Volume- 6
Issue- I
June-2021

Rajarata University Journal

ISSN: 2362-0080
www.ruj.ac.lk/journals/

Category: Research Article

DevOps Engine Modeling for Microservices Software Applications on Docker Containers

[*1]Kithulwatta,WMCJT, [2]Wickramaarachchi Wiraj Udara,[3]Warnajith Nalin

[*1] *Faculty of Graduate Studies, Sabaragamuwa University of Sri Lanka, Belihuloya, Sri Lanka*

[2] *Department of Computing, Faculty of Applied Sciences, Rajarata University of Sri Lanka, Mihintale, Sri Lanka*

[3] *Software Engineering Teaching Unit, Faculty of Science, University of Kelaniya, Dalugama, Sri Lanka*

ARTICLE DETAILS

Article History
Published Online: 30 June, 2021

Keywords
Container approach, DevOps engineering, Distributed computing

[*]Corresponding Author
Email:chiranthajtk@gmail.com

ABSTRACT

Existing DevOps infrastructures are: difficult to archive data, difficult to archive large virtual machines, large payments for cloud services and difficult to scale the infrastructure. By applying more convenient DevOps practices, an agile DevOps engine was designed. The proposed DevOps engine was deployed on the Docker container management platform and used separate Docker containers to deploy software applications and services to obtain the enterprise ready infrastructure by applying microservices architecture. The engine was evaluated with the same infrastructure in a cloud environment. According to the identified data and experimental results of the research study, the engine was performed fast execution speed. The host computer resources were utilized for the proposed engine. As well, container resource sharing was examined when shrinking and stretching containers. When transferring data within containers, the engine was secured since data were shared on directory paths. Furthermore, the engine performed more backup, portable and easy migration features. Advanced software engineering preliminaries and better Docker orchestration tools were applied for the proposed solution. The study found out that fast and light-weighted Docker containers help to ship the microservices software application in the enterprise-ready environment by utilizing host computer resources as the significance of the study. Cloud hosted Docker containers, three different software applications and a database management system container was used for the experiment. Accordingly, the study investigated the launch of a stable DevOps engine with Docker.

1. Introduction

Software and the internet have changed the culture of all kinds of industries such as manufacturing, apparel, education, health, government etc. into the digital world. Before more years ago software and the internet were a very small part of a business and were merely supported to the industry. But, within the current industry platforms, software and internet are an integrated component of the businesses and it plays a major role: for example, banking, bill payments, purchasing and factory automations. The usage of software and the internet makes an easy platform for companies to interact with customers by providing a huge amount of services. Other than that companies use software to make more values for their day to day operations like supply chain management, logistics, communication and operation. Mainly manufacturing and production companies are moving their design, build, deliver and all other manufacturing stuff into an automated way.

DevOps engineers play a massive role in the Information Technology industry to deliver the organizational software applications as a rapid service to the end users and clients. To deliver the final software applications, have to apply more tools, best practices and more ideologies to keep the standard of the delivery. DevOps engineers should have to engage in enhancing the product in a fast way rather than traditional software delivery and management process. If the DevOps enables to enhance the delivery process, it measures the efficiency of the team by market and customers. DevOps team needs to follow a set of practices and tools to accomplish their tasks without any delay. In the traditional way of DevOps, they can follow the manual process for every deployment process. But,

© RUJ 2021, All Rights Reserved

having a high velocity for the system application delivery is an essential factor. By driving with more speed, the process enables one to adapt to the changing market in a better way and expand the ideas efficiently with the business results. With the usage of some practices in DevOps, the protocol makes a quicker platform to enhance the updates of software version releases.

To develop software products faster, increasing the frequency and pace of releases are needed. By increasing the bug fixing and new features adding to the platform faster, it enables to meet customer goals in a more advanced way and to meet the competitive advantage. To give a positive experience to the end-users, maintaining and ensuring the application quality and system infrastructure is needed. All changes in the system must be functional and safe. Monitoring the system performance in real time is required for DevOps engineers and system administrators while changing the infrastructure.

To launch most computer system infrastructures, virtual machines were applied with the concept of virtualization. Virtualization is an old concept. Figure 1 presents the architecture of the virtualization [14].

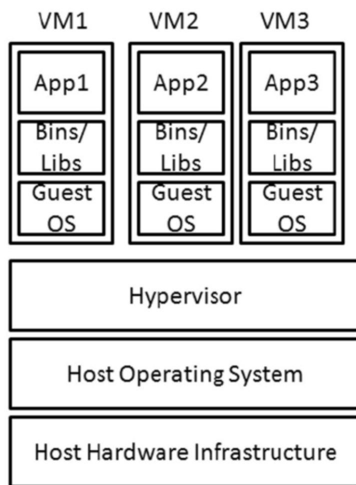

Figure 1: Virtualization architecture [14]

Hypervisor is the most important component for virtualization. Hypervisor manages the infrastructure to run multiple compute hosts as virtual machine instances with a full operating system. Since each virtual instance consists of the full package of the operating system, the virtualization carries higher overhead to the infrastructure.

The concept, “containerization” was introduced to the computing discipline to bring an alternative for virtualization. The containerization was designed to bring a lightweight infrastructure. Figure 2 presents the architecture for the containerization [14]. The container engine is a main component of the container architecture and on top of the container engine, separate containers can be deployed. Each container consists of full packages of dependencies, binaries and libraries which are essentially needed to run the software applications and services with the container.

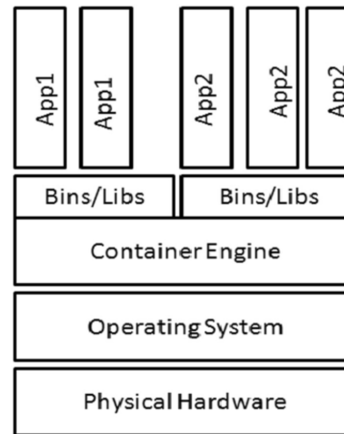

Figure 2: Container architecture [14]

Within the practitioner most container vendors are available. Linux containers, Rkt and Docker are some of them. Among all of them, Docker is the most trending container classification.

Docker is providing a platform to automate the application when they are deployed into containers [21]. After making any deployment or any execution on top of the Docker container, it makes an extra layer of deployment on top of the engine. Before deploying the source-code or application binaries into the production environment, it can test with an easy platform on Docker containers. Lots of comparisons have been done with containers and virtual machines by a lot of scholars. A container consists of executable application/s. All fundamental necessary software dependencies need to run a container. Containers are using Linux kernel mechanisms to allocate resources [26]. Engineers can allocate resources for the containers like network configurations, CPU and memory at the time of container creation. The “Density” is the next advantage of the Docker containers [11]. Docker is not combined with hypervisor and Docker uses all available resources of the host operating system. It causes the containers to run on a single host, with the virtual machines. Docker containers present higher performance with higher density.

Currently below problems (P_i) were identified within the domain of DevOps when using the traditional DevOps practices.

P_1= Difficulty of archiving large computer systems infrastructure

P_2= Difficulty to update, maintain, migrate and scale the infrastructure

P_3= Difficulty of using hardware & software in an efficient way

P_4= High cost scaling

Accordingly, the main objectives (OB_i) of the study are:

OB_1 = To develop conceptual and technically containerized DevOps engine in order to orchestrate the containers architecture user friendly,

OB_2 = To present long time data consistency approach

OB_3 = To create an agile DevOps platform in order to make it easy to apply with DevOps practices.

There were and there are a lot of discussions made on online forums to create containerized computer systems infrastructure platforms. Those forums presented only the experience of the authors without any theoretical proof.

2. Material and Methods

To overcome the identified problems and to achieve the overall objectives of the research study, an enterprise-ready DevOps engine was established. The proposed DevOps engine (named as Case 01) was established by using Docker container management platform on top of a Linux x86_64 Canonical Ubuntu 18.04.2 Long Term Support (LTS) operating system. Docker version 19.03.9 was used for the experiment. Six separated Docker containers were used to launch the infrastructure. For the deployment purposes and for the establishment of an enterprise-ready platform, fully functional microservices applications were used. Those applications were developed with most widely used and using technologies in the current day. Furthermore, the other selected software application services were mostly popular technologies in enterprise-environment.

By using software application reusability as a software engineering preliminary, Docker trusted images were used from the Docker Hub [8] [24]. (*Docker Hub is the public repository for Docker images and application templates*). Three different microservices software applications were used to get the advantage of microservices software architecture within this research study. The selected software applications were developed by using three different software technologies. *Apache Tomcat* web server, *NGNIX* web server and Ubuntu: bionic (for *Springboot* application container) containers were used to deploy those three software applications. To facilitate the database management system services, *MySQL* database management system containers were used. For the establishment of continuous integration and continuous delivery (CI/CD), *Jenkins* container was launched. To support each software application build, one single access point was created. Other than that, the access point was for organizing all binary resources including property libraries and remote artifacts. That single access point is, in-house *Jfrog Artifactory* container. Docker containers were used to provide an isolated environment for the microservices applications.

All Docker containers were mapped with "*docker volumes"* to create a long time data persistance approach. All key data and application path of each container were mapped on the path of */var/lib/docker/volume/* [24] on the host computer infrastructure. Figure 3 presents the pictorial way of architected DevOps engine.

All launched Docker containers are following distributed computing architecture. To make the communication among each container, Internet Protocols (IPs) were defined. As well, the internal Docker network was defined. The internal network was with 172.17.0.0/16 as the subnet and 172.17.0.1 as the gateway. For the internal and external data transactions, container ports and host ports were mapped with each container. Table 1 has presented the internal IP, container port and host port mapping with each container.

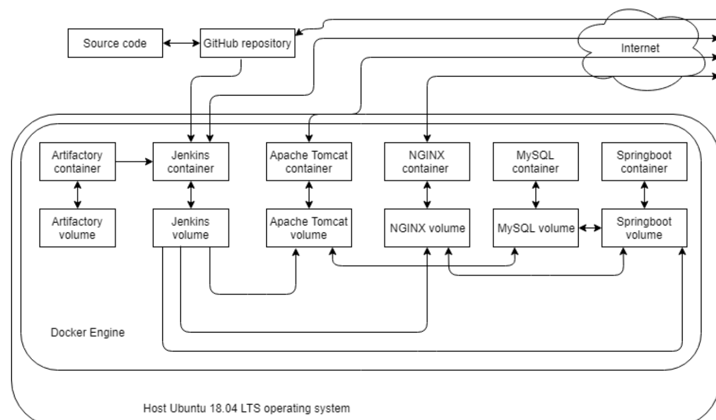

Figure 3: Proposed DevOps engine on Docker

© RUJ 2020, All Rights Reserved

Table 1: Container IP & port mapping with host ports

Container name	Internal IP	Container port	Host Port
***Artifactory* container**	172.17.0.2	8081	9090
***Jenkins* container**	172.17.0.3	8080	9091
***MySQL* container**	172.17.0.4	3306	9092
***Apache Tomcat* container**	172.17.0.5	8082	9093
***Springboot* container**	172.17.0.6	8080	9094
***NGNIX* container**	172.17.0.7	80	9095

To access each service from outside of the platform, "*host IP:host-port*" port mapping was needed to use. To access each service within the Docker platform, "*internal IP:container port*" port mapping was needed to use.

After establishing the stable Docker platform, all Docker containers were archived as images inside the local Docker registry and converted to ***.tar*** file format within the host computer infrastructure. For the monitoring and administrative purposes of the Docker platform, *portainer.io* tool was used instead of the traditional command line interface.

To evaluate the proposed Docker engine, the same architecture was deployed in a virtual machine on a cloud environment. The corresponding virtual machine based infrastructure is named as Case 02. Case 02 was also launched within an internal cloud network.

For the Case 02, each virtual machine was with 2 virtual CPUs. 15GB memory allocation was for each virtual machine and network bandwidth was 1Gbps. As well, for the proposed Docker engine, the host computer sever configurations are the same for above configurations of Case 02.

3. Results and Discussion

This research study was conducted to architect an enterprise-ready DevOps engine and evaluate the engine against traditional and existing common infrastructures. By considering the host computer resource usage & utilization, applied software engineering preliminaries, data transferring approach & network gap for the data transferring, infrastructure backup & migration, Docker platform monitoring were evaluated for the proposed engine.

Furthermore, to get essential decisions for the DevOps platform, core measurements were identified within this research study. *[For the ease of presentation results, the following abbreviations were used for Docker containers namely CPU % and MEM % (the percentage of the host's CPU and memory the container is using), MEM USAGE /LIMIT (the total memory the container is using, and the total amount of memory it is allowed to use), NET I/O (the amount of data the container has sent and received over its network interface), BLOCK I/O (the amount of data the container has read to and written from block devices on the host) and PIDs (the number of processes or threads the container has created)]* [7].

3.1. Resource usage & utilization

To analyze the proposed system infrastructure, the engine was experimented over the same infrastructure on virtual machines on the cloud service (Case 02). Restarting time was measured in each container against the particular virtual machine. Accordingly, the below table 2 presented the data obtained by calculating the mean-restart time of each container-virtual machine paradigm for ten times. First column of table 2 presents the container/cloud instance name. Last two columns are presenting the mean restart time for each container and cloud instances. Mean restart time was measured in seconds (s).

Table 2: Restart time consuming for containers and cloud virtual machines

Service name	Time-consuming for restart (s-seconds)	
	Container on Docker	Cloud insatnce
***MySQL* service**	3.25	17.26
***Artifactory* service**	3.45	21.21
***Apache Tomcat* servcie**	4.26	23.14
***NGNIX* service**	3.15	15.37
***Springboot* service**	4.49	21.22
***Jenkins* service**	6.05	31.47

According to Table 2, the performance of mean-restarting time was presented in a 1:5 (approximately) ratio between containers and virtual machines in the cloud for each service. This depicts, approximately 83% of performance increment is

© RUJ 2020, All Rights Reserved

presented in containerized engine approach than cloud instances.

According to container resource usage of the host, below statistical data were calculated for the *Jenkins* container. Table 3 depicts the data which has been collected when the *Jenkins* container was in normal up and running state. While executing 51 PIDs, the container consumed 13.86% mean memory usage and 1.84% mean CPU from the host.

Table 3: *Jenkins* container resource usage

Container Name	Container for *Jenkins*
Container ID	b4d6ba6100c3
CPU %	1.84
Memory usage/ Limit	2.035GiB / 14.68GiB
MEM %	13.86
Net I/O	1.09GB / 2.28GB
Block I/O	4.71GB / 1.55GB
PIDs	51

When the *Jenkins* container makes a software deployment, it performed 38.21% mean CPU usage from the host. But sometimes CPU usage was more than 100%. The proposed engine was launched on a multi-core operating system and the host was parallelized to all processes with many cores to get the benefit of the containerized approach. Hence most of the time, the *Jenkins* container consumed a minimum number of host resources for the normal execution. When the *Jenkins* container makes a deployment, it was consumed high performance and the container was scaled(stretched). Hence the container used the maximum resource stream. After the deployment, the container was shrunk and the container became normal as mentioned in above table 3.

Lot of literature had presented that isolated an environment is most suitable for microservices approach. Hence both two cases (Case 01 – the proposed containerized engine & Case 02) were launched to maintain the isolated environment. Below table 4 has presented host computers' resource measurements. To generate the experimental results, the mean values for each metrics were calculated by considering 30 days of performance with a one-hour interval per day. Particularly those metrics are CPU utilization [Activity level from CPU. Expressed as a percentage of total time (busy and idle) versus idle time.] and memory utilization (Space currently in use. Measured by pages. Expressed as a percentage of used pages versus unused pages). In table 4, the first column was used to denote the particular case and the second column was to denote the name of the cloud instance. CPU utilization and memory utilization was calculated as a percentage.

According to the collected measurements for both Case 01 & 02: Case 01 had utilized host computer resources. According to the microservices architectural software applications and services, all applications and services were executed on top of the host in Case 01. The Case 01 has provided an isolated environment for each application by using Docker containers. The Case 02 has given an isolated environment for each by using different cloud instances. Ultimately, the Case 02 has wasted more computer resources. Hence the proposed approach has proved that a containerized approach is more suitable than Case 02 according to the selected ***use case*** on behalf of resource utilization for the research study.

Table 4: Host computer CPU & memory utilization for both cases

Cases	**Cloud Instance Name**	**CPU utilization (%)**	**Memory utilization (%)**
Case (1)	Docker host instance	47.862	51.468
Case (2)	Instance for NGNIX service	0.137	3.488
	Instance for springboot service	0.353	4.261
	Instance for Apache Tomcat service	0.484	5.972
	Instance for MySQL database	0.322	5.854
	Instance for Jenkins	0.297	3.183
	Instance for Artifactory service	0.195	3.682

The figure 2 demonstrates the CPU utilization of the host of Case 01 – proposed containerized DevOps engine. Figure 3 shows the CPU utilization for NGNIX host in Case 02 of only one host among the six hosts as shown in table 4. It is difficult to compare the overall performance of Case 01 and Case 02 since in Case 02 consisted with six cloud computer instances. Hence in this research study, Case 01 & Case 02 were not compared over the host resources. But resource utilization of Case 01 and resource wastage of Case 02 hosts were presented in below graphs.

© RUJ 2020, All Rights Reserved

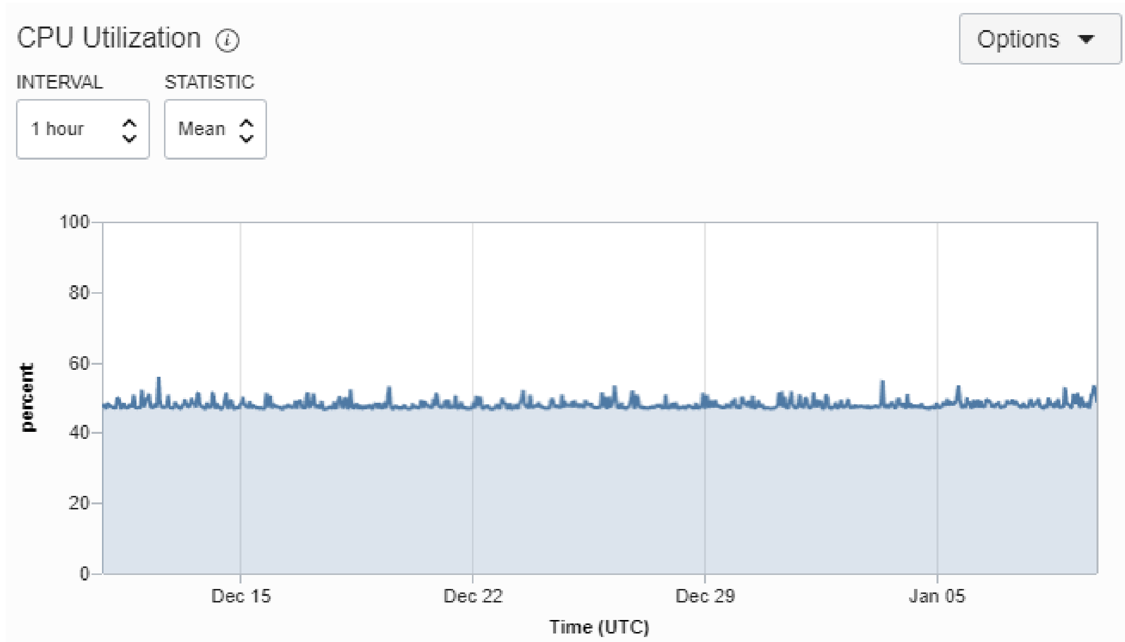

Figure 4: CPU utilization for Case 01 host

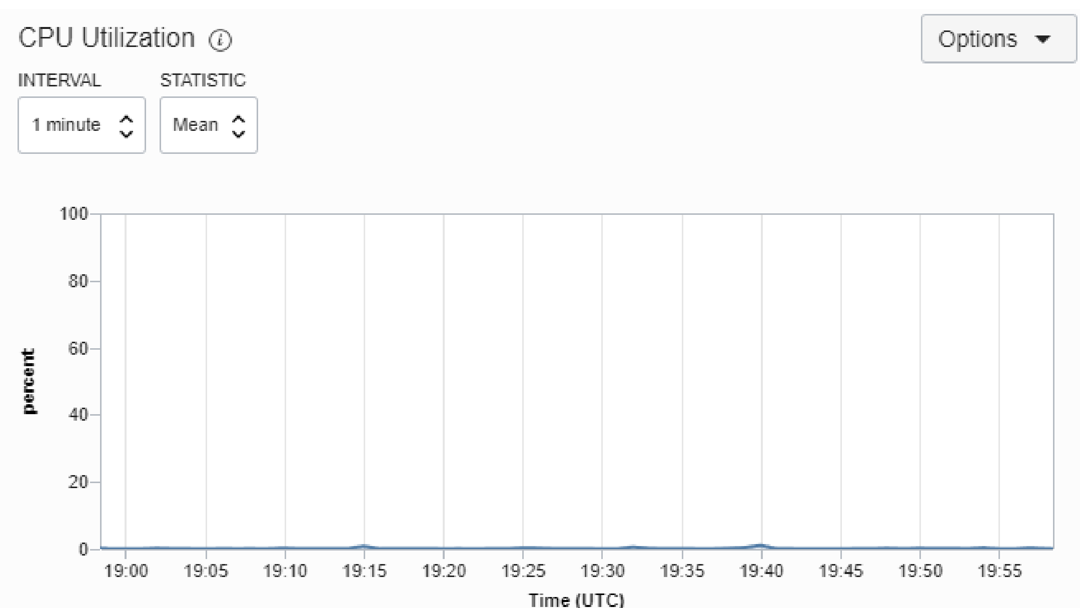

Figure 5: CPU utilization for NGNIX host in Case 02

Within the DevOps platform, it is essential to use the host computer hardware and software effectively and efficiently. Within the enterprise platform, large amounts of payments are made for the computer resources. Therefore, usage of host computer resources in an optimized way is needed.

3.2. Secured data transferring approach

In the proposed engine, all software artifacts were transferred from the *Jenkins* data volume to the application data volumes at each build/deployment of the software. The artifacts transferring happened inside the host computer infrastructure. Linux ***cp*** command was used to transfer the artifacts from one location to another location on the local Docker registry. Since Linux ***scp*** command was used in Case 02, the artifacts were transferred between two hosts (For the *Tomcat* service in both cases: embedded *Tomcat* container was used inside the *Jenkins* service). Data transmission happened only inside the host computer infrastructure at the proposed DevOps engine model. But in the cloud based instance approach, data comes out from the host and the data were transferred within the virtual cloud network as cloud based instance approach was made a time delay to transfer the data. In the Linux ***scp*** command approach, logging credentials of the hosts were needed to share within other hosts: username & passwords and SSH keys. Therefore,

© RUJ 2020, All Rights Reserved

Figure 6: CPU utilization for NGNIX host in Case 02

sharing those credentials between other hosts will be a threat. It depicts the security of the proposed engine is more than corresponding cloud based virtual machine approach or local virtual machine approach.

3.3. Network gap for data transferring

In both cases, each container and cloud instances were connected to an internal network. As discussed in immediate past sub section, all software build artifacts were transferred from *Jenkins* volumes to *NGNIX*, *Apache Tomcat* and *Springboot* container. Below table 5 has shown software artifacts deployment time for container and cloud instance environment.

For the time calculation, mean deployment time had been calculated in ten situations of software artifacts build deployments. Fresh deployments had not been considered since these happened only for the initial deployments and any exceptional situations.

According to the calculated mean deployment time, commonly cloud instances made some latency to make the deployment. For the *Apache Tomcat* service perspective, the deployment time gap between container platform and cloud instances platform is very few. In Case 02 (cloud instances), data transferring happened between separate hosts, but in Case 01, data transferring was inside the host. Therefore, the Case 02 had made some network gap between hosts rather than containers in Case 01.

Table 5: Deployment time calculation for Case 01 & 02

Service	Container	Cloud instance
NGNIX service	45 s	62 s
Springboot service	57 s	67 s
Apache Tomcat service	128 s	131 s

Fast iterations and continuous delivery are most the important criteria for the success of DevOps'. This is essentially to measure the time it takes to distribute the software and how often it is deployed. By tracking how often new code is deployed, the team and customer can track the development process. “Zero down time” is a most crucial task to maintain in the DevOps platforms to keep the software application availability continually. Even 5(five) seconds delay is also being affected on the customer satisfaction. Therefore, omitting or mitigating the system down time was more essential and the proposed engine presented minimal system down time at each software version deployments.

3.4. Infrastructure backup and migration

For a well-established infrastructure backup procedure, all Docker containers were archived as Docker images after making a stable Docker based infrastructure in the proposed engine. Archived Docker images were saved inside the local Docker registry. To take out the images from the local Docker registry, all images were converted to ***.tar*** format. The converted ***.tar*** formatted images were able to migrate from the host operating system to another host. As well those images are portable. Hence the proposed DevOps engine had provided better and advanced backup options. As well the proposed engine showed an easy migration capability.

Within Case 02, the infrastructure was established according to the traditional approach. Block volumes were attached to each cloud instance to archive key data of them. As well as boot volumes were attached for the booting purposes. To keep archived data in each block volume, payments were needed to be made for cloud service providers. Therefore, in Case 02 approach was made more budget consuming. But in the proposed engine(Case01) made free of charge to archive data in Docker volumes, but the approach was required to pay only for the block volumes of the host computer. Hence the proposed engine was performed with a less budget consuming approach. As it affects enhances the business profitability and reduction of unwanted payments is essential. Hence keeping Docker images is most suitable for an enterprise environment.

3.5. Application of Software Engineering preliminaries

To launch Docker containers, Docker trusted images were used from both the local Docker registry and open Docker community called Docker Hub. They are already configured with all packages which are needed to launch the container without installing manually. Since the software reusability is one of major preliminaries of the software evolution in the software engineering domain, particular containers were launched immediately as an easy platform to the DevOps activities. With the engagement of the reusability components in the proposed DevOps engine, the infrastructure designing and development were with both ***with reuse*** and ***for reuse*** (to launch the infrastructure, ***with reuse*** was applied and after launching the infrastructure by creating Docker images, ***for reuse*** was applied). The mounted data volumes were used to attach for several containers and therefore ***data reuse*** was applied. After migrating the platform, the new platform could implement the same configuration in the new platform. Hence

© RUJ 2020, All Rights Reserved

architectural reuse and ***design reuse*** was applied. After migrating the DevOps engine, any containers did not lose any executable code or application binaries. Therefore, ***program reuse*** was applied to the proposed DevOps engine [24].

3.6. Docker platform monitoring and administering

To monitor and administrate the Docker container platform, *portainer.io* tool was used. Excepting initial Docker configurations, all Docker activities can perform by using *portainer.io* tool: container creation, Docker network management, volume management, image creation, Docker container performance graphically monitoring, and etc. *Portainer.io* is a container based service, providing a better graphical user interface. To perform the internal functions of Docker containers, an embedded command line interface was provided in the *portainer.io* tool.

The table 6 was created using basic executing statistics of the *portainer.io* container. *Portainer.io* container executed 11 PIDs in the container. But the container consumed 0.14% mean CPU performance from the host computer CPU. As well as it consumed 0.1% mean memory performance from the host out of the total amount of the memory it is allowed to use. Hence, *portainer.io* totally consumed very low performance from the host. Since *portainer.io* container does not bring higher workload to the host. Therefore, portainer.io is a better tool to manage and administrate the Docker environment.

Table 6: Portainer.io container resource usage

Container Name	Container for *Portainer.io*
Container ID	a93c20a25dbb
CPU %	0.14
Memory usage/ Limit	15.23MiB / 14.68GiB
MEM %	0.1
Net I/O	22.8MB / 175MB
Block I/O	17.74 MB / 2.44GB
PIDs	11

Decision making is the most crucial task in the DevOps industry. To create an enterprise-ready platform, DevOps engineers have to look at more considerations. Those considerations are cost optimization for the platform, security of the platform, network gap of the distributed components, hardware & software effective usage, hardware & software usage efficiency. The main recognition is that the decision of the DevOps is dependent on the use case. The characteristics of the use case are technologies, tools, source code optimization, end-user requirements and client requirements. It depicts above proved results may be changed for another use case.

4. Conclusion

In this research study, an enterprise-ready system infrastructure was launched on top of the Docker container management service by using software applications and services which are commonly used in the enterprise-ready environment. The *portainer.io* was a better orchestration solution with a user-friendly, web-based interface for Docker containers to govern the Docker platform than the command-line interface. Therefore, portianer.io tool was recommended for better Docker management. After creating a stable Docker containerized DevOps engine, the author recommends to archive the containers as Docker images and ***.tar*** format. Those archived ***.tar*** formatted file can be used to extend the backup process of the engine and migrate the engine from one host platform to another platform (to any operating system platform). It recommends mounting a Docker volume before launching a Docker container, to archive key data, logs and applications on the host. For the long-time data persistence of the engine, one or more Docker volume attaching is recommended. It helps the Docker volumes to recover the most important data of the container although the container was crashed or destroyed.

According to the observed experimental test bed, the proposed containerized DevOps engine was proved theoretically and practically. Therefore, OB_1 objective was achieved within the study. The proposed engine was performed with advanced and higher approaches to archive data. Since it was with more data persistence protocol and it proves the OB_2. Since the platform was used most of DevOps and software engineering practices and preliminaries. Therefore, that was answered for the OB_3.

References

1. Al-Dhuraibi Y, Paraiso F, Djarallah N, Merle P. Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER. IEEE 10th International Conference on Cloud Computing. pp 472-479. Available from: https://doi.org/10.1109/CLOUD.2017.67
2. Apache Tomcat – Welcome! [Internet]. Available from: http://tomcat.apache.org/ [Accessed 26th February 2021]
3. Artifactory – Universal Artifact Repository Manager JFrog [Internet]. Available from:

https://jfrog.com/artifactory/ [Accessed 26th February 2021]

4. Casalicchio E, Perciballi V. Measuring Docker Performance: What a mess!!!*. Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion. pp 11-16. Available from:

   https://doi.org/10.1145/3053600.3053605

5. Chung MT, Quang-Hung N, Nguyen M, Thoai N. Using Docker in high performance computing applications. IEEE Sixth International Conference on Communications and Electronics (ICCE), 2016, pp.52-57, Available from: https://doi.org/10.1109/CCE.2016.7562612

6. Cito J, Gall HC. Using Docker containers to improve reproducibility in software engineering research. IEEE/ACM 38th International Conference on Software Engineering Comparison 2016. pp 906-907

7. docker stats [Internet]. Available from: https://docs.docker.com/engine/reference/commandline/stats/ [Accessed 25th February 2021]

8. Docker Hub [Internet]. Available from: https://hub.docker.com/ [Accessed 24th February 2021]

9. Jaramillo D, Nguyen CV, Smart R. Leveraging microservices architecture by using Docker technology. Southeast Con 2016. pp 1-5, Available from: https://doi.org/10.1109/SECON.2016.7506647

10. Jenkins [Internet]. Available from: https://www.jenkins.io/ [Accessed 26th February 2021] To face the upcoming challenges successfully in the DevOps environment, these microservices architectural solutions with Docker containers are adding more advantage to the DevOps industry with faster software delivery for the production.

11. Joy AM. Performance comparison between Linux containers and virtual machines. 2015 International Conference on Advances in Computer Engineering and Applications. Available from: https://doi.org/10.1109/ICACEA.2015.7164727

12. Kaewkasi C, Chuenmuneewong K. Improvement of container scheduling for Docker using Ant Colony Optimization. 9th International Conference on Knowledge and Smart Technology 2017. pp 254-259. Available from: https://doi.org/10.1109/KST.2017.7886112

13. Liu D, Zhao L. The research and implementation of cloud computing platform based on docker. 11th IEEE International Computer Conference on Wavelet Actiev Media Technology and Information Processing, 2014. pp475-478. Available from: https://doi.org/10.1109/ICCWAMTIP.2014.7073453

14. Martin JP, Kandasamy A, Chandrasekaran K. Exploring the support for high performance applications in the container runtime environment. Human-centric Computing and Information Sciences 2018. Available from: https://doi.org/10.1186/s13673-017-0124-3

15. MySQL [Internet]. Available from: https://www.mysql.com/ [Accessed 26th February 2021]

16. NGINX | High Performance Load balancer, Web Server, & Reverse Proxy [Internet]. Available from: https://www.ngnix.com/ [Accessed 26th February 2021]

17. Portainer | Open Source Container Management GUI for Kubernetes, Docker, Swam [Internet]. Available from: https://www.portainer.io/ [Accessed 26th February 2021]

18. Preeth EN, Mulerickal FJP, Paul B, Sastri Y. Evaluation of Docker containers based on hardware utilization. *I*nternational Conference on Control Communication & Computing India 2015. Pp 697-700, Available from: https://doi.org/10.1109/ICCC.2015.7432984

19. Rad BB, Bhatti HJ, Ahmadi M. An Introduction to Docker and Analysis of its Performance. International Journal of Computer Science and Network Security. VOL. 17 No.3, March 2017

20. Ruan B, Huang H, Wu S, Jin H. A performance Study of Containers in Cloud Environment. Advances in Services Computing. APSCC 216. Lecture Notes in Computer Science, vol 10065. Available from: https://doi.org/10.1007/978-3-319-49178-3_27

21. Russell B. KVM and docker LXC Benchmarking with OpenStack [Internet]. Available from: https://www.slideshare.net/BodenRussell/kvm-

    and-docker-lxc-benchmarking-with-openstack [Accessed 25th February 2021]

22. Tripathy P, Naik K. Software Evolution and Maintenance: A Practitioner's Approach. Published by John Wiley & Sons, Inc., Hoboken, New Jersy

23. Stubbs J, Moreira W, Dooley R. Distributed systems of microservices using Docker and Serfnode. Proceedings of the 7th International Workshop on Science gateways 2015. pp 34-39. Available from: https://doi.org/10.1109/IWSG.2015.16

24. Tripathy P, Naik K. Software Evolution and Maintenance: A Practitioner's Approach. Published by John Wiley & Sons, Inc., Hoboken, New Jersy

25. Use volumes [Internet]. Available from: https://docs.docker.com/storage/volumes/ [Accessed 25th February 2021]

26. Zhang Q, Liu L, Pu C, Dou Q, Wu L, et al. A Comparative Study of containers and Virtual machines in Big Data. 2018 IEEE 11th International Conferences on Cloud Computing. Available from: https://doi.org/10.1109/CLOUD.2018.00030

27. Zheng C, Thain D. Integrating containers into workflows: A case study using makeflow, work queue and Docker. Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing 2015. pp 31-38. Available from: https://doi.org/10.1145/2755979.2755984

© RUJ 2020, All Rights Reserved