Proceedings of the 2nd South Asia Conference on Multidisciplinary Research 2019 Vol. 2
Copyright © TIRDI / ISSN: 2659 – 2010 online

ENTERPRISE READY CONTAINERIZED AND MICROSERVICES ARCHITECTURAL DEVOPS ENGINE DESIGNING

W.M.C.J.T. Kithulwatta[1], D. Jayawickrama[2]

[1]*Software Engineering Teaching Unit, Faculty of Science, University of Kelaniya, Dalugama, Kelaniya, Sri Lanka*
[2]*mvv Information Technology, Level 9, East Tower, World Trade Center, Echelon Square, Colombo 01, Sri Lanka*

ABSTRACT Seamlessly software delivery and maintaining without any delay, is the major task of DevOps engineers in industrialization. In the traditional way, it is using bare metal hardware or cloud services to farm the computer system infrastructure. While using those modules, the main problems arising are, huge cloud service charges, disability to use infrastructure in the cross-platform, difficulty of infrastructure migration, system archiving problem, data persisting problems and smooth scalability issue. Main objectives of the research study are to create portable system infrastructure modules, to create technical and theoretical containerized DevOps engine, apply long-time data persisting approach to the enterprise applications and to apply high-velocity innovation to the computer systems infrastructure. The proposed DevOps engine was designed with the Docker container management system on top of the Linux operating system as the host. It was used Docker trusted images to deploy, isolated containers by using microservices architecture with advanced software engineering concepts with industrialized software applications. It was used enterprise-ready software applications and services on the proposed engine to validate the concept over the same configurations on the cloud service. With the usage of encapsulated components container approach, all internal data was secured on top of the host operating system. Due to the portability of Docker containers, it was easy to migrate the monolithic computer system to microservices architecture. By using fast Docker containers, it was facilitated to DevOps engineers on the engine to improve the scalability and security across the system infrastructure.

Keywords: *DevOps, Microservices, Containerization, Docker, Distributed computing*

1. Introduction

By reducing more complex computer system infrastructure, organizing the DevOps platform is one of the major tasks of DevOps engineers in the industrial approach. Involving with more advanced and high-velocity software application delivery mechanism is causing to increase the customer/ end-user satisfaction regarding the company.

Usually, in-house bare metal hardware or cloud services use to design the DevOps platform for the production-ready environment. In the DevOps platforms, DevOps engineers had to face several problems and issues: huge cloud service charges, disability to use infrastructure in the cross-platform, difficulty of infrastructure migration, system archiving problem, data persisting problems and smooth scalability issue. To create portable system infrastructure modules, to create technical and theoretical containerized DevOps engine, to apply long-time data persisting approach to the enterprise applications and to apply high-velocity innovation to the computer systems infrastructure: are the research objectives of this research study.

As mentioned in the [13], a software application or services reusability is a major preliminary of software system evolution. Since it is also applicable inside the DevOps environment to reduce the process and effort of the DevOps activities. According to the authors of [13], in the DevOps platform can reuse data, architecture, design and program under both concepts of *for-reuse* and *with-reuse*. Furthermore, reliability and maintainability can be enhanced in the DevOps platform with reusability. The authors of the [21] has mentioned that containers are very lightweight than virtual machines (VM). The same paper was presented that containers has consisted of fundamentally necessary software dependencies which needed to run by allocating all resources on top of a Linux kernel.

Omitting traditional and monolithic architectural software applications, microservices software applications has conceived in the industry. Microservices software applications were benefited to the enterprise community by providing major four characteristics [9]. Those benefits: organization around business capability, automated deployment, intelligence in the endpoints and decentralized languages and data. With the collaboration of those, microservices was provided with an easier platform to design, develop, test and release the services with great agility capacity. Furthermore, in the paper [9] has presented that microservices architecture was presented decentralized government and independent data management service. Microservices architecture has helped to omit the standardized for one single technology. Changing the technology for an application was very difficult on the monolithic architecture. The author of [12] has presented that the approach of microservices architecture was more suitable for the development tests and deployments.

Docker is a modern technology which was built for high-velocity innovations to deliver to the end-users. With enhancing developer productivity, deployment velocity, operational efficiency, infrastructure reduction and faster issue resolution [5]. According to the official website for the Docker [7] has presented that Docker volumes are the most preferred way to persist data in Docker containers and services. On the host operating system (OS), data has archived the particular data directory in the path of */val/lib/docker/volume_name/_data/*. As presented in the paper [9], the authors say that Docker is a good approach for microservices applications.

In term of distributed computing, it uses physically separated multiple computers by linking together via a network to accomplish a particular goal [19]. According to the [17], the authors have presented that the engagement of the container technology and Docker are making a profound impact on the distributed systems and cloud systems. Containers and microservices are a greater pair in the distributed computing systems.

2. Methods and materials

To design the enterprise-ready DevOps engine, a large enterprise-ready software application was used for experimental purposes. The software applications were developed using loosely coupled components/services by integrating microservices architecture. In the application, its own database (DB) were used for each service instead of sharing one DB with all services to get the benefit from the microservices. The software application was developed with *AngualrJS* for frontend application, Java-based *spring-boot* framework technology for the backend components and *MySQL* for DB services. *Jenkins* and *Jfrog artifactory* were used as services for deployments in DevOps activities. For the easiness and commonly used in industry, those software and services were selected. Using three different scenarios with the equal host OS resources, the research study was conducted. Case 01 is the proposed engine.

Case 01: To design the proposed DevOps engine, Docker container management platform was launched on top of a Linux x86_64 Ubuntu 18.04.2 LTS OS. Eight separate Docker containers were used by mounting Docker volumes for each container to archive key data directories [5]. On one host OS, distributed containers were launched. Figure 1 presents the architecture for the application and DB containers for the proposed DevOps engine.

To launch each container, Docker trusted images were used from the local Docker registry and Docker Hub: Ubuntu bionic containers for back-end services, *Apache HTTPD* container for front-end service, *MySQL* container for DBs, *Jenkins* container and *Jfrog Artifactory* container.

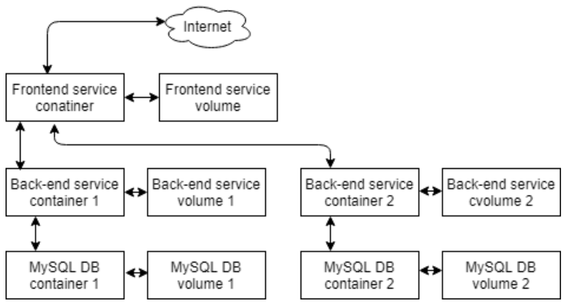
Internet
Frontend service conatiner
Frontend service volume
Back-end service container 1
Back-end service volume 1
Back-end service container 2
Back-end service cvolume 2
MySQL DB container 1
MySQL DB volume 1
MySQL DB container 2
MySQL DB volume 2

Figure 1: The basic architecture for the proposed engine

Figure 2 has shown that the artifacts delivery and sharing procedure within the containers from the *Jenkins* to application containers. Theoretically and according to the proposed methodology, the artifacts were delivered from the *Jenkins* volume to application container volumes.

For the data communication among the containers and link containers together, an internal Docker network was established in the local Docker engine with subnet 192.168.0.0/16 and the gateway as 192.168.0.1 instead of default Docker network. To open the containers to the outside world, containers were mapped with a host port. For the internal communication, containers were mapped with container ports. In the following Table1 presents the internet protocols (IP) and port mapping for each container.

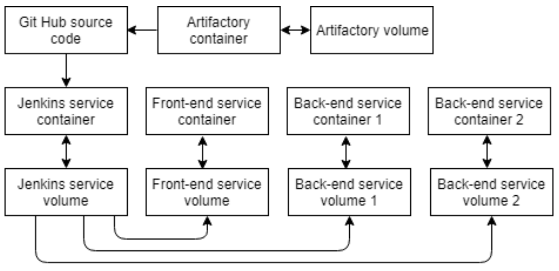
Git Hub source code
Artifactory container
Artifactory volume
Jenkins service container
Front-end service container
Back-end service container 1
Back-end service container 2
Jenkins service volume
Front-end service volume
Back-end service volume 1
Back-end service volume 2

Figure 2: Artifacts delivery mechanism

Container	Internal IP	Host port	Container port
MySQL container 1	192.168.0.1	13306	3306
MySQL container 1	192.168.0.2	23306	3306
Back-end service container 1	192.168.0.3	-	28088
Back-end service container 2	192.168.0.4	-	18088
Front-end service container	192.168.0.5	8000	80
Jenkins container	192.168.0.6	8090	8080
Artifactory container	192.168.0.7	8082	8081
Portainer container	192.168.0.8	9000	9000

Table 1: IPs and port mapping for the proposed engine

To access each service from the outside world, "*host IP:host-port*" was needed to use. To access the service within the Docker environment, *"internal IP: container-port"* was used. *Portainer* container was used to govern the Docker platform.

After created a stable Docker platform, all containers were archived as images in local Docker environment.

To evaluate the proposed engine, two corresponding cases were used as discussed below. For all cases: same software applications, DBs and other supporting services were used excepting the deployed platform and architecture of infrastructure.

Case 02: The platform was designed with three cloud instances according to the traditional distributed computing approach in the DevOps practices. In the traditional approach does not launch more separated instances for each service due to the large payments of the cloud service and to optimize the computer resource utilization. Payment optimization was a key task of DevOps engineers in the traditional approach.

Instance 1: continuous delivery and artifactory storing (as miscellaneous services: *Jenkins* & *Jfrog artifactory*)

Instance 2: all microservices software applications

Instance 3: DB services

The second instance was launched to deploy microservices software applications in separate directories. Instance 3 was facilitated with DB service to each component by keeping two databases. Artifacts delivery mechanism was the same in the Case 01 but in here, both *Jenkins* and *artifactory* services were launched in one instance.

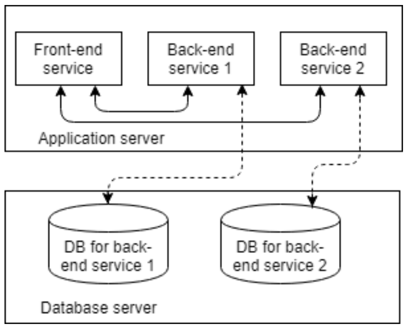

Figure 3: Used architecture for Case 02

Case 03: The platform was designed with seven separated cloud instances with the same configuration of Case 01 as distributed manner**.** Only differentiate is the deployed platform and no used third-party platform monitoring tool: in Case 01, the governing tool was the *portainer* tool as a container.

In both Case 02 & 03, for the network creation and monitoring the infrastructure, cloud service providers' facilities were used. To archive data of instances, cloud storages were used with payments.

To evaluate the proposed DevOps engine, all containers and cloud instances were archived in all cases. For the performance evaluation of the Docker platform, results of *docker stats* command and *portainer* tool was used. To evaluate the cloud machines in all cases, the default machine monitoring facility was used.

3. Results & discussion

For the evaluation of the proposed engine, the performances of the engine were evaluated by considering basic Docker container metrics as shown in Table 2., below. 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)[6].

By collecting the mean values for each metrics by using *docker stats* command on the host OS, the above Table 2, was created. According to Table 2, each container was executed using a minimum number of hardware and software resources while executing a large number of processes inside the containers. Sometimes, containers were presented more than 100% CPU usage since *docker stats* command presents the CPU usage as a percentage of a single CPU. Host OS for the proposed engine was a multi-core OS and it was parallelized the all the processes with many cores to get the benefit of the containerized approach. Within the Docker container approach, some containers were used extra resources of other containers to be scaled when the container was needed more hardware resources.

Furthermore, the proposed DevOps engine was evaluated against previously discussed Case 02 & Case 03. By considering host OSs performances for all 03 cases, Table 3 was created. 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. Particular metrics are CPU utilization [Activity level from CPU. Expressed as a percentage of total time (busy and idle) versus idle time.], memory utilization (Space currently in use. Measured by pages. Expressed as a percentage of used pages versus unused pages), disk read I/O (Activity level from I/O reads. Expressed as reads per second.), disk write I/O (Activity level from I/O writes. Expressed as writes per second.), disk read bytes (Read throughput. Expressed as bytes read per second.), disk write bytes (Write throughput. Expressed as bytes written per second), network receive bytes (Network receipt throughput. Expressed as bytes received per second.) and network transmit bytes (Network transmission throughput. Expressed as bytes transmitted per second.).

According to Table 03, in Case 01, the CPU utilization and memory utilization of Docker installed host computer instance was higher than all other instances in Case 02 & 03. But in Case 02 was performed more CPU and memory utilization against the Case 03. It depicts that, the host OS of Docker engine (in Case 01) uses the CPU and memory resources more efficiently and effectively than other cases by sharing all processors of Docker containers on the host OS. Due to Case 01 host, OS was executed more containers and processors than others. Without wasting the host OS resources in Case 01, it was utilized the host OS highly the Docker platform.

As mentioned in below Table 3, the Case 01 was consumed higher disk read I/O, disk write I/O, disk read bytes, disk write bytes, network receive bytes and network transmit bytes than others. Reason is: host instance was performed more containers with more workload. To perform high fast execution for the Docker engine, the host was needed to consume higher resources usage in Case 01 than other cases. By giving an isolated environment to the microservices software applications, the Docker platform was presented most suitable nature than separated cloud instances.

To transfer the files between distributed Docker containers, volumes were used since all key data/files were attached to Docker volumes. Linux *cp* command was used to send artifacts at each software version deployments to each application containers from the *Jenkins* container in Case 01. Due to, data artifacts transferring was happened between volumes on the host OS. At both Case 02 & 03, to send build artifacts from the *Jenkins* to each application instance (among distributed nodes), Linux *scp* command was used. The due reason was for that is the artifacts sending happened between two computers. In Linux *scp* command approach, credentials of the instances were needed to share with other instances: username-password or SSH key files of the instances. In Case 01 the data was shared without opening to the outside world. But in Case 02 & 03, the data could be opened to the outside world.

Container name	Container ID	CPU %	Memory usage/ Limit	MEM%	Net I/O	Block I/O	PIDs
Container for front-end	8aa9962bbc45	0.54	209.8MiB / 14.68BiB	1.4	329MB / 3.77MB	1.94MB / 1.36MB	89
Container for Back-end1	aef156e5eb1c	1.4	744MiB / 14.68GiB	4.95	226MB / 2.01MB	16.6MB / 1.5GB	38
Container for Back-end2	705a0f4d7d97	1.29	807.6MiB / 14.68GiB	5.37	969kB / 2.64MB	165MB / 173MB	103
Container for MySQL1	9b2296637611	2.45	1.64BiB / 14.68GiB	11.17	17.7MB / 176kB	46.7MB / 14.9MB	148
Container for MySQL2	fed771dda68a	3.07	806.9MiB / 14.68GiB	5.37	20.4MB / 6.43MB	14.2MB / 227MB	134
Container for Jenkins	b4d6ba6100c3	2.84	2.035GiB / 14.68GiB	13.86	1.09GB / 2.28GB	4.71GB / 1.55GB	51
Container for Artifactory	4b23863a0758	0.65	1.308GiB / 14.68GiB	8.9	742MB / 3.08GB	1.17GB / 12.7GB	74
Container for Portainer.io tool	a93c20a25dbb	0.14	15.23MiB / 14.68GiB	0.1	22.8MB / 175MB	17.74 MB / 2.44GB	11

Table 2: Docker container resource usage

Cases	Cloud Instance Name	CPU utilization (%)	Memory utilization (%)	Disk Read IO	Disk Write IO	Disk Read Bytes	Disk Write Bytes	Network Receive Bytes	Network Transmit Bytes
Case01	Docker Host instance	14.106	54.5	1.169M	16.907M	26.761G	291.307G	32.014G	39.165G
Case02	Application Instance	0.486	17.453	45.03K	1.158M	573.13M	16.683G	4.457G	3.671G
	DB Instance	0.427	14.181	96.353K	4.594M	829.335M	48.257G	74.330G	76.033G
	Miscellaneous Instance	0.71	6.646	87.436K	1.215M	1.226G	15.978G	20.544G	6.743G
Case03	Instance for Front end	0.129	3.356	30.489K	388.233K	266.042M	449.011M	958.294M	1.377G
	Instance for Backend 1	0.229	4.119	34.229K	229.762K	472.329M	603.873M	1.420G	1.383G
	Instance for Backend2	0.307	4.015	35.157K	303.117K	389.566M	785.418M	1.257G	1.567G
	Instance for MySQL01	0.291	6.115	41.121K	442.221K	498.338M	788.356M	1.56G	1.884G
	Instance for MySQL02	0.274	5.475	38.416K	376.556K	406.881M	677.854M	1.48G	1.854G
	Instance for Jenkins	0.266	3.066	31.844K	406.889K	376.674M	686.312M	1.69G	1.669G
	Instance for Atifactory	0.197	3.688	28.574K	364.637K	501.984M	853.112M	1.72G	1.828G

Table 3: Host computer resource usage and utilization

Without sharing public IPs, data transmission was happened with using internal IPs of containers or instances in all three cases. Hence among three approaches, the approach of Case 01 is more secure than others since all data transmission is happening inside the host OS.

After launched the Docker engine on top of the host OS, the *portainer* tool was launched on the Docker engine as a container. The tool was facilitated to manage all activities of the Docker platform with a web-based graphical user interface without using the command-line interface of the host OS. As shown in Table 2, *portainer* tool was consumed very low resources from the host OS. Hence it does not have any effect on other containers regarding the host computer resources.

Due to the usage of Docker templates for containers (e.g.: *MySQL* template of Docker, *Jenkins* templates of Docker and etc.) in local Docker registry (inside the host OS) and Docker Hub (open community), software reusability was applied to create the engine as an advanced software engineering practice to the DevOps platform. Due to those templates are already configured with all packages which are needed to launch the container without installing manually. Particular containers were launched immediately as an easy function in the DevOps platform. Since the software reusability is one of major preliminaries of the software evolution in the software engineering domain. 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*. Due to the mounted data volumes could attach to another container, *data reuse* was applied. After migrated the platform, the new platform could implement with the same configuration in the new platform. Hence *architectural reuse* and *design reuse* was applied. After migrated the DevOps engine any containers did not lose any executable code or processing tool. Therefore, *program reuse* was applied to the proposed DevOps engine.

If a container was destroyed or crashed, a new container was able to launch by attaching originally attached Docker volume. The reason was, all mounted data on the Docker volumes were protected on the host OS, without destroying even the container was destroyed. If there were more data in different directories, attaching more volumes was possible without attaching all directories to one volume to protect the data without any crash. If the host OS of Docker was crashed or volumes were destroyed directly, the mounted volumes were lost with data.

After created a stable platform on the Docker, all containers were archived as images, on the local Docker registry. Corresponding cloud instances were also able to archive as images/snapshots in both Case 02 & 03. They were able to use as base templates to create another container/instance on the platform, the image creation was used as container/instance backups on the platform. To extend the backup process furthermore, the containers were converted to *.tar* format. The converted format was able to migrate from the local Docker engine to the host OS. The converted format was able to migrate from the host OS to any another computer (any OS platform) easily as portable modules. After migrated the containers were able to launch on a new platform without losing any data with the same configurations. But archived instances were not able to convert any format or migrate from the platform to another one. It depicts that the proposed DevOps engine presents more backup options. The engine has easy & fast migration capabilities with portable modules.

With the applied theoretical concepts for the proposed DevOps engine, a technically feasible DevOps engine was able to develop and deploy. The engine was exhibited environmental independency due to the engine was able to deploy and migrate on any OS platform with more lightweight and portable modules. Since all those portable modules were able to migrate from the platform to another, without touching to basic configurations, both low coupling and high cohesion were embedded. Due to excepting large and complex configurations, easy understandability was with the proposed DevOps engine. According to the long-time data persistence of the proposed engine, the reliability of the engine was increased.

For the DevOps engineer's perspective, by architecting a DevOps engine for enterprise-ready microservices software applications most kinds of advantages were received. Since the easiness of the used tools to govern the architecture, the productivity was increased and the development of the architecture was accelerated. By investing less maintenance effort and time, the maintainability was improved. Since the more backup procedures of the proposed engine, process risk was reduced and reliability was increased by following more standards of the DevOps domain.

4. Conclusion & recommendations

As discussed above with the evidence, Docker containerized approach is an alternative for VMs/cloud instances with better performances. To get the benefit of the enterprise-ready microservices software applications, the distributed containerized engine provides the most suitable environment rather than cloud instances due to containers are with more virtualization benefits. Containers are with easy and automated scaling capability without touching to basic configurations of the infrastructure. To implement a high-velocity innovative DevOps engine with enterprise-ready microservices applications, Docker container approach is more benefited.

After created stable Docker containerized DevOps engine, the author recommends to archive the containers as Docker images and *.tar* format. Those archived *.tar* format can use to extend the backup process of the engine and migrate the engine from the host platform to another platform (to any OS platform). For the long-time data persistence of the engine, one or more Docker volume attaching is recommended before launching a container. Without using the traditional command-line interface, usage of a Docker monitoring tool is recommended (e.g.: *portainer*.io tool).

Acknowledgement

I thank Mr. D. Jayawickrama for the great the supervision for the research work and lecturer Mr. Wiraj Udara Wickramaarachchi for comments that greatly improved the manuscript.

References

[1] Anish Babu, S., Hareesh, M., Martin, J., Cherian, S. and Sastri, Y. (2014). System Performance Evaluation of Para Virtualization, Container Virtualization, and Full Virtualization Using Xen, OpenVZ, and XenServer. In: *2014 Fourth International Conference on Advances in Computing and Communications*. [Online] IEEE. Available at: https://doi.org/10.1109/ICACC.2014.66 [Accessed 22 Oct. 2019].

[2] Di Francesco, P., Malavolta, I. and Lago, P. (2017). Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. In: *2017 IEEE International Conference on Software Architecture (ICSA)*. [Online] IEEE. Available at: https://doi.org/10.1109/ICSA.2017.24 [Accessed 22 Oct. 2019].

[3] Digitalocean (2018) How to install and Use Docker on Ubuntu 18.04, Available at: https://www.digitalocean.com/ community /tutorials/ how-to-install-and-use-docker-on-ubuntu18-04

[4] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, G. Morgan, R. Ranjan, "A study on the evaluation of hpc microservices in containerized environment Concurrency and Computation", *Practice and Experience*, pp. 1-18, 2019.

[5] Docker (2019) Enterprise Application Container Platform | Docker, Available at https://www.docker.com/

[6] Docker Documentation. (2019). docker stats. [Online] Available at: https://docs.docker.com/engine/reference/commandline/stats/ [Accessed 24 Oct. 2019].

[7] Docker Documentation. (2019). Manage data in Docker. [Online] Available at: https://docs.docker.com/storage/

[8] Felter, W., Ferreira, A., Rajamony, R. and Rubio, J. (2015). An updated performance comparison of virtual machines and Linux containers. In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* [Online] IEEE. Available at: https://doi.org/10.1109/ISPASS.2015.7095802 [Accessed 22 Oct. 2019].

[9] Francesco, P., Malavolta, I. and Lago, P. (2017). Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. In: *2017 IEEE International Conference on Software Architecture (ICSA).* [Online] IEEE. Available at: https://doi.org/10.1109/ICSA.2017.24 [Accessed 22 Oct. 2019].

[10] Jha, D., Garg, S., Jayaraman, P., Buyya, R., Li, Z. and Ranjan, R. (2018). A Holistic Evaluation of Docker Containers for Interfering Microservices. In: *2018 IEEE International Conference on Services Computing (SCC).* [Online] IEEE. Available at: https://doi.org/10.1109/SCC.2018.00012 [Accessed 22 Oct. 2019].

[11] Joy, A. (2015). Performance comparison between Linux containers and virtual machines. In: *2015 International Conference on Advances in Computer Engineering and Applications*. [Online] Available at: https://doi.org/10.1109/ICACEA.2015.7164727 [Accessed 22 Oct. 2019].

[12] Medium. (2019). Introduction to Monolithic Architecture and MicroServices Architecture. [Online] Available at: https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63

[13] Mens, T. and Demeyer, S. (2010). *Software Evolution. Springer-Verlag.*

[14] Preeth, E., Mulerickal, F., Paul, B. and Sastri, Y. (2015). Evaluation of Docker containers based on hardware utilization. In: *2015 International Conference on Control Communication & Computing India (ICCC)*. [Online] IEEE. Available at: https://doi.org/10.1109/ICCC.2015.7432984 [Accessed 22 Oct. 2019].

[15] Russell, B. (2015). *Passive Benchmarking with docker LXC, KVM & OpenStack*.

[16]. Seo, K., Hwang, H., Moon, I., Kwon, O., Kim, B. (2014) Performance Comparison Analysis of Linux Container and Virtual Machine for Building Cloud, *Advanced Science and Technology Letters* 66:105-111.

[17] Stubbs, J., Moreira, W. and Dooley, R. (2015). Distributed Systems of Microservices Using Docker and Serfnode. In*: 2015 7th International Workshop on Science Gateways*. [Online] IEEE. Available at: https://doi.org/10.1109/IWSG.2015.16 [Accessed 23 Oct. 2019].

[18] Turnbull, J. (2014). *The Docker Book: Containerization is the new virtualization*.

[19] Tutorialspoint.com. (2019). Distributed Systems. [Online] Available at: https://www.tutorialspoint.com/Distributed-Systems [Accessed 23 Oct. 2019].

[20] Xavier, M., Neves, M. and Rose, C. (2014). A Performance Comparison of Container-Based Virtualization Systems for MapReduce Clusters. In: *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing.* [Online] IEEE. Available at: https://doi.org/10.1109/PDP.2014.78 [Accessed 22 Oct. 2019].

[21] Zhang, Q., Liu, L., Pu, C., Dou, Q., Wu, L. and Zhou, W. (2018). A Comparative Study of Containers and Virtual Machines in Big Data Environment. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD*). [Online] IEEE. Available at: https://doi.org/10.1109/CLOUD.2018.00030 [Accessed 22 Oct. 2019].