Automated Resource Management System Based upon Container Orchestration Tools Comparison

B. Purahong, J. Sithiyopasakul, P. Sithiyopasakul, A. Lasakul, and C. Benjangkaprasert *

Computer Engineering Department, School of Engineering, King Mongkut's Institute of Technology Ladkrabang, Bangkok, Thailand; Email: boonchana.pu@kmitl.ac.th (B.P.), 63601074@kmitl.ac.th (J.S.), paisan.si@kmitl.ac.th (P.S.), attasit.la@kmitl.ac.th (A.L.)
*Correspondence: chawalit.be@kmitl.ac.th (C.B.)

***Abstract*—The goal of this article is to study and analyze the container orchestration technology Kubernetes, Docker Swarm, and Apache Mesos by performing performance evaluations and inspecting how many requests and responses the server can handle. Due to the fact that managing information system resources is a challenge in terms of performance, usability, reliability, and the cost of information resources. Some orchestration tools cannot automatically allocate resources depending on the scope of the information system resource management. This leads to allocating resources more than the needs of system requirements, resulting in excessive costs. Therefore, this article proposed testing the system by measuring its effectiveness using a structured process by examining measurement variables such as the number of requests per second, number of responses to requests, and resource extension period using all three-orchestration technology. From the testing and analysis of all three variables as mentioned, it is possible to know the efficiency of the Kubernetes technology in such a similar environment and compared it with other orchestration tools like Docker Swarm and Apache Mesos orchestrator. For Kubernetes, Docker Swarm, and Apache Mesos, the mean value of its handling average request per minute is 30,677.25/min, 33,688.67/min, and 29,682.6/min, respectively. Swarm performed better in aspects of handling requests per minute by 9.35% of the difference when compared to Kubernetes and by 12.64% when compared to Apache Mesos. However, there are several things which should be taken into consideration because each orchestration tool has its own strong and weak points. The testing experiment could display a piece of information on the dashboard for visualization and analytic purposes and there is an elaboration at the end of when to use which container orchestration tool to suit the business proposes the most.**

***Keywords*—resource management system, Kubernetes, docker swarm, Apache Mesos, container orchestration**

I. Introduction

To obtain the optimal results of how the server can handle massive requests, it is always possible when the server receives too many requests in a short period of time causing the server down which in real-world production this problem directly affects the benefits and loss for the company which in this aspect is very crucial. Therefore, this proposed research will greatly help user to understand which of the container orchestration tools are the most suitable choice for them together with the elaboration of pros and cons of each orchestration tools.

Nowadays, cloud computing is used to supply information resources, and there are multiple cloud providers to choose from, including Azure Cloud Services, Amazon Web Services, and Google Cloud Platform. The open-source community's most popular platforms for coordinating containers include Kubernetes [1–7], Docker swarm [4, 8–10], and Apache Mesos [11]. The allocation of information system resources can generally be done by the system administrator to allocate the resources and fix when the system is not working normally, but now there is Kubernetes technology (K8S) [1] that can manage information resources automatically such as creating new Virtual Machines based on the number of requests that come to the system.

There is one K8S autoscaling system that estimates resource requirements called Resource Utilization Based Autoscaling System (RUBAS) which could adjust the allocation of running containers in a K8S cluster [1]. Also, it is possible to adjust and address the problem of managing resources with the use of absolute metrics [2] which results in enabling more accurate scaling decisions when facing CPU intense workload. Ferreira and Sinnott [3] showed how they experimented on relative usage measures that show how to assess the performance of Kubernetes Horizontal Pod Auto-Scaling (KHPA) to analyze more in CPU utilization and response time. K8S must be configured since system design to comply with K8S limitations, K8S users must understand HTTP, Application Programming Interface, and Microservice design [12–14]. Moreover, for the aspects of the reliability of the deployed server, it is very crucial to always keep the server running even when there is a scenario where the server handles an intense workload.

II. Hypothesis and Research Scope

The hypothesis for this research is K8S orchestration tools can be configured and adapted more to get better

Manuscript received July 1, 2022; revised August 12, 2022; accepted November 11, 2022; published June 1, 2023.

results in terms of handling massive requests in a short period of time. To be specific, the Horizontal Pod Autoscaler (HPA) configuration or Horizontal Pods AutoScaling would have a big role to get better performance and could receive more requests when compared to a default configuration deployment.

When it comes to K8S and Docker Swarm comparison, it was known that in Docker Swarm there is no auto-scaling provided by service. Instead, Swarm only supports scaling up or down manually with commands. Therefore, K8S might have a better performance in aspects of auto-scaling with HPA and might handle requests from users greatly and better than Docker Swarm.

The objective of this research is a presentation of an automated information resource management system based on Kubernetes technology, Docker Swarm, and Apache Mesos (DC/OS). Which featured to reduce the cost of deploying Cloud services and to increase the reliability of the system. The system consists of Kubernetes managing all available information resources. The system operates based on the amount of user traffic passing through the HTTP channel at the time; as access increases or decreases, the system instructs the Virtual Machines to increase or decrease the number of services. While the Admin is only responsible for observing the system from a distance, there is no need to manually allocate the system. Kubernetes is known for its scalability, and it includes a number of tools that allow both infrastructure and applications to be hosted on it to scale the workload based on requisition, efficiency, and configured metrics. When diving into managing the resources, it is crucial for operating applications once they are in production. Therefore, the tests were conducted which to solve the problem of how to make the Kubernetes server be able to handle a massive workload without having any traffic failures. Moreover, there are some comparisons on configuring and optimizing the usage of resources by adjusting the HPA configuration to obtain the optimal results and compare with other orchestration tools such as Docker Swarm and Apache Mesos which has functionality similar to the Kubernetes orchestration tool.

III. System Design and Development

A. *Kubernetes, Docker Swarm, and Apache Mesos System Design Comparison*

Fig. 1 shows the overall system design workflow diagram of K8S, Docker Swarm, and Apache Mesos in comparison. In which one thing that these three tools have in common is these individuals were deployed by using Google Cloud Platform. Moreover, the deploying processes of these three tools are quite similar including uploading an image, configuring the deployment file, and measuring the outcome efficiency. All of the mentioned processes require some fundamental knowledge of container technology which is based on Docker. Despite the aforementioned, there are some noticeable differences in Apache Mesos when compared to others due to the fact that Apache Mesos itself requires terraform as an external service in order to make the deployment successful.

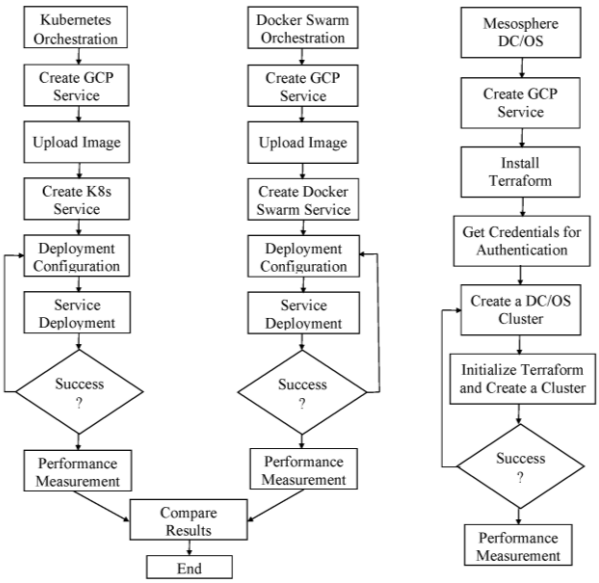

Figure 1. Flow diagram of Kubernetes, docker swarm, and Apache Mesos results comparison.

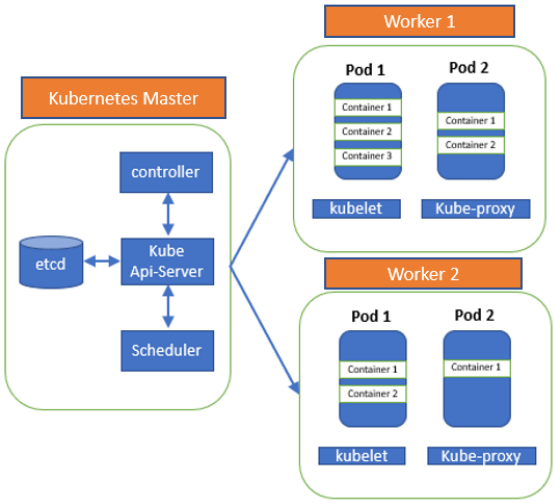

Figure 2. The overall architecture of Kubernetes.

Fig. 2 shows the K8S cluster's overall system structure. It consists of two types including the master node and slave node or worker node. K8S cluster can consist of multiple master nodes for high availability. However, by default, there is a single master node which is a controlling node for others slave nodes or nodes that are not masters. In each slave node, there are containerized applications that were deployed and encapsulated in a pod. The master node consists of various components. For instance, kube apiserver, kube controller manager, cloud controller manager, and kube scheduler. The master node is responsible for monitoring and controlling the usage of each slave node by displaying it in the form of a record. This indicates how much computing resources each node is using. For example, the usage of the estimator unit, the usage of memory, or even the usage of

writing and reading from the storage (Read/Write storage).

Docker Swarm is another open-source container orchestration platform built and maintained by Docker, as shown in Fig. 3. Basically, Docker Swarm converts multiple Docker instances into a single virtual host. A Docker Swarm cluster generally contains three main parts including nodes, services and tasks, and load balancers which the node structure is very similar to K8S but there are major differences. Unlike K8S, Docker swarm services can only be scaled with a command and there is no automatic way to scale. The interaction structure consists of three main sides. The admin interacts with the server orchestrator of the Docker Swarm deployment configuration in the Swarm manager. The admin role is to configure the application deployment metrics including choosing the based image and configuring scaling services. The Docker Swarm Manager itself is the deployment results of services created from the Nginx image.

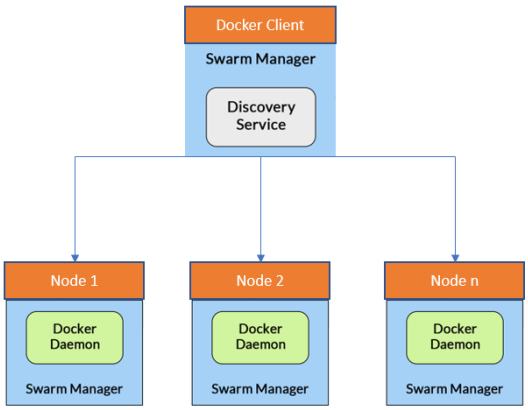

Figure 3. The overall architecture of Docker Swarm.

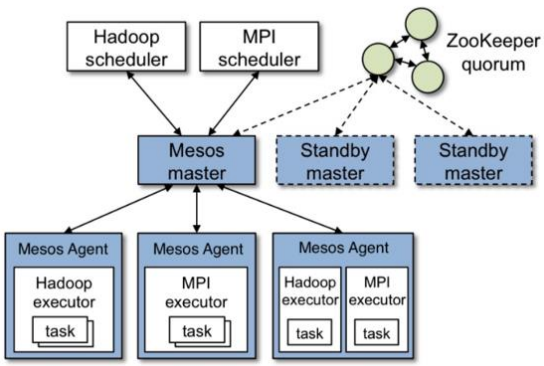

Figure 4. The overall architecture of Apache Mesos.

As shown in Fig. 4, all running tasks on DC/OS (Apache Mesos) are containerized and the container can be started by downloading images from a docker repository such as Docker Hub. The operating system of DC/OS is based on Linux which abstracts the cluster hardware and software resources and provides service on top of cluster management and container orchestration functionality. In the DC/OS cluster, it is possible to have several master nodes to manage the worker nodes or Mesos agent (Agent node) which in the individual agent node contained our containerized application or running Nginx docker images. The purpose of Zookeeper is to arrange the master node as the Hadoop or MPI scheduler and Zookeeper is one example of arranging the master node. DC/OS includes Marathon as a core component for a scheduler. With Marathon, it provides the ability to reach extreme scale, scheduling tens of thousands of tasks across thousands of nodes. It is possible for highly configurable declarative application definitions to enforce advanced placement constraints with node, cluster, and grouping affinities.

B. Based-Line Deployment Configuration and Tools

In this research, the Apache JMeter application was chosen to test sending HTTP requests to the server orchestrator and to see how many requests K8S server, Docker Swarm, and Apache Mesos can handle. Also, the application was originally designed for testing Web Applications but has since expanded to other test functions. There are many other usages that JMeter can do. For example, analyze and measure the performance with the built-in integration of a customizable dashboard [15]. Performance testing means testing a web application against heavy load, multiple and concurrent user traffic. The JMeter basic workflow is shown in Fig. 5. Firstly, the JMeter creates a request to the target server then it collects and calculates statistical information. Finally, the report is generated whether in form of a dashboard or table. The test result can be displayed in a different format such as a chart, table, tree, or even log file.

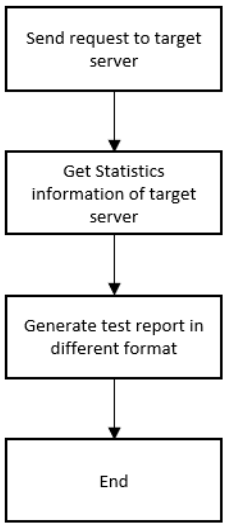

Figure 5. Basic workflow of JMeter.

In JMeter, there is one feature that can set up the number of threads that would like to send., the number of users (threads) which in the experiment was set up to 400 threads. By the meaning of this, it means that there is a simulation of sending requests with 400 users concurrently at the same time. The duration, the based line of sending a request duration was set up to 600 seconds or 11 minutes.

The selected based-line Docker image for this experiment was nginx. Basically, nginx is open-source software for web serving, reverse proxying, caching, load balancing, media streaming, and more. The nginx image was picked because of its HTTP server capabilities and is also designed for cloud-native architectures. Moreover, nginx functions as a load balancer for HTTP, TCP, and UDP servers.

C. Experimental Design on Adjusting CPU Target Utilization of Kubernetes Horizontal Pod Autoscaler (HPA) Based on Samples Metrics

In this system design, the K8S usage management of the metrics resources will be shown. Specifically, using Horizontal Pod Autoscaler, by adjusting the resources consumed by the application based on the actual load in real-time by HPA in K8S supports CPU and memory metrics. The Horizontal Pod Autoscaler (HPA) automatically scales the number of replicas according to the configured metric of target CPU utilization percentage. In other words, the number of pods in a replication controller, deployment, replica set, or stateful set. In this experiment, four target CPU utilization values will be adjusted and tested to obtain the results which consist of 20%, 40%, 60%, and 80%, as shown in Table I. All of the deployment images are nginx and every min and max replicas metrics were set to 1 and 5 respectively.

TABLE I. CONFIGURATION OF HPA METRICS

Based-line Images	target CPU Utilization Percentage	min Replicas	max Replicas
nginx	20%	1	5
nginx	40%	1	5
nginx	60%	1	5
nginx	80%	1	5

D. Experimental Design on Docker Swarm Scaling Services

Docker Swarm is another open-source container orchestration platform built and maintained by Docker. Basically, in Docker Swarm converts multiple Docker instances into a single virtual host. A Docker Swarm cluster generally contains three main parts including nodes, services and tasks, and load balancers which the node structure is very similar to K8S but there are major differences. Unlike K8S, Docker swarm services can only be scaled with a command and there is no automatic way to scale.

For this system design on Docker swarm manager, for each service, the number of tasks were declared to be scaled up or down which includes 1, 3, and 5 replicas, as shown in Table II. Similar with K8S, Docker Swarm can also define the actual state and expressed desired state. The swarm manager node constantly monitors the cluster state which will reconcile any failures. For instance, if you set up a service to run 5 replicas of the container, and a worker machine hosting two of those replicas' crashes, the manager will automatically generate two new replicas to replace replicas that crashed.

TABLE II. DOCKER SWARM SERVICES CONFIGURATION

Based-line Images	Number of Scaling services
nginx	1
nginx	3
nginx	5

E. Experimental Design on Mesosphere DC/OS Scaling Services

For this system design on DC/OS (Apache Mesos), for each service, the number of instances were declared to be scaled up or down which includes 1, 3, and 5 instances as shown in Table III. Similarly with Docker Swarm and K8S, it can also define the number of deployed instances. The cluster manager or master node constantly monitors the cluster state which will reconcile any failures. For instance, if you set up a service to run 5 services of the container, and a worker machine hosting two of those services crashes, the manager will automatically generate two new services to replace services that crashed [11].

DC/OS includes a group of agent nodes that are coordinated by a group of the master nodes which is similar to the K8S master node and worker node structure. As a cluster manager, it manages both resources and running tasks on the agent node. The agent node of DC/OS provides resources, and those resources are available to registered schedulers. Moreover, a container platform of DC/OS includes two built-in task schedulers which are Marathon and DC/OS (Metronome), and two combined container runtimes (Docker and Mesos). This functionality can be referred to as container orchestration. It also supports custom schedulers for handling more complex application workloads and operational logic.

TABLE III. DC/OS SERVICES CONFIGURATION

Based-line deployed services	Number of Scaling instances
nginx	1
nginx	3
nginx	5

IV. EXPERIMENTAL RESULTS AND DISCUSSION

In this section, the experimental results will be shown which includes some of the discussion. The experimental results are divided into three main parts. Firstly, the results after adjusting the CPU target utilization of Kubernetes Horizontal Pod Autoscaler (HPA) based on samples metrics and the resources consumed by the application based on the actual load in real-time by HPA in K8S supports CPU and memory metrics [15]. Secondly, the experimental results on Docker Swarm scaling services which in Docker Swarm converts multiple Docker instances into a single virtual host. Finally, the comparison of how both of these orchestration tools can handle a massive request which is a comparison between K8S and Docker swarm.

A. Experimental Results on Adjusting the Kubernetes Horizontal Pod Autoscaler (HPA)

The Kubernetes CPU target utilization adjustment of the metrics resources is shown. The adjustment of HPA

based on actual load automatically scales the number of replicas according to the configured metric of target CPU utilization percentage which includes 20%, 40%, 60%, and 80% to see the performance of how the individual adjustment can handle requests the best by letting users send requests for 400 threads at the same time with duration of 11 minutes. Fields that were selected to be shown from the table are as follows: Time (Min), response code status 200 (Successfully sent request), non-HTTP response code (Error sent request), throughput (Hits per second), and total requests.

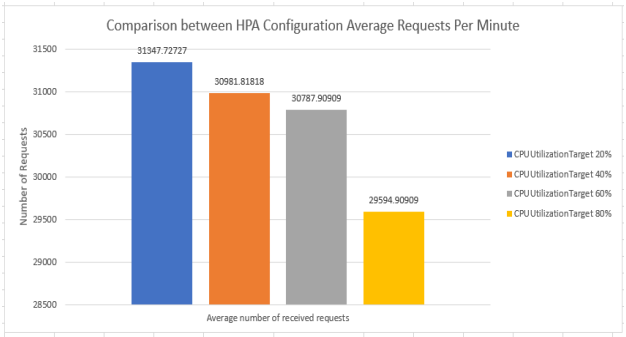

Figure 6. Comparison between HPA configuration average of received requests per minute.

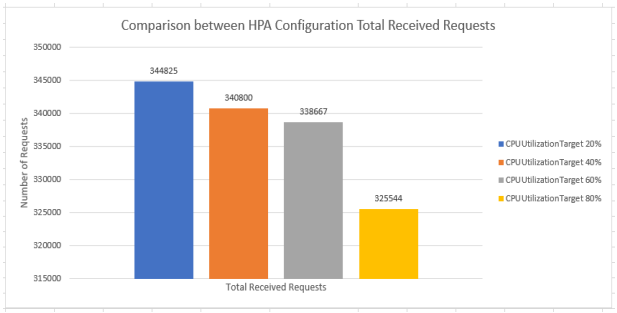

Figure 7. Comparison between HPA configuration total received requests.

The chart illustrates the average received requests comparison between HPA configuration of target CPU utilization (20%, 40%, 60%, and 80%) as shown in Fig. 6. Overall, the CPUUtilization-Target80% has the lowest performance with the number of averages received requests of 29594, due to slowly scale up the replicas to meets the requirements, it poorly handled the workload. Following with CPUUtilization-Target60% with the average requests of 30787, this configured target slightly having a better performance than CPUUtilization-Target80%. In term of performance, CPUUtilization-Target40% beaten both CPUUtilization-Target60% and CPUUtilization-Target80% by having average requests at 30981. On the other hand, CPUUtilization-Target20% performed the best with an average request of 31347 which is more than the average request of CPUUtilization-Target80%, CPUUtilization-Target60%, and CPUUtilization-Target40% by 5.75%, 1.8%, and 1.17% respectively. The aspects of how the server could handle the performance of a massive request also applied the same with the total received request as shown in Fig. 7.

B. Experimental Results on Docker Swarm Scaling Services

For these experimental results on Docker swarm manager configuration of services, the adjustment of services or replicas based on actual load results is shown in this section [4]. For each scaling service, the number of tasks was declared to be scaled up or down which includes 1, 3, and 5 replicas. The one downside that Docker swarm cannot provide like K8S is that in Swarm there are no autoscaling features. When it comes to scalability in Docker Swarm, services can be scaled through Docker Compose YAML templates and only support scaling up or down with commands.

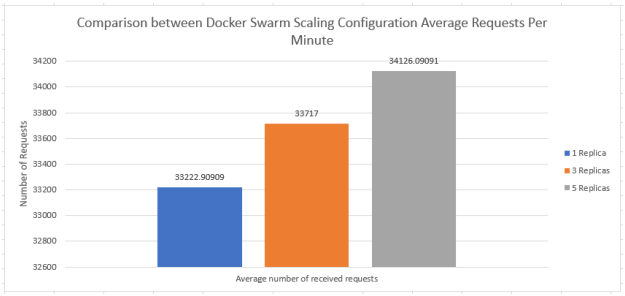

Figure 8. Comparison between Docker Swarm configuration average of received requests per minute.

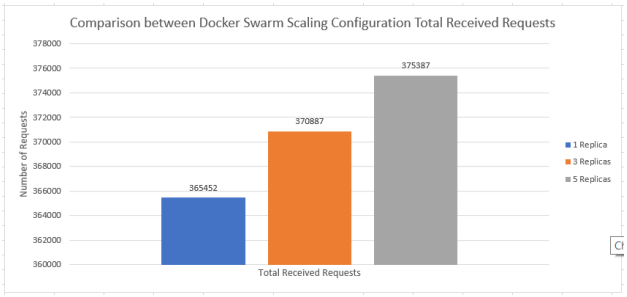

Figure 9. Comparison between Docker Swarm configuration total received requests.

Overall, Swarm allows users to deploy and scale faster and in an easier way, considering it enables scaling on demand. To see the performance of how the individual adjustment can handle requests the best by letting users send requests for 400 threads at the same time with a duration of 10 minutes. Fields that were selected to be shown from the table are as follows: Time (Min), response code status 200 (Successfully sent request), non-HTTP response code (Error sent request), throughput (Hits per second), and total requests.

The chart illustrates the average received requests comparison between Docker Swarm replicas scaling (1, 3, and 5 replicas) as shown in Fig. 8. Overall, the Swarm with single replicas performed the least with the number of averages received requests of 33,222, due to having just only a single service to handle requests. Followed by Swarm with 3 replicas with average requests of 33,717. The Swarm server with 5 replicas beats both Swarm with 1 replica and 3 replicas by having average requests of 34,126. To summarize, Swarm with replicas of 5 performed the best which handled requests more than the

average request of Swarm with replicas of 1 and Swarm with replicas of 3 by 2.68% and 1.2% respectively. Swarm with 5 replicas slightly has a better performance than 1 replica and 3 replicas. From the aforementioned, the same thing could be applied for the number of totals received requests as in Fig. 9 in terms of performance differences in percentage.

C. *Experimental Results on Mesosphere DC/OS (Apache Mesos) Scaling Services*

For these experimental results on Mesosphere DC/OS configuration of services, the adjustment of instances based on actual load results is shown in this section. For each scaling instance, the number of tasks was declared to be scaled up or down which includes 1, 3, and 5 instances. The key differences downside of DC/OS is that the mesosphere installations are quite complicated when compared to Docker Swarm and K8s because it needs to implement infrastructure which requires Terraform installation and Google API to connect with the cloud platform before the deployment.

When it comes to scalability, DC/OS provides a user interface to scale which is convenient. To see the performance of how the individual adjustment can handle requests the best by letting users send requests for 400 threads at the same time with a duration of 11 minutes. Fields that were selected to be shown from the table are as follows: Time (Min), response code status 200 (Successfully sent request), non-HTTP response code (Error sent request), throughput (Hits per second), and total requests.

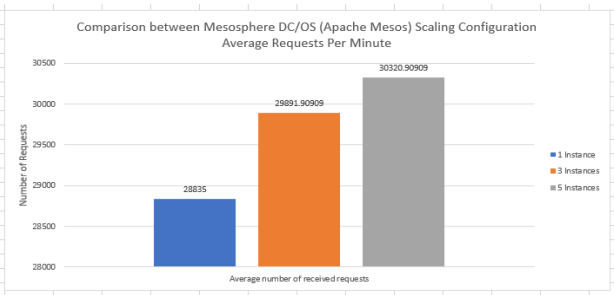

Figure 10. Comparison between Mesosphere DC/OS (Apache Mesos) configuration average of received requests per minute.

The chart illustrates the average received requests comparison between DC/OS instance scaling (1, 3, and 5 instances) as shown in Fig. 10. Overall, the DC/OS with a single instance performed the least with the number of averages received requests of 28,835, due to having just only a single service to handle requests. Following with DC/OS with 3 instances with average requests of 29,891. The DC/OS server with 5 instances beats both DC/OS with 1 instance and 3 instances by having average requests of 30,320. To summarize, DC/OS with instances of 5 performed the best which handled requests more than the average request of DC/OS with an instance of 1 and DC/OS with instances of 3 by 5.02% and 1.42% respectively. DC/OS with 5 instances has a better performance than 1 instance and 3 instances. From the aforementioned, the same thing could be applied for the number of total received requests as in Fig. 11 in terms of performance differences in percentage.

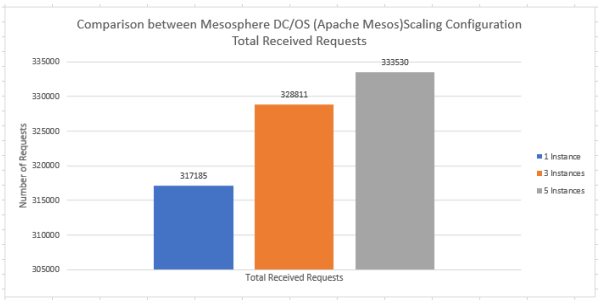

Figure 11. Comparison between Mesosphere DC/OS (Apache Mesos) configuration total received requests.

D. *Comparison Results between K8S and Docker Swarm*

The results in Fig. 12 and Fig. 13 show that Docker Swarm outperformed all of the Kubernetes CPU target configurations and DC/OS. Considering a comparison between the least performed Docker Swarm (Swarm with 1 replica) with a total request received of 365,452 requests and any of other K8S scaling replicas (80%, 60%, 40%, and 20%) with a total request received of 325,544, 338,667, 340,800, and 344,825 requests respectively. Docker Swarm's total received requests outperformed K8S by 11.55%, 7.60%, 6.98%, and 5.80% according to 80, 60, 40, and 20 CPU target utilization percentages respectively. In DC/OS, the total received requests in aspects of handling workloads results according to 1, 3, and 5 scaled instances with the total received requests of 317,185, 328,811, and 333,530 respectively. Nevertheless, Docker Swarm still outperformed them according to instances of 1, 3, and 5 by 14.14%, 10.55%, and 9.13% respectively.

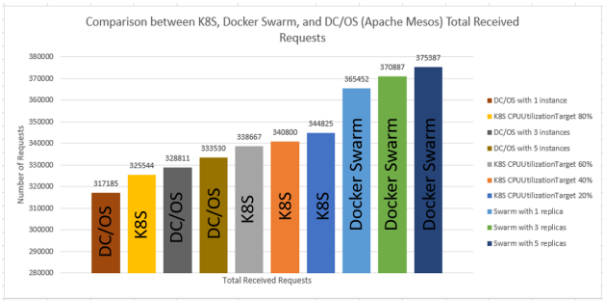

Figure 12. Comparison between K8S and Docker Swarm total received requests.

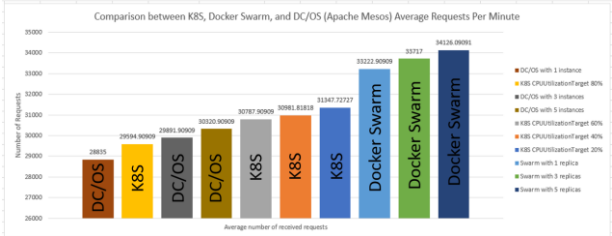

Figure 13. Comparison between K8S and Docker Swarm average requests per minute.

Despite the based-line of all the settings for K8S, Swarm, and Mesos on deployment being the same which includes Nginx docker image, the compute engine instance, and network detail, the lowest performance of Swarm (Swarm with 1 replica) still has better performance (In terms of handling a massive request concurrently with 400 threads) than K8S deployment server with the best performance (K8S CPU Utilization Target 20%) among its own orchestration service. This could prove that a single replicas instance deployment of Docker Swarm could handle more workload than any other HPA configuration from Kubernetes and DC/OS scaled instances. However, the big downside of Docker Swarm is the lack of functionality in auto-scaling. When it comes to scalability, Swarm only considers it enables scaling on demand through Swarm CLI. But in K8S, a one-in-all framework can comprise a complex system. It is complex because the cluster state utilizes a unified set of APIs (Application Programming Interfaces) that slugs container deployment and scaling.

E. Kubernetes, Docker Swarm, and Apache Mesos Features Comparison

1) Installation complexity

For K8S, learners with the introduction on how to deploy a containerized application, it is quite complex for starters. It is necessary to have enough amount of knowledge on container technology since it has a steeper learning curve when compared to Docker Swarm installation. Originally, K8S is designed to be developed containerized web applications in a large infrastructure. Moreover, K8S does not have a simple web UI to manage the cluster and needs to configure it via configuration files.

For Docker Swarm, it provides simplicity for installation. For learners who already know Docker containers and want to know how to deploy containers in a group for orchestration, Swarm is a great choice. Also, managing a Swarm cluster is not complex at all since Swarm is not designed to be used in a very large infrastructure.

For Apache Mesos or DC/OS, in terms of installation complexities, DC/OS is the most complex one to set up a cluster since it requires external frameworks like Marathon or Terraforms before it can even begin functioning as a container orchestration tool. To unlock the auto-scaling features in DC/OS, Marathon is required to be able to scale up to thousands or even ten thousand agents (nodes or servers). In contrast, DC/OS is quite flexible to set up unlike K8S and Docker Swarm but it comes with a high level of complexity. DC/OS is originally designed for large organizations with large infrastructure deployment.

2) Scalability

For K8S, the auto-scaling feature is provided which is integrated with K8S services itself which are called HPA (Horizontal Pods Autoscaler). The orchestration comes from containers in pods since several containers can be deployed and scheduled together as a group from a service. K8S provides the ability to schedule groups of containers even though the applications are complex. For scaling it is quite straight forward and many large organizations with a large infrastructure use K8S.

For Docker Swarm, the service itself does not provide any auto-scaling capability. Therefore, the users need to put a large amount of effort to make Swarm be able to scale automatically. Auto-scaling only supports scaling up or down via commands only which in a technical perspective is not practical to manually scale container up or down. But overall, Docker Swarm allows users to deploy an application faster and easier in terms of scaling.

Apache Mesos or DC/OS provides the largest scalability on container orchestration which stands out when compared to K8S and Docker Swarm. Mesos cluster is known to support the performance of scalability which could scale up to 10,000 agents using Marathon as a framework scale while a K8S cluster can scale up to a maximum of around 5,000 nodes. This scalability makes Mesos the container orchestration tool alternative for large organizations with a large deployment of containerized applications or even Virtual Machines to maintain massive clusters. Also, it is important to be noted that Mesos can even run a K8S service as a framework on top of its own cluster deployment.

3) Monitoring

K8S has its own built-in monitoring and supports integration with third-party monitoring tools [15]. Also, there are plenty of monitoring solutions such as Prometheus which is a native monitoring tool for K8S. Another popular tool is Grafana. it provides simplicity to set up on K8S and there are numerous deployment specifications that include a Grafana container by default and consists of a K8S monitoring dashboard for Grafana available for use. Moreover, there are others monitoring tools such as kubewatch, kube-ops-view, and kube-state-metrics.

Docker Swarm does not provide any built-in monitoring solution and requires third-party applications to be able to monitor the cluster.

Apache Mesos monitoring tools are quite difficult to find since Mesos orchestration is relatively new when compared to K8S and Swarm. However, monitoring a DC/OS cluster is available through using Marathon metrics. Also, to diagnose and scans all the cluster components, the data can be queried and aggregated through available APIs which is quite complex when compared to K8S available monitoring tools.

4) Integration tools

For K8S, it provides flexibility to integrate with other open sources tools such as monitoring (cAdvisor), security (Twistlock, Falco, and Aqua) [7], and deployment tools (Helm, Apollo, and Kubespray).

For Docker Swarm, the dependency on Docker creates little interest for developers and there are just a few integration tools for Swarm. For instance, an open-source plugin that automates and simplifies the script-building process is called Gradle. Moreover, there is a configuration management and deployment automation tool created by RedHat.

For Apache Mesos or DC/OS, it provides a lot of integration tools since Mesos tends to have a preference

for tools developed by Apache itself and Mesosphere such as Marathon. On top of that, the provided tools for Mesos direct towards the use of specialty tools and there is a lot of external or internal frameworks which included K8S as a framework itself for Mesos.

F. Scenarios of When to Use the Most Efficient Container Orchestration Tools

For scenarios that need an entry-level solution for smaller projects and testing purposes of working on a small project that requires the deployment of a few nodes, Docker Swarm is ideal especially if the users are already familiar with the Docker Container platform. Also, it provides simplicity on deployment, and the learning curve is quite low. Docker Swarm is a lightweight, easy-to-use orchestration tool with limited offerings compared to Kubernetes. In contrast, Kubernetes is complex but powerful and provides self-healing, auto-scaling capabilities out of the box. As shown in Table IV, from the experimental results, Docker Swarm has the best capability to handle requests which could handle the average total received requests of 370,375. In contrast, Docker Swarm does not have any auto-scaling capability and to able auto-scaling features, the user needs to write a script itself of when to scale up or down which is very overwhelming.

For scenarios of working on a massive project which involved several data centers where multiple complicated applications are needed to be set up and configured. Apache Mesos justifies the use of a high-level complexity platform since it offers an industrial-grade solution for very large clusters, but due to its complexity, it's generally only relevant for big corporations. Moreover, Mesos is a great alternative if multiple Kubernetes clusters are required within the data center and the intuitive architectural design of Mesos provides good options when it comes to handling legacy systems and large-scale clustered environments via its DC/OS. As shown in Table IV, from the experimental results, Apache Mesos has the lowest capability to handle requests with an average total received requests of 326,509. However, DC/OS is very flexible in terms of importing other frameworks to be used in the cluster which could increase the performance of the cluster according to the user specification. To be mentioned, DC/OS could use Kubernetes as a running framework in the cluster. So, Apache Mesos is focusing more terms on flexible deployment and large infrastructure rather than handling massive requests in a short period of time with its own default agent (Server).

TABLE IV. COMPARISON BETWEEN K8S AND DOCKER SWARM HANDLING REQUESTS AND TOTAL REQUESTS

	Handling Average Requests per minute (Mean)	**Handling total requests per interval of 11 minutes (Mean)**
Kubernetes	30,677.25	337,459.8
Docker Swarm	33,688.67	370,575.3
Apache Mesos	29,682.60	326,509.0

For scenarios of working on a project that requires an enterprise-level platform capable of running and managing thousands of containerized applications or services. Kubernetes is the best alternative choice and also it provides a powerful self-healing and auto-scaling which brings stability to the clusters. Moreover, the auto-scaling feature from the service itself is provided which is integrated with Kubernetes services itself which are called Horizontal Pods Autoscaler (HPA). So, it is easier to manage the minimum and maximum pods within the cluster and car write an HPA script to manage how you want to scale and configure a threshold of CPU utilization for when to scale up or scale down the pod. As shown in Table IV, from the experimental results, the performance of handling requests for Kubernetes was in mid-tier with the total average received requests of 337,459.8. However, Kubernetes provides more stability and reliable cluster when compared to Docker Swarm since the auto-scaling for Kubernetes is integrated with its own service by could configure in the deployment YAML file and could set minimum CPU Utilization and maximum CPU utilization when scaling their pods (server).

In-depth analyses of the Apache Mesos, Docker Swarm, and Kubernetes orchestration services are conducted in this study. The user's complexity of deploying apps affects whether orchestration solutions are considered to have better performance when comparing these three container orchestration services. Due to its auto-scaling characteristics, Kubernetes performed better than Docker Swarm in terms of scalability. In particular, the HPA configuration on Horizontal Pods Autoscaling while in Docker Swarm needs to scale manually via Swarm CLI. In contrast, Apache Mesos provides large infrastructure deployment which could deploy Kubernetes on top of the cluster itself.

The obtained results illustrate that in the aspect of handling a request, Docker Swarm performed better in terms of handling a massive request by Docker Swarm outperforming all of the Kubernetes CPU target configurations and Apache Mesos. Considering a comparison between least performed Docker Swarm (Swarm with 1 replica) with a total request received of 365,452 requests and any of other Kubernetes scaling replicas (80%, 60%, 40%, and 20%) with a total request received of 325,544, 338,677, 340,800, and 344,825 requests respectively. Docker Swarm total received requests outperformed Kubernetes by 11.55%, 7.60%, 6.98%, and 5.80% according to 80, 60, 40, and 20 of CPU target utilization percentages respectively. This could prove that a single replicas instance deployment of Docker Swarm could handle more workload than any other HPA configuration from Kubernetes and any deployed agents from Apache Mesos. However, the big downside of Docker Swarm is the lack of functionality in auto-scaling. When it comes to scalability, Swarm only considers it enables scaling on demand through Swarm CLI.

The testing experiments were conducted by using JMeter to send requests to both server orchestrators by generated users (threads) that configured up to 400

threads by sending requests concurrently. To summarize, choosing Kubernetes or Docker Swarm depends on the requirements of the application. If the server application on production was deployed in Kubernetes, there was a low possibility for the server to crash or down due to Kubernetes having its own flexible autoscaling server system. In contrast, if the production deployment was deployed by using Docker Swarm, it was certain that Docker Swarm could handle more requests from users greater than Kubernetes. However, Docker Swarm needs to manually adjust the number of replicas and there is more possibility for the Docker Swarm server to crash or not run due to receiving too many excessive requests in a short period.

V. Conclusion

The container orchestration services Kubernetes, Docker Swarm, and Apache Mesos are examined in-depth in this study. The results are from simulating users to send requests to the server using the based-line setup of both orchestrators to see how both orchestrators may perform while handling requests.

The outcomes in Kubernetes HPA configuration showed that HPA configuration has an impact on how well the server manages a workload from external sources. Due to slowly scaling up the replicas to satisfy the requirements, the CPUUtilization-Target80% performed worse than Docker Swarm in terms of the number of average requests received and the total number of requests received. CPUUtilization-Target20% performed the best which is more than the average request of CPUUtilization-Target80%, CPUUtilization-Target60%, and CPUUtilization-Target40% by 5.57%, 1.8%, and 1.17% respectively.

In future work, the comparison could be applied to other orchestration services and could also be used to measure other aspects like networking, load balancing, and manual scaling.

Conflict of Interest

The authors declare no conflict of interest

Author Contributions

B. Purahong, J. Sithiyopasakul, and C. Benjangkaprasert conducted the research and wrote the paper. J. Sithiyopasakul, P. Sithiyopasakul, A. Lasakul, and C. Benjangkaprasert analyzed the data. All authors had approved the final version.

References

[1] S. Burroughs, *et al*., "Towards autoscaling with guarantees on Kubernetes clusters," in *Proc. Int. Conference on Autonomic Computing and Self-Organizing Systems Companion*, 2021, pp. 295–296.

[2] Ł. Wojciechowski, *et al*., "NetMARKS: Network metrics-aware Kubernetes scheduler powered by service mesh," in *Proc. Int. Conference on Computer Communications*, 2021, pp. 1–9.

[3] A. P. Ferreira and R. Sinnott, "A performance evaluation of containers running on managed Kubernetes services," in *Proc. Int. Conference on Cloud Computing Technology and Science*, 2019, pp. 199–208.

[4] J. Shah and D. Dubaria, "Building modern clouds: Using Docker, Kubernetes & Google cloud platform," in *Proc. Int. Annual Computing and Communication Workshop and Conference*, 2019, pp. 184–189.

[5] H. Kitahara, K. Gajananan, and Y. Watanabe, "Highly-scalable container integrity monitoring for large-scale Kubernetes cluster," in *Proc. IEEE Int. Conference on Big Data*, 2020, pp. 449–454.

[6] M. K. Abhishek, D. R. Rao, and K. Subrahmanyam, "Framework for containers orchestration to handle the scientific workloads using Kubernetes," *Journal of Computer Science*, vol. 18, no. 9, pp. 860–867, October 2022.

[7] S. R. Rizvi, A. Lubawy, J. Rattz, A. Cherry, B. Killough, and S. Gowda, "A novel architecture of Jupyterhub on Amazon elastic Kubernetes service for open data cube sandbox," in *Proc. IEEE Int. Geoscience and Remote Sensing Symposium*, 2020, pp. 3387–3390.

[8] J. N. Acharya and A. C. Suthar, "Docker container orchestration management: A Review," in *Proc. International Conference on Intelligent Vision and Computing*, 2022, pp. 140–153.

[9] V. K. Thakur, "A review paper on open-source container orchestration," *International Research Journal of Modernization in Engineering Technology and Science*, vol. 2, pp. 1008–1016, October 2020.

[10] B. S. Kim, S. H. Lee, Y. R. Lee, Y. H. Park, and J. Jeong, "Design and implementation of cloud Docker application architecture based on machine learning in container management for smart manufacturing," *Applied Sciences*, vol. 12, no. 13, July 2022.

[11] G. Rattihalli, "Exploring potential for resource request right-sizing via estimation and container migration in Apache Mesos," in *Proc. Int. Conference on Utility and Cloud Computing Companion*, 2018, pp. 59–64.

[12] H. Bairagi, U. Chourasiya, S. Silakari, P. Dixit, and S. Sharma, "A survey on efficient container orchestration tools and techniques in cloud environment," *International Journal of Scientific & Technology Research*, vol. 9, pp. 1425–1430, January 2020.

[13] A. Valantasis, N. Makris, and T. Korakis, "Orchestration software for resource constrained datacenters: An experimental evaluation," in *Proc. International Workshop on Performance Evaluation of Next Generation Virtualized Environments and Software-Defined Networks*, 2022, pp. 121–126.

[14] S. Kaiser, S. Haq, A. S. Tosun, and T. Korkmaz, "Container technologies for ARM architecture: A comprehensive survey of the state-of-the-art," *IEEE Access*, vol. 10, pp. 84853–84881, August 2022.

[15] R. K. Lenka, S. Mamgain, S. Kumar, and R. K. Barik, "Performance analysis of automated testing tools: JMeter and TestComplete," in *Proc. Int. Conference on Advances in Computing, Communication Control and Networking*, 2018, pp. 399–407.