# Framework to Deploy Containers using Kubernetes and CI/CD Pipeline

Manish Kumar Abhishek, D. Rajeswara Rao, K. Subrahmanyam
Department of CSE
Koneru Lakshmaiah Education Foundation
Vaddeswaram, India

***Abstract*—Containers are continuously replacing the usage of virtual machines and gaining popularity in terms of scalability and agility in IT Industry. The key concept behind containers is Operating system based virtualization. In cloud, computing containers are getting deployed in terms of computing instances whereas in premises they are getting deployed using Docker as a part of CI/CD pipelines using Jenkin Server. When containers are going to be increased in number, its deployment and resource management is always a concern which is managed using the Kubernetes. Kubernetes is used to deploy and manage the containers in an autonomous manner and Rancher is used to manage the Kubernetes Cluster in an efficient manner. First Analysis is done for the scheduler, resource management which is used by Kubernetes to deploy the containers and proposed a framework which will automate the whole process using the helm-charts, ansible scripts from container deployment to the management of Kubernetes Cluster in a scalable manner. It is fully automated framework and can be used to deploy the scalable applications in form of containers as Docker images. CI/CD pipeline is also considered using Jenkin Server.**

***Keywords*—Containers; Docker; Jenkin; Kubernetes; rancher; virtualization**

## I. Introduction

In an agile environment, the application needs to be scalable, performant and highly available based on customer requirements and to achieve the same, containers are widely used in IT industry. The releases are so frequent in terms of delivery in cloud based SaaS (Software as a Service) environment. This is one of the main reasons containers are gaining popularity and used in cloud computing environments and High Performance Computing [1]. Containers are very lightweight in terms of application deployment as one whole package including its required libs, binaries and other dependencies, if any. When an application is split in terms of micro services, every feature is implemented as a micro service and going to be deployed using containers. As a whole Product, the entire application is split in terms of multiple micro services and to have scalability and high availability, every application has a fail back mechanism and to avail the same, each application is deployed with at least two containers. In case one is down another will be available to serve the request or both are used to distribute the workload. In result of this if one product has n numbers of application then need n multiply by two containers. The huge numbers of containers are going to be deployed and then needs to be managed. This requirement of managing high number of containers deployment is always a concern. These containers are provisioned using the Docker images. Docker is an open source platform to bundle the services in form of containers. Using various components, the applications are bundled as Docker images. The required libraries, binaries and other dependencies are defined as a part of configuration in terms of Docker File. Docker file is converted in form of image and then deployed by running commands. Using commands, the image is deployed and the application starts running within container within few seconds. The complete lifecycle of the application revolves around the container lifecycle. Using Docker the state of application is also maintained via commit the container state as an image and tagging it with multiple versions accordingly. If at one point, application crashes, the committed state of container in form of image can be easily deployed again and application will start running with same state when it is committed. This is one of the reason containers provides fault-tolerance and high availability for applications. They are highly trending for application deployment and wherever micro services are getting designed. In comparison of virtual machines, more weightage has been given to the containers in cloud computing as well as in high performance computing. The overhead of application deployment is reduced as it runs on an operating system isolated layer which is portable without use of a hypervisor.

Kubernetes is used for cluster systems to support the container based application deployment. Containers where they are going to be deployed in known as “pod” and it manages thee multiple pod deployment across the physical servers, scaling out the application at run time with multiple workloads. It provides multiple services and tools which are widely available. It is used to avoid downtime of an application. If one container gets stopped or crashed, the another one needs to be up and running in next second. This is the behaviour which is handled easily by Kubernetes. It also offers service registry and load balancing. Multiple containers can reside within a pod to use its file systems and other services belong to a particular pod. The functional or dynamic programming where resource provisioning is so frequent in terms of milliseconds and containers are used, the deployment and its performance need to be monitored. For example: AWS Lambda where multiple user streams are generating the events which are processed by a lambda function. The whole process is executed by deploying a container and billed at 100ms interval of time. The container will be stopped as the function completes its execution. This container deployment, management, monitoring where lambda function is hosted and its performance impacts the provider ability to facilitate the more efficient charging alternatives to the users to process the stream based applications.

Framework is proposed for deploying the containers using Kubernetes based on high performance and fully automated to process the requests which need multiple deployments of containers within few milliseconds. It will identify the required states associated with pods and containers. It can be further used as a configuration to monitor the resources and other details acquired from a Kubernetes Cluster deployment. Using this framework, individuals can plan the capacity support for applications scalability and can do the evaluation of containers and pods which can impact the application performance.

The structure of this paper is as follows: the second section is the analysis and related work; the third section describes the proposed design of framework using Jenkin server and CI/CD pipeline; the fourth section is the evaluation and results; and the fifth section is the conclusion and acknowledgement.

## II. Analysis and Related Work

Hardware virtualization and operating system virtualization in terms of virtual machines and containers are always being a research topic from a performance perspective in terms of computing resources such as CPU, memory and storage workloads [2]. In spite of being so much analysis, it is found that many are not familiar or reluctant to use the formal methods. Cloud computing is using a minimal amount of work done on using the formal methods from a performance perspective [3]. Under [4], it is provided as a cyclic design based on particular functional algorithm specifications. Later, it went with the computing resource availability specifications holding the data, control and resources workflow. It was based on Petri Net model [5] capturing the details of functionalities and computing resources requirements involved in the running environment. It starts first with the analyses of the application deployment lifecycle and then understanding of the execution behaviour at run time. It combines the analyses with simulation and predicts the non-functional and functional requirements. Over a time of period, this model is enhanced with the inclusion of performance minimal and maximum boundaries. It allows the competition of resources consumption via formulating the model which is not considered in nude queued networks. The requirement of these models to work is the historical data which need to be feed in form of temporary data. The virtual machine performance has been evaluated in cloud computing environments [6]. Using [7], [8] the containers and virtual machines performance have been evaluated with multiple performance metrics. The few designed have been evaluated in past to manage the containers using Docker and Kubernetes but there is a limitation exists in research area around containers deployment and its management using the Kubernetes architecture. In [9], Containers using Docker performance results into a degradation of network and CPU based negligible performance impact in specified configurations. Kubernetes is not using fully nested-container strategy. It uses the partial one having pod concept where the same IP is used across the containers deployed within that pod. It uses multiple performance metrics for Pod start up and REST API request-response time. Kubemark is used for the Kubernetes Cluster performance evaluation.

Kubernetes is not a traditional platform based system. It operates at container and offers flexibility, monitoring, scaling, load balancing and deployment of containers [10]. There is no limitation for application type with any amount of workload. It is used for containers not for source code deployment. Using its API, required specifications can be declared for the containers which eliminate the requirement of orchestration where steps are executed one by one in sequential order. It is holding a complete independent set of controlled processes which drives continuously the present state to the targeted one. How to reach from one point to another does not matter which make it easy to use extensible and resilient. It is formed using a set of worker machines known as nodes which are used to execute the application in containers and mainly hosts the pods. There is a control panel which is responsible to manager the worker nodes and pods in the Kubernetes Cluster including scheduling, start-up of the new pod, detecting and responding to the triggered events.

Kube-API server is used for the API which is the frontend of control panel and offers horizontal scalability. Fig. 1 shows the high level flow of container's deployment using the Kubernetes. Kube-Scheduler is used to select a node to the newly created pod to run on. For scheduling the node to the pod several factors need to be considered which includes resources requirements, specifications, deadline and infrastructure based policy constraints. There is also kube-controller manager which manages the different type of controllers. For example: Job Controller, Node Controller, Service Account & Token Controller and Endpoint Controller. etcd is used to store the info about the cluster in terms of key-value pair. For cloud based environment, it also offers cloud-controller manager which links the Kubernetes cluster to the cloud based API. Multiple components are also running on the nodes which manage the running pods. Kubelet and kube-proxy are among those components. Kubelet is an agent which runs on the node to make sure that containers are running fine in pod where kube-proxy manages the network rules to make the communication inside and outside of cluster. Containerd is one of the container runtime used by Kubernetes which is mainly holding the responsibility of running the containers [11]-[13]. Kubernetes monitors the container resources via saving the time-series based metrics in a centralised data base. It offers an UI in form Dashboard using which users can monitor the resources and can search on logging i.e. view the logged activity perform against the running application in container. The pod will live till the containers are running which are deployed inside it. Its lifecycle depends on the container lifecycle. The pod required to be waiting till the containers have been created. Using Object Nets [14]-[15] abstraction, pods and containers can be represented as System and Token Nets. To improvise the legibility, the creation part is hidden and apart from this, it is assumed that as long as resources are present, the scheduler is going to allocate a single node to a single pod. If the resource exhausted, the pods will reside in the waiting queue. The node represents the management of resources. For every node, there will be a token which identify the node and its available computing resources.

The allocated resources to a pod will be released based on the policy value i.e. “release” or “failure”. Containers creation will get started only when pod is going to be assigned to node. The pod will wait in waiting queue till containers creation will get over. Once all containers get created, pod will be moved to the running state. It will remain in same state till the time containers will not terminate. If any container gets terminated, pod will go in runningFailed state. If container gets restarted without any failure, pod will come back to running state and come to the success state once all containers finished without any failure. Fig. 2 shows the same behaviour. The different transition states are represented from TS1 to TS7. Table I elaborates all the transition states.

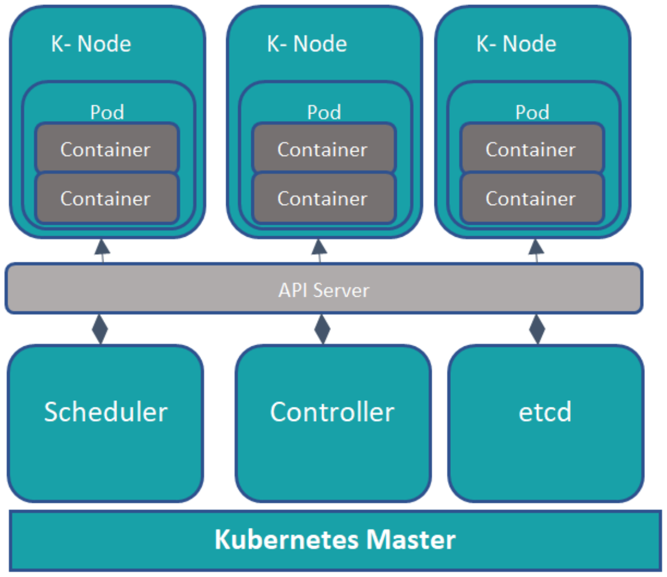

Fig. 1. Container Deployment using Kubernetes.

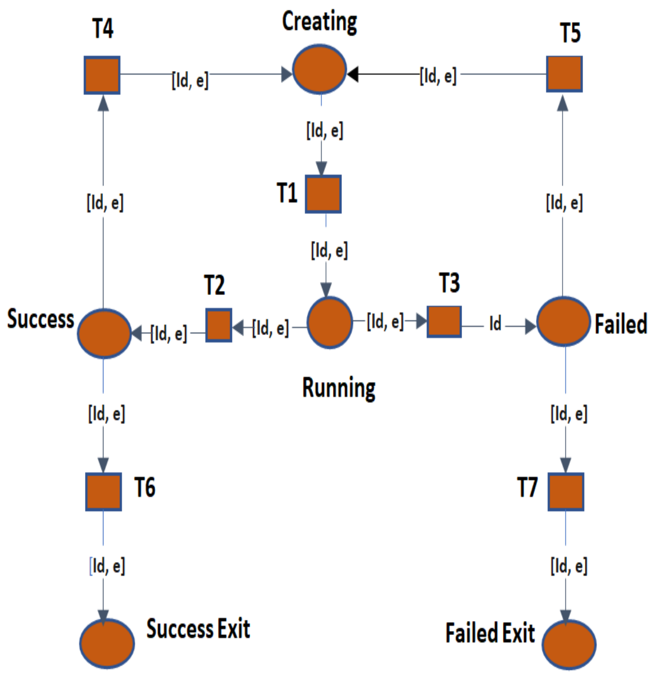

Fig. 2. Transition States Model.

TABLE I. TIME BASED TRANSITION STATES IN THE MODEL

| Transition States | Description |
|---|---|
| TS1 | Creation time of a conatiner. |
| TS2 | Execution time of a container |
| TS3 | Time until next failure |
| TS4, TS5 | Time taken to restart a container |
| TS6, TS7 | Successful termination of a conatiner |

## III. PROPOSED DESIGN

Here, the proposed framework is explained to deploy the containers using the Kubernetes cluster which is going to be managed by Rancher. The applications are bundled in form of a single entity as container which will get deployed on a pod. Multiple applications are considered based on different workloads and resource requirements. It consist multiple stages from building the application to its deployment. At first the application will be built and bundled in terms of a jar/war or based on application type. Secondly, Docker file will be created that will hold the multiple instructions, configurations required to execute the application. Once Docker file will be defined, Docker image will be created using the same; it will be tagged based on release version. Using the Jenkin server the application will be built as a part of CI/CD pipeline which will be responsible to take care of whole process from building the application till its execution. Helm Charts have been used in form of YAML files to deploy these container images. In these YAML files, multiple steps have been defined. For example: version, stages from checkout to deployment , repository from where it is needed to checkout the application source code, building the Docker image with name, Docker file, its tag, working directory and deployment details. Ansible scripts have been used to install the helm charts in an automated manner. Whenever the new version of application is available, using CI/CD pipeline; it will be checkout, built, converted into a Docker image and gets installed using the helm-charts via running the Kubernetes commands [16]. Fig. 3 shows the proposed architecture to deploy the containers using Kubernetes followed by CI/CD pipeline.

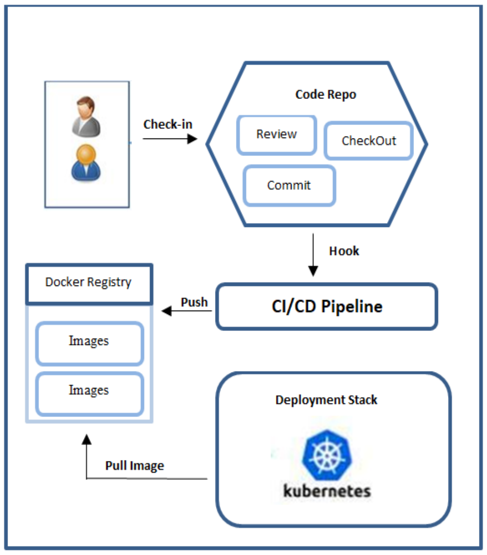

Fig. 3. Proposed Model using Kubernetes followed by CI/CD Pipeline.

### A. Jenkin Server

Jenkin Server is used to automate the whole process of check out the application source code, getting compiled, and bundled in appropriate package/build and its conversion to the Docker image and persist it to Docker registry. The main advantage of using the Jenkin server is facilitating the CI/CD pipeline [17] to deploy the containers in form of helm charts using Kubernetes Cluster. It is easily configurable and extendible. For every application, one pipeline is defined with respective set of instructions in Jenkin Server. It also helps us to check the error at run time by using its flexible UI. For Docker images and helm charts, leveraged its plugin architecture and found it really helpful at every stage of pipeline.

### B. Continuous Integration and Deployment (CI/CD)Pipeline

Jenkin Server is used for delivering and deploying the application as container in form Continuous Integration and deployment pipeline which is split into multiple stages. Main branch of source code is targeted to pull the source code. In the first stage of CI/CD pipeline [18], whenever the code will be pushed to the respective release branch, it will start building the application in terms of Docker image. The second pipeline will be created for the helm install. It will first setup the infrastructure in terms of database, ElasticSearch, Kafka, Zookeeper if any. After the infrastructure build, it will start pushing the helm packages and charts to respective node. The helm package will be extracted and start executing the YAML files using the ansible scripts. It will run the helm install commands to deploy the containers on pod. As a next stage of pipeline, it will check the health of container. If it is up and running, it will end the pipeline successfully else in case of failure, it will wait for some time as retry else will terminate the container and exists. The Kubernetes Cluster is responsible for the containers deployment but to manage the multiple Kubernetes Cluster health, Rancher Server is used [19]. It provides a Dashboard using which multiple Kubernetes Cluster are monitored and managed.

## IV. Evaluation and Result

Evaluation is carried out and benchmarked the overhead during the deployment of containers using Kubernetes with the consideration of following scenarios. (i) Multiple containers i.e. 17 in count are deployed within single pod. (ii) Multiple pods i.e. 4 in count and single container deployment per pod. These 4 pods are deployed on single physical host of 16 cores on Kubernetes node. Mean is represented by symbol ($\mu_i$) and standard deviation is represented by symbol ($\sigma_i$). For comparing the results it is equated like $N_0$ which is the difference both means as $\mu_1 - \mu_2 = 0$ and $N_1$ is going to be the difference as: $\mu_1 - \mu_2 \neq 0$. TC represents the total number of containers.

For the CPU intensive benchmarking used the pov-ray 3.7 for the measurement of overhead over the pods. Kubernetes allows the containers CPU reservation based on Docker. Sharing of multiple CPUs is directly proportional to the Docker-based reservations. IN scenario (i), as there are multiple containers on a single pod, so it is going to be distributed but in scenario (ii) where single container is deployed per pod, one container can consume the all CPU cores. Table II shows the results for same where execution time is linear. It is found that for the CPU usage, Kubernetes has introduced about 13% overhead. It is concluded that for CPU intensive applications instead of deploying single container per pod, multiple containers deployment on a single pod is recommended. Multiple containers are not resulting into the addition of any type of overhead. If application has the extensive tasks to do at the same time then deployment of application in terms of one replica is not recommendable.

For the I/O intensive, BZip is used for the measurement of overhead. During this experiment, $N_0$: $\mu_1 - \mu_2 = 0$ is targeted. Table III shows the outcome of measured results of using the BZip for all 17 containers execution time during the compression of the UNIX kernel. It is found that I/O intensive applications are not impacting the deployment. The overhead is almost negligible even the file system is shared across all the deployed containers.

For the network benchmarking, used the iperf server [20] and client deployment on the pods. Server and client are on same physical machine. The containers are deployed in a pod is going to share the IP address in terms of network connection. The TCP based traffic has been monitored by running the tests for about 1 minute. Both the scenarios (i) and (ii) have been considered to find out the impact of network connection. Tables IV and V show the results for both scenarios where single container per pod and multiple container in a pod. It is found that for the running application more than 5 containers. It is concluded that group of few containers on a single pod is better than having higher number of containers deployed within a pod. This number can be fine-tuned based on workloads and application type.

TABLE II. POV-Ray for CPU Intensive based Application

| TC | Scenario 1 | | Scenario 2 | | $N_0$? |
|---|---|---|---|---|---|
| | $\mu_1$ | $\sigma_1$ | $\mu_2$ | $\sigma_2$ | |
| 1 | 122..34 | 0.42 | 122.23 | 0.38 | Yes |
| 5 | 467.56 | 0.94 | 469.14 | 0.59 | No |
| 9 | 936.80 | 0.71 | 936.58 | 0.68 | Yes |
| 13 | 1411.66 | 1.57 | 1414.30 | 1.25 | No |
| 17 | 2360.22 | 1.14 | 2364.37 | 3.87 | Yes |

TABLE III. BZip for Input / Output Intensive based Application

| TC | Scenario 1 | | Scenario 2 | | $N_0$? |
|---|---|---|---|---|---|
| | $\mu_1$ | $\sigma_1$ | $\mu_2$ | $\sigma_2$ | |
| 1 | 15.04 | 0.15 | 14.97 | 0.23 | Yes |
| 5 | 15.93 | 0.14 | 15.916 | 0.15 | Yes |
| 9 | 18.19 | 1.40 | 18.88 | 0.53 | Yes |
| 13 | 21.73 | 1.42 | 20.33 | 1.09 | Yes |
| 17 | 35.34 | 2.67 | 34.58 | 0.98 | Yes |

TABLE IV. N/W BENCHMARKING USING C-PERF CLIENT FOR SCENARIO 1

| TC | $\mu_1$(GB) | $\sigma_1$ | $\sum BW_i/TC$(GB) |
|---|---|---|---|
| 1 | 1.86 | 0.07 | 1.86 |
| 5 | 9.62 | 0.24 | 2.14 |
| 9 | 16.65 | 0.11 | 1.90 |
| 13 | 15.98 | 0.25 | 1.26 |
| 17 | 17.87 | 1.23 | 2.13 |

TABLE V. N/W BENCHMARKING USING C-PERF CLIENT FOR SCENARIO 2

| TC | $\mu_1$(GB) | $\sigma_1$ | $\sum BW_i/TC$(GB) | $N_0$? |
|---|---|---|---|---|
| 1 | 1.88 | 0.05 | 1.88 | Yes |
| 5 | 9.82 | 0.08 | 2.19 | Yes |
| 9 | 16.95 | 0.12 | 2.04 | Yes |
| 13 | 17.18 | 0.20 | 1.16 | No |
| 17 | 18.17 | 1.35 | 2.19 | No |

## V. CONCLUSION

For containers deployment, Kubernetes is highly recommendable. Wherever there is a need to provision the high number of computing instances frequently within seconds, the overhead attached to the containers and resource allocation is a limitation. In this paper proposed a flexible, automated and performance based framework that can be used by developers or students in their labs to deploy the containers using Kubernetes. It can be used for any application release, capacity planning and for resource management. It is highly flexible in nature using the helm-charts. In this framework not only the deployment of containers are outlined but also focused on managing the Kubernetes cluster using the Rancher. The life cycle of container is also elaborated and pods internally. The CI/CD pipeline based on Jenkin Server is making this framework fully automated. It is not only offering the fault-tolerance but also support the horizontal scalability of an application in terms of containers. The fully automated framework is elastic in nature without any single manual interruptions.

## ACKNOWLEDGMENT

I am really thankful to the Koneru Lakshmaiah Education Foundation for allowing me to choose my research area as per my interest and my guide optimistic nature as well as other staff members who encouraged me to complete this research work.

## REFERENCES

[1] Abhishek, Manish. (2020). Containerization for shipping Scientific Workloads in Cloud. International Journal of Advanced Trends in Computer Science and Engineering. 9. 5327. 10.30534/ijatcse/2020/166942020.

[2] Abhishek, Manish. (2020). High Performance Computing using Containers in Cloud. International Journal of Advanced Trends in Computer Science and Engineering. 9. 5686. 10.30534/ijatcse/2020/220942020.

[3] V. Rastogi, C. Niddodi, S. Mohan, and S. Jha, "New directions for container debloating," in Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation, ser. FEAST '17. New York, NY, USA: ACM, November 2017.

[4] Merino, Alberto & Tolosana-Calasanz, Rafael & Bañares, José & Colom, José. (2015). A Specification Language for Performance and Economical Analysis of Short Term Data Intensive Energy Management Services. xxx-yyy. 10.1007/978-3-319-43177-2_10.

[5] T. Murata, "Petri nets: Properties, analysis and applications," in Proceedings of the IEEE, vol. 77, no. 4, pp. 541-580, April 1989, doi: 10.1109/5.24143.

[6] J. Hwang, S. Zeng, F. y. Wu and T. Wood, "A component-based performance comparison of four hypervisors," 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), 2013, pp. 269-276.

[7] W. Felter, A. Ferreira, R. Rajamony and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2015, pp. 171-172, doi: 10.1109/ISPASS.2015.7095802.

[8] M. Raho, A. Spyridakis, M. Paolino and D. Raho, "KVM, Xen and Docker: A performance analysis for ARM based NFV and cloud computing," 2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE), 2015, pp. 1-8, doi: 10.1109/AIEEE.2015.7367280.

[9] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar and M. Steinder, "Performance Evaluation of Microservices Architectures Using Containers," 2015 IEEE 14th International Symposium on Network Computing and Applications, 2015, pp. 27-34, doi: 10.1109/NCA.2015.49.

[10] Abhishek M.K., Rajeswara Rao D. (2022) A Scalable Framework for High-Performance Computing with Cloud. In: Tuba M., Akashe S., Joshi A. (eds) ICT Systems and Sustainability. Lecture Notes in Networks and Systems, vol 321. Springer, Singapore. https://doi.org/10.1007/978-981-16-5987-4_24.

[11] M ondal, S.K., Pan, R., Kabir, H.M.D. et al. Kubernetes in IT administration and serverless computing: An empirical study and research challenges. J Supercomput 78, 2937–2987 (2022). https://doi.org/10.1007/s11227-021-03982-3.

[12] A. Tesliuk, S. Bobkov, V. Ilyin, A. Novikov, A. Poyda and V. Velikhov, "Kubernetes Container Orchestration as a Framework for Flexible and Effective Scientific Data Analysis," 2019 Ivannikov Ispras Open Conference (ISPRAS), 2019, pp. 67-71, doi: 10.1109/ISPRAS47671.2019.00016.

[13] Wei-guo, Zhang & Xi-lin, Ma & Jin-zhong, Zhang. (2018). Research on Kubernetes' Resource Scheduling Scheme. ICCNS 2018: Proceedings of the 8th International Conference on Communication and Network Security. 144-148. 10.1145/3290480.3290507.

[14] Valk, Rüdiger. (2003). Object Petri Nets -- Using the Nets-within-Nets Paradigm. 819-848. 10.1007/b98282.

[15] Kummer, Olaf & Wienberg, Frank & Duvigneau, Michael & Schumacher, Jörn & Köhler-Bußmeier, Michael & Moldt, Daniel & Rölke, Heiko & Valk, Rüdiger. (2004). An Extensible Editor and Simulation Engine for Petri Nets: Renew. Applications and Theory of Petri Nets 2004. 3099. 484-493. 10.1007/978-3-540-27793-4_29.

[16] Spillner, Josef. (2019). Quality Assessment and Improvement of Helm Charts for Kubernetes-Based Cloud Applications.

[17] Moutsatsos, Ioannis & Hossain, Imtiaz & Agarinis, Claudia & Harbinski, Fred & Abraham, Yann & Dobler, Luc & Zhang, Xian & Wilson, Christopher & Jenkins, Jeremy & Holway, Nicholas & Tallarico, John & Parker, Christian. (2016). Jenkins-CI, an Open-Source Continuous Integration System, as a Scientific Data and Image-Processing Platform. Journal of Biomolecular Screening. 22. 1087057116679993. 10.1177/1087057116679993.

[18] S. A. I. B. S. Arachchi and I. Perera, "Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management," 2018 Moratuwa Engineering Research Conference (MERCon), 2018, pp. 156-161, doi: 10.1109/MERCon.2018.8421965.

[19] Vergara Vargas, Jeisson & Umaña, Henry. (2017). A Model-Driven Deployment Approach for Scaling Distributed Software Architectures on a Cloud Computing Platform.

[20] Tirumala, Ajay & Cottrell, Les & Dunigan, Tom. (2003). Measuring end-to-end bandwidth with Iperf using Web100. 10.2172/813039.