ANALISIS PERFORMANSI PROSES MIGRASI DENGAN METODE SELF HEALING DAN SCHEDULING PADA CONTAINER ORCHESTRATION

Yazri Pahlevi[1], Vera Suryani[2], Siti Amatullah Karimah[3]

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

Abstrak

***Cloud computing* merupakan sebuah teknologi yang menjadikan internet sebagai pusat pengelolaan data dan aplikasi. Teknologi ini sudah sangat banyak digunakan dalam sebuah perusahaan untuk membuat sistem dan menjalankan bisnis mereka. *Cloud computing* yang memiliki arsitektur *container* akan menampung semua layanan dalam satu sistem. Jika suatu saat sistem tersebut mengalami masalah seperti *system down* maka semua layanan pada sistem tersebut tidak dapat diakses, otomatis pengguna atau perusahaan tidak dapat melakukan kegiatannya. Oleh karena itu dibutuhkan solusi untuk mengatasi masalah tersebut yaitu dengan proses memindahkan suatu layanan dari satu sistem ke sistem yang lain, proses ini disebut migrasi, dan dengan menggunakan *container orchestration* solusi tersebut dapat berjalan dengan cepat karena dalam *container orchestration* terdapat controller yang memungkinkan perpindahan layanan dari satu *container* ke *container* yang lain. Proses migrasi di *container orchestration* akan dilakukan dengan menggunakan metode *Self Healing.* Metode ini yang akan membantu proses migrasi dalam *container orchestration*. Penelitian ini bertujuan untuk membandingkan *container orchestration* mana yang lebih cepat dalam proses migrasi. Melalui pengujian sebanyak sepuluh kali dapat disimpulkan bahwa Kubernetes lebih cepat dalam menjalankan proses migrasi dengan metode *self healing*.**

Kata kunci : ***Cloud Computing, Container Orchestration, Kubernetes ,DockerSwarm, Migration, Self Healing***

Abstract

Cloud computing is a technology that makes the internet the center of data and application management. This technology has been very widely used in a company to create systems and run their business. Cloud computing which has a container architecture will hold all services in one system. If one day the system experiences problems such as system down, all services on the system cannot be accessed, the user or company automatically cannot perform its activities. Therefore we need a solution to overcome these problems, namely by the process of moving a service from one cloud system to another, this process is called migration, and by using container orchestration the solution can run quickly because in the container orchestration there is a controller that allows movement services from one cloud system to another. The migration process in the container orchestration will be carried out using the Self Healing method. This method will help the migration process in container orchestration. This study aims to compare which container orchestration is faster in the migration process. Through testing as much as ten times it can be concluded that Kubernetes is faster in carrying out the migration process with the self healing method.

Keywords: Cloud Computing, Container Orchestration, Kubernetes, Dowcker Swarm, Migration, Self Healing

1. Pendahuluan

Latar Belakang

Cloud computing merupakan sebuah teknologi yang menjadikan internet sebagai pusat pengelolaan data dan aplikasi. Teknologi ini populer di banyak perusahaan untuk membuat sistem dan menjalankan bisnis mereka[1]. Dengan *container* sebagai arsitektur dari *cloud computing* maka semua layanan akan disimpan di satu tempat/*container*. *Container* ini bertujuan agar semua layanan yang dibuat akan lebih mudah untuk di *deploy* dan dapat lebih efisien untuk di pindahkan dari satu sistem ke sistem lainnya[2], jika dibandingkan dengan tanpa menggunakan *container*. Penggunaan dari *container* juga berpeluang besar untuk meningkatkan portabilitas, keamanan, dan otomatisasi[3].

Sistem dalam container tersebut suatu saat dapat mengalami *system down* seperti kasus dimana sistem tersebut sedang dalam *maintenance* atau terdapat *error*. Otomatis sistem layanan pada container tersebut tidak dapat diakses oleh pengguna, maka dari itu perlu adanya proses migrasi[4]. Dengan proses migrasi dan *container creating time* yang cepat dapat membuat nilai *availability* dari suatu layanan menjadi besar. Nilai *Availability* adalah nilai ketersediaan dari suatu layanan. Jika suatu layanan memiliki nilai *availability* yang besar, maka dapat dikategorikan sebagai layanan yang baik.

Proses migrasi akan memindahkan suatu sistem ke sistem lainnya. Penggunaan *container orchestration* akan sangat membantu mempercepat proses migrasi sistem dari satu *cloud* ke *cloud* yang lain[5]. Dengan menggunakan metode *self healing* pada proses migrasi, hal-hal performansi dalam mendeteksi anomali-anomali dapat ditangani pada *container orchetration*[6]. Anomali tersebut adalah *system down*, dimana sistem tersebut tidak dapat digunakan oleh pengguna karena sedang terjadi error atau sedang dalam *maintenance*. *Self healing* adalah sebuah

metode yang dalam beberapa waktu akan terus mengecek kesehatan dari sebuah sistem. Jika sistem tersebut sedang tidak dalam kondisi baik maka metode ini akan otomatis membuat cadangan untuk mengatasi masalah tersebut.

Penelitian ini bertujuan untuk membandingkan Kubernetes dan Docker Swarm dalam proses migrasi dengan metode *self healing* dan *scheduling*. Kedua *container orchestration* tersebut merupakan *container orchestration* yang sangat popular digunakan pada perusahaan besar seperti Google dan Ruang Guru yang menggunakan Kubernetes[13] serta perusahaan perbankan negara America Wells Fargo dan Capital One yang menggunakan Docker Swarm[14].

Topik dan Batasannya

Rumusan masalah dari penelitian ini adalah:

- Berapa lama waktu yang dibutuhkan saat proses migrasi?
- Bagaimana cara membuat sebuah layanan langsung dapat diakses sesaat setelah terjadi *system down*?

Tujuan

Tujuan dari penelitian ini adalah:

- Melakukan perbandingan waktu dalam proses migrasi dengan metode *self healing* dan *scheduling* dari dua jenis *container orchestration* yaitu Kubernetes dan Docker Swarm dengan perbandingan parameter *pod creating time* dan *container creating time*.
- Melakukan perbandingan performansi sistem berupa perbandingan *CPU usage* dan *memory usage* yang dibutuhkan dari proses *self healing* dan *scheduling* dari kedua *container orchestration*.

Organisasi Tulisan

Pada bab satu akan menjelaskan tentang abstrak, latar belakang, rumusan masalah, dan tujuan dari topik ini. Pada bab 2 akan membahas tentang penelitian yang terkait dengan topik ini. Skenario pengujian serta metode yang dipakai di penelitian ini akan dijelaskan di bab 3. Pada bab 4 akan dijelaskan evaluasi, dan kesimpulan akan dijelaskan pada bab 5.

2. Studi Terkait

2.1 Cloud Computing

Sebuah model yang memungkinkan akses jaringan *on demand* yang nyaman ke kumpulan sumber daya dan layanan komputasi yang dapat dikonfigurasi bersama. Sumber daya ini meliputi jaringan, server, penyimpanan, aplikasi dan layanan. Sumber daya ini dapat dengan cepat disediakan dan dirilis dengan upaya manajemen minimal atau interaksi penyedia layanan[7]. Cloud computing memiliki 3 layanan, yaitu SaaS (Software as a Service), Paas (Platform as a Service), Iaas (Infrastructure as a Service).

2.2 Kubernetes

Kubernetes adalah sebuah platform untuk mengelola banyak *container*. *Container* ini dapat berupa docker *container* atau *container* alternatif lainnya. Kubernetes akan mengelola dan membentuk jalur komunikasi antara *container-container* tersebut[8].

Kubernetes menyediakan sarana untuk mendukung penyebaran berbasis kontainer 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 skala keluar dari aplikasi dengan beban kerja yang berubah secara dinamis[9].

Pod adalah unit dasar di Kubernetes, unit terkecil dil dalam objek model Kubernetes yang dapat dibuat dan di *deploy*. Fungsinya adalah sebagai wadah dari satu atau beberapa *container*. Dan Kubernetes tidak mengelola *container* secara langsung melainkan mengelola pod tersebut.

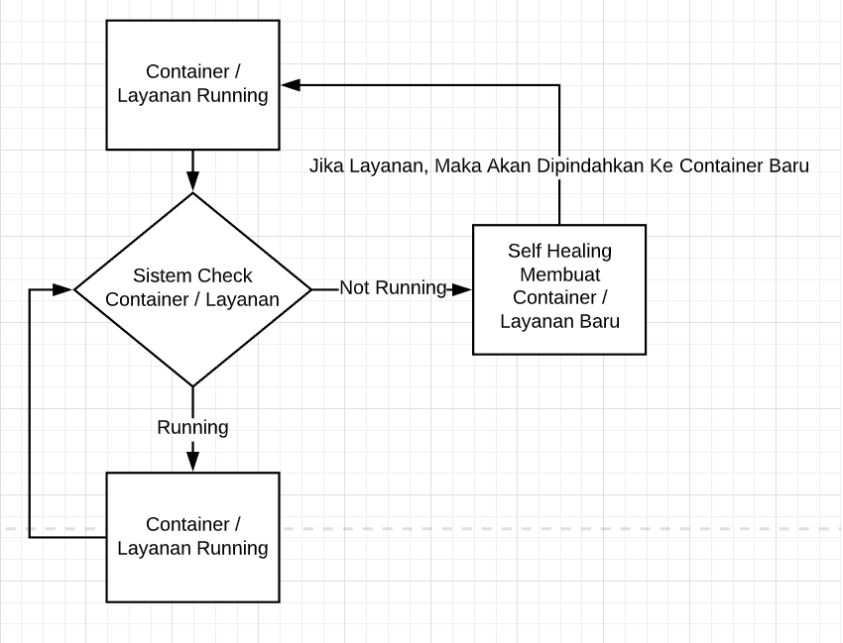
2.3 Docker Swarm

Docker adalah proyek sumber terbuka yang dirancang untuk mengembangkan, mengirim, dan menjalankan aplikasi di dalam virtualisasi berbasis *container*[10]. Mengurangi kemungkinan satu titik kegagalan dalam Arsitektur. Namun, mengelola beberapa *container* untuk menciptakan layanan tunggal adalah tugas yang menantang. Docker memecahkan ini masalah dengan menyediakan manajemen cluster kontainer yang disebut Docker Swarm[10].

2.4 Self Healing

Self healing adalah sebuah metode yang dalam beberapa waktu akan terus mengecek kesehatan dari sebuah sistem. Jika sistem tersebut sedang tidak dalam kondisi baik seperti sedang mengalami error, maka metode ini akan otomatis membuat cadangan untuk mengatasi masalah tersebut.

Metode S*elf Healing* digunakan untuk memberikan fleksibilitas, dan *high availability* untuk arsitektur *cloud* yang hirarki[11]. Dengan cara memonitor perilaku dan perubahannya konfigurasi atau arsitekturnya pada saat run time untuk meningkatkan atribut kualitasnya seperti kinerja, keandalan, dan keamanan di bawah kondisi operasi yang tidak pasti seperti kesalahan dan ancaman keamanan[12]. Proses *Self Healing* akan membuat sebuah pod atau layanan baru secara otomatis saat terdapat pod atau layanan yang mati.



Gambar 1. Flowchart Self Healing

2.5 Scheduler

Scheduler adalah komponen utama dari *container orchestration*, fitur ini membantu untuk memaksimalkan beban kerja sistem. *Scheduler* otomatis akan menghilangkan penyebaran layanan secara manual, jika tidak menggunakan *scheduler* maka akan menjadi tugas yang berat, terutama ketika layanan tersebut membutuhkan peningkatan dan penurunan secara horizontal.

Scheduler akan mendeteksi sebuah *container* baru dan akan diterapkan dengan sebuah node, sehingga *runtime* dapat menjalankan *container* tersebut. Untuk setiap pod yang terdeteksi oleh *scheduler*, *scheduler* akan bertanggung jawab untuk menemukan node terbaik untuk menjalankan pod tersebut.

Dalam sebuah cluster, node yang memenuhi persyaratan penjadwalan untuk pod disebut node *available*. Jika tidak ada node yang cocok, pod tetap tidak terjadwal sampai *scheduler* dapat menempatkannya.

2.6 Penelitian Terkait

Berikut ini adalah beberapa penelitian yang terkait dengan topik yang dibahas pada penelitian ini. Penelitian terkait ini dapat menjadi acuan dan dapat membantu dalam membuat topik yang sedang dibahas pada penelitian ini. Pada penelitian yang berjudul Techniques to Secure Data on Cloud: Docker Swarm or Kubernetes, menjelaskan tentang *container orchestration* mana yang lebih baik dalam mengamankan data, terutama pada Kubernetes dan Docker Swarm[7].

Selanjutnya pada penelitian di tahun 2016 yang berjudul "Moving to Autonomous and Self-Migrating *Container*s for Cloud Applications", menjelaskan tentang layanan yang disimpan pada sebuah *container*, hal ini bertujuan untuk mempermudah pemindahan layanan dari sebuah *cloud* ke *cloud* lainnya secara otomatis[2].

Lalu pada penelitian di tahun 2019 yang berjudul "Developing Self-Adaptive Microservice Systems: Challenges and Directions". Pada penelitian ini membahas tentang bagaimana cara memanajemen sebuah *container* dengan baik, yaitu dengan cara memonitor perilaku dan perubahannya konfigurasi atau arsitekturnya pada saat run time untuk meningkatkan atribut kualitasnya seperti kinerja, keandalan, dan keamanan di bawah kondisi operasi yang tidak pasti seperti kesalahan dan ancaman keamanan[12].

Pada penelitian ini akan membahas tentang perbandingan analisis performansi dari Kubernetes dan Docker Swarm dalam proses migrasi menggunakan metode *self healing.* Tujuan dari penelitian ini adalah untuk mengetahui *container orchestration* mana yang lebih baik dalam menangani *system down,* dengan cara menjalankan proses *self healing*.

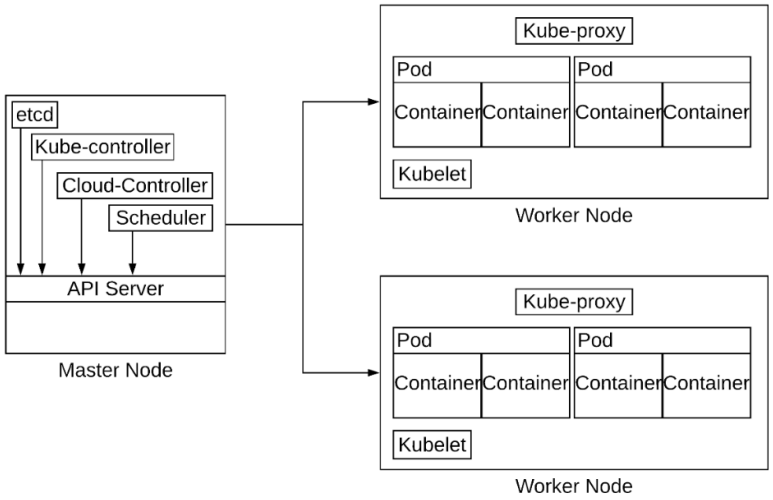
Judul Paper	Tahun	Sistem	Metode	Output
Moving to Autonomous and Self-Migrating Containers for Cloud Applications[2]	2016	*Container Based*	*Container Orchestration*	*Autonomous self migrating container*

Techniques to Secure Data on Cloud: Docker Swarm or Kubernetes?[7]	*2018*	*Container Based*	*Container Orchestration*	*Techniques to secure data by container orchestration*
The performance Analysis of Docker and Rkt Based on Kubernetes[6].	2019	*Container Based*	*Container Orchestration*	*Autonomic cloud management*

3. Perancangan Sistem

3.1 Topologi Sistem

Arsitektur Kubernetes dan Docker Swarm sebenarnya mirip, yaitu jika di Kubernetes terdapat *Master Node* dan *Worker Node*, sedangkan di Docker Swarm terdapat *Swarm Manager* dan *Swarm Node.* Berikut ini adalah topologi sistem dari Kubernetes.

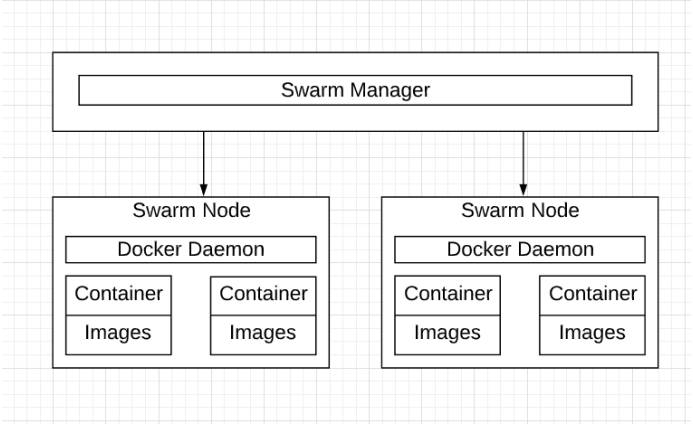


Gambar 2. Arsitektur Kubernetes

Komponen-komponen yang terdapat pada *Master Node Kubernetes* adalah *Kube-apiserver* sebagai komponen di master yang mengekspos API Kubernetes. Merupakan front-end dari kontrol plane Kubernetes. Komponen ini didesain agar dapat di-scale secara horizontal. Lalu komponen *etcd* sebagai komponen untuk penyimpanan key value konsisten yang digunakan sebagai penyimpanan data klaster Kubernetes. Komponen *Kube-scheduler* adalah komponen yang bertugas mengamati pod yang baru dibuat dan belum di-assign ke suatu node dan kemudian akan memilih sebuah node dimana pod baru tersebut akan dijalankan. Lalu komponen *Kube-controller-manager* adalah 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. Komponen terakhir adalah *cloud-controller-manager* sebagai kontroler yang berinteraksi dengan penyedia layanan cloud. Kontroler ini merupakat fitur alfa yang diperkenalkan pada Kubernetes versi 1.6.

Komponen-komponen yang terdapat pada *Worker Node Kubernetes* adalah *kubelet* yang berfungsi sebagai Agen yang dijalankan pada setiap node di klaster dan bertugas memastikan kontainer dijalankan di dalam pod, *kube-proxy* berfungsi untuk 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 *container runtime* adalah perangkat lunak yang bertanggung jawab dalam menjalankan kontainer. Kubernetes mendukung beberapa *runtime*, diantaranya adalah Docker, containerd, cri-o, rktlet dan semua implementasi Kubernetes CRI (Container Runtime Interface).

Berikut ini adalah topologi sistem dari Docker Swarm.



Gambar 3. Arsitektur Docker Swarm

Komponen pada *Docker Swarm* adalah *Swarm Manager*/*Master node* dan *Swarm node*. Pada *Swarm Node* terdapat komponen seperti *Docker Daemon, Container, dan Images*.

Komponen pada *Master Node* menyediakan *control plane* bagi kluster. Komponen ini berperan dalam proses pengambilan secara global pada kluster (contohnya, mekanisme schedule), serta berperan dalam proses deteksi serta pemberian respons terhadap events yang berlangsung di dalam kluster (contohnya, penjadwalan pod baru apabila jumlah replika yang ada pada *replication controller* tidak terpenuhi). Komponen master dapat dijalankan di mesin manapun yang ada di kluster. Meski begitu, untuk memudahkan proses yang ada, script inisiasi awal yang dijalankan biasanya memulai komponen master pada mesin yang sama, serta tidak menjalankan kontainer bagi pengguna di mesin ini.

Komponen pada *Worker Node* ini ada pada setiap node, fungsinya adalah melakukan pemeliharaan terhadap pod serta menyediakan environment runtime bagi *Container Orchestration*.

3.2 System Requirement

Berikut ini adalah spesifikasi sistem dan perangkat keras dari Kubernetes dan Docker Swarm yang digunakan dalam pengerjaan penelitian ini:

1. Master

Komponen	Spesifikasi
Processor	2 vCPUs, Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
RAM	4 GB
HDD	16 GB

2. Worker

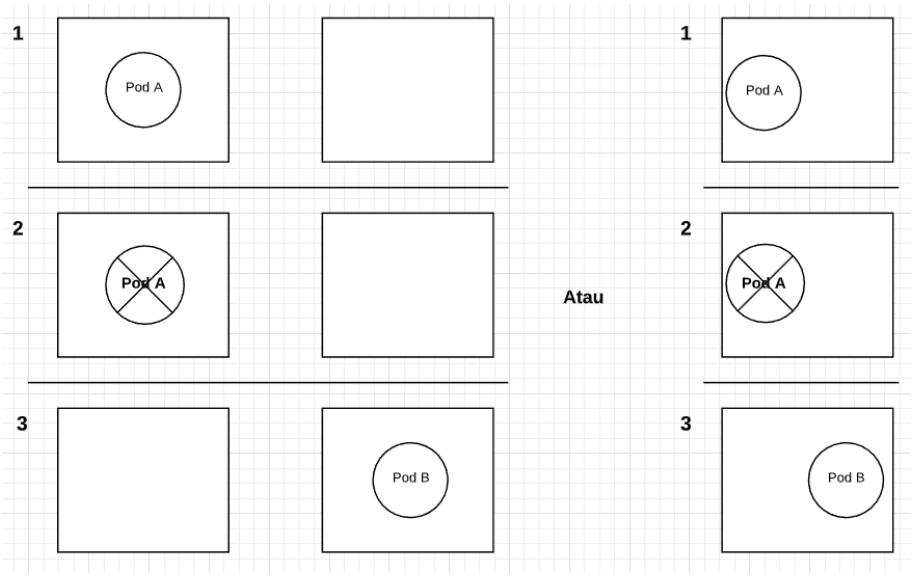
Komponen	Spesifikasi
Processor	1 vCPUs, Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
RAM	1 GB
HDD	16 GB

3.3 Skenario Pengujian

Pada penelitian ini container orchesrtration yang dipakai yaitu Kubernetes dan Docker Swarm akan berjalan pada masing-masing 3 virtual machine, virtual machine tersebut berupa satu node master dan 2 node worker. Pada virtual machine tersebut sudah terinstal docker container terlebih dahulu sebelum terinstalnya orchestration tersebut yaitu Kubernetes dan Docker Swarm. Setelah kedua container orchestration siap, maka layanan atau *service* sudah dapat di *deploy* di masing-masing container orchestration.

Skenario pengujian yang akan dilakukan adalah dengan menggunakan cara *self healing test* dan *scheduling*.
Skenario *self healing test:*

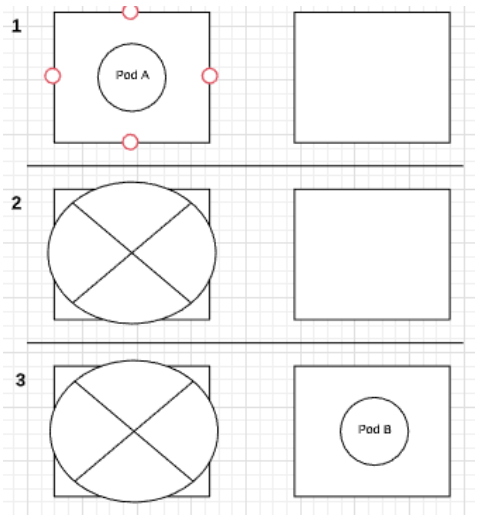
1. Membuat pod atau layanan baru
2. Menghapus/*terminate* pod atau layanan
3. Sistem otomatis akan melakukan proses *self healing* terhadap pod atau layanan yang mati dengan cara membuat pod atau layanan yang baru
4. Melakukan analisis dengan menggunakan parameter *pod creating time* dan *container creating time*
5. Melakukan analisis dalam penggunaan CPU dan *memory* dari tiap *container orchestration*



Gambar 4. Skema Self Healing

Skenario *scheduling*:

1. Membuat pod atau layanan baru
2. Mematikan atau membuat status sebuah node menjadi *Scheduling disable* atau *Drain.*
3. Node tersebut tidak dapat diterapkan tugas baru
4. Sistem otomatis akan memigrasikan pod atau layanan tersebut dari node yang mati ke node yang aktif
5. Melakukan analisis dengan menggunakan parameter *pod creating time* dan *container creating time*
6. Melakukan analisis dalam penggunaan CPU dan *memory* dari tiap *container orchestration*



Gambar 5. Skema Scheduling

Skenario *scheduling* sebenarnya juga menjalankan proses *self healing*, tetapi lebih berfokus pada node, dimana jika sebuah node mati atau di *terminate* maka *scheduling* akan otomatis memindahkan layanan pada node tersebut ke node yang tersedia. Sedangkan di skenario *self healing* berfokus pada layanan dan *container*, dimana jika sebuah layanan atau *container* mati, maka *self healing* akan otomatis membuat layanan atau *container* yang baru.

Parameter yang diukur pada penelitian ini adalah *pod creating time* dan *container creating time*. Semakin sedikit nilai dari parameter tersebut maka akan semakin baik *container orchestration* tersebut.

4. Evaluasi

4.1 Self Healing

Tujuan skenario *self healing* adalah untuk menguji performansi *container orchestration* dalam hal *self healing container* maupun layanan. Di skenario ini *container orchestration* akan melakukan *self healing* terhadap *container* atau layanan yaitu dengan cara membuat *container* atau layanan yang baru untuk mengganti *container* atau layanan yang mati atau di *terminate* oleh admin.

Dengan parameter *pod creating time* dan *container creating time* dapat menjadi acuan untuk menganalisis *container orchestration* mana yang lebih baik dalam skenario *self healing*

4.1.1 Hasil Self Healing

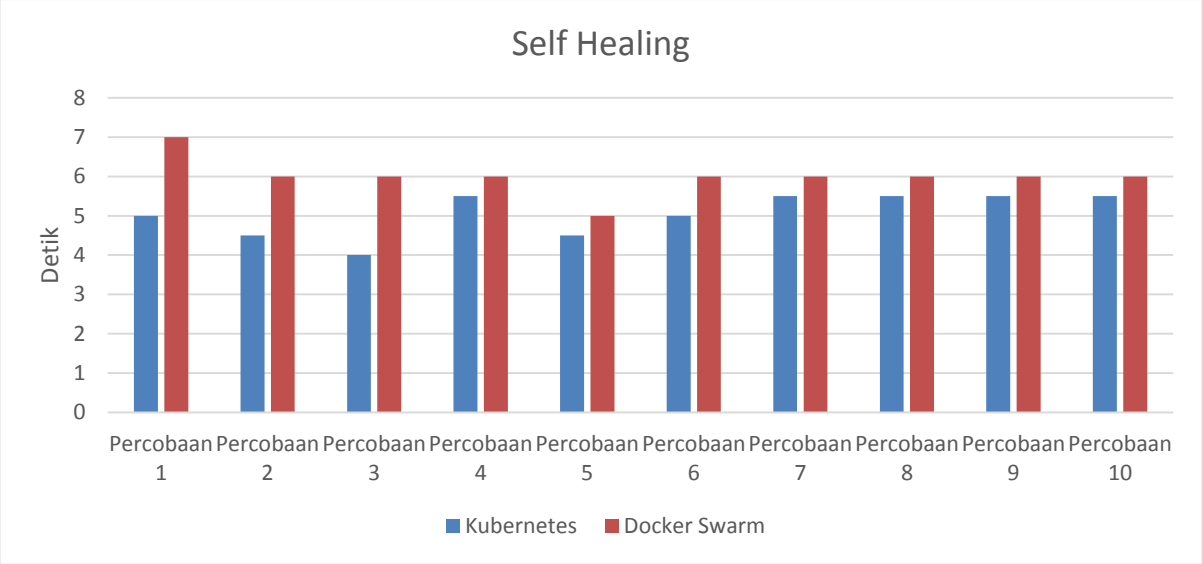


Table 1. Chart Table Self Healing

4.1.2 Hasil CPU Usage Self Healing

KUBERNETES SELF HEALING																					
	percobaan1		percobaan2		percobaan3		percobaan4		percobaan 5		percobaan 6		percobaan 7		percobaan 8		percobaan 9		percobaan 10	
	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir
master	11%	11%	16%	16%	16%	17%	17%	17%	17%	16%	17%	16%	16%	16%	11%	11%	16%	16%	16%	17%
user37	3%	4%	4%	4%	4%	3%	3%	10%	4%	3%	4%	4%	3%	4%	3%	7%	4%	8%	4%	4%
worker2	5%	3%	4%	3%	2%	3%	3%	2%	3%	3%	3%	3%	2%	9%	4%	3%	3%	6%	3%	8%

Table 2. Kubernetes CPU Usage Table

DOCKER SWARM SELF HEALING																				
Percobaan1		Percobaan2		Percobaan 3		Percobaan 4		Percobaan5		Percobaan6		Percobaan7		Percobaan8		Percobaan9		Percobaan10	
Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir
1.03%	34.08%	0.06%	34.18%	0.05%	27.78%	0.07%	27.48%	0.08%	44.11%	0.06%	37.42%	0.07%	32.81%	0.07%	40.01%	0.08%	32.13%	0.06%	35.63%

Table 3. Docker Swarm CPU Usage Table

4.1.3 Hasil Memory Usage Self Healing

KUBERNETES SELF HEALING																				
	Percobaan1		Percobaan 2		Percobaan 3		Percobaan4		Percobaan5		Percobaan6		Percobaan7		Percobaan8		Percobaan9		Percobaan10	
	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir
kuber11	1763Mi	1756Mi	1758Mi	1763Mi	1767Mi	1765Mi	1766Mi	1766Mi	1754Mi	1751Mi	1756Mi	1754Mi	1754Mi	1758Mi	1755Mi	1770Mi	1755Mi	1756Mi	1755Mi	1755Mi
user37	477Mi	575Mi	394Mi	520Mi	518Mi	447Mi	446Mi	528Mi	551Mi	455Mi	502Mi	592Mi	594Mi	540Mi	498Mi	516Mi	537Mi	619Mi	620Mi	549Mi
worker2	541Mi	486Mi	512Mi	454Mi	453Mi	514Mi	516Mi	453Mi	453Mi	549Mi	548Mi	476Mi	476Mi	553Mi	551Mi	491Mi	551Mi	476Mi	477Mi	556Mi

	Percobaan1		Percobaan 2		Percobaan 3		Percobaan4		Percobaan5		Percobaan6		Percobaan7		Percobaan8		Percobaan9		Percobaan10	
	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir
kuber11	45%	45%	45%	45%	45%	45%	45%	45%	45%	45%	45%	45%	45%	45%	45%	46%	45%	45%	45%	45%
user37	53%	64%	44%	58%	58%	50%	50%	59%	62%	51%	56%	66%	67%	61%	56%	58%	60%	69%	70%	62%
worker2	61%	54%	57%	51%	51%	58%	58%	51%	51%	62%	61%	53%	53%	62%	62%	55%	62%	53%	53%	62%

Table 4. Kubernetes Memory Usage Table

DOCKER SWARM SELF HEALING																			
	Percobaan1		Percobaan 2		Percobaan 3		Percobaan4		Percobaan5		Percobaan6		Percobaan7		Percobaan8		Percobaan9		Percobaan10	
	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir
	73.99MiB	72.96Mib	72.86Mib	73.11MiB	73MiB	73.06MiB	72.97MiB	72.96MiB	72.89MiB	73.91MiB	73.84MiB	73.02MiB	72.98MiB	73.02MiB	72.93MiB	73.04MiB	72.99MiB	73.05MiB	72.94MiB	73.07MiB
	Percobaan1		Percobaan 2		Percobaan 3		Percobaan4		Percobaan5		Percobaan6		Percobaan7		Percobaan8		Percobaan9		Percobaan10	
	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir
	7.51%	7.40%	7.39%	7.42%	7.41%	7.41%	7.40%	7.40%	7.40%	7.50%	7.49%	7.41%	7.41%	7.41%	7.40%	7.41%	7.41%	7.41%	7.40%	7.41%

Table 5. Docker Swarm Memory Usage Table

4.1.4 Analisis

Analisis dari skenario *self healing* dengan menggunakan parameter *pod creating time, container creating time*, dan *CPU Usage* adalah:

1. Waktu tercepat dari Kubernetes adalah 4 detik.
2. Waktu tercepat dari Docker Swarm adalah 5 detik.
3. Rata-rata waktu dari Kubernetes adalah 5,05 detik.
4. Rata-rata waktu dari Docker Swarm adalah 6 detik
5. CPU Usage maksimal dari Kubernetes adalah 17%
6. CPU Usage masksimal dari Docker Swarm adalah 44.11%
7. Rata-rata CPU Usage dari Kubernetes adalah 15.5%
8. Rata-rata CPU Usage dari Docker Swarm adalah 34.563%
9. Memory Usage maksimal dari Kubernetes adalah 70%
10. Memory Usage maksimal dari Docker Swarm adalah 7.51%
11. Memory Usage rata-rata dari Kubernetes adalah 55.9%
12. Memory Usage rata-rata dari Docker Swarm adalah 7.437%

Karena Kubernetes lebih cepat dalam menjalankan proses *self healing* otomatis penggunaan rata-rata CPU dan penggunaan maksimal CPU akan rendah, tetapi Kubernetes memiliki *memory usage* yang besar yaitu sebanyak 70%, hal ini dikarenakan oleh banyaknya komponen dalam Kubernetes. Penggunaan rata-rata CPU yang dihasilkan dari Kubernetes lebih kecil dari Docker Swarm dengan selisih sebanyak 19.063%. Dan selisih dari pemakaian CPU maksimal yang didapat dari Kubernetes juga lebih kecil dari Docker Swarm dengan selisih sebanyak 27.11%. Hal ini terjadi karena Kubernetes lebih cepat dalam menjalankan proses *self healing*.

4.2 Scheduling

Tujuan skenario *scheduling* adalah untuk menguji performansi *container orchestration* dalam proses *scheduling*. Skenario *scheduling* akan memigrasikan pod atau layanan dari node yang tidak aktif ke node yang aktif. Node yang tidak aktif ini memiliki status node *Scheduling disable* atau *Drain*.

Arti dari status node *Scheduling disable* adalah node tersebut sedang tidak dapat dilakukan *scheduling*, seperti meng-*assign pod, replicas*, dan *deployment* baru. Arti dari status node *Drain* adalah node tersebut tidak dapat diterapkan tugas baru. *Scheduler* yang sedang berjalan di dalam node tersebut juga akan dimatikan dan akan di migrasi ke node dengan status *Active.*

Dengan parameter *pod creating time* dan *container creating time* dapat menjadi acuan untuk menganalisis *container orchestration* mana yang lebih baik dalam skenario *scheduling*.

4.2.1 Hasil Scheduling

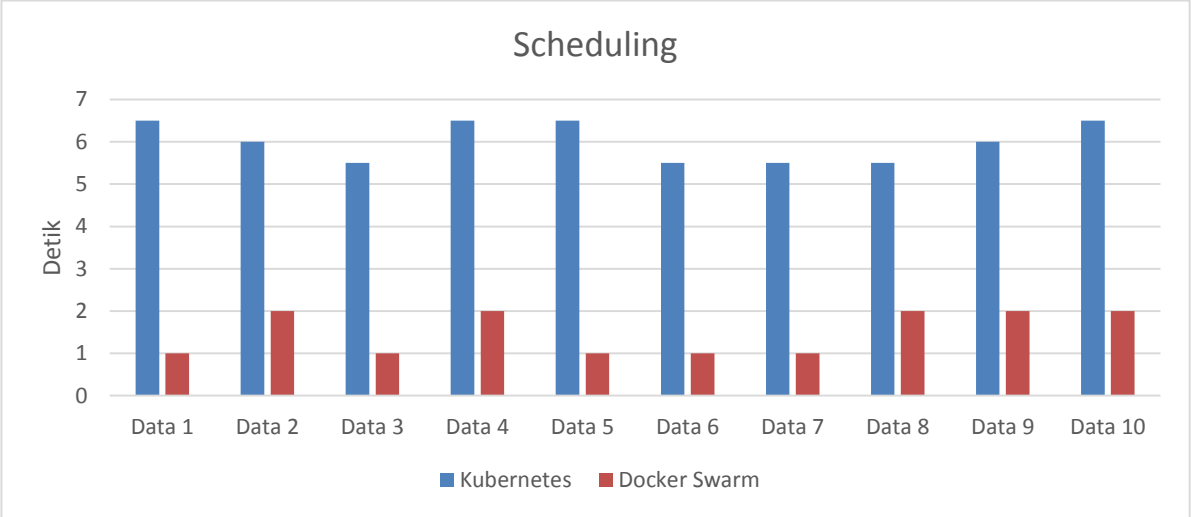


Table 6. Chart Table Scheduling

4.2.2 Hasil CPU Usage Scheduling

KUBERNETES SCHEDULING																				
	percobaan1		percobaan2		percobaan3		percobaan4		percobaan 5		percobaan 6		percobaan 7		percobaan 8		percobaan 9		percobaan 10	
	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir
master	4%	16%	13%	17%	17%	16%	16%	17%	16%	17%	19%	17%	16%	16%	17%	16%	16%	16%	16%	17%
user37	3%	6%	3%	2%	6%	3%	3%	2%	3%	2%	3%	2%	3%	3%	3%	2%	3%	2%	3%	4%
worker2	4%	13%	3%	5%	4%	3%	4%	4%	4%	4%	4%	4%	4%	4%	4%	5%	4%	4%	4%	15%

Table 7. Kubernetes CPU Usage Table

DOCKER SWARM SCHEDULING																				
Percobaan1		Percobaan2		Percobaan 3		Percobaan 4		Percobaan5		Percobaan6		Percobaan7		Percobaan8		Percobaan9		Percobaan10	
Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir
0.09%	16.52%	0.06%	27.67%	0.08%	17.04%	0.10%	31.69%	0.06%	17.32%	0.10%	23.63%	0.07%	27.75%	0.06%	22.09%	0.11%	17.83%	0.10%	30.49%

Table 8. Docker Swarm CPU Usage Table

4.2.3 Hasil Memory Usage Scheduling

KUBERNETES SCHEDULING																				
	Percobaan1		Percobaan 2		Percobaan 3		Percobaan4		Percobaan5		Percobaan6		Percobaan7		Percobaan8		Percobaan9		Percobaan10	
	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir
kuber11	1762Mi	1764Mi	1764Mi	1763Mi	1766Mi	1768Mi	1768Mi	1773Mi	1769Mi	1764Mi	1780Mi	1774Mi	1773Mi	1766Mi	1766Mi	1767Mi	1768Mi	1766Mi	1772Mi	1776Mi
user37	586Mi	515Mi	588Mi	514Mi	515Mi	590Mi	519Mi	592Mi	540Mi	466Mi	550Mi	474Mi	551Mi	474Mi	549Mi	476Mi	552Mi	477Mi	552Mi	541Mi
worker2	462Mi	538Mi	456Mi	535Mi	536Mi	461Mi	542Mi	463Mi	465Mi	541Mi	465Mi	540Mi	465Mi	540Mi	466Mi	541Mi	470Mi	542Mi	468Mi	522Mi
	Percobaan1		Percobaan 2		Percobaan 3		Percobaan4		Percobaan5		Percobaan6		Percobaan7		Percobaan8		Percobaan9		Percobaan10	
	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir
kuber11	45%	45%	45%	45%	45%	46%	46%	46%	46%	45%	46%	46%	46%	45%	45%	45%	45%	45%	46%	46%
user37	66%	58%	66%	58%	58%	66%	58%	66%	60%	52%	62%	53%	62%	53%	62%	53%	62%	53%	62%	61%
worker2	52%	60%	51%	60%	60%	52%	61%	52%	52%	61%	52%	61%	52%	61%	52%	61%	53%	61%	52%	58%

Table 9. Kubernetes Memory Usage Table

DOCKER SWARM SCHEDULING																				
Percobaan1		Percobaan 2		Percobaan 3		Percobaan4		Percobaan5		Percobaan6		Percobaan7		Percobaan8		Percobaan9		Percobaan10	
Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir
83.71MiB	73.66Mib	64.73MiB	73.38MiB	73.31MiB	72.86MiB	72.82MiB	73.95MiB	86.13MiB	73.04MiB	73.04MiB	73.19MiB	73.16MiB	72.94MiB	63.91MiB	73.27MiB	73.19MiB	73.05MiB	73.02MiB	73.27MiB
Percobaan1		Percobaan 2		Percobaan 3		Percobaan4		Percobaan5		Percobaan6		Percobaan7		Percobaan8		Percobaan9		Percobaan10	
Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir	Awal	Akhir
2.12%	7.47%	6.57%	1.86%	1.63%	7.39%	7.39%	1.87%	2.18%	7.41%	7.41%	1.86%	1.85%	7.40%	6.48%	1.86%	1.86%	7.41%	7.41%	1.86%

Table 10. Docker Swarm Memory Usage

4.2.4 Analisis

Analisis dari skenario *scheduling* dengan menggunakan parameter *pod creating time*, *container creating time*, dan *CPU Usage* adalah:

1. Waktu tercepat dari Docker Swarm adalah 1 detik
2. Waktu tercepat dari Kubernetes adalah 5,5 detik.
3. Rata-rata waktu Docker Swarm adalah 1,5 detik.
4. Rata-rata waktu Kubernetes adalah 6 detik.
5. CPU Usage maksimal dari Kubernetes adalah 19%
6. CPU Usage masksimal dari Docker Swarm adalah 31.69%
7. Rata-rata CPU Usage dari Kubernetes adalah 16.9%
8. Rata-rata CPU Usage dari Docker Swarm adalah 23.203%
9. Memory Usage maksimal dari Kubernetes adalah 66%
10. Memory Usage maksimal dari Docker Swarm adalah 7.47%
11. Memory Usage rata-rata dari Kubernetes adalah 56.2%
12. Memory Usage rata-rata dari Docker Swarm adalah 7.234%

Proses *scheduling* pada Kubernetes memakan waktu yang cukup lama dibandingkan Docker Swarm, dapat dilihat dari selisih rata-rata waktu untuk menjalankan proses *scheduling* yaitu sebanyak 58,5 detik. Dan *memory usage* yang dihasilkan oleh Kubernetes juga jauh lebih besar dari Docker Swarm yaitu sebanyak 66%, hal ini dikarenakan komponen yang terdapat pada Kubernetes lebih banyak daripada Docker Swarm.

Tetapi Kubernetes lebih sedikit dalam rata-rata penggunaan CPU daripada Docker Swarm, karena dalam proses *self healing* yang sebelumnya dilakukan Kubernetes sudah tercatat memiliki rata-rata penggunaan CPU yang lebih kecil. Hal ini dapat dilihat dari penggunaan rata-rata CPU yang dihasilkan dari

Kubernetes lebih kecil dari Docker Swarm dengan selisih sebanyak 6.303%.

5. Kesimpulan

Setelah melakukan analisis terhadap kedua *container orchestration* dalam melakukan proses *self healing* dan *scheduling* dengan menggunakan parameter *pod creating time, container creating time,* dan *CPU Usage.* Berdasarkan analisis dari sepuluh kali percobaan yang telah dilakukan, maka dapat diambil kesimpulan bahwa:

- Kubernetes lebih cepat dalam menangani proses *self healing,* dengan menggunakan parameter *pod creating time*, waktu tercepat yang didapat saat proses *self healing* adalah 4 detik.
- Docker Swarm lebih cepat dalam menangani proses *scheduling,* dengan menggunakan parameter *container creating time*, waktu tercepat yang didapat saat proses *scheduling* adalah 1 detik.
- Kubernetes jauh lebih ramah dalam pemakaian CPU. Untuk melakukan proses *self healing* maupun *scheduling,* rata-rata *CPU Usage* hanya sebesar 15.5% untuk proses *self healing* dan 16.9% untuk proses *scheduling.*
- Docker Swarm jauh lebih ramah dalam pemakaian *memory.* Untuk melakukan proses *self healing* maupun *scheduling,* rata-rata *memory usage* hanya sebesar 7.437% untuk proses *self healing* dan 7.234% untuk proses *scheduling*.

Dengan adanya kesimpulan diatas, dapat disimpulkan bahwa Kubernetes memiliki keunggulan dalam manajemen *resource* dan untuk menangani proses *self healing.* Sedangkan Docker Swarm lebih unggul dalam menangani proses *scheduling,* tetapi dengan catatan *Docker Swarm* memakan sangat banyak *CPU* sedangkan Kubernetes memakan sangat banyak *memory* dalam menjalankan proses *self healing* dan *scheduling.*

Dengan ini pengguna atau suatu perusahaan dapat menentukan *container orchestration* yang sesuai untuk membuat dan menjalankan sistem maupun proses bisnis mereka. Dengan pemilihan *container orchestration* yang tepat dapat membuat suatu perusahaan menjadi mudah untuk menangani kondisi *system down* dan memanajemen proses bisnis mereka dengan baik.

Daftar Pustaka

[1] M. Tvrdíková, "Effects of the transformation of company computer system on cloud computing services – a change in company management," vol. 8, pp. 1129-1132, 2016.

[2] D. S. Linthicum, "Moving to Autonomous and Self-Migrating Containers for Cloud Applications," pp. 6-9, 2016.

[3] D. Elliott, C. Otero, M. Ridley and X. Merino, "ACloud-AgnosticContainerOrchestratorforImprovingInteroperability," pp. 958-961, 2018.

[4] S. Nadgowda, S. Suneja, N. Bila and C. Isci , "Voyager: Complete Container State Migration," pp. 2137-2142, 2017.

[5] W. Fan, Z. Han, Y. Zhang and R. Wang, "A Locality Live Migration Strategy Based on Docker Containers," pp. 51-54, 2018.

[6] A. Samir and C. Pahl , "Self-Adaptive Healing for Containerized Cluster Architectures with Hidden Markov Models," pp. 68-73, 2019.

[7] A. Modak, . S. Prof., P. Prof. and S. Prof., "Techniques to Secure Data on Cloud: Docker Swarm or Kubernetes?," pp. 7-12, 2018.

[8] J. Shah and D. Dubaria , "Building Modern Clouds: Using Docker, Kubernetes & Google Cloud Platform," pp. 184-189, 2019.

[9] V. Medel , O. Rana, J. Á. Bañares and U. Arronategui , "Adaptive Application Scheduling under Interference in Kubernetes," pp. 426-427, 2016.

[10] M. R. M. Bella , M. Data and W. Yahya, "Web Server Load Balancing Based On Memory Utilization Using Docker Swarm," pp. 220-223, 2018.

[11] P. Stack, H. Xiong, D. Mersel, M. Makhloufi, G. Terpend and D. Dong, "Self-Healing in a Decentralised Cloud Management System," pp. 1-6, 2017.

[12] N. C. Mendonc¸a, P. Jamshidi, D. Garlan and C. Pahl, "Developing Self-Adaptive Microservice Systems: Challenges and Directions," pp. 1-7, 2019.

[13] "stackshare," 2018. [Online]. Available: https://stackshare.io/kubernetes.

[14] "HG Insight," [Online]. Available: https://discovery.hgdata.com/product/docker-swarm.