ANALISIS PERFORMANSI PROSES SCALING PADA KUBERNETES DAN DOCKER SWARM MENGGUNAKAN METODE HORIZONTAL SCALER

Bayu Arifat Firdaus[1], Vera Suryani[2], Siti Amatullah Karimah[3]

[1,2,3]Fakultas Informatika, Universitas Telkom, Bandung
[1]bayuarifat@student.telkomuniversity.ac.id[2]verasuryani@telkomuniversity.ac.id, [3]karimahsiti@telkomuniversity.ac.id

Abstrak

Container **merupakan teknologi yang belakangan ini banyak digunakan karena adanya fitur-fitur tambahan yang sangat mudah dan nyaman digunakan, khususnya bagi *development and operations (dev ops)*, dengan *Container* memudahkan system administrator dalam mengelola aplikasi termasuk membangun, memproses dan menjalankan aplikasi pada *server Container*. *Container Orchestration* adalah salah satu teknologi *Container*. Dengan *Container Orchestration* proses pembuatan maupun penggunaan *system* tersebut akan semakin mudah tetapi seiring dengan permintaan pengguna yang terlalu banyak sehingga layanan tersebut tidak berjalan maksimal.Oleh karena itu *Container Orchestration* harus memiliki skalabilitas dan performansi yang bagus. Skalabilitas di perlukan untuk system dapat menyesuaikan kebutuhan dengan permintaan user . Dan performansi di perlukan untuk menjaga kualitas layanan yang diberikan. Dalam penelitian ini membahas *Container Orchestration Kubernetes* dan *Docker Swarm* dari sisi skalabilitas dan performansinya. Yang menjadi parameter pembanding antara *Kubernetes* dan *Docker Swarm* adalah *Load Testing* untuk skalabilitas, waktu *scaling up* dan *scaling down* untuk performansi . Hasil penelitian menunjukan untuk skalabilitas *Kubernetes* memakan lebih banyak resource *Cpu Utilization* yaitu pada 10000 user *Kubernetes* memakan resource *Cpu Utilization* dengan rata rata 94,20 % sedangkan pada *Docker Swarm* dengan rata rata 92,28% di karenakan di dalam *Kubernetes* sendiri memiliki *system* yang kompleks terutama komponen komponen khusus seperti *API, Etcd, Scheduler ,Controller manager, kubelet,kube-proxy* untuk menjalankan Container . Sementara di dalam *Docker Swarm* hanya memiliki komponen *Swarm Manager* dan *Docker Daemon* saja . Untuk Performansi *scaling up* pada *Kubernetes* lebih di unggulkan karena penskalaan otomatis sedangkan *Docker Swarm* penskalaan dilakukan manual tetapi dari segi *Load Balancing Docker Swarm* lebih cepat yaitu dengan waktu rata rata 55,8 *second* sementara *Kubernetes* 61,2 *second* . Untuk *scaling down Docker Swarm* di unggulkan dari segi menghapus Container. Di karenakan penghapusan di lakukan manual yaitu dengan waktu rata-rata 11,4 *second*. Meskipun *Kubernetes* terlihat lebih lama dalam menghapus tapi di dalam *Kubernetes* terdapat penghapusan Container otomatis yaitu dengan waktu rata rata 4 *minute* 49 *second*.**

Kata kunci : ***Container Orchestration, Kubernetes ,DockerSwarm,scaling up dan scaling down***

Abstract

Container is a technology that is widely used lately because of the additional features that are very easy and convenient to use, especially for development and operations (dev ops), with Container making it easy for system administrators to manage applications including building, processing and running applications on Container servers. Container Orchestration is one of Container technology. With Container Orchestration, the process of making and using the system will be easier, but along with too many user requests, the service will not run optimally. Therefore, Container Orchestration must have good scalability and performance. Scalability is needed for the system to match the needs of the user request. And performance is needed to maintain the quality of services provided. In this study, discussing Container Orchestration Kubernetes and Docker Swarm in terms of scalability and performance. The comparison parameters between Kubernetes and Docker Swarm are load testing for scalability, scaling up time and scaling down for performance. The results showed that the scalability of Kubernetes consumed more resources Cpu Utilization, namely in 10000 Kubernetes users consumed resources Cpu Utilization with an average of 94.20%, while at Docker Swarm with an average of 92.28%, because inside Kubernetes itself had complex systems, especially special components such as API , Etcd, Scheduler, Controller manager to run Container. While in the Docker Swarm only has a Swarm Manager and Docker Daemon component only. For scaling up performance in Kubernetes is more favored due to automatic scaling while the Docker Swarm scaling is done manually but in terms of Load Balancing Docker Swarm is faster, with an average time of 55.8 seconds while Kubernetes 61.2 second. For Scaling Down Docker Swarm featured in terms of removing the container. Because the removal is done manually with an average time of 11.4 seconds. Although Kubernetes looks longer to delete but inside Kubernetes there is automatic Container removal, which is on average time 4 minutes 49 seconds..

1. Pendahuluan

Latar Belakang

Container dengan cepat mengganti Mesin Virtual (*VMs*) sebagai instance komputasi dalam penyebaran berbasis *cloud*. Biaya *overhead* yang jauh lebih rendah dibandingkan dengan *VMs* telah sering dikutip sebagai salah satu alasan kenapa *Container* lebih bagus dari pada Mesin Virtual (*VMs*) [1]. *Container* adalah cara atau metode menjalankan beberapa aplikasi perangkat lunak pada mesin yang sama. Masing-masing dijalankan dalam lingkungan terisolasi yang disebut *Container*. *Container* adalah lingkungan tertutup untuk perangkat lunak. Yang memungkinkan membungkus semua file dan *library* agar aplikasi berfungsi dengan benar[2].

Salah satu virtualisasi berbasis *Container* yang paling banyak digunakan saat ini adalah *Docker* [3]. *Docker* adalah salah satu alat yang berguna untuk menjalankan *Container*. Namun, mengelola *Container* untuk menciptakan banyak layanan dan menangani *user* yang banyak adalah tugas yang besar bagi *Docker* [4]. Proses *Scaling* adalah masalah utama dalam kasus ini . *Resource* yang tidak memadai yang tidak mampu mengatasi perubahan intensitas beban kerja dari waktu ke waktu , dalam hal ini aplikasi mengalami kinerja yang rendah atau permintaan pengguna yang terlalu banyak ,dalam hal ini pemanfaatan sumber daya yang dialokasikan ke *Container* rendah. Oleh karena itu, penskalaan pada saat runtime di perlukan [5].

Dalam mewujudkan hal tersebut maka dibutuhkan suatu *Container Orchestration* yang merupakan sebuah sistem open source berfungsi sebagai *Controller* yang akan mengatur beberapa *Cluster Container* sehingga akan lebih memudahkan manajemen *Container*[4]. Salah satu nya adalah penskalaan yang secara eksklusif didasarkan pada metrik infrastruktur level, seperti pemanfaatan *CPU Utilization.*

Oleh sebab itu penelitian ini di lakukan untuk membandingkan *Container Orchestration* manakah yang lebih baik dalam scaling menggunakan metode *Horizontal pod autoscaler. Container Orchestration* yang akan di bandingkan adalah *Kubernetes* dan *Docker Swarm* karena kedua *Container Orchestration* ini sangat popular di gunakan pada perusahan besar seperti Google dan Ruang Guru[6].yang menggunakan *Kubernetes* serta *hepsiburada* dan *PinCAMP* [7]. yang menggunakan *Docker Swarm* . Penelitian

Topik dan Batasannya

Rumusan masalah yang diangkat di dalam penelitian ini adalah *Container Orchestration* yang memiliki muatan yang sudah penuh atau permintaan pengguna yang terlalu banyak sehingga layanan tersebut tidak berjalan maksimal. Dan batasan masalah yang digunakan pada penelitian ini adalah sebagai berikut:

1. *Container Orchestration* yang digunakan adalah *Kubernetes* dan *Docker Swarm*.
2. *Container* yang digunakan adalah *Docker Container.*
3. Layanan yang digunakan adalah *webserver.*

Tujuan

Melakukan perbandingan performansi proses *scaling* pada *Kubernetes* dan *Docker Swarm*.Di dalam *Kubernetes* menggunakan metode *Horizontal Pod auto scaler* sedangkan di *Docker Swarm* menggunakan metode scaling. Dengan parameter pembanding antara *Kubernetes* dan *Docker Swarm* adalah *Load Testing* untuk skalabilitas, waktu *scaling up* dan *scaling down* untuk performansi.

Organisasi Tulisan

Penulisan penelitian ini tersusun atas 5 bab. Pada bab 2 akan dijelaskan tentang Studi Terkait, bab 3 dijelaskan tentang sistem yang dibangun, yaitu arsitektur proses scaling pada *container orchestration*. Bab 4 akan dijelaskan tentang evaluasi, dan bab 5 dijelaskan kesimpulan dan saran.

2. Studi Terkait

2.1 Virtualisasi

Virtualisasi sistem adalah semacam teknologi , yang memisahkan perangkat fisik dan sistem operasi. Mesin fisik tunggal dapat dibagi beberapa mesin untuk memaksimalkan sumber daya pemanfaatan dan fleksibilitas [8]. Teknologi ini memungkinkan pengguna untuk saling berbagi sumber daya pada kondisi apapun [9].

2.2.1 Container Based

Container adalah cara menjalankan beberapa aplikasi perangkat lunak pada mesin yang sama. Masing-masing dijalankan dalam lingkungan terisolasi yang disebut *Container*. *Container* adalah lingkungan tertutup untuk perangkat lunak. Yang memungkinkan membungkus semua file dan *library* agar aplikasi berfungsi dengan benar [2].

2.2.2 Docker

Docker adalah proyek *open source* yang dirancang untuk mengembangkan, mengirim, dan menjalankan aplikasi di dalam virtualisasi berbasis *Container*. Kita dapat menggunakan beberapa wadah aplikasi web menggunakan Docker untuk melayani jutaan pengguna[2].

Docker menyediakan kemampuan untuk menjalankan aplikasi dalam lingkungan yang terisolasi. Isolasi dan keamanan memungkinkan untuk menjalankan banyak wadah secara bersamaan pada host yang diberikan. Karena sifat ringan dari *Container* yang berjalan tanpa beban tambahan *hypervisor*, kita dapat menjalankan wadah lebih pada kombinasi *hardware* tertentu daripada jika menggunakan mesin virtual[10].

2.3 Cloud Computing

Cloud Computing adalah model yang memungkinkan kita sebagai user mengakses jaringan sesuai permintaan ke kumpulan sumber daya jaringan dan layanan komputasi yang dapat dikonfigurasi bersama. Sumber daya jaringan ini meliputi: jaringan, server, penyimpanan, aplikasi dan layanan. Sumber daya ini dapat dengan cepat disediakan dan dirilis oleh sedikit perintah atau yang bisa kita sebut *service provider interaction*[11].

2.4 Kubernetes

Kubernetes adalah peyedia sarana untuk mendukung penyebaran berbasis *Container* dalam *cloud Platform-as-a-Service (PaaS)*, dengan fokus khusus pada sistem berbasis *cluster*. Hal ini memungkinkan untuk menyebarkan beberapa "*pod*" di mesin fisik, memungkinkan skalabilitas dari aplikasi dengan beban kerja yang berubah secara dinamis. Setiap pod dapat mendukung beberapa *Docker Container*, yang dapat memanfaatkan layanan (Sistem file dan *I / O*) yang terkait dengan pod [1].

Dalam lingkungan berbasis *Container* seperti itu, ada kegunaan lain yang di miliki *Kubernetes* bahwa *Kubernetes* dapat secara dinamis memantau sumber daya / atau penggunaan aplikasi yang sedang berjalan, dan kemudian menyesuaikan sumber daya yang disediakan, untuk *Container* mengekelola dengan tepat, dan mencegah dari sumber daya yang berlebihan dan kekurangan penyediaan [12].

2.5 Docker swarm

Docker Swarm adalah pendekatan yang baru lahir untuk industri *cloud*, ia memiliki potensi besar untuk menyediakan lingkungan pengembangan *multi-cloud* tanpa mengkhawatirkan kompleksitasnya. *Docker Swarm* adalah alat *clustering* dan penjadwalan, yang menawarkan fungsionalitas untuk mengubah sekelompok Docker . Dengan *Docker Swarm* memungkinkan membangun kelompok sistem kooperatif yang dapat memberikan redundansi jika satu atau lebih node gagal. *Swarm* menyediakan penyeimbangan beban kerja untuk *Container*. Ini menetapkan *Container* ke node yang mendasarinya dan mengoptimalkan sumber daya dengan secara otomatis menjadwalkan beban kerja *Container* untuk dijalankan pada host yang paling sesuai dengan sumber daya yang memadai[13].

2.7 Penelitian Terkait

Pada penelitian sebelumnya, bertujuan untuk pemanfaatan sumber daya khusus nya pada memori untuk load balancing. Dari hasil riset, dapat menyimpulkan bahwa load balancing berdasarkan pemanfaatan memori dapat diimplementasikan menggunakan Docker swarm. Teknik ini dapat mendistribusikan beban server web berdasarkan pemanfaatan memori. Hasilnya menunjukkan bahwa setiap node *worker* menerima distribusi beban yang adil. Teknik ini dapat mengurangi kemungkinan satu titik kegagalan di cluster server web [4].Pada penelitian ini bertujuan untuk pemanfaatan memori untuk menjalankan Load Balancing. Sehingga penelitian ini dapat dijadikan acuan untuk membuat penelitian berikutnya yaitu membuat auto scaling dengan pemanfaatan CPU *utilization*.

Pada penelitian selanjutnya , bertujuan merancang sistem penskalaan otomatis untuk Sistem *API Gateway* berdasarkan *Kubernetes* dan *Prometheus* Dalam makalah ini, Sistem Penskalaan Otomatis dapat secara dinamis menyesuaikan jumlah instance layanan menurut beban API Gateway. Ia menggunakan jumlah instance paling sedikit untuk menyeimbangkan beban kerja saat beban rendah. Ketika memuat melebihi batas yang ditentukan, lebih banyak contoh dibuat secara dinamis untuk menyeimbangkan beban kerja. Ini dapat meningkatkan pemanfaatan sumber daya sistem sambil memastikan ketersediaan tinggi[14]. Pada penelitian ini hanya merancang penskalaan otomatis menurut beban API Gateway tanpa membandingkan dengan *Container Orchestration* lain. Sehingga penelitian ini dapat dijadikan acuan untuk membuat penelitian berikutnya yaitu membandingkan *Container orchestration* dengan *Container orchestration* lainnya dengan metode scaling

Pada penelitian yang lain telah dibandingkan performansi *Container* tanpa *Container Orchestration* dengan yang menggunakan *Container Orchestration* [8]. Pada penelitian ini dikatakan bahwa *CPU Performance* dan *Disk I/O Performance* antara *Container* dengan *Container Orchestration* , performansi yang dikatakan lebih efisien adalah *Container Orchestration* dikarenakan *Container Orchestration* memiliki karakteristik *high density* dan *high elactisity*. Pada Penilitian ini hanya membandingkan performansi *Container* tanpa *Container Orchestration* dengan yang menggunakan *Container Orchestration* sudah jelas dengan Container Orchestration lebih di unggulkan . Sehingga penelitian ini dapat dijadikan acuan untuk membuat penelitian berikutnya yaitu membandingkan *Container orchestration* dengan *Container orchestration* lainnya

Tabel 2-0-1 Penelitian Terkait

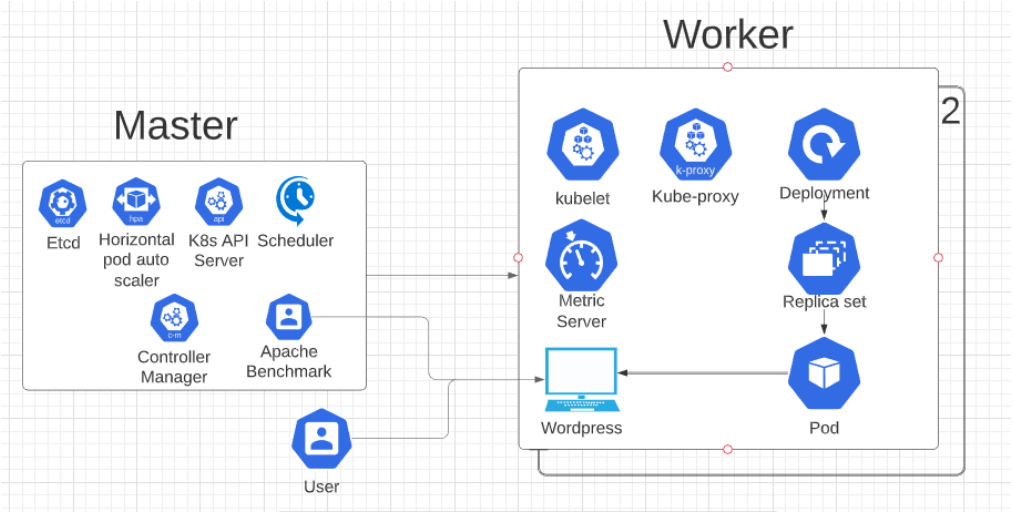
Judul Paper	Tahun	Sistem	Metode	Output
Web Server Load Balancing Based On Memory Utilization Using Docker Swarm[4].	2018	*Container based,*	*Container Orchestration*	*Load Balancing*
An Auto Scaling System for API Gateway Based on Kubernetes[14].	2018	*Container Based*	*Container Orchestration*	*Auto scaling system for API Gateway System based on Kubernetes and Prometheus*
The performance Analysis of Docker and Rkt Based on Kubernetes[8].	2017	*Container Based, Hypervisor Based*	*Container Orchestration*	*CPU Performance* dan *Disk I/O Perfomance*

3. Sistem yang Dibangun

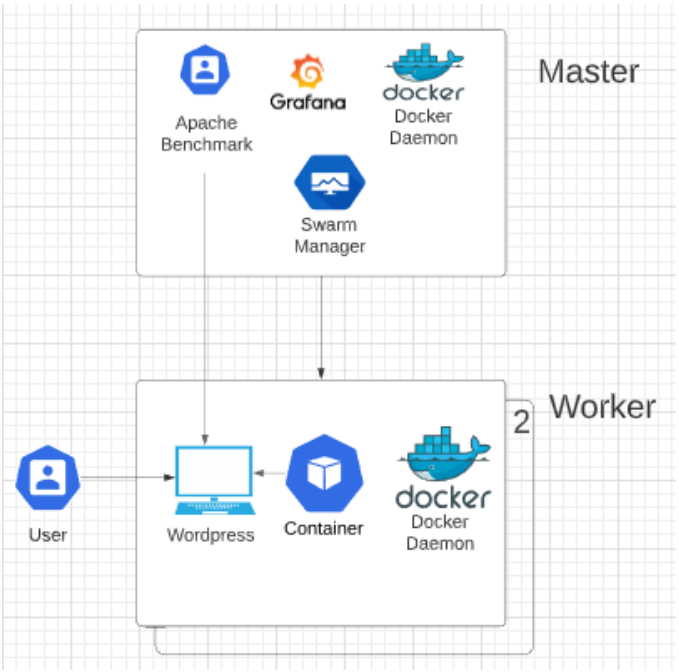
Penilitian ini memiliki dua sistem arsitektur utama yaitu dengan menggunakan *Kubernetes (Gambar 3-1-1)* dan menggunakan *Docker Swarm (Gambar 3-1-2)*.

Arsitektur pertama dimulai dengan memasang *Kubernetes Cluster (Gambar 3-1-1)* yaitu *Master Node* dan 2 *Worker Node*. Selanjutnya akan menginstall *Metric Server* di dalam sebuah *Worker Node* berfungsi sebagai *Monitoring CPU Usage* dari semua *Node*. Lalu memasang *Container Engine* yang sudah berisi layanan Web Server kedalam *Worker Node* . Dan mengaktifkan *Horizontal pod auto scaler* untuk scaling *Pod* otomatis pada *Worker Node*.

Arsitektur kedua dimulai dengan memasang *Docker Swarm Cluster (Gambar 3-1-2)* yaitu *Manager Swarm Master* dan 2 *Swarm node* . Lalu memasang *Container Engine* yang sudah berisi layanan Web Server kedalam *Swarm Node*.



Gambar 3-1-1 Architecture Kubernetes



Gambar 3-1-2 Architecture Docker Swarm

3.1.1 Controller Manager

Komponen di master yang menjalankan kontroler. Secara logis, setiap kontroler adalah sebuah proses yang berbeda, tetapi untuk mengurangi kompleksitas, kontroler-kontroler ini dikompilasi menjadi sebuah binary yang dijalankan sebagai satu proses. Kontroler-kontroler ini meliputi:

Kontroler Node : Bertanggung jawab untuk mengamati dan memberikan respons apabila jumlah node berkurang.
Kontroler Replikasi : Bertanggung jawab untuk menjaga jumlah pod agar jumlahnya sesuai dengan kebutuhan setiap objek kontroler replikasi yang ada di sistem.
Kontroler Endpoints : Menginisiasi objek Endpoints (yang merupakan gabungan Pods dan Services).
Kontroler Service Account & Token: Membuat akun dan akses token API standar untuk setiap namespaces yang dibuat [15].

3.1.2 K8s API Server

Komponen di *master* yang mengekspos *API Kubernetes*. Merupakan *front-end* dari kontrol plane *Kubernetes* dan sebagai penghubung antara node master dengan node worker [15].

3.1.3 Scheduler

.Komponen di *master* yang bertugas mengamati *pod* yang baru dibuat yang belum di-assign ke suatu *node* dan kemudian akan memilih sebuah *node* dimana *pod* baru tersebut akan dijalankan [15].

3.1.4 Etcd

Penyimpanan *key value* konsisten yang digunakan sebagai penyimpanan data klaster *Kubernetes* [15].

3.1.5 Horizontal Pod auto scaler

Horizontal Pod auto scaler adalah metode yang di gunakan untuk *scaling* secara otomatis dengan cara mereplikasi *pod* berdasarkan *CPU utilization* . *Horizontal Pod Autoscaler* secara berkala menyesuaikan jumlah replika agar sesuai dengan pemanfaatan *CPU* berdasarkan yang di pakai oleh pengguna [16].

3.1.6 Kubelet

Agen yang dijalankan pada setiap node di klaster dan bertugas memastikan kontainer dijalankan di dalam *pod* dan juga berfungsi untuk menghubungkan *Kubelet* dengan *Container Engine* [15].

3.1.7 Kube-Proxy

kube-proxy membantu *abstraksi service Kubernetes* melakukan tugasnya. Hal ini terjadi dengan cara memelihara aturan-aturan jaringan (network rules) serta meneruskan koneksi yang ditujukan pada suatu host dan berfungsi untuk menghubungkan *Kubelet* dengan *Container Engine* [15].

3.1.8 Pod

Pod sebagai sumber daya jaringan dan penyimpanan umum dari *Node worker*, serta spesifikasi yang menentukan bagaimana wadah dijalankan[16].

3.1.9 Metrics Server

Metrics Server mengumpulkan metrik sumber daya dari *Kubelets* dan memaparkannya dalam *API server Kubernetes* melalui *Metrik API* untuk digunakan oleh *Horizontal Pod Autoscaler* dan *Vertical Pod Autoscaler*. *Metrik API* juga dapat diakses oleh *kubectl top*, sehingga lebih mudah untuk men-*debug autoscaling pipelines*[17].

3.1.10 Swarm Manager

Menangani orkestrasi cluster. Memonitoring setiap kegiatan agar selalu sesuai dengan keadaan yang di inginkan[18].

3.1.11 Docker Daemon

Bertujuan untuk menjalankan Aplikasi di dalam Docker Swarm , dan menerapkan konfigurasi konfigurasi pada pod di swarm node[18].

3.2 Requirement Sistem

Beberapa *tools* dan perangkat keras yang nantinya akan digunakan pada penelitian yaitu

3.2.1 Master

Tabel 3-2.1 Spesifikasi Perangkat Keras Master

Komponen	Spesifikasi
Processor	2 vCPUs, 2.3 GHz, Intel Broadwell E5-2686v4
RAM	4 GiB memory
HDD	8 GB HDD

3.2.2 Worker

Tabel 3-2-2 Spesifikasi Perangkat Keras Worker

Komponen	Spesifikasi
Processor	1 vCPUs, 2.5 GHz, Intel Xeon Family
RAM	1 GiB memory
HDD	8 GB HDD

Perangkat lunak yang akan digunakan antara lain :

1. *AWS* sebagai *instance* untuk menjalankan *Container Orchestration*
2. *Docker* sebagai virtualisasi *Container* dan *Clustering Container*

3. *Kubernetes* sebagai *controller Container*
4. *DockerSwarm* sebagai *controller Container*
5. *Metric Server* sebagai Monitoring di *Kubernetes*
6. *Grafana* monitoring di *Docker Swarm*
7. *Apache Bench* sebagai stressing tools untuk *Webserver*

3.3 Skenario Pengujian

Parameter pembanding antara *Kubernetes* dan *Docker Swarm* adalah *Load Testing* untuk skalabilitas, *waktu scaling up* dan *scaling down* untuk performansi.

1.Load Testing
Untuk uji skalabilitas akan digunakan *Request Generator* yaitu *Apache Bench* dimana *Web Server* akan di isi oleh *Apache Bench* dengan user yang sudah di tentukan . Adapun parameter yang akan digunakan ialah Load Testing , dengan cara mengukur *resource CPU Utilization* dari masing masing *Container Orchestration* . Dengan scenario memasukan user yang telah di tentukan adalah 2.000 user ,5000 user , 10.000 user .

2.Waktu Scaling up Scaling down
Untuk uji performansi ketika *CPU Utilization* kedua *Container Orchestration* telah mencapai 90% yang dimana 90% ini adalah ambang atas optimal untuk melakukan *scaling* [19] .Akan di lakukan proses scaling pada masing masing *Container Orchestration.* Adapun parameter scaling up yang akan ukur yaitu waktu *Container creating time* dan *Load balancing* .

Dan untuk scaling down ketika layanan sudah tidak di gunakan masing masing *Container Orhcestration* akan menghapus *Container* yang sudah tidak di gunakan . Adapun parameter yang di ukur ialah waktu *delete Container*

4. Evaluasi

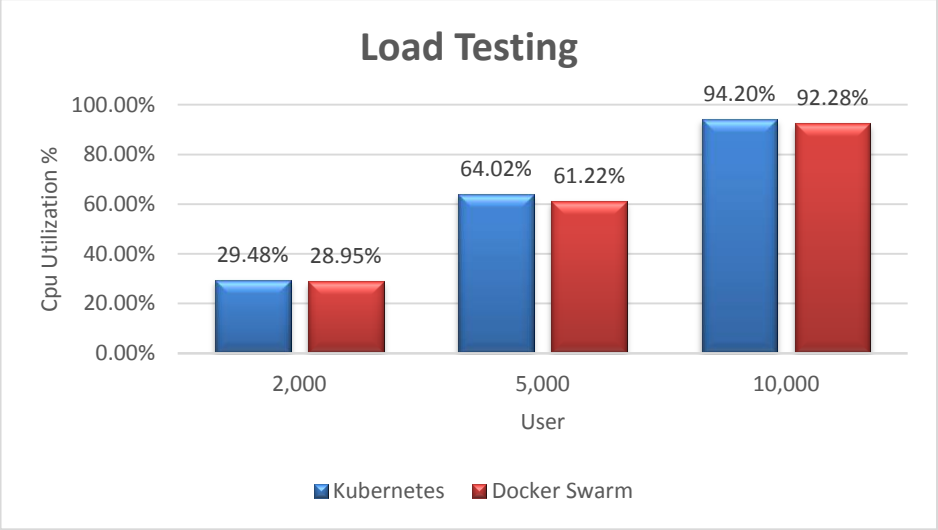
Bagian ini menampilkan hasil analisis. Seperti yang dijelaskan sebelumnya, parameter yang dipilih untuk diukur adalah *Load Testing* untuk skalabilitas, *waktu scaling up dan scaling down untuk* performansi.

4.1 Load Testing

4.1.2 Tujuan Load Testing

Tujuan Load Testing adalah untuk melihat skalabilitas dari masing masing *Container Orchestration* . Dengan cara melihat beban pada *CPU Utilization* dengan user yang sudah di tentukan. *CPU Utilazation* menunjukkan berapa presentase yang dibutuhkan dalam proses yang yang dilakukan pada percobaan.

4.1.3 Hasil Load Testing



Gambar 4-1-3-1 Hasil Rata rata Load Testing

4.1.4 Analisis Load Testing

Terlihat dari gambar hasil rata rata *Load Testing* saat generate 2000 user memakan CPU utilization dengan rata rata 29,48 % sedangkan pada *Docker Swarm* 28,95% .Selanjutnya saat generate 5000 user memakan CPU utilization dengan rata rata 64,02 % sedangkan pada *Docker Swarm* 61,22% .Selanjutnya saat 10000 user memakan CPU utilization dengan rata rata 94,20 % sedangkan pada *Docker Swarm* 92,28% . Di karenakan di dalam Kubernetes sendiri memiliki *system* yang kompleks terutama komponen komponen khusus seperti ,Etcd,Scheduler,Controller manager untuk menjalankan Container . Sementara di dalam *Docker Swarm* hanya memiliki komponen Swarm Manager dan *Docker Daemon* saja..

4.2 Waktu Scaling up Scaling Down

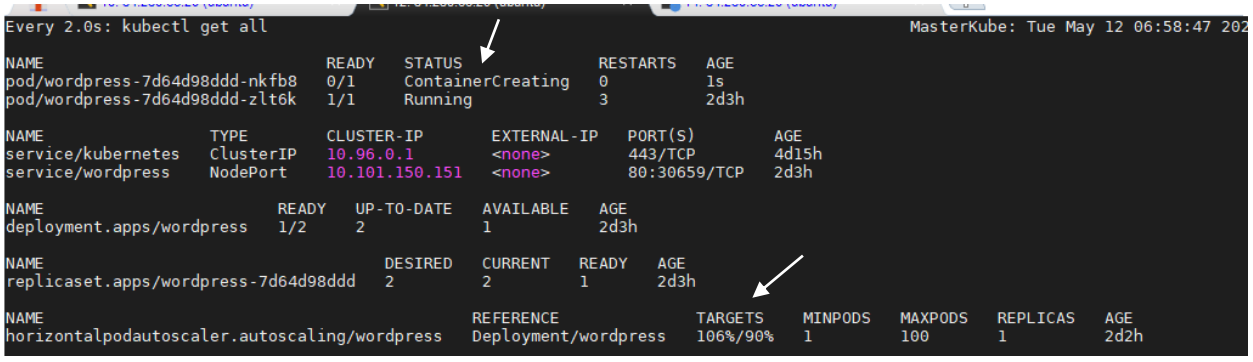
4.2.1 Tujuan Waktu Scaling up dan Scaling down

Tujuan Scaling up adalah untuk melihat performansi dari masing-masing *Container Orchestration* . Dengan cara melihat waktu *Container creating time* . dan waktu *Container Orchestration* melakukan *Load balancing.*

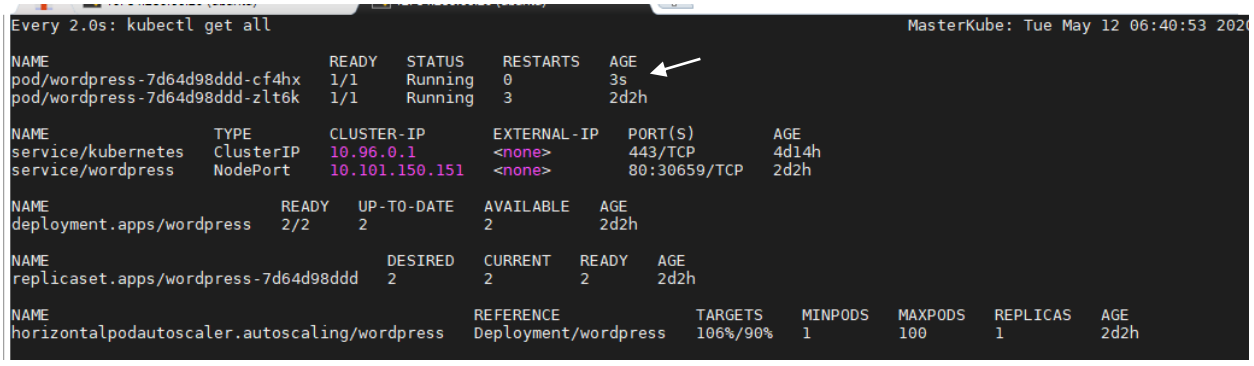
Tujuan Scaling down adalah untuk melihat performansi dari masing-masing *Container Orchestration.* Dengan cara mengukur waktu *Delete Container* ketika Container tersebut sudah tidak di gunakan

4.3 Waktu Scaling up Scaling Down Kubernetes

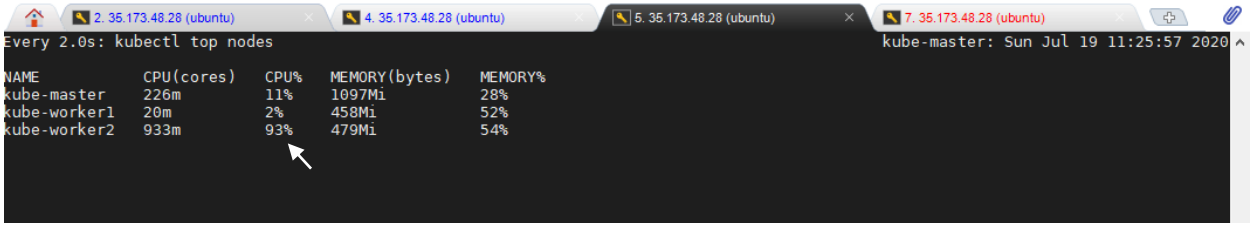
4.3.1 Hasil Waktu Scaling up



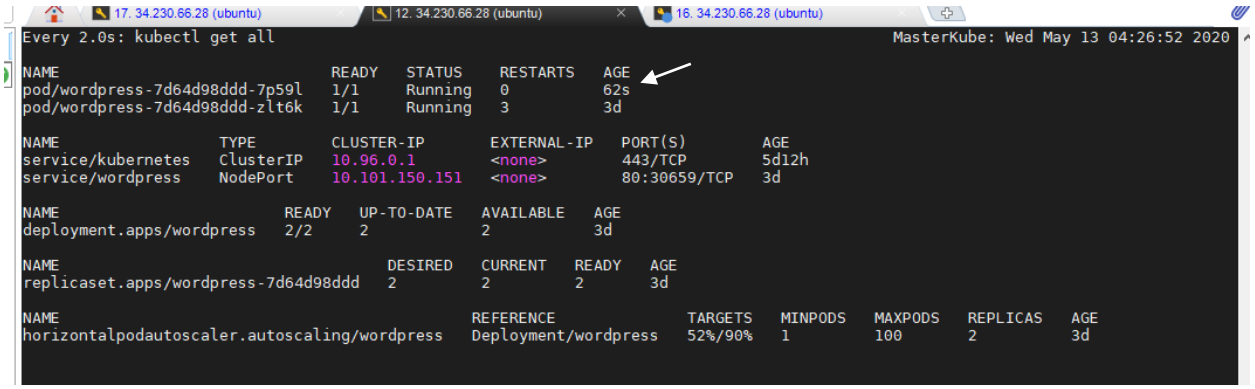
Gambar 4-3-1-1 Utilization Pod (Container) mencapai limit



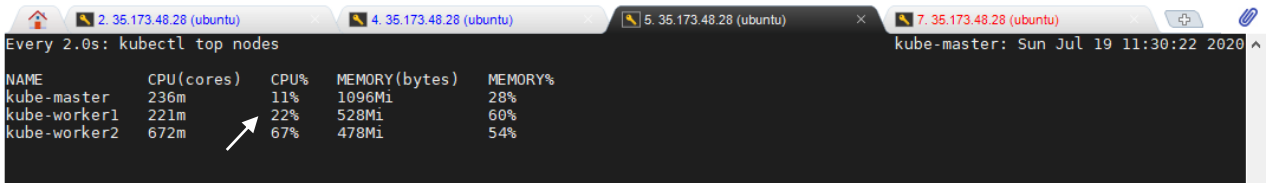
Gambar 4-3-1-2 Pod (Container) di buat dalam waktu 3s



Gambar 4-3-1-3 CPU Utilization sebelum Load Balancing

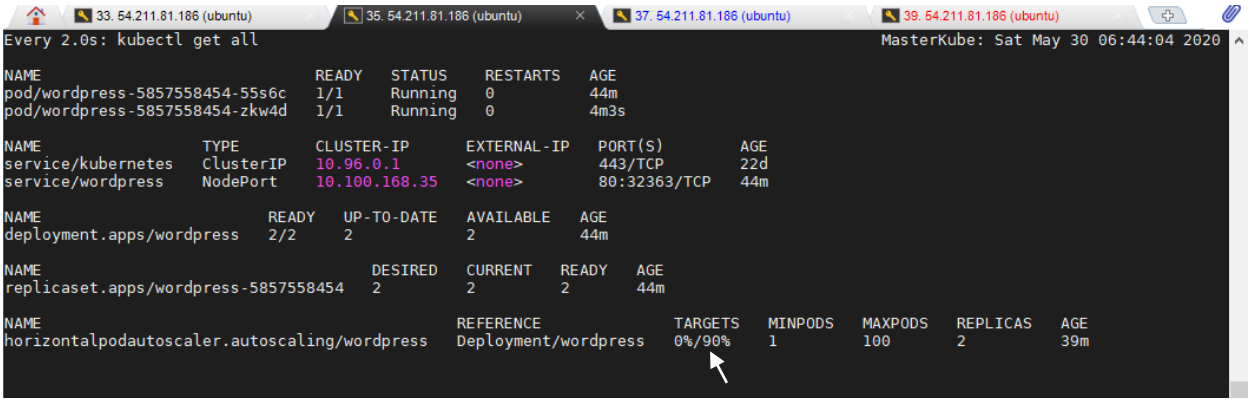


Gambar 4-3-1-4 Proses Load Balancing memerlukan waktu 62s

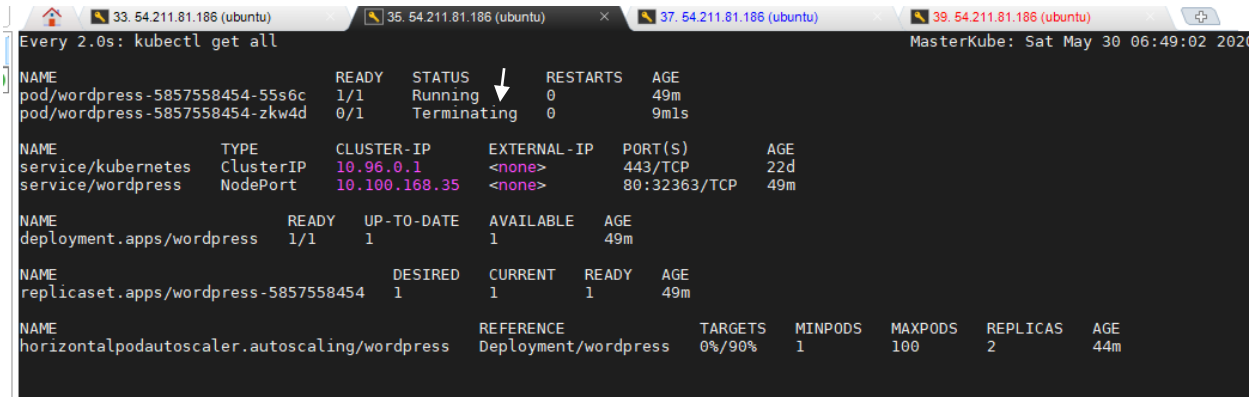


Gambar 4-3-1-5 Cpu Utilzation Ketika sudah Load balancing

4.3.2 Hasil Scaling Down



Gambar 4-3-2-1 Layanan sudah tidak di gunakan



Gambar 4-3-2-2 Pod (Container) di matikan

4.3.3 Analisis Scaling up

Berdasarkan skenario sebelumnya yaitu *Load Testing* dapat di peroleh data saat 10000 user telah di *Generate* oleh *Request Generator. CPU Utilization Container Orchestration* akan mencapai kapasistas 90% yang dimana 90% ini adalah ambang atas optimal untuk melakukan *scaling* [19] (*Gambar 4-3-1-1*) dan *Horizontal Pod auto scaler* (HPA) akan membuat Pod (Container) baru (*Gambar 4-3-1-1*) pada node yang lain tujuan nya untuk membagi sebagian beban yang di tampung pada node pertama (*Load balancing*).

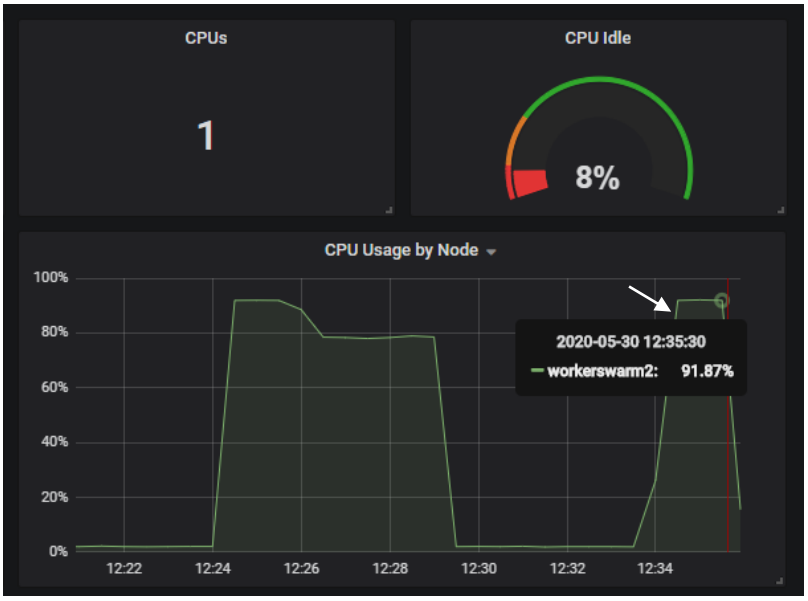
Pembuatan Pod (Container) baru pada node yang lain memerlukan waktu *3s* (3 detik).(*Gambar 4-3-1-2*) . pada *(Gambar 4-3-1-3)* kondisi sebelum terjadi *Load Balancing* beban pada node masih di tampung pada node *workerkube2* yaitu 93% . Pembagian beban dari node *workerkube2* ke *workerkube* (*Load balancing*) memerlukan waktu yaitu *62s* (*62 detik*). (*Gambar 4-3-1-4*). Pada (*Gambar 4-3-1-5*) setelah *62s* (62 detik) Horizontal pod auto scaler melakukan Load balancing pada node workerkube.

4.3.4 Analisis Scaling Down

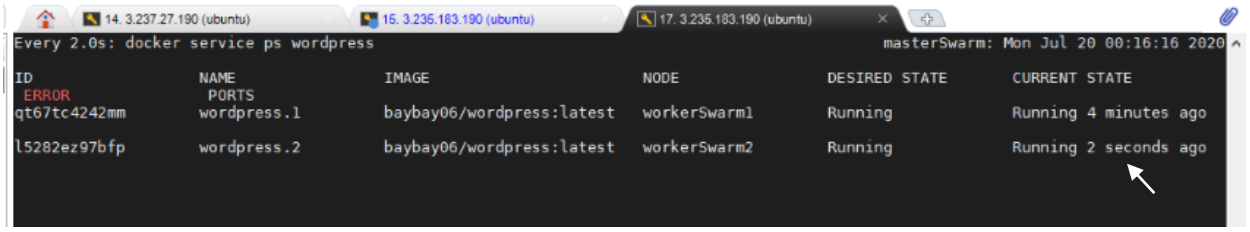
Saat Reqeust Generator selesai meng-*Generate user* . *Pod (Container)* atau layanan akan kembali menuju 0 % . *(Gambar 4-3-2-1)* dan *CPU Utilization* dari setiap node akan kembali normal. *Horizaontal Pod auto scaler (HPA)* akan menghapus *Pod (Container)* yang baru pada node lain dimana di butuhkan waktu untuk menghapus *Pod* (*Container*) tersebut yaitu *4m 51s*. Nilai *default* dari *system Kubernetes* untuk *downscale* atau menghapus *Pod (Container)* yang baru pada node lain adalah 5 *minute* 0 *s*[16].

4.4 Scaling up Scaling Down Docker Swarm

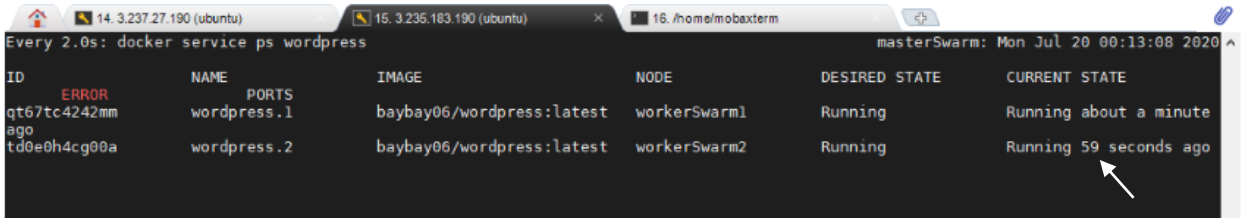
4.4.1 Hasil Scaling up



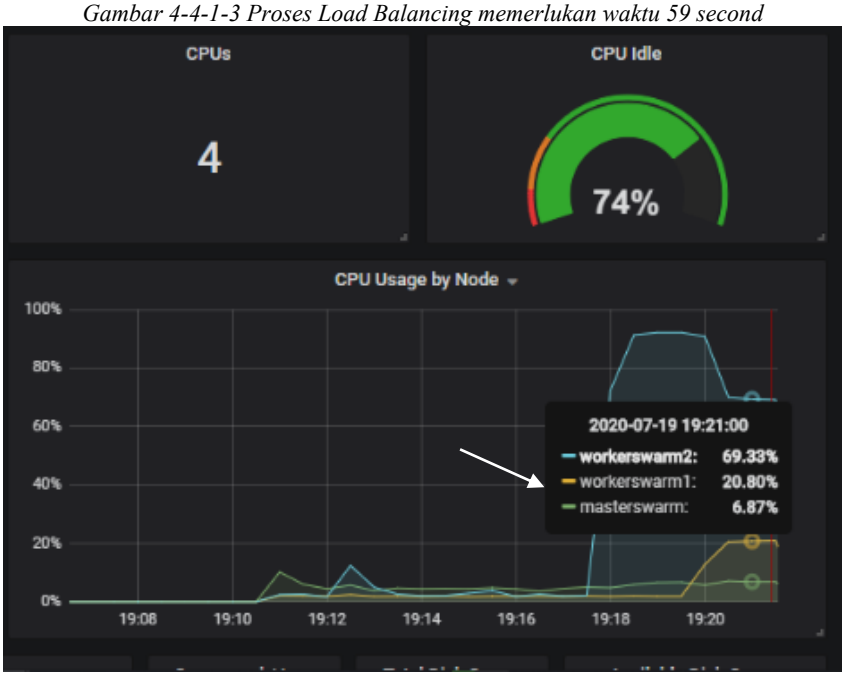
Gambar 4-4-1-1 CPU Utilization sebelum Load Balancing



Gambar 4-4-1-2 Container di buat dalam 2 second

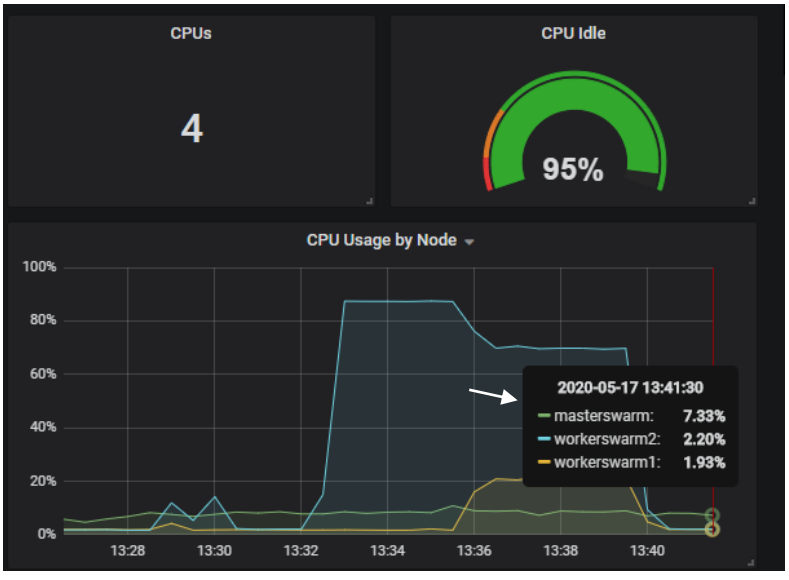


Gambar 4-4-1-3 Proses Load Balancing memerlukan waktu 59 second

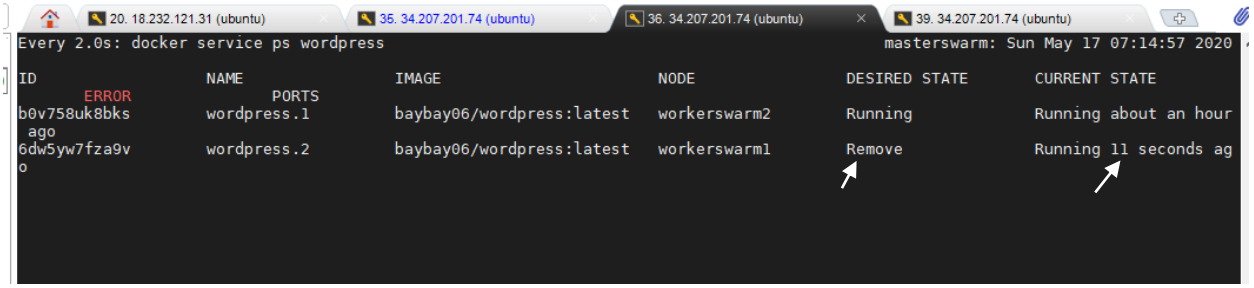


Gambar 4-4-1-4 Cpu Utilization setelah Load Balancing

4.4.2 Hasil Scaling Down



Gambar 4-4-2-1 Cpu Utilization ketika layanan sudah tidak di gunakan



Gambar 4-4-2-2 Container di matikan ketika layanan sudah tidak di gunakan

4.3.3 Analisis Scaling Up

Berdasarkan skenario sebelumnya yaitu Load Testing dapat di peroleh data saat 10000 user telah di *Generate* oleh *Request Generator. CPU Utilization Container Orchestration* akan mencapai kapasistas 90% yang dimana 90% ini adalah ambang atas optimal untuk melakukan *scaling* [19] (*Gambar 4-4-1-1*) . Berbeda dengan *Kubernetes Docker Swarm* tidak memiliki fitur *auto scaling* seperti *Horizontal Pod auto scaler* pada *Kubernetes.* Pembuatan *(Container)* pada node lain di lakukan manual , bertujuan nya untuk membagi sebagian beban yang di tampung pada node pertama (*Load balancing*).

Pembuatan *(Container)* baru saat *CPU Utilization* mencapai 90% [19] memerlukan waktu *2s (2 second).* (*Gambar 4-4-1-2*) .Pembagian beban dari node *workerswarm2* ke *workerswarm1* (*Load balancing*) memerlukan waktu yaitu *59second* (*59 second*). (*Gambar 4-4-1-3*). Pada gambar (*Gambar 4-4-1-3*) adalah pembagian beban pada node yang sudah di *Load Balancing.*

4.3.4 Analisis Scaling Down

Saat Reqeust Generator selesai meng-*Generate user* . *(Container)* atau *CPU Utilization* dari setiap node akan kembali normal. (*Gambar 4-4-2-1*). *(Container)* yang baru pada node *workerswarm1* akan di hapus secara manual karena sudah tidak di gunakan dan memerlukan waktu selama 11s *(11 second).* (*Gambar 4-4-2-2*)

4.4 Ringkasan

Berikut adalah ringkasan dari skenario *Load Testing* dan waktu *Scaling* up dan waktu *Scaling* down rata-rata 10x percobaan

4.4.1 Load Testing

Tabel 3-2.1 Load Testing

User	Kubernetes	Docker Swarm
2.000	29,47965	28,95117
5.000	64,02291	61,21728
10.000	94,20372	92,98229

4.4.2 Scaling Up

Tabel 3-2.1 Scaling Up Scaling Down

Method	Kubernetes	Docker Swarm
Container Creating Time	1,6 second	1,5 second
Load Balancing	61,2 second	55,8 second

Analisis Scaling up Kubernetes

Berdasarkan skenario sebelumnya yaitu Load Testing dapat di peroleh data saat 10000 user telah di *Generate* oleh *Request Generator. CPU Utilization Container Orchestration* akan mencapai kapasistas 90% yang dimana 90% ini adalah ambang atas optimal untuk melakukan *scaling* [19] dan *Horizontal Pod auto scaler* (HPA) akan membuat Pod (Container) baru pada node yang lain tujuan nya untuk membagi sebagian beban yang di tampung pada node pertama (*Load balancing*).

Pembuatan Pod (Container) baru pada node yang lain memerlukan waktu dengan rata rata *1,6s* (1,6 detik). sebelum terjadi *Load Balancing* beban pada node masih di tampung pada node *workerkube2* yaitu 93% . Pembagian beban dari node *workerkube2* ke *workerkube* (*Load balancing*) memerlukan waktu dengan rata rata yaitu *61,2s* (*61,2 detik*). Setelah *62s* (62 detik) Horizontal pod auto scaler melakukan Load balancing pada node workerkube .

Analisis Scaling Up Docker Swarm

Berdasarkan skenario sebelumnya yaitu Load Testing dapat di peroleh data saat 10000 user telah di *Generate* oleh *Request Generator. CPU Utilization Container Orchestration* akan mencapai kapasistas 90% yang dimana 90% ini adalah ambang atas optimal untuk melakukan *scaling* [19] . Berbeda dengan *Kubernetes Docker Swarm* tidak memiliki fitur *auto scaling* seperti *Horizontal Pod auto scaler* pada *Kubernetes.* Pembuatan *(Container)* pada node lain di lakukan manual , bertujuan nya untuk membagi sebagian beban yang di tampung pada node pertama (*Load balancing*).

Pembuatan *(Container)* baru saat *CPU Utilization* mencapai 90% [19] memerlukan waktu dengan rata rata *1,5 (1,5 second).* .Pembagian beban dari node *workerswarm2* ke *workerswarm1* (*Load balancing*) memerlukan waktu yaitu *2min* (*2 minute*).

4.4.3 Scaling Down

Tabel 3-2.1 Scaling Down

Method	Kubernetes	Docker Swarm
Delete Container	4 minite 49 second	11,4 second

Analisis Scaling Down Kubernetes

Saat Reqeust Generator selesai meng-*Generate user* . *Pod (Container)* atau layanan akan kembali menuju 0 % . *CPU Utilization* dari setiap node akan kembali normal. *Horizaontal Pod auto scaler (HPA)* akan menghapus *Pod (Container)* yang baru pada node lain dimana di butuhkan waktu untuk menghapus *Pod* (*Container*) tersebut yaitu dengan rata rata waktu *4m 49s*. Nilai *default* dari *system Kubernetes* untuk *downscale* atau menghapus *Pod (Container)* yang baru pada node lain adalah 5 *minute* 0 *s*[16].

Analisis Scaling Down Docker Swarm

Saat Reqeust Generator selesai meng-*Generate user* . *(Container)* atau *CPU Utilization* dari setiap node akan kembali normal. *(Container)* yang baru pada node *workerswarm1* akan di hapus secara manual karena sudah tidak di gunakan dan memerlukan dengan rata rata waktu 11,4s *(11,4 second).* (*Gambar 4-4-2-2*)

5. Kesimpulan

Hasil analisis dari seluruh pengujian yang dilakukan dalam penelitian tugasakhir ini dapat menarik kesimpulan sebagai berikut:

5.1 Load Testing

Berdasarkan hasil yang telah di dapat Kubernetes memakan lebih banyak resource *Cpu Utilization* yaitu pada 10000 user Kubernetes memakan resource dengan rata rata 94,20 % sedangkan pada *Docker Swarm* 92,28% di karenakan di dalam Kubernetes sendiri memiliki *system* yang kompleks terutama komponen komponen khusus seperti API,Etcd,Scheduler,Controller manager untuk menjalankan Container . Sementara di dalam *Docker Swarm* hanya memiliki komponen Swarm Manager dan *Docker Daemon* saja.

5.2 Waktu Scaling Up Scaling Down

Berdasarkan hasil yang telah di dapat Scaling up di dalam Kubernetes lebih di unggulkan karena penskalaan otomatis tetapi dari segi *Load Balancing Docker Swarm* lebih cepat yaitu dengan waktu rata rata 55,8 *second* sementara *Kubernetes* 61,2 *second* .

Untuk *Scaling Down Docker Swarm* di unggulkan dari segi menghapus Container. Di karenakan penghapusan di lakukan manual yaitu dengan waktu rata-rata 11,4 *second*. Meskipun *Kubernetes* terlihat lebih lama dalam menghapus tapi di dalam *Kubernetes* terdapat penghapusan Container otomatis yaitu dengan waktu rata rata 4 *minute* 49 *second*. Nilai *default* dari *system Kubernetes* untuk *downscale* atau menghapus *Pod (Container)* yang baru pada node lain adalah 5 *minute* 0 *s*[16].

Berdasarkan data-data tersebut di atas *Kubernetes* lebih cocok di gunakan untuk perusahaan yang berhubungan dengan user yang tidak dapat di prediksi seperti *e-commerce* yaitu *Lazada*, *Shopee*, *Tokopedia*, *Olx*, *Amazon*. Karena *Kubernetes* memiliki *auto scaler* yaitu *Horizontal pod auto scaler*. Sementara *Docker Swarm* lebih cocok di gunakan untuk perusahaan yang berhubungan dengan user yang dapat di predikisi seperti internal perusahaan dengan jumlah karyawan tertentu . Karena *Docker swarm* tidak memiliki *auto scaler* seperti *Kubernetes*.

5.3 Saran

Penelitian ini tentunya masih terdapat kekurangan dan mampu di eksplorasi lagi lebih lanjut. Oleh karena itu, penulis memberikan saran yaitu

1. Proses *Scaling* dapat dikembangkan dengan menggabungkan pemanfaatan metrik *Cpu Utilization* dan pemanfaatan *Memori* sehingga proses Scaling dapat lebih baik .

2. Membandingkan Container Orchestration dengan menggabungkan proses Scaling dengan proses lain sehingga mendapatkan Container Orchestration manakah yang terbaik.

Daftar Pustaka

[1] V. Medel, O. Rana, J. Á. Bañares, and U. Arronategui, "Adaptive application scheduling under interference in Kubernetes," in *Proceedings of the 9th International Conference on Utility and Cloud Computing - UCC '16*, 2016, pp. 426–427.

[2] J. Shah and D. Dubaria, "Building modern clouds: Using docker, kubernetes google cloud platform," *2019 IEEE 9th Annu. Comput. Commun. Work. Conf. CCWC 2019*, pp. 184–189, 2019.

[3] B. Kostadinov, M. Jovanov, and E. Stankov, "Cost-effective Website Failover through a CDN Network and Asynchronous Replication," no. July, pp. 6–8, 2017.

[4] M. Rexa and M. Bella, "Web Server Load Balancing Based On Memory Utilization Using Docker Swarm," *2018 Int. Conf. Sustain. Inf. Eng. Technol.*, pp. 220–223, 2018.

[5] S. Taherizadeh and V. Stankovski, "Dynamic Multi-level Auto-scaling Rules for Containerized Applications," *Comput. J.*, vol. 62, no. 2, pp. 174–197, Feb. 2019.

[6] "Kubernetes - Reviews, Pros & Cons | Companies using Kubernetes." [Online]. Available: https://stackshare.io/kubernetes. [Accessed: 20-Jun-2020].

[7] "Docker Swarm - Reviews, Pros & Cons | Companies using Docker Swarm." [Online]. Available: https://stackshare.io/docker-swarm. [Accessed: 20-Jun-2020].

[8] X. L. Xie, P. Wang, and Q. Wang, "The performance analysis of Docker and rkt based on Kubernetes," *ICNC-FSKD 2017 - 13th Int. Conf. Nat. Comput. Fuzzy Syst. Knowl. Discov.*, pp. 2137–2141, 2018.

[9] V. G. da Silva, M. Kirikova, and G. Alksnis, "Containers for Virtualization: An Overview," *Appl. Comput. Syst.*, vol. 23, no. 1, pp. 21–27, May 2018.

[10] M. I. Djomi, I. R. Munadi, and R. M. Negara, "ANALISIS PERFORMANSI NETWORK FUNCTION VIRTUALIZATION PADA CONTAINERS MENGUNAKAN DOCKER PERFORMANCE ANALYSIS OF NETWORK FUNCTION VIRTUALIZATION ON CONTAINERS USING DOCKER Universitas Telkom | 2," vol. 5, no. 2, pp. 1974–1981, 2018.

[11] A. Modak, S. D. Chaudhary, P. S. Paygude, and S. R. Ldate, "Techniques to Secure Data on Cloud: Docker Swarm or Kubernetes?," in *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*, 2018, no. Icicct, pp. 7–12.

[12] C. Chang and S. Yang, "A Kubernetes-Based Monitoring Platform for Dynamic Cloud Resource Provisioning," 2017.

[13] N. Naik, "Building a virtual system of systems using docker swarm in multiple clouds," *ISSE 2016 - 2016 Int. Symp. Syst. Eng. - Proc. Pap.*, pp. 7–9, 2016.

[14] M. Song, C. Zhang, and E. Haihong, "An Auto Scaling System for API Gateway Based on Kubernetes," in *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, 2018, vol. 2018-Novem, pp. 109–112.

[15] "Komponen-Komponen Kubernetes - Kubernetes." [Online]. Available: https://kubernetes.io/id/docs/concepts/overview/components/. [Accessed: 27-May-2020].

[16] "Horizontal Pod Autoscaler - Kubernetes." [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. [Accessed: 27-Oct-2019].

[17] "GitHub - kubernetes-sigs/metrics-server: Cluster-wide aggregator of resource usage data." [Online]. Available: https://github.com/kubernetes-sigs/metrics-server. [Accessed: 27-May-2020].

[18] "High Availability and Horizontal Scaling with Docker Swarm." [Online]. Available: https://medium.com/brian-anstett-things-i-learned/high-availability-and-horizontal-scaling-with-docker-swarm-76e69845825e. [Accessed: 10-Dec-2019].

[19] F. Al-Haidari, M. Sqalli, and K. Salah, "Impact of CPU Utilization Thresholds and Scaling Size on Autoscaling Cloud Resources," in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, 2013, vol. 2, pp. 256–261.

Lampiran

2000 user

Jumlah Pengujian	Persentase %	
	Kubernetes	Docker Swarm
1	29,0666	28,3173
2	28,9811	28,3188
3	30,9333	28,3078
4	28,6745	21,7674
5	28,4251	30,4512
6	29,5721	28,3663
7	31,1430	28,0666
8	30,8576	29,2768
9	28,4273	30,5431
10	29,8749	28,7641

5000 user

Jumlah Pengujian	Persentase %	
	Kubernetes	Docker Swarm
1	58,4489	56,5236
2	63,1079	63,7383
3	65,2050	57,9871
4	57,8976	67,8765
5	66,0758	48,7833
6	67,9629	67,0551
7	68,8436	67,2416
8	49,9430	67,2706
9	67,9824	58,9076
10	64,7620	56,7891

10000 user

Jumlah Pengujian	Persentase %	
	Kubernetes	Docker Swarm
1	86,2141	80,0051
2	94,2230	95,5203
3	95,2147	95,5389
4	94,3356	95,5182
5	95,2160	84,6333
6	95,3425	95,6027
7	95,3864	95,6231
8	95,3190	95,6703
9	95,3987	95,5990
10	95,3872	95,712

Container Creating time

Jumlah Pengujian	Second	
	Kubernetes	Docker Swarm
1	3 s	2 s
2	1 s	1 s
3	2 s	2 s
4	2 s	1 s
5	1 s	1 s
6	2 s	2 s
7	1 s	1 s
8	2 s	2 s
9	1 s	2 s
10	1 s	1 s

Load Balancing

Jumlah Pengujian	Second	
	Kubernetes	Docker Swarm
1	62 s	54 s
2	61 s	56 s
3	61 s	58 s
4	63 s	59 s
5	61 s	56 s
6	60 s	55 s
7	61 s	56 s
8	61 s	55 s
9	62 s	54 s
10	60 s	55 s

Delete Container

Jumlah Pengujian	Second	
	Kubernetes	Docker Swarm
1	4 min 51 s	11 s
2	4 min 46 s	12 s
3	4 min 48 s	11 s
4	4 min 52s	11 s
5	4 min 51 s	11 s
6	4 min 46 s	12 s
7	4 min 47 s	12 s
8	4 min 51 s	11 s
9	4 min 53 s	12 s
10	4 min 45s	11 s