Multi-Objective Optimization untuk Container Scheduling Menggunakan Algoritma Ɛ-NSGA-II

Eka Wijaya Susilo, Henry Novianus Palit, Rolly Intan
Program Studi Teknik Informatika Fakultas Teknologi Industri Universitas Kristen Petra
Jl. Siwalankerto 121 – 131 Surabaya 60236
Telp. (031) – 2983455, Fax. (031) – 8417658
E-Mail: ekawijayasusilo@gmail.com, hnpalit@petra.ac.id, rintan@petra.ac.id

ABSTRAK

Teknologi *containerization* sebagai implementasi virtualisasi di *level* OS meraih popularitas di beberapa tahun terakhir untuk menggantikan beberapa peranan *virtual machine.* Dalam implementasinya, seringkali dibutuhkan sejumlah *container* untuk menjalankan sebuah peran, sehingga diperlukan sistem orkestrasi untuk proses scheduling dan management *container.* Namun *container scheduler* yang tersedia pada sistem orkestrasi seperti Kubernetes umumnya hanya berfokus pada ketersediaan *system resource* dan *general threshold.* Padahal optimasi dalam hal efisiensi *cluster* sambil mempertahankan performa *cluster* tentulah penting untuk diimplementasikan sehingga dapat meminimalkan biaya operasional.

Optimasi pada *container scheduler* dilakukan secara multi-objektif dengan fokus pada efisiensi *cluster* dan *throughput* aplikasi yang di-*schedule.* Optimasi dilakukan dengan memanipulasi jumlah replika *container* yang menyala dan alokasinya di dalam *cluster* berdasarkan jumlah *request* yang diterima aplikasi. Proses optimasi dilakukan dengan menggunakan algoritma *Epsilon-Dominance Non-Dominated Sorting Genetic Algorithm II* (Ɛ-NSGA-II).

Hasil pengujian dalam *cluster* berisi 5 *node* menunjukkan bahwa proses optimasi *scheduling* dapat menghasilkan keputusan alokasi *container* yang optimal dan sesuai target yang ditentukan *user.* Proses algoritma Ɛ-NSGA-II untuk menghasilkan alokasi tersebut juga dapat selesai dalam rentang waktu yang pendek.

Kata Kunci: *genetic algorithm, multi-objective optimization, Kubernetes, container*

ABSTRACT

Containerization technology as an implementation of virtualization on OS level has gained popularity in recent years to replace some roles of virtual machines. In its implementation, a number of containers it is often needed to carry out a role. Thus an orchestration system is needed for container scheduling and management. Container scheduler available on orchestration systems like Kubernetes usually just focus on system resource availability and other general threshold. Whereas optimization in terms of cluster efficiency, while maintaining cluster performance, is certainly important to implement as to minimize operational costs.

Optimization in the container scheduler is done through a multi-objective manner with focus on cluster efficiency and scheduled application throughput. Optimization is done by manipulating the number of container replicas and its allocation in the cluster based on the number of requests received by the application. The optimization process is carried out using the Epsilon-Dominance Non-Dominated Sorting Genetic Algorithm II (Ɛ-NSGA-II) algorithm.

The test results in a cluster containing 5 nodes indicate that scheduler optimization can produce optimal container allocation decisions and meet user-specified targets. The process of the Ɛ-NSGA-II algorithm to produce these allocations can also be completed in a short span of time.

Keywords: *genetic algorithm, multi-objective optimization, Kubernetes, container*

1. PENDAHULUAN

Teknologi *containerization* sebagai implementasi virtualisasi di *level* OS meraih popularitas di beberapa tahun terakhir untuk menggantikan beberapa peranan *virtual machine*, untuk menjalankan *server application* dalam mengimplementasikan konsep-konsep terbaru dalam *software development*, seperti *DevOps* dan *microservices architecture*. Dalam implementasinya, *server application* tersebut umumnya melibatkan sejumlah *container* sekaligus. Dalam kondisi ini, diperlukan sistem untuk *scheduling*, *management*, dan *provisioning* dari setiap *container* yang berjalan dalam suatu *cluster*, atau lebih dikenal sebagai *container orchestration system*. Namun *container scheduler* yang tersedia pada sistem orkestrasi seperti Kubernetes umumnya hanya berfokus pada ketersediaan *system resource* dan *general threshold* [6]. Hal ini membuat pilihan optimasi *scheduling* pada suatu *container orchestration system* menjadi terbatas. Padahal melakukan optimasi *scheduling* yang berfokus pada efisiensi *cluster* sambil mempertahankan performa *cluster* tentulah penting untuk diimplementasikan sehingga dapat meminimalkan biaya operasional.

Scheduler dibuat untuk berjalan pada sistem Kubernetes dan melakukan proses optimasi alokasi *container* pada aplikasi berarsitektur *microservices* menggunakan algoritma Ɛ-NSGA-II dengan fokus pada:

1. Efisiensi *Cluster*
   Memanipulasi alokasi *container* pada *cluster* untuk meminimalkan jumlah *node* yang aktif menjalankan *container* dari aplikasi *microservices*.
2. *Balanced Throughput Hosted Application*
   Melakukan *scaling* jumlah *container* yang menjalankan tiap *microservice* dari aplikasi *microservices* berdasarkan jumlah HTTP *request* yang diterima pada tiap *endpoint* dan target *request* dan limitasi konsumsi *resource* yang diinputkan *user*.

Setelah proses optimasi *scheduling* selesai maka *scheduler* dapat mengimplementasikan keputusan alokasi *container* yang dihasilkan ke dalam *cluster* Kubernetes.

2. DASAR TEORI

2.1 Kubernetes

Kubernetes adalah *open-source container orchestration system* yang diciptakan oleh Google dan sekarang di-*maintain* oleh *Cloud Native Computing Foundation*. Sebagai *container orchestration system*, Kubernetes digunakan untuk mengotomasi *deployment*, *scaling*, dan manajemen aplikasi berbasis *container.* Kubernetes ditulis dalam bahasa pemrograman Go.

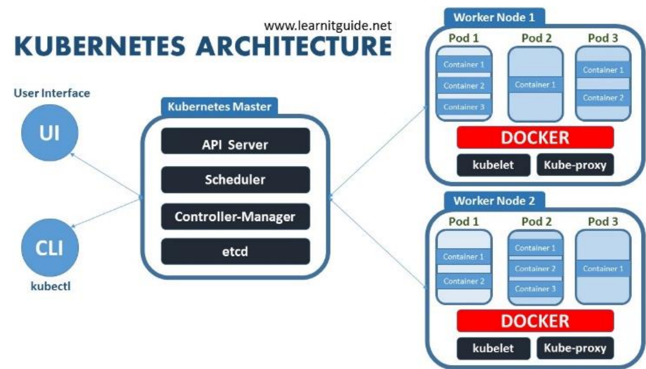
2.1.1 Konsep

Untuk dapat memahami mekanisme kerja Kubernetes, diperlukan pemahaman akan konsep-konsep berikut :

1. *Pod* : Unit terkecil yang dapat di-*deploy* di Kubernetes. Sebuah *pod* terdiri dari satu atau lebih *container*. *Pod* didesain untuk memiliki *lifetime* yang pendek, mudah di-*create* dan di-*destroy*, sehingga membuat IP *address pod* mudah berubah.
2. *Service* : Abstraksi atas kumpulan *pod* yang sejenis, untuk menjadi pintu masuk bagi *request* yang ditujukan untuk *pod-pod* tersebut. *Service* dibutuhkan karena *lifetime pod* yang pendek dan mudah berubah IP *address*. *Service* mengawasi perubahan *state* dari *pod-pod* yang dibawahinya.
3. *Deployment* : Abstraksi untuk menyatakan *desired state* dari *pod* dan ditulis dalam *file yaml*. Ketika sebuah *deployment* diaplikasikan pada *cluster*, *deployment* bertugas memastikan kondisi *cluster* telah memenuhi *desired state* yang ditentukan.
4. *Node* : Unit komputasi terkecil yang terdaftar sebagai bagian dari *cluster* Kubernetes sebagai tempat dijalankannya *pod* dan *service*. Terdapat 2 jenis *node* di Kubernetes, yaitu *master node* yang bertugas mengatur kondisi *cluster* dan *worker node* yang berperan sebagai tempat menjalankan *pod* dan *service*.
5. *Volume* : *Persistent storage* dalam *cluster* Kubernetes yang dapat di-*mount* ke suatu *pod* dalam *cluster* tersebut. Data yang disimpan dalam *volume* tidak ikut hilang ketika *pod* yang terasosiasi dengan *volume* tersebut di-*destroy.*
6. *Controller : Non-terminating loop* yang ditanamkan ke *core control loop* Kubernetes dan berfungsi untuk mengawasi dan memodifikasi kondisi *cluster* melalui *kube-apiserver* agar sesuai dengan *desired state*. Terdapat 4 jenis *controller* pada Kubernetes, yaitu *replication controller*, *endpoints controller*, *namespace controller*, dan *serviceaccounts controller*.

2.1.2 Arsitektur

Kubernetes menganut arsitektur *client-server*. Sebuah Kubernetes *cluster* terdiri dari minimal satu *master node* dan beberapa *worker node*. Kubernetes mendukung pengaturan *multi-master* untuk memenuhi aspek *high availability*. *Master node* berfungsi untuk mengelola *cluster*, mengekspos API sebagai *frontend* dalam manajemen *cluster*, dan melakukan *scheduling pod* ke dalam *node*. Berkaitan dengan fungsi tersebut, *master node* memiliki beberapa komponen, diantaranya adalah *kube-apiserver* untuk manajemen *cluster*, *etcd storage* untuk menyimpan data *cluster state*, *kube-controller-manager* untuk mengelola berbagai jenis *controller* ke dalam *core control loop* Kubernetes dan *kube-scheduler* untuk *scheduling pod* ke *node*. Sementara itu, *worker node* berperan untuk menjalankan *pod* dan *service* sesuai keputusan *scheduling master node*. Berkaitan dengan peran tersebut, *worker node* memiliki beberapa komponen, diantaranya adalah *kubelet* sebagai *agent master node* dan *kube-proxy*. Setiap *node* dalam Kubernetes *cluster* memerlukan sebuah *container runtime*, untuk dapat menjalankan dan mengelola *lifecycle* dari *container* dalam sebuah *pod*.



Gambar 1. Diagram Arsitektur Kubernetes [9]

2.2 Arsitektur Microservices

Microservices adalah gaya arsitektur yang menyusun aplikasi sebagai kumpulan *service* kecil yang saling terhubung. Setiap *service* dalam arsitektur *microservices* dapat dijalankan secara mandiri dan menjalankan fungsi tunggal [7]. Kumpulan *service* dengan fungsi berbeda dapat saling berkomunikasi secara *stateless* untuk menjalankan rangkaian fitur aplikasi. Dalam mengimplementasikan arsitektur *microservices*, umumnya digunakan *container* untuk menjalankan tiap *service*.

2.3 Multi-Objective Optimization

Multi-objective optimization adalah salah satu bidang bahasan *multiple criteria decision making* yang berfokus pada optimasi problem secara matematis yang melibatkan lebih dari satu *objective function* untuk dioptimasi secara bersamaan. Dalam optimasi ini, objektif yang dioptimasi saling bertentangan sehingga keputusan optimal yang diambil selalu disertai dengan *trade-off*. Selain dari nilai *objective function*, set solusi pada problem optimasi multi-objektif dapat dibatasi dengan sekumpulan *constraint function*.

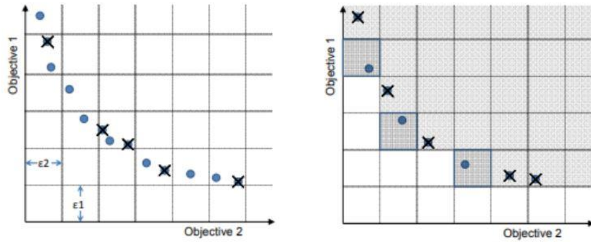
Dalam memahami teori optimasi multi-objektif, diperlukan pemahaman konsep-konsep berikut :

1. *Dominance* : Ukuran kualitas sebuah solusi dalam problem multi-objektif. Sebuah solusi S_1 dikatakan mendominasi solusi S_2 jika tidak ada nilai objektif S_1 yang lebih buruk daripada nilai objektif S_2 dan S_1 memiliki setidaknya satu nilai objektif yang lebih baik daripada S_2. Dengan demikian, solusi S1 memiliki kualitas yang lebih baik daripada solusi S_2.
2. *Feasible decision set* : Kumpulan solusi dengan nilai variabel yang memenuhi keseluruhan *constraint* problem. Satu solusi dalam *feasible decision set* di-*mapping* ke satu solusi dalam *feasible objective space* dengan menerapkan *objective function* ke nilai variabel solusi tersebut untuk mendapatkan solusi yang memiliki kombinasi nilai objektif tertentu.
3. *Pareto-optimal set* : Kumpulan dari semua solusi pada *feasible decision set* yang, ketika di-*mapping* ke *objective space*, tidak terdominasi solusi lain dalam set tersebut. Garis batas yang terbentuk dari kumpulan titik hasil *mapping Pareto-optimal set* ke *objective space* disebut *Pareto-optimal front*.

2.4 Algoritma Ɛ-NSGA-II

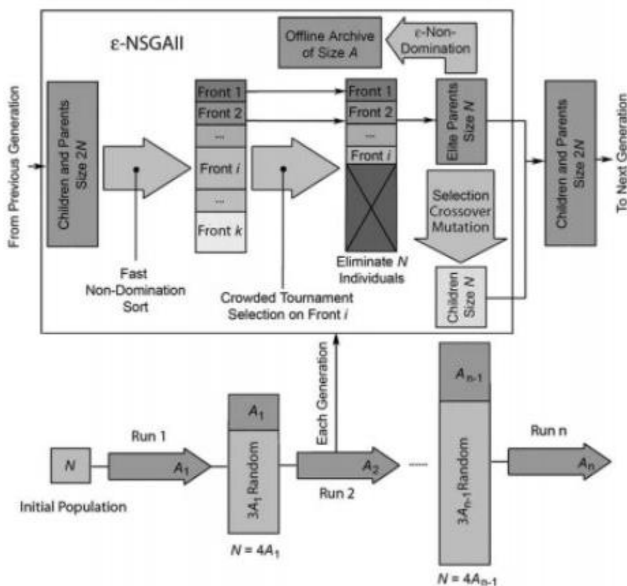
Algoritma Ɛ-NSGA-II dibuat berdasarkan algoritma NSGA-II ciptaan Deb, Pratap, Agarwal dan Meyarivan, sebuah algoritma *evolutionary multi-objective optimization* (EMOO) yang menggunakan pendekatan *sorting* secara *non-dominated* untuk

mengklasifikasikan solusi berdasarkan tingkatan *non-domination* dan operator *crowding distance* untuk mempertahankan *diversity* dari solusi. Ɛ-NSGA-II menambahkan konsep Ɛ-*dominance*, *adaptive population sizing*, dan *self termination* [4].



Gambar 2. Konsep Ɛ-dominance [1]

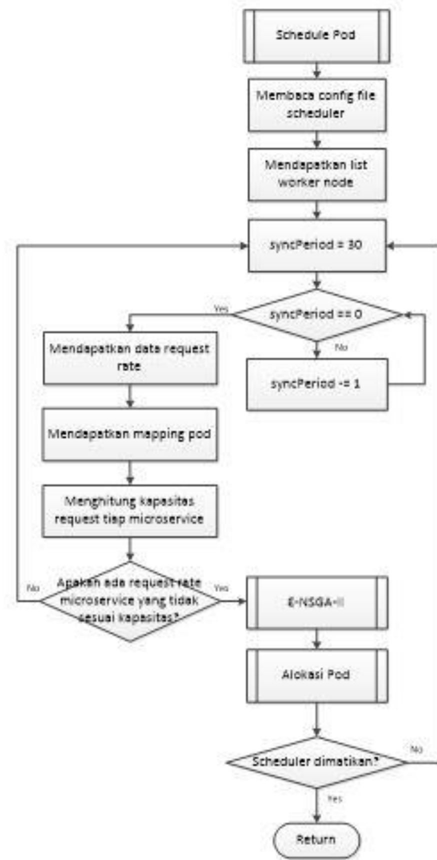
Ɛ-*dominance* merupakan implementasi konsep *dominance* yang dilakukan pada tingkat presisi tertentu dalam pencarian *Pareto-optimal solutions*. Tingkat presisi itu dinyatakan dengan membubuhkan *grid* berukuran Ɛ ke *problem search space*. Kemudian dilakukan pemilihan satu solusi terbaik dari tiap *box* untuk dilanjutkan dengan menerapkan *non-domination search* pada solusi-solusi tersebut dalam skala *grid sehingga* menghasilkan *non-dominated solution.* Kumpulan solusi tersebut kemudian dibandingkan dengan isi Ɛ-*archive.* Solusi yang mampu mendominasi salah satu isi *archive* disimpan untuk menggantikan isi *archive* yang terdominasi olehnya. *Adaptive population sizing* dan *self termination* adalah konsep yang digunakan untuk menghilangkan kebutuhan kalibrasi parameter *genetic algorithm* seperti jumlah generasi dan jumlah populasi. Kebutuhan kalibrasi parameter dapat dihapuskan dengan membuat Ɛ-NSGA-II menjadi sebuah rangkaian *connected runs* dimana algoritma dimulai dengan jumlah populasi yang kecil, melakukan *search* terhadap populasi tersebut hingga tidak ada peningkatan yang cukup berarti pada Ɛ-*archive*, untuk kemudian berpindah ke *run* selanjutnya dengan meningkatkan jumlah populasi dan menginjeksi isi Ɛ-*archive* ke populasi baru tersebut.



Gambar 3. Diagram algoritma Ɛ-NSGA-II [5]

3. DESAIN SISTEM

3.1 Proses Schedule Pod



Gambar 4. Flowchart proses *scheduling*

Proses *schedule pod* dilakukan melalui 2 tahapan utama, yaitu melakukan optimasi pemetaan alokasi *pod* ke *node* menggunakan algoritma Ɛ-NSGA-II dan melakukan alokasi *pod* sesuai pemetaan yang dihasilkan. Proses dimulai dengan *scheduler* membaca *config file* dan mencatat *list worker node* untuk mendapatkan informasi *metadata* aplikasi dan ketersediaan *resource* pada tiap *node*, yang diperlukan dalam perhitungan *fitness function*. Lalu *scheduler* menunggu periode sinkronisasi selama 30 detik sebelum melakukan optimasi *mapping*. Periode sinkronisasi adalah waktu tunggu *scheduler* sebelum mengulangi proses *scheduling*. Langkah berikutnya adalah mendapatkan data *request rate* tiap *microservice* dari *monitoring tool* dan mendapatkan *mapping pod* yang sekarang telah terimplementasi pada *cluster* untuk digunakan dalam perhitungan *request threshold* dari tiap *microservice* berdasarkan jumlah *pod* yang menjalankan *microservice* tersebut. Kedua jenis data ini kemudian dibandingkan untuk menentukan apakah ada *microservice* yang kekurangan ataupun kelebihan jumlah *pod*. Jika ada microservice yang kapasitasnya tidak sesuai dengan *request* yang diterimanya, maka *scheduler* akan memulai proses optimasi *mapping*. Dari eksekusi algoritma Ɛ-NSGA-II dihasilkan sebuah *mapping pod* yang digunakan *scheduler* sebagai acuan untuk melakukan alokasi *pod*. Setelah alokasi *pod* dilakukan, *scheduler* menunggu periode sinkronisasi selanjutnya untuk melakukan optimasi ulang jika diperlukan. Proses ini dilakukan berulang-ulang selama *scheduler* tidak dimatikan oleh *user*.

3.2 Perencanaan Metode Ɛ-NSGA-II

3.2.1 Permodelan Sistem

Tabel 1. Tabel Permodelan Sistem

Class	Atribut	Deskripsi
Microservice Chain	mc_i	*microservice chain* dengan id *i*
	$mschain_i$	*list microservice* yang perlu dipanggil dalam 1 *request* mc_i
	req_i	jumlah *request* per detik yang diterima mc_i
Microservice	ms_j	*microservice* dengan id *j*
	$scale_j$	jumlah *pod* yang menjalankan ms_j
	req_j	jumlah *request* per detik yang diterima ms_j
	$reqthr_j$	target jumlah *request* per detik yang perlu ditangani 1 *pod* yang menjalankan ms_j
	$resthr_j$	*compute resource* yang dikonsumsi 1 *pod* ms_j untuk memenuhi target jumlah *request*
Pod	pod_k	*pod* dengan id *k*
	msp_k	*microservice* pada pod_k
Node	$node_l$	*worker node* dengan id *l*
	cap_l	kapasitas *compute resource* $node_l$
	$mslist_l$	*list microservice* yang berjalan di $node_l$
	$podlist_l$	*list pod* yang berjalan di $node_l$

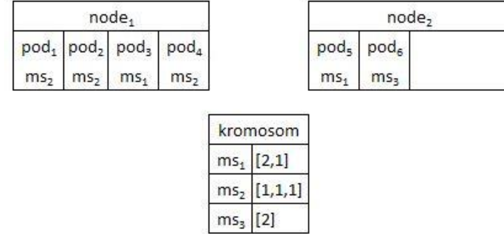
Permodelan sistem yang digunakan dibuat berdasarkan permodelan yang digunakan Guerrero, Lera dan Juiz dalam penelitiannya untuk mengimplementasikan algoritma NSGA-II [3]. Pada permodelan yang dibuat, aplikasi berarsitektur *microservices* yang dijalankan pada *cluster* direpresentasikan sebagai kumpulan *microservice chain*. Setiap *microservice chain* (mc_i) melambangkan sebuah fitur dari aplikasi yang dapat di-*request* oleh *user*. Untuk memenuhi sebuah *request* ke *microservice chain*, dibutuhkan pemanggilan satu atau lebih *microservice*. Daftar pemanggilan *microservice* untuk sebuah *microservice chain* mc_i dilambangkan dengan $mschain_i$, sementara jumlah *user request* per detik atas *microservice chain* tersebut dilambangkan dengan atribut req_i.

Dalam implementasinya, setiap *microservice* (ms_j) diwakilkan oleh satu atau lebih *pod* (pod_k) dimana atribut $scale_j$ menunjukkan jumlah *pod* yang mewakilkan *microservice* ms_j dalam sebuah *cluster*. Total *request* terhadap sebuah *microservice* ms_j didapatkan dari penjumlahan *user request* dari semua *microservice chain* yang melakukan pemanggilan terhadap *microservice* ms_j. Sementara itu, atribut $reqthr_j$ dan $resthr_j$ melambangkan target jumlah *request* per detik untuk ditangani oleh sebuah *pod* yang mengimplementasikan *microservice* ms_j dan *compute resource* yang dikonsumsi untuk memenuhi *request* sebanyak target tersebut.

Masing-masing *pod* dari suatu *microservice* dieksekusi pada suatu *node* dalam *cluster*. Setiap *node* ($node_l$) memiliki kapasitas *resource* yang dilambangkan dengan cap_l. Atribut $mslist_l$ menunjukkan daftar *microservice* yang sedang berjalan pada $node_l$. $Node_l$ menjadi tidak aktif ketika atribut $mslist_l$ *node* tersebut kosong.

3.2.2 Representasi Kromosom

Dalam mengimplementasikan algoritma Ɛ-NSGA-II, sebuah kandidat solusi untuk problem *scheduling container* aplikasi berarsitektur *microservices* direpresentasikan sebagai *array* objek *microservice*, dimana tiap objek *microservice* memiliki *list* id *node* untuk menunjukkan pada *node* mana *pod* yang menjalankan *microservice* tersebut dialokasikan [3]. Dimungkinkannya alokasi beberapa *pod* dari *microservice* yang sama dalam sebuah *node* membuat *list* id *node* pada objek *microservice* dapat memiliki elemen yang sama.



Gambar 5. Contoh representasi kromosom dari sebuah solusi

3.2.3 Rumusan Fitness Function

Fitness function pertama berhubungan dengan objektif efisiensi *cluster* dan dirumuskan sebagai berikut :

$$Active\ Node\ Amount = \sum_{\forall node_l} [mslist_l \neq \emptyset] \quad (1)$$

Dalam rumusan di atas, optimasi efisiensi sebuah *cluster* diukur melalui jumlah *node* yang aktif. Sebuah *node* dikatakan aktif ketika menjalankan satu atau lebih *microservice*. Dengan demikian, *fitness value* sebuah solusi menjadi semakin bagus ketika mampu meminimalkan penggunaan *node*.

Fitness function kedua berhubungan dengan objektif *balanced throughput* aplikasi dan dirumuskan sebagai berikut :

$$Throughput\ Sufficiency = \sum_{\forall ms_j} \left| \frac{req_j}{scale_j} - reqthr_j \right| \quad (2)$$

dimana :

$$req_j = \sum_{\forall mc_i} \left(req_i [ms_j \in mschain_i] \right) \quad (3)$$

Dalam rumusan di atas, *request* terhadap suatu *microservice* merupakan total *user request* per detik dari setiap *microservice chain* yang mengimplementasikannya. Dengan nilai *threshold* yang selalu tetap, maka optimasi *throughput* agar dapat memenuhi *demand* aplikasi dilakukan dengan menyesuaikan jumlah *pod* masing-masing *microservice* dengan jumlah *request* yang ditujukan terhadap *microservice* tersebut. *Fitness value* suatu solusi menjadi semakin bagus ketika jumlah *request* per detik yang diterima setiap *pod* semakin mendekati nilai *request threshold*. Solusi dengan jumlah *pod* kurang memiliki *fitness value* yang buruk karena *pod* harus melayani *request* lebih dari target *request*, sementara jumlah *pod* yang berlebih juga memiliki *fitness value* buruk karena kemampuan pemrosesan masing-masing *pod* tidak dimanfaatkan secara maksimal.

Selain kedua *fitness function* di atas, problem *scheduling container* juga memiliki *constraint* berkaitan dengan kapasitas komputasi *node* dan dirumuskan sebagai berikut :

$$\sum_{\forall pod_k \in podlist_l} res_k < cap_l \quad \forall node_l \quad (4)$$

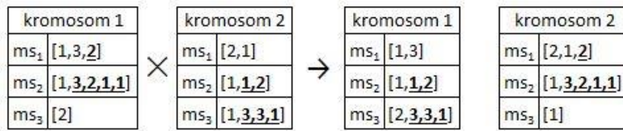
dimana :

$$res_k = \sum_{\forall ms_j \equiv msp_k} resthr_j \qquad (5)$$

Setiap solusi yang dihasilkan perlu memenuhi syarat bahwa total pemakaian *resource* pada suatu *node* tidak boleh melebihi kapasitas *resource* yang dimiliki *node* tersebut. Jika *constraint* tersebut tidak dapat dipenuhi, maka kedua *fitness value* solusi tersebut ditetapkan menjadi *infinity*.

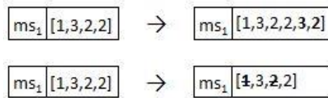
3.2.4 Operator Mutasi dan Crossover

Jenis *crossover* yang digunakan adalah *single-point crossover*, dimana setiap segmen *microservice* pada kromosom disilangkan dengan segmen *microservice* yang sama pada kromosom lainnya. Penyilangan dilakukan dengan memilih angka secara acak sebagai titik potong *crossover* untuk tiap segmen *microservice* dalam pasangan kromosom. Angka di-*random* dengan *range* antara 0 hingga satu kurangnya dari panjang segmen yang memiliki elemen lebih sedikit. Setelah angka dipilih, dibentuk sepasang solusi *offspring* dengan menukar bagian dibelakang titik potong setiap segmen *microsevice* dalam suatu kromosom, sesuai dengan angka acak masing-masing segmen, terhadap kromosom pasangannya [3].



Gambar 6. Contoh operasi *crossover*

Mutasi diperlukan untuk menghindarkan solusi dari keadaan *local minima* dengan mengubah isi kromosom secara acak. Operator mutasi yang digunakan adalah *growth mutation* dan *shrink mutation*. *Growth mutation* dilakukan dengan menambahkan alokasi *pod* untuk suatu *microservice* di *node* yang dipilih secara acak. *Shrink mutation* dilakukan dengan mengurangi alokasi *pod* suatu *microservice* secara acak [3].



Gambar 7. Contoh operasi *growth* dan *shrink mutation*

3.2.5 Metrik Pemilihan Solusi

Ketika iterasi yang diperlukan algoritma Ɛ-NSGA-II untuk menghasilkan kumpulan solusi yang memenuhi kriteria telah tercapai, diperlukan sebuah metrik sebagai tolak ukur pemilihan solusi terbaik dari populasi yang ada, untuk diaplikasikan *scheduler* ke dalam *cluster*. Dalam permasalahan ini, solusi terbaik dipilih berdasarkan *aggregated value* dari nilai normalisasi kedua *fitness value* masing-masing solusi. Metrik *aggregated value* dirumuskan sebagai berikut :

$$Aggregated\,Value = 0.25 * fit_{ANA} + 0.75 * fit_{TE} \qquad (6)$$

dimana fit_{ANA} dan fit_{TE} menunjukkan *fitness value active node amount* dan *throughput exactness*. Bobot tiap objektif menjadi parameter yang menunjukkan prioritas. Bobot yang semakin besar menunjukkan tingkat prioritas yang lebih tinggi. Dari rumusan di atas, bobot *fitness function active node amount* di-*set* lebih kecil karena memiliki *search space* yang yang lebih sempit.

4. IMPLEMENTASI SISTEM

Master node dijalankan dalam bentuk *virtual machine* dengan spesifikasi:

- Prosesor : 4vCPUs
- RAM : 8 GB
- Sistem operasi : Ubuntu Desktop 18.04 64-bit

Worker node dijalankan dalam bentuk *virtual machine* dengan spesifikasi:

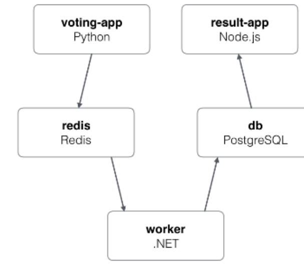
- Prosesor : 2vCPUs
- RAM : 2 GB
- Sistem operasi : Ubuntu Server 18.04 64-bit

Implementasi pengkodean *scheduler*, menggunakan Bahasa pemrograman Python dengan versi 3.6.3. Sementara versi Kubernetes dimana *scheduler* dijalankan adalah versi 1.11.3 dengan versi Docker 18.0.3

5. ANALISA DAN PENGUJIAN

5.1 Aplikasi *Microservices* yang Diuji

Pengujian dilakukan dengan melakukan *scheduling* terhadap aplikasi *microservices voting* [2]. Aplikasi tersebut terdiri dari 5 *microservice,* dengan desain arsitektur dan pola komunikasi tiap *microservice* yang dapat dilihat pada Gambar 8.



Gambar 8. Arsitektur aplikasi *microservices* voting

5.2 *Scheduler* Pembanding

Kube-scheduler melakukan pemilihan *node* melalui tahapan *filtering*, *priority calculation*, dan *selection*, berdasarkan kebutuhan *resource pod* dan ketersediaan *resource* tersebut pada masing-masing *node*. *Kube-scheduler* melakukan *scheduling* dengan fokus pada kesetaraan *load* pada tiap *node*. *Node* yang memenuhi kriteria *pod* dan memiliki lebih banyak *resource* yang tidak terpakai lebih diprioritaskan untuk dipilih dalam proses *scheduling* [8].

Sementara itu, untuk melakukan proses *scaling pod* pada Kubernetes, diperlukan *Horizontal Pod Autoscaler* untuk menentukan jumlah replika *pod* yang harus ditambah ataupun dikurangi, untuk kemudian dapat ditentukan penempatannya oleh *kube-scheduler*. Jumlah replika *pod* ditentukan berdasarkan perbandingan antara nilai rata-rata *metric* pada *cluster* saat itu dan target nilai rata-rata *metric* yang ditentukan oleh *user*. Rumusan perhitungan jumlah replika tersebut adalah sebagai berikut:

$$desiredReplicas = \left\lceil currentReplicas * \frac{currentMetricValue}{desiredMetricValue} \right\rceil \qquad (7)$$

5.3 Pengujian terhadap Nilai Objektif

Pengujian terhadap nilai objektif dilakukan untuk mengetahui keunggulan hasil *scheduling custom scheduler* yang dibuat dalam memenuhi objektif yang ditentukan ketika dibandingkan dengan kombinasi *kube-scheduler* dan *Horizontal Pod Autoscaler*. Pengujian dilaksanakan dengan melakukan *load testing* pada *endpoint* aplikasi *voting*. *Endpoint* yang digunakan dalam pengujian adalah *endpoint* untuk menampilkan halaman *vote* dan *endpoint* untuk menampilkan hasil *voting*.

Kesesuaian terhadap objektif efisiensi *cluster* diukur dengan menghitung jumlah *node* digunakan untuk menjalankan *pod* aplikasi, sedangkan kesesuaian terhadap objektif *throughput* aplikasi diukur dengan mempertimbangkan perbandingan realisasi *throughput* terhadap target *throughput* untuk masing-masing *endpoint* dan perbandingan realisasi rata-rata pemakaian CPU *resource* terhadap target *resource usage* masing-masing *microservice*.

5.3.1 Skenario Pengujian

Pengujian dilakukan dengan melakukan *load testing* terhadap kedua *endpoint* secara bersamaan, dengan 5 skenario target *throughput* yang dijalankan secara sekuensial, masing-masing selama 3 menit. Kemudian dilakukan evaluasi pada tiap variabel pengukur keberhasilan tiap 1 menit. Target *throughput* untuk tiap skenario pengujian yang dilakukan dapat dilihat pada Tabel 2.

Tabel 2. Tabel Skenario Pengujian Pada *Endpoint*

Skenario	***Endpoint***	**Target *Throughput* (*req/s*)**
1	*GetVote*	1950
	GetResult	500
2	*GetVote*	1600
	GetResult	1100
3	*GetVote*	1250
	GetResult	2200
4	*GetVote*	600
	GetResult	2800
5	*GetVote*	300
	GetResult	3300

Pada pengujian yang dilakukan, ditentukan *threshold throughput* sebesar 130 *request* per detik per *pod* untuk *microservice* Vote dan 180 *request* per detik per *pod* untuk *microservice* Result. Sedangkan nilai *threshold* untuk CPU *usage* adalah sebesar 300 *milicore* per *pod* untuk *microservice* Vote dan 200 *milicore* per *pod* untuk *microservice* Result.

5.3.2 Pengujian Kube-Scheduler dan Horizontal Pod Autoscaler

Tabel 3. Tabel Pengujian Keberhasilan *Kube-Scheduler* dan *Horizontal Pod Autoscaler*

Kriteria	**Nilai**
Pod Throughput untuk *Microservice* Vote	158.7333 *req/s*
Pod CPU *Load* untuk *Microservice* Vote	310.5333 milicores

Tabel 3. Tabel Pengujian Keberhasilan *Kube-Scheduler* dan *Horizontal Pod Autoscaler* (Lanjutan)

Standar Deviasi *Pod Throughput* untuk *Microservice* Vote	76.2
Pod Throughput untuk *Microservice* Result	315.4 *req/s*
Pod CPU *Load* untuk *Microservice* Result	260.8667 milicores
Standar Deviasi *Pod Throughput* untuk *Microservice* Result	69.96
Jumlah *Node* Aktif	5

Dari hasil pengujian pada Tabel 3, didapati bahwa rata-rata *throughput* untuk *pod microservice* Vote mendekati nilai *threshold*, namun dengan nilai standar deviasi sebesar 76.2. Standar deviasi yang tinggi menunjukkan adanya ketimpangan *throughput microservice* selama proses pengujian karena rendahnya responsivitas *scheduler* dan *scaler* Kubernetes dalam menangani perubahan *load* pada *pod*. Sedangkan untuk *pod microservice* Result, didapatkan rata-rata *throughput* yang melebihi *threshold* dengan perbedaan yang signifikan, yaitu 315.4 *request* per detik. Selain itu, didapatkan pula nilai standar deviasi sebesar 69.96.

Hasil pengujian kesesuaian objektif efisiensi *cluster* menunjukkan karakteristik *kube-scheduler* dalam melakukan *scheduling* lebih berfokus pada aspek *load-balancing* sehingga jumlah *node* yang aktif menjalankan *pod* aplikasi pada *cluster* menjadi maksimal.

5.3.3 Pengujian Custom Scheduler dengan Algoritma Ɛ-NSGA-II

Tabel 4. Tabel Pengujian Keberhasilan Custom Scheduler dengan Algoritma Ɛ-NSGA-II

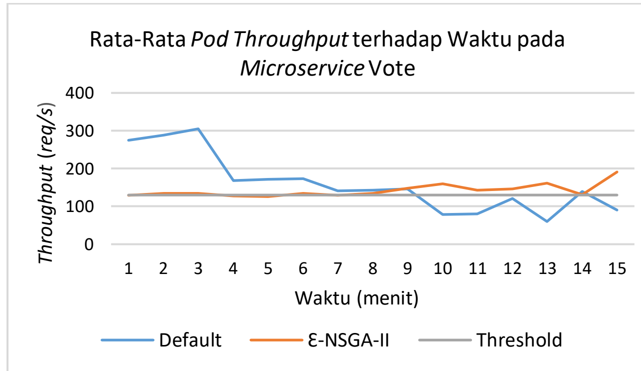
Kriteria	**Nilai**
Pod Throughput untuk *Microservice* Vote	142.0667 *req/s*
Standar Deviasi *Pod Throughput* untuk *Microservice* Vote	17.38
Pod CPU *Load* untuk *Microservice* Vote	284.9333 milicores
Pod Throughput untuk *Microservice* Result	213.5333 *req/s*
Standar Deviasi *Pod Throughput* untuk *Microservice* Result	20.31
Pod CPU *Load* untuk *Microservice* Result	190.4667 milicores
Jumlah *Node* Aktif	3.066667

Dari hasil pengujian pada Tabel 4, didapati bahwa rata-rata *throughput* pada kedua *microservice* mendekati nilai *threshold,* yaitu 142.0667 dari *threshold* 130 untuk *microservice* Vote dan 213.5333 dari *threshold* 180 untuk microservice Result, disertai dengan nilai standar deviasi yang rendah. Nilai standar deviasi yang rendah menunjukkan bahwa *custom scheduler* juga responsif dalam menangani perubahan *load.*

Hasil pengujian kesesuaian objektif efisiensi *cluster* menunjukkan rata-rata jumlah *node* yang aktif selama pengujian adalah sebesar 3.066667. Dengan nilai tersebut, berarti ada penghematan jumlah *node* aktif sebesar 38.66667% dibandingkan hasil dari *kube-scheduler* dan *Horizontal Pod Autoscaler*.

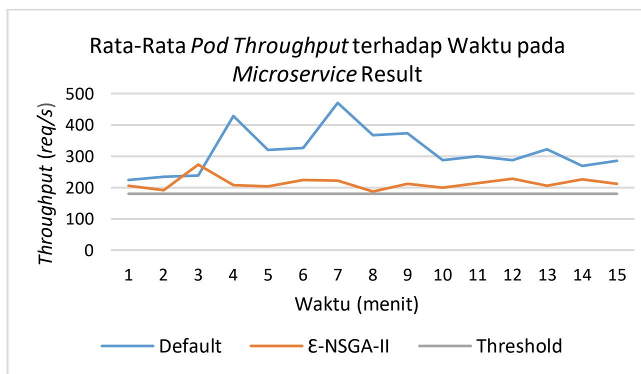
5.3.4 *Perbandingan Hasil Pengujian*

Dari ketiga hasil pengujian, dilakukan perbandingan nilai variabel pengukur keberhasilan tiap pengujian tersebut terhadap *threshold* yang ditentukan. Pada Gambar 9, divisualisasikan perbandingan variabel pengukuran *pod throughput* pada *microservice* Vote. Dari visualisasi tersebut, dapat dilihat bahwa *custom scheduler* menyesuaikan jumlah replika dari jumlah *request* yang diterima sehingga menghasilkan *throughput* yang mendekati *threshold*. Sementara itu, kombinasi *kube-scheduler* dan *Horizontal Pod Autoscaler* tidak dapat menyesuaikan jumlah replika secara tepat waktu sehingga menghasilkan *throughput* yang jauh diatas *threshold* pada menit awal ketika *load microservice* Vote sedang tinggi dan dibawah *threshold* pada menit akhir ketika *load microservice* Vote menjadi rendah.



Gambar 9. Grafik perbandingan rata-rata *pod throughput* pada *microservice* Vote

Pada Gambar 10, divisualisasikan perbandingan variabel pengukuran *pod throughput* pada *microservice* Result. Dari visualisasi tersebut, dapat dilihat bahwa *custom scheduler* menghasilkan *throughput* yang mendekati *threshold*. Sementara itu, kombinasi *kube-scheduler* dan *Horizontal Pod Autoscaler* gagal menambah jumlah replika yang cukup untuk menangani kenaikan *load microservice* Result secara signifikan sehingga menghasilkan *throughput* yang jauh diatas *threshold* pada menit-menit tengah.



Gambar 10. Grafik perbandingan rata-rata *pod throughput* pada *microservice* Result

6. KESIMPULAN

Dari hasil perancangan dan pembuatan optimasi multi-objektif untuk *container scheduling* menggunakan algoritma Ɛ-NSGA-II, dapat diambil kesimpulan antara lain:

- Permodelan sistem yang dibuat dapat merepresentasikan permasalahan *container scheduling* dan dapat dipakai dalam algoritma Ɛ-NSGA-II
- Program dapat menghasilkan keputusan *scheduling* dengan rata-rata penghematan jumlah *node* aktif sebesar 38.66667% untuk objektif efisiensi *cluster*
- Program dapat menghasilkan keputusan *scheduling* dengan rata-rata *throughput pod* sebesar 142.0667 dari nilai threshold 130 untuk *pod microservice* Vote dan 213.5333 dari nilai threshold 180 untuk *pod microservice* Result pada objektif *balanced throughput*

Saran yang diberikan untuk penyempurnaan dan pengembangan lebih lanjut untuk program ini adalah sebagai berikut :

- Dapat melakukan *scheduling* untuk aplikasi yang *scalable* dan tidak *scalable* secara bersamaan
- Dapat melakukan *scheduling* pada aplikasi dengan kebutuhan *hardware* khusus
- Dapat menangani penambahan atau pengurangan *node* saat *runtime*
- Memiliki sistem kontrol yang dapat mematikan atau menyalakan *node* ketika dibutuhkan

7. DAFTAR PUSTAKA

[1] Brands, T., Wismans, L., & van Berkum, E. 2014. Multi-objective transportation network design: Accelerating search by applying Ɛ-NSGAII. *2014 IEEE Congress On Evolutionary Computation (CEC)*, 405-412. doi: 10.1109/cec.2014.6900486

[2] Example Voting App. 2018. GitHub repository: https://github.com/dockersamples/example-voting-app

[3] Guerrero, C., Lera, I., & Juiz, C. 2017. Genetic Algorithm for Multi-Objective Optimization of Container Allocation in Cloud Architecture. *Journal Of Grid Computing*, *16*(1), 113-135. doi: 10.1007/s10723-017-9419-x

[4] Kollat, J., & Reed, P. 2005. The value of online adaptive search: a performance comparison of NSGA-II, Ɛ-NSGAII, and ƐMOEA. *The Third International Conference On Evolutionary Multi-Criterion Optimization (EMO 2005), Lecture Notes in Computer Science, vol. 3410*, p. 386–98.

[5] Kollat, J., & Reed, P. 2006. Comparing state-of-the-art evolutionary multi-objective algorithms for long-term groundwater monitoring design. *Advances In Water Resources*, *29*(6), 792-807. doi: 10.1016/j.advwatres.2005.07.010

[6] kube-scheduler. 2018. Retrieved from https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/

[7] Kumar, A. 2018. Top 5+ Microservices Architecture and Design Best Practices - DZone Microservices. Retrieved from https://dzone.com/articles/top-5-microservices-architecture-and-design-best-p

[8] Topliceanu, A. 2018. Scheduling in Kubernetes. Retrieved from http://alexandrutopliceanu.ro/post/scheduling-in-kubernetes/

[9] What is Kubernetes - Learn Kubernetes from Basics. 2018. Retrieved from https://www.learnitguide.net/2018/08/what-is-kubernetes-learn-kubernetes.html