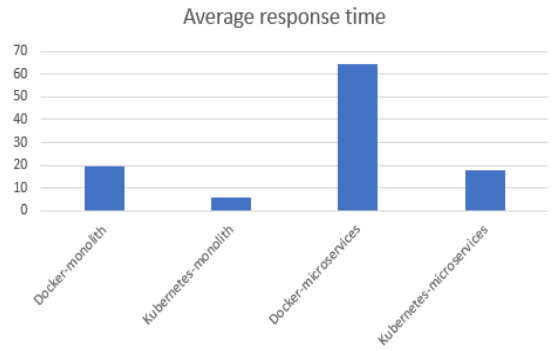

Fig. 6. Average response time of monolith and microservices using Docker and Kubernetes.

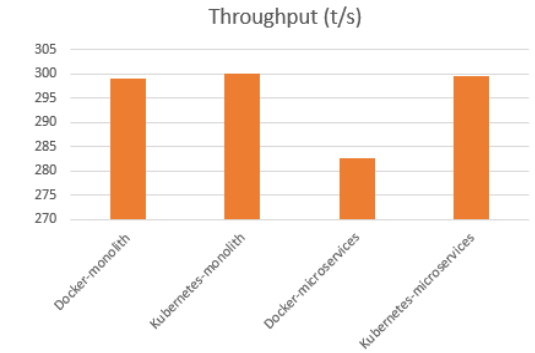

Fig. 7. Throughput of monolith and microservices using Docker and Kubernetes.

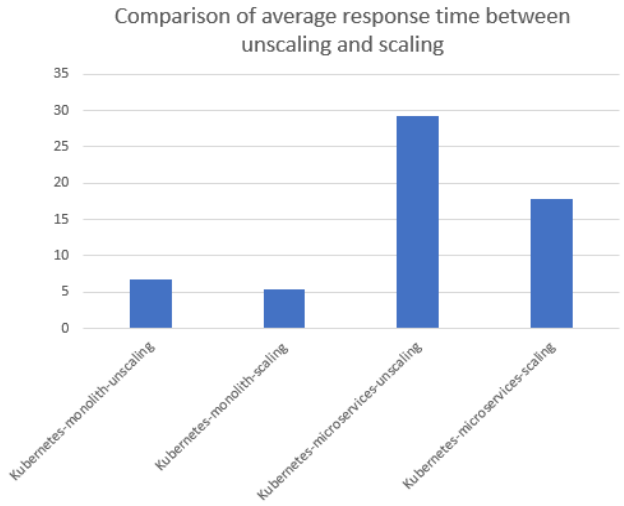

Fig. 8. Comparison of average response time between unscaling and scaling system.

Based on the chart in Fig. 6 and Fig. 7, the response time of the architecture that uses Kubernetes is better than Docker. It was discovered that running scalable work-loads on Kubernetes reduced the response time by approximately 25 percent and increased throughput in the system, which can be used with any monolith and microservices architecture effectively and without any impact. It can be observed that a scaled system reduced response time and increased throughput. Therefore, to find the root cause of this experiment, it was tested by comparing the average response time of monolith and microservices using Kubernetes, which implemented the horizontal pod autoscaler, which can adjust the number of pods if the CPU utilization is more than 50 percent and the other part developed with Kubernetes that uses 100 percent of the CPU, but without the ability to scale. The results of the comparison are summarized in Fig. 8.

Based on the chart in Fig. 8. It was discovered that running scalable work-loads on Kubernetes reduced the response time more than the system that running with full CPU without the ability to scale.

IV. Conclusion

Building autoscaling allows developers to automatically scale both monolith and microservice systems. There are 2 methods to scaling the system. First, predictive scaling can be triggered by well-known trends. For example, the system peak load is between 9 a.m. and 5 p.m., so developers can plan to bring up additional instances before the peak time and shut down the instances to reduce costs. In this method, developers should understand the behavior of the system workload over time. The last method is reactive scaling, building reactive scaling by bringing up additional instances when the system detects a higher system utilization exceeding the upper threshold. To ensure that the threshold is appropriate for the production environment it requires load test to test autoscaling rules.

The definition of bounded context is a specific responsibility enforced by explicit boundaries [10]. With regard to microservice, developers need to understand the bounded contexts of service. Getting service boundaries wrong can result in having to make lots of changes in services-to-services collaboration and expensive operation. If the development and business teams are clearly not able to understand the business domain that consists of multiple bounded contexts, consider starting development from Monolith is still a good choice.

This research was developed to find the efficiency of using a monolith and microservice architecture with modern technology such as Docker and Kubernetes. It presents the scalability of a system designed to accommodate existing customers and to prepare for new customers in the future with less effort and cost.

Conflict of Interest

The authors declare no conflicts of interest concerning the content matter of this manuscript.

Author Contributions

Napawit Toomwong and Waraporn Viyanon have participated in designing and conducting research for analysis of experimental results and manuscript writing.

Acknowledgment

This research was funded by the Graduate School of Srinakharinwirot University, Thailand. We would like to thank the instructors from the Department of Computer Science at Srinakharinwirot University.

References

[1] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, O'Reilly Media, Inc., 2015.
[2] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using docker technology," in *Proc. SoutheastCon*, 2016.
[3] Concepts. [Online]. Available: https://kubernetes.io/docs/concepts/

[4] A. Zhao, Q. Huang, Y. Huang, L. Zou, Z. Chen, and J. Song, "Research on resource prediction model based on kubernetes container auto-scaling technology," *IOP Conference Series: Materials Science and Engineering*, vol. 569, issue 5, p. 052092, 2019.
[5] Horizontal pod autoscaler. [Online]. Available: https://kubernetes.io/docs/tasks/runapplication/horizontal-pod-autoscale/how-does-the-horizontal-podautoscaler-work
[6] J. Yang, C. Liu, Y. Shang, Z. Mao, and J. Chen, "Workload predicting-based automatic scaling in service clouds," in *Proc. 2013 IEEE Sixth International Conference on Cloud Computing*, 2013, pp. 810–815.
[7] M. Villamizar, O. Garc´es, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Proc. 2015 10th Computing Colombian Conference (10CCC)*, 2015, pp. 583–590.
[8] M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano *et al*., "Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures," in *Proc. 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 179–182.
[9] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems*, O'Reilly Media, Inc., 2017.
[10] V. Vernon, *Domain-Driven Design Distilled*, Addison-Wesley Professional, 2016.

Copyright © 2021 by the authors. This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited (CC BY 4.0).

Napawit Toomwong has graduated with a bachelor's degree in computer science from Chiang Mai University, Thailand (2011-2015). He is currently studying for a master's degree in computer science at Srinakharinwirot University. He works at an IT company as a senior software engineer and takes part in every stage of applications development, including design and advanced applications.

Waraporn Viyanon got her Ph.D. in computer science from Missouri University of Science and Technology, formerly named University of Missouri at Rolla, Missouri, USA. She is a faculty member at the Computer Science Department at Srinakharinwirot University. Her research interests include data analytics, Artificial Intelligence (AI), cloud computing, web, mobile technologies, and database management.