TUGAS AKHIR - IF184802

PENGEMBANGAN KEMAMPUAN SKALA OTOMATIS UNTUK INFRASTRUKTUR *CLOUD-NATIVE* BERBASIS KUBERNETES MENGGUNAKAN *CUSTOM METRIC* DAN *SIDECAR PATTERN*

AGUEL SATRIA WIJAYA
NRP 05111640000056

Dosen Pembimbing I
Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D.

Dosen Pembimbing II
Rizky Januar Akbar, S.Kom., M.Eng.

DEPARTEMEN TEKNIK INFORMATIKA
Fakultas Teknologi Elektro dan Informatika Cerdas
Institut Teknologi Sepuluh Nopember
Surabaya 2020

(Halaman ini sengaja dikosongkan)

TUGAS AKHIR - IF184802

PENGEMBANGAN KEMAMPUAN SKALA OTOMATIS UNTUK INFRASTRUKTUR *CLOUD-NATIVE* BERBASIS KUBERNETES MENGGUNAKAN *CUSTOM METRIC* DAN *SIDECAR PATTERN*

AGUEL SATRIA WIJAYA
NRP 05111640000056

Dosen Pembimbing I
Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D.

Dosen Pembimbing II
Rizky Januar Akbar, S.Kom., M.Eng.

DEPARTEMEN TEKNIK INFORMATIKA
Fakultas Teknologi Elektro dan Informatika Cerdas
Institut Teknologi Sepuluh Nopember
Surabaya 2020

(Halaman ini sengaja dikosongkan)

UNDERGRADUATE THESIS - IF184802

DEVELOPING AUTOMATIC SCALING FOR KUBERNETES-BASED CLOUD-NATIVE USING CUSTOM METRIC AND SIDECAR PATTERN

AGUEL SATRIA WIJAYA
NRP 05111640000056

Supervisor I
Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D.

Supervisor II
Rizky Januar Akbar, S.Kom., M.Eng.

DEPARTMENT OF INFORMATICS ENGINEERING
Faculty of Intelligent Electrical and Informatics Technology
Institut Teknologi Sepuluh Nopember
Surabaya 2020

(Halaman ini sengaja dikosongkan)

LEMBAR PENGESAHAN

PENGEMBANGAN KEMAMPUAN SKALA OTOMATIS UNTUK INFRASTRUKTUR *CLOUD-NATIVE* BERBASIS KUBERNETES MENGGUNAKAN *CUSTOM METRIC* DAN *SIDECAR PATTERN*

TUGAS AKHIR

Diajukan Guna Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
pada
Bidang Studi Arsitektur Jaringan dan Komputer
Program Studi S1 Departemen Teknik Informatika
Fakultas Teknologi Elektro dan Informatika Cerdas
Institut Teknologi Sepuluh Nopember

Oleh :
AGUEL SATRIA WIJAYA
NRP: 05111640000056

Disetujui oleh Dosen Pembimbing Tugas Akhir :

Royyana Muslim I., S.Kom., M.Kom., Ph.D. .....
NIP: 197708242006041001 (Pembimbing 1)

Rizky Januar Akbar, S.Kom., M.Eng. .....
NIP: 198701032014041001 (Pembimbing 2)

SURABAYA
Juli 2020

(Halaman ini sengaja dikosongkan)

PENGEMBANGAN KEMAMPUAN SKALA OTOMATIS UNTUK INFRASTRUKTUR *CLOUD-NATIVE* BERBASIS KUBERNETES MENGGUNAKAN *CUSTOM METRIC* DAN *SIDECAR PATTERN*

Nama	:	**AGUEL SATRIA WIJAYA**
NRP	:	**05111640000056**
Departemen	:	**Teknik Informatika FTEIC**
Pembimbing I	:	**Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D.**
Pembimbing II	:	**Rizky Januar Akbar, S.Kom., M.Eng.**

Abstrak

Virtualisasi merupakan sebuah abstraksi dari sumber daya komputer. Dengan adanya virtualisasi pada komputasi awan memungkinkan adanya pemisahan sistem operasi pada suatu perangkat keras. Pada teknologi virtualisasi terdapat aplikasi yang sejak awal sudah didesain untuk dijalankan pada cloud. Aplikasi tersebut bisa disebut dengan Cloud Native.

API gateway merupakan sebuah aplikasi pusat untuk mengarahkan pengguna ke aplikasi yang dituju. Sehingga tingkat akses pada API gateway ini kemungkinan sangat tinggi. Jika hanya ada satu layanan saja maka aplikasi API gateway bisa kewalahan dalam melayani permintaan dari pengguna jadi diperlukan infrastruktur kluster yang adaptif sehingga bisa menyesuaikan jumlah layanan yang ada terhadap beban layanan pada waktu sebenarnya. Ketika jumlah pengguna meningkat, layanan akan bisa melakukan skala otomatis untuk meningkatkan jumlah layanan. Begitu juga dengan sebaliknya ketika jumlah pengguna menurun maka layanan akan juga melakukan skala otomatis dengan menurunkan jumlah layanan yang ada.

Dalam tugas akhir ini akan dibuat sebuah infrastruktur dengan orkestrasi kubernetes agar dapat mekakukan skala otomatis terhadap beban layanan sesuai dengan metric yang telah ditentukan untuk platform MyITS. Selain itu akan diimplementasi juga sidecar pattern pada layanan API gateway yang berfungsi untuk mengambil data dari container utama agar bisa disediakan ulang untuk menjadi custom metric. Custom metric ini akan digunakan sebagai salah satu parameter untuk menentukan apakah suatu pod perlu di skala atau tidak. Pada pengerjaan, akan melibatkan beberapa tools pendukung seperti prometheus, rancher dan kubernetes.

Hasil uji coba menunjukkan bahwa skala otomatis bisa berjalan sesuai dengan algoritma yang telah ditentukan. Selain itu juga skala otomatis bisa mengurangi beban layanan sehingga meningkatkan performa mulai dari total requests, requests yang berhasil dilakukan, dan tingkat requests per second semakin meningkat sejajar dengan penambahan jumlah pod dari hasil skala otomatis. Sidecar pattern bekerja sesuai dengan fungsinya sebagai pengambil data dan penyedia metric untuk digunakan dalam skala otomatis.

***Kata-Kunci**: kubernetes, autoscale, cloud-native, docker, sidecar pattern.*

DEVELOPING AUTOMATIC SCALING FOR KUBERNETES-BASED CLOUD-NATIVE USING CUSTOM METRIC AND SIDECAR PATTERN

Name	:	**AGUEL SATRIA WIJAYA**
NRP	:	**05111640000056**
Department	:	**Informatics Engineering FTEIC**
Supervisor I	:	**Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D.**
Supervisor II	:	**Rizky Januar Akbar, S.Kom., M.Eng.**

Abstract

Virtualization is an abstraction of computer resources. With the virtualization in cloud computing allows the separation of the operating system on a hardware device. In virtualization technology there are applications that have been designed to run on the cloud from the start. The application can be called Cloud Native.

The API gateway is a central application for directing users to the intended application. So the level of access to this API gateway can be high. If there is only one service, the API gateway application can be overwhelmed in serving requests from users, so an adaptive cluster infrastructure is needed so that it can adjust the number of services available based on service load in real time. When the number of users increases, the service will be able to scale up automatically to increase the number of services. Likewise, vice versa when the number of users decreases, the service will also scale down automatically by reducing the number of existing services.

In this final project an infrastructure with kubernetes orchestration will be made in order to be able to scale automatically based on service load in accordance with the

metrics specified for the MyITS platform. Besides that, a sidecar pattern will also be implemented on the API gateway service that functions to retrieve data from the main container so that it can be deployed to be a custom metric. This custom metric will be used as a parameter to determine whether a pod needs to be scaled or not. In the process, it will involve several supporting tools such as prometheus, rancher and Kubernetes.

The trial results show that the automatic scale can run according to a predetermined algorithm. In addition, automatic scaling can reduce the service burden, thereby increasing performance starting from total requests, successful requests made, and the level of requests per second is increasing parallel with the addition of the number of pods from the automatic scale results. Sidecar pattern works according to its function as a data picker and metric provider for use in automatic scaling.
***Keyword**: kubernetes, autoscale, cloud-native, docker, sidecar pattern.*

KATA PENGANTAR

Dengan segala pujian saya aturkan terima kasih kepada Tuhan Yang Maha Esa karena atas segala rahmat-Nya penulis dapat menyelesaikan Tugas Akhir yang berjudul **”Pengembangan Kemampuan Skala Otomatis untuk Infrastruktur *Cloud-native* Berbasis Kubernetes Menggunakan *Custom Metric* dan *Sidecar Pattern*”**. Pengerjaan Tugas Akhir ini merupakan suatu kesempatan yang sangat baik bagi penulis. Dengan pengerjaan Tugas Akhir ini, penulis bisa belajar lebih banyak untuk memperdalam dan meningkatkan apa yang telah didapatkan penulis selama menempuh perkuliahan di Departemen Informatika ITS. Dengan Tugas Akhir ini penulis juga dapat menghasilkan suatu implementasi dari apa yang telah penulis pelajari. Selesainya Tugas Akhir ini tidak lepas dari bantuan dan dukungan beberapa pihak. Sehingga pada kesempatan ini penulis mengucapkan syukur dan terima kasih kepada:

1. Allah Yang Maha Kuasa atas anugerahnya yang tidak terkira kepada penulis.
2. Keluarga penulis yang selalu menyemangati.
3. Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D selaku pembimbing I yang telah membantu, membimbing dan memotivasi penulis mulai dari pengerjaan proposal hingga terselesaikannya Tugas Akhir ini.
4. Bapak Rizky Januar Akbar, S.Kom., M.Eng. selaku pembimbing II yang juga telah membantu, membimbing dan memotivasi penulis mulai dari pengerjaan proposal hingga terselesaikannya Tugas Akhir ini.
5. Dr.Eng.Chastine Fatichah S.Kom, M.Kom selaku Kepala Departemen Informatika ITS pada masa pengerjaan Tugas Akhir dan segenap dosen Departemen Informatika yang telah memberikan ilmu dan pengalamannya.
6. Sahabat penulis Abiyoso Hartono dan Dian Eko Saputro yang selalu menemani dan menyemangati penulis.
7. Teman-teman angkatan 2016 (Upik, Gilbert, Wawan,

Bagus, Fadli, Ganen, Denden, Teja, Jere, Hafid, Vicky, Farras, Dandy, dan lainnya) yang selalu menghibur dan menemani penulis mulai dari awal semester 1.

8. Keluarga besar *Administrator* laboratorium AJK (Raldo, Khawari, Tamtam, Yoga, Mail, Sulton, Fawwaz, Dhana, Lia, Haura, Yohan, Sultan, Restu, Zahra) yang bersedia direpotkan, merepotkan dan menemani penulis dalam mengerjakan Tugas Akhir.
9. Alumni-alumni AJK (Mas Daniel, Mas Toni, Mas Syukron, Mas Fatih, Ambon, Mas Oing, Mbak Bebet, Mbak Vivi, Mas Satria, Mas Awan, Mas Penyok, Mas Didin, Mas Fuad, Mbak Hana, Mbak Daus, Mbak Nahda) yang selalu membuat saya termotivasi untuk lulus dan memberikan arahan dalam pengerjaan Tugas Akhir ini.
10. Teman-teman proyek (Mas Nafiar, Mas Hero, Mas Yoga, Raldo, Ical, Mail, Sulton) yang telah bersama-sama bekerja dengan keras.
11. Teman-teman satu bimbingan (Mail, Hilmi, Azki) yang telah sama-sama berjuang dalam mengerjakan dan mengimplementasikan Tugas Akhir.

Penulis menyadari bahwa Tugas Akhir ini masih memiliki banyak kekurangan. Sehingga dengan kerendahan hati, penulis mengharapkan kritik dan saran dari pembaca untuk perbaikan ke depannya.

Surabaya, Juli 2020

Aguel Satria Wijaya

DAFTAR ISI

ABSTRAK **vii**

ABSTRACT **ix**

KATA PENGANTAR **xi**

DAFTAR ISI **xiii**

DAFTAR TABEL **xvii**

DAFTAR GAMBAR **xix**

DAFTAR KODE SUMBER **xxi**

BAB I PENDAHULUAN **1**

1.1 Latar Belakang 1
1.2 Rumusan Masalah 3
1.3 Batasan Masalah 3
1.4 Tujuan 4
1.5 Manfaat 4
1.6 Metodologi 4
1.6.1 Penyusunan Proposal Tugas Akhir 5
1.6.2 Studi Literatur 5
1.6.3 Analisis dan Desain Perangkat Lunak . . . 5
1.6.4 Implementasi Perangkat Lunak 5
1.6.5 Pengujian dan Evaluasi 6
1.6.6 Penyusunan Buku Tugas Akhir 6
1.7 Sistematika Penulisan 6

BAB II TINJAUAN PUSTAKA **9**

2.1 API Gateway 9
2.2 Autoscaling 9
2.3 Docker 9
2.4 Kong Gateway 11

2.5 Kubernetes 12
2.6 Prometheus 14
2.7 Python 15
2.8 Rancher 15
2.9 Sidecar-Pattern 16

BAB III DESAIN DAN PERANCANGAN **17**
3.1 Deskripsi Umum Sistem 17
3.2 Arsitektur Umum Sistem 18
3.2.1 Desain Umum Sistem 18
3.2.2 Perancangan *Sidecar Pattern* 19
3.2.3 Perancangan Skala Otomatis 19
3.3 Menentukan Algoritma *Scaling* 23

BAB IV IMPLEMENTASI **27**
4.1 Lingkungan Implementasi 27
4.1.1 Perangkat Keras 28
4.1.2 Perangkat Lunak 28
4.2 Mempersiapkan Orkestrasi Kubernetes 29
4.3 Implementasi *Sidecar Pattern* 29
4.4 Implementasi Skala Otomatis 30
4.4.1 Implementasi *Custom Exporter* 30
4.4.2 Implementasi *Application* 35
4.4.3 Implementasi *Kubelet Exporter* 46
4.4.4 Implementasi *Collector* 46
4.4.5 Implementasi *Connector* 50
4.4.6 Implementasi *Scaler* 56

BAB V PENGUJIAN DAN EVALUASI **61**
5.1 Lingkungan Uji Coba 61
5.2 Skenario Uji Coba 63
5.2.1 Skenario Uji Coba Fungsionalitas 63
5.2.2 Skenario Uji Coba Performa 67
5.3 Hasil Uji Coba dan Evaluasi 69

5.3.1 Uji Fungsionalitas 69
5.3.2 Hasil Uji Performa 71

BAB VI PENUTUP **85**
6.1 Kesimpulan 85
6.2 Saran 85

DAFTAR PUSTAKA **87**

BAB A INSTALASI PERANGKAT LUNAK **89**

BAB B KODE SUMBER **93**

BIODATA PENULIS **137**

(Halaman ini sengaja dikosongkan)

DAFTAR TABEL

4.1 Pembagian Peran dan alamat IP 28

5.1 Spesifikasi komponen 61
5.2 Skenario uji fungsionalitas *sidecar pattern* 64
5.3 Skenario uji fungsionalitas skala otomatis 66
5.4 Hasil uji fungsionalitas *sidecar pattern* 70
5.5 Hasil uji fungsionalitas skala otomatis 71
5.6 Hasil uji batas beban *pod* kong pada admin API . 72
5.7 Hasil uji batas beban *pod* kong pada test *service* . 73
5.8 Hasil percobaan pada keadaan maksimal 83

(Halaman ini sengaja dikosongkan)

DAFTAR GAMBAR

2.1 Infrastruktur Docker 10
2.2 Infrastruktur Kubernetes 13
2.3 *Sidecar Pattern* 16

3.1 Perancangan komponen skala otomatis 18
3.2 *Application* dan *Custom Exporter* pada Satu *Pod* . 19

4.1 Kluster Api Gateway 27

5.1 Infrastruktur kluster 62
5.2 Potongan *event* kluster kubernetes 74
5.3 Jumlah *pod* dari berbagai tingkat *concurrency* . . 75
5.4 Total *requests* dari berbagai tingkat *concurrency* . 76
5.5 Jumlah *requests* berhasil dari berbagai tingkat *concurrency* 77
5.6 Nilai *requests* per detik dari berbagai tingkat *concurrency* 78
5.7 Jumlah *pod* dari percobaan terhadap `http://10.199.2.175:8000/mock` 79
5.8 Total *requests* dari percobaan terhadap `http://10.199.2.175:8000/mock` 80
5.9 Jumlah *requests* berhasil dari percobaan terhadap `http://10.199.2.175:8000/mock` 81
5.10 Nilai *requests* per detik dari percobaan terhadap `http://10.199.2.175:8000/mock` 82

(Halaman ini sengaja dikosongkan)

DAFTAR KODE SUMBER

IV.1 Menambah *sidecar container* 30
IV.2 Potongan program *custom metric active connection* 31
IV.3 Potongan program *custom metric writing connection* 32
IV.4 Potongan program *custom metric request time* . . 33
IV.5 Dockerfile *sidecar* 34
IV.6 Membangun *image sidecar* 35
IV.7 Pengaturan kong defaults 36
IV.8 Dockerfile kong 36
IV.9 Membangun image Kong Gateway 37
IV.10 Potongan postgres-persistent.yaml 38
IV.11 Pemasangan PV dan PVC 38
IV.12 Berkas postgres-deployment.yaml 39
IV.13 Pemasangan postgreSQL 40
IV.14 Berkas postgres-service.yaml 41
IV.15 Pemasangan *service* postgres 41
IV.16 Mempersiapkan *database* Kong Gateway 41
IV.17 Pemasangan konga 42
IV.18 Berkas kong-deployment.yaml 42
IV.19 Memasang Kong Gateway 45
IV.20 Mengecek Kong Gateway 45
IV.21 Memasang metric-server 46
IV.22 Memasang kebutuhan Prometheus 47
IV.23 Potongan prometheus-configMap.yaml 47
IV.24 Memasang configMap 48
IV.25 Berkas prometheus-deployment.yaml 49
IV.26 Memasang Prometheus 50
IV.27 Berkas values.yaml 51
IV.28 Memasang Prometheus-adapter 56
IV.29 Berkas kong-customhpa.yaml 57
IV.30 Berkas postgres-customhpa.yaml 58
IV.31 Memasang *scaler* 59
V.1 Menguji *kubelet exporter* 65
V.2 Mengecek *connector* 65
A.1 Instalasi kebutuhan docker 89

A.2 Menambah kunci GPG 89
A.3 Menambah repository docker 89
A.4 Instalasi aplikasi docker 89
A.5 Mengecek status docker 90
A.6 Menjalankan rancher 90
A.7 Memastikan rancher berjalan 91
A.8 Manmbah master node 91
A.9 Menambah worker node 92
B.1 Kode sumber nginx_kong.lua 93
B.2 Kode sumber nginx_kong.lua 94
B.3 Kode sumber main.py 107
B.4 Kode sumber requirements.txt 112
B.5 Kode sumber postgres-persistent.yaml 112
B.6 Kode sumber migration-bootstrap.yaml 113
B.7 Kode sumber migration-up.yaml 114
B.8 Kode sumber konga-prepare.yaml 116
B.9 Kode sumber konga-deployment.yaml 117
B.10 Kode sumber konga-service.yaml 118
B.11 Kode sumber kong-service.yaml 119
B.12 Kode sumber aggregated-metric-read.yaml 120
B.13 Kode sumber auth-delegator.yaml 120
B.14 Kode sumber auth-reader-conf.yaml 121
B.15 Kode sumber metrics-apiservice.yaml 121
B.16 Kode sumber metrics-deployment.yaml 122
B.17 Kode sumber metric-service.yaml 124
B.18 Kode sumber resource-reader.yaml 125
B.19 Kode sumber prometheus-namespace.yaml 126
B.20 Kode sumber prometheus-clusterrole.yaml 126
B.21 Kode sumber prometheus-clusterrolebinding.yaml 127
B.22 Kode sumber prometheus-serviceaccount.yaml . . 127
B.23 Kode sumber prometheus-configMap.yaml 128
B.24 Kode sumber prometheus-service.yaml 134

BAB I

PENDAHULUAN

Pada bab ini akan dipaparkan mengenai garis besar tugas akhir yang meliputi latar belakang, tujuan, rumusan, dan batasan permasalahan, metodologi pembuatan tugas akhir dan sistematika penulisan.

1.1 Latar Belakang

Virtualisasi merupakan sebuah abstraksi dari sumber daya komputer. Dengan adanya virtualisasi pada komputasi awan memungkinkan adanya pemisahan sistem operasi pada suatu perangkat keras. Virtualisasi memiliki kemampuan untuk membuat sebuah sumber daya fisik tunggal untuk seolah-olah menjadi banyak sumber daya virtual[1].

Pada teknologi virtualisasi terdapat aplikasi yang sejak awal sudah didesain untuk dijalankan pada *cloud*. Aplikasi tersebut bisa disebut dengan *Cloud Native*. Aplikasi tersebut sangat bergantung dengan infrastruktur komputasi awan untuk dapat bekerja dengan baik. Salah satu contoh dari *cloud native* adalah orkestrasi kubernetes.

Penggunaan *cloud native* memiliki beberapa kelebihan dalam membangun suatu lingkungan hidup suatu infrastruktur komputasi awan. Aplikasi *cloud native* didesain untuk dapat *scalable*, dapat dipantau secara berkala, dan mudah dikelola. Selain hal itu, *cloud native* juga memiliki laju perkembangan yang pesat dan penemuan aplikasi-aplikasi pendukung yang baru juga semakin banyak.

Institut Teknologi Sepuluh Nopember (ITS) memiliki sebuah platform yang bernama myITS. Platform myITS menggunakan teknologi *cloud native* kubernetes sebagai orkestrasinya. Salah satu teknologi yang dikembangkan ada aplikasi *Single Sign-on* (SSO). Kedepan akan dikembangkan lagi sebuah aplikasi *API gateway* yang menggunakan Kong Gateway sebagai alatnya.

API gateway pada myITS merupakan sebuah aplikasi pusat untuk mengarahkan pengguna ke aplikasi yang dituju. Sehingga tingkat akses pada *API gateway* ini kemungkinan sangat tinggi. Jika hanya ada satu layanan saja maka aplikasi *API gateway* bisa kewalahan dalam melayani permintaan dari pengguna jadi diperlukan infrastruktur kluster yang adaptif sehingga bisa menyesuaikan jumlah layanan yang ada terhadap beban layanan pada waktu sebenarnya. Ketika jumlah pengguna meningkat, layanan akan bisa melakukan skala otomatis untuk meningkatkan jumlah layanan. Begitu juga dengan sebaliknya ketika jumlah pengguna menurun maka layanan akan juga melakukan skala otomatis dengan menurunkan jumlah layanan yang ada.

Demi skala otomatis bisa berjalan dengan baik maka perlu adanya *metric* sebagai penentu kapankah layanan perlu di skala atas ataupun skala bawah. Salah satu *metric* yang diperlukan adalah berbagai *metrics* agar bisa tepat sasaran dalam mengelola sumber daya. Aplikai Kong Gateway memiliki limitasi tersendiri dalam mengolah jumlah requests yang bisa ditangani perdetiknya. Oleh sebab itu akan ditambahkan *metric* koneksi aktif agar bisa dilihat berapa jumlah koneksi yang terhubung saat ini. Perlu juga dibuat *metrics* yang dapat melihat apakah *requests* saat ini lambat atau cepat, sehingga akan dibuat 2 buah *metrics* yaitu *writing connection* yang memiliki fungsi untuk melihat berapa jumlah *requests* yang saat ini sedang ditangani oleh suatu *pod* dan akan ada *requests time* di mana akan diberikan data nilai rata-rata waktu dari setiap *requests* perdetiknya. Namun ketiga *metrics* tidaklah ada secara standar pada kubernetes sehingga perlu dibuat sebuah *custom metric* untuk mengambil data lain dari layanan *API gateway*.

Custom metric pada layanan *API gateway*, diperlukan adanya suatu *container* tambahan yang mendukung *container* utama Kong gateway yang berfungsi untuk mengambil nilai koneksi aktif dan menyediakannya untuk digunakan sebagai salah satu

parameter skala tanpa mengganggu fungsi *container* utama yaitu sebagai *API gateway*. Pola pemasangan *container* tambahan pada *container* utama ini bisa disebut dengan *sidecar pattern*.

Pada tugas akhir ini akan dibuat sebuah infrastruktur *API gateway* yang mampu melakukan skala otomatis terhadap beban yang ada dengan menerapkan *sidecar pattern*. Diharapkan tugas akhir ini dapat memberi solusi atas permasalahan tersebut dengan menggunakan orkestrasi Kubernetes.

1.2 Rumusan Masalah

Rumusan masalah yang diangkat dalam tugas akhir ini dapat dipaparkan sebagai berikut:

1. Bagaimana mengimplementasikan infrastruktur *server* dengan orkestrasi kubernetes agar dapat melakukan skala otomatis terhadap beban layanan *API gateway* untuk platform MyITS?
2. Bagaimana mengimplementasikan *sidecar pattern* pada orkestrasi kubernetes untuk mendukung skala otomatis pada *server*?
3. Bagaimana membuat *custom metric* pada orkestrasi kubernetes untuk mendukung skala otomatis pada *server*?
4. Bagaimana menentukan algoritma *scaling* dalam infrastruktur berbasis orkestrasi kubernetes agar bisa dinamik sesuai dengan *multi-criteria* yang ditentukan?

1.3 Batasan Masalah

Dari permasalahan yang telah diuraikan di atas, terdapat beberapa batasan masalah pada tugas akhir ini, yaitu:

1. Layanan yang digunakan adalah layanan *API gateway* dengan menggunakan Kong Gateway.

2. Infrastruktur *server* menggunakan orkestrasi kubernetes.
3. *Internal metric* yang digunakan CPU dan/atau *Memory*.
4. *Custom metric* yang digunakan adalah *active connection*.
5. Kemampuan skala otomatis dibatasi hanya pada *Horizontal Pod*.
6. *Sidecar pattern* hanya diterapkan pada *deployment* Kong.

1.4 Tujuan

Tujuan pembuatan tugas akhir ini antara lain:

1. Mengimplementasi infrastruktur *server* dengan orkestrasi kubernetes agar dapat melakukan skala otomatis terhadap beban layanan *API gateway* untuk platform MyITS.
2. Mengimplementasi *sidecar pattern* pada orkestrasi kubernetes untuk mendukung skala otomatis pada *server*.
3. Membuat *custom metric* pada orkestrasi kubernetes untuk mendukung skala otomatis pada *server*.
4. Menentukan algoritma *scaling* dalam infrastruktur berbasis orkestrasi kubernetes agar bisa dinamik sesuai dengan *multi-criteria* yang ditentukan.

1.5 Manfaat

Manfaat dari pembuatan tugas akhir ini adalah memberi informasi mengenai cara mengimplementasikan infrastruktur *server* dengan orkestrasi kubernetes agar dapat melakukan skala otomatis terhadap beban layanan *API gateway* untuk platform MyITS yang berbasis Kong Gateway.

1.6 Metodologi

Metodologi yang digunakan dalam pembuatan tugas akhir ini adalah sebagai berikut.

1.6.1 Penyusunan Proposal Tugas Akhir

Proposal tugas akhir ini berisi tentang deskripsi pendahuluan dari tugas akhir yang akan dibuat. Pendahuluan tugas akhir ini terdiri dari hal yang menjadi latar belakang diajukannya usulan tugas akhir, rumusan masalah yang diangkat, batasan masalah pada tugas akhir, tujuan dari pembuatan tugas akhir dan manfaat dari hasil pembuatan tugas akhir. Selain itu dijabarkan pula tinjauan pustaka yang digunakan sebagai referensi pendukung pembuatan tugas akhir. Sub bab metodologi berisi penjelasan mengenai tahapan penyusunan tugas akhir mulai dari penyusunan proposal hingga penyusunan buku tugas akhir. Terdapat pula sub bab jadwal kegiatan yang menjelaskan jadwal pengerjaan tugas akhir.

1.6.2 Studi Literatur

Pada tahap ini dilakukan pencarian informasi dan referensi mengenai Kubernetes dan *Sidecar Pattern* untuk mendukung dan memastikan setiap tahap pembuatan tugas akhir sesuai dengan prosedur yang berlaku serta dapat diimplementasikan. Sumber informasi dan referensi bisa didapatkan melalui buku, jurnal, dan internet.

1.6.3 Analisis dan Desain Perangkat Lunak

Pada tahap ini dilakukan analisis dan perancangan terhadap arsitektur tugas akhir yang akan dibuat. Tahap ini merupakan tahap yang paling penting dimana segala bentuk implementasi dapat berjalan dengan baik ketika arsitektur sistem juga baik.

1.6.4 Implementasi Perangkat Lunak

Pada tahap ini dilakukan implementasi atau realisasi dari anilisis dan perancangan arsitektur sistem yang sudah dibuat

sebelumnya, sehingga menjadi infrastruktur yang sesuai dengan apa yang direncanakan.

1.6.5 Pengujian dan Evaluasi

Pengujian akan dilakukan untuk mengukur performa layanan setelah diberlakukan implementasi infrastruktur yang dibuat pada tahap sebelumnya lalu akan dilakukan evaluasi terkait dengan hasil dari pengujian. Uji coba meliputi keberhasilan dari skala otomatis berjalan sebagai mana seharusnya dan pengaruh dari penambahan service terhadap ketersediaan layanan dan sumber daya yang ada.

1.6.6 Penyusunan Buku Tugas Akhir

Pada tahap ini dilakukan penyusunan buku tugas akhir yang berisi dokumentasi yang mencakup teori, konsep, implementasi dan hasil pengerjaan tugas akhir.

1.7 Sistematika Penulisan

Sistematika penulisan laporan tugas akhir secara garis besar adalah sebagai berikut :

1. Bab I. Pendahuluan
   Bab ini berisi penjelasan mengenai latar belakang, rumusan masalah, batasan masalah, tujuan, manfaat, metodologi dan sistematika penulisan dari pembuatan tugas akhir.
2. Bab II. Tinjauan Pustaka
   Bab ini berisi kajian teori atau penjelasan metode, algoritma, *library*, dan *tools* yang digunakan dalam pembuatan tugas akhir ini. Kajian teori yang dimaksud berisi tentang penjelasan singkat mengenai *Docker*, *Kubernetes*, *Sidecar Pattern*, dan *Kong Gateway*.

3. Bab III. Desain dan Perancangan
   Bab ini berisi mengenai analisis dan perancangan arsitektur sistem yang akan diimplementasikan dalam pembuatan tugas akhir.
4. Bab IV. Implementasi
   Bab ini berisi mengenai implementasi dari arsitektur sistem yang dibuat sebelumnya. Penjelasan berupa kode program dan pengaturan yang digunakan untuk implementasi arsitektur sistem.
5. Bab V. Pengujian dan Evaluasi
   Bab ini berisi tentang tahapan ujicoba terhadap performa arsitektur sistem dan evaluasi terhadap sistem yang dibuat.
6. Bab VI. Penutup
   Bab ini merupakan bab terakhir yang memaparkan kesimpulan dari hasil pengujian dan evaluasi yang telah dilakukan. Pada bab ini juga terdapat saran yang ditujukan bagi pembaca yang berminat untuk melakukan pengembangan terhadap tugas akhir ini.
7. Daftar Pustaka
   Bab ini berisi daftar pustaka yang dijadikan literatur dalam tugas akhir.
8. Lampiran
   Dalam lampiran terdapat kode sumber program secara keseluruhan.

(Halaman ini sengaja dikosongkan)

BAB II

TINJAUAN PUSTAKA

2.1 API Gateway

API gateway adalah sebuah titik masuk tunggal untuk semua client[2]. API gateway menerima semua panggilan API dari clients, dan mengarahkannya menuju ke microservice yang sesuai dengan request routing, komposisi, dan terjemahan protokol. Biasanya API gateway menangani sebuah request dengan meminta dari beberapa micorservice dan membandingkan hasilnya untuk menentukan mana yang terbaik. Hal ini dapat menerjemahkan antara protokol web yang baik dengan yang tidak baik yang digunakan secara internal[3].

2.2 Autoscaling

Scaling atau skala merupakan kemampuan sistem untuk menyesuaikan sumber daya yang dimiliki seperti menurunkan atau menambah jumlah proses sesuai kebutuhan sistem tanpa mengganggu proses yang sedang berjalan (Bhowmik, 2017). Sehingga yang disebut dengan autoscaling adalah kegiatan yang dilakukan secara otomatis untuk menurunkan atau menambah jumlah proses sesuai kebutuhan sistem[4].

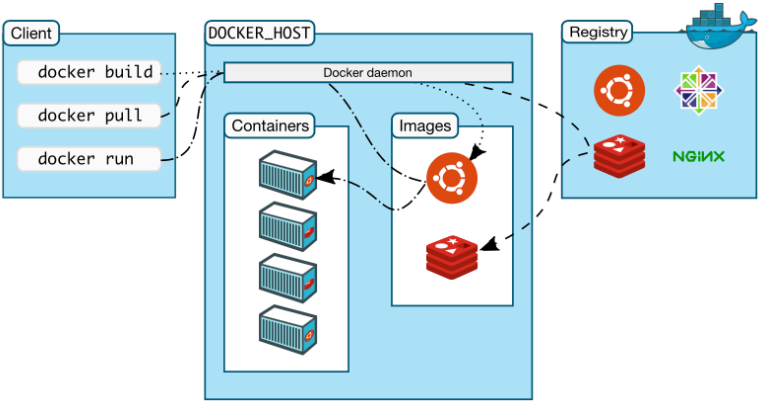
2.3 Docker

Docker adalah sebuah aplikasi yang bersifat *open source* yang berfungsi untuk mengemas sebuah aplikasi kedalam bentuk wadah/*container*. Pengguna bisa mempersiapkan berbagai kebutuhan baik menambah, mengurangi, maupun memodifikasi berbagai *file*/*resource* lainnya agar aplikasi yang tersebut bisa berjalan sesuai dengan yang diinginkan. Proses mempersiapkan ini nantinya akan menghasilkan dengan *Docker Image*. *Docker Image* inilah yang berikutnya bisa dijadikan sebuah *instance*

yang disebut *container*.

Docker memungkinkan pengguna untuk memisahkan aplikasi dengan infrastruktur sehingga bisa menjalankan program dengan cepat. Dengan Docker, pengguna dapat mengatur infrastruktur sama seperti pengguna tersebut mengatur aplikasi. Dengan memanfaatkan metodologi Docker untuk pengiriman, percobaan, dan menjalankan kode dengan cepat, pengguna bisa secara signifikan mengurangi delay antara menulis kode dan menjalankannya pada produksi.

Docker menggunakan arsitektur *client-server*. Docker *client* memanggil *daemon* yang nantinya bertugas untuk membangun, menjalankan dan mendistribusikan Docker *containers*. Docker *client* dan *daemon* bisa berjalan di satu sistem yang sama maupun berbeda karena Docker *client* dan *daemon* berkomunikasi dengan menggunakan *REST API*, melewati UNIX *sockets* atau *network interface*.



Gambar 2.1: Infrastruktur Docker[5]

Beberapa konsep kunci yang terdapat pada Docker adalah sebagai berikut:

1. *Image*
   Image adalah sebuah *read-only template* dengan berbagai instruksi untuk membuat *container*. Instuksi-instuksi ini biasanya diletakkan ke dalam suatu *file* yang disebut dengan Dockerfile. Terkadang *image* dibuat berdasarkan *image* yang lain dengan beberapa perubahan. *Image* yang dibuat sendiri tersebut bisa di sebarkan melalui Docker *image repository*, sehingga nantinya pengguna bisa dengan mudah menjalankan aplikasi tanpa harus melakukan *build* ulang.
2. *Container*
   Container adalah sebuah instan dari *image* yang dapat berjalan. *Container* bisa dibuat, dimulai, diberhentikan, dipindah, atau dihapus dengan menggunakan *API* dari Docker.
   Secara standar, *container* relatif terisolasi dari *container* yang lain dan dari komputer induk. Namun *container* tersebut bisa diatur sejauh mana tingkat isolasinya. Bisa diatur mulai dari jaringan, tempat penyimpanan atau hal subsistem lainnya. Tempat penyimpanan pada *container* secara standarnya tidaklah *persistent* sehingga ketika *container* tersebut dihapus, data di dalam *container* tersebut akan ikut terhapus. Namun hal ini juga bisa diatur agar *persistent* dengan menambah pengaturan pada Dockerfile[5].

2.4 Kong Gateway

Kong Gateway adalah sebuah aplikasi *API gateway* yang *open-source*, ringan sehingga bisa digunakan sebagai layanan *microservice*, memberikan kinerja latensi dan skalabilitas yang tak tertandingi. Kong Gateway adalah aplikasi berbasis Lua yang berjalan di Nginx. Kong Gateway bisa menggunakan Postgres

atau cassandra sebagai *database*-nya. Kong Gateway juga bisa *db-less* yang artinya semua data *routing* disimpan di *memory*. Untuk pengerjaan tugas akhir akan menggunakan Postgres[6].

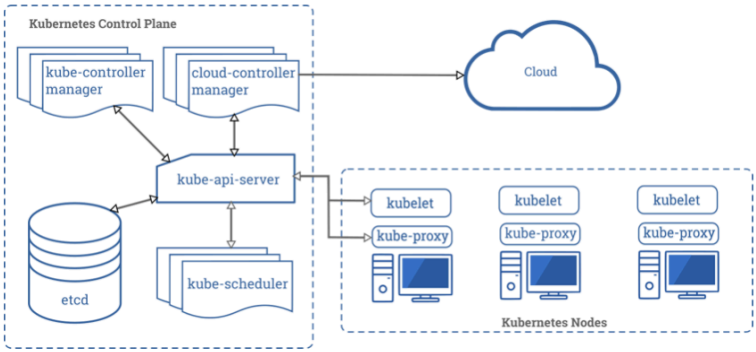
2.5 Kubernetes

Kubernetes merupakan sebuah aplikasi orkestrasi yang open-source untuk melakukan penyebaran otomatis, skala, dan managemen dari aplikasi berbasis container.

Jika hanya menggunakan Docker *container* biasa maka pihak operator akan sedikit kesulitan dalam hal *deployment* dan *maintenance*. Seperti ketika ada *container* yang perlu di skala atas karena beban sudah terlalu besar, secara standar Docker tidak bisa melakukannya secara otomatis. Contoh lainnya adalah *load balancing*. Pada Kubernetes sudah disediakan layanan *load balancing* dari dalam orkestrasi Kubernetes sehingga memudahkan operator agar tidak perlu melakukan setup *load balancer* terhadap *containers* yang ada. Jadi pada dasarnya Kubernetes ini menyiapkan sebuah *framewor*k bagi operator agar mempermudah dalam hal *deployment* dan *maintenance* sehingga mengurangi *downtime*.

Kubernetes memiliki beberapa komponen yang membuat Kubernetes ini bisa bekerja dengan baik. Komponen pertama adalah komponen *control plane*. Komponen *control plane* adalah 'otak' dibalik cara kerja dari Kubernetes. Komponen ini mengatur mulai dari *scheduling*, menyimpan data kluster, melakukan kontrol proses, dan masih ada banyak lagi fungsi dari komponen *control plane* ini. Komponen kedua adalah komponen *node*. Komponen ini berada di seluruh *node* yang terdaftar di Kubernetes. Fungsi dari *node* ini lebih kepada *maintenance* setiap *pod* agar bisa berjalan dengan baik. Selain itu juga komponen *node* ini bertugas untuk menyediakan data-data *pod* yang biasa disebut dengan *metric* seperti memori dan CPU

kepada `kube-api-server` yang ada di komponen *control plane*[7].



Gambar 2.2: Infrastruktur Kubernetes[7]

Beberapa konsep kunci yang terdapat pada Kubernetes adalah sebagai berikut:

1. *Pod*
   Pod merupakan unit terkecil yang ada di Kubernetes. *Pod* berisikan *container* yang menjalankan suatu aplikasi dari sebuah Docker *image*. *Pod* bisa menjalankan lebih dari satu *container*. *Pod* memiliki tempat penyimpanan, jaringan, dan cara menjalankan yang berbeda-beda. Namun jika suatu *pod* terdiri dari banyak *container*, maka biasanya *container* yang satu dan yang lain akan saling berbagi sumber daya[8].
2. *Deployment*
   Deployment adalah sebuah kumpulan dari *pod* yang memiliki fungsi yang sama. *Deployment* bersifat *stateless* yang artinya *pod* yang satu dengan yang lainnya dalam satu *deployment* tidak mempengaruhi satu sama lain.
3. *Stateful*

Stateful memiliki fungsi yang sama dengan *deployment* yaitu sebuah kumpulan dari *pod* yang memiliki fungsi yang sama, namun *stateful* memiliki peran yang relatif berbeda antar *pod*-nya. Sebagai contoh Postgres jika memiliki beberapa instan maka salah satu harus ada yang jadi *master* dan yang lainnya adalah *slave* untuk mencegah terjadinya tidak sinkron antar Postgres.

4. *Job*
   Job adalah *pod* yang bertugas menjalankan perintah tertentu. Ketika *pod* tersebut selesai menjalankan tugasnya maka pod tersebut akan dimatikan.
5. *Service*
   Service adalah suatu cara di Kubernetes untuk memaparkan sebuah aplikasi yang sedang berjalan pada satu kumpulan *pod*. *Service* ini bisa bertindak sebagai *load balancer* ketika *service* ini diarahkan kepada satu kumpulan *pod*. Secara standar, *load balancer* menggunakan algoritme round-robin dalam pengambilan keputusan.

2.6 Prometheus

Prometheus adalah aplikasi *open-source* untuk melakukan pemantauan dan peringatan kepada sistem. Prometheus awalnya dibangun di SoundCloud. Sejak 2012, banyak perusahaan dan organisasi telah mengadopsi Prometheus, dan proyek Prometheus ini memiliki pengembang dan komunitas yang aktif. Sekarang Prometheus merupakan proyek *standalone open-source* dan di kelola secara mandiri oleh perusahaan manapun. Untuk menekankan hal ini, dan untuk memperjelas struktur tata kelola proyek, Prometheus bergabung dengan yayasan Cloud Native Computing pada 2016 sebagai proyek yang di-*hosting* kedua, setelah Kubernetes[9].

2.7 Python

Python adalah bahasa pemrograman yang diinterpretasi, interaktif, dan berorientasi objek. Python menggabungkan *modules*, *exceptions*, pengetikan yang dinamik, tipe data dinamis tingkat tinggi, dan *classes*. Python juga memiliki sintaksis yang jelas. Python mempunyai antarmuka ke berbagai *system calls* dan *libraries*, juga berbagai macam *window systems*, dan dapat dikembangkan dalam C atau C++. Python bisa sebagai bahasa ekstensi untuk aplikasi yang memerlukan antarmuka. Python adalah bahasa pemrograman yang portabel karena bisa berjalan di berbagai macam variasi Unix, Mac, dan pada Windows 2000 ke atas[10].

Pada tugas akhir ini, Python akan digunakan sebagai *scraper* dari data stub Nginx yang dimiliki oleh *service* Kong Gateway untuk mendapat data terkait *active connection* pada Nginx. Setelah data tersebut didapat, Python juga akan mengkonversi data Nginx ke format *time series* milik Prometheus agar nantinya bisa diambil oleh Prometheus. Pada program Python juga akan membuat *web-server* untuk membuka *metrics* ke protokol http.

2.8 Rancher

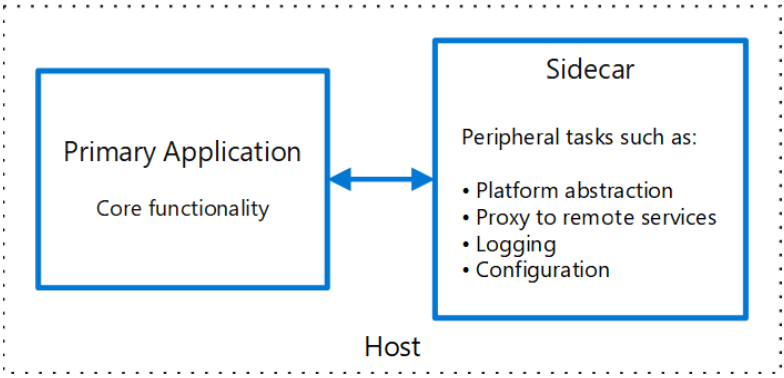
Rancher adalah sebuah aplikasi open-source yang menggabungkan semua kebutuhan organisasi dan mejalankan containers dalam lingkungan produksi. Dibuat dalam orkestrasi Kubernetes sehingga mempermudah tim DevOps untuk melakukan percobaan, penyediaan, dan mengatur aplikasinya[11].

Rancher memberi kemudahan karena dengan menyalakan satu Rancher *server*, bisa mengatur ribuan Kluster Kubernetes dari berbagai *nodes*[12].

2.9 Sidecar-Pattern

Sidecar pattern adalah *pattern* yang cara kerjanya mirip seperti sespan pada motor. *Sidecar* terhubung dengan *parent application* dan menyediakan bantuan kepada aplikasi. Jika *parent application* terbentuk maka *sidecar* akan juga otomatis terbentuk. Begitu juga ketika *parent application* mati, maka *sidecar* akan juga mati. Terkadang *sidecar pattern* biasa disebut *sidekick pattern* dan juga *decomposition pattern*[13].

Pada Kubernetes, Jika menggunakan *sidecar pattern* maka antara aplikasi dan *sidecar*-nya akan saling berbagi *resource* baik jaringan, media penyimpanan, dan sumber daya. Hal ini memungkinkan bagi *sidecar* untuk menjalankan kegiatan pendukung supaya aplikasi utama tidak terganggu fungsionalitasnya.



Gambar 2.3: *Sidecar Pattern*[13]

BAB III

DESAIN DAN PERANCANGAN

Pada bab ini dibahas mengenai analisis dan perancangan sistem.

3.1 Deskripsi Umum Sistem

Pada tugas akhir ini yang akan dibuat adalah perancangan infrastruktur untuk Kong Gateway yang mampu menyesuaikan jumlah *pod* pada orkestrasi Kubernetes sesuai dengan beban yang diterima oleh masing-masing *pod*. Kluster Kubernetes dibentuk menggunakan rancher dengan 2 *nodes*. *Node* pertama akan bertugas sebagai *master node* yang berfungsi untuk mengatur kluster termasuk pengaturan skala otomatis dari kluster. Sedangkan *node* kedua akan bertugas sebagai *worker node* yang berisi *pod* dari *deployments* yang ada.

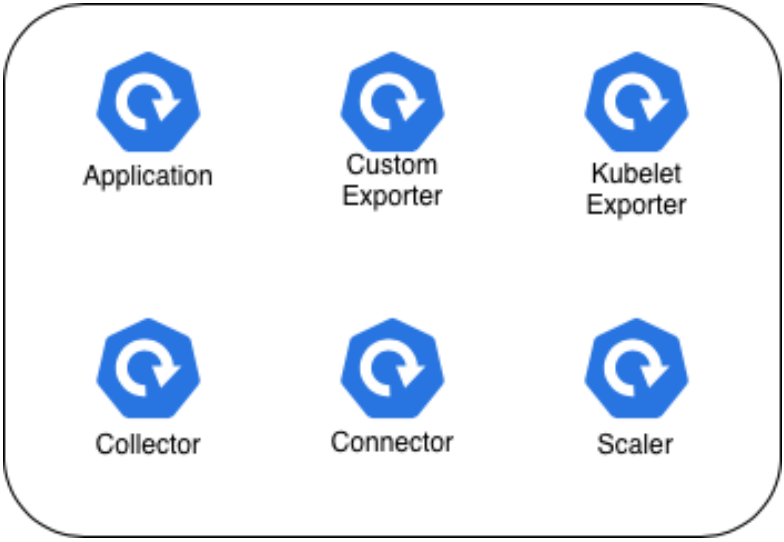
Untuk melakukan pemantauan koneksi aktif dari Kong Gateway, akan dibuat sebuah *container* yang bertugas sebagai pendukung aplikasi utama (Kong Gateway) yang disebut dengan *sidecar*. *Sidecar* ini bertugas untuk mengambil data dari stub Nginx yang digunakan oleh Kong Gateway lalu melakukan konversi menjadi data *time series* sesuai format pembacaan dari aplikasi *monitoring* yang digunakan. Monitoring akan dilakukan secara berkala dan terus menerus melakukan perubahan sesuai dengan keadaan *real time*. *Metrics* yang nantinya akan dipantau adalah *Memory*, CPU, dan koneksi aktif dari Kong Gateway. Pengambilan keputusan akan didasarkan dari ketiga hal ini dengan menentukan *threshold* masing-masing *metrics* sehingga *master node* bisa menentukan apakah aplikasi Kong Gateway perlu dilakukan skala atas atau skala bawah.

Pada tidak ditempatkan *pod* untuk postgres karena sifat postgres yang *stateful* dan memerlukan kesamaan data dari setiap *pod*-nya. Untuk itu postgres ditempatkan pada *master node* agar bisa menggunakan *block storage* yang sama sehingga bisa

dipastikan kesamaan datanya dimasing-masing *pod*. Untuk *deployment* yang sifatnya *stateless* akan dibuat *pod* secara acak sehingga memungkinkan untuk terbuat *pod* di *master node* atau *worker node*.

3.2 Arsitektur Umum Sistem

Pada sub-bab ini, dibahas mengenai tahap analisis arsitektur, analisis teknologi, dan desain sistem yang akan dibangun. Berikut adalah gambaran komponen-komponen yang ada dalam sistem



Gambar 3.1: Perancangan komponen skala otomatis

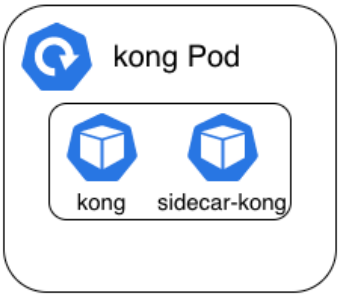
3.2.1 Desain Umum Sistem

Berdasarkan yang dijelaskan pada deskripsi umum sistem, dapat diperoleh kebutuhan sistem sebagai berikut:

1. *Sidecar Pattern* adalah pola deployment yang akan diterapkan pada *service* Kong Gateway.
2. Skala Otomatis adalah pembuatan kemampuan adaptif pada kluster sehingga terbuat sistem yang bisa menyesuaikan dengan beban layanan.

3.2.2 Perancangan *Sidecar Pattern*

Pada perancangan *sidecar pattern* akan dibuat sebuah *pod* yang memiliki pola *sidecar* di mana pada *container* utama akan memiliki sebuah *container* pendukung yang menjalankan suatu fungsi khusus. Pembentukan *container* pendukung secara khusus ini agar *container* utama tidak terganggu kerjanya. Pada prakteknya nanti *container* utama yang dimaksud adalah *container* yang memuat aplikasi Kong Gateway. Sedangkan *container* pendukung yang dimaksudkan adalah *container* yang berfungsi untuk mengambil data *custom metric* dan menyediakannya untuk diambil oleh *collector*.



Gambar 3.2: *Application* dan *Custom Exporter* pada Satu *Pod*

3.2.3 Perancangan Skala Otomatis

Untuk dapat menerapkan suatu sistem yang adaptif diperlukan enam komponen agar kemampuan skala otomatis bisa

berjalan dengan baik. Keenam komponen tersebut adalah sebagai berikut:

1. *Custom Exporter* adalah pemberi data yang sifat datanya adalah *custom* bukan dari kubelet.
2. *Application* adalah objek utama baik untuk pengambilan data maupun untuk di skala.
3. *Kubelet Exporter* adalah pemberi data yang sifat datanya sudah ada di kubelet namun hanya perlu di-*expose* saja.
4. *Collector* adalah sebuah komponen untuk mengumpulkan data *metrics* dari tiap *pod* yang disajikan oleh *custom exporter*.
5. *Connector* adalah komponen untuk menyambungkan antara *scaler* dengan data *metrics* dari *collector* agar data *metrics* tersebut bisa jadi salah satu pertimbangan oleh *scaler*.
6. *Scaler* adalah komponen untuk mengatur skala atas maupun skala bawah sesuai dengan data *metrics* yang disediakan.

3.2.3.1 Perancangan *Custom Exporter*

Custom exporter adalah sebuah komponen yang bertugas untuk menyiapkan data yang sifatnya tidak ada di kubelet seperti data koneksi aktif lalu data tersebut di konversi menjadi *time series* sesuai dengan format dari Prometheus.

Pada tugas akhir ini akan dibentuk 3 *custom metrics* pada *custom exporter*. Ketiga *custom metrics* tersebut adalah sebagai berikut:

1. *Active connection* atau koneksi aktif adalah *metric* untuk mendapatkan berapa jumlah koneksi yang terhubung ke suatu *pod* kong. Untuk mendapatkan jumlah koneksi aktif yang terhubung ke *pod* kong adalah dengan memanfaatkan *web server* Nginx yang digunakan oleh Kong Gateway. Nginx memiliki fitur *stub* data yang jika diaktifkan, data koneksi aktif akan dapat diakses pada suatu *url*. Kemudian membuat sebuah program Python untuk mengambil data

koneksi aktif pada suatu *url* tersebut dan menyediakannya kembali dalam bentuk *time series* Prometheus.

2. *Writing connecion* adalah *metrics* yang memuat nilai jumlah koneksi yang saat ini sedang dalam keadaan memproses *requests* sampai nanti dikembalikan ke *client*. Untuk mendapatkan *writing connection*, langkah yang digunakan adalah sama dengan mendapatkan *metric active connection*. Nilai *writing connection* juga disediakan dari fitur *stub* data Nginx.
3. *Request time* adalah *metric* yang berisi nilai rata-rata dari beberapa *requests* yang terjadi selama 1 detik. *Metric* ini bisa didapatkan dengan memanfaatkan fitur *logging* dari Nginx. Fitur *logging* Nginx menyimpan data *requests time* dari setiap *requests* yang terjadi. Dengan menggunakan program Python, berkas *logging* tersebut bisa diolah sedemikian sehingga data *request time* per detik dari beberapa *requests* bisa didapatkan. Cara yang digunakan adalah dengan menyalin berkas *logging* tersebut ke suatu folder lalu membandingkan dengan log sebelumnya untuk diambil data perubahannya saja lalu di rata-rata *request time* tersebut sehingga didapat nilai rata-rata *request time* per detik. Berkas yang disalin tadi dijadikan *file* pembanding untuk detik berikutnya.

3.2.3.2 Perancangan *Application*

Application adalah sebuah komponen target sebagai objek skala pada kluster. Selain itu *Application* juga berperan sebagai objek yang diambil datanya oleh baik *custom exporter* maupun oleh kubelet, yang nantinya diambil oleh kubelet *exporter*. Pada tugas akhir ini, Kong Gateway dilengkapi dengan postgres dan konga merupakan aplikasi utamanya. Pada saat melakukan *deployment* akan ditentukan berapa sumber daya CPU dan *memory* yang digunakan untuk keperluan skala nantinya. Selain

itu pada *deployment application* nantinya akan dibuat *load balancer* yang dinamakan *service* di orkestrasi Kubernetes sebagai pembagian beban antar pod pada satu *deployment* yang sama. Hal ini diperlukan agar ketika melakukan skala atas maupun skala bawah, *routing* pada Kubernetes tahu mana saja *pod* yang bisa digunakan mana yang tidak.

3.2.3.3 Perancangan *Kubelet Exporter*

Pada Kubernetes terdapat suatu aplikasi yang bernama kubelet. Kubelet menyimpan data dari seluruh *pod* yang ada di suatu *node*. Contoh datanya adalah bisa berupa penggunaan CPU atau *memory* pada masing-masing *container* yang ada pada suatu *pod*. Data ini tidaklah ter-*expose* untuk dijadikan *metric* sehingga perlu sebuah aplikasi *metric-server* agar data dari kubelet ini bisa dijadikan sebuah *metric* untuk nantinya jadi pertimbangan oleh *scaler*.

3.2.3.4 Perancangan *Collector*

Custom metric yang telah di-*expose* oleh *custom exporter* sifatnya adalah masih individu yang artinya jika ada 10 *pod* maka akan ada total 10 data *custom metrics*. Sehingga perlu dilakukan pengumpulan data untuk aplikasi yang sejenis agar nantinya bisa jadi pertimbangan untuk di skala. Pada tugas akhir kali ini *collector* akan menggunakan Prometheus sebagai aplikasinya.

Jika semisal data pada *custom exporter* dapat dilihat pada suatu *url* dan *port* maka *collector* perlu mengambil semua data dari *url* dan *port* tersebut untuk setiap *pod*-nya. Jika jumlah *pod* totalnya hanya 1, maka *collector* hanya perlu mengambil data menggunakan *url* dan *port* yang digunakan oleh *custom exporter*, namun permasalahan akan muncul ketika jumlah *pod* tidak hanya satu tetapi lebih dari satu. Jika *collector* menggunakan alamat *url* dan *port* akan bisa terjadi *load*

balancing pada *service* yang membuat data jadi tidak konsisten karena masing-masing *pod* memiliki beban masing-masing. Sehingga perlu diatur pada *collector* untuk menggunakan *regex* pada saat pengambilan data agar semua *metrics* pada setiap *pod* bisa terambil semua tanpa ter-*load balancing*.

3.2.3.5 Perancangan *Connector*

Data *metrics* yang terkumpul di *collector* masih belum bisa digunakan sebagai pertimbangan untuk melakukan skala pada *application*. *Metrics* tersebut perlu didaftarkan pada *custom metrics API* yang ada di Kubernetes agar bisa digunakan sebagai pertimbangan melakukan skala. Hal ini bisa dicapai dengan menggunakan aplikasi pihak ketiga yang bernama Prometheus-adapter. Prometheus-adapter akan menyambungkan antara *collector* yang menggunakan Prometheus dengan *scaler*.

3.2.3.6 Perancangan *Scaler*

Scaler adalah komponen yang bertugas untuk mengatur skala atas maupun skala bawah pada *application* sesuai dengan beban yang ada. Pada Kubernetes hal ini bisa dicapai dengan menggunakan *Horizontal Pod Autoscaling* (HPA). Pada *scaler* perlu ditetapkan seperti nilai batas wajar, *metrics* yang digunakan, target untuk melakukan skala.

3.3 Menentukan Algoritma *Scaling*

Kunci utama pada tugas akhir ini adalah keberhasilan melakukan skala baik skala atas maupun skala bawah sesuai dengan beban layanan yang ada. Penentuan algoritma *scaling* sangatlah penting karena sebagai penentu kapankah aplikasi perlu di skala atas atau kapankah aplikasi perlu di skala bawah. Keadaan yang paling optimal yang ingin dicari adalah ketika jumlah *worker* aplikasi sesuai dengan tingkat beban yang ada.

Jika *worker* terlalu banyak maka akan ada pemborosan sumber daya baik CPU dan *memory* untuk menjalankan *pod* yang seharusnya tidak diperlukan. Namun jika *worker pod* terlalu sedikit dan tidak sebanding dengan beban layanan yang ada maka sistem akan kewalahan dan bisa menyebabkan *restart service* yang bisa mengganggu tingkat keadaan aplikasi.

Pada Kubernetes algoritma yang digunakan pada *Horizontal Pod Autoscaling* (HPA) adalah dengan membandingkan jumlah *threshold* dengan nilai *metric* keadaan saat ini. Jika melebihi batas *threshold* maka HPA akan melakukan perhitungan desireReplicas dengan menggunakan rumus berikut[14]:

```
desRep = ceil[curRep*(curValue/desValue)]
```

Keterangan rumus di atas:

1. desRep adalah jumlah *pod* replikasi yang diinginkan.
2. curRep adalah jumlah *pod* replikasi saat ini.
3. curValue adalah nilai *metric* saat ini.
4. desValue adalah nilai *metric* yang diinginkan (*threshold*).

Sebagai contoh jika saat ini terdapat 1 *pod* yang berjalan lalu koneksi aktif yang diinginkan adalah 20 dan diketahui nilai *metric* saat ini adalah 60 maka jumlah *pod* replika yang diinginkan adalah 3 sehingga nantinya akan terbentuk 3 *pod* dari hasil skala atas.

Sedangkan algoritma skala bawah pada HPA milik Kubernetes adalah melihat keadaan dari nilai *metric* keadaan saat ini dan dibandingkan dengan *threshold*. Jika nilai *metric* keadaan saat ini lebih kecil dari *threshold* maka akan ditunggu 5 menit sebagai waktu *delay*. Jika sudah 5 menit tidak ada lonjakan nilai *metric* saat ini maka HPA akan melakukan skala bawah pada *deployment* yang ada.

Operator memiliki kemampuan untuk mengubah *threshold* dan mengatur berbagai waktu *delay* baik *delay* ketika sebelum dilakukan skala atas maupun sebelum dilakukan skala bawah. Namun untuk cara penghitungan `desireReplicas` sudah diatur Kubernetes terutama oleh *Horizontal Pod Autoscaling*.

Penentuan *threshold* dilakukan dengan cara mencari nilai CPU dan *memory* di masing-masing *container* pada suatu *pod* ketika sedang senggang dan dijadikan sebagai *threshold*. Hal ini dilakukan agar ketika aplikasi sedang senggang, tidak terjadi skala atas yang menyebabkan *pod* terus terbuat dan membuat sumber daya jadi habis terpakai. Sedangkan untuk menentukan batas atas CPU dan *memory*, dilakukan *multiple requests* secara bersamaan ke aplikasi Kong Gateway hingga nilai penggunaan CPU dan *memory* pada puncaknya. Nilai CPU dan *memory* ini nantinya akan dijadikan *resource limit*. *Resource limit* diperlukan untuk membatasi *pod* sehingga sumber daya bisa terbagi sama rata tanpa perlu adanya perebutan sumber daya antar *pod*. Perebutan sumber daya antar *pod* bisa menyebabkan *pod* mengalami *restart* karena kekurangan kapasitas baik CPU maupun *memory*. Selain itu *resource limit* diperlukan karena untuk penggunaan *memory* maupun CPU ada pilihan yaitu sebagai *targetvalue* dan *utilization*. Pada *targetvalue* data yang diterima merupakan nilai CPU ataupun *memory* yang sedang terpakai saat ini. Sedangkan *utilization* mengubah nilai *metric* untuk dijadikan persentase. Semisal CPU dibatasi 200m, dan dari *metric* ditemukan saat ini sudah terpakai 100m maka *utilization*-nya adalah 50%.

(Halaman ini sengaja dikosongkan)

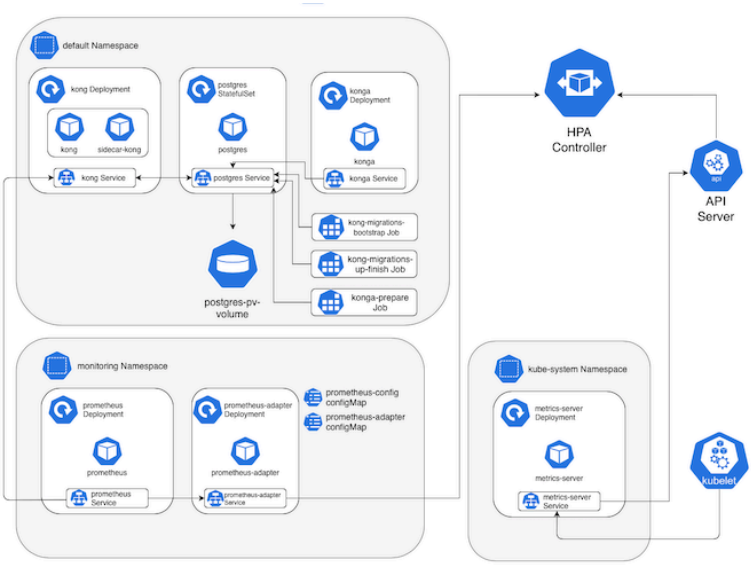
BAB IV

IMPLEMENTASI

Bab ini membahas mengenai implementasi sistem yang dilakukan berdasarkan rancangan yang telah dijabarkan pada bab sebelumnya. Implementasi berupa kode sumber untuk membangun program. Implementasi yang dijelaskan meliputi lingkungan pembangunan sistem, orkestrasi lingkungan sistem, dan implementasi komponen sistem.

4.1 Lingkungan Implementasi

Implementasi dilakukan pada lingkungan eksperimen yang disediakan oleh DPTSI. Dalam mengimplementasikan sistem pada lingkungan eksperimen tersebut, digunakan 3 buah perangkat *Virtual Private Server*.



Gambar 4.1: Kluster Api Gateway

4.1.1 Perangkat Keras

Perangkat keras yang digunakan dalam pengembangan sistem adalah sebagai berikut:

1. *Virtual Private Server* rancher dengan CPU Intel Xeon CPU E5-2690 v4 @ 2.60GHz (16 cores), RAM 32GB, 50GB penyimpanan.
2. *Virtual Private Server* api-gw-2-175 dengan CPU Intel Xeon Gold 5120 CPU @ 2.20GHz (16 cores), RAM 16GB, 100GB penyimpanan.
3. *Virtual Private Server* api-gw-2-176 dengan CPU Intel Xeon Gold 5120 CPU @ 2.20GHz (16 cores), RAM 8GB, 100GB penyimpanan.

Pada masing-masing *virtual private server* sudah diberikan alamat IP statik dan dengan pembagian peran sebagai berikut:

Tabel 4.1: Pembagian Peran dan alamat IP

No	*Hostname*	Peran	Alamat IP
1	rancher	rancher *server*	10.199.2.32
2	api-gw-2-175	*etcd*, *controlplane*, *worker*	10.199.2.175
3	api-gw-2-176	*worker*	10.199.2.176

4.1.2 Perangkat Lunak

Perangkat lunak yang digunakan dalam pengembangan adalah sebagai berikut:

1. Sistem Operasi Ubuntu 18.04.4 LTS 64 Bit.
2. Docker versi 19.03.8, *build* afacb8b7f0.
3. Rancher versi 2.3.5.
4. Kubernetes 1.17.

5. Prometheus versi 2.17.1.
6. Kong gateway *based image* versi 2.0-centos.
7. Konga 0.14.7 untuk pengaturan Kong Gateway menggunakan layanan web.
8. Postgres versi 9.6.
9. Python 3 untuk menghasilkan *custom metric*.

4.2 Mempersiapkan Orkestrasi Kubernetes

Orkestrasi Kubernetes digunakan untuk mengelola kluster pada tugas akhir ini. kluster yang terbuat nantinya akan terdiri dari berbagai macam *container* yang memiliki fungsinya masing-masing terutama untuk menyelesaikan masalah pada tugas akhir ini.

Pada DPTSI pengelolaan kluster dengan orkestrasi Kubernetes digunakanlah rancher sebagai pengaturnya dengan ekspetasi kedepan akan ada infrastruktur *multi-cluster*. Rancher bertugas untuk mengelola berbagai macam kluster yang terdaftar dalam sistem Rancher yang ditampilkan dalam bentuk web.

Langkah awal dalam mempersiapkan orkestrasi Kubernetes adalah melakukan pemasangan docker. Pemasangan docker dapat dilihat pada lampiran instalasi perangkat lunak. Setelah docker terpasang. Berikutnya adalah melakukan instalasi rancher server. Instalasi rancher server hingga menambahkan *etcd*, *controlplane*, *worker nodes* bisa dilihat pada lampiran instalasi perangkat lunak buku tugas akhir ini.

4.3 Implementasi *Sidecar Pattern*

Pemasangan *sidecar image* pada *main container* dilakukan dengan menambah *container* sehingga nantinya pada *pod* tersebut terdapat 2 *container* yaitu *main container* dan *sidecar container*. Oleh karena itu nantinya pada berkas konfigurasi

yang berfungsi untuk memasang kong perlu ditambahkan konfigurasi berikut agar *pod* kong bisa memiliki 2 *container* yaitu kong *container* dan *sidecar container*.

```
    spec:
      containers:
        - name: sidecar-kong
        image: lerufic/sidecar-kong:v2.0.0
        ports:
        - containerPort: 8888
          name: sidecarport
          protocol: TCP
        resources:
          requests:
            memory: "50Mi"
            cpu: "20m"
          limits:
            memory: "50Mi"
            cpu: "20m"
```

Kode Sumber IV.1: Menambah *sidecar container*

4.4 Implementasi Skala Otomatis

4.4.1 Implementasi *Custom Exporter*

Custom exporter bertugas untuk mendapatkan nilai *metric* yang nilai *metric*-nya tidak disediakan oleh *kubelet exporter* sehingga diperlukan aplikasi terpisah untuk mendapatkan nilai *metric*-nya. Pada tugas akhir ini *custom exporter* akan diimplementasikan pengambilan 3 *custom metrics* yang berbeda.

Pembentukan *custom metrics* akan dibuat menggunakan bahasa Python dengan pustaka `prometheus_client`. Sehingga *image* dasar yang digunakan adalah python:alpinedan akan bisa di akses di porta 8888.

4.4.1.1 Pembuatan *Custom Metric Active Connection*

Metric active connection berguna untuk mendapatkan berapa jumlah koneksi yang terhubung ke suatu *pod* kong. Data koneksi aktif tersebut disediakan oleh *stub* data Nginx. Pada implementasinya data tersebut yang berada di *container* utama harus diambil oleh *container* pendukung (*sidecar*) dengan memanfaatkan tingkah laku dari *sidecar-pattern* yaitu *container* utama dan *container* pendukung saling berbagi sumber daya salah satunya jaringan. Kemudian data tersebut disediakan kembali dalam bentuk *time series* Prometheus.

Untuk dapat membuat *metric active connection*. Beberapa pustaka Python perlu digunakan. *Stub* data Nginx disediakan dan dapat diakses melalui alamat *url* `http://127.0.0.1:8001/nginx_status`, maka perlu digunakan pustaka `requests`. Program Python akan mengambil data dari *url* tersebut lalu data yang diambil tersebut diolah agar bisa didapatkan nilainya. Setelah didapatkan nilainya, nilai tersebut diolah agar bisa menjadi format *time series* Prometheus.

```
...
66   active_conn = prom.Gauge('konghq_active',
     'Gauge untuk mendapatkan jumlah active
     connection')
...
78   url= "http://127.0.0.1:8001/nginx_status"
...
99   r = requests.get(url)
100  data = r.text.split(" ")
101  active_value = int(data[2])
...
109  active_conn.set(active_value)
...
```

Kode Sumber IV.2: Potongan program *custom metric active connection*

4.4.1.2 Pembuatan *Custom Metric Writing Connection*

Metric writing connection berguna untuk mendapatkan jumlah koneksi yang pada tahap pengolahan data setelah *request* diterima hingga dikirim kembali ke *client*. Sama seperti *active connection*, *writing connection* juga didapatkan dari *stub* data Nginx dari *container* utama Kong Gateway.

Pustaka yang digunakan adalah sama yaitu menggunakan `requests` untuk mengambil data *writing connection* dari *url* `http://127.0.0.1:8001/nginx_status`. Jika pada *metric active connection* data yang diambil adalah data `Active connection`, pada *metric writing connection* data yang diambil adalah data `writing`.

```
...
71   write_conn = prom.Gauge('konghq_writing',
     'Gauge untuk mendapatkan jumlah writing
     connection')
...
78   url= "http://127.0.0.1:8001/nginx_status"
...
99   r = requests.get(url)
100  data = r.text.split(" ")
...
106  write_value = int(data[13])
...
114  write_conn.set(write_value)
...
```

Kode Sumber IV.3: Potongan program *custom metric writing connection*

4.4.1.3 Pembuatan *Custom Metric Request Time*

Custom metric request time adalah sebuah *metric* yang menunjukan nilai rata-rata waktu dari *request* yang terjadi

selama 1 detik. Pengambilan data *metric* didapatkan dari fitur *logging* dari Nginx yang menyimpan *request time* dari setiap koneksi yang terjadi. Berkas *logging* tersebut ada pada *container* utama sehingga perlu diatur pada Implementasi *Application* untuk membuat sebuah *shared* folder antara *container* utama dengan *container* pendukung yang berada di folder `/cuslog`.

Pembuatan *custom metric request time* memerlukan beberapa pustaka seperti `os` dan *shutil*. Awalnya program Python akan melakukan pengecekan apakah berkas `/cuslog/custom_nginx.log` terdapat pada *container* pendukung. Jika tidak ada, maka otomatis nilai *request time* adalah 0, sedangkan jika ada maka akan dilakukan pengecekan lebih lanjut. Pengecekan berikutnya setelah berkas `/cuslog/custom_nginx.log` terbuat adalah dengan menyalin berkas tersebut menjadi `/app/new.log`. Jika terdapat `/app/old.log`, maka berkas `/app/new.log` akan dibandingkan dengan `/app/old.log` untuk mendapatkan nilai-nilai *requests* terbaru. Namun jika tidak ada berkas `/app/old.log`, maka akan dilakukan perhitungan keseluruhan isi dari `/app/new.log`. Setelah mendapatkan nilai rata-rata waktu dari beberapa *requests*, `/app/new.log` akan diubah namanya menjadi `/app/old.log` untuk menjadi pembanding pada detik berikutnya.

```
...
74  rtime_desc = prom.Gauge('konghq_rtime', '
    Gauge untuk mendapatkan rata-rata rtime tiap
     1 detik')
...
83  rtime_value = 0
84  if (os.path.exists('/cuslog/custom_nginx.
    log')):
85      if file_flag == 1:
86          shutil.copyfile('/cuslog/
```

```
custom_nginx.log', '/app/new.log')
87          rtime_value = readNewFile()
88          file_flag = 0
89      else:
90          shutil.copyfile('/cuslog/
custom_nginx.log', '/app/new.log')
91          same_line = checkFirstLine()
92          if same_line == 1:
93              rtime_value = diffFile()
94          else:
95              rtime_value = readNewFile()
96
97      shutil.move('/app/new.log', '/app/old.
log')
...
117  rtime_desc.set(rtime_value)
...
```

Kode Sumber IV.4: Potongan program *custom metric request time*

Kode sumber lengkap dari program Python dengan nama `main.py` yang bertujuan untuk membuat *custom metrics* bisa dilihat pada lampiran buku tugas akhir ini beserta dengan berkas `requirements.txt` yang berisi daftar pustaka yang digunakan untuk mendukung kerja dari program Python tersebut.

Setelah terbuatnya program Python untuk membuat *custom metric*, langkah berikutnya adalah membuat program Python ini bisa bekerja sebagai *container* pendukung pada *pod* kong nantinya. Untuk membuat *image* sidecar-kong, perlu dibuat sebuah berkas `Dockerfile` agar *image* dasar python:alpine secara otomatis menjalankan program python yang telah dibuat ketika dijalankan nantinya.

```
FROM python:alpine
WORKDIR /app
```

```
COPY main.py /app
COPY requirements.txt /app
RUN cd /app && pip install -r requirements.
    txt
EXPOSE 8888
CMD ["python", "/app/main.py"]
```

Kode Sumber IV.5: Dockerfile *sidecar*

Setelah itu perlu dilakukan pembangunan *image* dan disimpan pada *docker image repository* agar nantinya bisa di akses oleh Kubernetes pada *server*.

```
$ docker build -t lerufic/sidecar-kong:v2.0.0
$ docker Login
$ docker push lerufic/sidecar-kong:v2.0.0
```

Kode Sumber IV.6: Membangun *image sidecar*

4.4.2 Implementasi *Application*

Implementasi *Application* terbagi kedalam beberapa tahapan penting yang dimulai dari pembuatan *custom image* Kong Gateway dan pemasangan *application*.

4.4.2.1 Pembuatan *Custom Image* Kong Gateway

Image bawaan dari Kong Gateway tidak memperlihatkan nilai stub dari Nginx secara terbuka. Selain itu pengaturan *worker processes*, *worker rlimit*, *worker connection* pada Nginx juga perlu disesuaikan agar penggunaan *memory* dan CPU tidak terlalu besar untuk setiap *pod*-nya. Penyesuaian hal tersebut dilakukan dengan membuat ulang *image* kong dan menaruhnya pada *docker image repository*.

Untuk dapat memperlihatkan nilai stub pada Nginx perlu membuat berkas `nginx_kong.lua` dan `nginx.lua` lalu

mengisinya dengan pengaturan sesuai pada kode sumber di lampiran buku tugas akhir ini.

Hal tersebut memungkinkan untuk nantinya *sidecar* dari *container* kong untuk bisa mengakses nilai stub agar bisa mengambil nilai *active connections* yang nantinya sebagai pertimbangan skala otomatis pada kluster. Perilaku dari *sidecar-pattern* adalah *main container* dan *sidecar container* saling berbagi jaringan maka nilai stub Nginx hanya bisa diakses pada IP address 127.0.0.1 dan menolak jika ada akses dari luar.

Perubahan berikutnya adalah mengatur *worker processes*, *worker rlimit*, dan *worker connection* untuk membatasi penggunaan CPU dan *memory*. Untuk melakukan perubahan ini perlu dibuat sebuah berkas bernama `kong_defaults.lua`. Untuk kode sumber lengkapnya terdapat pada lampiran tugas akhir ini. Bagian yang melakukan perubahan terhadap ketiga nilai tersebut adalah terletak pada baris 39, 40, dan 41. Nilai ketiga hal tersebut disesuaikan dengan mempertimbangkan kebutuhan dan sumber daya kluster Kubernetes.

```
...
39  nginx_main_worker_processes = 4
40  nginx_main_worker_rlimit_nofile = 65536
41  nginx_events_worker_connections = 16384
...
```

Kode Sumber IV.7: Pengaturan kong defaults

Berikutnya setelah kedua berkas tersebut dibuat adalah membuat berkas `Dockerfile` untuk memasukan berkas `nginx_kong.lua` dan `kong_defaults.lua` kedalam *based image* Kong Gateway agar nantinya perubahan kedua berkas tersebut bisa di-*compile* oleh aplikasi Kong Gateway. Isi dari `Dockerfile` tersebut adalah sebagai berikut.

```
FROM kong:2.0-centos
```

```
COPY nginx_kong.lua /usr/local/share/lua/5.1/
    kong/templates/nginx_kong.lua
COPY kong_defaults.lua /usr/local/share/lua/5.1
    /kong/templates/kong_defaults.lua
```

Kode Sumber IV.8: Dockerfile kong

Setelah `Dockerfile` terbuat, berikutnya adalah proses *build custom image* dan menyimpan *image* tersebut dalam *docker image repository*. Untuk memenuhi hal tersebut perlu menjalakan beberapa perintah di terminal sebagai berikut.

```
$ docker build -t lerufic/kong-its:v2.0.7
$ docker Login
$ docker push lerufic/kong-its:v2.0.7
```

Kode Sumber IV.9: Membangun image Kong Gateway

4.4.2.2 Pemasangan Application

Setelah semua *image* yang dibutuhkan sudah tersedia di *docker image repository*, langkah berikutnya adalah memasang *application* hingga memastikan *application* tersebut bisa berjalan sebagai mana mestinya. Langkah-langkah yang perlu dilakukan haruslah dilakukan dengan berurutan agar tidak terjadi kegagalan ketika pemasangan nantinya.

Langkah pertama adalah pemasangan postgreSQL. Pemasangan postgreSQL terdiri dari beberapa langkah yaitu melakukan reservasi volume pada *server* agar data yang nantinya simpan akan tetap tersimpan secara permanen karena secara default ketika pod postgres dihapus maka data juga akan terhapus. Akan ada 2 *kind* yang terbuat yaitu `PersistentVolume` dan `PersistentVolumeClaim`. `PersistentVolume` (PV) berguna untuk melakukan reservasi tempat penyimpanan pada *server*. Sedangkan

`PersistentVolumeClaim` (PVC) adalah menetapkan `PersistentVolume` tersebut agar digunakan oleh suatu *pod*. Untuk melakukan reservasi *storage* perlu dibuat sebuah berkas `postgres-persistent.yaml` yang isinya bisa dilihat pada kode sumber lampiran tugas akhir ini.

```
...
9   storageClassName: manual
10  capacity:
11    storage: 5Gi
...
12  accessModes:
13    - ReadWriteMany
14  hostPath:
15    path: "/mnt/data"
...
27  resources:
28    requests:
29      storage: 5Gi
```

Kode Sumber IV.10: Potongan postgres-persistent.yaml

Pada potongan berkas `postgres-persistent.yaml` di atas adalah beberapa nilai penting. Pada baris 11 dan 29 adalah besar tempat penyimpanan yang ingin direservasi. Pada kasus di atas, besar tempat penyimpanan yang ingin direservasi adalah 5Gi. Pada baris 15 adalah konfigurasi untuk mengatur tempat di mana data disimpan. Pada kasus di atas, berkas akan disimpan pada folder `/mnt/data`. Ketika *pod* postgres nantinya dihapus data akan tetap bisa kembali karena semua data postgres nantinya akan disimpan secara permanen di folder `/mnt/data`. Untuk melakukan pemasangan PV dan PVC, bisa dilakukan dengan melakukan perintah berikut.

```
$ kubectl -f apply postgres-persistent.yaml
```

Kode Sumber IV.11: Pemasangan PV dan PVC

Setelah memasang PV dan PVC, berikutnya adalah memasang postgres pada kluster Kubernetes. *Kind* yang akan digunakan pada pemasangan postgres adalah `StatefulSet` agar *pod* yang dibuat terurut dan memiliki peran. Sebelum dilakukan pemasangan `StatefulSet`, perlu dibuat sebuah berkas `postgres-deployment.yaml` yang isinya sebagai berikut.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgres
  namespace: default
spec:
  serviceName: "postgres"
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      nodeName: api-gw-2-175
      containers:
        - env:
          - name: POSTGRES_DB
            value: kong
          - name: POSTGRES_USER
            value: kong
          - name: POSTGRES_PASSWORD
```

```
              value: kong
            - name: POSTGRES_HOST_AUTH_METHOD
              value: trust
            name: postgres
            image: postgres:9.6
            resources:
              requests:
                memory: "200Mi"
                cpu: "200m"
              limits:
                memory: "200Mi"
                cpu: "200m"
            ports:
            - containerPort: 5432
              name: dbport
              protocol: TCP
            volumeMounts:
            - mountPath: /var/lib/postgresql/data
              name: postgredb
        restartPolicy: Always
        volumes:
          - name: postgredb
            persistentVolumeClaim:
              claimName: postgres-pv-claim
```

Kode Sumber IV.12: Berkas postgres-deployment.yaml

Berikutnya adalah melakukan pemasangan pada kluster Kubernetes dengan perintah berikut.

```
$ kubectl -f apply postgres-deployment.yaml
```

Kode Sumber IV.13: Pemasangan postgreSQL

Karena tingkah laku dari Kubernetes yang membuat setiap *pod* tersebut terisolasi, maka perlu dibuat sebuah *service* agar

postgreSQL bisa diakses dari luar. Oleh karena itu perlu dibuat berkas dengan nama `postgres-service.yaml` yang berisi sebagai berikut.

```
apiVersion: v1
kind: Service
metadata:
  name: postgres
  labels:
    app: postgres
spec:
  type: NodePort
  ports:
   - port: 5432
  selector:
   app: postgres
```

Kode Sumber IV.14: Berkas postgres-service.yaml

Untuk pemasangan *service* postgres bisa dilakukan dengan menjalankan perintah berikut.

```
$ kubectl -f apply postgres-service.yaml
```

Kode Sumber IV.15: Pemasangan *service* postgres

Setelah sudah melakukan pemasangan postgreSQL, langkah berikutnya adalah menyiapkan *database* untuk Kong Gateway. Untuk itu perlu dipersiapkan 3 berkas yaitu `migration-bootstrap.yaml`, `migration-up.yaml`, dan yang terakhir adalah `konga-prepare.yaml`. Isi ketiga berkas bisa dilihat di lampiran kode sumber pada tugas akhir ini. Pemasangan ketiga berkas ini harus dilakukan secara berurutan dan dijalankan sesuai dengan perintah di bawah ini.

```
$ kubectl -f apply migration-bootstrap.yaml
$ kubectl -f apply migration-up.yaml
```

```
$ kubectl -f apply konga-prepare.yaml
```

Kode Sumber IV.16: Mempersiapkan *database* Kong Gateway

Berikutnya adalah pemasangan aplikasi konga agar pengembang bisa melakukan pengaturan API *gateway* menggunakan GUI. Untuk memasang konga, perlu dibuat berkas `konga-deployment.yaml` dan `konga-service.yaml` yang isinya bisa dilihat pada lampiran kode sumber. Untuk memasangnya perlu dijalankan perintah berikut.

```
$ kubectl -f apply konga-deployment.yaml
$ kubectl -f apply konga-service.yaml
```

Kode Sumber IV.17: Pemasangan konga

Langkah terakhir adalah pemasangan aplikasi utama yaitu Kong Gateway. Untuk melakukan pemasangan Kong Gateway perlu dibuat berkas `kong-deployment.yaml` dan `kong-service.yaml`. berkas `kong-service.yaml` bisa dilihat pada lampiran kode sumber. Untuk `kong-deployment.yaml` berisi konfigurasi seperti berikut.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kong
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kong
  template:
    metadata:
      labels:
```

```
        app: kong
    spec:
      containers:
      - env:
        - name: KONG_ADMIN_ACCESS_LOG
          value: /dev/stdout
        - name: KONG_ADMIN_ERROR_LOG
          value: /dev/stderr
        - name: KONG_ADMIN_LISTEN
          value: 0.0.0.0:8001, 0.0.0.0:8444 ssl
        - name: KONG_CASSANDRA_CONTACT_POINTS
          value: kong-database
        - name: KONG_DATABASE
          value: postgres
        - name: KONG_PG_HOST
          value: postgres
        - name: KONG_PG_USERNAME
          value: kong
        - name: KONG_PG_PASSWORD
          value: kong
        - name: KONG_PROXY_ACCESS_LOG
          value: /dev/stdout
        - name: KONG_ADMIN_ACCESS_LOG
          value: /dev/stdout
        - name: KONG_PROXY_ERROR_LOG
          value: /dev/stderr
        - name: KONG_ADMIN_ERROR_LOG
          value: /dev/stderr
        - name: KONG_LOG_LEVEL
          value: debug
        - name: KONG_PG_HOST
          value: postgres
        - name: KONG_PLUGINS
```

```
            value: bundled,myplugin
          name: kong
          image: lerufic/kong-its:v2.0.7
          ports:
          - containerPort: 8000
            name: port1
            protocol: TCP
          - containerPort: 8443
            name: port2
            protocol: TCP
          - containerPort: 8001
            name: port3
            protocol: TCP
          - containerPort: 8444
            name: port4
            protocol: TCP
          volumeMounts:
          - name: shared-folder
            mountPath: /cuslog
          resources:
            requests:
              memory: "400Mi"
              cpu: "200m"
            limits:
              memory: "400Mi"
              cpu: "200m"
        - name: sidecar-kong
          image: lerufic/sidecar-kong:v2.0.0
          ports:
          - containerPort: 8888
            name: sidecarport
            protocol: TCP
          volumeMounts:
```

```
          - name: shared-folder
            mountPath: /cuslog
          resources:
            requests:
              memory: "50Mi"
              cpu: "20m"
            limits:
              memory: "50Mi"
              cpu: "20m"
        volumes:
        - name: shared-folder
          emptyDir: {}
        restartPolicy: Always
```

Kode Sumber IV.18: Berkas kong-deployment.yaml

Pada konfigurasi di atas, sudah diimplementasi *sidecar pattern* sesuai dengan subbab 4.3.2 Implementasi *Sidecar Pattern*. Ketika akan melakukan skala otomatis perlu dilakukan penentuan *resource* agar skala bisa melakukan reservasi terlebih dahulu sehingga tidak membebani kluster. Untuk melakukan pemasangan Kong Gateway dan *service*-nya bisa dilakukan dengan perintah berikut.

```
$ kubectl -f apply kong-deployment.yaml
$ kubectl -f apply kong-service.yaml
```

Kode Sumber IV.19: Memasang Kong Gateway

Untuk memastikan semua bisa berjalan bisa dilakukan *request* HTTP menggunakan *curl* menuju alamat IP dari *deployment* kong dan *port* 8001. Jika *header* yang diterima 200 maka *application* berhasil terpasang.

```
$ curl -i http://10.199.2.175:8001/
```

Kode Sumber IV.20: Mengecek Kong Gateway

4.4.3 Implementasi *Kubelet Exporter*

Kubelet Exporter bertugas untuk menyediakan nilai *metrics* yang sifatnya sudah ada pada aplikasi kubelet. Secara standar nilai *metrics* dari kubelet tidak diperlihatkan dan digunakan untuk internal Kubernetes. Oleh karena itu diperlukan aplikasi pendukung untuk dapat melihat nilai *metrics*-nya. Nilai *metrics* yang ingin diambil oleh *kubelet* exporter pada tugas akhir ini berupa *memory* dan CPU.

Mengimplementasi *kubelet exporter* dilakukan dengan cara memasang aplikasi metric-server pada kluster Kubernetes. Untuk dapat memasang metric-server, sebelumnya perlu disiapkan beberapa berkas terlebih dahulu. Beberapa berkas tersebut adalah `aggregated-metric-read.yaml`, `auth-delegator.yaml`, `auth-reader-conf.yaml`, `metrics-apiservice.yaml`, `metrics-deployment.yaml`, `metric-service.yaml`, dan `resource-reader.yaml`. Isi berkas tersebut dapat dapat dilihat pada lampiran kode sumber.

Pemasangan metric-server dilakukan secara berurutan sesuai dengan urutan perintah berikut.

```
$ kubectl -f apply aggregated-metric-read.yaml
$ kubectl -f apply auth-delegator.yaml
$ kubectl -f apply auth-reader-conf.yaml
$ kubectl -f apply metrics-apiservice.yaml
$ kubectl -f apply metrics-deployment.yaml
$ kubectl -f apply metric-service.yaml
$ kubectl -f apply resource-reader.yaml
```

Kode Sumber IV.21: Memasang metric-server

4.4.4 Implementasi *Collector*

Nilai *metrics* pada *custom exporter* masih terpisah sehingga *collector* bertugas untuk mengumpulkan data *metrics* dari

berbagai *pod* yang sejenis agar bisa diolah. Aplikasi yang digunakan sebagai *collector* pada tugas akhir ini adalah Prometheus. Prometheus akan mengambil nilai *metrics* dari *pod* berlabel kong sehingga semua data *metrics* dari berbagai *pod* akan terkumpul di Prometheus. Untuk melakukan pemasangan Prometheus perlu dipersiapkan beberapa hal. Agar infrastruktur server lebih terkategori, Prometheus akan ditaruh pada *namespace monitoring*. Berikutnya adalah menyiapkan beberapa berkas yaitu `prometheus-namespace.yaml`, `prometheus-clusterrole.yaml`, `prometheus-clusterrolebinding.yaml`, dan `prometheus-serviceaccount.yaml` yang isinya sesuai pada lampiran. Untuk memasang berkas tersebut bisa dilakukan dengan perintah berikut.

```
$ kubectl -f apply prometheus-namespace.yaml
$ kubectl -f apply prometheus-clusterrole.yaml
$ kubectl -f apply prometheus-
    clusterrolebinding.yaml
$ kubectl -f apply prometheus-serviceaccount.
    yaml
```

Kode Sumber IV.22: Memasang kebutuhan Prometheus

Secara standar jika mengikuti alur pemasangan Prometheus biasa, aplikasi Prometheus tidak bisa menemukan bahwa *pod* kong memiliki *exporter* pada *port* 8888-nya sehingga perlu dibuat sebuah *configMap* agar ketika nantinya memasang Prometheus, aplikasi Prometheus yang dipasang bisa menggunakan pengaturan yang sudah ditentukan sebelumnya. Pertama yang perlu dilakukan adalah membuat berkas `prometheus-configMap.yaml` isi lengkapnya bisa dilihat pada lampiran kode sumber.

```
...
23 scrape_configs:
```

```
24    - job_name: 'kong'
25
26      kubernetes_sd_configs:
27      - role: pod
28
29      relabel_configs:
30      - source_labels: [
    __meta_kubernetes_namespace]
31        action: replace
32        target_label: k8s_namespace
33      - source_labels: [
    __meta_kubernetes_pod_name]
34        action: replace
35        target_label: k8s_pod_name
36      - source_labels: [__address__]
37        action: replace
38        regex: ([^:]+)(?::\d+)?
39        replacement: ${1}:8888
40        target_label: __address__
41      - source_labels: [
    __meta_kubernetes_pod_label_app]
42        action: keep
43        regex: kong
...
```

Kode Sumber IV.23: Potongan prometheus-configMap.yaml

Potongan konfigurasi di atas adalah untuk mengatur Prometheus agar dapat mengambil *metric* dari *pod* dengan *regex* label kong dan mengambil pada *port* 8888. Berikutnya adalah memasang *configMap* tersebut ke dalam kluster Kubernetes dengan perintah seperti berikut.

```
$ kubectl -f apply prometheus-configMap.yaml
```

Kode Sumber IV.24: Memasang configMap

Setelah memasang *configMap*, langkah berikutnya adalah membuat berkas `prometheus-service.yaml` sesuai dengan lampiran kode sumber yang terdapat di tugas akhir ini dan kemudian membuat berkas `prometheus-deployment.yaml` yang berisi konfigurasi seperti berikut.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: prometheus
  namespace: monitoring
spec:
  selector:
    matchLabels:
      app: prometheus
  replicas: 1
  template:
    metadata:
      labels:
        app: prometheus
    spec:
      serviceAccountName: prometheus
      containers:
      - name: prometheus
        image: prom/prometheus:v2.17.1
        args:
            - "--config.file=/etc/prometheus/
                prometheus.yml"
            - "--storage.tsdb.path=/prometheus
                /"
        ports:
        - containerPort: 9090
          name: default
        volumeMounts:
```

```
          - name: prometheus-config-volume
            mountPath: /etc/prometheus/
          - name: prometheus-storage-volume
            mountPath: /prometheus/
        volumes:
        - name: prometheus-config-volume
          configMap:
            defaultMode: 420
            name: prometheus-config
        - name: prometheus-storage-volume
          emptyDir: {}
```

Kode Sumber IV.25: Berkas prometheus-deployment.yaml

Berikutnya setelah kedua berkas tersebut berhasil dibuat adalah memasang kedua berkas tersebut ke dalam kluster dengan perintah ini.

```
$ kubectl -f apply prometheus-deployment.yaml
$ kubectl -f apply prometheus-service.yaml
```

Kode Sumber IV.26: Memasang Prometheus

4.4.5 Implementasi *Connector*

Collector yang sudah dibuat tidaklah bisa secara langsung diakses oleh *Scaler* untuk digunakan dalam skala otomatis. Data *metric* yang terkumpul tersebut semua masih terdapat dalam aplikasi Prometheus sehingga perlu adanya *connector* yang bertugas untuk menjembatani antara *scaler* dengan *collector*. Untuk mengimplementasikan *connector* akan digunakan sebuah aplikasi pihak ketiga yang bernama Prometheus-adapter. Prometheus-adapter adalah aplikasi untuk mengimplementasi API di custom.metrics.k8s.io menggunakan Prometheus.

Pada implementasi *connector* di tugas akhir ini, akan juga

dimplementasikan bukan hanya untuk *custom metric* melainkan juga untuk *external metric*. *Custom metric* berfokus pada nilai *metric* yang ada *pod* tersebut yang tidak disediakan oleh *kubelet exporter*. Sedangkan *external metric* adalah penggunaan nilai *metric* pada suatu *pod* dengan memanfaatkan nilai *metric* dari *pod* yang lain.

Untuk memasang Prometheus-adapter akan sedikit berbeda dengan pemasangan aplikasi sebelumnya karena untuk Prometheus akan digunakan helm chart dengan memanfaatkan *custom* `values.yaml` yang sudah dimodifikasi agar bisa langsung menyambungkan nilai dari *collector* yang sudah dibuat sebelumnya dengan *scaler*. Langkah pertama adalah menyiapkan berkas `values.yaml` yang berisi konfigurasi seperti berikut.

```
# Default values for k8s-prometheus-adapter..
affinity: {}

image:
  repository: directxman12/k8s-prometheus-
     adapter-amd64
  tag: v0.6.0
  pullPolicy: IfNotPresent

logLevel: 10

metricsRelistInterval: 1m

nodeSelector: {}

priorityClassName: ""

# Url to access prometheus
prometheus:
  url: http://prometheus.monitoring
```

```
    port: 9090

  replicas: 1

  rbac:
    # Specifies whether RBAC resources should be
        created
    create: true

  serviceAccount:
    # Specifies whether a service account should
        be created
    create: true
    # The name of the service account to use.
    # If not set and create is true, a name is
        generated using the fullname template
    name:

  resources: {}
    # requests:
    #   cpu: 100m
    #   memory: 128Mi
    # limits:
    #   cpu: 100m
    #   memory: 128Mi

  rules:
    default: false
    custom:
    - seriesQuery: '{__name__="konghq_active"}'
      seriesFilters: []
      resources:
        overrides:
```

```
          k8s_namespace:
            resource: namespace
          k8s_pod_name:
            resource: pod
      name:
        matches: "konghq_active"
        as: "konghq_active"
      metricsQuery: <<.Series>>{<<.LabelMatchers
          >>,container_name!="POD"}
    existing:
    external:
    - seriesQuery: '{__name__="konghq_active"}'
      seriesFilters: []
      resources:
        overrides:
          k8s_namespace:
            resource: namespace
          k8s_pod_name:
            resource: pod
      name:
        matches: "konghq_active"
        as: "konghq_active"
      metricsQuery: <<.Series>>{<<.LabelMatchers
          >>,container_name!="POD"}
    resource: {}
  #   cpu:
  #     containerQuery: sum(rate(
      container_cpu_usage_seconds_total{<<.
      LabelMatchers>>}[3m])) by (<<.GroupBy>>)
  #     nodeQuery: sum(rate(
      container_cpu_usage_seconds_total{<<.
      LabelMatchers>>, id='/'}[3m])) by (<<.
      GroupBy>>)
```

```
#     resources:
#       overrides:
#         instance:
#           resource: node
#         namespace:
#           resource: namespace
#         pod:
#           resource: pod
#     containerLabel: container
#   memory:
#     containerQuery: sum(
    container_memory_working_set_bytes{<<.
    LabelMatchers>>}) by (<<.GroupBy>>)
#     nodeQuery: sum(
    container_memory_working_set_bytes{<<.
    LabelMatchers>>,id='/'}) by (<<.GroupBy>>)
#     resources:
#       overrides:
#         instance:
#           resource: node
#         namespace:
#           resource: namespace
#         pod:
#           resource: pod
#     containerLabel: container
#   window: 3m

service:
  annotations: {}
  port: 443
  type: ClusterIP

tls:
```

```
    enable: false
    ca: |-
      # Public CA file that signed the APIService
    key: |-
      # Private key of the APIService
    certificate: |-
      # Public key of the APIService

  # Any extra volumes
  extraVolumes: []
    # - name: example-name
    #   hostPath:
    #     path: /path/on/host
    #     type: DirectoryOrCreate
    # - name: ssl-certs
    #   hostPath:
    #     path: /etc/ssl/certs/ca-bundle.crt
    #     type: File

  # Any extra volume mounts
  extraVolumeMounts: []
    #   - name: example-name
    #     mountPath: /path/in/container
    #   - name: ssl-certs
    #     mountPath: /etc/ssl/certs/ca-
        certificates.crt
    #     readOnly: true

  tolerations: []

  # Labels added to the pod
  podLabels: {}
```

```
# Annotations added to the pod
podAnnotations: {}

hostNetwork:
  # Specifies if prometheus-adapter should be
     started in hostNetwork mode.
  #
  # You would require this enabled if you use
     alternate overlay networking for pods and
  # API server unable to communicate with
     metrics-server. As an example, this is
     required
  # if you use Weave network on EKS
  enabled: false
```

Kode Sumber IV.27: Berkas values.yaml

Setelah membuat berkas tersebut, proses dilanjutkan dengan pemasangan Prometheus-adapter dengan memanfaatkan helm chart. aplikasi Prometheus-adapter yang akan digunakan adalah yang sudah *stable* dan nantinya akan dipasang di *namespace* yang sama dengan Prometheus agar lebih terkategorikan. Untuk menjalankan hanya perlu menjalankan perintah berikut ini.

```
$ helm install prometheus-adapter -f values.
    yaml -n monitoring stable/prometheus-adapter
```

Kode Sumber IV.28: Memasang Prometheus-adapter

4.4.6 Implementasi *Scaler*

Scaler bertugas untuk mengatur jumlah *pod* yang ada sesuai dengan data *metric* dari *kubelet exporter* dan *collector* yang sudah tersambung dengan HPA controller melalui *connector*. *Scaler* akan meningkatkan jumlah *pod* jika nilai *metric* saat ini

lebih besar dari batas yang sudah ditentukan. Juga *scaler* bertugas untuk menurunkan jumlah *pod* jika nilai *metric* saat ini lebih kecil dibanding batas yang sudah ditetapkan. Untuk implementasinya Kubernetes sudah memiliki HPA controller dengan *kind* `HorizontalPodAutoscaler`, sehingga hanya perlu melakukan konfigurasi untuk mengatur *scaler*.

Pada implementasi *scaler* akan ada 2 berkas yang dibuat yaitu `kong-customhpa.yaml` yang bertugas untuk mengatur *scaler* dari *pod* kong dengan *metric* CPU dan *custom metric* `konghq_active`, `konghq_writing`, dan `konghq_rtime`. Sedangkan `postgres-customhpa.yaml` akan mengatur *scaler* dari *pod* postgres dengan *metrics* CPU, *memory*, dan *external metric* `konghq_active`.

Pertama membuat berkas `kong-customhpa.yaml` dengan isi seperti berikut.

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: kong
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: kong
  minReplicas: 1
  maxReplicas: 20
  metrics:
  - type: Pods
    pods:
      metricName: konghq_active
      targetAverageValue: 50
  - type: Pods
```

```
      pods:
        metricName: konghq_writing
        targetAverageValue: 50
    - type: Pods
      pods:
        metricName: konghq_rtime
        targetAverageValue: 10
    - type: Resource
      resource:
        name: cpu
        targetAverageUtilization: 50
```

Kode Sumber IV.29: Berkas kong-customhpa.yaml

Berikutnya adalah membuat berkas `postgres-customhpa.yaml` dengan konfigurasi berikut.

```
  apiVersion: autoscaling/v2beta1
  kind: HorizontalPodAutoscaler
  metadata:
    name: postgres
    namespace: default
  spec:
    scaleTargetRef:
      apiVersion: apps/v1
      kind: StatefulSet
      name: postgres
    minReplicas: 1
    maxReplicas: 30
    metrics:
    - type: Resource
      resource:
        name: memory
        targetAverageValue: 100Mi
    - type: Resource
```

```
      resource:
        name: cpu
        targetAverageUtilization: 50
    - type: External
      external:
        metricName: konghq_active
        targetValue: 30
```

Kode Sumber IV.30: Berkas postgres-customhpa.yaml

Setelah semua berkas dibuat, berikutnya adalah memasang ketiga berkas tersebut ke kluster Kubernetes dengan perintah berikut.

```
$ kubectl -f apply kong-customhpa.yaml
$ kubectl -f apply postgres-customhpa.yaml
```

Kode Sumber IV.31: Memasang *scaler*

(Halaman ini sengaja dikosongkan)

BAB V

PENGUJIAN DAN EVALUASI

Pada bab ini akan dibahas uji coba dan evaluasi dari sistem yang sudah dibuat. Sistem akan diuji coba fungsionalitasnya dengan menjalankan skenario pengujian fitur-fitur dari sistem yang dibuat. Sistem juga akan diuji coba performa dengan skenario pengujian beban terhadap sistem. Uji coba dilakukan untuk mengevaluasi kinerja dari sistem dengan lingkungan uji coba yang ditentukan.

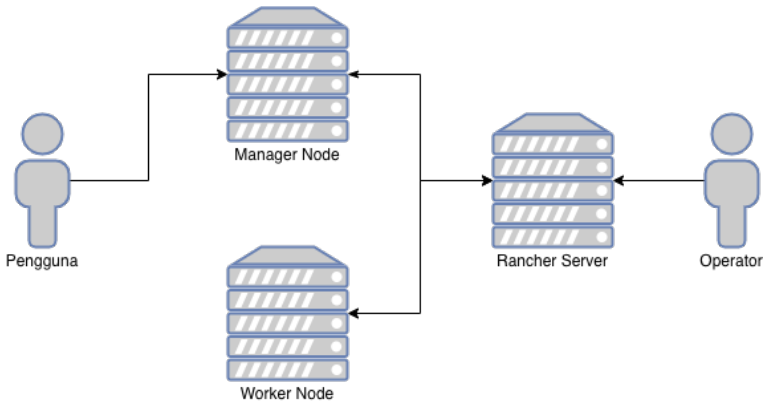
5.1 Lingkungan Uji Coba

Uji coba dilakukan dengan menggunakan total 3 *Virtual Private Server*. 1 *Virtual Private Server* bertindak sebagai penyedia rancher, 1 *Virtual Private Server* bertindak sebagai node manager sekaligus worker, dan 1 *Virtual Private Server* yang bertindak sebagai worker. Spesifikasi masing-masing *Virtual Private Server* dapat dilihat pada tabel berikut ini:

Tabel 5.1: Spesifikasi komponen

Komponen	Spesifikasi
Rancher *Server*	
CPU	Intel Xeon CPU E5-2690 v4 @ 2.60GHz (16 cores)
Sistem Operasi	Ubuntu 18.04.4 LTS 64 Bit
RAM	32 GB
Penyimpanan	50 GB
Alamat IP	10.199.2.32
***Manager* (*etcd*, *controlplane*, *worker*)**	
CPU	Intel Xeon Gold 5120 CPU @ 2.20GHz (16 cores)
Sistem Operasi	Ubuntu 18.04.4 LTS 64 Bit
RAM	16 GB

Penyimpanan	100 GB
Alamat IP	10.199.2.175
Worker	
CPU	Intel Xeon Gold 5120 CPU @ 2.20GHz (16 cores)
Sistem Operasi	Ubuntu 18.04.4 LTS 64 Bit
RAM	8 GB
Penyimpanan	100 GB
Alamat IP	10.199.2.176
Tester	
CPU	Intel Xeon CPU E5-2690 v4 @ 2.60GHz (4 cores)
Sistem Operasi	Ubuntu 18.04.3 LTS 64 Bit
RAM	8 GB
Penyimpanan	64 GB
Alamat IP	10.199.16.55



Gambar 5.1: Infrastruktur kluster

5.2 Skenario Uji Coba

Uji coba ini dilakukan untuk menguji fungsionalitas dari sistem yang dibuat telah sesuai dengan perancangan dan sistem benar-benar diimplementasikan dan bekerja sesuai seharusnya. Skenario pengujian dibedakan menjadi 2 yaitu:

- **Uji Fungsionalitas**
  Pengujian fungsionalitas bertujuan untuk menguji fungsi dari berbagai komponen berjalan sebagai mana mestinya sesuai dengan perancangan sistem.
- **Uji Performa**
  Pengujian performa bertujuan untuk menguji kemampuan skala otomatis pada kluster dan pengaruhnya terhadap kinerja kluster.

Uji coba akan dilakukan pada *Virtual Private Server* Tester dan pada *container* masing-masing layanan *server* untuk menguji kluster yang terdapat pada *Manager* dan *Worker*.

5.2.1 Skenario Uji Coba Fungsionalitas

Uji fungsionalitas akan difokuskan untuk menguji *sidecar pattern* dan menguji kemampuan skala otomatis dari kluster. Pengujian dilakukan untuk memastikan apakah *sidecar pattern* dan kemampuan skala otomatis sudah berjalan sebagai mana seharusnya. Jika kedua hal tersebut berjalan dengan sebagaimana mestinya, maka skala otomatis dengan menggunakan *custom metric* dan *sidecar pattern* akan dapat dikatakan berhasil sesuai dengan apa yang telah dirancangkan.

5.2.1.1 Uji Fungsionalitas *Sidecar Pattern*

Uji fungsionalitas *sidecar pattern* ini bertujuan untuk memastikan *container* pendukung pada *deployment* kong bisa berjalan dengan baik. Fungsi utama dari *sidecar* pada tugas akhir

ini adalah melakukan pengambilan data dari stub nginx dan menyediakannya kembali dalam format *time series* prometheus.

Uji coba ini dilakukan dengan memanfaatkan tingkah laku dari *sidecar pattern* sendiri yaitu *container* utama dan *container* pendukung saling berbagai sumber daya termasuk jaringan. Sehingga uji coba akan dilakukan dengan cara mengakses `http://127.0.0.1:8001/nginx status` pada *container pendukung*. Selain itu tugas *sidecar* adalah menyediakan kembali nilai pada stub nginx yang sudah diubah formatnya menjadi *time series* prometheus. Hal ini dapat diuji coba dengan cara mengakses `http://127.0.0.1:8888`.

Uji coba dikatakan berhasil jika *container* pendukung mampu mengakses halaman stub nginx dari *container* utama dan juga bisa menampilkannya kembali dalam format *time series* prometheus.

Tabel 5.2: Skenario uji fungsionalitas *sidecar pattern*

No	URL	*Method*	Harapan
1	`http://127.0.0.1:8001/nginx status`	GET	Sistem dapat menampilkan halaman stub nginx dari container utama.
2	`http://127.0.0.1:8888`	GET	Sistem dapat menampilkan metric dalam format time series prometheus.

5.2.1.2 Uji Fungsionalitas Skala Otomatis

Uji coba ini bertujuan untuk memastikan kemampuan skala otomatis berjalan dengan seharusnya. Pada perancangan skala otomatis terdapat 6 komponen pembentuk. Keenam komponen tersebut adalah *custom exporter*, *application*, *kubelet exporter*,

collector, *connector*, dan *scaler*. Komponen-komponen tersebut harus berjalan sesuai dengan fungsinya masing-masing agar kemampuan skala otomatis bisa dikatakan berjalana baik.

Uji coba dilakukan dengan menguji masing-masing keenam komponen tersebut. *Custom exporter* akan diuji dengan mengakses halaman *custom metric* melalui alamat IP dari *deployment* kong yang sudah ter-*load balancing* di `http://10.199.2.175:8888`. *Application* akan diuji coba dengan mengakses admin API dari kong melalui alamat IP dari *deployment* kong yang juga sudah ter-*load balancing* di `http://10.199.2.175:8001`. *Kubelet exporter* akan diuji dengan menajalankan 2 perintah berikut.

```
$ kubectl top pods --containers=true
$ kubectl top node
```

Kode Sumber V.1: Menguji *kubelet exporter*

Pengujian *collector* akan dilakukan dengan mengakses alamat url dari layanan prometheus `http://10.199.2.175:3773` dan memasukan nama *metric* `konghq` pada *search bar*. Untuk melakukan uji coba *connector* bisa dilakukan dengan cara menjalankan perintah `kubectl get --raw` dan jika menggunakan rancher perlu menyesuaikan kode kluster sesuai kode yang diberikan oleh rancher. Pada kasus tugas akhir ini kode kluster adalah `c-qntkf`. Keluaran yang muncul adalah *metric* dengan nama `konghq_active`, `konghq_writing`, dan `konghq_rtime`.

```
$ kubectl get --raw /k8s/clusters/c-qntkf/apis/
    custom.metrics.k8s.io/v1beta1 #untuk rancher
$ kubectl get --raw /apis/custom.metrics.k8s.io
    /v1beta1 #untuk minikube
```

Kode Sumber V.2: Mengecek *connector*

Terakhir adalah pengujian *scaler*. Pengujian *scaler* bisa dilakukan dengan melakukan request ke admin API milik kong untuk men-*trigger* nilai aktif koneksi, *writing connection*, dan *request time* pada *metric*. Menggunakan rumus berikut dalam algoritme skala, jika dengan batas 50 pada aktif koneksi yang sudah di *deploy* maka seharusnya jika melakukan 100 *requests* dengan nilai aktif koneksi mencapai lebih dari 50 akan terbuat 1 *pod* tambahan sehingga total ada 2 *pods* kong.

```
desRep = ceil[curRep*(curValue/desValue)]
```

Pengujian bisa dikatakan berhasil jika *custom exporter* berhasil menampilkan *metric* dalam bentuk *time series* prometheus ketika diakses, *application* berhasil menampilkan JSON layanan kong, *kubelet exporter* berhasil menampilkan status memori dan CPU dari *node* maupun *pod*, *collector* berhasil mengumpulkan data *metric* dari berbagai *pod* kong, *connector* mampu menampilkan *metric* `konghq_active`, `konghq_writing`, dan `konghq_rtime` dalam HPA controller, dan *scaler* berhasil melakukan skala otomatis baik skala atas maupun skala bawah.

Tabel 5.3: Skenario uji fungsionalitas skala otomatis

No	Komponen	Harapan
1	*Custom exporter*	Sistem dapat menampilkan *metric* dalam bentuk *time series* prometheus.
2	*Application*	Sistem dapat menampilkan JSON dari layanan kong.
3	*Kubelet Exporter*	Sistem dapat menampilkan status memori dan CPU.
4	*Collector*	Prometheus mampu mengumpulkan data *metric* dari berbagai *pods*.
5	*Connector*	HPA controller mampu membaca *metric* dari prometheus.

6	*Scaler*	HPA controller dapat melakukan skala otomatis berdasar *metric* yang ditentukan.

5.2.2 Skenario Uji Coba Performa

Uji performa bertujuan untuk menguji kemampuan skala otomatis pada kluster dan pengaruhnya terhadap kinerja kluster. Pada uji performa akan diuji juga batas kemampuan dari satu *pod* kong dalam menangani permintaan pengguna. Selain itu akan diuji juga mengenai kecepatan *deploy pod* baru ketika melakukan skala atas otomatis dengan mempertimbangkan juga jumlah kegagalan dalam pembuatan *pod*.

Uji performa akan dilakukan dengan cara melakukan *requests* terhadap url admin API kong yang sudah ter-*load balancing* di `http://10.199.2.175:8001` secara *concurrent* selama 120 detik lalu diberi waktu kosong 60 menit untuk melihat perubahan jumlah *pod* dan nantinya dilanjutkan kembali di uji selama 120 detik. Jadi total untuk 1 variasi nilai *concurrent* akan memakan kurang lebih 5 menit. Jumlah variasi *concurrent* akan dibuat berbeda-beda dengan variasi nilai 10, 50, 100, 150, dan 200 *requests*. Setelah itu akan dilihat berapa perubahan jumlah *pod*, keberhasilan dalam menangani *requests* dan *requests per second*.

Untuk uji coba batas kemampuan dari satu *pod* akan dicari nilai *concurrent* di mana jumlah kegagalan lebih dari 10% dari total *requests*. 10% ini didasarkan pada pengujian performa pada aplikasi awan oleh Sen He di mana Sen He menggunakan nilai 90% pada probabilitas yang ingin dicapai pada setiap konfigurasi[15]. Pengujian dilakukan dengan cara mengakses admin API kong pada url `http://10.199.2.175:8001` dan secara bertahap tiap 60 detik dinaikan *concurrent requests*-nya sejumlah 10 hingga akhirnya ditemukan jumlah kegagalan lebih

dari 10% dari total *requests*.

Selain melakukan request ke admin API kong. Pengujian juga akan dilakukan untuk melihat kemampuan kluster ketika memanggil service mock yang berada di alamat url `http://10.199.2.175:8000/mock`. Pengujian kan dilakukan secara bertahap dimulai dari 1000 *concurrent requests*, kemudian tiap 60 detik dinaikan *concurrent requests*-nya sejumlah 100, hingga nantinya ditemukan jumlah kegagalan lebih dari 10% dari total *requests*.

Terakhir terkait pengujian kecepatan *deploy pod* dan jumlah kegagalannya dalam pembuatan akan dilakukan dengan melakukan *concurrent requests* sejumlah 200 *requests* selama 60 detik lalu diamati status dari *pod* tersebut.

Hasil yang diharapkan dari pengujian performa adalah kluster masih mampu menangani *requests* dengan tingkat keberhasilan menangani *requests* dan nilai *requests per second* semakin tinggi sejajar dengan peningkatan variasi nilai *concurrent requests*. Diharapkan juga bahwa performa satu *pod* kong bisa mengolah *requests* dengan baik sebanyak mungkin. Juga kluster memiliki kecepatan yang tinggi dalam menambah *pod* ketika skala atas dan memiliki nilai kegagalan yang rendah dalam menambah *pod* ketika skala atas. Uji performa yang berikutnya adalah melihat perubahan dari total *requests*, keberhasilan *requests*, nilai *requests per seconds*, dan jumlah perubahan *pod* pada saat melakukan uji coba mengirim *concurrent requests* dengan nilai 1000, 2000, 3000, 4000, dan 5000 menuju url `http://10.199.2.175:8000/mock`. Tiap nilai *concurrency* akan dilakukan selama 2 menit lalu berhenti selama 60 detik dan kemudian dilanjutkan kembali 2 menit. Pembagian tersebut didasarkan pada 60 detik pertama adalah awal untuk memicu skala otomatis oleh *scaler*, lalu 60 detik berikutnya untuk memastikan pods dalam keadaan runnning dan 60 detik diberhentikan agar pods bisa masuk ke dalam *service*,

lalu 120 detik terakhir dilakukan pengujian dengan jumlah pod yang sudah diperbaharui dan agar bisa dibandingkan dengan 120 detik pertama. Khusus untuk uji coba ini akan diterapkan nilai batas *metric* yang berbeda karena url ini adalah url *production* sehingga perlu dilakukan beberapa penyesuaian seperti *metric* `konghq_active` memiliki batas 1000, `konghq_writing` adalah 500, CPU dinaikan menjadi 60%, dan batas *pod* yang bisa terbuat pada kong adalah 27 *pods*. Terakhir adalah menguji kemampuan maksimal ketika 25 *pods* kong menyala. Pada VPS untuk melakukan uji coba, hanya mampu mengirim 28000 *requests*. Di atas nilai tersebut akan menyebabkan masalah mengenai *socket* telah habis digunakan. Pengujian akan dilakukan dengan cara melakukan *requests* ke url `http://10.199.2.175:8000/mock` dengan keadaan jumlah *pod* kong adalah 25. Data yang akan diambil adalah jumlah total *requests*, jumlah *requests* berhasil, jumlah *requests* gagal, dan nilai *requests* per detik.

5.3 Hasil Uji Coba dan Evaluasi

Berikut ini dijelaskan hasil coba dan evaluasi berdasarkan skenario yang sudah dijelaskan pada subbab sebelumnya.

5.3.1 Uji Fungsionalitas

Berikut ini dijelaskan hasil dari pengujian fungsionalitas pada sistem yang sudah dibangun.

5.3.1.1 Uji Fungsionalitas *Sidecar Pattern*

Uji coba ini dilakukan dengan cara mengakses stub nginx di `http://127.0.0.1:8001/nginx_status` dari *container* utama yaitu kong melalui *container* pendukung dengan memanfaatkan tingkah laku dari *sidecar pattern* yaitu saling

berbagi sumber daya salah satunya jaringan. Selain itu *container* pendukung juga turut diuji fungsinya yaitu sebagai penyedia *custom metric yang sudah* di ubah formatnya menjadi *time series* prometheus di `http://127.0.0.1:8888/`.

Tabel 5.4: Hasil uji fungsionalitas *sidecar pattern*

No	URL	*Method*	Harapan	Hasil
1	`http://127.0.0.1:8001/nginx_status`	GET	Sistem dapat menampilkan halaman stub nginx dari *container* utama.	OK
2	`http://127.0.0.1:8888`	GET	Sistem dapat menampilkan *metric* dalam format *time series* prometheus.	OK

5.3.1.2 Uji Fungsionalitas Skala Otomatis

Uji coba dilakukan dengan cara menguji satu persatu dari keenam komponen yang ada dan memastikan keenam komponen tersebut berjalan sebagai mana seharusnya sesuai yang telah dirancang.

Komponen *custom exporter* diuji dengan cara mengakses url custom metric melalui alamat IP dari *deployment* kong yang telah ter-*load balancing* di `http://10.199.2.175:8888`. *Application* akan diuji dengan mengakses admin API dari kong melalui alamat IP yang telah ter-*load balancing* di `http://10.199.2.175:8001`. Kubelet *exporter* akan diuji dengan melihat status penggunaan memori dan CPU dari *node* dan *pod*. Pengujian *collector* dilakukan dengan cara mengakses url dari layanan prometheus dan mencari *metric* `konghq` di

alamat url `http://10.199.2.175:3773`. *Connector* diuji dengan cara mengecek API dari *custom metric* kubernetes. Jika berhasil ditemukan `konghq_active`, `konghq_writing`, dan `konghq_rtime` maka *connector* berfungsi dengan baik. *Scaler* akan diuji dengan cara mencoba melakukan *requests* secara paralel untuk men-*trigger* HPA controller agar kluster melakukan skala atas otomatis.

Tabel 5.5: Hasil uji fungsionalitas skala otomatis

No	Komponen	Harapan	Hasil
1	*Custom exporter*	Sistem dapat menampilkan *metric* dalam bentuk *time series* prometheus.	OK
2	*Application*	Sistem dapat menampilkan JSON dari layanan kong.	OK
3	*Kubelet Exporter*	Sistem dapat menampilkan status memori dan CPU.	OK
4	*Collector*	Prometheus mampu mengumpulkan data *metric* dari berbagai *pods*.	OK
5	*Connector*	HPA controller mampu membaca *metric* dari prometheus.	OK
6	*Scaler*	HPA controller dapat melakukan skala otomatis berdasar *metric* yang ditentukan.	OK

5.3.2 Hasil Uji Performa

Uji performa dilakukan pada untuk mencari kemampuan dari skala otomatis pada kluster. Pada pengujian ini ditemukan kemampuan dari satu *pod* kong terkait batas *pod* tersebut dalam menerima *requests*. Selain batas *requests* dari satu *pod*, juga diuji

kemampuan skala otomatis dari kluster terkait kecepatan dalam proses *build* dan *deploy pod*. Terakhir dari pengujian skala otomatis juga telah ditemukan jumlah *pod* yang terbuat dari skala otomatis, total *requests*, jumlah *requests* yang berhasilan, dan tingkat *requests* per detik dari berbagai variasi nilai *concurrency requests* yang diuji selama kurang lebih 300 detik.

Pengujian batas *requests* dari satu *pod* kong adalah dengan cara mengirim *requests* ke `http://10.199.2.175:8001` secara paralel selama 60 detik dengan kemampuan skala otomatis dimatikan. Dimulai dari tingkat *concurrency* 10 dan terus bertambah sebanyak 10 hingga akhirnya ditemukan di mana tingkat kegagalan menangani *requests* adalah 10% dari total *requests* yang ada.

Tabel 5.6: Hasil uji batas beban *pod* kong pada admin API

No	Tingkat *Concurrent*	*Total Requests*	*Failed Requests*	Persentase
1	10	13929	0	0,00%
2	20	15185	0	0,00%
3	30	14797	0	0,00%
4	40	15354	4	0,03%
5	50	16223	83	0,51%
6	60	15982	239	1,50%
7	70	16153	387	2,40%
8	80	16032	754	4,70%
9	90	16222	1325	8,17%
10	100	15558	1714	11,02%

Pengujian batas *requests* dari satu *pod* kong juga dilakukan terhadap request service mock yang berada di url `http://10.199.2.175:8000/mock`. Pengujian dilakukan secara paralel mulai dari tingkat *concurrency* 1001 selama 60

detik, dan akan bertambah 100 untuk pengujian berikutnya hingga akhirnya ditemukan di mana tingkat kegagalan menangani *requests* adalah 10% dari total *requests* yang ada.

Tabel 5.7: Hasil uji batas beban *pod* kong pada test *service*

No	Tingkat *Concurrent*	*Total Requests*	*Failed Requests*	Persentase
1	1000	2711	57	1,10%
2	1100	2675	149	5,57%
3	1200	2764	86	3,21%
4	1300	2749	180	6,55%
5	1400	2674	245	9,16%
6	1500	2656	263	9,90%
7	1600	3071	437	14,23%

Pada tabel 5.6 dapat dilihat bahwa semakin tinggi tingkat *concurrency*-nya maka semakin besar juga persentase kegagalannya. Pada saat tingkat *concurrency* berada pada nilai 100, *pod* kong sudah memiliki persentase tingkat kegagalan 11,02% dari total *requests*. Sedangkan pada tabel 5.7 kemampuan batas dari pod kong ketika mengakses halaman url `http://10.199.2.175:8000/mock` adalah berada pada tingkat *concurrency* 1600 dengan persentase kegagalannya 14,23% dari total *requests*.

Pengujian berikutnya adalah terkait dengan kecepatan dalam proses *build* dan *deploy* pada saat melakukan skala otomatis. Pengujian dilakukan dengan cara melakukan *requests* secara paralel hingga nantinya *event trigger* dari skala otomatis berjalan dan melakukan skala atas pada *pod* kong. Berdasarkan pengamatan proses *build* dan *deploy* berjalan dengan sangat cepat dan membutuhkan waktu kurang dari sama dengan 1 detik pada *development*. Namun jika pada *production* perlu sekitar 1

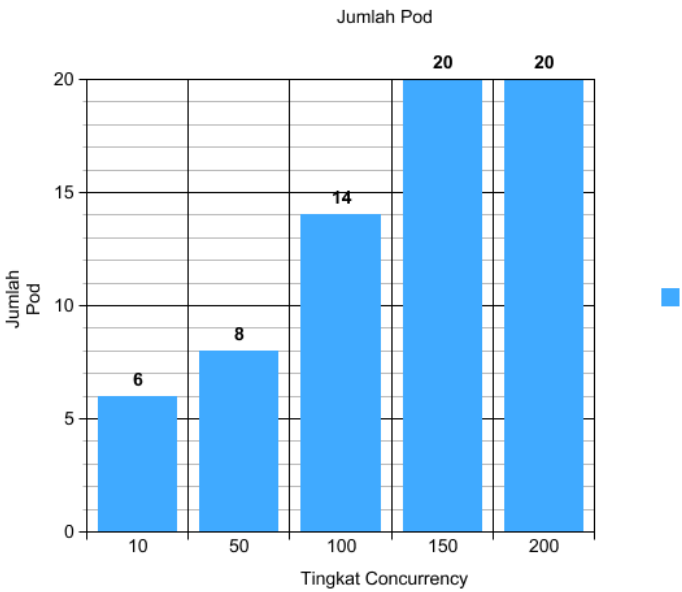
menit. Tingkat kegagalan pada *development* adalah tidak ditemukan kegagalan sama sekali. Namun pada *production* adalah antara 0 sampai 2 kali *pod* harus melakukan *restart* hingga statusnya aktif. Namun hal ini bisa terjadi jika sebelumnya pada *node* telah melakukan *pre-pulling image* kong versi yang saat ini digunakan. Jika belum ada *image* pada penyimpanan lokal *node* maka kubernetes akan mengunduh *image* kong terlebih dahulu sebelum melakukan *deploy* layanan. Jika hal ini terjadi maka waktu *deploy* akan menyesuaikan kecepatan unduh *image*, namun setelah *image* diunduh proses *build* dan *deploy* akan membutuhkan waktu kurang dari sama dengan 1 detik.

```
86s     Created pod: kong-66f576dc74-mkpnz
86s     Scaled up replica set kong-66f576dc74 to 2
```

Gambar 5.2: Potongan *event* kluster kubernetes

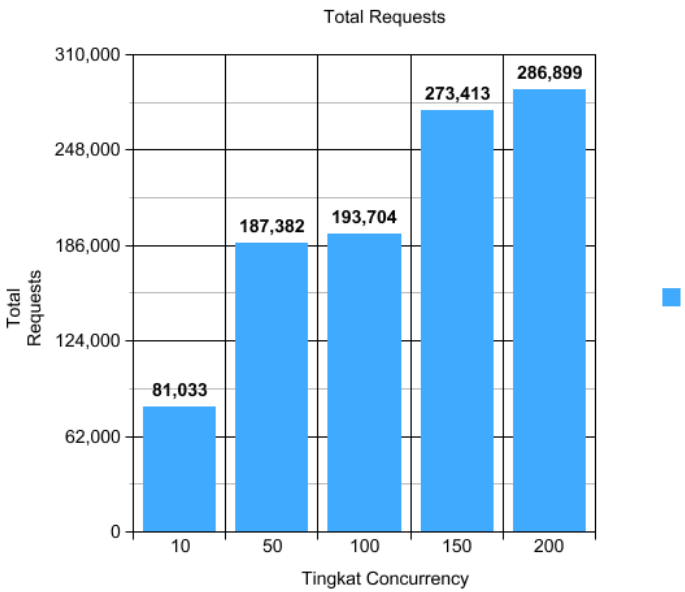
Gambar 5.2 merupakan potongan dari daftar *event* yang terjadi di kluster kubernetes. Terlihat pada gambar tersebut bahwa HPA controller terpanggil untuk melakukan skala atas pada 86 detik yang lalu, dan dalam waktu yang sama *pod* kong berhasil dibuat juga pada 86 detik yang lalu.

Pengujian berikutnya adalah terkait dengan performa dari kemampuan skala atas pada kluster. Pengujian dilakukan dengan cara melakukan *requests* menuju admin API kong secara paralel dengan tingkat *concurrency* 10, 50, 100, 150, dan 200. Masing-masing tingkat *concurrency* dijalankan selama 120 detik lalu diberi waktu kosong selama 1 menit lalu dilanjutkan lagi selama 120 detik, sehingga untuk satu tingkat *concurrency* akan dijalankan selama total kurang lebih 300 detik. Hasil yang didapat pada pengujian ini adalah jumlah *pod* dari skala otomatis, total *requests*, *requests* yang berhasil dilakukan, dan tingkat *requests* per detik dari kelima variasi *concurrency*.



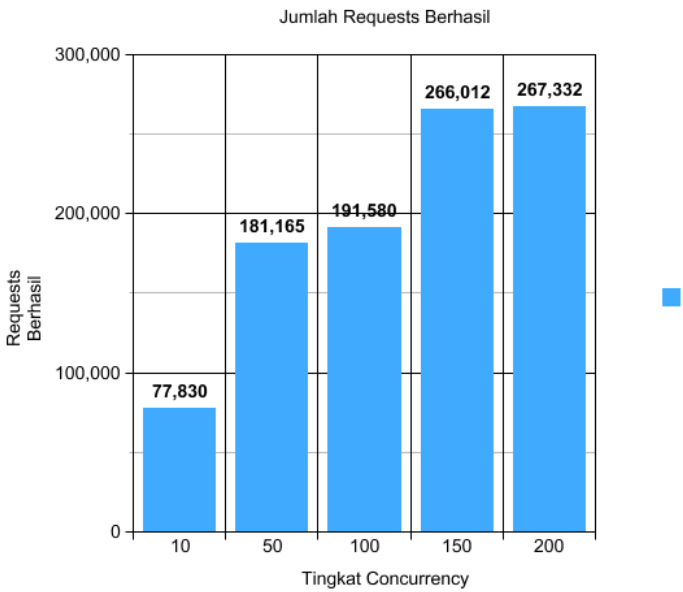
Gambar 5.3: Jumlah *pod* dari berbagai tingkat *concurrency*

Dari grafik 5.3 dapat diamati bahwa semakin besar beban *requests* maka semakin tinggi juga jumlah *pod* yang dibuat dari hasil skala otomatis. Skala otomatis terjadi karena *metric* CPU, jumlah koneksi aktif, jumlah *writing connection*, dan *request time* dari kong melebihi batas yang ada. Untuk tingkat 150 dan 200 memiliki jumlah *pod* yang sama dikarenakan dari implementasi jumlah *pod* yang bisa terbuat dari skala otomatis dibatasi hingga 20 *pods* kong.



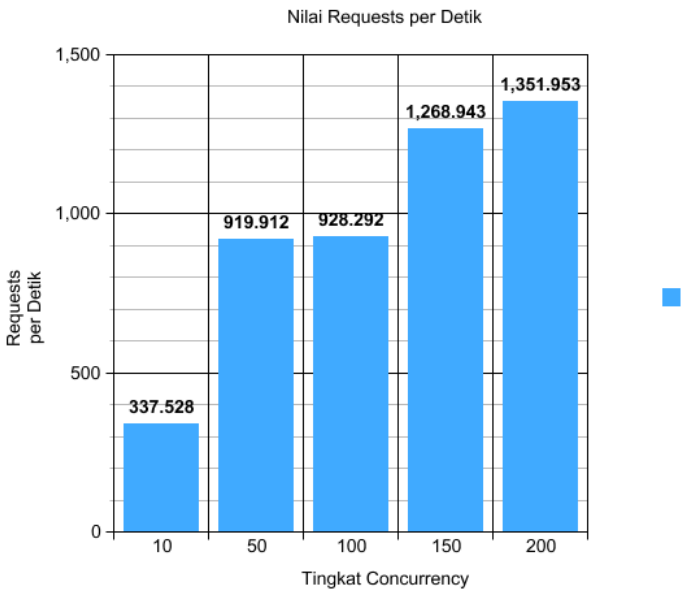
Gambar 5.4: Total *requests* dari berbagai tingkat *concurrency*

Dari grafik 5.4 dapat dilihat bahwa nilai total *requests* yang terjadi terus bertambah sejajar dengan tingkat *concurrency*. Hal ini terjadi karena bertambahnya jumlah *pod* membuat *requests* bisa terbagi ke *pods* yang ada sehingga pemrosesan *requests* bisa berjalan lebih cepat. Selain itu, tingkat *concurrency* juga mempengaruhi berapa jumlah yang dikirim per detiknya sehingga semakin tinggi tingkat *concurrency* maka semakin tinggi juga jumlah *requests* per detiknya.



Gambar 5.5: Jumlah *requests* berhasil dari berbagai tingkat *concurrency*

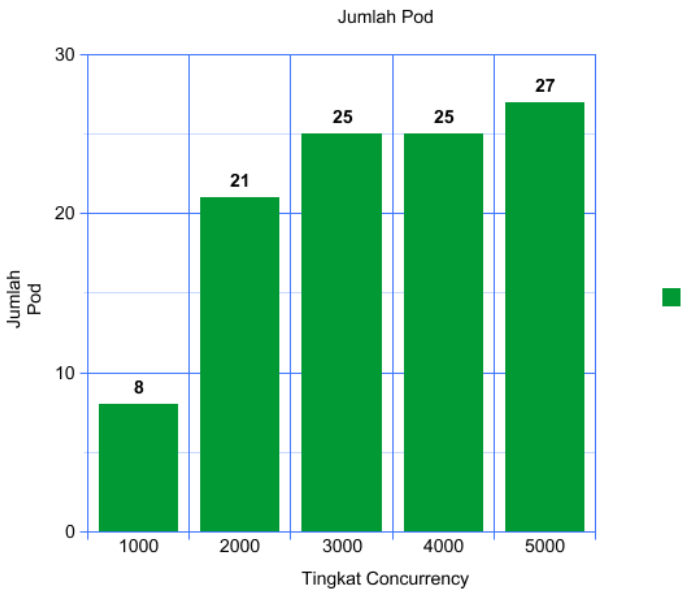
Dari grafik 5.5 jumlah keberhasilan *requests* semakin meningkat sejajar dengan jumlah *concurrency*. Banyak faktor yang mempengaruhi keberhasilan suatu *request*. Kemampuan postgres dalam melayani koneksi dari layanan kong juga salah satu hal yang menentukan. Beberapa kali koneksi antara postgres dan kong gagal karena limit dari jumlah koneksi yang bisa terhubung ke postgres. Selain itu juga memori dan CPU pada pod kong juga menentukan keberhasilan dari *request*. Bisa terjadi jika CPU dan memori sudah melewati batas yang ada, maka layanan kong akan melakukan *restart* dan membuat *request* gagal. Oleh sebab itu dengan bertambahnya jumlah *pod* pada setiap tingkat *concurrency* membuat beban dari *pod* kong semakin berkurang sehingga jumlah keberhasilannya semakin meningkat.



Gambar 5.6: Nilai *requests* per detik dari berbagai tingkat *concurrency*

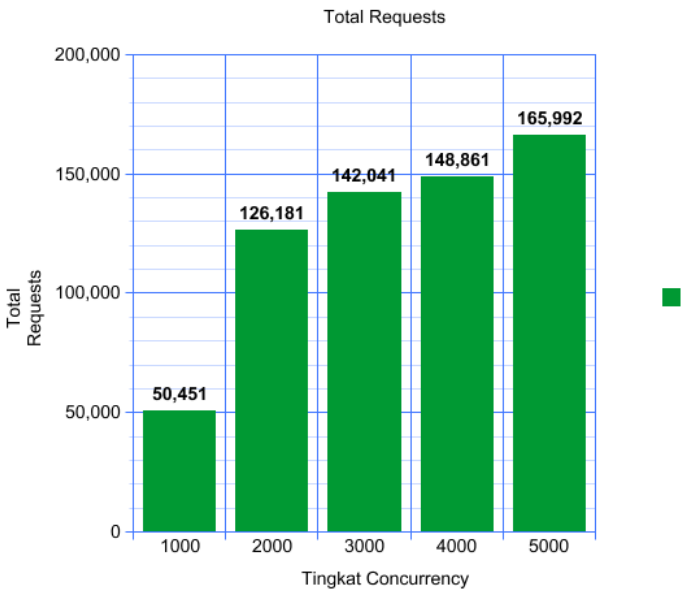
Grafik 5.6 menunjukan juga bahwa semakin tinggi tingkat *concurrency* maka semakin tinggi juga nilai dari *requests* per detik. Hal ini dikarenakan jumlah *pod* yang semakin meningkat membuat kemampuan pemrosesan *requests* bisa berjalan dengan lebih cepat.

Berikutnya adalah hasil uji coba dari pengiriman concurrent requests dengan nilai 1000, 2000, 3000, 4000, dan 5000 menuju url `http://10.199.2.175:8000/mock` untuk mendapatkan perubahan dari total *requests*, keberhasilan *requests*, nilai *requests per seconds*, dan jumlah perubahan *pod*.



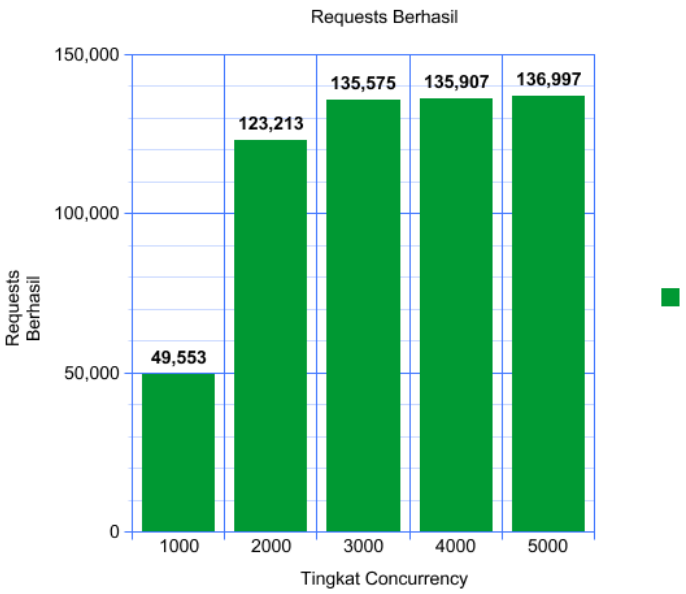
Gambar 5.7: Jumlah *pod* dari percobaan terhadap `http://10.199.2.175:8000/mock`

Dari grafik 5.7 terlihat terjadi penambahan jumlah *pod* sebanding dengan kenaikan tingkat *concurrency* dari mengakses url `http://10.199.2.175.8000/mock`. Hal ini terjadi karena *pod* mengalami beban yang lebih besar ketika diakses oleh lebih banyak *requests*.



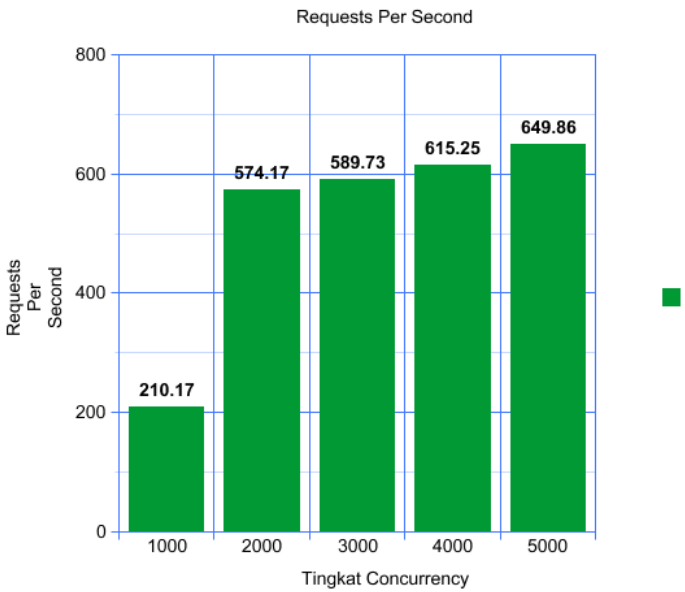
Gambar 5.8: Total *requests* dari percobaan terhadap `http://10.199.2.175:8000/mock`

Pada grafik 5.8 dapat diamati bahwa nilai total *requests* akan terus meningkat sejajar dengan penambaan nilai *concurrency*. Hal ini juga disebabkan karena tingkat *concurrency* lebih tinggi artinya lebih banyak *request* yang dikirim ke dalam *pod*. Namun hal ini juga diimbangi dengan penambahan *pod* pada grafik 5.7 maka pengolahan *requests* akan bisa dilakukan lebih cepat karena terbagi bebannya.



Gambar 5.9: Jumlah *requests* berhasil dari percobaan terhadap `http://10.199.2.175:8000/mock`

Grafik 5.9 memperlihatkan bahwa jumlah keberhasilan *requests* paling banyak berada pada nilai *concurrency* 5000. Hal ini dikarenakan lebih banyak *pod* yang bisa mengolah *requests* sehingga bisa lebih banyak *request* yang berhasil.



Gambar 5.10: Nilai *requests* per detik dari percobaan terhadap `http://10.199.2.175:8000/mock`

Dari grafik 5.10 dapat dilihat bahwa nilai *requests* per detik dari berbagai tingkat *concurrency*, menunjukan bahwa semakin tinggi tingkat concurrency nya dengan diiringi penambahan *pod* dari *autoscale*, akan meningkatkan nilai *requests* per detik yang ada.

Berikut adalah hasil percobaan untuk mendapatkan nilai total *requests*, jumlah *requests* berhasil, jumlah *requests* gagal, nilai *requests* per detik dari tingkat *concurrency* maksimal yang bisa dilakukan pada VPS uji coba yaitu 28000 *requests*. Pengujian akan dilakukan dengan mengakses *upstream service* mock di alamat url `http://10.199.2.175:8000/mock`.

Tabel 5.8: Hasil percobaan pada keadaan maksimal

No	Nama	Nilai
1	Total *requests*	32314
2	Jumlah *requests* berhasil	20607
3	Jumlah *requests* gagal	11707
4	Nilai *requests* per detik	264,705

(Halaman ini sengaja dikosongkan)

BAB VI

PENUTUP

Bab ini membahas kesimpulan yang dapat diambil dari tujuan pembuatan sistem dan hubungannya dengan hasil uji coba dan evaluasi yang telah dilakukan. Selain itu, terdapat beberapa saran yang bisa dijadikan acuan untuk melakukan pengembangan dan penelitian lebih lanjut.

6.1 Kesimpulan

Dari proses perencangan, implementasi dan pengujian terhadap sistem, dapat diambil beberapa kesimpulan berikut:

1. Sistem dapat melakukan skala otomatis sesuai dengan *metric* yang telah ditentukan.
2. *Sidecar pattern* bekerja sesuai dengan fungsinya yaitu mendapatkan data dari *container* utama dan menyediakannya kembali ke dalam bentuk *time series prometheus*.
3. *Custom metric* yang telah dibuat dapat terbaca oleh HPA controller untuk nantinya digunakan sebagai salah satu parameter untuk melakukan skala otomatis.
4. Algoritma skala berjalan sebagaimana seharusnya sehingga kluster bisa melakukan skala otomatis berdasarkan koneksi aktif dan besar CPU dari layanan kong.
5. Skala otomatis mampu meningkatkan kinerja dari kluster sehingga kemampuan kluster dalam menangani permintaan dari user bisa diselesaikan lebih cepat.

6.2 Saran

Berikut beberapa saran yang diberikan untuk pengembangan lebih lanjut:

1. Kemampuan skala otomatis pada tugas akhir ini lebih difokuskan pada tipe *deployment* yang *stateless* sehingga

ketika dihadapkan pada tipe *deployment* yang setiap *pod* memiliki peran seperti *master* atau *slave*, masih belum ada implementasi lebih lanjutnya.

2. Proses pembaharuan *image* kong yang baru perlu disertai *pre-pulling* secara manual di setiap *node* agar *image* kong baru berada di penyimpanan lokal setiap *node*. Jika tidak ketika saat skala otomatis sudah berjalan, peluang untuk mengalami kegagalan proses *build* dan *deploy pod* baru akan semakin besar karena kegagalan unduh *image* kong yang baru tersebut.
3. Sistem dapat dikembangkan dengan membuat sistem *monitoring* yang lebih handal terutama untuk melakukan uji coba terhadap kemampuan dari *single pod* layanan kong. Hal ini berguna untuk menentukan nilai batas dari suatu *metric* karena ketika melakukan pembaharuan *image* kong baru maka perlu disesuaikan kembali nilai batas *metric*-nya agar lebih tepat sasaran dalam melakukan skala otomatis.

DAFTAR PUSTAKA

[1] N. Jain dan S. Choudhary, "Overview of virtualization in cloud computing," in *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*. Indore, Madhya Pradesh, India: IEEE, Mar. 2016, hal. 1–4, 7 Juni 2020. [Daring]. Tersedia pada: http://ieeexplore.ieee.org/document/7570950/. [Diakses: 7 Juni 2020].

[2] "Microservices Pattern: API gateway pattern," 20 April 2020. [Daring]. Tersedia pada: http://microservices.io/patterns/apigateway.html. [Diakses: 20 April 2020].

[3] "What is an API Gateway? | NGINX Learning," 20 April 2020. [Daring]. Tersedia pada: https://www.nginx.com/learn/api-gateway/. [Diakses: 20 April 2020].

[4] S. Bhowmik, *Cloud computing*. Cambridge, United Kingdom: Cambridge University Press, 2017.

[5] "Docker overview," 18 April 2020. [Daring]. Tersedia pada: https://docs.docker.com/get-started/overview/. [Diakses: 18 April 2020].

[6] KongHQ, "Open-Source API Management and Microservice Management," 23 April 2020. [Daring]. Tersedia pada: https://docs.konghq.com. [Diakses: 23 April 2020].

[7] "Kubernetes Components," 19 April 2020. [Daring]. Tersedia pada: https://kubernetes.io/docs/concepts/overview/components/. [Diakses: 19 April 2020].

[8] "Pod Overview," 19 April 2020. [Daring]. Tersedia pada: https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/. [Diakses: 19 April 2020].

[9] "Overview | Prometheus," 23 April 2020. [Daring]. Tersedia pada: https://prometheus.io/docs/introduction/overview/. [Diakses: 23 April 2020].

[10] "General Python FAQ — Python 3.8.2 documentation," 2020, 23 April 2020. [Daring]. Tersedia pada: https://docs.python.org/3/faq/general.html#what-is-python. [Diakses: 23 April 2020].

[11] "Documentation," 15 Mei 2020. [Daring]. Tersedia pada: https://rancher.com/docs/. [Diakses: 15 Mei 2020].

[12] "Rancher 2.x," 15 Mei 2020. [Daring]. Tersedia pada: https://rancher.com/docs/rancher/v2.x/en/. [Diakses: 15 Mei 2020].

[13] dragon119, "Sidecar pattern - Cloud Design Patterns," 20 April 2020. [Daring]. Tersedia pada: https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar. [Diakses: 20 April 2020].

[14] "Horizontal Pod Autoscaler," 29 April 2020. [Daring]. Tersedia pada: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. [Diakses: 29 April 2020].

[15] S. He, G. Manns, J. Saunders, W. Wang, L. Pollock, dan M. L. Soffa, "A statistics-based performance testing methodology for cloud applications," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*. Tallinn, Estonia: ACM Press, 2019, hal. 188–199, 14 Juli 2020. [Daring]. Tersedia pada: http://dl.acm.org/citation.cfm?doid=3338906.3338912. [Diakses: 14 Juli 2020].

LAMPIRAN A

INSTALASI PERANGKAT LUNAK

Instalasi Docker

Langkah awal pada instalasi docker adalah memasang beberapa aplikasi yang mendukung untuk penggunaan `apt` melewati protokol HTTPS. Perintah untuk melakukan hal tersebut adalah sebagai berikut.

```
$ sudo apt install apt-transport-https ca-
    certificates curl software-properties-common
```

Kode Sumber A.1: Instalasi kebutuhan docker

Selanjutnya menambahkan kunci GPG dari halaman resmi docker ke dalam sistem.

```
$ curl -fsSL https://download.docker.com/linux/
    ubuntu/gpg | sudo apt-key add -
```

Kode Sumber A.2: Menambah kunci GPG

Setelah menambah kunci GPG, ditambahkan juga repository APT dari docker dan dilanjutkan dengan memperharui daftar aplikasi.

```
$ sudo add-apt-repository "deb [arch=amd64]
    https://download.docker.com/linux/ubuntu
    bionic stable"
$ sudo apt update
```

Kode Sumber A.3: Menambah repository docker

Berikutnya adalah instalasi aplikasi docker dengan menggunakan `apt`.

```
$ sudo apt install docker-ce
```

Kode Sumber A.4: Instalasi aplikasi docker

Pastikan status daemon docker dalam keadaan menyala dan berjalan dengan menggunakan perintah berikut. Hasilnya kurang lebih sama seperti dibawah.

```
$ sudo systemctl status docker
docker.service - Docker Application Container
    Engine
Loaded: loaded (/lib/systemd/system/docker.
    service; enabled; vendor preset: enabled)
Active: active (running) since Thu 2018-07-05 1
    5:08:39 UTC; 2min 55s ago
Docs: https://docs.docker.com
Main PID: 10096 (dockerd)
Tasks: 16
CGroup: /system.slice/docker.service├─
  10096 /usr/bin/dockerd -H fd://└─
  10113 docker-containerd --config /var/run/
      docker/containerd/containerd.toml
```

Kode Sumber A.5: Mengecek status docker

Instalasi Rancher Server

Untuk menginstall rancher server pastikan docker sudah melakukan instalasi docker terlebih dahulu. Untuk membuat service rancher bisa dilakukan dengan perintah.

```
$ sudo docker run -d --restart=unless-stopped -
    p 80:80 -p 443:443 rancher/rancher
```

Kode Sumber A.6: Menjalankan rancher

Untuk memastikan bahwa rancher sudah terpasang bisa dijalankan perintah berikut dan melihat hasilnya apakah ada container dengan image rancher/rancher pada output.

```
$ docker ps --filter status=running --format "
    table {{.ID}}\t{{.Image}}"
CONTAINER ID        IMAGE
b6b7a9da1108        rancher/rancher
```

Kode Sumber A.7: Memastikan rancher berjalan

Setelah itu buka browser dan coba akses rancher web UI dengan memasukan IP address `https://127.0.0.1`. Jika berhasil dibuka maka akan diberikan form untuk memasukan password yang nantinya digunakan untuk mengakses rancher web UI. Setelah membuat password akan diberikan form lanjutan untuk memasukan URL. URL yang dimasukan adalah URL yang digunakan untuk mengakses rancher server.

Setelah itu langkah selanjutnya adalah membuat cluster. Cluster dibuat dengan menekan tombol `Add Cluster` yang ada dihalaman web. Setelah itu akan muncul halaman untuk memilih cluster type. Dikarenakan cluster yang nantinya dibuat berupa local cluster maka tipe yang digunakan adalah `From existing nodes (custom)`. Setelah menekan pilihan tipe akan muncul halaman untuk mengatur cluster. Masukan cluster name untuk memberi nama cluster yang akan dibuat. Pada bagian node port range bisa diatur range port yang digunakan pada cluster. Sesuaikan range port yang digunakan dengan kebutuhan yang ada. Jika sudah klik tombol `Next`.

Berikutnya adalah mendaftarkan node yang ada ke dalam cluster sesuai dengan role masing-masing. Pada tugas akhir ini virtual private server api-gw-2-175 digunakan sebagai etcd, controlplane, dan worker sehingga di terminal pada virtual private server api-gw-2-175 dijalankan perintah berikut.

```
$ sudo docker run -d --privileged --restart=
    unless-stopped --net=host -v /etc/
    kubernetes:/etc/kubernetes -v /var/run:/
```

```
var/run rancher/rancher-agent:v2.3.5 --
server https://127.0.0.1 --token 7l94
mpqxtmxnhwwlzv2jmllg4shb5bhfksbkqxdr78t98
tp9g9hqfp --ca-checksum 0b63eb498d969aa0a9
33dfb654772bb9c76353acae443a618d620048a816
2bd1 --etcd --controlplane --worker
```

Kode Sumber A.8: Manmbah master node

Sedangkan pada virtual private server api-gw-2-176 hanya bertugas sebagai worker sehingga pada terminal di api-gw-2-176 dijalankan perintah berikut.

```
$ sudo docker run -d --privileged --restart=
unless-stopped --net=host -v /etc/
kubernetes:/etc/kubernetes -v /var/run:/
var/run rancher/rancher-agent:v2.3.5 --
server https://127.0.0.1 --token 7l94
mpqxtmxnhwwlzv2jmllg4shb5bhfksbkqxdr78t98
tp9g9hqfp --ca-checksum 0b63eb498d969aa0a9
33dfb654772bb9c76353acae443a618d620048a816
2bd1 --worker
```

Kode Sumber A.9: Menambah worker node

LAMPIRAN B

KODE SUMBER

Kode Sumber Implementasi Application

Kode Sumber nginx_kong.lua

```
return [[
pid pids/nginx.pid;

# overriding kong's env variable
env KONG_SSL_CERT_KEY;
env KONG_SSL_CERT_DER;

error_log ${{PROXY_ERROR_LOG}} ${{LOG_LEVEL}};

# injected nginx_main_* directives
> for _, el in ipairs(nginx_main_directives) do
$(el.name) $(el.value);
> end

events {
    # injected nginx_events_* directives
> for _, el in ipairs(nginx_events_directives)
   do
    $(el.name) $(el.value);
> end
}

> if role == "control_plane" or #
   proxy_listeners > 0 or #admin_listeners > 0
   or #status_listeners > 0 then
http {
  log_format custom_log '$request_id
     $request_time $upstream_response_time';
    include 'nginx-kong.conf';
```

```
}
> end

> if #stream_listeners > 0 then
stream {
    include 'nginx-kong-stream.conf';
}
> end
]]
```

Kode Sumber B.1: Kode sumber nginx_kong.lua

Kode Sumber nginx_kong.lua

```
return [[
charset UTF-8;
server_tokens off;

> if anonymous_reports then
${{SYSLOG_REPORTS}}
> end
error_log ${{PROXY_ERROR_LOG}} ${{LOG_LEVEL}};

lua_package_path       '${{LUA_PACKAGE_PATH}}
    ;;';
lua_package_cpath      '${{LUA_PACKAGE_CPATH}}
    ;;';
lua_socket_pool_size   ${{LUA_SOCKET_POOL_SIZE}}
    };
lua_socket_log_errors  off;
lua_max_running_timers 4096;
lua_max_pending_timers 16384;
```

```
lua_ssl_verify_depth   ${{LUA_SSL_VERIFY_DEPTH}
    };
> if lua_ssl_trusted_certificate then
lua_ssl_trusted_certificate '${{
    LUA_SSL_TRUSTED_CERTIFICATE}}';
> end

lua_shared_dict kong                        5m;
lua_shared_dict kong_locks                  8m;
lua_shared_dict kong_healthchecks           5m;
lua_shared_dict kong_process_events         5m;
lua_shared_dict kong_cluster_events         5m;
lua_shared_dict kong_rate_limiting_counters 12m
    ;
lua_shared_dict kong_core_db_cache          ${{
    MEM_CACHE_SIZE}};
lua_shared_dict kong_core_db_cache_miss     12m
    ;
lua_shared_dict kong_db_cache               ${{
    MEM_CACHE_SIZE}};
lua_shared_dict kong_db_cache_miss          12m
    ;
> if database == "off" then
lua_shared_dict kong_core_db_cache_2        ${{
    MEM_CACHE_SIZE}};
lua_shared_dict kong_core_db_cache_miss_2   12m
    ;
lua_shared_dict kong_db_cache_2             ${{
    MEM_CACHE_SIZE}};
lua_shared_dict kong_db_cache_miss_2        12m
    ;
> end
> if database == "cassandra" then
```

```
lua_shared_dict kong_cassandra                5m;
> end
> if role == "control_plane" then
lua_shared_dict kong_clustering               5m;
> end

underscores_in_headers on;
> if ssl_ciphers then
ssl_ciphers ${{SSL_CIPHERS}};
> end

# injected nginx_http_* directives
> for _, el in ipairs(nginx_http_directives) do
$(el.name) $(el.value);
> end

init_by_lua_block {
    Kong = require 'kong'
    Kong.init()
}

init_worker_by_lua_block {
    Kong.init_worker()
}

> if (role == "traditional" or role == "
   data_plane") and #proxy_listeners > 0 then
upstream kong_upstream {
    server 0.0.0.1;
    balancer_by_lua_block {
        Kong.balancer()
    }
```

```
    # injected nginx_upstream_* directives
> for _, el in ipairs(nginx_upstream_directives
   ) do
    $(el.name) $(el.value);
> end
}

server {
    server_name kong;
> for _, entry in ipairs(proxy_listeners) do
    listen $(entry.listener);
> end

    error_page 400 404 408 411 412 413 414 417
        494 /kong_error_handler;
    error_page 500 502 503 504
                              /kong_error_handler
        ;

    access_log ${{PROXY_ACCESS_LOG}};
    access_log /cuslog/custom_nginx.log
        custom_log;
    error_log  ${{PROXY_ERROR_LOG}} ${{
        LOG_LEVEL}};

> if proxy_ssl_enabled then
    ssl_certificate     ${{SSL_CERT}};
    ssl_certificate_key ${{SSL_CERT_KEY}};
    ssl_session_cache   shared:SSL:10m;
    ssl_certificate_by_lua_block {
        Kong.ssl_certificate()
    }
> end
```

```
    # injected nginx_proxy_* directives
> for _, el in ipairs(nginx_proxy_directives)
   do
    $(el.name) $(el.value);
> end
> for i = 1, #trusted_ips do
    set_real_ip_from  $(trusted_ips[i]);
> end

    rewrite_by_lua_block {
        Kong.rewrite()
    }

    access_by_lua_block {
        Kong.access()
    }

    header_filter_by_lua_block {
        Kong.header_filter()
    }

    body_filter_by_lua_block {
        Kong.body_filter()
    }

    log_by_lua_block {
        Kong.log()
    }

    location / {
        default_type                     '';
```

```
        set $ctx_ref                    '';
        set $upstream_te                '';
        set $upstream_host              '';
        set $upstream_upgrade           '';
        set $upstream_connection        '';
        set $upstream_scheme            '';
        set $upstream_uri               '';
        set $upstream_x_forwarded_for   '';
        set $upstream_x_forwarded_proto '';
        set $upstream_x_forwarded_host  '';
        set $upstream_x_forwarded_port  '';
        set $kong_proxy_mode            'http';

        proxy_http_version    1.1;
        proxy_set_header      TE
                          $upstream_te;
        proxy_set_header      Host
                        $upstream_host;
        proxy_set_header      Upgrade
                     $upstream_upgrade;
        proxy_set_header      Connection
                  $upstream_connection;
        proxy_set_header      X-Forwarded-For
             $upstream_x_forwarded_for;
        proxy_set_header      X-Forwarded-Proto
            $upstream_x_forwarded_proto;
        proxy_set_header      X-Forwarded-Host
            $upstream_x_forwarded_host;
        proxy_set_header      X-Forwarded-Port
            $upstream_x_forwarded_port;
        proxy_set_header      X-Real-IP
                   $remote_addr;
        proxy_pass_header     Server;
```

```
        proxy_pass_header     Date;
        proxy_ssl_name        $upstream_host;
        proxy_ssl_server_name on;
> if client_ssl then
        proxy_ssl_certificate ${{
            CLIENT_SSL_CERT}};
        proxy_ssl_certificate_key ${{
            CLIENT_SSL_CERT_KEY}};
> end
        proxy_pass            $upstream_scheme:
            //kong_upstream$upstream_uri;
    }

    location @grpc {
        internal;
        default_type         '';
        set $kong_proxy_mode 'grpc';

        grpc_set_header      TE
            $upstream_te;
        grpc_set_header      Host
            $upstream_host;
        grpc_set_header      X-Forwarded-For
            $upstream_x_forwarded_for;
        grpc_set_header      X-Forwarded-Proto
            $upstream_x_forwarded_proto;
        grpc_set_header      X-Forwarded-Host
            $upstream_x_forwarded_host;
        grpc_set_header      X-Forwarded-Port
            $upstream_x_forwarded_port;
        grpc_set_header      X-Real-IP
            $remote_addr;
        grpc_pass_header     Server;
```

```
        grpc_pass_header     Date;
        grpc_pass            grpc://
            kong_upstream;
    }

    location @grpcs {
        internal;
        default_type         '';
        set $kong_proxy_mode 'grpc';

        grpc_set_header      TE
            $upstream_te;
        grpc_set_header      Host
            $upstream_host;
        grpc_set_header      X-Forwarded-For
            $upstream_x_forwarded_for;
        grpc_set_header      X-Forwarded-Proto
            $upstream_x_forwarded_proto;
        grpc_set_header      X-Forwarded-Host
            $upstream_x_forwarded_host;
        grpc_set_header      X-Forwarded-Port
            $upstream_x_forwarded_port;
        grpc_set_header      X-Real-IP
            $remote_addr;
        grpc_pass_header     Server;
        grpc_pass_header     Date;
        grpc_ssl_name        $upstream_host;
        grpc_ssl_server_name on;
> if client_ssl then
        grpc_ssl_certificate ${{CLIENT_SSL_CERT
            }};
        grpc_ssl_certificate_key ${{
            CLIENT_SSL_CERT_KEY}};
```

```
> end
        grpc_pass            grpcs://
           kong_upstream;
    }

    location = /kong_buffered_http {
        internal;
        default_type         '';
        set $kong_proxy_mode 'http';

        rewrite_by_lua_block       {;}
        access_by_lua_block        {;}
        header_filter_by_lua_block {;}
        body_filter_by_lua_block   {;}
        log_by_lua_block           {;}

        proxy_http_version 1.1;
        proxy_set_header      TE
                           $upstream_te;
        proxy_set_header      Host
                         $upstream_host;
        proxy_set_header      Upgrade
                      $upstream_upgrade;
        proxy_set_header      Connection
                   $upstream_connection;
        proxy_set_header      X-Forwarded-For
              $upstream_x_forwarded_for;
        proxy_set_header      X-Forwarded-Proto
             $upstream_x_forwarded_proto;
        proxy_set_header      X-Forwarded-Host
             $upstream_x_forwarded_host;
        proxy_set_header      X-Forwarded-Port
             $upstream_x_forwarded_port;
```

```
        proxy_set_header      X-Real-IP
                    $remote_addr;
        proxy_pass_header     Server;
        proxy_pass_header     Date;
        proxy_ssl_name        $upstream_host;
        proxy_ssl_server_name on;
> if client_ssl then
        proxy_ssl_certificate ${{
            CLIENT_SSL_CERT}};
        proxy_ssl_certificate_key ${{
            CLIENT_SSL_CERT_KEY}};
> end
        proxy_pass            $upstream_scheme:
            //kong_upstream$upstream_uri;
    }

    location = /kong_error_handler {
        internal;
        default_type                 '';

        uninitialized_variable_warn  off;

        rewrite_by_lua_block {;}
        access_by_lua_block  {;}

        content_by_lua_block {
            Kong.handle_error()
        }
    }
}
> end -- (role == "traditional" or role == "
    data_plane") and #proxy_listeners > 0
```

```
> if (role == "control_plane" or role == "
    traditional") and #admin_listeners > 0 then
server {
    server_name kong_admin;
> for _, entry in ipairs(admin_listeners) do
    listen $(entry.listener);
> end

    access_log ${{ADMIN_ACCESS_LOG}};
    access_log /cuslog/custom_nginx.log
        custom_log;
    error_log  ${{ADMIN_ERROR_LOG}} ${{
        LOG_LEVEL}};

    client_max_body_size    10m;
    client_body_buffer_size 10m;

> if admin_ssl_enabled then
    ssl_certificate     ${{ADMIN_SSL_CERT}};
    ssl_certificate_key ${{ADMIN_SSL_CERT_KEY}}
        ;
    ssl_session_cache   shared:AdminSSL:10m;
> end

    # injected nginx_admin_* directives
> for _, el in ipairs(nginx_admin_directives)
    do
    $(el.name) $(el.value);
> end

    location / {
        default_type application/json;
        content_by_lua_block {
```

```
                Kong.admin_content()
            }
            header_filter_by_lua_block {
                Kong.admin_header_filter()
            }
        }

        location /nginx_status {
            access_log off;
            stub_status;
            allow 127.0.0.1;
            deny all;
        }

        location /robots.txt {
            return 200 'User-agent: *\nDisallow:
                /';
        }
    }
    > end -- (role == "control_plane" or role == "
        traditional") and #admin_listeners > 0

    > if #status_listeners > 0 then
    server {
        server_name kong_status;
    > for _, entry in ipairs(status_listeners) do
        listen $(entry.listener);
    > end

        access_log ${{STATUS_ACCESS_LOG}};
        access_log /cuslog/custom_nginx.log
            custom_log;
```

```
    error_log  ${{STATUS_ERROR_LOG}} ${{
        LOG_LEVEL}};

    # injected nginx_status_* directives
> for _, el in ipairs(nginx_status_directives)
    do
    $(el.name) $(el.value);
> end

    location / {
        default_type application/json;
        content_by_lua_block {
            Kong.status_content()
        }
        header_filter_by_lua_block {
            Kong.status_header_filter()
        }
    }

    location /nginx_status {
        access_log off;
        stub_status;
        allow 127.0.0.1;
        deny all;
    }

    location /robots.txt {
        return 200 'User-agent: *\nDisallow:
            /';
    }
}
> end
```

```
  > if role == "control_plane" then
  server {
      server_name kong_cluster_listener;
  > for _, entry in ipairs(cluster_listeners) do
      listen $(entry.listener) ssl;
  > end

      access_log off;

      ssl_verify_client   optional_no_ca;
      ssl_certificate     ${{CLUSTER_CERT}};
      ssl_certificate_key ${{CLUSTER_CERT_KEY}};
      ssl_session_cache   shared:ClusterSSL:10m;

      location = /v1/outlet {
          content_by_lua_block {
              Kong.serve_cluster_listener()
          }
      }
  }
  > end -- role == "control_plane"
  ]]
```

Kode Sumber B.2: Kode sumber nginx_kong.lua

Kode Sumber main.py

```
  import prometheus_client as prom
import random
import time
import requests
import os.path
import shutil
```

```
def readNewFile():
    file_time = 0
    file_sum = 0
    file_n = 0
    file_new=open("/app/new.log","r")
    for file_line in file_new:
        file_data = file_line.split(" ")
        file_time = float(file_data[1])

        file_sum = file_sum+file_time
        file_n = file_n + 1

    file_new.close()
    if file_n == 0:
        file_avg = 0
    else:
        file_avg = file_sum / file_n
    return file_avg

def checkFirstLine():
    flag = 0
    file_new=open("/app/new.log","r")
    file_old=open("/app/old.log","r")
    for line1 in file_new:
        for line2 in file_old:
            if line1==line2:
                flag = 1
                break
            else:
                flag = 0
                break
    file_new.close()
```

```
    file_old.close()
    return flag

def diffFile():
    old = set((line.strip() for line in open('/
        app/old.log')))
    new = set((line.strip() for line in open('/
        app/new.log')))

    file_sum = 0
    file_n = 0

    for line in new:
        if line not in old:
            file_data = line.split(" ")
            file_time = float(file_data[1])

            file_sum = file_sum+file_time
            file_n = file_n + 1

    if file_n == 0:
        file_avg = 0
    else:
        file_avg = file_sum / file_n
    return file_avg

if __name__ == '__main__':

    active_conn = prom.Gauge('konghq_active', '
        Gauge untuk mendapatkan jumlah active
        connection')
    accept_conn = prom.Counter('konghq_accept', '
        Counter untuk mendapatkan total accepted
```

```
        connection')
    handle_conn = prom.Counter('konghq_handle', '
        Counter untuk mendapatkan total hundled
        connection')
    request_conn = prom.Counter('konghq_request',
         'Counter untuk mendapatkan total client
        request')
    read_conn = prom.Gauge('konghq_reading', '
        Gauge untuk mendapatkan jumlah reading
        connection')
    write_conn = prom.Gauge('konghq_writing', '
        Gauge untuk mendapatkan jumlah writing
        connection')
    wait_conn = prom.Gauge('konghq_waiting', '
        Gauge untuk mendapatkan jumlah waiting
        connection')

    rtime_desc = prom.Gauge('konghq_rtime', '
        Gauge untuk mendapatkan rata-rata rtime
        tiap 1 detik')


    prom.start_http_server(8888)
    url= "http://127.0.0.1:8001/nginx_status"

    file_flag = 1
    while True:
        try:
            rtime_value = 0
            if (os.path.exists('/cuslog/
                custom_nginx.log')):
                 if file_flag == 1:
```

```
                    shutil.copyfile('/cuslog/
                        custom_nginx.log', '/app/
                        new.log')
                    rtime_value = readNewFile()
                    file_flag = 0
                else:
                    shutil.copyfile('/cuslog/
                        custom_nginx.log', '/app/
                        new.log')
                    same_line = checkFirstLine()
                    if same_line == 1:
                        rtime_value = diffFile()
                    else:
                        rtime_value = readNewFile
                            ()

                shutil.move('/app/new.log', '/app
                    /old.log')

            r = requests.get(url)
            data = r.text.split(" ")
            active_value = int(data[2])
            accept_value = int(data[7])
            handle_value = int(data[8])
            request_value = int(data[9])
            read_value = int(data[11])
            write_value = int(data[13])
            wait_value = int(data[15])

            active_conn.set(active_value)
            accept_conn.inc(accept_value)
            handle_conn.inc(handle_value)
            request_conn.inc(request_value)
```

```
            read_conn.set(read_value)
            write_conn.set(write_value)
            wait_conn.set(wait_value)

            rtime_desc.set(rtime_value)

        except:
            pass

        time.sleep(1)
```

Kode Sumber B.3: Kode sumber main.py

Kode Sumber requirements.txt

```
prometheus_client
Flask
flask-prometheus
requests
```

Kode Sumber B.4: Kode sumber requirements.txt

Kode Sumber postgres-persistent.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: postgres-pv-volume
  labels:
    type: local
    app: postgres
spec:
  storageClassName: manual
```

```
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  hostPath:
    path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pv-claim
  labels:
    app: postgres
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
```

Kode Sumber B.5: Kode sumber postgres-persistent.yaml

Kode Sumber migration-bootstrap.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: kong-migrations-bootstrap
  namespace: default
spec:
  template:
    metadata:
```

```
        name: kong-migrations-bootstrap
      spec:
        initContainers:
        - name: wait-for-postgres
          image: busybox
          env:
          - name: KONG_PG_HOST
            value: postgres
          - name: KONG_PG_PORT
            value: "5432"
          command: [ "/bin/sh", "-c", "until nc -
              zv $KONG_PG_HOST $KONG_PG_PORT -w1;
              do echo 'waiting for db'; sleep 1;
              done" ]
        containers:
        - name: kong-migrations-bootstrap
          image: lerufic/kong-its:v2.0.7
          env:
          - name: KONG_PG_PASSWORD
            value: kong
          - name: KONG_PG_HOST
            value: postgres
          - name: KONG_PG_PORT
            value: "5432"
          command: [ "/bin/sh", "-c", "kong
              migrations bootstrap" ]
        restartPolicy: OnFailure
```

Kode Sumber B.6: Kode sumber migration-bootstrap.yaml

Kode Sumber migration-up.yaml

```
  apiVersion: batch/v1
```

```
kind: Job
metadata:
  name: kong-migrations-up-finish
  namespace: default
spec:
  template:
    metadata:
      name: kong-migrations-up-finish
    spec:
      initContainers:
      - name: wait-for-postgres
        image: busybox
        env:
        - name: KONG_PG_HOST
          value: postgres
        - name: KONG_PG_PORT
          value: "5432"
        command: [ "/bin/sh", "-c", "until nc -
            zv $KONG_PG_HOST $KONG_PG_PORT -w1;
            do echo 'waiting for db'; sleep 1;
            done" ]
      containers:
      - name: kong-migrations-up-finish
        image: lerufic/kong-its:v2.0.7
        env:
        - name: KONG_PG_PASSWORD
          value: kong
        - name: KONG_PG_HOST
          value: postgres
        - name: KONG_PG_PORT
          value: "5432"
        command: [ "/bin/sh", "-c", "kong
            migrations up && kong migrations
```

```
            finish" ]
      restartPolicy: OnFailure
```

Kode Sumber B.7: Kode sumber migration-up.yaml

Kode Sumber konga-prepare.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: konga-prepare
  namespace: default
spec:
  template:
    metadata:
      name: konga-prepare
    spec:
      initContainers:
      - name: wait-for-postgres
        image: busybox
        env:
        - name: KONG_PG_HOST
          value: postgres
        - name: KONG_PG_PORT
          value: "5432"
        command: [ "/bin/sh", "-c", "until nc -
            zv $KONG_PG_HOST $KONG_PG_PORT -w1;
            do echo 'waiting for db'; sleep 1;
            done" ]
      containers:
      - name: konga-prepare
        image: pantsel/konga
```

```
          command: ["/bin/bash","start.sh", "-c",
              "prepare", "-a","postgres", "-u","
              postgresql://kong:kong@postgres:5432
              /konga_db"]
        restartPolicy: OnFailure
```

Kode Sumber B.8: Kode sumber konga-prepare.yaml

Kode Sumber konga-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: konga
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: konga
  template:
    metadata:
      labels:
        app: konga
    spec:
      containers:
      - env:
        - name: DB_ADAPTER
          value: postgres
        - name: DB_HOST
          value: postgres
        - name: DB_PORT
          value: "5432"
```

```
          - name: DB_USER
            value: kong
          - name: TOKEN_SECRET
            value: km1GUr4RkcQD7
                DewhJPNXrCuZwcKmqjb
          - name: DB_DATABASE
            value: konga_db
          - name: NODE_ENV
            value: production
          name: konga
          image: pantsel/konga
          ports:
          - containerPort: 1337
            name: port1
            protocol: TCP
        restartPolicy: Always
```

Kode Sumber B.9: Kode sumber konga-deployment.yaml

Kode Sumber konga-prepare.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: konga
  labels:
    app: konga
spec:
  type: NodePort
  ports:
  - port: 1337
    protocol: TCP
    name: port1
```

```
  selector:
   app: konga
```

Kode Sumber B.10: Kode sumber konga-service.yaml

Kode Sumber kong-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: kong
  labels:
    app: kong
spec:
  type: NodePort
  ports:
  - port: 8000
    protocol: TCP
    name: port1
  - port: 8443
    protocol: TCP
    name: port2
  - port: 8001
    protocol: TCP
    name: port3
  - port: 8444
    protocol: TCP
    name: port4
  - port: 8888
    protocol: TCP
    name: port5
  selector:
   app: kong
```

Kode Sumber B.11: Kode sumber kong-service.yaml

Kode Sumber aggregated-metric-read.yaml

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: system:aggregated-metrics-reader
  labels:
    rbac.authorization.k8s.io/aggregate-to-view
        : "true"
    rbac.authorization.k8s.io/aggregate-to-edit
        : "true"
    rbac.authorization.k8s.io/aggregate-to-
        admin: "true"
rules:
- apiGroups: ["metrics.k8s.io"]
  resources: ["pods", "nodes"]
  verbs: ["get", "list", "watch"]
```

Kode Sumber B.12: Kode sumber aggregated-metric-read.yaml

Kode Sumber auth-delegator.yaml

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: metrics-server:system:auth-delegator
roleRef:
```

```
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:auth-delegator
subjects:
- kind: ServiceAccount
  name: metrics-server
  namespace: kube-system
```

Kode Sumber B.13: Kode sumber auth-delegator.yaml

Kode Sumber auth-reader-conf.yaml

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: metrics-server-auth-reader
  namespace: kube-system
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: extension-apiserver-authentication-
      reader
subjects:
- kind: ServiceAccount
  name: metrics-server
  namespace: kube-system
```

Kode Sumber B.14: Kode sumber auth-reader-conf.yaml

Kode Sumber metrics-apiservice.yaml

```
---
```

```
apiVersion: apiregistration.k8s.io/v1beta1
kind: APIService
metadata:
  name: v1beta1.metrics.k8s.io
spec:
  service:
    name: metrics-server
    namespace: kube-system
  group: metrics.k8s.io
  version: v1beta1
  insecureSkipTLSVerify: true
  groupPriorityMinimum: 100
  versionPriority: 100
```

Kode Sumber B.15: Kode sumber metrics-apiservice.yaml

Kode Sumber metrics-deployment.yaml

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: metrics-server
  namespace: kube-system
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: metrics-server
  namespace: kube-system
  labels:
    k8s-app: metrics-server
spec:
```

```
    selector:
      matchLabels:
        k8s-app: metrics-server
    template:
      metadata:
        name: metrics-server
        labels:
          k8s-app: metrics-server
      spec:
        serviceAccountName: metrics-server
        volumes:
        # mount in tmp so we can safely use from-
            scratch images and/or read-only
            containers
        - name: tmp-dir
          emptyDir: {}
        containers:
        - name: metrics-server
          image: k8s.gcr.io/metrics-server-amd64:
              v0.3.6
          imagePullPolicy: IfNotPresent
          command:
            - /metrics-server
            - --v=2
            - --kubelet-preferred-address-types=
                InternalIP
            - --kubelet-insecure-tls
          args:
            - --cert-dir=/tmp
            - --secure-port=4443
          ports:
          - name: main-port
            containerPort: 4443
```

```
            protocol: TCP
          securityContext:
            readOnlyRootFilesystem: true
            runAsNonRoot: true
            runAsUser: 1000
          volumeMounts:
          - name: tmp-dir
            mountPath: /tmp
        nodeSelector:
          kubernetes.io/os: linux
          kubernetes.io/arch: "amd64"
```

Kode Sumber B.16: Kode sumber metrics-deployment.yaml

Kode Sumber metric-service.yaml

```
---
apiVersion: v1
kind: Service
metadata:
  name: metrics-server
  namespace: kube-system
  labels:
    kubernetes.io/name: "Metrics-server"
    kubernetes.io/cluster-service: "true"
spec:
  selector:
    k8s-app: metrics-server
  ports:
  - port: 443
    protocol: TCP
    targetPort: main-port
```

Kode Sumber B.17: Kode sumber metric-service.yaml

Kode Sumber resource-reader.yaml

```yaml
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: system:metrics-server
rules:
- apiGroups:
  - ""
  resources:
  - pods
  - nodes
  - nodes/stats
  - namespaces
  - configmaps
  verbs:
  - get
  - list
  - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: system:metrics-server
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:metrics-server
subjects:
- kind: ServiceAccount
  name: metrics-server
  namespace: kube-system
```

Kode Sumber B.18: Kode sumber resource-reader.yaml

Kode Sumber prometheus-namespace.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: monitoring
```

Kode Sumber B.19: Kode sumber prometheus-namespace.yaml

Kode Sumber prometheus-clusterrole.yaml

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: prometheus
rules:
- apiGroups: [""]
  resources:
  - nodes
  - nodes/proxy
  - services
  - endpoints
  - pods
  verbs: ["get", "list", "watch"]
- apiGroups:
  - extensions
  resources:
  - ingresses
  verbs: ["get", "list", "watch"]
```

```
  - nonResourceURLs: ["/metrics"]
    verbs: ["get"]
```

Kode Sumber B.20: Kode sumber prometheus-clusterrole.yaml

Kode Sumber prometheus-clusterrolebinding.yaml

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
- kind: ServiceAccount
  name: prometheus
  namespace: monitoring
```

Kode Sumber B.21: Kode sumber prometheus-clusterrolebinding.yaml

Kode Sumber prometheus-serviceaccount.yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus
  namespace: monitoring
```

Kode Sumber B.22: Kode sumber prometheus-serviceaccount.yaml

Kode Sumber prometheus-configMap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
name: prometheus-config
labels:
  name: prometheus-config
namespace: monitoring
data:
prometheus.yml: |-
  global:
    scrape_interval: 5s
    evaluation_interval: 5s

  #rule_files:
  #  - /etc/prometheus/prometheus.rules
  #alerting:
  #  alertmanagers:
  #  - scheme: http
  #    static_configs:
  #    - targets:
  #      - "alertmanager.monitoring.svc:9093"

  scrape_configs:
    - job_name: 'kong'

      kubernetes_sd_configs:
      - role: pod

      relabel_configs:
      - source_labels: [
          __meta_kubernetes_namespace]
```

```
          action: replace
          target_label: k8s_namespace
        - source_labels: [
            __meta_kubernetes_pod_name]
          action: replace
          target_label: k8s_pod_name
        - source_labels: [__address__]
          action: replace
          regex: ([^:]+)(?::\d+)?
          replacement: ${1}:8888
          target_label: __address__
        - source_labels: [
            __meta_kubernetes_pod_label_app]
          action: keep
          regex: kong

      # - job_name: 'prometheus'
      #   static_configs:
      #   - targets: ['kong.default:8888']

      - job_name: 'kubelet'

        kubernetes_sd_configs:
        - role: node
        scheme: https

        tls_config:
          ca_file: /var/run/secrets/kubernetes.io
              /serviceaccount/ca.crt
          insecure_skip_verify: true  # Required
              with Minikube.

      - job_name: 'kubernetes-apiservers'
```

```
        kubernetes_sd_configs:
        - role: endpoints
        scheme: https

        tls_config:
          ca_file: /var/run/secrets/kubernetes.io
             /serviceaccount/ca.crt
        bearer_token_file: /var/run/secrets/
            kubernetes.io/serviceaccount/token

        relabel_configs:
        - source_labels: [
            __meta_kubernetes_namespace,
            __meta_kubernetes_service_name,
            __meta_kubernetes_endpoint_port_name]
          action: keep
          regex: default;kubernetes;https

      - job_name: 'kubernetes-nodes'

        scheme: https

        tls_config:
          ca_file: /var/run/secrets/kubernetes.io
             /serviceaccount/ca.crt
        bearer_token_file: /var/run/secrets/
            kubernetes.io/serviceaccount/token

        kubernetes_sd_configs:
        - role: node

        relabel_configs:
```

```
        - action: labelmap
          regex: __meta_kubernetes_node_label_
              (.+)
        - target_label: __address__
          replacement: kubernetes.default.svc:443
        - source_labels: [
            __meta_kubernetes_node_name]
          regex: (.+)
          target_label: __metrics_path__
          replacement: /api/v1/nodes/${1}/proxy/
              metrics


      - job_name: 'kubernetes-pods'

        kubernetes_sd_configs:
        - role: pod

        relabel_configs:
        - source_labels: [__meta_kubernetes_pod_
        annotation_prometheus_io_scrape]
          action: keep
          regex: true
        - source_labels: [__meta_kubernetes_pod_
        annotation_prometheus_io_path]
          action: replace
          target_label: __metrics_path__
          regex: (.+)
        - source_labels: [__address__, __meta_
        kubernetes_pod_annotation_
        prometheus_io_port]
          action: replace
          regex: ([^:]+)(?::\d+)?;(\d+)
```

```
          replacement: $1:$2
          target_label: __address__
        - action: labelmap
          regex: __meta_kubernetes_pod_label_(.+)
        - source_labels: [
            __meta_kubernetes_namespace]
          action: replace
          target_label: kubernetes_namespace
        - source_labels: [
            __meta_kubernetes_pod_name]
          action: replace
          target_label: kubernetes_pod_name

      - job_name: 'kube-state-metrics'
        static_configs:
          - targets: ['kube-state-metrics.kube-
              system.svc.cluster.local:8080']

      - job_name: 'kubernetes-cadvisor'

        scheme: https

        tls_config:
          ca_file: /var/run/secrets/kubernetes.io
              /serviceaccount/ca.crt
        bearer_token_file: /var/run/secrets/
            kubernetes.io/serviceaccount/token

        kubernetes_sd_configs:
        - role: node

        relabel_configs:
        - action: labelmap
```

```
          regex: __meta_kubernetes_node_label_
              (.+)
        - target_label: __address__
          replacement: kubernetes.default.svc:443
        - source_labels: [
            __meta_kubernetes_node_name]
          regex: (.+)
          target_label: __metrics_path__
          replacement: /api/v1/nodes/${1}/proxy/
              metrics/cadvisor

      - job_name: 'kubernetes-service-endpoints'

        kubernetes_sd_configs:
        - role: endpoints

        relabel_configs:
        - source_labels: [__meta_kubernetes_
        service_annotation_prometheus_io_scrape]
          action: keep
          regex: true
        - source_labels: [__meta_kubernetes_
        service_annotation_prometheus_io_scheme]
          action: replace
          target_label: __scheme__
          regex: (https?)
        - source_labels: [__meta_kubernetes_
        service_annotation_prometheus_io_path]
          action: replace
          target_label: __metrics_path__
          regex: (.+)
        - source_labels: [__address__, __meta_
        kubernetes_service_annotation_
```

```
        prometheus_io_port]
          action: replace
          target_label: __address__
          regex: ([^:]+)(?::\d+)?;(\d+)
          replacement: $1:$2
        - action: labelmap
          regex: __meta_kubernetes_service_label_
              (.+)
        - source_labels: [
            __meta_kubernetes_namespace]
          action: replace
          target_label: kubernetes_namespace
        - source_labels: [
            __meta_kubernetes_service_name]
          action: replace
          target_label: kubernetes_name
```

Kode Sumber B.23: Kode sumber prometheus-configMap.yaml

Kode Sumber prometheus-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: prometheus
  namespace: monitoring
  labels:
    app: prometheus
spec:
  type: NodePort
  ports:
  - port: 9090
    protocol: TCP
```

```
    selector:
      app: prometheus
```

Kode Sumber B.24: Kode sumber prometheus-service.yaml

(Halaman ini sengaja dikosongkan)

BIODATA PENULIS

Aguel Satria Wijaya lahir pada 24 Agustus 1998 di Semarang. Penulis menempuh pendidikan formal di TK PL Bernardus Semarang (2002-2004), SD PL Bernardus Semarang (2004-2010), SMP Domenico Savio Semarang (2010-2013), SMA Kolese Loyola Semarang (2013-2016), dan Departemen Informatika Institut Teknologi Sepuluh Nopember (ITS) Surabaya (2016-2020). Selama menempuh studi penulis aktif di organisasi seperti staff pada tahun ke-2 dan staff ahli pada tahun ke-3 di Departemen Kaderisasi dan Pemetaan (KDPM) Himpunan Mahasiswa Teknik Computer-Informatika (HMTC). Penulis juga pernah menjadi Wakil Koordinator II di SCHEMATICS 2017 dan Koordinator di SCHEMATICS 2018 pada divisi keamanan dan perizinan. Penulis juga merupakan *administrator* lab Arsitektur dan Jaringan Komputer (AJK) dan juga pernah menjadi asisten dosen mata kuliah Sistem Operasi, Jaringan Komputer, dan Pemrograman Jaringan. Penulis juga aktif dalam beberapa proyek seperti *E-Vote* pemilihan rektor ITS 2019, PPDB Surabaya 2018, dan PPDB Surabaya 2019. Penulis dapat dihubungi melalui surat elektronik di `aguelsw@gmail.com`.