Software Availability Enhancement in Preemptible Instance Kubernetes Cluster

Rika Dewi
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Surabaya, Indonesia
rikadewi4444@gmail.com

Rinaldi Munir
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Bandung, Indonesia
rinaldi@informatika.org

Abstract— **From users perspective, cloud is often seen as an unlimited resource that can be used anytime and anywhere. In order to create the illusion as an unlimited resource, cloud service providers must always provide excess resources that exceed users demand. To increase the efficiency usage of their resources, some cloud service providers rent out their excess resources at lower prices with various limitations. Google Cloud Platform is one of the cloud service providers that rents out its excess resources called preemptible instances. Various limitations that preemptible instances have causes a decrease in the availability of the software running on it. In this study, a tool called preemptible lifecycle scheduler is implemented to enhance availability on top of preemptible instances. This is done by scheduling the termination of the preemptible instance so that it occurs outside the application's peak hour range. Based on experiments, the use of preemptible instances in Kubernetes clusters can reduce infrastructure costs by up to 53.085%, but the software will experience a decrease in availability and no graceful shutdown period. By using the preemptible lifecycle scheduler tool, it is proven that it can increase the availability of the software system up to 0.629% during peak hours and increase the chance of graceful shutdown period by 37.1429% to 75% on software that is terminated by the preemptible lifecycle scheduler tool when scheduling the instance life cycle.**

Keywords—preemptible instance; availability; Kubernetes

I. INTRODUCTION

As technology evolves, the software development process also evolves. Various technologies, architectural patterns, and best practices have continued to emerge over the last few years. IT companies are striving to build a sustainable and scalable software system. One of the architectural patterns that offer solutions for sustainable and scalable system is microservices.

The microservice architecture adopts the Single Responsibility Principle which states to gather things that change for the same reason, and separate things that change for different reasons [1]. A microservice architecture will consist of various services that are loosely coupled to each other so that they can be developed independently [2]. For example, an e-commerce application can consist of several services, such as service for payment, logistic, promotion, etc.

With a lot of small services running independently on a microservice architecture, there comes a diversity of environments in which a service runs. Imagine there is an e-commerce company that has thousands of software engineers. One of them might use Ubuntu Linux, while the other uses Windows, MacOS, or other operating systems. This diversity can result in a service which running well in one environment, but failing in another environment. To avoid this, the delivery of these services is carried out with the help of container technology. Containers provide an isolated environment to run a service within. With containers, the process of packaging and shipping applications across environments is made easier.

As the development goes, many more containers have to be set up to support one same software. For reference, in 2020 Netflix has more than 1000 microservices [3]. Rooted in the complexity problem of managing multiple containers, Google released a platform called Kubernetes as a container orchestration solution for container management. Kubernetes makes it easier to deploy multiple containers on multiple virtual machines, manage resource consumption by a container, migrate a container from one host to another, and many more [4].

Google Cloud Platform provides an infrastructure composed of multiple virtual machines (VMs) and utilize Kubernetes as an orchestration mechanism called Google Kubernetes Engine (GKE). In GKE, there are two types of VM which called on-demand instances and preemptible instances. On-demand instances have a much higher level of availability than preemptible instances, but at a much higher price as well. With the same specs and performance, preemptible instances can cost 70-80% less than on-demand instances. However, preemptible instances cannot live for more than 24 hours, and may be shut down at any time.

Related research has been done by Veena, et al. in 2017 concerning the challenges of using preemptible instances on AWS. In that study, there are several hypothetical solutions to the existing challenges, but there are no practical implementation, nor testing to these proposed solutions [5]. Another study conducted by Costa, et al. in 2018 to review the performance and cost differences required to run programs on on-demand and preemptible instances. The results of this study prove that preemptible instances have a cheaper price with the same performance compared to on-demand instances [6]. In this study, a research will be conducted to increase the availability of a software system that runs on preemptible GKE instances.

II. Related Topics

A. Kubernetes

Kubernetes is an open source container orchestration system. Everything in Kubernetes is a declarative configuration of objects that represent the desired state of the system. Kubernetes aims to ensure that the actual conditions on the system match the desired conditions at all times. That is, Kubernetes not only initializes the system according to the desired conditions, but also protects the system from failures that cause the system to become unstable [4].

A pod is the smallest unit in Kubernetes that represents a service unit consisting of one or more tightly connected containers. Pods deployed in Kubernetes run on a node, a computing unit within Kubernetes that can take the form of a physical machine or a virtual machine. Each container in a pod will share the same network data store [7].

Kubernetes is used to manage container workloads on a set of nodes that are joined to form a Kubernetes cluster. In a Kubernetes cluster there will be at least one node that represents a VM/computer and a Control Plane. At each node, there is a kubelet and a kube-proxy. The kubelet is in charge of managing the pod work and communication between the Kubernetes master and other nodes, while the kube-proxy is in charge of forwarding the network from outside to inside the node and vice versa.

All decisions that are global in a Kubernetes cluster will be governed by the Control Plane including the decision to schedule. In Kubernetes, scheduling is the process of a scheduler finding and deciding the best node to run a pod on. This scheduling process is specifically regulated by a Control Plane component called the kube-scheduler. Control Plane also provides an API server that connects to the cloud-controller-manager to interact with cloud provider services such as Google Cloud Platform.

B. Cost Optimization over Amazon EC2 Spot Instances-Research Challenges

A study to find the challenges in performing cost optimization on AWS spot instances was conducted by Veena, et al. in 2017. AWS spot instances cost is determined by bidding from the user for the instance. Spot instances will be given to the user who bids the highest price for that instance. Therefore, distribution and price prediction of spot instances is a challenge in performing cost optimization on AWS spot instances.

Apart from the bids to be made, the challenge for AWS spot instances is to build a fault-tolerant system. This is because systems that are running on top of the spot instance can be terminated at any time if there is a higher bid for that instance. Several techniques that can be used to build a fault-tolerant system are check-pointing and process level redundancy (PLR). In the check-pointing technique, it is necessary to pay attention to the overhead generated when forming check-pointing [5].

C. Performance and Cost Analysis Between On-Demand and Preemptive Virtual Machines

Costa, et al. conducted a study comparing performance and cost between on-demand VMs with high availability and high costs against preemptive instances with the same specifications, only that the availability depends on the cloud service provider. In this study, an analysis was carried out on two cloud service providers, which are AWS with preemptive instances called spot instances and GCP with preemptive instances called preemptible instances.

The test is carried out by running the map reduce program on a cluster consisting of preemptive and on-demand instances. The result is that there is no significant difference in performance between on-demand and preemptive instances of the two cloud service providers, but there is a significant difference in cost. On AWS, there is a 68% cost reduction when using preemptive instances. Whereas in GCP, a 26% cost reduction was obtained when using preemptive instances but this cost reduction can be increased as the use of a larger cluster [6].

III. Proposed Solution

A. Preemptible Instance Limitations

Preemptible instances have the following limitations:

1. Preemptible instances can be terminated at any time by Google Compute Engine.
2. Google Compute Engine always terminates preemptible instances after running for 24 hours.
3. Preemptible instances are limited resources, so there is a possibility that preemptible instances are not available.
4. Preemptible instances cannot automatically migrate to on-demand instances, or are automatically restarted during maintenance on Google Compute Engine.

The thing that is most affected by these four limitations is the level of availability. In addition, another impact that occurs is that applications running on preemptible instances cannot have time to gracefully shutdown. Graceful shutdown itself is a preparation period for the application just prior the termination. Without this graceful shutdown period, data corruption may occur or there may be remaining unreleased resources such as connections to databases.

B. Peak Hour

In general, a software system will have peak hours that occur when there is a high traffic from transactions. In software systems, these peak hours tend to form a pattern that can be easily predicted. For example in e-commerce applications, peak hours will occur when there is a big sale. Another example, in the online transportation application, peak hour occurs during the hours of leaving and returning from work. The unavailability of applications during peak hours will have a more fatal impact, compared to the unavailability of applications at other times.

The proposed solution in this study is based from the fact that there is a peak hour pattern in the software system. Out of the four limitations possessed by preemptible instances, the second limitation is a limitation that also has a pattern, which is the instance is not available after running for 24 hours. By looking at these two patterns, scheduling can be done so that preemptible instances will always be available during peak hours. This scheduling will be done by a tool that will run in the background and is called the Preemptible Lifecycle Scheduler (PLS). This application will perform lifecycle scheduling for each preemptible instance based on the age of the instance

C. *Implementation*

The PLS will be built using Go language which will communicate with Kubernetes using Kubernetes Go client library and communicate with Google Cloud Platform (GCP) using the Google Cloud API. To be able to communicate using the Google Cloud API, PLS uses a service account as an authentication method. There are two main function that will be built in PLS which are scheduling function and node processing function.

1) *Scheduling*

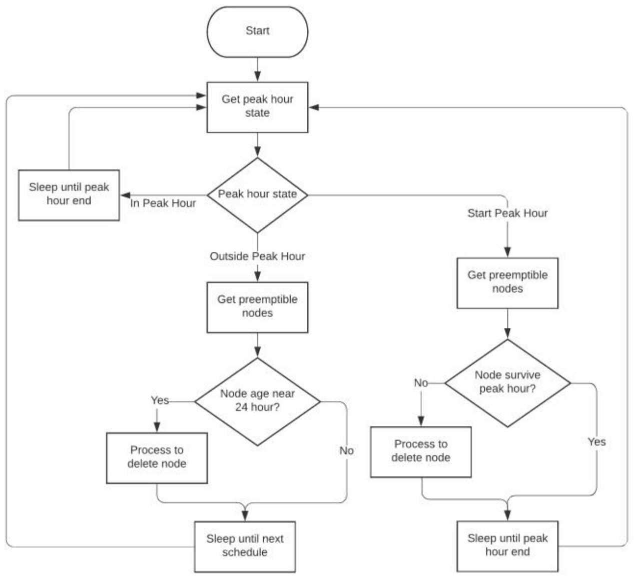

Fig. 1. Scheduling Process

The scheduler is a component that is responsible for scheduling the termination of a node. Fig. 1, shows a flow chart that represents the processes involved in the scheduling process. At first, the scheduler will get the current state based on the peak hour range. There are three possible states, which are in peak hour, outside peak hour, and start peak hour. In peak hour state, PLS will not schedule node termination, so PLS will wait until peak hours are over. In outside peak hour state, PLS will scan preemptible node pool. Each node age will be checked. If any node approaches 24 hours limit, the node will be processed to do gracefully termination. After all nodes have been checked, PLS will wait until the next scheduling time. The PLS wait duration is the smallest remaining node life duration or just before the next peak hour starts. In start peak hour state, PLS will scan preemptible node pool, then each node age will be checked. If the node created time added by 24 hours is less than the end time of peak hour range, then the node cannot survive the peak hour. Nodes that do not persist during peak hours will be processed. Then PLS will wait until the peak hour ends.

2) *Node Processing*

Node processing is a process to terminate node gracefully. Node processing itself consists of three main processes as shown in the flow chart in Fig. 2.

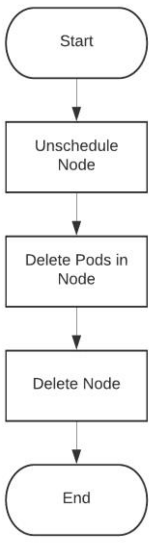

Fig. 2. Node Processing Function

a) *Unschedule Node*

Prior to termination, PLS will notify the kube-scheduler so that no pods are scheduled to the node other than the pods that are already running on the node. The purpose of this process is to avoid scheduling the load (in this case the pod) to the node because the node will soon be terminated.

b) *Delete Pods in Node*

After unscheduled, the next process is to terminate all pods in the node. Termination of a pod is done by sending a termination signal (SIGTERM) to every container running in the pod. The purpose of this process is to gracefully terminate pod while moving the pod to another node. By moving a pod to another node, it can remain available to perform their job even after the node is terminated. The process of moving the pod to another node goes as follows:

1. PLS perform pod termination through Kubernetes API.
2. The terminated pod will send a termination signal (SIGTERM) to every container running in the pod.

3. After each container has stopped running, the pod will be terminated. This changes will be known by the controller of the pod, namely ReplicaSet. The ReplicaSet is responsible for ensuring the number of pod replicas in actual conditions matches the desired number of pod replicas.
4. A ReplicaSet that knows that a pod has been terminated will ask the kube-scheduler to schedule a new pod that will replace the terminated pod so that the number of pod replicas will satisfy the desired number of pod replicas.
5. When scheduling a pod, kube-scheduler will look for the appropriate node for that pod. Note that the first process of node processing is to unscheduled node, so the kube-scheduler will not select current node for scheduling.
6. After the kube-scheduler finds a suitable node, the pod will run on that node so the pod will move from the node to be terminated to another node.

In terminating pods in a node, not all pods need to be terminated. This is because there are certain types of pods that cannot be moved from nodes. This pod will be attached to the node even if the node is unscheduled. These types of pods include pods from the kube-system namespace and pods from DaemonSet. The kube-system namespace is a part of the Kubernetes cluster dedicated to running resources created by the Kubernetes system [7]. Pods residing in the kube-system namespace are generally created to manage nodes in a Kubernetes cluster, so this kind of pods will be re-scheduled on the same node after termination. DaemonSet itself is a component in Kubernetes that ensures that some or all nodes are running a particular pod. DaemonSet is usually used to collect logs or perform monitoring on all or some nodes. Therefore, pods originating from DaemonSet will continue to be scheduled on the same node after termination.

c) Delete Node

The third process after terminating the pods in the node is terminating the node itself. To terminate the node, PLS will communicate with the Kubernetes API which will communicate with the Google Cloud API to terminate the node that represents a preemptible instance on GCP.

IV. EVALUATION

A. Availability

The purpose of this test is to analyze changes in availability level metrics as a result using Preemptible Lifecycle Scheduler on preemptible instances especially during application peak hours.

1) Scenario

For the availability test, this study use a tool called uptime monitor. Uptime monitor will send HTTP requests to health check API endpoints owned by the service every 5 seconds interval. If the response obtained from the HTTP request is an HTTP 200 status code, then the service is available. Vice versa, if the response is not an HTTP 200 status code, or the uptime monitor tool doesn't even get a response, then the service is not available. The uptime monitor will record the service availability based on the number of available responses compared to the total requests sent. The whole process of this scenario is shown in Fig. 3.

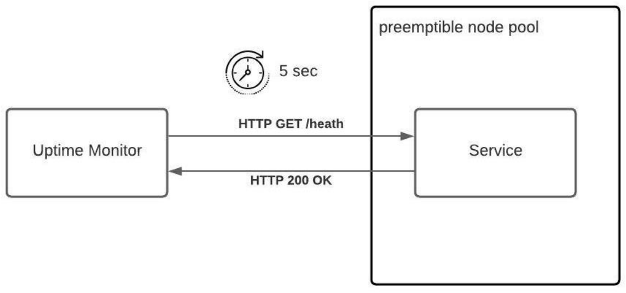

Fig. 3. Availability Test Scenario

Each test case in this test scenario will be carried out for 3 days (3 x 24 hours) with several variations of test cases shown in Table I.

TABLE I. AVAILABILITY TEST CASES

Test Code	Number of Nodes	Number of Pods*	PLS Existence
A-1	7	198	Exist
A-2	7	198	Not Exist
A-3	1	1	Exist
A-4	1	1	Not Exist

**this number does not include pods in the kube-system and DaemonSet namespaces*

The results of this test scenario will produce two types of availability levels, which are the availability level in the peak hour range and the overall availability level. Availability in the peak hour range is obtained when testing is carried out in the peak hour range, while the overall availability is carried out in both the peak hour range and the outside peak range. All test cases in Table I, use the peak hour range from 03.00 to 23.00.

2) Result

Table II, shows the test results according to the variation of test cases in Table I. The results of the 1st day are the test results obtained in the first 24 hours after the test starts, while the results of the 2nd day are the test results obtained at the duration of 2 x 24 hours after the test starts. Similarly, the results of the 3rd day are the test results obtained at a duration of 3 x 24 hours after the test started.

TABLE II. AVAILABILITY TEST RESULTS

Day	Availability (%)							
	A-1		*A-2*		*A-3*		*A-4*	
	Peak Hour	*All*	*Peak Hour*	*All*	*Peak Hour*	*All*	*Peak Hour*	*All*
1	99.5	100	98.5	98.2	99.9	100	99.8	99.7
2	99.1	99.6	98.9	98.7	99.9	100	99.8	99.7
3	98.8	99.1	98.8	98.5	99.9	100	99.8	99.7

In Fig. 4, and Fig. 5, it can be seen that there is an increase in the availability level of the software system with the PLS tool installed in the environment. If calculated on the third day, in the Fig. 4, which shows A-1 and A-2 environments, there was an increase of 0.0508% in the all hour range and an increase of 0.629% in the peak hour range from not using PLS (A-2) to using PLS (A-1). Likewise in the Fig. 5, which shows A-3 and A-4 environments, there was an increase of 0.116% in the all hours range and an increase of 0.26% in the peak hours from not using PLS (A-4) to using PLS (A-3). This proves that the use of PLS on preemptible instances increases the availability of the software system.

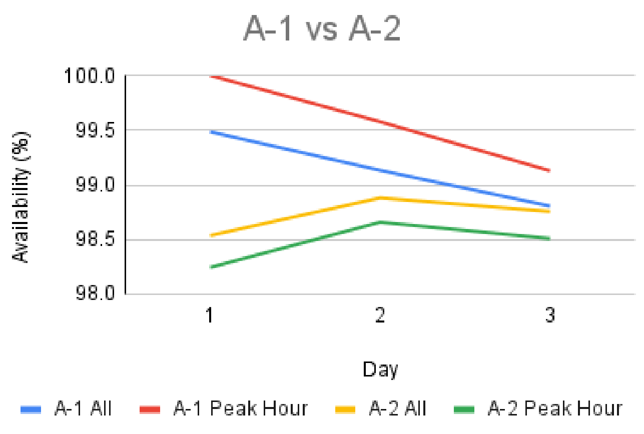

Fig. 4. Availability Test A-1 vs A-2

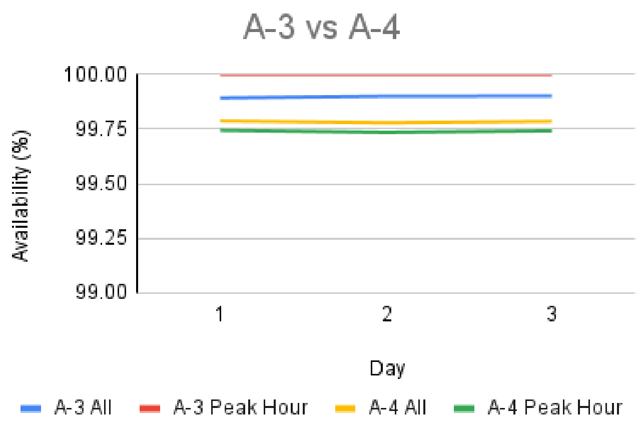

Fig. 5. Availability Test A-3 vs A-4

If we look further at the increase in the level of availability of this software, it is seen that the most significant increase occurs in the peak hour range. This is because PLS is designed to avoid downtime during peak hours. This is most clearly seen in the results of the A-3 test, especially in the peak hour range. It is seen that the software system is able to maintain its availability level of up to 100% in the peak hour range. However, in the A-1 test results, especially in the peak hour range, it is seen that the software system is only able to maintain a 100% availability level on the first day. This is because PLS is only able to overcome the second limitation of preemptible instances, which is the 24 hour age limitation on the instance. There are several other limitations of preemptible instances that may occur which reduce the level of availability of the software.

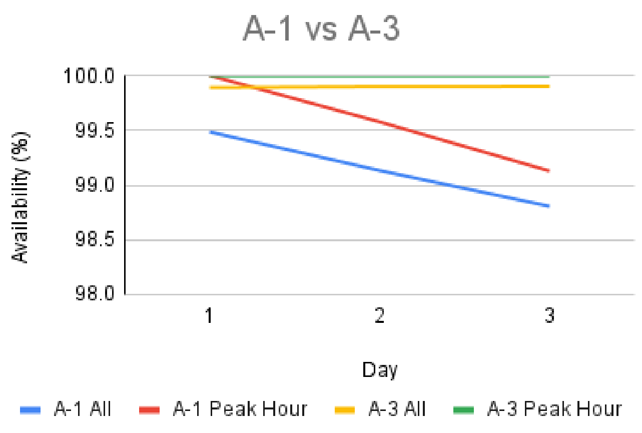

Fig. 6. Availability Test A-1 vs A-3

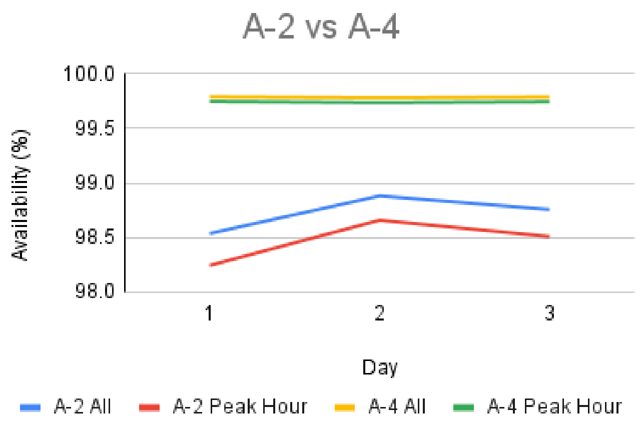

Fig. 7. Availability Test A-2 vs A-4

In addition, it can be seen that the number of pods and nodes also determines the level of software availability. The comparison can be seen in Fig. 6, and Fig. 7. In these figures, it can be seen that the level of availability in a small number of pods and nodes (environments A-3 and A-4) is higher than the level of availability in an environment with a higher number of pods and nodes. This is most likely caused by the pod scheduling process. In scheduling pod to node, kube-scheduler will perform two operations, which are filtering and scoring. The filtering process will produce a set of nodes that meet the requirements for scheduling pods on that node. Then, nodes that pass the filtering process will enter the next process, which is the scoring process. In the scoring process, kube-scheduler will assign scores to each node based on a certain scoring rules. The results of this scoring process will be sorted and the node with the highest rank is used for scheduling the pod.

In the case of a preemptible node pool, the filtering and scoring process will run as follows. First of all, the kube-scheduler will get all nodes in the preemptible node pool except the node that will be terminated (because this node has been subjected to the unscheduled node process). In the experiment conducted in this study, there is no specific scoring algorithm used in the kube-scheduler so it is very likely that the kube-scheduler will choose nodes randomly from the filtered set of nodes. This causes the pod to probably move to a node that is near termination period. This causes a decrease in the availability level of pods in the preemptible node pool with a large number of nodes.

In the case of 1 node, as in environments A-3 and A-4, the kube-scheduler still does not have a scoring algorithm, but the number of nodes is only 1 so that the kube-scheduler will always allocate new nodes that are 0 hours old. This causes pods that have recently moved will not be immediately terminated because the pod is moving to a newly allocated node. To overcome this problem, the scoring algorithm in the kube-scheduler must take into consideration the age of the nodes. The newly allocated nodes will acquire higher score, so pods will tend to move to newly allocated nodes.

B. *Graceful Shutdown*

The purpose of this test is to analyze whether there is a graceful shutdown period in the software as a result of using the Preemptible Lifecycle Scheduler tool on preemptible instances.

1) *Scenario*

For the graceful shutdown test, this study use tools called caller and callee. Caller will be installed in the preemptible node pool. Caller will send an HTTP request to the callee immediately after the caller is started and immediately after receiving the SIGTERM. The callee will be responsible for receiving HTTP requests from the caller and recording the total requests it has received. This test scenario assumes that if the software succeeds in carrying out activities after being terminated, as the caller successfully sends a request shortly after receiving SIGTERM, then the software successfully performs a graceful shutdown. The whole process of this scenario is shown in Fig. 8.

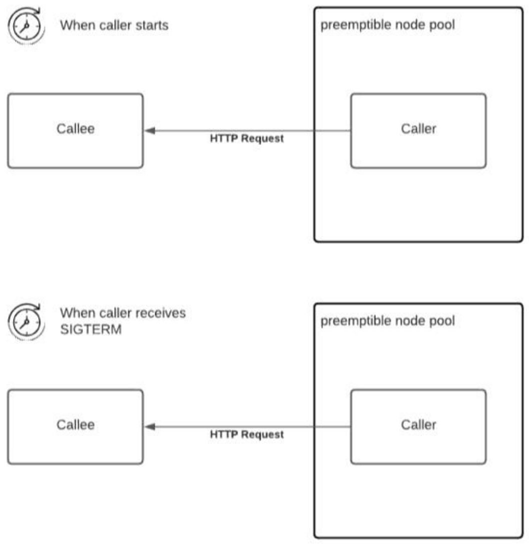

Fig. 8. Graceful Shutdown Test Scenario

Each test case in this test scenario will be carried out for 3 days (3 x 24 hours) with several variations of test cases shown in Table III. All test cases in Table III, use the peak hour range from 03.00 to 23.00.

TABLE III. GRACEFUL SHUTDOWN TEST CASES

Test Code	**Number of Nodes**	**Number of Pods***	**PLS Existence**
G-1	7	198	Exist
G-2	7	198	Not Exist
G-3	1	1	Exist
G-4	1	1	Not Exist

**this number does not include pods in the kube-system and DaemonSet namespaces*

2) *Result*

Table IV, shows the results of the graceful shutdown test after 3 days (3 x 24 hours). The percentage of graceful shutdown in Table IV, is obtained from dividing the number of successful graceful shutdowns (obtained from the total requests received by the callee immediately after the caller receives SIGTERM) to the number of software running (obtained from the total requests received by the callee when caller starts).

TABLE IV. GRACEFUL SHUTDOWN TEST RESULTS

State	**Total Request Received by Callee**			
	G-1	***G-2***	***G-3***	***G-4***
When caller starts	7	5	4	5
When caller receives SIGTERM	4	1	3	0
Graceful Shutdown (%)	57.143	20	75	0

The results from Table IV, shows that there is an increase in the percentage of graceful shutdown after using PLS on preemptible instances. From G-2 to G-1, there is an increase from 20% without using PLS (G-2) to 57.1429% after using PLS (G-1). Meanwhile, in an environment with 1 node and 1 pod, there was an increase from 0% without PLS (G-4) to 75% after using PLS (G-3). This proves that the use of PLS can increase the chance of graceful shutdown for software systems running on preemptible instances.

From the results of G-2 and G-4, it can be seen that without PLS, software running on preemptible instances hardly gets a chance to perform a graceful shutdown. With PLS, the software has a better opportunity to gracefully terminate. However, PLS can only gracefully terminated application that is terminated by PLS when scheduling the node life cycle which was carried out to overcome the 24-hour life limit of preemptible instances.

C. *Cost*

The purpose of the test is to analyze the cost of using preemptible instances compared to using on-demand instances in the Kubernetes GKE cluster. Testing is done by substitution the node pool with the same specifications that initially used on-demand instances into preemptible instances. During the test, daily cost will be recorded for 30 days before and after substitution.

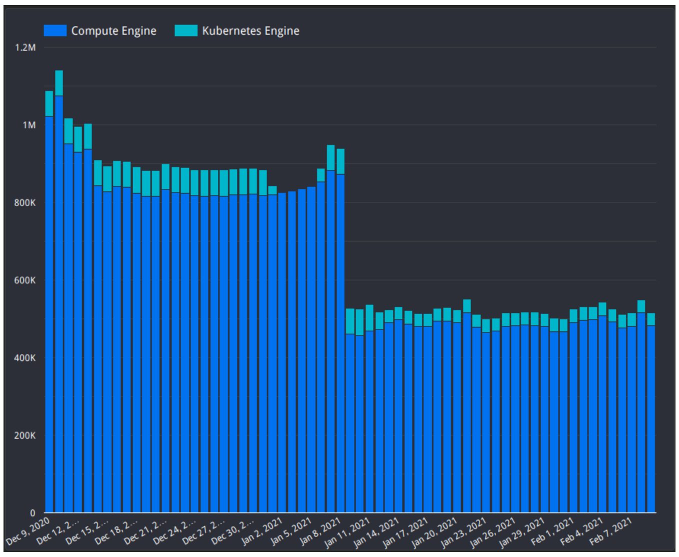

Fig. 9. Cost Test Result

Fig. 9, shows a graph of the cost per day for 2 months used in a Kubernetes cluster. From Fig. 9, it can be seen that there was a significant cost reduction around January 9, 2021. This reduction was due to substitution of virtual machine used from on-demand instances to preemptible instances. From December 9, 2020 to January 8, 2021, the total cost needed to run this Kubernetes cluster is IDR 20,922,607. On the other hand, from January 9, 2021 to February 8, 2021, the total cost needed to run this Kubernetes cluster only IDR 9,815,861. The rate of reduction after using preemptible instances in Kubernetes clusters is 53.085%. This proves that the use of preemptible instances can reduce costs on Kubernetes clusters.

V. Conclusion

The Preemptible Lifecycle Scheduler tool can increase the availability of software systems running on preemptible instances on a Kubernetes cluster up to 0.629% especially during peak hours. The Preemptible Lifecycle Scheduler also increases the chance of having graceful shutdown period by 37.1429% to 75% depends on the number of pod and node. By using preemptible instances instead of on-demand instance in a Kubernetes cluster, the cost used to run the software can be decreased by 53.085%. The number of pods and nodes in a Kubernetes cluster determines the level of availability of the software systems running on it. The higher the number of pods and nodes, the lower the availability level. This is because the scoring algorithm does not consider the age of the node when scheduling pods.

For future works, a better scoring algorithm on the kube-scheduler that takes into account the age of the node during the scoring process can be used. Future research to see the impact of the number of replicas can also be done to achieve zero downtime when terminating node.

Acknowledgment

The author would like to thank God the Almighty for grace and blessings. The author would also thank Dr. Ir. Rinaldi, M.T. for valuable and constructive guidance and teaching during the whole process of this research. The author also express gratitude to family and friends for their support in the making of this paper

References

[1] Martin, Robert C. (2003). Agile Software Development, Principles, Patterns, and Practices. Prentice Hall. p. 95. ISBN 978-0135974445.

[2] Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Inc. ISBN: 1491950315, 9781491950319.

[3] Venugopal, S. (2020). The Story of Netflix and Microservices. GeeksForGeeeks, https://www.geeksforgeeks.org/the-story-of-netflix-and-microservices/, accessed Desember 4, 2020.

[4] Beda, J., Hightower, K., & Burns, B. (2017). Kubernetes: Up and Running. O'Reilly Media, Inc. ISBN: 9781491935675.

[5] Veena, K., Chaturvedi, A., & Gupta, C.P. (2017). Cost Optimization over Amazon EC2 Spot Instances-Research Challenges.

[6] Costa, B., Reis, M., Araújo, A., & Solis, P. (2018). Performance and Cost Analysis Between On-Demand and Preemptive Virtual Machines. In Proceedings of the 8th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER, ISBN 978-989-758-295-0, pages 169-178. DOI: 10.5220/0006709001690178.

[7] Kubernetes. (2021). Kubernetes Documentation, https://Kubernetes.io/docs, accessed April 28, 2021.