ARCHITECTING ADVANCED DEVOPS ENGINE WITH DOCKER BY USING MICROSERVICES FOR ENTERPRISE SOFTWARE APPLICATIONS

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

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

[2]mvv Information Technology, Elmo Tower, Mabola, Wattala, Sri Lanka

Keywords: Docker, Containerization, Microservices, DevOps

INTRODUCTION AND OBJECTIVES

In the industry approach, bare-metal hardware, virtual machines (VMs) or cloud infrastructure are using to launch enterprise-ready applications. At the time of using those platforms, the main problems that occur are the difficulty of scaling the infrastructure and maintain the infrastructure, long-time data persisting issues, the difficulty in archiving the systems and large payments on cloud services. To overcome the identified problems, an advanced DevOps engine was developed on the Docker container management system with microservices applications. The objectives of this study are to develop conceptual and technical DevOps engine module, to apply the containerization platform on Docker engine for enterprise-ready microservices applications and to make an agile DevOps platform. The proposed system was analyzed over the cloud infrastructure with the same configurations. Jha, et al. (2018) describes that Docker is a better approach for microservices applications but there are no existing researches for microservices architecture for the enterprise-ready platform with Docker.

RESEARCH METHODS

The proposed engine was deployed with the Docker container management system on top of the Ubuntu 18.04 LTS host. To design the engine, microservices architectural enterprise-ready software applications were used. The applications were deployed by using Apache Tomcat and NGNIX as web-server containers. For database services, MySQL container and deployment purposes, Jenkins and Aartifactory containers were used. Above discussed each software application and software service was deployed on separated Docker containers by using Docker trusted images from local Docker registry. All container was mounted with a Docker volume to archive most key data in each container for long time data persistency [1]. All data, applications and log directories of each container were attached to particular data volume on the Docker. For source code controlling it was used GitHub repository. Each Docker volume attached with the path / var/lib/docker/volume/ on the host operating system (OS) to data persistence and archive. Periodically each Docker containers were archived on both Docker local registry and converted them into portable modules. Meanwhile, cloud instances (CIs) were also archived to images.

Internal architecture for the proposed engine is shown in below Figure 1.

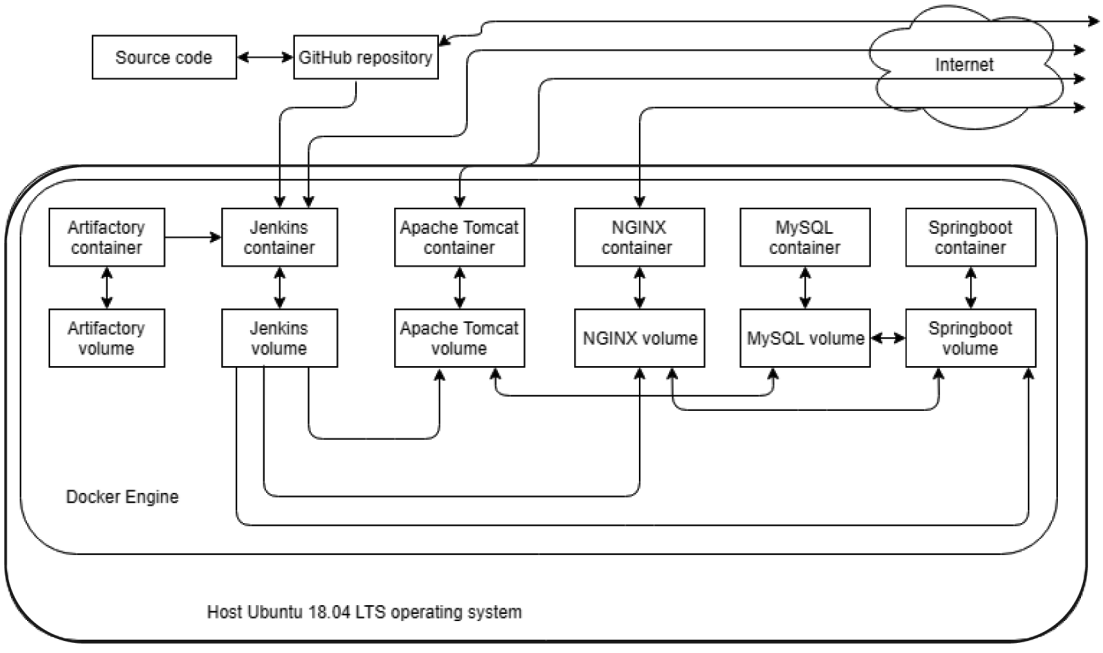

Figure 1: Proposed DevOps engine architecture on Docker

All the containers were communicated through an internal Docker network. Container ports and host ports were used for internal and external communications respectively. To compare the features of the proposed engine, the same configuration was deployed on cloud infrastructure. To orchestrate the proposed engine, portainer.io tool was used instead of the command-line interface to create an easy platform.

RESULTS AND DISCUSSION

The data in the containers were communicated through each volume inside the host OS. The data which was not in the mounted volumes, were transferred through internal IP addresses of containers and internal container ports. In the proposed engine, all data transmission happened inside the local Docker environment without opening to outside. Therefore, data transmission was more secured in the proposed engine. Host operating system IP address and host port of particular container were used to access the data/service of containers from outside world.

To analyze the proposed system infrastructure, the engine has experimented over the same infrastructure on the CI. Restarting time was measured in each container against to the particular CI as presented in below Table 1., by calculating the mean-restart time of each container-CI paradigm for ten times.

Table 1: Time to restart containers and VMs

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

According to the Table 1, the performance of mean-restarting time was presented 1:5 ratio between containers and CIs for each service. This depicts, approximately 83% of performance increment in presented containerized engine approach than CIs. To analyze the proposed engine further, all containers and corresponding CIs were archived: tabled archived image size is shown in below Table 2.

Table 2:Archived Image size of containers and VMs

Service name	Archived image size (MB)	
	Container on Docker	Cloud instance
MySQL service	372.10	2176.14
Artifactory service	1312.88	3207.24
Apache Tomcat service	513.46	2269.48
NGNIX server	504.15	2039.47
Springboot service	734.14	2656.57
Jenkins service	3790.11	6638.54

According to the generated results in Table 2, containers were presented more lightweight than corresponding CIs. The lightweight of the containers were affected to the increment of the performance and fast execution of containers.

Due to created portable Docker images; all archived container images were able to migrate from one platform to another. But CIs were not able to migrate from one platform to another. Therefore, the proposed engine was consisted with fast and simple migration features. For the proposed engine, all the resources were open-source software services and tools. Therefore, it did not cost large amount of payments for services.

CONCLUSIONS

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 Docker management. It recommends mounting a Docker volume before launch a Docker container, to archive key data, logs and applications on the host. It helps the Docker volumes to recover the most important data of the container although the container was crashed or destroyed. To face for 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.

REFERENCES

Enterprise Container Platform | Docker. (n.d.). Retrieved October 1, 2019, from https://www.docker.com/

Jha, D. N., Garg, S., Jayaraman, P. P., Buyya, R., Li, Z., & Ranjan, R. (2018). A Holistic Evaluation of Docker Containers for Interfering Microservices. Presented at the IEEE International Conference on Services Computing, San Francisco, CA, USA. https://doi.org/10.1109/SCC.2018.00012

Preeth, E. N., Mulerickal, Fr. J. P., Paul, B., & Sastri, Y. (2015). Evaluation of Docker containers based on hardware utilization. Presented at the International Conference on Control Communication & Computing India (ICCC), Trivandrum, India. https://doi.org/10.1109/ICCC.2015.7432984

How To Install and Use Docker on Ubuntu 18.04. (2019, September 18). Retrieved October 1, 2019, from https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-18-04