Hasil Penelitian
Yang Tidak Dipublikasikan

Tinjauan Pustaka
Orkestrasi Layanan Docker Swarm dan OpenShift Container Management

Berkah I. Santoso, ST, MTI
Guson P. Kuntarto, ST, M.Sc
Irwan Prasetya Gunawan, Ph.D
Yusuf Lestanto, ST, M.Sc, MBA

UNIVERSITAS BAKRIE

Fakultas Teknik dan Ilmu Komputer
Universitas Bakrie
Jakarta

2023/2024

LEMBAR PENGESAHAN HASIL PENELITIAN YANG TIDAK DIPUBLIKASIKAN

1.	Judul Penelitian	: Tinjauan Pustaka Orkestrasi Layanan Docker Swarm dan OpenShift Container Management
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 Oktober 2023 – 28 Desember 2023

Jakarta, 29 Desember 2023

Menyetujui,

Ketua Lembaga Penelitian dan Pengembangan

(**Deffi Ayu Puspito Sari, Ph.D.**)
NIDN: 0308078203

Peneliti Utama

(**Berkah I. Santoso, ST, MTI**.)
NIDN: 0309068003

Abstrak

Pengembangan aplikasi yang memiliki waktu rilis cepat, bersifat modular dan stabil merupakan hal wajib yang menjadi tuntutan organisasi atau perusahaan saat ini. Kecepatan pengembangan aplikasi modular tersebut membutuhkan sumber daya komputasi yang harus selalu tersedia, kapasitas sumber daya yang bersifat elastis dan orkestrasi varian sumber daya yang efisien. Variasi model sumber daya komputasi mulai dari 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 percepatan *time-to-deliver* aplikasi modern dari ranah *development* kepada *production*[1]. Terdapat beragam terobosan pada sisi penyediaan sumber daya komputasi terhadap kebutuhan pengembangan aplikasi secara *micro-services*–penambahan komponen-komponen layanan pada kode sumber aplikasi yang bersifat modular, adaptif dan berskala mikro, fungsional, membutuhkan dukungan layanan *multi-cloud* dengan pendekatan berbasis *container*. Beberapa platform orkestrasi *container* seperti Kubernetes dan pengelolaan Docker memiliki keterbatasan berikut keunggulan masing-masing. Kubernetes dirasakan masih memiliki keterbatasan kinerja secara keseluruhan apabila dibandingkan dengan Docker, sedangkan Docker pada mode Swarm terlihat memiliki keterbatasan pada pengelolaan platform yang bersifat heterogen[2]. Tantangan tim *developer* dan tim *operations* (DevOps) adalah bagaimana memanfaatkan kekayaan fitur sekaligus melakukan orkestrasi antara fleksibilitas Kubernetes dengan kinerja maksimal Docker pada mode Swarm untuk membentuk aplikasi modern berbasis *micro-services* dalam rangka peningkatan waktu pengembangan dan implementasi. Penulis mencoba untuk memberikan tinjauan literatur yang diharapkan dapat bermanfaat bagi para pengembang aplikasi modern yang masih membutuhkan mekanisme orkestrasi terintegrasi terhadap 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. Kubernetes, Docker Swarm dan Apache Mesos dapat menangani cluster container Docker secara efisien dan mampu melakukan pengelolaan beragam layanan, walaupun pendekatan teknis antara ketiga platform orkestrasi tersebut memiliki perbedaan. Beban overhead Kubernetes memberikan hasil yang cukup tinggi apabila dibandingkan dengan Docker pada mode Swarm yang memiliki beban overhead lebih rendah. Sedangkan pada Apache Mesos memberikan unjuk kerja yang tinggi pada mekanisme berbagi pakai sumber daya komputasi untuk beragam *platform*. Pada hasil keseluruhan, kita dapat melihat kinerja dan overhead ketiga platform orkestrasi container tersebut, yaitu Kubernetes memiliki kinerja lebih rendah dibandingkan dengan Docker pada mode Swarm atau Apache Mesos. Pada sisi lain, Kubernetes yang dijalankan pada implementasi desain kompleks menghasilkan sifat response yang lebih fleksibel dan memiliki kemampuan lebih baik apabila dibandingkan dengan Docker pada mode Swarm dan Apache Mesos, sehingga kedua *orchestration containerized platform* tersebut memiliki karakteristik unik masing-masing.

Daftar Isi

1 Pendahuluan **3**
1.1 Latar Belakang . . . 3
1.2 Rumusan Masalah . . . 4
1.3 Batasan Masalah . . . 4
1.4 Tujuan Penelitian . . . 4
1.5 Sistematika Penulisan . . . 4

2 Mengenal Arsitektur Docker Swarm Container Management **6**
2.1 Task . . . 7
2.2 Docker Services . . . 7
2.3 Raft Consensus Group . . . 8
2.4 Internal Distributed Stated Store . . . 8
2.5 Manager Nodes . . . 8
2.6 Worker Nodes . . . 9
2.7 Keuntungan Penggunaan Docker Swarm Container Management . . . 9
2.8 *Drawbacks* dari Orkestrasi Docker Swarm Container Management . . . 10

3 Mengenal Arsitektur OpenShift Container Management **11**
3.1 Control Plane Nodes . . . 12
3.2 Compute Nodes . . . 13
3.3 Keuntungan Penggunaan OpenShift Container Management . . . 15
3.4 *Drawbacks* dari Orkestrasi OpenShift Container Management . . . 15

4 Tinjauan Algoritma Orkestrasi Docker Swarm dan OpenShift Container Management **17**
4.1 Algoritma terkait Penjadwalan . . . 17
4.2 Algoritma terkait Autoscaling . . . 18
4.3 Algoritma terkait Penjadwalan Ulang . . . 19

5 Kesimpulan **21**

Bab 1

Pendahuluan

1.1 Latar Belakang

Trend pemanfaatan *container* pada layanan *cloud computing* saat ini telah menjadi paradigma penyediaan sumber daya komputasi untuk pengembangan aplikasi korporat yang digunakan secara cepat dan luas. Beberapa tujuan penggunaan *container* adalah peningkatan modularitas layanan aplikasi, kemudahan pengelolaan layanan modul aplikasi tanpa harus mengganggu layanan aplikasi lainnya, peningkatan fleksibilitas penempatan dan perpindahan layanan aplikasi baik pada *virtual server* ataupun *physical server* serta kemudahan mekanisme pembagian beban komputasi untuk modul layanan aplikasi [3]. Pengelolaan *container* menggunakan platform Docker Swarm, Kubernetes dan OpenShift yang tersedia pada beberapa penyedia layanan *public cloud computing* seperti Google Cloud®, Amazon Web Services (AWS)®, Alibaba®mengadopsi teknik virtualisasi berikut *virtual machine* (VM) dan *container* dalam rangka otomasi implementasi aplikasi pada infrastruktur komputasi. Penerapan *container* dengan sifat modular dan lincah dalam kaitannya terhadap penyediaan sumber daya komputasi bagi kebutuhan percepatan implementasi aplikasi 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 aman, dapat diakses kapan saja, handal dan dinamis mengikuti besaran beban komputasi tanpa harus direpotkan oleh kerumitan pengelolaan *container*. Orkestrasi pengelolaan *container*, saat ini merupakan hal yang bersifat *mandatory*, baik bagi pihak *developer* maupun pihak *cloud administrator* dalam rangka memastikan semua obyek pembentuk berfungsi secara maksimal [1]. Beberapa terobosan pada sisi pengembangan aplikasi secara *micro-services*–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 *container* seperti Docker dan Kubernetes 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 pada mode Swarm masih memiliki keterbatasan pada fleksibilitas untuk platform yang bersifat heterogen, sedangkan Kubernetes juga masih memiliki keterbatasan kinerja secara keseluruhan apabila dibandingkan dengan Docker [2]. Apabila tim *developer* dan *operations* (DevOps) dapat memanfaatkan serta melakukan orkestrasi antara fleksibilitas Kubernetes dengan kinerja maksimal Docker pada mode 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 *orchestration 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 orkestrasi *container* tepat guna pada layanan *cloud computing* dengan menggunakan pendekatan deskripsi komparasi terkait bagaimana penentuan *framework* orkestrasi *container* tepat guna sesuai dengan kebutuhan komputasi untuk sumber daya komputasi dalam rangka implementasi aplikasi modern.

1.3 Batasan Masalah

Ruang lingkup pada pembahasan ini meliputi beberapa hal sebagai berikut, yaitu :

1. Framework orkestrasi *container* yang diteliti adalah Docker Swarm dan OpenShift Container Management.
2. Fokus pembahasan terkait aspek fitur dan unjuk kerja orkestrasi pada kedua framework *container*.

1.4 Tujuan Penelitian

Penggunaan pendekatan tinjauan fitur unggulan dan unjuk kerja pada laporan ini ditujukan untuk identifikasi terkait bagaimana penentuan *framework* orkestrasi *contaier* tepat guna sebagai pengelola *container instance* yaitu Docker Swarm dan OpenShift Container Management. Penentuan tersebut bertujuan agar terdapat ketepatan penggunaan *framework* orkestrasi *container* dengan pemenuhan prasyarat dan kondisi sebelum dilakukan *deployment* aplikasi modern pada infrastruktur komputasi terdistribusi.

1.5 Sistematika Penulisan

Adapun sistematika penulisan pada laporan penelitian ini meliputi :

1. Bab 1 membahas latar belakang permasalahan dari munculnya beberapa *framework* orkestrasi *container* seperti Docker Swarm dan OpenShift Container Management pada infrastruktur yang bersifat terdistribusi. 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 pengenalan *framework* orkestrasi *container* OpenShift Container Management, berikut keuntungan dan *drawbacks* yang menyertai.
4. Bab 4 membahas tinjauan unjuk kerja dari masing-masing *framework* orkestrasi *container*, yaitu Docker Swarm dan OpenShift Container Management.

5. Bab 5 memberikan kesimpulan dari hasil penelitian terkait tinjauan fitur dan unjuk kerja pada penentuan *framework* orkestrasi *container*.

Bab 2

Mengenal Arsitektur Docker Swarm Container Management

Manajemen *container* pada arsitektur komputasi mutlak diperlukan bagi tersedianya *framework* orkestrasi *container* untuk aplikasi modern. Perkembangan dan pilihan *framework* orkestrasi *container* cukup banyak tersedia, mulai dari solusi berbasis *open source software* maupun solusi bagi korporat yang bersifat *proprietary software*. Review terkait fitur dan komparasi unjuk kerja dari maing-masing *framework* manajemen *container* diperlukan oleh *solution architect* dan penanggung jawab infrastruktur TI dalam rangka penyediaan solusi tepat guna bagi organisasi dan perusahaan.

Dari beberapa solusi *framework container management* seperti Kubernetes, Docker Swarm, Apache Mesos dan OpenShift yang memiliki aspek keamanan, implementasi, stabilitas, skalabilitas, instalasi cluster[5], maka penulis memberikan perhatian lebih kepada *framework container management* yang memiliki dukungan komunitas yang kuat dan memiliki persebaran pengguna yang luas, yaitu Docker Swarm dan OpenShift Container Management[6][7]. Kedua *framework container management* tersebut memiliki keunggulan dan karakteristik masing-masing seperti kemudahan penggunaan pada Docker Swarm dan peningkatan orkestrasi keamanan pada OpenShift sehingga diperlukan pendekatan komprehensif dalam rangka pemilihan arsitektur implementasi sumber daya komputasi aplikasi modern.

Docker Swarm *framework container management* dirilis pada lingkungan produksi tahun 2016 dengan ciri khas kesederhanaan pada pustaka orkestrasi dan kemudahan integrasi komponen pada semua lingkungan kerja Docker, misal integrasi pada Docker Application Programming Interface (API). Docker Swarm memiliki mekanisme unik dalam mengelola dan melakukan konfigurasi *container* yang pada awal pengembangan ditujukan untuk berjalan secara mandiri, dengan mekanisme pengembangan terintegrasi menjadi orkestrasi sekumpulan *container* berjumlah banyak melalui perantaraan jaringan [3].

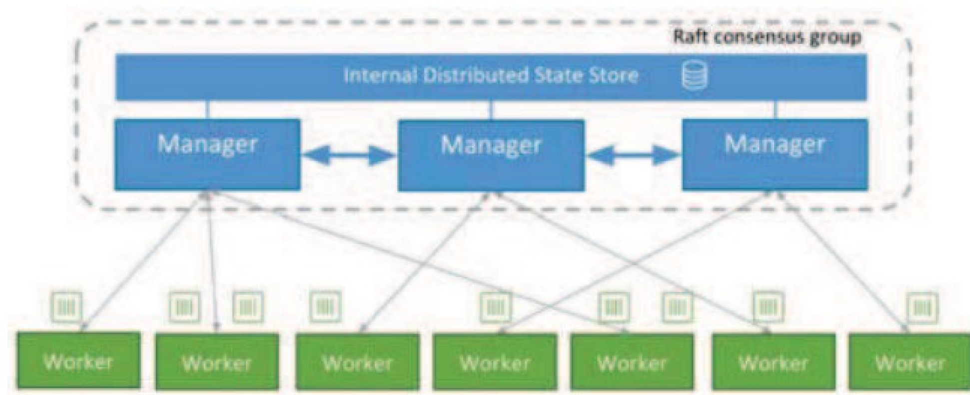
Pengembangan orkestrasi pengelolaan *container* telah dilakukan menggunakan bahasa pemrograman Go Lang yang bertujuan untuk dapat langsung berintegrasi dengan Docker API, sehingga semua fitur yang terdapat pada *container* Docker dapat digunakan dan diterapkan pada Docker Swarm. Pengembang aplikasi modern 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 [3]

2.1 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 terdiri dari banyak **task** yang membentuk layanan Docker, sehingga dapat memungkinkan orkestrasi 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, aman dan efisien.

2.3 Raft Consensus Group

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

Salah satu nilai utama pada Docker Swarm terkait aspek *fault tolerance* adalah bahwa Docker Swarm memperbolehkan eksistensi lebih dari satu *control node* dalam *cluster container* tunggal, akan tetapi Docker Swarm juga memperbolehkan eksistensi 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 Internal Distributed Stated Store

Internal Distributed Stated Store 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 dapat dikelola dengan baik.

2.5 Manager Nodes

Manager Nodes memiliki peran sebagai pengelola *node container*, dengan kelengkapan komponen sebagai berikut :

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

Worker nodes menerima tugas-tugas beban kerja komputasi secara langsung dari *control node* untuk dapat dijalankan dengan 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

Beberapa keuntungan penggunaan Docker Swarm *Container Management* adalah sebagai berikut :

- Pengelolaan *container cluster* langsung menggunakan lingkungan kerja Docker Command Line Interface (CLI) tanpa adanya tambahan *software* pengelolaan orkestrasi lainnya.
- Desain Docker Swarm bersifat terdesentralisasi, sehingga terdapat mekanisme perbedaan 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 yang bersifat *declarative* untuk mendefinisikan 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 *Drawbacks* dari Orkestrasi Docker Swarm Container Management

Pada versi Docker Swarm bulan Mei 2019, beberapa *drawbacks* yang masih dalam penyempurnaan lebih lanjut adalah sebagai berikut :

- Docker Swarm belum terdapat mekanisme peningkatan skalabilitas aplikasi secara otomatis.
- Docker Swarm belum menudukung 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

Mengenal Arsitektur OpenShift Container Management

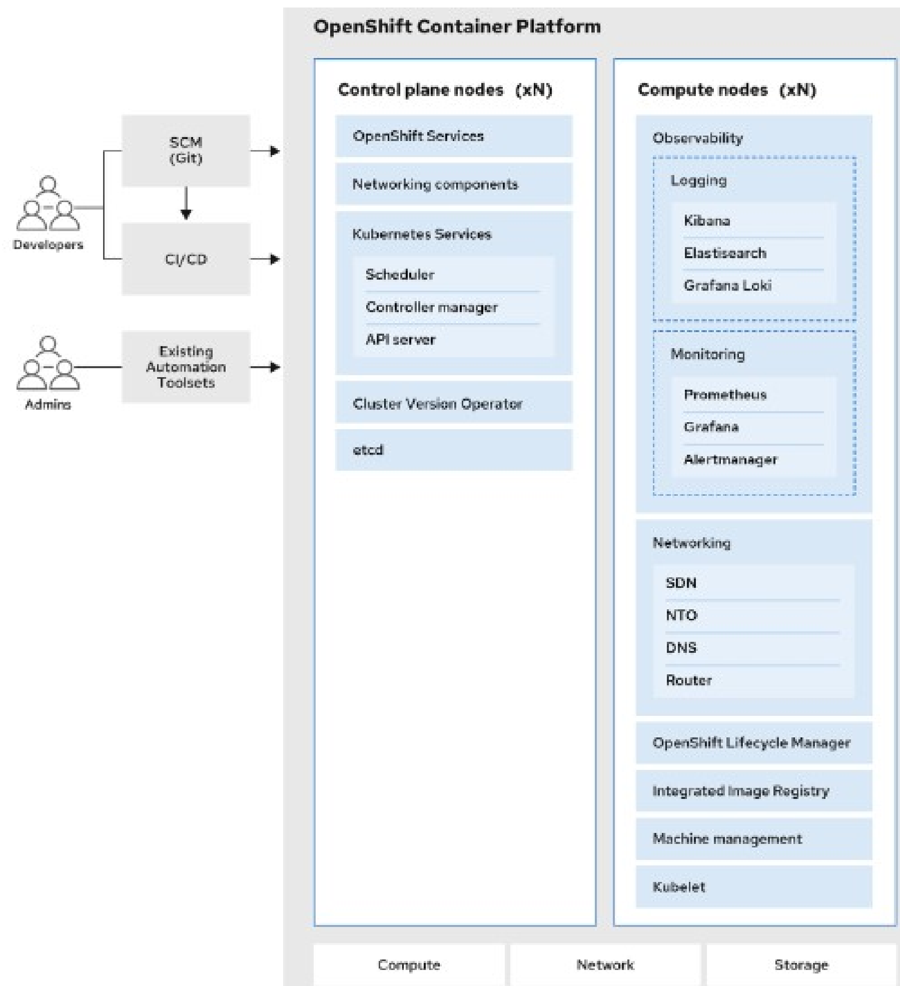
Setelah ulasan mengenai *framework container management* Docker Swarm disampaikan penulis pada Bab 2 sebelumnya, maka pada Bab 3 ini, penulis mencoba menyajikan uraian terkait OpenShift Container Management yang tersedia untuk versi komunitas maupun untuk versi korporasi yang dipaketkan oleh perusahaan perangkat lunak berbasis *open source* dan sistem operasi korporat yaitu Red Hat®. Banyaknya solusi *framework container management* seperti Docker Swarm, OpenShift, Kubernetes dan Apache Mesos dimana masing-masing solusi memiliki aspek stabilitas, skalabilitas, kemudahan instalasi *cluster*, keamanan dan implementasi [5], terdapat *framework container management* yang memiliki persebaran pengguna yang luas dan dukungan komunitas yang kuat, yaitu Docker Swarm dan OpenShift Container Management[6][7].

Framework container management tersebut memiliki karakteristik dan keunggulan masing-masing seperti peningkatan orkestrasi keamanan pada OpenShift dan kemudahan penggunaan pada Docker Swarm, sehingga diperlukan pendekatan yang komprehensif untuk menentukan arsitektur implementasi sumber daya komputasi bagi aplikasi modern, dimana masing-masing layanan aplikasi diimplementasikan pada masing-masing *container cluster* yang berjalan pada mekanisme orkestrasi secara mandiri. Melalui pendekatan tersebut, pengelola infrastruktur *container* dan pengembang aplikasi modern dapat melakukan prediksi peningkatan beban kerja komputasi dan proses *maintenance* aplikasi tanpa mengganggu keseluruhan layanan aplikasi.

OpenShift dapat kita pahami sebagai suatu *container platform* yang disediakan sebagai bentuk dari Platform as a Service (PaaS). *Platform* OpenShift dikembangkan oleh Red Hat®sebagai bagian dari rangkaian perangkat lunak untuk pengelolaan *container* dengan menggunakan bahasa pengembangan GoLang dan AngularJS dan berlisensi perangkat lunak dibawah pengelolaan Apache Foundation. OpenShift yang memiliki kemiripan dengan Kubernetes, suatu *container platform* untuk membantu para pengembang aplikasi berbasis cloud dengan waktu rilis cepat, digunakan pada lingkungan korporasi secara luas, sehingga sering disebut sebagai *enterprise Kubernetes* [3].

OpenShift mengelola *autoscaling* pada tingkatan *cluster* untuk melakukan pengaturan ukuran *cluster* pada saat terjadi kegagalan pada penjadwalan tugas-tugas komputasi karena kekurangan sumber daya komputasi. Pada kondisi tersebut OpenShift hanya dapat diperbolehkan untuk meningkatkan skalabilitas obyek-obyek *container* yang sejenis untuk setiap siklus pengaturan skalabilitas pada mekanisme operasi *autoscaling* dalam rangka pembatasan skalabilitas beban kerja secara dinamis. Oleh sebab itu mekanisme *autoscaling* pada tingkatan *cluster* dengan menggunakan sumber daya komputasi yang beragam merupakan pengembangan lebih lanjut untuk peningkatan efisiensi dan pengelolaan *quality of service* [8]. Agar

pembaca memiliki penggambaran arsitektur OpenShift *Container Management*, berikut ini merupakan arsitektur OpenShift :



Gambar 3.1: Arsitektur OpenShift Container Management [9]

3.1 Control Plane Nodes

Konsep sederhana yang terdapat pada OpenShift Container Management adalah sebagai berikut :

- Proses awal berupa konfigurasi salah satu *worker node* untuk menjalankan beban kerja komputasi pada *container*.
- Pengelolaan implementasi container untuk menangani beban kerja komputasi dari satu *control plane node* ke *control plane node* yang lain untuk proses orkestrasi *container*.
- Mekanisme enkapsulasi beberapa *container* dalam suatu obyek implementasi yang disebut sebagai **pod**. Pod digunakan untuk menyediakan metadata tambahan terkait dengan *container*. Pod juga memiliki kemampuan untuk membentuk kelompok beberapa *container* kedalam entitas implementasi secara tunggal dan mandiri.
- OpenShift membuat daftar aset *container* berikut atribut penyertanya. Sebagai contoh, suatu layanan direpresentasikan oleh sekumpulan **pods** dan kebijakan komputasi yang mendefinisikan

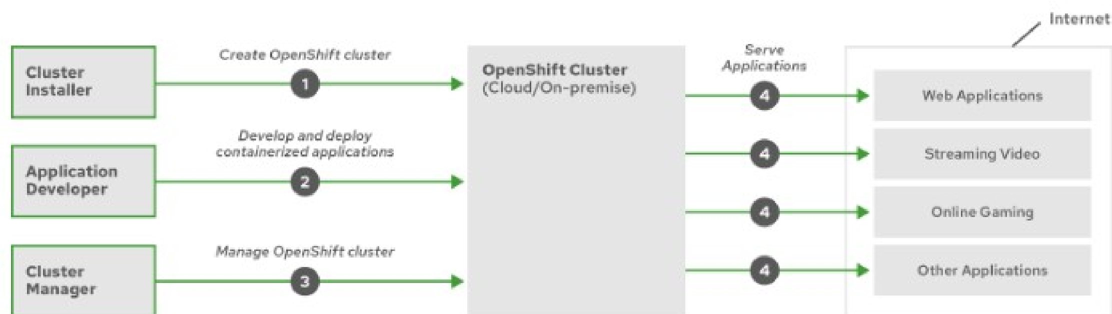
mengenai cara pods tersebut diakses. Kebijakan komputasi terhadap pods tersebut memungkinkan *container* untuk terhubung pada layanan yang dibutuhkan walaupun *container* tersebut belum memiliki alamat IP untuk terkoneksi dengan layanan tersebut.

- OpenShift memiliki mekanisme pengelolaan replikasi yang merupakan asset khusus untuk mengindikasikan jumlah replika *pod* yang dibutuhkan untuk dapat berjalan pada suatu waktu eksekusi, sehingga pengelola *container* dapat meningkatkan skalabilitas aplikasi secara otomatis agar mengikuti permintaan kebutuhan layanan aplikasi.

Berikut ini merupakan siklus dasar OpenShift *Container Platform*, meliputi :

1. Proses pembuatan *cluster* pada OpenShift Container Platform.
2. Pengelolaan *cluster*.
3. Pengembangan dan implementasi aplikasi modern.
4. Proses peningkatan skalabilitas sumber daya komputasi *container* untuk aplikasi.

Agar pembaca memiliki penggambaran penjelasan terkait Control Plane Node, dapat pembaca lihat pada gambar berikut ini :



Gambar 3.2: High level OpenShift Container Platform overview [9]

3.2 Compute Nodes

Pengelolaan *compute node* sebagian besar terletak pada konfigurasi sekumpulan *container* sebagai *virtual machine*. Sebuah kumpulan *virtual machine* merupakan kelompok *compute node* dalam *cluster* yang memiliki konfigurasi sama untuk keperluan kemudahan pengelolaan. Pengelola infrastruktur *container* dapat melakukan edit konfigurasi pada sekumpulan *virtual machine* seperti peningkatan skalabilitas komputasi *container*, penambahan label pada *node* dan penambahan tanda pada *container*.

Tipe obyek pada *compute node* mendefinisikan alokasi virtual Central Processing Unit (vCPU) dan alokasi memori untuk masing-masing *compute node* pada sekumpulan *virtual machine* [10]. Suatu *virtual machine* merupakan unit dasar yang menjelaskan suatu *host* untuk dikelola pada *worker node*. Sedangkan komponen sumber daya pada *compute node* yaitu MachineSet merupakan sekumpulan *virtual machine* untuk kebutuhan komputasi. Apabila pengelola infrastruktur *container* membutuhkan peningkatan maupun penurunan skalabilitas *virtual machine*, maka diperlukan konfigurasi pada perubahan jumlah replika pada sekumpulan *virtual machine* yang telah ditetapkan sebelumnya. Sekumpulan *virtual machine* merupakan tingkatan konstruksi yang lebih tinggi dari sekumpulan sumber daya komputasi

pada *container*, berfungsi sebagai pengelola sekumpulan *clone* dengan konfigurasi yang sama pada zona *cluster* eksisting.

Pada kondisi *default*, sebuah *cluster* dibuat dengan menggunakan satu kelompok *virtual machine*. Pengelola infrastruktur *container* dapat menambahkan satu kelompok *virtual machine* tambahan terhadap *cluster* eksisting, melakukan modifikasi kondisi *default* dari kelompok *virtual machine* dan menghapus kelompok *virtual machine*. Beberapa kelompok *virtual machine* dapat berada pada suatu *cluster* tunggal dan kelompok *virtual machine* tersebut dapat memiliki tipe yang berbeda baik pada aspek ukuran *node* maupun jumlah *node* yang dihasilkan.

Apabila pengelola infrastruktur *container* membuat sekumpulan *virtual machine* pada *cluster* dengan zona ketersediaan yang bermacam-macam, maka satu kelompok *virtual machine* dapat memiliki 3 (tiga) zona, yaitu satu kelompok untuk masing-masing zona pada *cluster* dari *virtual machine* yang dikelola. Sedangkan pada proses penghapusan kelompok *virtual machine*, maka daftar kelompok *virtual machine* tersebut akan dihapus dari semua zona, sehingga dapat terjadi dampak yang bersifat multiplikasi atau berantai. Mekanisme efek berantai tersebut akan mengkonsumsi *quota* sumber daya komputasi pada saat pengelola infrastruktur *container* menerapkan penggunaan kelompok *virtual machine* pada *cluster* dengan area ketersediaan zona lebih dari satu [10].

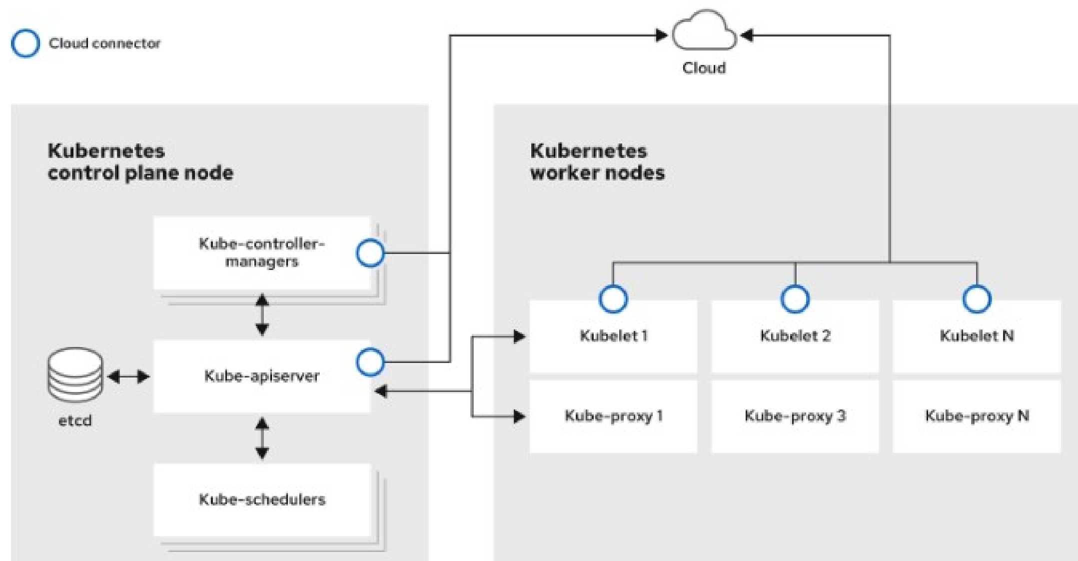
Beberapa komponen berikut ini merupakan komponen dari suatu *node* yang bertanggung jawab untuk mengelola *pods* yang sedang berjalan. Komponen berikut ditujukan juga sebagai penyedia lingkungan kerja komputasi Kubernetes atau OpenShift.

- *Container runtime*, bertanggung jawab untuk menjalankan *container*. OpenShift menawarkan beberapa lingkungan pustaka komputasi seperti containerd, cri-o, rktlet, dan Docker.
- *Kubelet*, bertanggung jawab untuk menjalankan *node* dan membaca keterangan pembuktian paket-paket *container*. Kubelet berfungsi untuk memastikan bahwa beberapa *container* yang telah didefinisikan dapat dilakukan inisiasi dan pada proses menjalankan fungsinya. Proses pada *Kubelet* mengelola kondisi kinerja komputasi dan *node server* serta pengaturan jaringan *container* yang hanya dibuat oleh OpenShift.
- Kube-proxy, bertanggung jawab untuk dapat menjalankan setiap *node* pada *cluster* dan mengelola *network traffic* antar sumber daya komputasi pada OpenShift, serta memastikan bahwa lingkungan jaringan *container* terisolasi serta dapat diakses.
- Domain Name Service (DNS), merupakan server DNS yang memberikan layanan OpenShift terhadap pengelolaan DNS *records*. Pada saat layanan OpenShift dilakukan konfigurasi pertama kali, maka layanan DNS akan selalu digunakan untuk memberikan nama kepada masing-masing *container* secara otomatis dalam rangka mempermudah pengembang aplikasi dan pengelola infrastruktur *container*.

Pada operasi pembacaan *node* memungkinkan seorang administrator infrastruktur *container* atau pengembang aplikasi modern untuk mendapatkan informasi mengenai *node* pada *cluster* OpenShift Container Platform. Operasi pembacaan *node* tersebut dapat terdiri dari :

- Menampilkan daftar semua *node* pada suatu *cluster*.
- Mendapatkan informasi mengenai *node*, seperti penggunaan memori, Central Processing Unit (CPU), status *container* dan usia penggunaan.
- Menampilkan daftar *pods* pada *node*.

Pembaca dapat melihat gambar berikut untuk mendapatkan penggambaran terkait *compute node* pada OpenShift sebagai Kubernetes *enterprise* yang merupakan turunan dari Kubernetes *container platform*.



Gambar 3.3: Komponen suatu node untuk mengelola pods pada Kubernetes atau OpenShift [10]

3.3 Keuntungan Penggunaan OpenShift Container Management

Beberapa keuntungan komputasi dari penggunaan OpenShift *Container Management* adalah sebagai berikut [11]:

- OpenShift mendukung banyak bahasa pengembangan aplikasi seperti Go Lang, NodeJS, Ruby, Python, Java, Perl, PHP dan lainnya.
- OpenShift memiliki fitur implementasi aplikasi menggunakan metode Development and Operations (DevOps) sehingga membantu percepatan implementasi aplikasi.
- OpenShift memiliki kemampuan untuk meningkatkan skalabilitas aplikasi modern yang ditempatkan pada *container* baik secara otomatis maupun secara manual.
- OpenShift memiliki pemeriksaan aspek keamanan yang telah ditanamkan pada *container* dan lingkungan pengembangan aplikasi.
- OpenShift dengan bantuan komponen aplikasi Prometheus dapat melakukan mekanisme monitoring aplikasi.
- OpenShift menggunakan mekanisme penyeragaman implementasi terkait kebijakan sumber daya *container* untuk pengguna.
- OpenShift memiliki kompatibilitas terhadap lingkungan komputasi Kubernetes *container cluster*.
- OpenShift memberikan fasilitas untuk mempermudah migrasi pada sistem operasi yang lebih baru tanpa harus mengganggu struktur orkestrasi *container* itu sendiri.

3.4 *Drawbacks* dari Orkestrasi OpenShift Container Management

OpenShift yang digunakan sebagai Platform as a Service (PaaS) turunan dari Kubernetes, dibangun menggunakan *command line syntax* tersendiri yaitu kubectl, sehingga memiliki kompleksitas implementasi dan pengelolaan komponen yang cukup tinggi bagi pengembang aplikasi dan pengelola infrastruktur *container*.

Beberapa *drawbacks* dari penggunaan OpenShift *container management* adalah sebagai berikut [12] :

- Kompleksitas, dimana OpenShift menggunakan banyak komponen untuk membentuk platform yang tidak sederhana bagi organisasi atau korporasi, terutama untuk investasi training staff pengelola infrastruktur *container* dan waktu untuk mempelajari keseluruhan komponen platform OpenShift tersebut.
- Biaya, dimana OpenShift merupakan produk komersil yang dipaketkan oleh Red Hat®dengan model bisnis *support subscription* atau lisensi penggunaan sesuai periode waktu tertentu. Aspek biaya tersebut menjadikan solusi OpenShift membutuhkan tambahan biaya investasi dari platform *container management* berbasis *open source* lainnya.
- *Vendor Lock-In*, hal ini dapat terjadi ketika kurangnya sumber daya pengetahuan yang dimiliki staff perusahaan terhadap OpenShift yang dipaketkan oleh Red Hat®.
- Terbatasnya dukungan untuk beberapa bahasa pengembangan dan *framework*, dimana tidak semua bahasa pemrograman, khususnya yang bersifat *proprietary* dapat langsung digunakan pada OpenShift *container management platform*.

Bab 4

Tinjauan Algoritma Orkestrasi Docker Swarm dan OpenShift Container Management

Penggunaan beberapa algoritma pada orkestrasi Docker Swarm dan OpenShift *Container Management* adalah untuk mendapatkan efisiensi atas beban kerja komputasi pada berbagai macam konfigurasi sumber daya komputasi dalam 3 (tiga) cara, diantaranya adalah [8]:

- 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.

4.1 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 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*.

Algoritma BFD dan Time-bin BFD 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 → 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
```

4.2 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.

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 :

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

4.3 Algoritma terkait Penjadwalan Ulang

Apabila rasio utilisasi sumber daya komputasi dari *node* yang berjalan di 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**

Bab 5

Kesimpulan

Docker Swarm dan OpenShift dapat menangani *cluster container* secara efektif dan mampu melakukan pengelolaan beragam layanan, walaupun pendekatan teknis antara kedua *container orchestrator platform* tersebut memiliki perbedaan, dimana Docker Swarm menggunakan lingkungan kerja Docker sedangkan OpenShift menggunakan turunan Kubernetes yang telah disederhanakan.

Beban *overhead* pada OpenShift memberikan hasil yang tinggi apabila dibandingkan dengan Docker Swarm dengan beban *overhead* lebih rendah. Pada hasil keseluruhan, kita dapat melihat kinerja dan *overhead* kedua *container orchestrator platform* tersebut, yaitu OpenShift memiliki kinerja lebih rendah dibandingkan dengan Docker Swarm.

Pada sisi lain, OpenShift yang dijalankan dengan desain yang lebih kompleks dapat menghasilkan respons komputasi yang lebih fleksibel. Selain itu OpenShift memiliki kemampuan lebih baik dalam penyederhanaan penyelesaian tugas-tugas komputasi apabila dibandingkan dengan Docker Swarm.

Penggunaan beberapa algoritma untuk mekanisme penjadwalan, *autoscaling* dan penjadwalan ditujukan agar *container management platform* mendapatkan efisiensi atas beban kerja komputasi pada berbagai macam konfigurasi sumber daya komputasi dalam 3 (tiga) cara, yaitu 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 dan mekanisme penjadwalan ulang terhadap *cluster* dan implementasi ulang terhadap obyek-obyek *vritual machine* yang memiliki utilisasi rendah.

Beberapa hal yang terbuka untuk dikembangkan lebih lanjut dari penelitian ini adalah karakteristik respons *container* dari kedua *framework container management* Docker Swarm dan OpenShift terhadap adanya 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 kecepatan *backup and recovery* layanan secara adaptif pada kedua *framework container management* apabila terjadi perawatan sistem atau terjadi gangguan teknis pada *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, 6, 7, 9, 10, 11

[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. 6, 11

[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. 6, 11

[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. 6, 11

[8] 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 11, 17

[9] K. Alexander, "Introduction to openshift container platform," accessed November 25th, 2023. [Online]. Available: https://docs.openshift.com/container-platform/4.15/architecture/architecture.html#architecture 12, 13

[10] B. M. Munilla, "Managing compute nodes," accessed November 30th, 2023. [Online]. Available: https://docs.openshift.com/dedicated/osd_cluster_admin/osd_nodes/osd-managing-worker-nodes.html 13, 14, 15

[11] S. India, "Kubernetes vs. openshift: A thorough comparison," accessed November 25th, 2023. [Online]. Available: https://hackernoon.com/kubernetes-vs-openshift-a-detailed-comparison-7r3z53zlv 15

[12] A. Rampersad, "Openstack vs openshift: Understanding the differences and choosing the right platform for your business," accessed November 30th, 2023. [Online]. Available: https://openmetal.io/docs/edu/openstack-vs-openshift/ 16