Hasil Penelitian

Yang Tidak Dipublikasikan

Tinjauan Pustaka
Docker Swarm Container Management Algorithms

Berkah I. Santoso, ST, MTI

Guson P. Kuntarto, ST, M.Sc

Irwan Prasetya Gunawan, Ph.D

Yusuf Lestanto, ST, M.Sc, MBA

Fakultas Teknik dan Ilmu Komputer

Universitas Bakrie

Jakarta

2024

LEMBAR PENGESAHAN HASIL PENELITIAN YANG TIDAK DIPUBLIKASIKAN

1.	Judul Penelitian	: Tinjauan Pustaka Docker Swarm Container Management Algorithms
2.	Peneliti Utama	
	a. Nama Lengkap	: Berkah I. Santoso, ST, MTI
	b. Jenis Kelamin	: Laki Laki
	c. Pangkat/Golongan/NIDN	: Asisten Ahli / III b / 0309068003
	d. Bidang Keahlian	: Cloud Computing, Networking
	e. Program Studi	: Informatika
3.	Tim Peneliti	: Guson P. Kuntarto, ST, M.Sc, Irwan Prasetya Gunawan, Ph.D, Yusuf Lestanto, ST, M.Sc, MBA
4.	Jangka waktu penelitian	: 1 Juni 2024 – 28 Juli 2024

Jakarta, 29 Juli 2024

Menyetujui,

Ketua Lembaga Penelitian dan Pengembangan	**Peneliti Utama**
(**Deffi Ayu Puspito Sari, Ph.D.**)	(**Berkah I. Santoso, ST, MTI**.)
NIDN: 0308078203	NIDN: 0309068003

Abstrak

Rapid application development dengan mengutamakan kestabilan, kecepatan waktu rilis, modularitas komponen, interoperabilitas fungsi dan keamanan aplikasi merupakan beberapa syarat wajib yang menjadi tuntutan organisasi atau perusahaan terkait pengembangan aplikasi berskala korporat saat ini. Aplikasi berskala korporat sudah dapat dipastikan memerlukan sumber daya komputasi yang harus selalu siap tersedia, pengelolaan varian sumber daya yang efisien dan manajemen kapasitas yang bersifat adaptif. Beberapa terminologi terkait penyediaan sumber daya komputasi seperti Infrastructure-as-a-Service (IaaS) hingga Platform-as-a-Service (PaaS) merupakan kebutuhan *developer* maupun pihak *operational* aplikasi yang bersifat *mandatory* dan umum digunakan dalam rangka mencapai *rapid time-to-deliver* pada sifat aplikasi modern dari ranah *development* kepada *production*[1]. Terdapat berbagai *breakthrough* terkait ketersediaan sumber daya komputasi pada kebutuhan pengembangan aplikasi secara *micro-services*–penambahan berbagai komponen layanan pada kode sumber aplikasi yang berskala mikro, bersifat modular, adaptif dan fungsional, siap terhadap kebutuhan layanan antar *multi-cloud* dengan pendekatan pengelolaan berbasis *container*. Saat ini Docker pada mode Swarm terlihat memiliki keterbatasan pada pengelolaan platform yang bersifat heterogen[2], sehingga beberapa tantangan tim *developer* dan tim *operations* (DevOps) adalah pada mekanisme pemanfaatan kekayaan fitur sekaligus pengelolaan kinerja Docker pada mode Swarm untuk membentuk aplikasi modern berbasis *micro-services* dalam rangka pencapaian ketersediaan *object* yang maksimal untuk peningkatan *development time and implementation*. Penulis mencoba untuk memberikan tinjauan literatur yang diharapkan dapat bermanfaat bagi para pengembang aplikasi modern yang masih membutuhkan mekanisme pengelolaan terintegrasi terhadap layanan berbasis *container* dengan memanfaatkan dukungan layanan *multi-cloud* agar antar komponen *micro-services* memiliki fungsionalitas pada *cloud infrastructure* dan percepatan implementasi aplikasi. Docker Swarm dapat menangani Docker *cluster container* secara efisien dan mampu melakukan pengelolaan beragam layanan, walaupun perlu dilakukan pengaturan lebih lanjut terkait pendekatan teknis Docker pada mode Swarm yang memiliki beban *overhead* relatif rendah. Pada hasil keseluruhan, kita dapat melihat kinerja dan *overhead* Docker pada mode Swarm untuk pengelolaan *container*, yaitu pencapaian kinerja lebih tinggi dan sifat response yang lebih fleksibel dengan pengaturan parameter spesifik pada algoritma pengelolaan *container*.

Daftar Isi

Bab 1

Pendahuluan

1.1 Latar Belakang

Peranan penggunaan *container* terutama pada penyedia layanan *cloud computing* dewasa ini telah menjadi paradigma baru dalam rangka percepatan dan perluasan pengembangan aplikasi skala korporat. Beberapa manfaat yagn didapatkan dari penggunaan *container* diantaranya adalah peningkatan fleksibilitas penempatan dan perpindahan layanan aplikasi baik pada *virtual server* ataupun *physical server*, kemudahan pengelolaan layanan modul aplikasi tanpa harus mengganggu layanan aplikasi lainnya, peningkatan modularitas layanan aplikasi dan kemudahan mekanisme pembagian beban komputasi untuk modul layanan aplikasi [3]. Tata kelola *container* menggunakan platform Docker Swarm yang tersedia pada beberapa penyedia layanan *public cloud computing* seperti Google Cloud®, Amazon Web Services (AWS)®, Alibaba®dipastikan mengadopsi teknik virtualisasi berikut *virtual machine* (VM) dan *container* untuk otomasi aplikasi pada infrastruktur komputasi. Penerapan *container* dengan sifat modular dan sifat lincah dalam kaitannya terhadap penyediaan sumber daya komputasi dirasakan mulai menggeser penggunaan VM pada beberapa tahun terakhir [4]. Semakin besar aplikasi yang dikembangkan tentunya memerlukan jumlah sumber daya komputasi berupa *container* yang cukup besar sehingga pengelolaan *container* menjadi lebih rumit dan menyita waktu *cloud administrator* [3]. Baik tim pengembang aplikasi dan tim operasional pada dasarnya menginginkan aplikasi yang dikelola memiliki sifat handal, dinamis mengikuti besaran beban komputasi tanpa harus direpotkan oleh kerumitan pengelolaan *container*, aman dan dapat diakses kapan saja. Pengelolaan *container*, saat ini merupakan keharusan, baik bagi pihak *developer* maupun pihak *cloud administrator*. Pemenuhan syarat kehandalan bertujuan untuk memastikan bahwa semua obyek pembentuk berfungsi secara maksimal [1]. Beberapa terobosan pada sisi pengembangan aplikasi secara *micro-services*, yaitu dengan penambahan komponen-komponen layanan pada kode sumber aplikasi yang bersifat modular, fungsional, adaptif dan berskala mikro, membutuhkan dukungan layanan *multi-cloud* dengan pendekatan berbasis *container* dalam rangka percepatan *time-to-deliver* aplikasi modern dari ranah *development* kepada *production*. Beberapa platform orkestrasi *container* seperti Docker Swarm dengan segala keterbatasan berikut keunggulan masing-masing, merupakan pembentuk layanan yang saat ini banyak digunakan oleh pengembang aplikasi, hanya saja masih digunakan terbatas pada *single-cloud infrastructure*. Docker Swarm masih memiliki keterbatasan pada fleksibilitas untuk platform yang bersifat heterogen [2]. Apabila tim *developer* dan *operations* (DevOps) dapat memanfaatkan serta melakukan orkestrasi antara fleksibilitas Kubernetes dengan kinerja maksimal Docker Swarm, kedekatan erat terhadap kernel pada Apache Mesos untuk membentuk aplikasi modern berbasis *micro-services* dengan dukungan *multi-cloud infrastructure* diharapkan dapat mencapai peningkatan waktu pengembangan dan implementasi. Para pengembang aplikasi modern masih membutuhkan *orches-*

tration framework berbentuk *platform terintegrasi* yang mampu melakukan pengelolaan layanan berbasis *container* dengan memanfaatkan dukungan layanan *multi-cloud* agar antar komponen *micro-services* memiliki fungsionalitas maksimal dari masing-masing *cloud infrastructure* dan percepatan *deployment* aplikasi modern.

1.2 Rumusan Masalah

Laporan ini mencoba membahas penggunaan *framework* pengelolaan *container* tepat guna pada layanan *cloud computing* dengan menggunakan pendekatan deskripsi dan simulasi pada bahasa pemrograman Python terkait bagaimana penerapan beberapa algoritma pada pengelolaan *container* sesuai dengan kebutuhan komputasi untuk implementasi aplikasi modern.

1.3 Batasan Masalah

Ruang lingkup pada pembahasan ini meliputi beberapa hal berikut, diantaranya adalah :

1. Algoritma framework pengelolaan *container* yang diteliti adalah pada Docker Swarm.
2. Fokus pembahasan terkait aspek fitur, simuliasi algoritma pengelolaan *container* pada Docker Swarm.

1.4 Tujuan Penelitian

Penggunaan pendekatan tinjauan fitur unggulan dan simulasi pada laporan ini ditujukan untuk identifikasi terkait bagaimana sifat algoritma pengelolaan *container* pada *container instance* yaitu Docker Swarm. Identifikasi sifat algoritma dengan simulasi menggunakan bahasa pemrograman Python tersebut bertujuan agar dapat mempelajari karakteristik *framework* pengelolaan *container* dengan pemenuhan prasyarat dan kondisi sebelum dilakukan *deployment* aplikasi modern berskala korporat.

1.5 Sistematika Penulisan

Adapun sistematika penulisan pada laporan penelitian ini meliputi :

1. Bab 1 membahas latar belakang permasalahan yang dimulai dari hadirnya *framework* orkestrasi *container* seperti Docker Swarm pada infrastruktur terdistribusi hingga optimalisasi algoritma pengelolaan *container*. Selanjutnya merupakan perumusan masalah yang dibingkai berikut batasan masalah pembahasan agar dapat memenuhi tujuan penelitian yang diharapkan.
2. Bab 2 membahas konsep yang mendasari penelitian terkait arsitektur pembentuk *framework* orkestrasi *container* seperti Docker Swarm Container Management, berikut keuntungan dan *drawbacks* pada *framework* orkestrasi *container*.
3. Bab 3 membahas tinjauan simulasi algoritma pengelolaan *container* menggunakan bahasa pemrograman Python dari masing-masing kondisi pada Docker Swarm.
4. Bab 4 memberikan kesimpulan dari hasil penelitian terkait tinjauan fitur dan studi karakteristik masing-masing algoritma pada pengelolaan *container*.

Bab 2

Gambaran Arsitektur Docker Container Management

Pengelolaan obyek-obyek *container* pada penyediaan layanan komputasi sangat diperlukan bagi keberlangsungan operasi *framework* orkestrasi *container*. Review yang dapat dilaksanakan secara internal maupun dengan penggunaan tenaga ahli eksternal terkait fitur, keuntungan, kelemahan dan komparasi unjuk kerja dari masing-masing *framework* manajemen *container* sangat diperlukan untuk membantu *solution architect* dan penanggung jawab infrastruktur TI dalam rangka penyediaan solusi tepat guna bagi organisasi dan perusahaan.

Pada solusi *framework container management* dalam hal ini adalah Docker Swarm, memiliki aspek keamanan, implementasi, stabilitas, skalabilitas, instalasi cluster[5]. Docker Swarm merupakan *framework container management* yang memiliki dukungan komunitas yang kuat dan memiliki persebaran pengguna yang luas[6][7]. Salah satu keunggulan Docker Swarm *framework container management* adalah kemudahan penggunaan fitur dan obyek pada arsitektur sumber daya komputasi aplikasi modern.

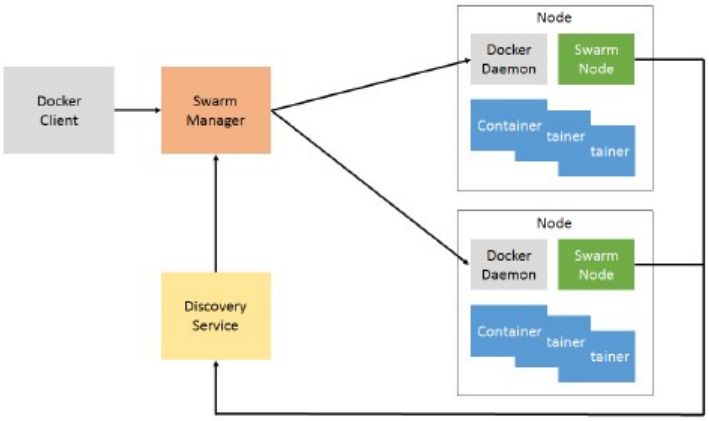
Docker Swarm *framework container management* diluncurkan kepada publik pada lingkungan produksi tahun 2016 yang memiliki ciri khas kesederhanaan pada pustaka orkestrasi dan kemudahan integrasi komponen pada semua lingkungan kerja Docker. Beberapa fitur seperti integrasi pada Docker Application Programming Interface (API), kemudahahan pengelolaan obyek Docker merupakan faktor kunci dalam penggunaan dan pengelolaan *container*. Docker Swarm memiliki mekanisme unik dalam mengelola dan melakukan konfigurasi *container* dimana pada awal pengembangan Docker Swarm tujuannya adalah dapat berjalan secara mandiri, dengan mekanisme pengembangan yang terintegrasi sehingga menjadi orkestrasi sekumpulan *container* berjumlah banyak melalui perantaraan jaringan [3].

Pengembangan orkestrasi pengelolaan *container* telah dilakukan menggunakan bahasa pemrograman Go Lang yang bertujuan untuk dapat langsung berintegrasi dengan Docker API, sehingga semua fitur yang terdapat pada *container* Docker dapat digunakan dan diterapkan pada Docker Swarm. Pengembang aplikasi modern atau pengelola infrastruktur berbasis *container* Docker dapat langsung menggunakan konsep orkestrasi Docker Swarm tanpa harus mempelajari lebih lanjut konsep pengelolaan orkestrasi *container* dari manufaktur aplikasi lainnya. YAML *Ain't Markup Languange* (YAML) merupakan bahasa serialisasi yang semula digunakan sebagai format konfigurasi berkas seperti halnya mekanisme *porting* pada bahasa pemrograman Python (PyYAML), sehingga YAML dapat digunakan sebagai landasan pengelolaan *container* pada Docker Swarm [3].

Pada penggunaan Docker Swarm yang memiliki keterkaitan erat dengan arsitektur Docker *container*, pengelola infrastruktur *container* dapat melakukan pembuatan dan pengelolaan *container cluster* yang terbentuk dari komponen-komponen berikut ini [3]:

- *Task*
- *Services*
- *Raft Consensus Group*
- *Internal Distributed Stated Store*
- *Manager Nodes*, terdiri dari :
  - *API*
  - *Orchestrator*
  - *Allocator*
  - *Scheduler*
  - *Dispatcher*
- *Worker Nodes*

Pembaca dapat melihat penggambaran arsitektur Docker Swarm seperti terlihat pada gambar berikut :



Gambar 2.1: Arsitektur Docker Swarm [8]

2.1 Docker Task

Task dapat kita pahami sebagai kombinasi *container* Docker tunggal berikut perintah-perintah yang mendefinisikan bagaimana *container* tersebut akan dijalankan. Dari penggunaan *task* tersebut memungkinkan operasi *container* dijalankan pada Docker Swarm sehingga terjadi orkestrasi *container* dalam bentuk *cluster*.

2.2 Docker Services

Docker services dapat terbentuk dari beberapa **task** untuk membentuk layanan Docker, sehingga terjadinya orkestrasi dapat tercapai untuk *cluster container*. Pengelolaan *task* pada Docker Swarm merupakan hal yang signifikan karena aplikasi modern saat ini seringkali membutuhkan banyak operasi *container* agar layanan pada aplikasi tetap berjalan dengan stabil dan efisien.

2.3 Docker Raft Consensus Group

Raft Consensus Group merupakan sekumpulan kondisi *database* dan *worker nodes* yang bersifat terdistribusi, baik secara internal maupun secara eksternal. *Worker nodes* tersebut menerima tugas secara langsung dari *control node* untuk dikerjakan pada *container*.

Nilai utama pada Docker Swarm terkait aspek *fault tolerance* yaitu Docker Swarm memperbolehkan eksistensi obyek kendali *node* lebih dari satu *control node* dalam *cluster container* tunggal, akan tetapi Docker Swarm juga memperbolehkan eksistensi terhadap banyak *control node*.

Container Cluster dapat melakukan pemulihan kondisi dari kesalahan tanpa diharuskan untuk melakukan mekanisme *interrupt* terhadap operasi yang sedang dilakukan. Apabila terdapat kesempatan untuk melakukan pemilihan *management node* sebagai koordinator utama secara kebetulan terjadi keadaan *interrupt* atau tidak dapat tersedia, maka kelompok Raft Consensus akan melakukan pemilihan *container* koordinator untuk dapat menjalankan tugas-tugas orkestrasi tertentu.

2.4 Docker Internal Distributed Stated Store

Internal Distributed Stated Store merupakan zona komputasi untuk menyimpan data mengenai kondisi *cluster container* pada format nilai yang digunakan pada lingkungan kerja *container*. Data-data kondisi tersebut digunakan untuk mekanisme orkestrasi *cluster container* sehingga diharapkan pengelolaan *container* dapat dilakukan dengan baik.

2.5 Docker Manager Nodes

Manager Nodes memiliki peran penting yaitu sebagai pengelola *node container*. Adapun *manager nodes* memiliki kelengkapan komponen sebagai berikut :

- **API**, yang berfungsi sebagai penerima perintah dari pengguna dan membuat layanan baru yang didasarkan kepada parameter yang telah didefinisikan pada perintah yang diberikan.
- **Orchestrator**, yang berfungsi untuk mengambil definisi layanan agar dapat membuat serta mengelola tugas-tugas *container*.
- **Task Allocator**, yang berfungsi sebagai pengelola pengalamatan *Internet Protocol* (IP) untuk *container*.
- **Scheduler**, yang berfungsi sebagai pemberi jadwal tugas-tugas *container* dan meneruskan perintah tugas tersebut pada *worker nodes*.
- **Dispatcher**, yang berfungsi untuk mengelola *worker nodes* agar dapat menjalankan tugas-tugas yang diberikan.

Pada saat *manager node* dipilih sebagai koordinator, maka *manager node* tersebut dapat melaksanakan tugas-tugas komputasi terkait pengelolaan dan orkestrasi. Beberapa tugas-tugas komputasi terkait pengelolaan *container* adalah sebagai berikut :

- Melakukan mekanisme penerimaan layanan yang sudah didefinisikan oleh pengguna.
- Melakukan mekanisme replikasi dari beberapa tugas-tugas komputasi berdasarkan penggunaan definisi pada layanan yang terbentuk.
- Melakukan pengiriman tugas-tugas komputasi terhadap *node* yang sedang berjalan.

- Melaksanakan mekanisme orkestrasi dan mengelola *container cluster*.
- Melaksanakan mekanisme *load balancing* pada input *container*.
- Melakukan mekanisme kerjasama dengan *internal distributed status database* untuk tujuan monitoring *update* kondisi *container cluster*, sebagai dasar untuk penentuan kapasitas komputasi bagi aplikasi modern.

2.6 Docker Worker Nodes

Worker nodes berfungsi untuk menerima tugas-tugas beban kerja komputasi secara langsung dari *control node* agar dapat menjalankan tugas-tugas tersebut. *Worker nodes* juga bertugas untuk mengirimkan *update* informasi kondisi *container* terkait tugas-tugas komputasi terhadap *control node* dan *worker nodes* mengirimkan informasi sendiri kepada *control node* agar diketahui bahwa *worker node* masih dapat menerima tugas-tugas komputasi dan dapat menjalankan tugas-tugas komputasi tersebut [3].

2.7 Keuntungan Penggunaan Docker Swarm Container Management

Adapun beberapa keuntungan penggunaan Docker Swarm *Container Management* dapat kami jabarkan sebagai berikut :

- Pengelolaan *container cluster* langsung dapat menggunakan lingkungan kerja Docker Command Line Interface (CLI) tanpa adanya tambahan *software* pengelolaan orkestrasi lainnya.
- Desain Docker Swarm bersifat terdesentralisasi, sehingga terdapat mekanisme perbedaan cara pengelolaan antara satu *container cluster* dengan *container cluster* yang lain pada saat menjalankan *container*. Mekanisme tersebut menjadikan keseluruhan *container cluster* dapat disusun dari *image* aplikasi tunggal tanpa memerdulikan tipe *container node* yang akan dimasukkan pada *cluster*.
- Model desain Docker Swarm bersifat *declarative* dalam rangka pendefinisian layanan aplikasi sehingga memungkinkan pengguna Docker Swarm untuk memberikan definisi kondisi *container* yang diperlukan dengan beragam layanan. Sebagai contoh, apabila pengguna membangun aplikasi yang membutuhkan mekanisme *end-service queues* yaitu web dan mekanisme *multiple back-end services* yaitu aplikasi untuk antrian pesan dan database.
- Mekanisme pengelolaan skalabilitas yang diterapkan pada masing-masing layanan. Pada saat dilakukan peningkatan skalabilitas, maka Docker Swarm dapat melakukan penambahan atau membuang perintah komputasi secara otomatis pada saat diperlukan, sehingga lingkungan komputasi *container* dapat terus terjaga sesuai dengan kebutuhan pengguna Docker Swarm.
- Mekanisme penyediaan kondisi komputasi *container cluster* yagn diperlukan untuk pengelolaan monitoring *node* secara konstan, sehingga update kondisi *container cluster* yang mengalami peningkatan kebutuhan sumber daya komputasi dapat terpantau dan terkoreksi dari kesalahan agar tidak mengalami perubahan dari kondisi yang diinginkan pengguna Docker Swarm.
- Mekanisme *networking* dalam *container cluster* Docker Swarm yang memungkinkan pengguna untuk membangun *overlay networks* dalam rangka pemenuhan kebutuhan layanan komputasi. Pada saat Docker Swarm melakukan inisialisasi atau *update* suatu aplikasi, maka *control node* akan

memberikan alamat Internet Protocol (IP Address) secara otomatis terhadap semua container yang terdapat pada *subnetwork* tersebut.

- Mekanisme *load balancing* pada Docker Swarm menggunakan *ports* dari beberapa layanan individual untuk penyediaan fitur pembagian beban terhadap kebutuhan komputasi eksternal. Pengelola distribusi beban komputasi pada internal Docker Swarm digunakan untuk memastikan bahwa terdapat pengaturan yang efektif dan adil pada *node* secara individual untuk persebaran *container cluster*.
- Mekanisme keamanan pada Docker Swarm ditujukan agar masing-masing *node* pada *cluster* dapat menjalankan otentikasi dan enkripsi untuk pengamanan komunikasi antar node dalam cluster. Pengamanan pada operasi otentikasi dan enkripsi dilakukan dengan menggunakan sertifikat Trusted Layer Security (TLS), sertifikat mandiri dari *user root* dan dapat juga menggunakan sertifikat dari Certification Authority (CA).
- Mekanisme pembentukan *restore points* untuk keperluan *update* pada Docker Swarm dalam rangka *update* versi layanan secara berurutan dengan memperhatikan *restore points* pada masing-masing versi *update*. Pembentukan *restore points* tersebut bertujuan untuk menangani kondisi kegagalan *update* terhadap layanan dengan mekanisme kembali yaitu *rollback* pada versi layanan sebelumnya.
- Mekanisme pengelolaan implementasi layanan pada Docker Swarm, dimana *control node* melakukan pengelolaan waktu implementasi layanan terhadap *node* secara individual.

2.8 Kelemahan pada pengelolaan Docker Swarm Container eksisting

Pada Docker Swarm sebelumnya terdapat beberapa *drawbacks* yang masih dalam proses penyempurnaan lebih lanjut dengan keterangan sebagai berikut :

- Docker Swarm belum memiliki mekanisme peningkatan skalabilitas aplikasi secara otomatis.
- Docker Swarm belum mendukung mekanisme *load balancing* yang dilakukan dari pihak eksternal Docker Swarm.
- Mekanisme tambahan peningkatan skalabilitas aplikasi pada Docker Swarm harus dilakukan secara manual atau dilakukan melalui solusi dari pengembang diluar Docker Swarm.
- Mekanisme *load balancing* secara eksternal dapat dilakukan seperti penggunaan Amazon Web Services Elastic Load Balancing (AWS ELB).

Penggunaan solusi Swarmpit pada pengembangan Docker Swarm selanjutnya, ditujukan untuk menyediakan pengelolaan *container* yang baik dengan menggunakan antar muka grafis untuk membantu pengguna Docker Swarm [3].

Bab 3

Tinjauan Algoritma dan Script Docker Container Management

Penggunaan beberapa algoritma pada orkestrasi Docker Swarm bertujuan untuk mendapatkan efisiensi atas beban kerja komputasi pada berbagai macam konfigurasi sumber daya komputasi dalam 3 (tiga) cara, diantaranya adalah [9]:

- Mendukung konfigurasi tugas kerja komputasi yang bervariasi dalam rangka optimalisasi penempatan *container* pada saat proses inisialisasi.
- Pemberian ukuran *cluster* melalui mekanisme *autoscaling* sebagai respons atas terjadinya fluktuasi beban kerja komputasi pada saat dijalankan.
- Mekanisme penjadwalan ulang terhadap *cluster* dan implementasi ulang terhadap obyek-obyek *vritual machine* yang memiliki utilisasi rendah.

3.1 Algoritma terkait Penjadwalan

Pada layanan yang berjalan untuk kurun waktu lama maka proses penjadwalan dapat menggunakan versi *online* terhadap paket pemecahan masalah dalam dua dimensi, dimana masing-masing *pod* dikaitkan dengan dua obyek yaitu permintaan terhadap CPU dan permintaan terhadap memori. Penggunaan algoritma penjadwalan tersebut bertujuan untuk memberikan alokasi yang tepat pada *pods* dengan jumlah *virtual machine* diusahakan paling sedikit. Algoritma *Best Fit Decreasing* (BFD) merupakan algoritma penempatan obyek dengan berbagai macam ukuran untuk dapat mengisi tempat penyimpanan secara efisien dan untuk mendapatkan hasil penempatan secara optimal. Suatu obyek baru ditempatkan pada suatu wadah penyimpanan yang tepat dan optimal. Apabila obyek tersebut gagal ditempatkan pada wadah penyimpanan yang tersedia, maka obyek dapat mengajukan permintaan untuk mendapatkan wadah penyimpanan yang baru. Algoritma BFD dapat diterapkan selama kurun waktu O (n log n) dengan menggunakan pendekatan pohon biner yang diseimbangkan dari ukuran terbesar hingga ukuran terkecil dan tersusun oleh sisa kapasitas untuk kunci pencarian. Paket yang dipilih kemudian dihapus secara berulang dan wadah penyimpanan akan diisi kembali berdasarkan kapasitas sisa yang lebih baru [10]. Algoritma BFD merupakan paket algoritma yang akan mengatur masing-maing *pod* terhadap obyek komputasi yang memiliki kecukupan utilisasi atas permintaan sumber daya secara maksimal. CPU diasumsikan sebagai sumber daya komputasi yang dapat dimampatkan sedangkan memory dilakukan evaluasi pada prioritas yang lebih tinggi daripada CPU ketika dilakukan mekanisme pemeringkatan obyek yang tersedia pada algoritma BFD untuk masing-masing *pod*. Sedangkan algoritma Time-bin BFD merupakan paket algoritma

yang menggunakan pendekatan fungsi waktu penempatan untuk mengatur masing-maing *pod* terhadap obyek komputasi yang memiliki kecukupan utilisasi atas permintaan sumber daya secara optimal.

Algoritma BFD dan Time-bin BFD berikut file-file Python terkait algoritma tersebut dapat pembaca lihat pada bagian berikut ini :

ALGORITHM 1: BFD

Input: Pending pod p and node group ng
Output: Schedule plan $S = \{p \rightarrow node\}$

1. **select** *node*, min(*node*.availableRAM, *node*.availableCPU) from *ng*
2. **where** node.availableRAM >= *p*.memory && *node*.availableCPU >= *p*.CPU
3. $S = \{p \rightarrow node\}$
4. **If**(p.runtime > *node*.runtime) **then**
5. *node*.runtime = p.runtime
6. end **if**
7. return *S*

ALGORITHM 2: Time-bin BFD

Input: Pending pod p, batch node group ng, and scaling cycle sc
Output: Schedule plan $S = \{p \rightarrow node\}$

1. int bin = p.runtime/sc
2. **for**(int i = *bin*; i <= *ng.maxBin*; i++)
3. S = BFD(p, ng_i)
4. **If**(S != null) **then**
5. **return** *S*
6. **end if**
7. **end for**
8. **for**(int i = *bin*; i >= *ng.minBin*; i–)
9. S = BFD(p, ng_i)
10. **If**(S != null) **then**
11. **return** *S*
12. **end if**
13. **end for**
14. **return** null

Listing 3.1: Listing Python untuk Algoritma Best Fit Decreasing(BFD)

```
class Bin(object):
    """ Container untuk menyimpan paket yang sedang berjalan secara keseluruhan """
    def __init__(self):
        self.items = []
        self.sum = 0
    def append(self, item):
        self.items.append(item)
        self.sum += item
    def __str__(self):
        """ Tampilan pada layar """
        return 'Container (sum=%d, items=%s)' % (self.sum, str(self.items))
def pack(values, maxValue):
    values = sorted(values, reverse=True)
```

```
    bins = []

    for item in values:
        # Mencoba untuk menempatkan paket pada container
        for bin in bins:
            if bin.sum + item <= maxValue:
                #print 'Penambahan', item, 'pada', bin
                bin.append(item)
                break
        else:
            # Paket tidak dapat disimpan pada container, mengulangi pencarian container
                baru
            #print 'Membuat container baru untuk', item
            bin = Bin()
            bin.append(item)
            bins.append(bin)

    return bins

if __name__ == '__main__':
    import random

    def packAndShow(aList, maxValue):
        """ Paket pada daftar yang disimpan pada container dan menampilkan hasilnya """
        print ('Daftar dengan jumlah paket', sum(aList), 'membutuhkan sekurang-kurangnya',
             (sum(aList)+maxValue-1)/maxValue, 'container')

        bins = pack(aList, maxValue)

        print ('Hasilnya adalah dengan menggunakan', len(bins), 'container:')
        for bin in bins:
            print (bin)
        print

    aList = [10,9,8,7,6,5,4,3,2,1]
    packAndShow(aList, 11)

    aList = [ random.randint(1, 11) for i in range(100) ]
    packAndShow(aList, 11)
```

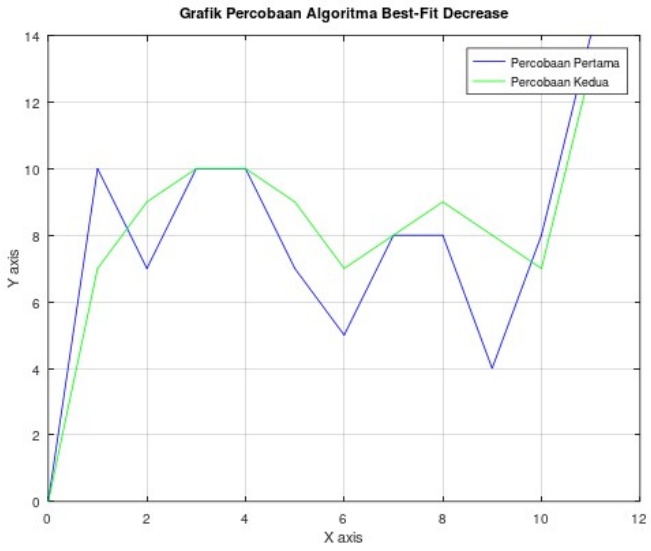
Listing 3.2: Listing Python untuk Algoritma Time-Bin Best Fit Decreasing(BFD)

```
import math

class Item:
    def __init__(self, val, id):
        self.val = val
```

```
        self.id = id

def best_fit(items, assignment, free_space):
    n = len(items)
    bin_count = 0
    cap = 1.0

    for i in range(n):
        min_left_bin = -1
        min_left = cap + 1.0

        for j in range(bin_count):
            if (items[i] < free_space[j] or math.isclose(items[i], free_space[j])) and
                free_space[j] - items[i] < min_left:
                min_left_bin = j
                min_left = free_space[j] - items[i]

        if min_left_bin == -1:  # Apabila paket tidak dapat menempati suatu container maka
             melakukan permintaan container baru
            free_space.append(cap - items[i])
            assignment[i] = bin_count
            bin_count += 1
        else:  # Mencari container dimana dapat tepat optimal digunakan untuk menyimpan
            paket
            free_space[min_left_bin] -= items[i]
            assignment[i] = min_left_bin

    for k in range(len(free_space)):
        if math.isclose(free_space[k], 0.0):
            free_space[k] = 0.0

def best_fit_d(items, assignment, free_space):
    n = len(items)
    bin_count = 0
    cap = 1.0

    for i in range(n):
        min_left_bin = -1
        min_left = cap + 1.0

        for j in range(bin_count):
            if (items[i].val < free_space[j] or math.isclose(items[i].val, free_space[j]))
                 and free_space[j] - items[i].val < min_left:
                min_left_bin = j
                min_left = free_space[j] - items[i].val

```

```
48          if min_left_bin == -1:  # Apabila paket tidak dapat disimpan pada container maka
                mulai mencari container dengan ukuran tepat
49              free_space.append(cap - items[i].val)
50              assignment[items[i].id] = bin_count
51              bin_count += 1
52          else:  # Melakukan pencarian container dengan ukuran paling optimal
53              free_space[min_left_bin] -= items[i].val
54              assignment[items[i].id] = min_left_bin
55
56      for k in range(len(free_space)):
57          if math.isclose(free_space[k], 0.0):
58              free_space[k] = 0.0
59
60  def best_fit_decreasing(items, assignment, free_space):
61      items_d = []
62      n = len(items)
63      for i in range(n):
64          itm = Item(items[i], i)
65          items_d.append(itm)
66
67      items_d.sort(key=lambda x: x.val, reverse=True)
68      best_fit_d(items_d, assignment, free_space)
```

Untuk hasil eksekusi listing Python dari 2 (dua) kali percobaan pada algoritma Best-Fit Decreasing (BFD) dapat kita lihat pada gambar berikut ini:



Gambar 3.1: Hasil percobaan eksekusi *script* Python untuk algoritma BFD

3.2 Algoritma terkait Autoscaling

Pada kelompok *node* yang berjalan untuk kurun waktu yang panjang maka algoritma Greedy Autoscaling (GA) dapat diimplementasikan untuk mencari kombinasi yang tepat pada *vritual machine* dengan

variasi yang berbeda dalam rangka mencapai efisiensi biaya komputasi yang paling efisien terkait dengan volume sumber daya komputasi yang dibutuhkan. Mekanisme pemenuhan kebutuhan sumber daya komputasi dilakukan secara iterasi melalui pembuatan daftar obyek komputasi yang tersedia beberapa kali. Algoritma Greedy merupakan kelas algoritma yang membuat pilihan optimal secara lokal pada setiap tahapan dengan harapan agar mendapatkan hasil pilihan optimal, sehingga sistem yang menjalankan algoritma Greedy akan memilih pilihan terbaik yang tersedia tanpa mempertimbangkan akibat lebih jauh atas pilihan tersebut pada tahapan-tahapan selanjutnya. Sebagai contoh penggunaan algoritma Greedy terkait tantangan masalah Fractional Knapsack.

Pada kelompok *node* yang berjalan pada latar belakang, maka kapasitas sumber daya komputasi seluruhnya yang memiliki *nodes* sedang berjalan pada lingkungan kerja komputasi akan dilakukan pemeriksaan terlebih dahulu dimana waktu kerja *nodes* kurang dari siklus peningkatan skalabilitas. Sumber daya komputasi tersebut dapat dilepaskan sebelum terjadi akuisisi pada obyek komputasi sehingga kapasitas *node* tersebut dapat mempengaruhi penentuan peningkatan skalabilitas. Mekanisme *autoscaling* menjadi hal yang kurang diperlukan apabila kapasitas eksisting masih dapat digunakan untuk memenuhi kebutuhan tugas-tugas komputasi yang belum selesai dilaksanakan.

Algoritma Greedy Autoscaling dan Batch Node Group Autoscaling dapat pembaca lihat pada bagian berikut ini :

Listing 3.3: Listing Python untuk Algoritma Greedy Algorithm(GA)

```
def minimize_lateness(ttimes, dtimes):
    # index = [0, 1, 2, ..., n - 1] for n requests
    index = list(range(len(dtimes)))
    # sort according to deadlines
    index.sort(key=lambda i: dtimes[i])

    min_lateness = 0
    start_time = 0
    for i in index:
        min_lateness = max(min_lateness,
                           (ttimes[i] + start_time) - dtimes[i])
        start_time += ttimes[i]

    return min_lateness, index

n = int(input('Enter number of requests: '))
ttimes = input('Enter the time taken to complete the {} request(s) in order: '
              .format(n)).split()
ttimes = [int(tt) for tt in ttimes]
dtimes = input('Enter the deadlines of the {} request(s) in order: '
               .format(n)).split()
dtimes = [int(dt) for dt in dtimes]

min_lateness, schedule = minimize_lateness(ttimes, dtimes)
print('The minimum maximum lateness:', min_lateness)
print('The order in which the requests should be scheduled:', schedule)
```

ALGORITHM 3: GA

Input: Pending pods p and available VM instance flavors f
Output: VM instance combination vc

1. $tcpu \leftarrow$ total CPU request sum($p.CPU$)
2. $tram \leftarrow$ total RAM request sum($p.memory$)
3. **while**($tcpu > 0 \parallel tram > 0$)
4. $f_i \leftarrow$ flavor with the highest cost-efficiency score max($score_f$)
5. tcpu = tcpu - f_i.CPU
6. tram = tram - f_i.RAM
7. vc.append(f_i)
8. **end while**
9. **return** vc

ALGORITHM 4: Batch node group autoscaling

Input: Pending pods p, batch node group np, and available VM instance flavors f
Output: VM instance combination vc

1. $tcpu \leftarrow$ total CPU request sum($p.CPU$)
2. $tram \leftarrow$ total RAM request sum($p.memory$)
3. **for** $node$ in np_0
4. tcpu = tcpu - node.CPUCapacity
5. tram = tram - node.RAMCapacity
6. **end for**
7. **if**($tcpu <= 0$ && $tram <= 0$) **then**
8. **return** null
9. **end if**
10. **while**($tcpu > 0 \parallel tram > 0$)
11. $f_i \leftarrow$ flavor with the highest cost-efficiency score max($score_f$)
12. tcpu = tcpu - f_i.CPU
13. tram = tram - f_i.RAM
14. vc.append(f_i)
15. **end while**
16. **return** vc

Listing 3.4: Listing Python untuk Algoritma Greedy Autoscaling(GA) case Fractional Knapsack

```
def fractional_knapsack(capacity, weights, values):
    n = len(weights)
    ratios = [(values[i] / weights[i], weights[i], values[i], i) for i in range(n)]
    ratios.sort(reverse=True, key=lambda x: x[0])

    total_value = 0
    knapsack = [0] * n

    for ratio, weight, value, index in ratios:
        if capacity >= weight:
            knapsack[index] = 1
            total_value += value
            capacity -= weight
        else:
```

```
            fraction = capacity / weight
            knapsack[index] = fraction
            total_value += fraction * value
            break

    return total_value, knapsack

# Example usage:
knapsack_capacity = 50
item_weights = [10, 20, 30]
item_values = [60, 100, 120]

max_total_value, selected_items = fractional_knapsack(knapsack_capacity, item_weights,
    item_values)
print("Nilai Total Maksimum:", max_total_value)
print("Paket yang Terpilih pada fraksi:", selected_items)
```

3.3 Algoritma terkait Penjadwalan Ulang

Apabila rasio utilisasi sumber daya komputasi dari *node* yang berjalan pada latar belakang terus menerus berada pada kondisi dibawah batas atas utilisasi *node* yang dapat dikonfigurasi, yaitu sebesar 50 persen, maka tugas-tugas komputasi yang berjalan pada latar belakang akan dimigrasikan oleh algoritma penjadwalan ulang.

Apabila tidak terdapat cukup ruang komputasi pada *cluster* untuk implementasi ulang semua *container* pada *pods*, maka tugas-tugas komputasi terkait migrasi tidak akan diaktivasi. Sementara itu *node* yang memiliki utilisasi rendah akan dimatikan untuk penghematan biaya komputasi setelah mekanisme penjadwalan ulang selesai tanpa terjadi kehilangan perkembangan tugas-tugas komputasi. Pada mekanisme pod m yang dijadwalkan ulang, maka waktu migrasi T_m ditentukan oleh ukuran *image container* C dengan bandwidth yang tersedia B dan waktu untuk menyelesaikan tugas-tugas komputasi K pada kode sumber dengan persamaan :

$$T_m = \frac{C}{B} + K.$$

Algoritma Rescheduling dapat pembaca lihat sebagai penjelasan lebih lanjut pada bagian berikut ini :

ALGORITHM 5: Rescheduling

Input: Underutilized **batch** node n, its running pods p, and its corresponding node group ng

1. $an \leftarrow ng$ - n, other available nodes in ng
2. **for** each p_i in p
3. S_i = schedule(p_i, an)
4. **if**(S_i is null)
5. **break**;
6. **end if**
7. **end for**
8. **if**(all S_i is not null) **then**
9. deploy migrations defined by all S_i and shut down n
10. **else**
11. rescheduling is not triggered due to resource shortage
12. **end if**

Listing 3.5: Listing Python untuk Algoritma Rescheduling dan Scheduling

```
import argparse
import itertools

class Process():
    def __init__(self, /,
            name=None,
            burst=None,
            arrival_time=None,
            priority=None,
            time_slice=None,
            waiting_time=None,
            turnaround_time=None):
        self.name            = name
        self.burst           = burst
        self.arrival_time    = arrival_time
        self.priority        = priority
        self.time_slice      = time_slice
        self.waiting_time    = waiting_time
        self.turnaround_time = turnaround_time

    def __repr__(self):
        args = [('name',         self.name),
                ('burst',        self.burst),
                ('arrival_time', self.arrival_time),
                ('priority',     self.priority),
                ('time_slice',   self.time_slice)]
        if self.waiting_time is not None:    args.append(('waiting_time',    self.
            waiting_time))
        if self.turnaround_time is not None: args.append(('turnaround_time', self.
            turnaround_time))
```

```
        args = [f'{arg[0]}={repr(arg[1])}' for arg in args]
        return f'Process({", ".join(args)})'

def parse_line(line):
    line = line.strip()
    assert(line[0] == '@' and line[-1] == '&')
    line = line[1:-1]
    fields = line.split(';')
    assert(len(fields) == 5)
    process = Process()
    process.name         = fields[0]
    process.burst        = int(fields[1])
    process.arrival_time = int(fields[2])
    process.priority     = int(fields[3])
    process.time_slice   = int(fields[4])
    return process

def read_file(filename):
    with open(filename) as file:
        return [parse_line(line) for line in file.readlines() if line]

# First Come First Serve
def fcfs(processes):
    processes.sort(key=lambda process: process.arrival_time)
    # Start simulation at the first arrival.
    current_time = processes[0].arrival_time
    result = []
    for process in processes:
        # Adjust current time if CPU was idle.
        if process.arrival_time > current_time:
            current_time = process.arrival_time
        process.waiting_time = current_time - process.arrival_time
        current_time += process.burst
        process.turnaround_time = current_time - process.arrival_time
        result.append(process)
    return result

# Shortest Job First
def sjf(processes):
    processes.sort(key=lambda process: process.arrival_time)
    # Start simulation at the first arrival.
    current_time = processes[0].arrival_time
    # Yields the job with the shortest burst that has already arrived.
    def next_process():
        while True:
            nonlocal current_time
```

```
            if not processes: return
            next = 0
            # Adjust current time if CPU was idle.
            if processes[next].arrival_time > current_time:
                current_time = processes[next].arrival_time
            for i in range(1, len(processes)):
                # This condition relies on the list being sorted.
                if processes[i].arrival_time > current_time:
                    break
                if processes[i].burst < processes[next].burst:
                    next = i
            # ‘pop‘ removes the process from the ‘processes‘ list.
            yield processes.pop(next)
    result = []
    for process in next_process():
        process.waiting_time = current_time - process.arrival_time
        current_time += process.burst
        process.turnaround_time = current_time - process.arrival_time
        result.append(process)
    return result

# Shortest Remaining Time First
def srtf(processes):
    processes.sort(key=lambda process: process.arrival_time)
    # Since this is a preemptive algorithm, that is, processes will not
    # necessarily run until completion uninterrupted, we need to keep track of
    # the remaining burst.
    for i in range(len(processes)):
        processes[i].remaining_burst = processes[i].burst
    # Start simulation at the first arrival.
    current_time = processes[0].arrival_time
    result = []
    while True:
        if not processes: break
        # Find the process with the shortest remaining time.
        next = 0
        # Adjust current time if CPU was idle.
        if processes[next].arrival_time > current_time:
            current_time = processes[next].arrival_time
        for i in range(1, len(processes)):
            # This condition relies on the list being sorted.
            if processes[i].arrival_time > current_time:
                break
            if processes[i].remaining_burst < processes[next].remaining_burst:
                next = i
        # Simulate one unit of time at the time in case another processes
```

```
        # arrives.
        processes[next].remaining_burst -= 1
        current_time += 1
        if processes[next].remaining_burst == 0:
            processes[next].completion_time = current_time
            result.append(processes.pop(next))
    for i in range(len(result)):
        result[i].waiting_time = result[i].completion_time - result[i].arrival_time -
            result[i].burst
        result[i].turnaround_time = result[i].completion_time - result[i].arrival_time
    return result

# Round Robin
def rr(processes):
    processes.sort(key=lambda process: process.arrival_time)
    # Since this is a preemptive algorithm, that is, processes will not
    # necessarily run until completion uninterrupted, we need to keep track of
    # the remaining burst.
    for i in range(len(processes)):
        processes[i].remaining_burst = processes[i].burst
    # Start simulation at the first arrival.
    current_time = processes[0].arrival_time
    for i in itertools.cycle(range(len(processes))):
        # Terminate if all processes are done.
        if all(process.remaining_burst == 0 for process in processes):
            break
        # Skip this process if it has not yet arrived or if it is completed.
        if processes[i].arrival_time > current_time or processes[i].remaining_burst == 0:
            continue
        current_burst = min(processes[i].remaining_burst, processes[i].time_slice)
        processes[i].remaining_burst -= current_burst
        current_time += current_burst
        if processes[i].remaining_burst == 0:
            processes[i].completion_time = current_time
    for i in range(len(processes)):
        processes[i].waiting_time = processes[i].completion_time - processes[i].
            arrival_time - processes[i].burst
        processes[i].turnaround_time = processes[i].completion_time - processes[i].
            arrival_time
    return processes

if __name__ == '__main__':
    arg = argparse.ArgumentParser()
    option = arg.add_mutually_exclusive_group()
    option.add_argument('--fcfs', help='First Come First Serve (default)', action='
        store_true', default=True)
```

```
    option.add_argument('--sjf',  help='Shortest Job First',               action='
        store_true')
    option.add_argument('--srtf', help='Shortest Remaining Time First',    action='
        store_true')
    option.add_argument('--rr',   help='Round Robin',                      action='
        store_true')
    arg.add_argument('file', help='Input file', type=str)
    arg = arg.parse_args()

    algorithm = fcfs
    if arg.sjf:  algorithm = sjf
    if arg.srtf: algorithm = srtf
    if arg.rr:   algorithm = rr

    processes = read_file(arg.file)
    processes = algorithm(processes)
    for process in processes:
        print(process)
    print(f'Average waiting time: {sum(process.waiting_time for process in processes) /
        len(processes)}')
    print(f'Average turnaround time: {sum(process.turnaround_time for process in processes
        ) / len(processes)}')
```

Bab 4

Kesimpulan

Docker Swarm dapat mengelola *cluster container* secara tepat guna dalam melakukan pengelolaan beragam layanan yang mengedepankan efisiensi lingkungan kerja dan pelaksanaan sifat-sifat operasi Docker. Beban *overhead* komputasi pada Docker Swarm memiliki beban *overhead* lebih rendah, dengan kinerja komputasi yang cukup tinggi. Mekanisme pengelolaan *container* pada Docker Swarm masih sangat terbuka untuk pengembangan lebih lanjut, terutama untuk penyederhanaan tugas-tugas komputasi yang lebih kompleks. Hal tersebut ditujukan untuk meningkatkan adaptasi pengelolaan komponen *container* pada Docker Swarm.

Penggunaan beberapa algoritma untuk mekanisme penjadwalan, *autoscaling* dan penjadwalan ulang ditujukan agar Docker Swarm sebagai *platform* pengelolaan *container* mendapatkan efisiensi atas beban kerja komputasi pada berbagai macam konfigurasi sumber daya komputasi dalam 3 (tiga) cara, yaitu pemberian ukuran *cluster* melalui pengaturan *autoscaling* sebagai respons atas terjadinya fluktuasi beban kerja komputasi pada saat dijalankan, mekanisme penjadwalan ulang terhadap *container cluster* dan mendukung konfigurasi tugas kerja komputasi yang bervariasi dalam rangka optimalisasi penempatan *container* pada saat proses inisialisasi dan implementasi ulang terhadap obyek-obyek *vritual machine* yang memiliki utilisasi rendah.

Beberapa hal yang masih terbuka untuk dikembangkan lebih lanjut dari penelitian ini adalah karakteristik respons *container* dari *framework container management* Docker Swarm atas kemungkinan terjadinya lonjakan beban kerja komputasi secara tiba-tiba, baik karena terdapat permintaan layanan sebenarnya yang tinggi atau karena terdapat serangan terhadap infrastruktur *container*. Aspek lainnya yang masih dapat dikembangkan dari penelitian ini adalah peningkatan kecepatan *backup and recovery* layanan Docker Swarm secara adaptif, terlebih pada saat *system maintenance* atau terdapat *techincal faulty* pada pengelolaan *container cluster*.

Bibliography

[1] N. Naik, "Building a virtual system of systems using docker swarm in multiple clouds," in *2016 IEEE International Symposium on Systems Engineering (ISSE)*, 2016, pp. 1–3. 1, 3

[2] Y. Pan, I. Chen, F. Brasileiro, G. Jayaputera, and R. Sinnott, "A performance comparison of cloud-based container orchestration tools," in *2019 IEEE International Conference on Big Knowledge (ICBK)*, 2019, pp. 191–198. 1, 3

[3] M. Moravcik and M. Kontsek, "Overview of docker container orchestration tools," in *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, 2020, pp. 475–480. 3, 5, 8, 9

[4] E. Casalicchio and S. Iannucci, "The state-of-the-art in container technologies: Application, orchestration and security. concurrency and computation: Practice and experience," *Special Issue: Special Issue on Computational Intelligence Techniques for Industrial and Medical Applications (CITIMA2018). Special Issue on Cloud Computing, IoT, and Big Data: Technologies and Applications (CloudTech17)*, vol. 32, no. 17, sep 2020. [Online]. Available: https://doi.org/10.1002/cpe.5668 3

[5] A. Malviya and R. K. Dwivedi, "A comparative analysis of container orchestration tools in cloud computing," in *2022 9th International Conference on Computing for Sustainable Global Development (INDIACom)*, 2022, pp. 698–703. 5

[6] S. R. Lokhande and S. A. Kumar, "Deployment strategy using devops methodology: Cloud container based orchestration frameworks," in *2022 International Conference on Edge Computing and Applications (ICECAA)*, 2022, pp. 113–117. 5

[7] M. Aly, F. Khomh, and S. Yacout, "Kubernetes or openshift? which technology best suits eclipse hono iot deployments," in *2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)*, 2018, pp. 113–120. 5

[8] C.-H. Huang and C.-R. Lee, "Enhancing the availability of docker swarm using checkpoint-and-restore," in *2017 14th International Symposium on Pervasive Systems, Algorithms and Networks 2017 11th International Conference on Frontier of Computer Science and Technology 2017 Third International Symposium of Creative Computing (ISPAN-FCST-ISCC)*, 2017, pp. 357–362. 6

[9] Z. Zhong and R. Buyya, "A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources," *ACM Trans. Internet Technol.*, vol. 20, no. 2, apr 2020. [Online]. Available: https://doi.org/10.1145/3378447 10

[10] M. T. Goodrich, "Bin packing," accessed July 24th, 2024. [Online]. Available: https://ics.uci.edu/~goodrich/teach/cs165/notes/BinPacking.pdf 10