IMPLEMENTASI SISTEM *FAILOVER* OTOMATIS MENGGUNAKAN HAPROXY DAN KEEPALIVED PADA INFRASTRUKTUR BERBASIS DOCKER UNTUK SISTEM INFORMASI BERBASIS *WEB*

SKRIPSI

JOKO PRASETYO

2107421012

PROGRAM STUDI TEKNIK MULTIMEDIA DAN JARINGAN

JURUSAN TEKNIK INFORMATIKA DAN KOMPUTER

POLITEKNIK NEGERI JAKARTA

2025

IMPLEMENTASI SISTEM *FAILOVER* OTOMATIS MENGGUNAKAN HAPROXY DAN KEEPALIVED PADA INFRASTRUKTUR BERBASIS DOCKER UNTUK SISTEM INFORMASI BERBASIS *WEB*

SKRIPSI

Diajukan sebagai salah satu syarat untuk memperoleh gelar

Sarjana Terapan

JOKO PRASETYO

2107421012

PROGRAM STUDI TEKNIK MULTIMEDIA DAN JARINGAN

JURUSAN TEKNIK INFORMATIKA DAN KOMPUTER

POLITEKNIK NEGERI JAKARTA

2025

SURAT PERNYATAAN BEBAS PLAGIARISME

Saya yang bertanda tangan di bawah ini:

Nama : Joko Prasetyo

NIM : 2107421012

Jurusan/Program Studi : Teknik Informatika dan Komputer / Teknik Multimedia dan Jaringan

Judul Skripsi : IMPLEMENTASI SISTEM *FAILOVER* OTOMATIS MENGGUNAKAN HAPROXY DAN KEEPALIVED PADA INFRASTRUKTUR BERBASIS DOCKER UNTUK SISTEM INFORMASI BERBASIS *WEB*

Dengan ini menyatakan dengan sebenar-benarnya bahwa skripsi ini benar-benar merupakan hasil karya saya sendiri, bebas dari peniruan terhadap karya orang lain. Kutipan pendapat dan tulisan orang lain telah saya cantumkan sesuai dengan kaidah penulisan karya ilmiah yang berlaku.

Apabila di kemudian hari terbukti atau dapat dibuktikan bahwa di dalam skripsi ini terdapat unsur-unsur plagiat atau bentuk-bentuk pelanggaran hak cipta lainnya, maka saya bersedia menerima sanksi sesuai dengan peraturan yang berlaku.

Depok, 10 Juni 2025

Yang Membuat Pernyataan,

Joko Prasetyo

NIM.2107421012

LEMBAR PENGESAHAN

Skripsi diajukan oleh:

Nama : Joko Prasetyo

NIM : 2107421012

Program Studi : Teknik Multimedia dan Jaringan

Judul Skripsi : IMPLEMENTASI SISTEM *FAILOVER* OTOMATIS MENGGUNAKAN HAPROXY DAN KEEPALIVED PADA INFRASTRUKTUR BERBASIS DOCKER UNTUK SISTEM INFORMASI BERBASIS *WEB*

Telah diuji oleh tim penguji dalam Sidang Skripsi pada hari Jum'at, Tanggal 20, Bulan Juni, Tahun 2025 dan dinyatakan **LULUS.**

Disahkan Oleh

Pembimbing I : Dr. Prihatin Oktivasari, S.Si., M.Si. ( )

Penguji I : Dr. Indra Hermawan., M.Kom. ( )

Penguji II : Asep Kurniawan, S.Pd., M.Kom. ( )

Penguji III : Fachroni Arbi Murad, S.Kom., M.Kom. ( )

Mengetahui:

Jurusan Teknik Informatika dan Komputer

Ketua

Dr. Anita Hidayati, S.Kom., M.Kom.

NIP. 197908032003122003

KATA PENGANTAR

Puji syukur penulis panjatkan ke hadirat Allah SWT atas segala rahmat, karunia, dan kemudahan yang telah diberikan, sehingga penulis dapat menyelesaikan penyusunan skripsi ini. Skripsi ini disusun sebagai salah satu syarat untuk memperoleh gelar Sarjana Terapan pada Program Studi Teknik Multimedia dan Jaringan, Jurusan Teknik Informatika dan Komputer, Politeknik Negeri Jakarta. Dalam proses penyusunan skripsi ini, penulis banyak mendapatkan bantuan, bimbingan, serta dukungan dari berbagai pihak. Oleh karena itu, pada kesempatan ini penulis ingin menyampaikan ucapan terima kasih yang sebesar-besarnya kepada:

1. Orang tua dan keluarga atas segala doa, dukungan moral, dan semangat yang senantiasa mengiringi penulis.
2. Ibu Prihatin Oktivasari selaku dosen pembimbing yang telah memberikan bimbingan, arahan, dan motivasi kepada penulis selama proses penelitian hingga penyusunan skripsi ini.
3. Bapak Deden Solihin, selaku analis di Loka PSPL yang telah memberikan arahan dan bantuan selama proses pengerjaan skripsi ini.
4. Rekan-rekan mahasiswa serta semua pihak yang tidak dapat penulis sebutkan satu per satu, yang telah membantu dan memberikan dukungan selama proses penelitian dan penyusunan skripsi ini.

Akhir kata, penulis berharap semoga skripsi ini dapat memberikan manfaat bagi pengembangan ilmu pengetahuan, khususnya di bidang teknologi sistem *High Availability* dan implementasi infrastruktur berbasis *container*.

Jakarta, 10 Juni 2025

Joko Prasetyo

SURAT PERNYATAAN PERSETUJUAN PUBLIKASI SKRIPSI UNTUK KEPENTINGAN AKADEMIS

Sebagai sivitas akademik Politeknik Negeri Jakarta, saya bertanda tangan dibawah ini:

Nama : Joko Prasetyo

NIM : 2107421012

Jurusan/Program Studi : T.Informatika dan Komputer / T.Multimedia dan Jaringan

Demi pengembangan ilmu pengetahuan, menyetujui untuk memberikan kepada Politeknik Negeri Jakarta Hak Bebas Royalti Non-Eksklusif atas karya ilmiah saya yang berjudul:

"Implementasi Sistem *Failover* Otomatis Menggunakan Haproxy Dan Keepalived Pada Infrastruktur Berbasis Docker Untuk Sistem Informasi Berbasis *Web*"

Beserta perangkat yang ada (jika diperlukan). Dengan Hak Bebas Royalti Non-Eksklusif ini Politeknik Negeri Jakarta Berhak menyimpan, mengalihmediakan/formatkan, mengelola dalam bentuk pangkalan data (*database*), merawat, dan memublikasikan skripsi saya tanpa meminta izin dari saya selama tetap mencantumkan nama saya sebagai penulis/pencipta dan sebagai pemilik Hak Cipta.

Demikian pernyatan ini saya buat dengan sebenarnya.

Depok, 10 Juni 2025

Yang Menyatakan,

(Joko Prasetyo)

NIM.2107421012

Implementasi Sistem *Failover* Otomatis Menggunakan Haproxy Dan Keepalived Pada Infrastruktur Berbasis Docker Untuk Sistem Informasi Berbasis *Web*

ABSTRAK

Ketersediaan layanan sistem informasi berbasis web menjadi aspek penting dalam menunjang operasional yang andal. Penelitian ini bertujuan untuk mengimplementasikan sistem failover otomatis dan load balancing guna meningkatkan ketersediaan layanan dengan memanfaatkan HAProxy, Keepalived, dan Docker Swarm. Sistem dibangun untuk mendukung layanan identifikasi sirip punggung hiu berbasis web yang memerlukan kontinuitas layanan tanpa gangguan. Metode penelitian yang digunakan adalah eksperimen, dengan perancangan dan implementasi infrastruktur High Availability (HA) berbasis Docker Swarm, HAProxy sebagai load balancer, serta Keepalived sebagai mekanisme failover otomatis. Pengujian dilakukan mencakup aspek failover, failback, downtime, load balancing, serta performa sistem di bawah beban tinggi. Hasil pengujian menunjukkan bahwa proses failover berlangsung rata-rata dalam waktu 1 detik, downtime layanan minimal (0–1 detik), dan failback berjalan otomatis dalam waktu 0–4 detik. Sistem load balancing berjalan sesuai desain dengan algoritma round robin dan least connection. Pada pengujian beban hingga 10.000 request, sistem tetap dapat melayani permintaan dengan tingkat error yang sangat rendah, meskipun terjadi peningkatan waktu respons dan pemakaian sumber daya. Implementasi arsitektur ini membuktikan bahwa integrasi HAProxy, Keepalived, dan Docker Swarm dapat meningkatkan ketersediaan layanan sistem informasi berbasis web dengan performa yang masih dapat diterima. Penelitian ini dapat menjadi acuan bagi pengembangan sistem serupa yang memerlukan tingkat ketersediaan layanan yang tinggi.

Kata kunci: High Availability, Failover, Load balancing, HAProxy, Keepalived, Docker Swarm, Sistem Informasi Berbasis Web.

DAFTAR ISI

SURAT PERNYATAAN BEBAS PLAGIARISME ..... **i**

LEMBAR PENGESAHAN ..... **ii**

KATA PENGANTAR ..... **iii**

SURAT PERNYATAAN PERSETUJUAN PUBLIKASI SKRIPSI UNTUK KEPENTINGAN AKADEMIS ..... **iv**

ABSTRAK ..... **v**

DAFTAR ISI ..... **vi**

DAFTAR GAMBAR ..... **x**

DAFTAR TABEL ..... **xi**

BAB I PENDAHULUAN **1**

1.1. Latar Belakang 1

1.2. Perumusan Masalah 3

1.3. Batasan Masalah 3

1.4. Tujuan Masalah 3

1.5. Sistematika Penulisan 4

BAB II TINJAUAN PUSTAKA **6**

2.1. Penelitian Terkait 6

2.2. High Availability 8

2.3. Failover 8

2.4. Load balancing 9

2.5. Docker 9

2.6. HAProxy 9

2.7. Keepalived 10

2.8. ISO/IEC 25010:2011 10

BAB III PERANCANGAN DAN REALISASI 12

3.1. Rancangan Penelitian 12

3.2. Tahapan Penelitian..... 12

3.3. Objek Penelitian 14

BAB IV HASIL DAN PEMBAHASAN..... 15

4.1. Analisis Kebutuhan 15

4.1.1. Kebutuhan Fungsional 15

4.1.1.1. Kebutuhan Fungsional Topologi *High Availability* Web Server dan Layanan 15

4.1.1.2. Kebutuhan Fungsional *Load balancing*..... 15

4.1.2. Kebutuhan Non-fungsional 16

4.1.2.1. Analisis Perangkat Lunak..... 16

4.1.2.2. Analisis Perangkat Keras 17

4.1.2.3. Analisis Mesin Virtual 18

4.2. Perancangan Sistem..... 18

4.2.1. Arsitektur Sistem..... 19

4.2.2. Flow Diagram Sistem..... 20

4.2.3. Flow Diagram Keepalived 21

4.2.4. Flow Diagram *Load balancing* 22

4.2.4.1. Flow Diagram Algoritma Load Balancing Round Robin 23

4.2.4.2. Flow Diagram Algoritma Load Balancing Least connection .. 25

4.2.5. Perancangan *Website* 26

4.2.5.1. Perancangan MySQL..... 26

4.2.5.2. Perancangan MinIO 26

4.2.5.3. Perancangan FastAPI..... 27

4.2.5.4. Perancangan Flask 28

4.3. Implementasi Sistem 30
4.3.1. Implementasi *Website Dashboard* Sirip Hiu 30
4.3.1.1. Implementasi FastAPI 30
4.3.1.2. Implementasi Flask 31
4.3.2. Implementasi dan Konfigurasi Infrastruktur *High Availability* 32
4.3.2.1. Konfigurasi Jaringan pada Mesin Virtual 32
4.3.2.2. Konfigurasi Docker Swarm 36
4.3.2.3. Konfigurasi Deployment Layanan 37
4.3.2.4. Konfigurasi HAProxy 38
4.3.2.5. Konfigurasi Keepalived 43
4.4. Pengujian 46
4.4.1. Deskripsi Pengujian 46
4.4.2. Prosedur Pengujian 47
4.4.2.1. Uji Fungsionalitas *Website* 47
4.4.2.2. Uji Failover, Failback, dan Downtime 48
4.4.2.3. Uji Beban (Load & *Stress Test*) 48
4.4.2.4. Uji Load balancing 48
4.4.2.5. Uji Throughput dan Response Time 49
4.4.2.6. Uji *Resource* Sistem 49
4.4.3. Data Hasil Pengujian 49
4.4.3.1. Hasil Uji Fungsionalitas *Website* 49
4.4.3.2. Hasil Uji Failover, Failback, dan Downtime 50
4.4.3.3. Uji Downtime 51
4.4.3.4. Hasil Uji Fungsi *Load balancing* 52
4.4.3.5. Hasil Uji Beban (*Stress test*) 54
4.4.3.6. Hasil Uji Throughput dan Response Time 54

4.4.3.7. Hasil Monitoring Resource (VM Worker 1 & Worker 2) 55
4.4.4. Analisis Data/Evaluasi Pengujian 57
4.4.4.1. Analisis Pengujian *Failover* 57
4.4.4.2. Analisis Pengujian *Failback* 58
4.4.4.3. Analisis Pengujian *Downtime* 59
4.4.4.4. Analisis Pengujian Distribusi *Load balancing* 59
4.4.4.5. Analisis Pengujian Performa Algoritma *Load balancing* 61
4.4.4.6. Analisis Hasil Pengujian *Throughput* dan *Response Time* 62
4.4.4.7. Analisis Hasil Pengujian *Stress Test* 63
4.4.4.8. Analisis Hasil Pengujian Pemakaian *Resource* 64
BAB V PENUTUP 65
5.1. Kesimpulan 65
5.2. Saran 65
DAFTAR PUSTAKA 67
DAFTAR RIWAYAT HIDUP 69
LAMPIRAN 70

DAFTAR GAMBAR

Gambar 3. 1 Tahapan Penelitian 12
Gambar 4. 1 Arsitektur Sistem High Availability 19
Gambar 4. 2 Alur Keseluruhan Sistem High Availability 20
Gambar 4. 3 Alur Sistem Failover Otomatis oleh Keepalived 21
Gambar 4. 4 Alur Sistem Load balancing oleh HAProxy 22
Gambar 4. 5 Alur Algoritma Round Robin 23
Gambar 4. 6 Alur Algoritma Least connection 25
Gambar 4. 7 Dokumentasi FastAPI 30
Gambar 4. 8 Halaman Dashboard Website 31
Gambar 4. 9 Daftar Node pada Docker Cluster Swarm 37
Gambar 4. 10 Hasil curl melalui IP Load balancer 1 42
Gambar 4. 11 Hasil curl melalui IP Load balancer 2 43
Gambar 4. 12 Hasil curl melalui VIP Keepalived 46
Gambar 4. 13 Grafik Data Pengujian Failover 57
Gambar 4. 14 Grafik Data Pengujian Failback 58
Gambar 4. 15 Grafik Data Pengujian Downtime 59
Gambar 4. 16 Distribusi Load balancing Algoritma Round Robin 59
Gambar 4. 17 Distribusi Load balancing Algoritma Least connection 60
Gambar 4. 18 Grafik Hasil Pengujian Performa Algoritma Load Balancing 61
Gambar 4. 19 Grafik Hasil Pengujian Throughput dan Response Time 62
Gambar 4. 20 Grafik Data Hasil Uji Stress Test 63
Gambar 4. 21 Grafik Data Hasil Uji Pemakaian Resource 64

DAFTAR TABEL

Tabel 2. 1 Penelitian Terkait 6
Tabel 4. 1 Analisis Perangkat Lunak 16
Tabel 4. 2 Analisis Perangkat Keras 17
Tabel 4. 3 Analisis Mesin Virtual 18
Tabel 4. 4 Perancangan dari Implementasi Basis Data MySQL 26
Tabel 4. 5 Struktur Layanan dari Deployment MinIO 26
Tabel 4. 6 Daftar Endpoint FastAPI 27
Tabel 4. 7 Menu dan Navigasi dari Dashboard Website 29
Tabel 4. 8 Pengaturan Adapter Jaringan Mesin Virtual 32
Tabel 4. 9 Pengaturan Alamat IP Mesin Virtual 33
Tabel 4. 10 Perintah Edit File Netplan 33
Tabel 4. 11 Konfigurasi IP Static VM-LoadBalancer 1 33
Tabel 4. 12 Konfigurasi IP Static VM-LoadBalancer 2 34
Tabel 4. 13 Konfigurasi IP Static VM-DockerSwarmManager 34
Tabel 4. 14 Konfigurasi IP Static VM-DockerSwarmWorker 1 35
Tabel 4. 15 Konfigurasi IP Static VM-DockerSwarmWorker 2 35
Tabel 4. 16 Perintah untuk Menerapkan Konfigurasi Netplan 35
Tabel 4. 17 Perintah untuk Masuk ke Konfigurasi Hosts 36
Tabel 4. 18 Konfigurasi Hosts 36
Tabel 4. 19 Perintah untuk Inisiasi Docker Swarm 36
Tabel 4. 20 Perintah untuk Join ke Docker Swarm Cluster 36
Tabel 4. 21 Perintah untuk Menampilkan Daftar Node dalam Cluster Swarm 37
Tabel 4. 22 Layanan dalam Docker 37
Tabel 4. 23 Perintah untuk Deploy Layanan 37
Tabel 4. 24 Perintah untuk Menampilkan Layanan yang Berjalan di Docker 38
Tabel 4. 25 Perintah untuk Menjalankan Instalasi HAProxy 38
Tabel 4. 26 Perintah untuk Mengubah Konfigurasi HAProxy 38
Tabel 4. 27 Konfigurasi HAProxy 38
Tabel 4. 28 Perintah untuk Menjalankan Layanan HAProxy 42
Tabel 4. 29 Perintah untuk Mengecek Status HAProxy 42

Tabel 4. 30 Perintah untuk Menjalankan Instalasi Keepalived 43
Tabel 4. 31 Perintah untuk Mengubah Konfigurasi Keepalived 43
Tabel 4. 32 Konfigurasi Keepalived Master 44
Tabel 4. 33 Konfigurasi Keepalived Backup 44
Tabel 4. 34 Perintah untuk Menjalankan Layanan Keepalived..... 45
Tabel 4. 35 Perintah untuk Mengecek Status Keepalived..... 46
Tabel 4. 36 Data Hasil Uji Fungsionalitas Website 49
Tabel 4. 37 Data Hasil Uji Failover 50
Tabel 4. 38 Data Hasil Uji Failback 51
Tabel 4. 39 Data Hasil Uji Downtime..... 52
Tabel 4. 40 Data Hasil Pengujian Load balancing menggunakan Algoritma Round Robin..... 52
Tabel 4. 41 Data Hasil Pengujian Load balancing menggunakan Algoritma Least connection 53
Tabel 4. 42 Data Hasil Pengujian Performa Algoritma Round Robin 53
Tabel 4. 43 Data Hasil Pengujian Algoritma Least connection 54
Tabel 4. 44 Data Hasil Uji Beban..... 54
Tabel 4. 45 Data Hasil Uji Throughput dan Response Time..... 55
Tabel 4. 46 Data Hasil Uji Penggunaan Resource 56

BAB I
PENDAHULUAN

1.1. Latar Belakang

Hiu merupakan salah satu spesies laut yang memiliki peran penting dalam menjaga keseimbangan ekosistem laut. Lebih dari 500 spesies hiu ditemukan di seluruh lautan di dunia, dan 117 di antaranya ditemukan di Indonesia (Dermawan & Sadili, 2015). Banyaknya spesies hiu di Indonesia menjadikan negara ini sebagai salah satu eks*port*ir hiu terbesar di dunia (Lack & Sant, 2011).

Sirip punggung hiu merupakan bagian tubuh hiu yang paling banyak dijual karena memiliki nilai ekonomi tinggi di pasar internasional. Sirip punggung hiu digunakan dalam berbagai produk, seperti sup sirip hiu yang menjadi hidangan mewah di beberapa negara. Data menunjukkan bahwa volume ekspor sirip punggung hiu dari Indonesia terus mengalami peningkatan setiap tahunnya, mencerminkan tingginya permintaan pasar internasional (Rusandi et al., 2019).

Saat ini, identifikasi sirip punggung hiu umumnya dilakukan secara visual dan manual. Identifikasi visual menggunakan panduan taksonomi adalah metode yang paling sederhana dan ekonomis untuk mengidentifikasi spesies hiu dan pari. Namun, metode ini membutuhkan pelatihan khusus untuk memastikan akurasi. Pelatihan ini diperlukan karena identifikasi sirip punggung hiu memerlukan keahlian dalam membedakan ciri-ciri morfologis yang sering kali sangat mirip antar spesies (Rigby et al., 2019).

Berdasarkan wawancara dengan Bapak Deden Solihin, yang merupakan seorang Analis Konservasi dan Rehabilitasi Wilayah Pesisir di Loka Pengelolaan Sumber Daya Pesisir Laut Serang, keterbatasan tersebut juga dialami, yang mana memengaruhi proses identifikasi sirip punggung hiu. Hingga saat ini, proses tersebut masih mengandalkan metode sampling, yaitu pemeriksaan terhadap sebagian kecil dari total produk yang diperiksa, hal tersebut dilakukan karena keterbatasan waktu dan sumber daya manusia. Namun, pendekatan ini sering kali tidak cukup representatif dan berpotensi

menimbulkan kesalahan, terutama dalam memastikan bahwa spesies yang dilindungi tidak masuk ke dalam rantai perdagangan. Hal tersebut menunjukkan kebutuhan akan teknologi yang dapat mendukung identifikasi dan pengelolaan data terkait jenis sirip punggung hiu secara lebih cepat dan akurat.

Pengembangan sistem yang mengintegrasikan beberapa elemen teknologi, seperti sistem *conveyor* otomatis dan *computer vision*, dilakukan untuk menangani identifikasi jenis sirip punggung hiu dalam volume besar secara efisien. Sistem *conveyor* otomatis memungkinkan pemisahan sirip punggung hiu secara cepat dan terorganisir, sementara *computer vision*, yang didukung oleh algoritma kecerdasan buatan, dapat meningkatkan akurasi dalam proses identifikasi dengan meminimalkan kesalahan manusia. Dengan kombinasi kedua teknologi ini, proses identifikasi dan pemisahan jenis sirip punggung hiu dapat dilakukan lebih cepat, akurat, dan efisien, bahkan ketika menghadapi data dalam jumlah besar.

Untuk memaksimalkan hasil dari sistem identifikasi sirip punggung hiu yang telah diproses oleh sistem *conveyor* otomatis dan *computer vision*, diperlukan sebuah sistem informasi berbasis *website* yang tidak hanya menyajikan data secara visual (melalui grafik dan tabel), tetapi juga menjamin ketersediaan yang tinggi. Dalam konteks ini, penerapan teknologi *load balancing* menjadi krusial untuk mengelola trafik yang tinggi dan mencegah terjadinya *single point of failure*.

Beberapa penelitian terdahulu, seperti studi yang dilakukan oleh (Putra et al., 2020) mengenai "Implementasi *High Availability Cluster* Web Server Menggunakan *Virtualisasi Container* Docker", telah mengkaji distribusi beban trafik pada *server web* menggunakan HAProxy dan Docker. Namun, penelitian tersebut umumnya tidak mengintegrasikan mekanisme *failover* otomatis untuk mengatasi kegagalan *node*. Begitu pula, studi perbandingan teknologi *failover* yang dilakukan oleh (Pratama, 2021) dalam penelitiannya yang berjudul "Perbandingan Kinerja Teknologi *Failover* Berbasis Klaster (Heartbeat) dan Teknologi *Failover* Berbasis Jaringan (Keepalived)" menunjukkan bahwa mekanisme *failover* otomatis dapat mengurangi *downtime*, namun belum

diaplikasikan dalam lingkungan *container* modern. Penelitian yang akan dilakukan dalam skripsi ini berfokus pada pengembangan sistem *load balancing* dengan penekanan pada mekanisme *failover* otomatis menggunakan Docker Swarm, HAProxy, dan Keepalived. Pendekatan ini diharapkan tidak hanya mendistribusikan trafik secara efisien, tetapi juga secara otomatis mengalihkan *Virtual IP* ketika terjadi kegagalan pada salah satu *node*, sehingga memastikan layanan tetap tersedia dan handal.

1.2. Perumusan Masalah

Berdasarkan dari latar belakang tersebut, masalah yang perlu diatasi antara lain:

1. Bagaimana merancang dan mengimplementasikan sistem *load balancing* yang dilengkapi dengan mekanisme *failover* otomatis menggunakan Docker, HAProxy dan Keepalived untuk memastikan ketersediaan dan performa layanan *web*?
2. Bagaimana mekanisme *failover* otomatis dapat meningkatkan ketersediaan dan keandalan sistem, terutama dalam mengatasi kegagalan *node* dan mencegah terjadinya *single point of failure*?

1.3. Batasan Masalah

Fokus pada penelitian ini dibatasi pada:

1. Penggunaan Docker Swarm, HAProxy, dan Keepalived untuk mendistribusikan trafik dan menjamin ketersediaan layanan *web*.
2. Pengujian dan evaluasi dilakukan dalam lingkungan pengujian skala kecil, sehingga aspek implementasi di lingkungan produksi berskala besar tidak dibahas.
3. Penelitian ini membatasi pembahasan pada penggunaan Docker Swarm, HAProxy, dan Keepalived sebagai komponen utama, dan tidak membahas solusi *load balancing* alternatif lainnya.

1.4. Tujuan Masalah

Tujuan dari penelitian ini adalah:

1. Mengimplementasikan sistem *load balancing* yang mengintegrasikan Docker Swarm, HAProxy, dan Keepalived dengan penekanan pada mekanisme *failover* otomatis untuk menjaga ketersediaan layanan.
2. Mengukur dan menganalisis ketersediaan serta *downtime* sistem saat terjadi kegagalan pada salah satu *node*.

Manfaat dari penelitian ini adalah:

1. Penelitian ini memberikan solusi teknis untuk meningkatkan ketersediaan dan kehandalan sistem melalui penerapan *load balancing* dan mekanisme *failover* otomatis, yang relevan bagi pengembangan sistem informasi berbasis *web*.
2. Dengan optimasi distribusi trafik, penelitian ini diharapkan dapat mengurangi *downtime* dan memastikan performa sistem tetap optimal dalam kondisi beban tinggi, yang berpotensi meningkatkan efisiensi operasional di sektor aplikasi berbasis data.

1.5. Sistematika Penulisan

Berikut adalah sistematika penulisan yang digunakan dalam penyusunan proposal penelitian ini:

1. BAB I PENDAHULUAN

   Bab ini membahas latar belakang penelitian, merumuskan masalah yang muncul dari latar belakang tersebut, menjelaskan batasan masalah dalam penelitian, serta menguraikan tujuan dan manfaat dari pengembangan sistem *load balancing* dan *High Availability* untuk sistem *web* dengan menggunakan Docker, HAProxy, dan Keepalived.
2. BAB II TINJAUAN PUSTAKA

   Membahas teori dan konsep yang relevan, seperti *load balancing*, *failover* otomatis, Docker Swarm, HAProxy, Keepalived, dan penelitian terdahulu yang berkaitan dengan topik ini. Kajian literatur juga mencakup perbandingan antara solusi yang telah ada dan sistem yang akan dikembangkan.
3. BAB III PERENCANAAN DAN REALISASI

Bab ini menjelaskan metode penelitian yang digunakan, yaitu eksperimental, rancangan penelitian, tahapan pelaksanaan, serta objek penelitian yang berfokus pada sistem *failover* otomatis.

4. BAB IV HASIL DAN PEMBAHASAN

   Bab ini menjelaskan metodologi, rancangan arsitektur sistem, flowchart dari tiap sistem, konfigurasi dari mesin virtual, konfigurasi Docker Swarm, HAProxy, serta konfigurasi Keepalived.

5. BAB V PENUTUP

   Bab ini menyajikan kesimpulan dari hasil penelitian serta memberikan saran-saran yang relevan berdasarkan hasil penelitian yang telah dilakukan.

BAB II
TINJAUAN PUSTAKA

2.1. Penelitian Terkait

Tabel 2. 1 Penelitian Terkait

Judul dan Penulis	**Hasil**	**Persamaan**	**Perbedaan**
Perbandingan Kinerja Teknologi *Failover* Berbasis Klaster (Heartbeat) Dengan Teknologi *Failover* Berbasis Jaringan (Keepalived) (Pratama, 2021)	Penelitian tersebut menitikberatkan pada perbandingan *failover* antara Heartbeat dan Keepalived. Hasil pengujian menunjukkan bahwa kedua teknologi mampu mencapai *downtime* 0 detik, serta *failback* time yang sangat rendah. Secara umum, Keepalived cenderung memberikan performa *failback* yang	Penelitian tersebut menekankan implementasi teknologi *failover* menggunakan Keepalived dan HAProxy, serupa dengan penelitian ini yang juga menggunakan kedua komponen tersebut sebagai solusi untuk meningkatkan ketersediaan layanan *web*. Keduanya menerapkan konsep *High Availability*	Perbedaannya terdapat pada lingkungan virtualisasi dan cakupan teknologi *failover*. Penelitian ini menerapkan Docker *container* sebagai basis infrastruktur, sedangkan penelitian Dian Pratama masih menggunakan VirtualBox dengan sistem mesin virtual tradisional. Selain itu, penelitian ini hanya memfokuskan pada Keepalived sebagai komponen utama *failover*, sementara Dian Pratama membandingkan performa dua

Judul dan Penulis	Hasil	Persamaan	Perbedaan
	lebih cepat (contoh skenario 1: 0,48 detik) dibandingkan Heartbeat (0,68 detik). Hal ini memperlihatkan bahwa Keepalived merupakan solusi *failover* yang efektif dalam meningkatkan ketersediaan sistem, baik pada *web server* maupun *load balancer*	dengan tujuan utama meminimalkan *downtime* serta menjamin kontinuitas layanan ketika terjadi gangguan pada salah satu *node*.	mekanisme *failover*, yaitu Heartbeat dan Keepalived, dalam konteks *High Availability* baik pada *web server* maupun *load balancer*.
Implementasi *High Availability Cluster* Web Server Menggunakan *Virtualisasi Container* Docker (Putra et al., 2020)	HAProxy *load balancing* pada Docker *Cluster* sukses meningkatkan *throughput* dan mengurangi *response time*. *Least connection* lebih unggul dibanding *round*	Penelitian tersebut menerapkan konsep *clustering web server* berbasis Docker *container* dan menggunakan HAProxy sebagai *load balancer*. Hal	Perbedaannya terletak pada lingkup penelitian. Aldi Putra lebih memfokuskan penelitian pada analisis performa *load balancing*, dengan menguji algoritma *round robin* dan *least connection* tanpa mengimplementasikan mekanisme *failover*.

Judul dan Penulis	Hasil	Persamaan	Perbedaan
	robin, *CPU utilization* juga lebih kecil dengan *clustering*.	tersebut sejalan dengan penelitian ini yang juga memanfaatkan teknologi Docker sebagai platform virtualisasi ringan dan efisien dalam membangun arsitektur sistem informasi berbasis *web*.	Berbeda dengan itu, penelitian ini menitikberatkan pada implementasi *failover* otomatis, di mana layanan tetap berjalan secara otomatis ketika terjadi kegagalan pada salah satu *node*, sehingga menjamin ketersediaan sistem informasi berbasis *web* secara menyeluruh.

2.2. High Availability

Menurut (B. C. Lahti & Peterson, 2005) *High Availability* adalah konsep yang memastikan bahwa suatu sistem atau aplikasi tetap menjalankan layanan dalam jangka waktu singkat setelah mengalami suatu kegagalan. *High Availability* perlu diterapkan di setiap sistem yang dibuat dengan tujuan untuk mencegah masalah performa dan *downtime* yang dapat mengganggu pengguna dari suatu sistem (Cross et al., 2003).

2.3. Failover

Failover merupakan salah satu mekanisme High Availibility yang menerapkan proses "*failover*" atau pengalihan dari *server* utama (aktif) ke *server* sekunder (pasif). Setiap *server* dikonsepkan berada dalam suatu *cluster* dan disebut sebagai *node*. Mekanisme ini menggunakan teknologi aktif/pasif yang membutuhkan satu atau lebih *server* cadangan yang *standby* jika terjadi

kegagalan di *server* aktif. Jika terjadi kegagalan pada *node* aktif, *node* pasif akan menjadi aktif dan menjalankan semua aktivitas yang sebelumnya dilakukan oleh *node* aktif (Hannifin et al., 2010).

2.4. Load balancing

Load balancing adalah metode pendistribusian beban kerja antara dua sistem atau lebih. *Load balancing* sering digunakan untuk membagi trafik jaringan di antara beberapa *server*. Hal tersebut dapat mengurangi beban pada setiap *server* dan membuat kinerja *server* lebih efisien dan mengurangi latensi (Cloudflare, 2020).

2.5. Docker

Docker adalah platform yang memungkinkan proses pengembangan, pengelolaan, dan distribusi aplikasi berada dalam satu lingkungan terisolasi yang disebut *container*. *Container* ini memungkinkan aplikasi berjalan secara konsisten di berbagai lingkungan tanpa perlu khawatir tentang perbedaan konfigurasi sistem, sehingga memastikan konsistensi di berbagai lingkungan (Docker, 2024).

Dalam penelitian ini, salah satu fitur dari docker, yaitu docker swarm digunakan untuk mengelola tiap *cluster web server* yang berjalan. Docker Swarm adalah fitur manajemen klaster dan orkestrasi yang terintegrasi dalam Docker Engine. Fitur ini memungkinkan pengguna untuk mengelola sekelompok Docker Engine yang disebut "swarm". Swarm ini terdiri dari beberapa *node* Docker yang berjalan dalam mode Swarm, yang dapat berperan sebagai *manager* atau *worker*. Fitur ini menawarkan berbagai kemampuan seperti skalabilitas, jaringan *multi-host*, *service discovery*, dan *load balancing* (Docker, 2016).

2.6. HAProxy

HAProxy adalah perangkat lunak open-source yang dapat membantu mendistribusikan trafik secara merata ke beberapa *server*, sehingga aplikasi tetap berjalan lancar meskipun terjadi lonjakan trafik atau kegagalan pada salah satu *server*. Dengan menggunakan HAProxy, aliran data dapat diatur secara efisien menggunakan teknik *load balancing*. Selain itu, HAProxy juga berfungsi sebagai reverse proxy yang membantu mengamankan dan mengelola

aliran data (HAProxy, 2021). Pada penelitian ini, HAProxy digunakan sebagai *load balancer* yang nantinya akan mendistribusikan trafik ke tiap *cluster* docker yang menjalankan *web server*, sehingga beban yang masuk akan terbagi rata.

2.7. Keepalived

Keepalived adalah perangkat lunak *open-source* yang dirancang untuk meningkatkan ketersediaan layanan dan melakukan *load balancing* pada sistem berbasis Linux. Dengan menggunakan *Virtual Router Redundancy Protocol* (VRRP), Keepalived memungkinkan pengelolaan *IP virtual* dan *health checking* secara berkala pada *server* untuk memastikan status *server* tersebut (Keepalived, 2020). Penelitian ini memanfaatkan Keepalived untuk menjaga kestabilan dengan menjalankan skema *failover*, yaitu jika satu *server* gagal, *server* lain bisa segera mengambil alih guna memastikan layanan tetap berjalan lancar tanpa gangguan signifikan.

2.8. ISO/IEC 25010:2011

ISO/IEC 25010:2011 merupakan standar internasional yang mendefinisikan model kualitas perangkat lunak yang terdiri dari delapan karakteristik utama (ISO/IEC 25010, 2011). Salah satu karakteristik tersebut adalah *performance efficiency*, yang sangat relevan dengan pengujian sistem informasi berbasis web seperti pada penelitian ini.

Subkarakteristik dari *performance efficiency* meliputi:

1. *Time Behavior*: Mengukur sejauh mana sistem mampu merespons permintaan dalam waktu yang sesuai. Metrik yang digunakan antara lain *response time, latency*, dan *processing time.*
2. *Resource Utilization*: Sejauh mana sistem menggunakan sumber daya (CPU, memori, *bandwidth*) secara efisien.
3. *Capacity*: Menggambarkan kemampuan sistem dalam menangani volume beban tertentu, biasanya diukur melalui *throughput* atau jumlah permintaan yang dapat dilayani per satuan waktu.

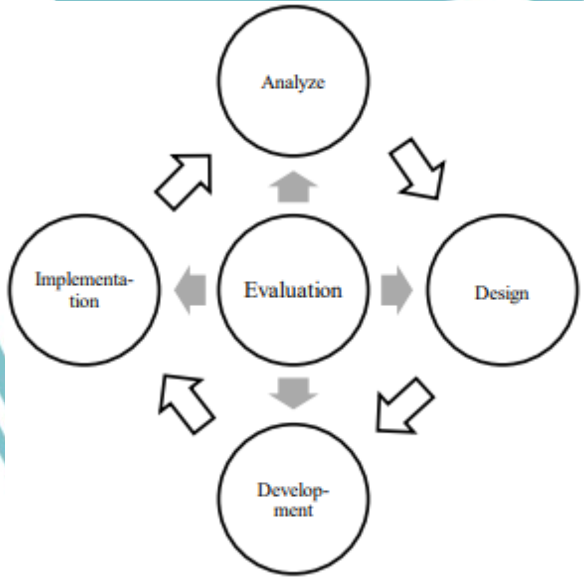
Standar ini tidak memberikan nilai yang pasti, tetapi menyediakan kerangka kerja yang evaluatif. Oleh karena itu, nilai acuan pada penelitian ini diadaptasi dari praktik industri dan referensi akademik.

BAB III
PERANCANGAN DAN REALISASI

3.1. Rancangan Penelitian

Penelitian ini menggunakan pendekatan kuantitatif dengan jenis riset eksperimental untuk mengukur kinerja mekanisme *failover* otomatis secara objektif. Fokus utama penelitian ini adalah mengukur kinerja dan keandalan mekanisme *failover* otomatis. Prototipe sistem dikembangkan dengan mengintegrasikan Docker Swarm, HAProxy, dan Keepalived untuk mengotomatisasi pengalihan trafik melalui *Virtual IP* saat terjadi kegagalan pada salah satu *node*. Pengujian dari sistem ini dilakukan dalam lingkungan virtual yang terkontrol dengan mensimulasikan kondisi pada saat mengalami kegagalan. Data metrik seperti *downtime*, *response time*, serta kecepatan respon *failover* dikumpulkan dan dianalisis secara statistik untuk menilai efektivitas mekanisme *failover* dalam menjaga ketersediaan layanan.

3.2. Tahapan Penelitian



Gambar 3. 1 Tahapan Penelitian

Untuk tahapan penelitian yang akan dilakukan adalah sebagai berikut:

1. **Analisis Masalah**

   Pada tahap ini, dilakukan studi literatur dan identifikasi kebutuhan untuk mengatasi masalah ketersediaan dalam sistem informasi berbasis *website*.

Data dikumpulkan dari jurnal, artikel, dan dokumentasi teknis guna menetapkan kebutuhan sistem secara spesifik, terutama terkait dengan mekanisme *failover* dan *load balancing*.

2. **Desain Sistem**

   Tahap desain melibatkan penyusunan arsitektur sistem yang mengintegrasikan Docker Swarm, HAProxy, dan Keepalived. Diagram arsitektur dan flow diagram disusun untuk menggambarkan infrastruktur, alur kerja sistem *failover*, serta bagaimana sistem mendistribusikan trafik dan melakukan pengalihan *Virtual IP* jika terjadi kegagalan.

3. **Pengembangan Sistem**

   Pada tahap pengembangan, prototipe sistem dibuat berdasarkan desain yang telah disusun. Konfigurasi awal dilakukan pada Docker Swarm, HAProxy, dan Keepalived untuk membangun lingkungan uji yang mendukung mekanisme *failover* dan *load balancing*.

4. **Implementasi**

   Implementasi dimulai dengan *deployment container* melalui Docker Swarm. HAProxy dikonfigurasi untuk mendistribusikan trafik secara merata ke *backend server*, sedangkan Keepalived diatur untuk memastikan *failover* otomatis dengan memindahkan *Virtual IP* jika *load balancer* utama mengalami kegagalan.

5. **Evaluasi**

   Sistem dievaluasi melalui pengujian *failover* yang dilakukan untuk mengukur durasi pergantian dari *node backup* menjadi *master*, uji *downtime* yang dilakukan untuk mengukur waktu turunnya kinerja *server* saat salah satu *node* Keepalived mati, uji *failback* dilakukan untuk mengukur durasi yang dibutuhkan *node master* yang mati untuk mengambil alih posisinya, dan juga penggunaan sumber daya dari docker *container* yang berjalan.

6. **Penulisan Laporan Penelitian**

   Menyusun laporan penelitian yang mencakup analisis masalah, perancangan sistem, realisasi melalui integrasi Docker Swarm, dan HAProxy, Keepalived. Serta menyusun kesimpulan terkait penelitian ini.

3.3. Objek Penelitian

Fokus penelitian ini adalah pada sistem *failover* otomatis yang diimplementasikan pada infrastruktur berbasis Docker untuk mendukung sistem informasi berbasis *web*. Sistem ini mengintegrasikan HAProxy sebagai *load balancer* dan Keepalived sebagai mekanisme *failover* otomatis guna memastikan ketersediaan layanan meskipun terjadi kegagalan pada salah satu komponen. Penelitian ini mengkaji konfigurasi, performa, dan keandalan sistem dalam mengelola trafik serta proses *failover* antara *load balancer* utama dan cadangan.

BAB IV
HASIL DAN PEMBAHASAN

4.1. Analisis Kebutuhan

4.1.1. Kebutuhan Fungsional

4.1.1.1. Kebutuhan Fungsional Topologi *High Availability* Web Server dan Layanan

1. Setiap VM *worker* mampu menjalankan layanan aplikasi *web* (FastAPI dan Flask) serta layanan penyimpanan objek (MinIO) dan basis data (MySQL) secara terintegrasi.
2. Ketika salah satu layanan mengalami kegagalan, baik itu layanan aplikasi *web*, maupun penyimpanan objek, Docker Swarm secara otomatis akan menjalankan ulang atau memindahkan replika layanan tersebut ke *node* lain yang masih aktif, sehingga layanan tetap dapat diakses oleh pengguna.
3. Jika seluruh VM *worker* tidak aktif, maka layanan aplikasi *web*, penyimpanan objek, dan basis data tidak dapat diakses oleh pengguna.

4.1.1.2. Kebutuhan Fungsional *Load balancing*

1. Setiap VM *Load balancer* yang telah diimplementasikan HAProxy dan Keepalived mampu mendistribusikan trafik pengguna ke VM *worker* secara efisien dengan algoritma *least connection* dan *round robin*.
2. Ketika salah satu VM *Load balancer* mati, pengguna tetap dapat mengakses layanan aplikasi *web* karena *failover* otomatis akan mengalihkan trafik ke *Load balancer* cadangan melalui *Virtual IP* (*VIP*).
3. Jika seluruh *Load balancer* dalam kondisi mati, pengguna tidak dapat mengakses layanan aplikasi *web* karena tidak ada *entry point* yang aktif untuk menerima *request* dari *client*.

4.1.2. Kebutuhan Non-fungsional

4.1.2.1. Analisis Perangkat Lunak

Berbagai aplikasi dan sistem operasi dipilih untuk mendukung proses pengembangan dan *deployment* dari layanan pada sistem *High Availability*. Daftar lengkap perangkat lunak beserta deskripsinya disajikan pada Tabel 4.1.

Tabel 4. 1 Analisis Perangkat Lunak

No.	Kebutuhan Perangkat Lunak	Fungsi
1.	Docker	Sebagai alat untuk mengelola *cluster* dari *web server*
2.	VirtualBox	Sebagai lingkungan untuk menjalankan mesin virtual yang diperlukan dalam *deployment web server*.
3.	Ubuntu Server 24.04.2 LTS	Sebagai sistem operasi berbasis Linux untuk menjalankan layanan *web server*.
4.	Keepalived	Untuk mengatur *failover* otomatis dan menjaga ketersediaan layanan dengan memonitor status *server*.
5.	HAProxy	Sebagai *load balancer* yang mendistribusikan trafik ke *server backend* secara efisien.
6	MySQL	Sebagai sistem manajemen basis data relasional yang mengelola, menyimpan, dan menyediakan data aplikasi *web* secara terstruktur.

No.	Kebutuhan Perangkat Lunak	Fungsi
7	MinIO	Sebagai penyimpanan objek (*object storage*) untuk menyimpan data dari sirip punggung hiu.
8	FastAPI	Sebagai framework untuk membangun API dari *dashboard website* sirip punggung hiu.
9	Flask	Sebagai framework *web* WSGI yang digunakan untuk membangun *backend* dari *dashboard website* sirip punggung hiu.

4.1.2.2. Analisis Perangkat Keras

Untuk memastikan sistem *High Availability* dapat beroperasi dengan baik, diperlukan perangkat keras dengan spesifikasi yang memadai. Seluruh proses implementasi dijalankan pada satu unit *laptop* yang berfungsi sebagai *host* untuk virtualisasi. Detail spesifikasi perangkat keras *laptop* yang digunakan dapat dilihat pada Tabel 4.2

Tabel 4. 2 Analisis Perangkat Keras

No.	Deskripsi	Spesifikasi
1.	CPU	AMD Ryzen 5 3500U
2.	RAM	12 GB
3.	SSD	256 GB
4.	Sistem Operasi	Windows 10 22H2

4.1.2.3. Analisis Mesin Virtual

Spesifikasi mesin virtual dirancang agar mampu memenuhi kebutuhan operasional setiap layanan dalam sistem *High Availability*. Detail spesifikasi masing-masing VM disajikan pada Tabel 4.3.

Tabel 4. 3 Analisis Mesin Virtual

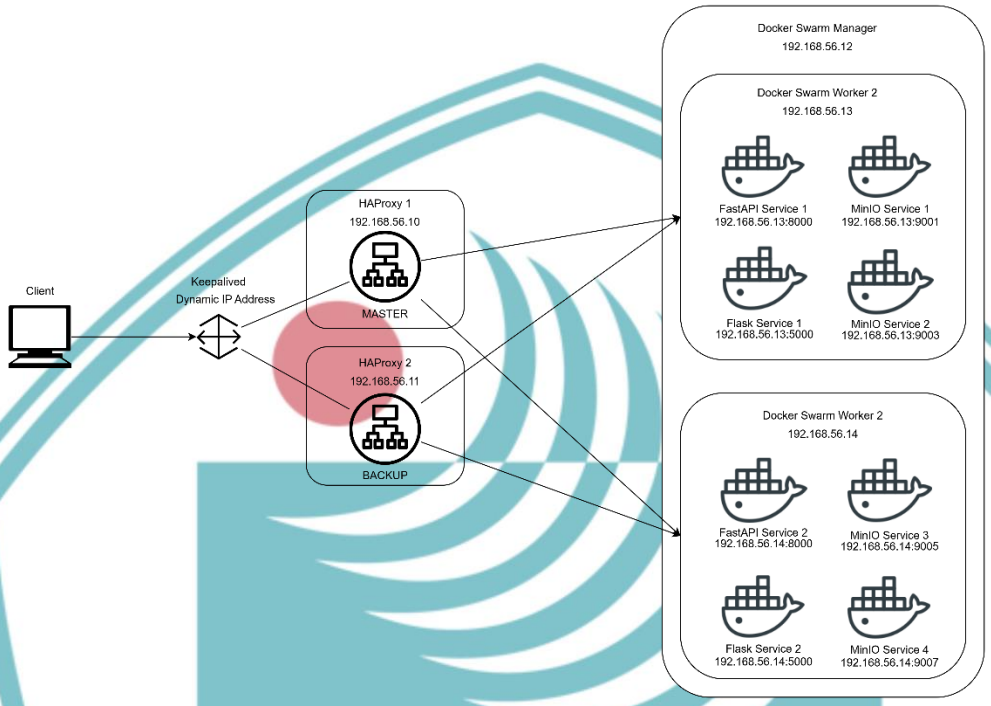
No.	Nama Mesin Virtual	*CPU (Core)*	*RAM (GB)*	*Disk Storage (GB)*	Keterangan
1.	VM-LoadBalancer 1	1	1	10	HAProxy & Keepalived
2.	VM-LoadBalancer 2	1	1	10	HAProxy & Keepalived
3.	VM-DockerSwarmManager	2	1	30	Docker Swarm, Orchestrator, MySQL
4	VM-DockerSwarmWorker 1	2	2	30	Docker Swarm, MySQL, FastAPI, Flask, MinIO
5	VM-DockerSwarmWorker 2	2	2	30	Docker Swarm, MySQL, FastAPI, Flask, MinIO

4.2. Perancangan Sistem

Perancangan sistem bertujuan untuk merancang arsitektur dan alur kerja yang mampu mempertahankan ketersediaan layanan melalui mekanisme *failover* otomatis dan *load balancing*. Tahapan ini berfokus pada pengintegrasian setiap

komponen agar sistem tetap dapat beroperasi meskipun terjadi kegagalan pada salah satu *node* atau layanan.

4.2.1. Arsitektur Sistem

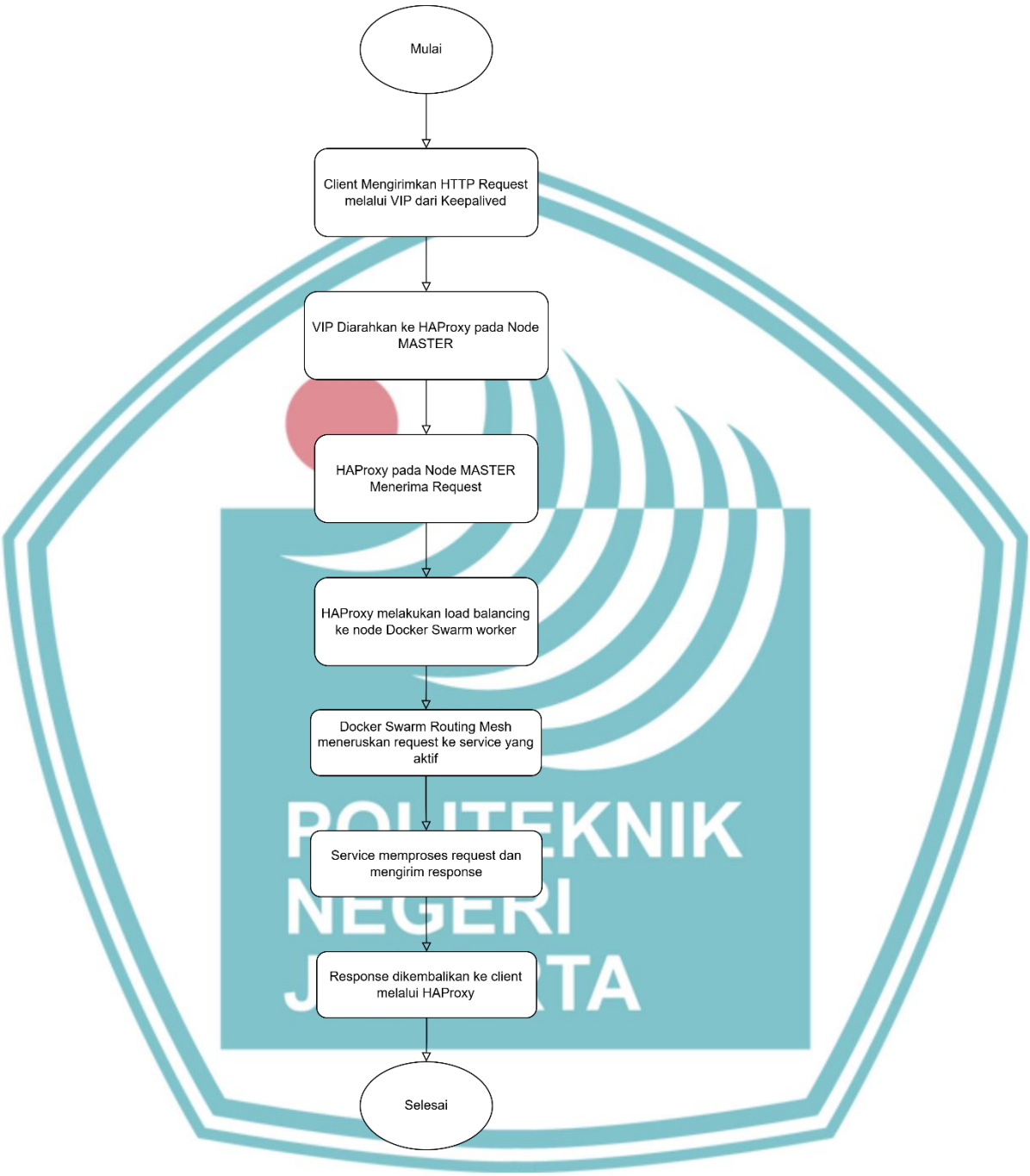


Gambar 4. 1 Arsitektur Sistem *High Availability*

Topologi arsitektur dalam penelitian ini terdiri dari beberapa lapisan yang saling terintegrasi untuk memastikan *High Availability* melalui mekanisme *failover* otomatis. Pada lapisan terdepan terdapat *client*, yaitu pengguna yang mengirimkan *request* ke sistem. Permintaan dari *client* diarahkan ke *Virtual IP* (*VIP*) yang dikelola oleh Keepalived. Keepalived bertugas memastikan bahwa *VIP* selalu diarahkan ke *node* yang berstatus *Master*. Dalam konfigurasi ini, terdapat dua *node* yang menjalankan HAProxy dan Keepalived secara bersamaan. Jika *node Master* mengalami kegagalan, Keepalived secara otomatis mengalihkan *VIP* ke *node Backup*, sehingga trafik tetap dapat diteruskan ke HAProxy pada *node* tersebut.

Di belakang HAProxy, terdapat *cluster* Docker Swarm yang terdiri dari beberapa *node worker*. HAProxy mengarahkan trafik ke *port* layanan yang dipublish pada *cluster* Swarm, dan selanjutnya, distribusi *request* ke *service* atau *container* aktif diatur oleh mekanisme internal Swarm Routing Mesh.

4.2.2. Flow Diagram Sistem

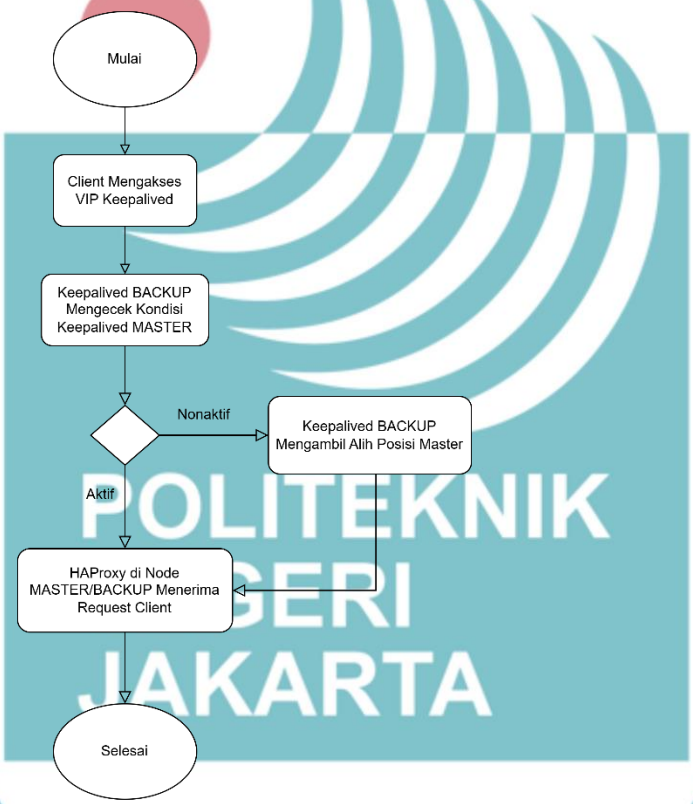


Gambar 4. 2 Alur Keseluruhan Sistem *High Availability*

Flow sistem secara keseluruhan dimulai ketika *client* mengirimkan HTTP *request* ke *Virtual IP* (*VIP*) yang dikelola oleh Keepalived. Penggunaan *VIP* ini memastikan bahwa *client* selalu mengakses sistem melalui satu titik akses yang konsisten, tanpa perlu mengetahui status *load balancer* yang sedang aktif. Ketika *request* masuk, Keepalived memastikan bahwa

VIP selalu terhubung dengan *node load balancer* (HAProxy) yang memiliki status *Master*. HAProxy pada *node* yang memegang *VIP* kemudian menerima *request* dan meneruskannya ke *cluster* Docker Swarm melalui *port* layanan yang telah dipublish. Selanjutnya, Docker Swarm Routing Mesh akan mendistribusikan *request* tersebut secara otomatis ke *instance service* (*container* aplikasi *web*) yang aktif di seluruh *node worker*. Setelah *service backend* memproses *request*, respons akan dikembalikan ke *client* melalui jalur yang sama.

4.2.3. Flow Diagram Keepalived



Gambar 4. 3 Alur Sistem *Failover* Otomatis oleh Keepalived

Flow sistem *failover* otomatis dimulai dengan penerimaan *request* dari *client* ke *Virtual IP* (*VIP*) yang dikelola oleh Keepalived melalui protokol VRRP (*Virtual Router Redundancy Protocol*). Keepalived bertugas memantau status *node load balancer* dan mengelola alokasi *VIP* berdasarkan prioritas yang ditetapkan. Pada konfigurasi dua *node load balancer*, satu *node* berstatus *Master* (prioritas lebih tinggi) yang memegang *VIP*, sementara *node* lain berstatus *Backup* (prioritas lebih

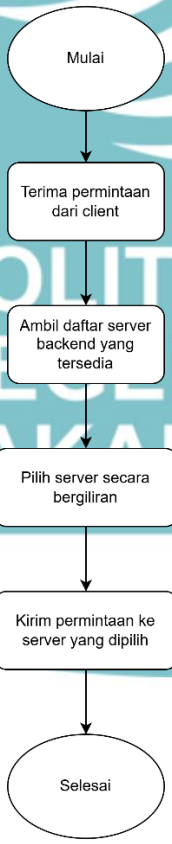
rendah). Saat *node Master* aktif, seluruh *request* dari *client* akan diarahkan ke HAProxy pada *node* tersebut. Jika *node Master* mengalami kegagalan, Keepalived pada *node Backup* akan otomatis mengambil alih *VIP*. HAProxy pada *node Backup* yang kini memegang *VIP* akan meneruskan *request* ke *cluster* Docker Swarm.

4.2.4. Flow Diagram *Load balancing*

Gambar 4. 4 Alur Sistem *Load balancing* oleh HAProxy

Sistem *load balancing* dimulai ketika *client* mengirimkan *request* ke *Virtual IP* (*VIP*) yang dikelola oleh Keepalived dan diarahkan ke *node* HAProxy yang sedang berstatus aktif (*Master*). Setelah menerima *request*, HAProxy melakukan *health check* ke setiap *service* yang berjalan di *node* Docker Swarm *worker*, sesuai dengan konfigurasi *backend* yang menggunakan alamat IP *node* dan *port service* yang dipublish. HAProxy kemudian mendistribusikan *request* ke *service* yang sehat menggunakan algoritma seperti *least connection* atau *round robin*, sesuai pengaturan *backend*. Selanjutnya, Docker Swarm Routing Mesh memastikan *request* yang masuk ke *port service* akan diteruskan ke *container* aktif yang menjalankan *service* terkait. *Service* dalam *container* tersebut lalu memproses permintaan dan mengirimkan respons kembali ke *client* melalui jalur yang sama.

4.2.4.1. Flow Diagram Algoritma Load Balancing Round Robin

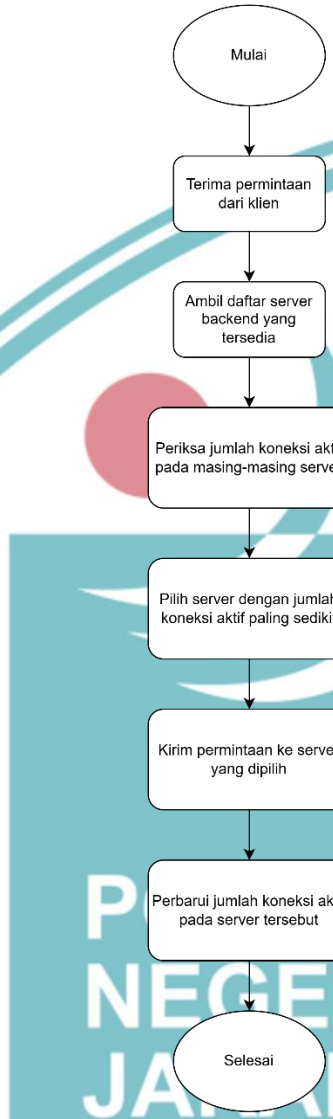


Gambar 4. 5 Alur Algoritma *Round Robin*

Pada algoritma *round robin*, alur yang terjadi ialah permintaan yang diterima diarahkan ke *node* HAProxy yang sedang berstatus aktif

(*Master*). HAProxy kemudian menerima *request* dan melakukan pengecekan terhadap daftar *service backend* yang tersedia berdasarkan hasil *health check*. Selanjutnya, HAProxy memilih *service* tujuan berdasarkan urutan giliran. Jika pada *request* sebelumnya *server* A digunakan, maka pada *request* berikutnya *server* B yang dipilih, dan seterusnya mengikuti urutan daftar. Ketika urutan mencapai *server* terakhir, giliran kembali ke *server* pertama, sehingga proses ini membentuk siklus bergiliran (*round robin*). Setelah *backend* ditentukan, *request* diteruskan melalui Docker Swarm Routing Mesh ke *container* yang menjalankan *service* tersebut. *Container* kemudian memproses permintaan dan mengirimkan respons kembali ke *client* melalui jalur yang sama.

4.2.4.2. Flow Diagram Algoritma Load Balancing Least connection



Gambar 4. 6 Alur Algoritma *Least connection*

Pada algoritma *least connection*, Setelah HAProxy menerima *request* dari *client* melalui *Virtual IP* (VIP) yang dikelola oleh Keepalived, sistem akan mengevaluasi kondisi setiap *backend* yang terdaftar. HAProxy melakukan pemeriksaan terhadap jumlah koneksi aktif pada masing-masing *service* yang tersedia. Berdasarkan hasil evaluasi tersebut, HAProxy akan memilih *service* yang memiliki jumlah koneksi aktif paling sedikit untuk menerima *request* baru. Hal ini dilakukan agar distribusi beban lebih merata dan tidak terjadi penumpukan koneksi pada satu *node* saja. Setelah *request* diteruskan, Docker Swarm Routing

Mesh akan menangani penerusan *request* ke *container* yang menjalankan *service* tersebut. Respons dari *container* akan dikembalikan ke *client* melalui jalur yang sama.

4.2.5. Perancangan *Website*

4.2.5.1. Perancangan MySQL

Tabel 4. 4 Perancangan dari Implementasi Basis Data MySQL

Node	***Role***	***IP***	***Port***
Worker 1	Master	192.168.56.13	3306
Worker 2	Slave	192.168.56.14	3306

Pada tabel 4.4 menunjukkan *role* dari tiap *instance* dari MySQL yang dijalankan. MySQL diimplementasikan dengan skema *Master-Slave Replication* yang dijalankan secara *native* pada masing-masing mesin virtual *Worker*. Untuk menjaga kestablilan basis data, MySQL ini juga dilengkapi dengan Orchestrator yang berfungsi sebagai alat *monitoring* status kesehatan dari tiap *instance*.

4.2.5.2. Perancangan MinIO

Layanan MinIO digunakan sebagai komponen *object storage* dalam sistem ini, yang berfungsi untuk menyimpan *file* gambar sirip hiu yang diunggah melalui model *Machine Learning*.

Untuk struktur layanan yang di*deploy* dapat dilihat pada tabel 4.5 di bawah:

Tabel 4. 5 Struktur Layanan dari *Deployment* MinIO

Node	***Instance***	***Port API***	***Port Web UI***	***Volume***
Worker 1	minio1	9000	9001	minio1-data
Worker 1	minio2	9002	9003	minio2-data
Worker 2	minio3	9004	9005	minio3-data
Worker 2	minio4	9006	9007	minio4-data

4.2.5.3. Perancangan FastAPI

Layanan FastAPI dalam sistem ini dirancang sebagai *backend* API yang menyediakan *endpoint* RESTful untuk seluruh fitur manajemen data sirip punggung hiu. Sebagai basis data utamanya, MySQL digunakan untuk penyimpanan data aplikasi, termasuk data sirip hiu, data pengguna, *log* aktivitas, serta target capaian. Integrasi antara FastAPI dan MySQL dilakukan melalui ORM SQLModel, yang memungkinkan proses pengolahan data dilakukan secara lebih efisien dan mudah dikelola.

Layanan FastAPI juga terintegrasi dengan MinIO, yang digunakan untuk menyimpan *file* gambar sirip punggung hiu. Integrasi ini dilakukan melalui modul *client* MinIO, sehingga proses *upload*, *retrieve*, dan *delete file* gambar dapat dilakukan langsung melalui API FastAPI.

Endpoint dari API yang tersedia dapat dilihat pada tabel 4.6 di bawah:

Tabel 4. 6 Daftar *Endpoint* FastAPI

No.	*Endpoint*	Metode	Fungsi / Deskripsi
1	`/register`	POST	Registrasi *user* baru, mengembalikan JWT.
2	`/login`	POST	Proses *login*, mengembalikan JWT.
3	`/user`	GET	Melihat daftar seluruh *user*
4	`/user/{user_id}`	GET	Melihat detail *user* berdasarkan ID.
5	`/user/{user_id}`	PUT	Mengubah data *user* (*user*name, *username*, *password*).
6	`/user/{user_id}`	DELETE	Menghapus *user.*

No.	*Endpoint*	Metode	Fungsi / Deskripsi
7	`/sirip-hiu`	GET	Melihat seluruh data sirip hiu.
8	`/sirip-hiu/{sirip_id}`	GET	Melihat detail data sirip hiu.
9	`/sirip-hiu`	POST	Menambahkan data sirip hiu baru + *upload* gambar ke MinIO.
10	`/sirip-hiu/{sirip_id}`	DELETE	Menghapus data sirip hiu dan *file* gambar di MinIO.
11	`/sirip-hiu/export-data`	GET	Mengekspor data sirip hiu ke *file* Excel.
12	`/sirip-hiu/get-image/{filename}`	GET	Mengambil gambar sirip hiu dari MinIO.
13	`/log-aktivitas`	GET	Melihat seluruh log aktivitas *user.*
14	`/target-capaian`	POST	Menambahkan target capaian baru.
15	`/target-capaian/{tahun}`	GET	Melihat target capaian berdasarkan tahun.
16	`/dashboard/summary`	GET	Mendapatkan ringkasan data *dashboard* (*summary*).

4.2.5.4. Perancangan Flask

Layanan Flask dalam sistem ini dirancang sebagai *dashboard web* yang menjadi antarmuka utama bagi *operator* dan *admin* dalam mengelola data sirip punggung hiu. *Dashboard* ini dibangun untuk mempermudah pengguna dalam melakukan pengelolaan data, *monitoring*, serta penyajian laporan dan visualisasi data dari sirip punggung hiu.

Tabel 4.7 di bawah menunjukkan struktur menu dan navigasi utama yang tersedia dalam *dashboard*:

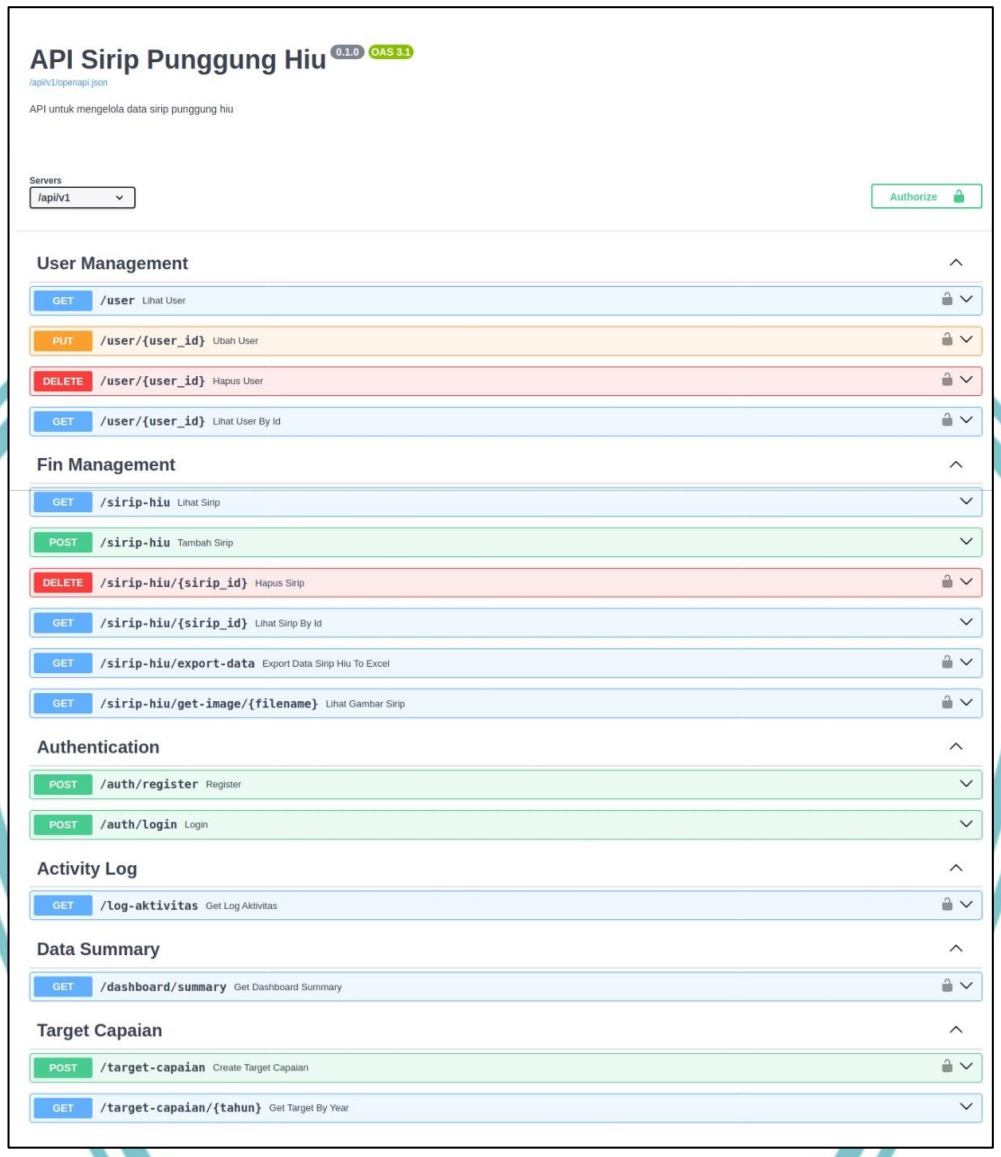
Tabel 4. 7 Menu dan Navigasi dari *Dashboard Website*

No.	Menu / Halaman	Fungsi / Fitur
1	Login	Halaman *login user* (*username* + *password*).
2	Dashboard	Menampilkan ringkasan data (grafik *summary*).
3	Data Sirip Hiu	Menampilkan tabel data sirip hiu, fitur *filter* & pencarian.
4	Tambah Data Sirip Hiu	*Form* tambah data sirip hiu + *upload* gambar.
5	Download Data Sirip Hiu	Fitur *download* data sirip hiu ke Excel.
6	Target Capaian	Melihat & menambah target capaian.
7	Manajemen User	Melihat, menambah, mengubah, menghapus *user.*
8	Log Aktivitas	Melihat riwayat aktivitas *user.*
9	Logout	Keluar dari sesi *login.*

4.3. Implementasi Sistem

4.3.1. Implementasi *Website Dashboard* Sirip Hiu

4.3.1.1. Implementasi FastAPI



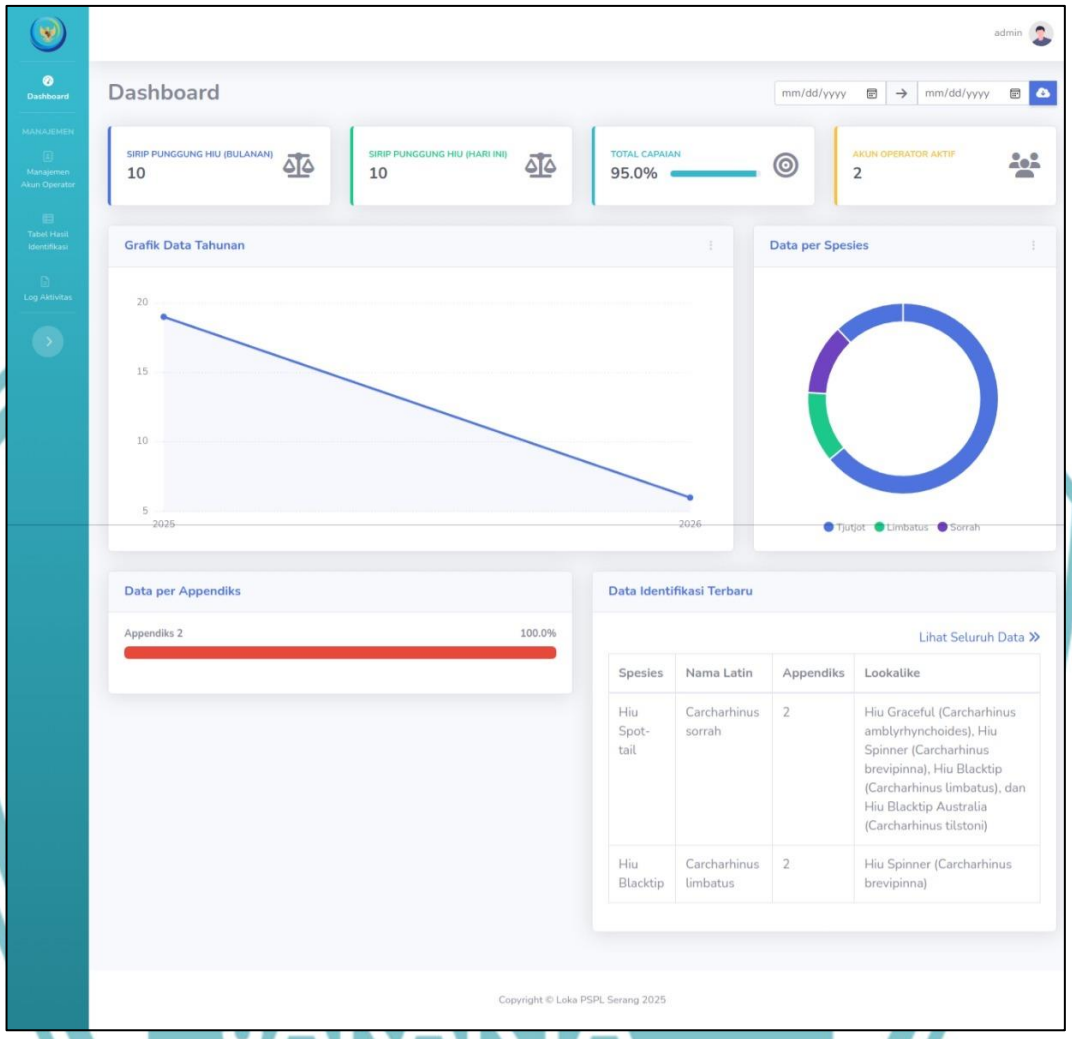
Gambar 4. 7 Dokumentasi FastAPI

Pada gambar 4.7 menunjukkan implementasi dari FastAPI yang dilakukan dengan membuat aplikasi berbasis Python yang menyediakan layanan API *backend* untuk *dashboard* sirip punggung hiu. Aplikasi FastAPI dikembangkan dengan beberapa modul utama yang mencakup:

1. Endpoint API untuk mengelola data sirip punggung hiu.

2. Dokumentasi otomatis dengan Swagger UI yang memudahkan interaksi dan pengujian.

4.3.1.2. Implementasi Flask



Gambar 4. 8 Halaman *Dashboard Website*

Pada gambar 4.8 menampilkan hasil implementasi dari Flask berupa halaman *dashboard web* yang terhubung langsung dengan layanan API FastAPI. *Dashboard* ini memiliki beberapa menu utama, seperti:

1. Menu awal: Menampilkan halaman interaktif untuk visualisasi data sirip hiu.
2. Manajemen *User*: Melihat, menambah, mengubah, menghapus *user*.
3. Data Sirip Hiu: Menampilkan tabel data sirip hiu yang dilengkapi dengan fitur *filter* & pencarian.

4. *Log* Aktivitas: Melihat riwayat aktivitas *user*.

4.3.2. Implementasi dan Konfigurasi Infrastruktur *High Availability*

4.3.2.1. Konfigurasi Jaringan pada Mesin Virtual

Dalam penelitian ini, perangkat lunak virtualisasi yang akan digunakan untuk membangun mesin virtual adalah VirtualBox. Ubuntu Server 22.04 digunakan sebagai sistem operasi dari mesin virtual dalam penelitian ini. Dalam pengaturan *adapter* jaringan, mesin-mesin virtual ini akan menggunakan tiga jenis *adapter*, yaitu: Bridged, NAT, dan Host-Only. Pengaturan *adapter* dari mesin virtual dapat dilihat pada tabel 4.8 berikut:

Tabel 4. 8 Pengaturan *Adapter* Jaringan Mesin Virtual

Nama Mesin Virtual	***Adapter* 1**	***Adapter* 2**
VM-LoadBalancer 1	Bridged	Host-Only
VM-LoadBalancer 2	Bridged	Host-Only
VM-DockerSwarm*Manager*	NAT	Host-Only
VM-DockerSwarm*Worker* 1	NAT	Host-Only
VM-DockerSwarm*Worker* 2	NAT	Host-Only

Adapter Bridged pada kedua mesin virtual *load balancer* digunakan dengan tujuan agar layanan Keepalived yang berjalan dalam mesin virtual tersebut dapat diakses oleh *client* dari luar yang masih dalam lingkup satu jaringan. *Adapter* NAT yang digunakan oleh mesin virtual Docker Swarm digunakan agar ketiga mesin virtual tersebut bisa mendapat akses *internet*, sedangkan *adapter* Host-Only digunakan oleh semua mesin virtual agar bisa saling berkomunikasi satu sama lain.

Seluruh *adapter* dari mesin virtual perlu dikonfigurasi alamat IP *static*. Alamat *IP static* yang digunakan tiap mesin virtual ini dapat dilihat pada tabel 4.9.

Tabel 4. 9 Pengaturan Alamat IP Mesin Virtual

Nama Mesin Virtual	**Alamat IP enp0s3**	**Alamat IP enp0s8**
VM-LoadBalancer 1	192.168.1.10	192.168.56.10
VM-LoadBalancer 2	192.168.1.11	192.168.56.11
VM-DockerSwarm*Manager*	-	192.168.56.12
VM-DockerSwarm*Worker* 1	-	192.168.56.13
VM-DockerSwarm*Worker* 2	-	192.168.56.14

Konfigurasi alamat IP *static* bisa dilakukan dengan menggunakan tool netplan yang dapat dijalankan dengan perintah seperti pada tabel 4.10.

Tabel 4. 10 Perintah *Edit* File Netplan

```
1   $ sudo nano /etc/netplan/50.cloud.init.yaml
```

Langkah selanjutnya adalah melakukan konfigurasi untuk mengatur alamat IP *static* dari *adapter* enp0s3 dan juga enp0s8 seperti pada tabel 4.11 sampai tabel 4.16 di bawah.

Tabel 4. 11 Konfigurasi *IP Static* VM-LoadBalancer 1

```
    Konfigurasi IP VM-LoadBalancer 1
1   network:
2     version: 2
3     ethernets:
4       enp0s3:
5         dhcp4: no
6   addresses:
7           - 192.168.1.10/24
8         routes:
9           - to: default
10            via: 192.168.1.1
11      enp0s8:
12        dhcp4: no
13        addresses:
14        - 192.168.56.10/24
```

	Konfigurasi IP VM-LoadBalancer 1
15	nameservers:
16	addresses: [8.8.8.8, 8.8.4.4]

Tabel 4. 12 Konfigurasi *IP Static* VM-LoadBalancer 2

	Konfigurasi IP VM-LoadBalancer 2
1	network:
2	version: 2
3	ethernets:
4	enp0s3:
5	dhcp4: no
6	addresses:
7	- 192.168.1.11/24
8	routes:
9	- to: default
10	via: 192.168.1.1
11	enp0s8:
12	dhcp4: no
13	addresses:
14	- 192.168.56.11/24
15	nameservers:
16	addresses: [8.8.8.8, 8.8.4.4]

Tabel 4. 13 Konfigurasi *IP Static* VM-DockerSwarm*Manager*

	Konfigurasi IP VM-DockerSwarmManager
1	network:
2	version: 2
3	ethernets:
4	enp0s3:
5	dhcp4: true
6	enp0s8:
7	dhcp4: no
8	addresses:
9	- 192.168.56.12/24
10	nameservers:
11	addresses: [8.8.8.8, 8.8.4.4]

Tabel 4. 14 Konfigurasi *IP Static* VM-DockerSwarm*Worker* 1

	Konfigurasi IP VM-DockerSwarmWorker 1
1	network:
2	version: 2
3	ethernets:
4	enp0s3:
5	dhcp4: true
6	enp0s8:
7	dhcp4: no
8	addresses:
9	- 192.168.56.13/24
10	nameservers:
11	addresses: [8.8.8.8, 8.8.4.4]

Tabel 4. 15 Konfigurasi *IP Static* VM-DockerSwarm*Worker* 2

	Konfigurasi IP VM-DockerSwarmWorker 2
1	network:
2	version: 2
3	ethernets:
4	enp0s3:
5	dhcp4: true
6	enp0s8:
7	dhcp4: no
8	addresses:
9	- 192.168.56.14/24
10	nameservers:
11	addresses: [8.8.8.8, 8.8.4.4]

Setelah mengganti alamat IP di tiap *adapter*, langkah selanjutnya adalah menjalankan perintah seperti pada tabel 4.16 untuk menerapkan konfigurasi IP. Langkah yang sama juga dilakukan ke mesin virtual lainnya.

Tabel 4. 16 Perintah untuk Menerapkan Konfigurasi Netplan

1	$ sudo netplan apply

Setelah mengganti semua alamat IP mesin virtual yang telah ditentukan menjadi *static*, berikutnya adalah menambahkan alamat IP dari *adapter* enp0s8 mesin virtual yang masuk dalam *cluster* Docker Swarm, yaitu VM-DockerSwarmManager, VM-DockerSwarmWorker 1, dan VM-DockerSwarmWorker 2 ke konfigurasi *host*s yang dapat dilakukan dengan menjalankan perintah seperti pada tabel 4.17.

Tabel 4. 17 Perintah untuk Masuk ke Konfigurasi Hosts

1	`$ sudo nano /etc/hosts`

Konfigurasi yang dilakukan adalah seperti pada tabel 4.18 di bawah.

Tabel 4. 18 Konfigurasi Hosts

	Konfigurasi File Hosts
1	127.0.0.1 localhost
2	127.0.1.1 vm-dockerswarmmanager
3	192.168.56.10 haproxy1
4	192.168.56.11 haproxy2
5	192.168.56.12 *manager*
6	192.168.56.13 *worker1* vm-dockerswarmworker1
7	192.168.56.14 *worker2* vm-dockerswarmworker2

4.3.2.2. Konfigurasi Docker Swarm

Langkah konfigurasi dari Docker Swarm dimulai dengan melakukan inisialisasi Docker Swarm pada VM-DockerSwarmManager dengan perintah seperti pada tabel 4.19.

Tabel 4. 19 Perintah untuk Inisiasi Docker Swarm

1	`$ docker swarm init --advertise-addr 192.168.56.12`

Selanjutnya adalah membuat *node worker* untuk *join* ke *cluster* menggunakan perintah seperti pada tabel 4.20.

Tabel 4. 20 Perintah untuk *Join* ke Docker Swarm *Cluster*

1	`$ docker swarm join --token <token-swarm> 192.168.56.12:2377`

Setelah proses *join cluster* selesai, validasi tiap *node* yang sudah *join* dengan menggunakan perintah seperti pada tabel 4.21.

Tabel 4. 21 Perintah untuk Menampilkan Daftar *Node* dalam *Cluster* Swarm

```
1  $ docker node ls
```

Gambar 4.9 di bawah menunjukkan tampilan dari *node manager* yang sudah memiliki dua *worker*.

```
vm3@vm-dockerswarmmanager:~$ docker node ls
ID                            HOSTNAME                STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
uohg50mopizgc9679skanz3ma *   vm-dockerswarmmanager   Ready     Active         Leader           28.1.1
4nzdstt5e98v0gh7mnu8bnhl9     vm-dockerswarmworker1   Ready     Active                          28.1.1
883o7kkohgzc0vedgbdkf16oc     vm-dockerswarmworker2   Ready     Active                          28.1.1
vm3@vm-dockerswarmmanager:~$
```

Gambar 4. 9 Daftar *Node* pada Docker *Cluster* Swarm

4.3.2.3. Konfigurasi Deployment Layanan

Seluruh layanan aplikasi (FastAPI, Flask, MinIO, dan MySQL) di*deploy* menggunakan skema stack *deployment* Docker Swarm yang didefinisikan dalam *file* docker-compose.yml. Layanan-layanan tersebut dapat dilihat pada tabel 4.22.

Tabel 4. 22 Layanan dalam Docker

Nama Layanan	Jumlah Instance	Port
FastAPI	2	8000
Flask	2	5000
MinIO	4	9000 (API), 9001 (Web)

Langkah *deployment* dari seluruh layanan dijalankan dengan menggunakan perintah seperti pada tabel 4.23.

Tabel 4. 23 Perintah untuk Deploy Layanan

```
1  $ docker stack deploy -c docker-compose.yml sirip-app
```

Setiap layanan yang sudah berjalan dapat dilihat dengan menggunakan perintah seperti pada tabel 4.24.

Tabel 4. 24 Perintah untuk Menampilkan Layanan yang Berjalan di Docker

1	`$ docker service ls`

4.3.2.4. Konfigurasi HAProxy

Langkah pertama adalah melakukan instalasi perangkat lunak HAProxy pada dua buah mesin virtual yang berperan sebagai Loadbalancer, yaitu pada VM-Loadbalancer 1 dan VM-Loadbalancer 2.

Untuk mengunduh dan menginstal HAProxy dapat dilakukan dengan perintah seperti pada Tabel 4.25.

Tabel 4. 25 Perintah untuk Menjalankan Instalasi HAProxy

1	`$ sudo apt install haproxy`

Setelah langkah tersebut selesai, selanjutnya adalah melakukan konfigurasi HAProxy. Untuk dapat menggunakan HAProxy, terdapat satu *file* konfigurasi utama yang harus disesuaikan, yaitu haproxy.cfg.

Untuk membuka dan mengubah isi *file* tersebut dapat dilakukan dengan perintah seperti pada Tabel 4.26.

Tabel 4. 26 Perintah untuk Mengubah Konfigurasi HAProxy

1	`$ sudo nano /etc/haproxy/haproxy.cfg`

Setelah *file* haproxy.cfg terbuka, langkah selanjutnya adalah menambahkan konfigurasi HAProxy seperti pada tabel 4.26.

Tabel 4. 27 Konfigurasi HAProxy

	Konfigurasi File haproxy.cfg
1	`global`
2	`    log /dev/log local0`
3	`    maxconn 2048`
4	`    daemon`
5	
6	`defaults`
7	`    log     global`
8	`    mode    http`

	Konfigurasi File haproxy.cfg
9	timeout connect 10s
10	timeout client 30s
11	timeout server 30s
12	option httplog
13	option dontlognull
14	
15	frontend http_front
16	bind *:80
17	
18	acl is_api path_beg /api/v1
19	acl is_api_docs path_beg /docs
20	acl is_openapi path_beg /openapi.json
21	acl is_front_end path_beg /dashboard
22	
23	use_backend fastapi_backend if is_api or is_api_docs or is_openapi
24	use_backend flask_backend if is_front_end
25	default_backend flask_backend
26	
27	frontend minio_api_front
28	bind *:9000
29	default_backend minio_api_backend
30	
31	frontend minio_web_front
32	bind *:9001
33	default_backend minio_web_backend
34	
35	frontend mysql
36	bind *:3306
37	mode tcp
38	default_backend mysql_servers
39	
40	backend fastapi_backend
41	balance leastconn
42	server fastapi1 192.168.56.13:8000 check
43	server fastapi2 192.168.56.14:8000 check
44	

	Konfigurasi File haproxy.cfg
45	backend flask_backend
46	balance leastconn
47	server flask1 192.168.56.13:5000 check
48	server flask2 192.168.56.14:5000 check
49	
50	backend minio_api_backend
51	balance roundrobin
52	server minio1 192.168.56.13:9000 check
53	server minio2 192.168.56.13:9002 check
54	server minio3 192.168.56.14:9004 check
55	server minio4 192.168.56.14:9006 check
56	
57	backend minio_web_backend
58	balance roundrobin
59	server minio1 192.168.56.13:9001 check
60	server minio2 192.168.56.13:9003 check
61	server minio3 192.168.56.14:9005 check
62	server minio4 192.168.56.14:9007 check
63	
64	backend mysql_servers
65	mode tcp
66	option mysql-check
67	server mysql1 192.168.56.13:3306 check
68	server mysql2 192.168.56.14:3306 check backup

Penjelasan dari konfigurasi HAProxy di atas adalah sebagai berikut:

Baris 1 – 4 : Menentukan parameter umum seperti lokasi pencatatan *log* di /dev/log, menentukan batas maksimal koneksi sebanyak 2048, serta menjalankan HAProxy sebagai proses daemon di latar belakang.

Baris 6 – 13 : Menetapkan pengaturan standar yang berlaku untuk semua layanan yang tidak memiliki aturan khusus. Dalam bagian ini, HAProxy dikonfigurasi menggunakan mode HTTP, mencatat aktivitas *log* secara global, mengatur batas waktu koneksi, *client*, dan *server* masing-masing 10

hingga 30 detik, serta mengaktifkan pencatatan aktivitas HTTP dan akan mengabaikan koneksi yang kosong.

Baris 14 – 23 : HAProxy mendengarkan *request* HTTP di *port* 80 dan melakukan pengaturan trafik berdasarkan URL yang diminta. Permintaan ke URL dengan path tertentu seperti /api/v1, /docs, atau /openapi.json akan diteruskan ke *backend* FastAPI. Sedangkan permintaan ke URL /dashboard diarahkan ke *backend* Flask. Jika *request* tidak mengakses salah satu dari *path* yang ditentukan, secara *default* HAProxy akan mengirim permintaan ke *backend* Flask.

Baris 25 – 27 : Mendengarkan *request* di *port* 9000 dan meneruskan permintaan langsung ke *backend* MinIO API.

Baris 29 – 31 : Mendengarkan di *port* 9001 dan meneruskan permintaan langsung ke *backend* MinIO *Web*.

Baris 33 – 36 : Mengelola lalu lintas TCP untuk MySQL di *port* 3306, setelah itu langsung diteruskan ke *backend* MySQL.

Baris 38 – 41 : Melakukan *load balancing* pada layanan FastAPI dan memilih *server* dengan koneksi paling sedikit antara dua *server*.

Baris 43 – 46 : Melakukan *load balancing* pada layanan Flask dan memilih *server* dengan koneksi paling sedikit antara dua *server*.

Baris 48 – 53 : *Load balancing* API MinIO di antara empat *instance* yang tersedia pada berbagai *port*.

Baris 55 – 60 : *Load balancing* layanan *web* MinIO, juga di antara empat *instance* dengan *port* berbeda.

Baris 62 – 65 : Menentukan *server* utama MySQL dan cadangan (*backup*).

Setelah konfigurasi HAProxy selesai dilakukan pada tiap mesin virtual *load balancer*, selanjutnya adalah melakukan perintah untuk memulai layanan HAProxy pada tiap mesin virtual *load balancer* dengan perintah seperti pada tabel 4.28.

Tabel 4. 28 Perintah untuk Menjalankan Layanan HAProxy

```
1  $ sudo systemctl start haproxy
```

Setelah layanan HAProxy dijalankan, dilakukan langkah verifikasi untuk memastikan bahwa layanan berjalan dengan normal dan siap menerima serta mendistribusikan trafik.

Pengecekan status layanan dilakukan dengan perintah seperti pada Tabel 4.29.

Tabel 4. 29 Perintah untuk Mengecek Status HAProxy

```
1  $ sudo systemctl status haproxy
```

Setelah layanan aktif, pengetesan secara langsung juga dilakukan untuk memastikan bahwa konfigurasi dari HAProxy sudah tepat. Pengetesan dilakukan dengan melakukan curl ke layanan FastAPI melalui IP dari *load balancer* seperti pada gambar 4.10 dan gambar 4.11.

```
vm3@vm-dockerswarmmanager:~/dashboard-website-sirip-hiu$ curl 192.168.56.10/docs
    <!DOCTYPE html>
    <html>
    <head>
    <link type="text/css" rel="stylesheet" href="https://cdn.jsdelivr.net/npm/swagger-ui-dist@5/swagger-ui.css">
    <link rel="shortcut icon" href="https://fastapi.tiangolo.com/img/favicon.png">
    <title>API Sirip Punggung Hiu - Swagger UI</title>
    </head>
    <body>
    <div id="swagger-ui">
    </div>
    <script src="https://cdn.jsdelivr.net/npm/swagger-ui-dist@5/swagger-ui-bundle.js"></script>
    <!-- `SwaggerUIBundle` is now available on the page -->
    <script>
    const ui = SwaggerUIBundle({
        url: '/api/v1/openapi.json',
    "dom_id": "#swagger-ui",
"layout": "BaseLayout",
"deepLinking": true,
"showExtensions": true,
"showCommonExtensions": true,
oauth2RedirectUrl: window.location.origin + '/api/v1/docs/oauth2-redirect',
    presets: [
        SwaggerUIBundle.presets.apis,
        SwaggerUIBundle.SwaggerUIStandalonePreset
        ],
    })
    </script>
    </body>
    </html>
    vm3@vm-dockerswarmmanager:~/dashboard-website-sirip-hiu$
```

Gambar 4. 10 Hasil curl melalui IP *Load balancer* 1

```
vm3@vm-dockerswarmmanager:~/dashboard-website-sirip-hiu$ curl 192.168.56.11/docs

    <!DOCTYPE html>
    <html>
    <head>
    <link type="text/css" rel="stylesheet" href="https://cdn.jsdelivr.net/npm/swagger-ui-dist@5/swagger-ui.css">
    <link rel="shortcut icon" href="https://fastapi.tiangolo.com/img/favicon.png">
    <title>API Sirip Punggung Hiu - Swagger UI</title>
    </head>
    <body>
    <div id="swagger-ui">
    </div>
    <script src="https://cdn.jsdelivr.net/npm/swagger-ui-dist@5/swagger-ui-bundle.js"></script>
    <!-- `SwaggerUIBundle` is now available on the page -->
    <script>
    const ui = SwaggerUIBundle({
        url: '/api/v1/openapi.json',
    "dom_id": "#swagger-ui",
"layout": "BaseLayout",
"deepLinking": true,
"showExtensions": true,
"showCommonExtensions": true,
oauth2RedirectUrl: window.location.origin + '/api/v1/docs/oauth2-redirect',
    presets: [
        SwaggerUIBundle.presets.apis,
        SwaggerUIBundle.SwaggerUIStandalonePreset
        ],
    })
    </script>
    </body>
    </html>
    vm3@vm-dockerswarmmanager:~/dashboard-website-sirip-hiu$
```

Gambar 4. 11 Hasil curl melalui IP *Load balancer* 2

4.3.2.5. Konfigurasi Keepalived

Langkah pertama adalah melakukan instalasi perangkat lunak Keepalived pada dua buah mesin virtual yang sebelumnya telah memiliki HAProxy yang terkonfigurasi.

Untuk menjalankan instalasi Keepalived dapat dilakukan dengan perintah seperti pada Tabel 4.30.

Tabel 4. 30 Perintah untuk Menjalankan Instalasi Keepalived

```
1   $ sudo apt install keepalived
```

Setelah langkah tersebut selesai, selanjutnya adalah melakukan konfigurasi Keepalived. Untuk dapat menggunakan Keepalived, terdapat satu *file* konfigurasi utama yang harus disesuaikan, yaitu keepalived.conf.

Untuk membuka dan meng*edit file* tersebut dapat dilakukan dengan perintah seperti pada Tabel 4.31.

Tabel 4. 31 Perintah untuk Mengubah Konfigurasi Keepalived

```
1   $ sudo nano /etc/keepalived/keepalived.conf
```

Setelah *file* keepalived.conf terbuka, langkah selanjutnya adalah menambahkan konfigurasi Keepalived seperti pada tabel 4.32 dan tabel 4.33.

Tabel 4. 32 Konfigurasi Keepalived *Master*

	Konfigurasi file keepalived.conf Master
1	vrrp_instance VI_1 {
2	state Master
3	interface enp0s3
4	virtual_router_id 51
5	priority 100
6	advert_int 1
7	authentication {
8	auth_type PASS
9	auth_pass 1234
10	}
11	virtual_ipaddress {
12	192.168.1.20
13	}
14	}

Tabel 4. 33 Konfigurasi Keepalived *Backup*

	Konfigurasi file keepalived.conf Backup
1	vrrp_instance VI_2 {
2	state Backup
3	interface enp0s3
4	virtual_router_id 51
5	priority 90
6	advert_int 1
7	authentication {
8	auth_type PASS
9	auth_pass 1234
10	}
11	virtual_ipaddress {
12	192.168.1.20
13	}
14	}

Penjelasan dari konfigurasi keepalived di atas adalah sebagai berikut:

Baris 1 : Membuat sebuah *instance* VRRP bernama VI_1. Instance ini berfungsi mengelola *IP virtual* untuk *failover* otomatis dalam suatu *cluster*.

Baris 2 : Mengatur *instance* yang berjalan bertindak sebagai *instance* utama atau *master* yang akan mengelola *IP virtual*.

Baris 3 : Menentukan antarmuka jaringan yang digunakan oleh Keepalived, yaitu *interface* enp0s3.

Baris 4 : Menentukan ID unik untuk grup VRRP.

Baris 5 : Menentukan prioritas *node* ini dalam pemilihan *master*. Semakin tinggi angka prioritas, semakin besar kemungkinan *node* tersebut menjadi *master*.

Baris 6 : Menentukan interval waktu pengiriman pesan untuk mengecek kondisi antar *instance* dalam satu grup VRRP.

Baris 7 – 9 : Menentukan metode autentikasi sederhana berbasis *password* dan menentukan *password* yang digunakan untuk otentikasi antar *instance*.

Baris 11 – 12: Alamat *IP virtual* yang akan digunakan oleh *instance* yang sedang aktif sebagai IP yang bisa diakses oleh *client* secara konsisten.

Setelah konfigurasi Keepalived selesai dilakukan pada tiap mesin virtual, selanjutnya adalah melakukan perintah untuk memulai layanan Keepalived pada tiap mesin dengan perintah seperti pada tabel 4.34.

Tabel 4. 34 Perintah untuk Menjalankan Layanan Keepalived

```
1  $ sudo systemctl start keepalived
```

Setelah layanan Keepalived dijalankan, langkah selanjutnya adalah melakukan verifikasi untuk memastikan bahwa layanan berjalan dengan normal.

Pengecekan status layanan dapat dilakukan dengan perintah seperti pada Tabel 4.35.

Tabel 4. 35 Perintah untuk Mengecek Status Keepalived

```
1  $ sudo systemctl status keepalived
```

Setelah layanan aktif, pengetesan secara langsung juga dilakukan untuk memastikan bahwa konfigurasi dari Keepalived sudah tepat. Pengetesan dilakukan dengan melakukan curl ke layanan FastAPI melalui *VIP* dari Keepalived seperti pada gambar 4.12.

```
vm3@vm-dockerswarmmanager:~/dashboard-website-sirip-hiu$ curl 192.168.1.20/docs

    <!DOCTYPE html>
    <html>
    <head>
    <link type="text/css" rel="stylesheet" href="https://cdn.jsdelivr.net/npm/swagger-ui-dist@5/swagger-ui.css">
    <link rel="shortcut icon" href="https://fastapi.tiangolo.com/img/favicon.png">
    <title>API Sirip Punggung Hiu - Swagger UI</title>
    </head>
    <body>
    <div id="swagger-ui">
    </div>
    <script src="https://cdn.jsdelivr.net/npm/swagger-ui-dist@5/swagger-ui-bundle.js"></script>
    <!-- `SwaggerUIBundle` is now available on the page -->
    <script>
    const ui = SwaggerUIBundle({
        url: '/api/v1/openapi.json',
    "dom_id": "#swagger-ui",
"layout": "BaseLayout",
"deepLinking": true,
"showExtensions": true,
"showCommonExtensions": true,
oauth2RedirectUrl: window.location.origin + '/api/v1/docs/oauth2-redirect',
    presets: [
        SwaggerUIBundle.presets.apis,
        SwaggerUIBundle.SwaggerUIStandalonePreset
        ],
    })
    </script>
    </body>
    </html>
    vm3@vm-dockerswarmmanager:~/dashboard-website-sirip-hiu$
```

Gambar 4. 12 Hasil curl melalui *VIP* Keepalived

4.4. Pengujian

4.4.1. Deskripsi Pengujian

Pengujian dilakukan untuk membuktikan keberhasilan implementasi *High Availability* (HA) pada arsitektur sistem dengan dua VM *Load balancer* (Keepalived + HAProxy), satu VM Swarm Manager, dua VM Swarm Worker yang menjalankan layanan MySQL *Replication* (*non-container*), 2

instance FastAPI, 2 *instance* Flask, dan 4 *instance* MinIO (*distributed mode*).

Fokus pengujian sistem ini mencakup lima aspek utama, yaitu pengujian *downtime*, *failover*, dan *failback*, pengujian beban, pengujian fungsi *load balancing*, pengujian *throughput* dan *response time*, serta pengujian pemakaian *resource*.

Kelima aspek ini diuji untuk memastikan bahwa sistem mampu memenuhi kebutuhan *High Availability*, performa, dan efisiensi pemakaian sumber daya.

4.4.2. Prosedur Pengujian

4.4.2.1. Uji Fungsionalitas *Website*

Pengujian dilakukan secara manual dengan uji coba langsung pada antarmuka *website*. Setiap fitur diuji satu per satu, meliputi:

- *Login*/*Log*out:
  Melakukan *login* menggunakan akun yang *valid* dan menguji proses *logout* untuk memastikan sistem dapat mengelola sesi pengguna dengan benar.
- Menampilkan Data Sirip Punggung Hiu:
  Mengakses menu data sirip punggung hiu dan memastikan seluruh data tampil sesuai yang ada di basis data.
- Filter Pencarian Data:
  Menggunakan fitur *filter* atau pencarian, lalu memastikan hasil yang muncul sesuai dengan kriteria yang dimasukkan.
- Manajemen Data Operator:
  Menguji penambahan, peng*edit*an, dan penghapusan data *operator* untuk memastikan setiap aksi berjalan sesuai fungsi.
- Download Rekap Data Sirip:
  Mengunduh rekap data sirip punggung hiu berdasarkan *filter* tertentu, lalu memeriksa kesesuaian data hasil unduhan.
- Input Data Capaian:

Melakukan input data capaian baru dan memastikan data tersebut tersimpan dan tampil di sistem.

– Menampilkan Data *Log* Aktivitas:
  Mengakses menu *log* aktivitas untuk memastikan seluruh aktivitas pengguna tercatat dan dapat ditampilkan.
– Menampilkan Diagram Grafik:
  Mengakses fitur grafik dan memastikan visualisasi yang muncul sesuai dengan data sirip punggung hiu yang tersimpan.

4.4.2.2. Uji Failover, Failback, dan Downtime

Pengujian ini dilakukan dengan mematikan salah satu VM *Load balancer* atau *Worker* secara paksa untuk mensimulasikan gangguan sistem. Selama proses ini, layanan dipantau menggunakan JMeter untuk mencatat *downtime*. Setelah itu, *node* yang dimatikan diaktifkan kembali untuk mengamati proses *failback* serta memastikan layanan kembali berjalan normal secara otomatis.

4.4.2.3. Uji Beban (Load & *Stress Test*)

Pengujian beban dilakukan menggunakan JMeter dengan mengirimkan *request* serentak ke *endpoint* API (FastAPI/Flask) dengan variasi 1.000, 3.000, 5.000, 7.000, dan 10.000 *request*. Selama proses ini, *response time*, *error rate*, *throughput*, serta stabilitas sistem dicatat. Di saat yang sama, dilakukan *monitoring* penggunaan *resource* dengan perintah docker stats pada kedua VM *Worker*.

4.4.2.4. Uji Load balancing

Pengujian ini bertujuan memastikan distribusi *request* dari *client* ke *backend* melalui HAProxy berjalan merata. Dua algoritma diterapkan pada HAProxy, yaitu: *round robin* dan *least connection*. Lalu, *request* dengan jumlah 300 dikirim ke virtual IP *load balancer* dan dilakukan pengamatan terhadap distribusi *request* yang diterima *backend*.

Pengujian terhadap performa dari algoritma *round robin* dan *least connection* juga dilakukan dengan skenario yang sama seperti pengujian

distribusi *load balancing* dan dilakukan pengamatan terhadap data waktu respons yang didapat.

4.4.2.5. Uji Throughput dan Response Time

Pengujian dilakukan dengan mengirimkan sejumlah *request* sebanyak 500 *request* per pengujian ke *endpoint* sistem secara paralel menggunakan tool JMeter. Setelah seluruh *request* terkirim, dicatat rata-rata *response time*, nilai *throughput*, serta *error rate* dari hasil *summary* setiap pengujian.

4.4.2.6. Uji *Resource* Sistem

Monitoring *resource* (CPU, memori) dilakukan pada setiap *worker* menggunakan docker stats, khususnya pada *service* FastAPI, Flask, dan MinIO, selama uji beban berlangsung.

4.4.3. Data Hasil Pengujian

4.4.3.1. Hasil Uji Fungsionalitas *Website*

Pada tabel 4.36 disajikan data hasil pengujian fungsionalitas dari seluruh fitur yang tersedia pada *website dashboard* sirip punggung hiu.

Pengujian dilakukan secara manual melalui antarmuka pengguna untuk memastikan bahwa setiap fitur berfungsi sesuai dengan rancangan sistem.

Tabel 4. 36 Data Hasil Uji Fungsionalitas Website

Pengujian	Hasil
Login/Logout	Berhasil
Menampilkan data sirip punggung hiu	Berhasil
Melakukan filter untuk pencarian data	Berhasil
Melakukan manajemen data operator (tambah, edit, dan hapus)	Berhasil

Pengujian	Hasil
Download rekap data sirip dengan filter	Berhasil
Melakukan input data capaian	Berhasil
Menampilkan data log aktivitas	Berhasil
Menampilkan diagram grafik berdasarkan data sirip punggung hiu	Berhasil

4.4.3.2. Hasil Uji Failover, Failback, dan Downtime

Untuk mengetahui kemampuan sistem dalam menghadapi kegagalan pada *node Load balancer* utama, dilakukan pengujian *failover*.

Tabel 4.37 di bawah ini menyajikan waktu perpindahan *Virtual IP* (*VIP*) dan waktu pemulihan akses layanan saat proses *failover* berlangsung.

Tabel 4. 37 Data Hasil Uji *Failover*

Pengujian ke-	Waktu MASTER Down	Waktu Failover	Durasi (detik)
1	2025-06-01 07:55:40	2025-06-01 07:55:41	1
2	2025-06-01 08:04:27	2025-06-01 08:04:28	1
3	2025-06-01 09:34:57	2025-06-01 09:34:58	1
4	2025-06-01 09:39:29	2025-06-01 09:39:30	1
5	2025-06-01 09:41:02	2025-06-01 09:41:03	1
Rata-rata			**1**

Setelah pengujian *failover* dilakukan, dilanjutkan dengan pengujian *failback* guna mengamati proses pemulihan peran *node Load balancer* utama.

Data pada tabel 4.38 berikut menunjukkan durasi proses *failback* yang tercatat saat *node* utama kembali diaktifkan.

Tabel 4. 38 Data Hasil Uji *Failback*

Pengujian ke-	**Waktu MASTER Up**	**Waktu Failback**	**Durasi (detik)**
1	2025-06-01 07:56:01	2025-06-01 07:56:01	0
2	2025-06-01 08:05:32	2025-06-01 08:05:36	4
3	2025-06-01 09:35:06	2025-06-01 09:35:09	3
4	2025-06-01 09:36:28	2025-06-01 09:36:32	4
5	2025-06-01 09:39:43	2025-06-01 09:39:47	4
Rata-rata			**4**

4.4.3.3. Uji Downtime

Pengujian *downtime* dilakukan untuk mengukur berapa lama layanan tidak dapat diakses akibat simulasi kegagalan yang diberikan pada sistem. Data dari pengujian *downtime* diukur selama proses pengujian *failover* dan *failback* berlangsung.

Tabel 4.39 di bawah ini menyajikan hasil pengukuran durasi *downtime* yang terjadi selama skenario pengujian dijalankan.

Tabel 4. 39 Data Hasil Uji *Downtime*

Pengujian ke-	Mulai Downtime	Selesai Downtime	Durasi (detik)
1	2025-06-03 09:31:03	2025-06-03 09:31:04	1
2	2025-06-03 09:35:55	2025-06-03 09:35:56	1
3	2025-06-03 09:37:16	2025-06-03 09:37:16	0
4	2025-06-03 09:37:34	2025-06-03 09:37:35	1
5	2025-06-03 09:38:43	2025-06-03 09:38:43	0
Rata-rata			**1**

4.4.3.4. Hasil Uji Fungsi *Load balancing*

Untuk memastikan layanan HAProxy berjalan, dilakukan pengujian distribusi *request* menggunakan algoritma *round robin* dan juga *least connection*.

Data pada tabel 4.40 berikut memperlihatkan jumlah *request* yang diterima oleh masing-masing *container backend* selama pengujian dengan menggunakan algoritma *round robin* berlangsung.

Tabel 4. 40 Data Hasil Pengujian *Load balancing* menggunakan Algoritma *Round Robin*

Pengujian ke-	Container 1 (99335c0b1a54)	Container 2 (e3ce44eac7f5)	Total Request	Distribusi (%)
1	156	144	300	52 / 48
2	148	152	300	49 / 51
3	181	119	300	60 / 40
4	145	155	300	48 / 52

Pengujian ke-	Container 1 (99335c0b1a54)	Container 2 (e3ce44eac7f5)	Total Request	Distribusi (%)
5	215	85	300	72 / 28

Tabel 4.41 di bawah ini menyajikan hasil pengujian jumlah *request* yang diterima oleh tiap *container backend* saat algoritma *least connection* diterapkan.

Tabel 4. 41 Data Hasil Pengujian *Load balancing* menggunakan Algoritma *Least connection*

Pengujian ke-	Container 1 (99335c0b1a54)	Container 2 (e3ce44eac7f5)	Total Request	Distribusi (%)
1	222	78	300	74 / 26
2	77	223	300	26 / 74
3	176	124	300	59 / 41
4	234	66	300	78 / 22
5	145	155	300	48 / 52

Data pada Tabel 4.42 memperlihatkan hasil pengujian performa sistem saat menggunakan algoritma *load balancing round robin*. Pengujian dilakukan sebanyak tiga kali dengan masing-masing terdiri dari 300 request menggunakan Apache JMeter.

Tabel 4. 42 Data Hasil Pengujian Performa Algoritma *Round Robin*

Pengujian ke-	Waktu Respons (detik)
1	2.65
2	1.64
3	1.82
Rata-rata	**1.94**

Data pada Tabel 4.43 menunjukkan hasil pengujian performa sistem dengan menggunakan algoritma *load balancing least connection*. Sama seperti pengujian sebelumnya, metode ini diuji sebanyak tiga kali dengan jumlah *request* yang sama.

Tabel 4. 43 Data Hasil Pengujian Algoritma *Least connection*

Pengujian ke-	**Waktu Respons (detik)**
1	0.46
2	0.47
3	0.51
Rata-rata	**0.48**

4.4.3.5. Hasil Uji Beban (*Stress test*)

Uji Beban dilakukan untuk menguji ketahanan sistem ketika menerima beban *request* dalam jumlah besar secara simultan.

Data hasil stress test pada tabel 4.44 mencerminkan performa sistem dalam merespon beban tinggi, baik dari segi *response time*, *error rate*, maupun *throughput*.

Tabel 4. 44 Data Hasil Uji Beban

Jumlah Request	**Rerata Respons (detik)**	**Error (%)**	**Throughput (req/detik)**
1.000	2.87	0.00	13.68
3.000	4.92	0.00	13.08
5.000	7.66	1.95	6.22
7.000	7.38	0.00	12.24
10.000	8.09	0.09	11.46

4.4.3.6. Hasil Uji Throughput dan Response Time

Pengujian *throughput* dan *response time* bertujuan untuk mengukur performa sistem dalam menangani *request* pada berbagai tingkat beban.

Pada tabel 4.45 disajikan data rata-rata *throughput* dan *response time* yang diperoleh dari pengujian yang telah dilakukan

Tabel 4. 45 Data Hasil Uji *Throughput* dan *Response Time*

Pengujian ke-	**Rerata Respons (detik)**	**Error (%)**	**Throughput (req/detik)**
1	0.465	0.0	17.53
2	0.476	0.0	17.56
3	0.510	0.0	15.80
Rata-rata	**0.483**	**0.0**	**16.96**

4.4.3.7. Hasil Monitoring Resource (VM Worker 1 & Worker 2)

Selama proses pengujian, dilakukan pula *monitoring* terhadap pemakaian *resource* pada masing-masing *node worker*.

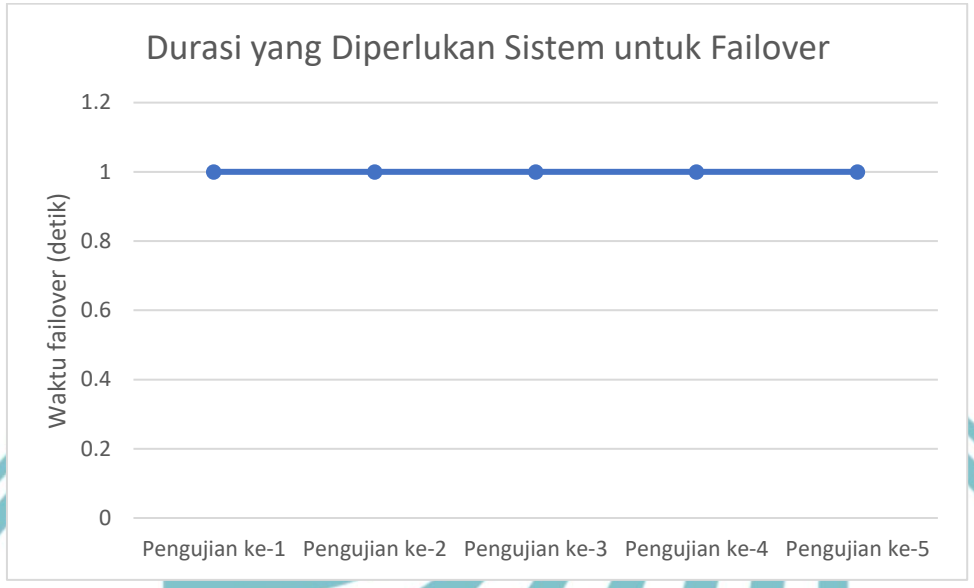
Data pada tabel 4.46 berikut menunjukkan konsumsi CPU dan memori oleh layanan yang berjalan pada VM Worker 1 dan VM Worker 2 selama pengujian berlangsung.

Tabel 4. 46 Data Hasil Uji Penggunaan *Resource*

Request	Worker	Service	Pemakaian CPU (%)	Mem (MiB)
1.000	*Worker* 1	FastAPI	11.93	121.29
1.000	*Worker* 1	MinIO	5.04	435.43
1.000	*Worker* 2	FastAPI	11.93	121.29
1.000	*Worker* 2	MinIO	5.04	435.43
3.000	*Worker* 1	FastAPI	22.33	122.15
3.000	*Worker* 1	MinIO	12.045	541.165
3.000	*Worker* 2	FastAPI	22.33	122.15
3.000	*Worker* 2	MinIO	12.045	541.165
5.000	*Worker* 1	FastAPI	0	0
5.000	*Worker* 1	MinIO	0	0
5.000	*Worker* 2	FastAPI	23.71	109.17
5.000	*Worker* 2	MinIO	11.66	585.37
7.000	*Worker* 1	FastAPI	31.85	121.35
7.000	*Worker* 1	MinIO	15.28	597.345
7.000	*Worker* 2	FastAPI	31.85	121.35
7.000	*Worker* 2	MinIO	15.28	597.345
10.000	*Worker* 1	FastAPI	33.41	117.22
10.000	*Worker* 1	MinIO	16.51	621.13
10.000	*Worker* 2	FastAPI	33.41	117.22
10.000	*Worker* 2	MinIO	16.51	621.13

4.4.4. Analisis Data/Evaluasi Pengujian

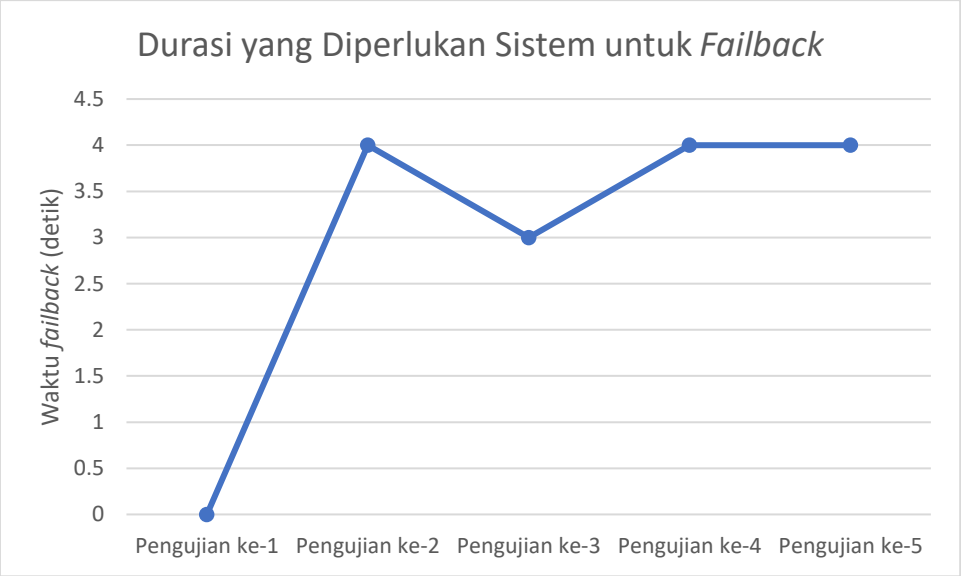
4.4.4.1. Analisis Pengujian *Failover*



Gambar 4. 13 Grafik Data Pengujian *Failover*

Pada gambar 4.13 menunjukkan bahwa seluruh proses *failover* berlangsung sangat cepat dan konsisten, dengan waktu rata-rata perpindahan hanya 1 detik pada setiap siklus uji. Hal ini menandakan sistem *High Availability* yang dibangun telah mampu mendeteksi kegagalan *node* utama dan melakukan pemindahan layanan ke *node* cadangan dalam waktu yang sangat singkat, sehingga layanan tetap dapat diakses oleh pengguna tanpa gangguan yang berarti.

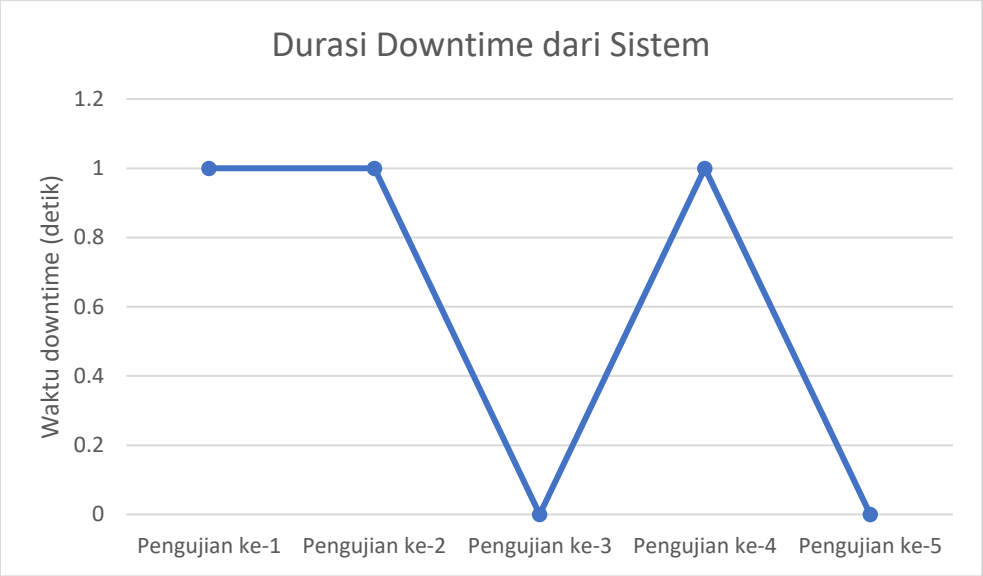
4.4.4.2. Analisis Pengujian *Failback*



Gambar 4. 14 Grafik Data Pengujian *Failback*

Pada gambar 4.14 menunjukkan waktu *failback* yang bervariasi antara 0 hingga 4 detik. Menurut (Hinden, 2004) dalam dokumen standar RFC 3768, angka tersebut adalah durasi normal untuk *failback* sesuai dengan perhitungan *master down interval*. Mekanisme ini berjalan otomatis tanpa intervensi manual, yang membuktikan bahwa sistem dapat memulihkan status awal *High Availability* secara efisien setelah terjadi pemulihan pada *node* utama.

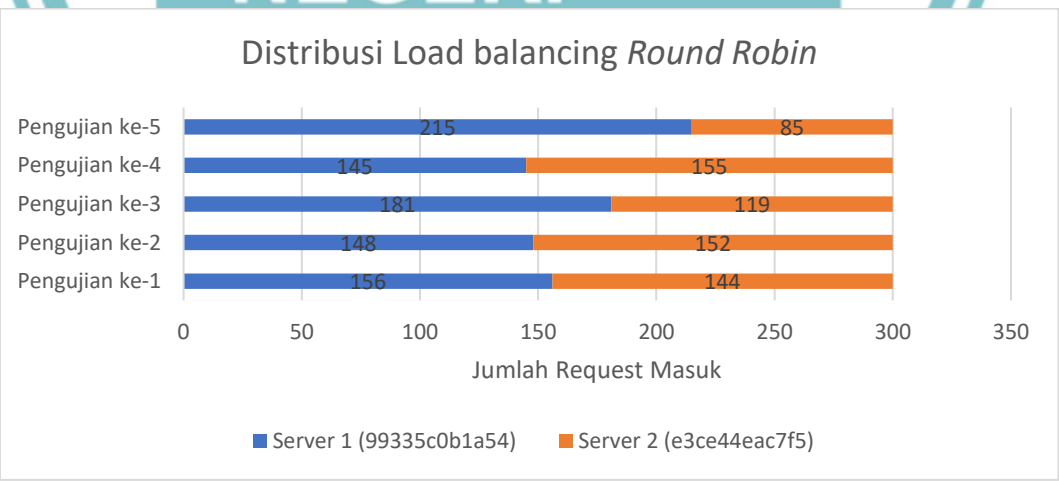
4.4.4.3. Analisis Pengujian *Downtime*



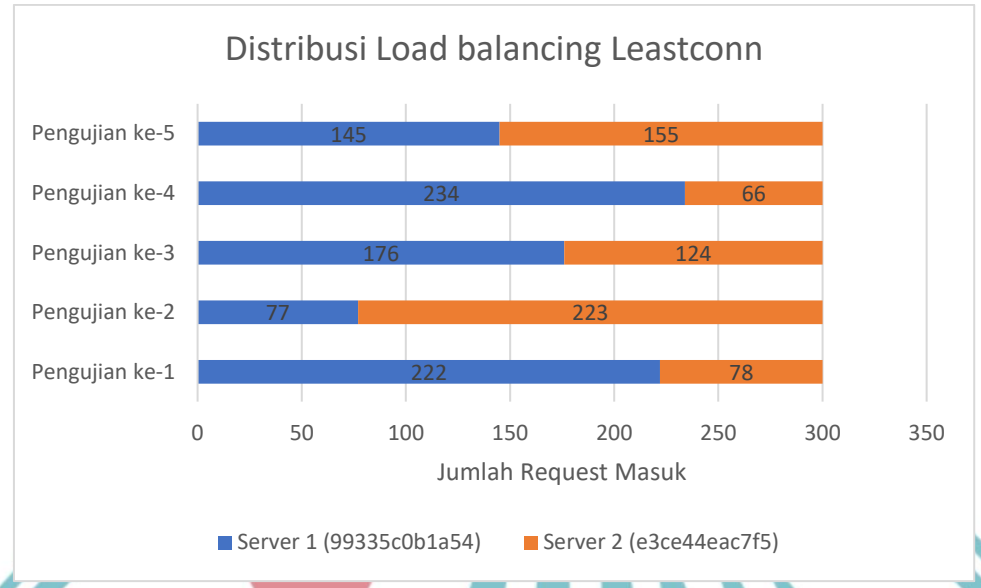
Gambar 4. 15 Grafik Data Pengujian *Downtime*

Dari gambar 4.15 di atas menunjukkan bahwa *downtime* aktual yang terdeteksi umumnya terjadi selama 1 detik, dengan beberapa pengecualian yang hanya mencapai 0 detik. Hal ini membuktikan bahwa infrastruktur HA yang diterapkan sangat efektif dalam menjaga ketersediaan layanan dengan gangguan waktu minimum selama proses *failover* ataupun gangguan pada salah satu *node*.

4.4.4.4. Analisis Pengujian Distribusi *Load balancing*



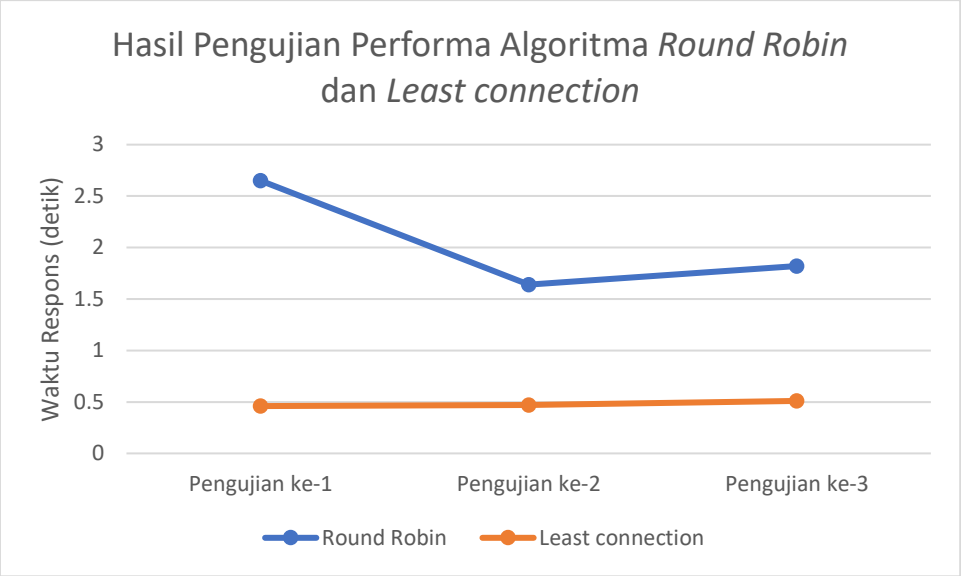
Gambar 4. 16 Distribusi *Load balancing* Algoritma *Round Robin*



Gambar 4. 17 Distribusi *Load balancing* Algoritma *Least connection*

Pada gambar 4.16 dan gambar 4.17 menunjukkan distribusi dari *request* yang diterima *backend*. Pada algoritma *Round Robin*, distribusi permintaan antara dua *container* terpantau relatif merata, dengan proporsi rata-rata di sekitar 52:48 hingga 60:40. Pada algoritma *least connection*, distribusi bisa menjadi lebih tidak merata, bergantung pada kondisi beban, namun sistem tetap mampu mendistribusikan *request* sesuai algoritma *load balancing* yang diimplementasikan. Hasil ini menunjukkan bahwa fitur *load balancing* bekerja sesuai desain dan mampu mencegah penumpukan beban pada satu *container* saja.

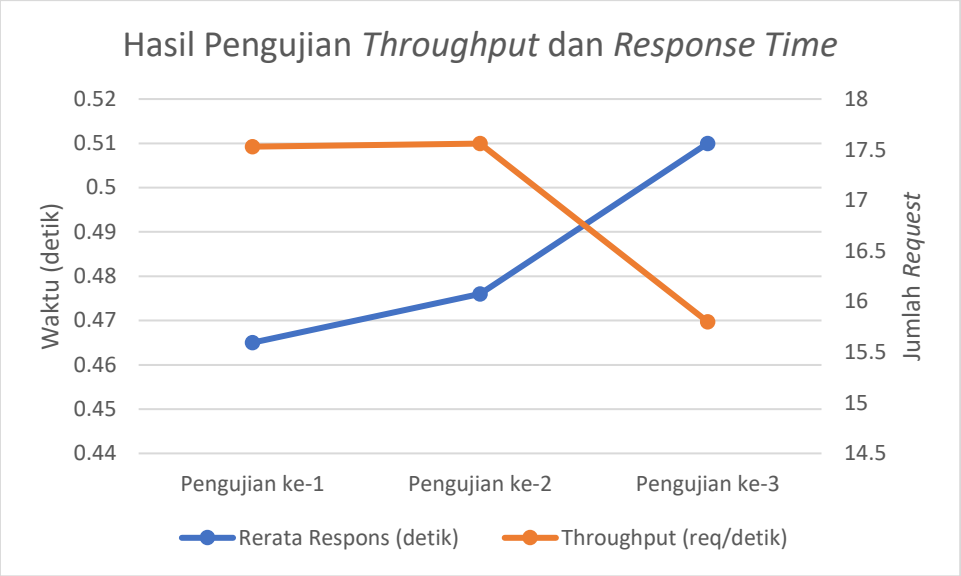
4.4.4.5. Analisis Pengujian Performa Algoritma *Load balancing*



Gambar 4. 18 Grafik Hasil Pengujian Performa Algoritma *Load Balancing*

Pada gambar 4.18 dapat dilihat bahwa algoritma *Least connection* menunjukkan performa yang lebih unggul dibandingkan *Round Robin*. Rata-rata waktu respons dari algoritma *Least connection* lebih rendah, yaitu sekitar 0,48 detik, dibandingkan dengan *Round Robin* yang berada di angka 1,94 detik. Hal ini menunjukkan bahwa *Least connection* mampu mendistribusikan permintaan dengan lebih efisien, terutama ketika terdapat perbedaan beban antar *container backend*.

4.4.4.6. Analisis Hasil Pengujian *Throughput* dan *Response Time*



Gambar 4. 19 Grafik Hasil Pengujian *Throughput* dan *Response Time*

Pada gambar 4.19 menunjukkan hasil dari pengujian *throughput* dan *response time*. Rerata waktu respons sistem tetap stabil di bawah 0,51 detik, sedangkan *throughput* mencapai 15,8 hingga 17,56 *request*/detik tanpa *error* yang berarti.

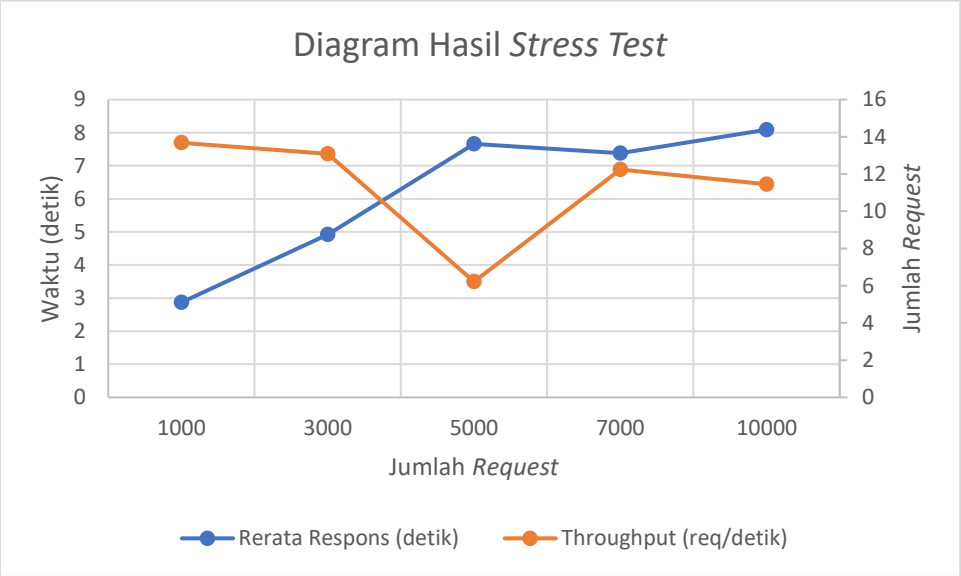
Mengacu pada standar ISO/IEC 25010:2011, evaluasi performa sistem dilakukan berdasarkan dua subkarakteristik utama, yaitu:

1. *Time Behavior*: Dengan *response time* rata-rata di bawah 1 detik, sistem tergolong responsif dan memenuhi ekspektasi kualitas layanan berdasarkan SLA umum dari dokumentasi Google SRE yang merekomendasikan waktu kurang dari 1 detik untuk waktu respons yang dimiliki layanan *web* (Beyer et al., 2016).
2. *Capacity*: Berdasarkan aturan praktis dari University of California, Los Angeles (UCLA), untuk konten dinamis, sistem diharapkan mampu menangani sekitar 10 *request* per detik per *core (Cho J, 2021)*. Dalam konteks sistem skala kecil hingga menengah yang diimplementasikan pada penelitian ini, *throughput* sebesar 15-17 *request* per detik dapat dianggap sebagai performa yang baik,

terutama jika sistem menunjukkan stabilitas dan tidak mengalami *error* saat menangani beban tersebut.

Hasil ini menunjukkan performa sistem yang sangat baik pada beban normal hingga menengah.

4.4.4.7. Analisis Hasil Pengujian *Stress Test*

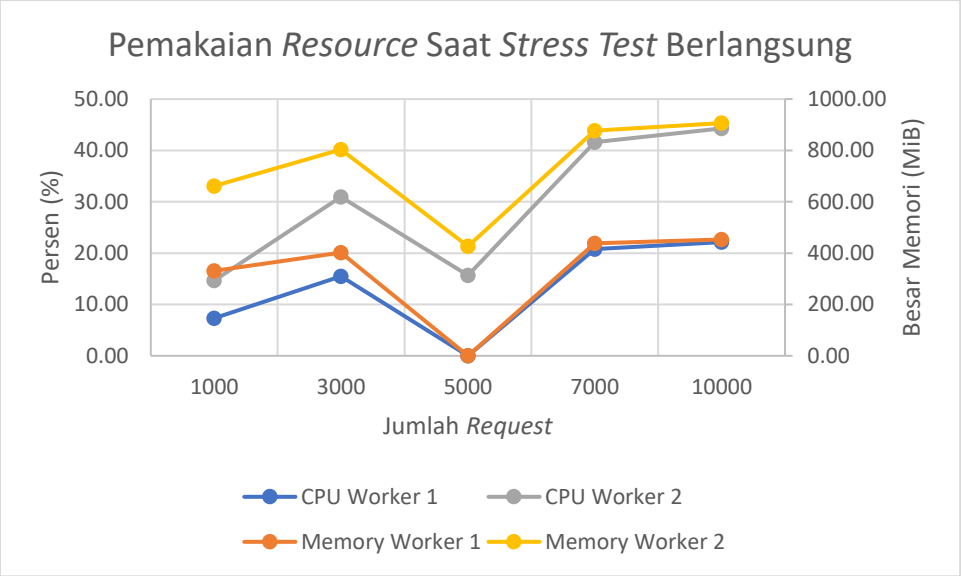


Gambar 4. 20 Grafik Data Hasil Uji *Stress Test*

Pada gambar 4.20 terlihat bahwa rerata *response time* meningkat seiring dengan kenaikan jumlah *request*, dari 2,87 detik (1.000 *request*) hingga 8,09 detik (10.000 *request*). *Error rate* tetap sangat rendah (hampir 0%) pada sebagian besar skenario, menunjukkan sistem tetap stabil di bawah tekanan tinggi. *Throughput* juga cenderung stabil pada kisaran 11–14 *request*/detik, kecuali pada beban 5.000 *request*.

Penurunan *throughput* pada pengujian 5.000 *request* diindikasikan sebagai *bottleneck resource* yang bersifat sementara, hal ini terjadi karena seluruh antrian proses hanya dijalankan oleh VM-DockerSwarmWorker 2 yang sedang intensif pada saat pengujian tersebut. Kemungkinan hal ini terjadi karena *container* pada VM-DockerSwarmWorker 1 menganggap terjadi kekurangan *resource* dan berpindah ke VM lain pada saat pengujian 5.000 *request.*

4.4.4.8. Analisis Hasil Pengujian Pemakaian *Resource*



Gambar 4. 21 Grafik Data Hasil Uji Pemakaian *Resource*

Dari gambar 4.21 terlihat bahwa peningkatan jumlah *request* berdampak langsung terhadap beban kerja pada masing-masing *worker*. Pada skenario 1000 *request*, rata-rata penggunaan CPU tercatat sebesar 7,33% dengan konsumsi memori sekitar 330 MiB di kedua *worker*. Ketika jumlah *request* ditingkatkan menjadi 10.000, rata-rata penggunaan CPU meningkat menjadi 22,1%, dan konsumsi memori mencapai 453 MiB pada kedua *worker*. Kenaikan ini menunjukkan bahwa sistem merespons beban dengan proporsional dan stabil. Selain itu, pola konsumsi *resource* antara *Worker* 1 dan *Worker* 2 tampak seimbang, hal ini mengindikasikan distribusi layanan sudah merata dalam arsitektur *High Availability* berbasis Docker Swarm.

BAB V
PENUTUP

5.1. Kesimpulan

1. Penelitian ini berhasil mengimplementasikan sistem *High Availability* (HA) dengan mekanisme *failover* otomatis menggunakan HAProxy dan Keepalived pada infrastruktur berbasis Docker Swarm. Sistem yang dibangun mampu menjaga ketersediaan layanan *web* meskipun terjadi kegagalan pada salah satu *node load balancer*.
2. Pengujian menunjukkan bahwa proses *failover* berlangsung rata-rata dalam waktu 1 detik, dengan *downtime* layanan minimal (0–1 detik), serta proses *failback* yang berjalan otomatis. Sistem *load balancing* berfungsi sesuai desain, mendistribusikan trafik ke *container backend* menggunakan algoritma *round robin* dan *least connection*.
3. Pada pengujian beban tinggi hingga 10.000 *request*, sistem tetap dapat melayani permintaan dengan tingkat *error* yang sangat rendah, meskipun terjadi peningkatan waktu respons dan pemakaian sumber daya yang signifikan.

5.2. Saran

Berdasarkan hasil penelitian dan pengujian yang telah dilakukan, terdapat beberapa saran yang dapat dipertimbangkan untuk pengembangan lebih lanjut:

1. Untuk penelitian selanjutnya, diharapkan adanya penambahan untuk sistem *monitoring*. Hal tersebut ditambahkan untuk memantau performa sistem secara *real-time*, kesehatan *node*, dan juga pemakaian sumber daya.
2. Untuk meningkatkan tingkat ketersediaan dan skalabilitas, disarankan untuk menambahkan jumlah VM, baik pada *layer load balancer, manager node,* maupun *worker node*. Dengan menambah jumlah *node load balancer*, sistem dapat memiliki redundansi yang lebih baik. Penambahan *manager node* dapat menyediakan redundansi untuk mengatasi kegagalan yang mungkin terjadi, dan penambahan *worker*

node juga memungkinkan distribusi beban yang lebih merata serta peningkatan kapasitas layanan.

DAFTAR PUSTAKA

Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (2016). *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly. https://books.google.co.id/books?id=81UrjwEACAAJ

Cho J. (2021). *Scaling Web Service*.

Cloudflare. (2020, September 24). *What is load balancing? | How load balancers work | Cloudflare*. https://www.cloudflare.com/learning/performance/what-is-load-balancing/

Cross, M., Martin, J. A., Walls, T. A., Grasdal, M., Shinder, D. L., & Shinder, T. W. (2003). MCSA/MCSE 70-294: Ensuring Active Directory Availability. *MCSE (Exam 70-294) Study Guide*, 689–769. https://doi.org/10.1016/B978-193183694-4/50017-9

Dermawan, A., & Sadili, D. (2015). *Rencana Aksi Nasional (RAN) Konservasi dan Pengelolaan Hiu dan Pari*. KKP-Direktorat Jenderal Pengelolaan Ruang Laut. //perpustakaan.kkp.go.id%2Fknowledgerepository%2Findex.php%3Fp%3Dshow_detail%26id%3D1073221%26keywords%3D

Docker. (2016, June 20). *Swarm mode key concepts | Docker Docs*. https://docs.docker.com/engine/swarm/key-concepts/

Docker. (2024, August 20). *What is Docker? | Docker Docs*. https://docs.docker.com/get-started/docker-overview/

Hannifin, D., Alpern, N. J., & Alpern, J. (2010). Windows Server 2008 R2 high-availability and recovery features. *Microsoft Windows Server 2008 R2 Administrator's Reference*, 399–459. https://doi.org/10.1016/B978-1-59749-578-3.00009-8

HAProxy. (2021, November 24). *HAProxy - The Reliable, High Perf. TCP/HTTP Load Balancer*. https://www.haproxy.org/

Hinden, R. (2004). *Virtual Router Redundancy Protocol (VRRP)* (R. Hinden, Ed.). https://doi.org/10.17487/rfc3768

ISO/IEC 25010. (2011). *ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*.

Keepalived. (2020, June 13). *Keepalived for Linux*. https://www.keepalived.org/

Lack, M., & Sant, G. (2011). *The Future of Sharks: A Review of Action and Inaction*.

Lahti, B. C., & Peterson, R. (2005). Chapter 7 - Domain III: Delivery and Support. In C. B. Lahti & R. Peterson (Eds.), *Sarbanes-Oxley IT Compliance Using COBIT and Open Source Tools* (pp. 175–221). Syngress. https://doi.org/https://doi.org/10.1016/B978-159749036-8/50010-0

Pratama, D. (2021). *PERBANDINGAN KINERJA TEKNOLOGI FAILOVER BERBASIS KLASTER (HEARTBEAT) DENGAN TEKNOLOGI FAILOVER BERBASIS JARINGAN (KEEPALIVED)*. UNIVERSITAS BRAWIJAYA.

Putra, M. A. A., Fitri, I., & Iskandar, A. (2020). Implementasi High Availability Cluster Web Server Menggunakan Virtualisasi Container Docker. *JURNAL MEDIA INFORMATIKA BUDIDARMA*, *4*(1), 9. https://doi.org/10.30865/mib.v4i1.1729

Rigby, C. L., Appleyard, S., Chin, A., Heupel, M., Humber, F., Jeffers, V., Simpfendorfer, C., White, W. T., & Campbell, I. (2019). Rapid Assessment Toolkit for Sharks and Rays. In *WWF International and CSTFA*. https://sharks.panda.org/images/PDF/Bahasa_RAT_Toolkit_for_web___low_res.pdf

Rusandi, A., Hadi, S., Ariansyah, W. F., Muttaqin, E., Sudarisman, R., Sualia, I., & Prihardiyanto, R. W. (2019). KAJIAN PENYUSUNAN DESAIN KETELUSURAN PRODUK HIU DAN PARI DI INDONESIA DAN PETA JALANNYA. In *Dokumen USAID 2019*. https://pdf.usaid.gov/pdf_docs/PA00XGR6.pdf

DAFTAR RIWAYAT HIDUP

Joko Prasetyo

Lahir di Jakarta pada 20 Oktober 2002. Berdomisili di Jakarta Barat. Peneliti memasuki pendidikan formal di SDN Kapuk 11 Pagi dan lulus pada tahun 2015. Kemudian melanjutkan pendidikan menengah di SMPN 132 Jakarta dan lulus pada tahun 2018. Setelah lulus, peneliti melanjutkan pendidikan di SMKN 53 Jakarta dan selesai pada tahun 2021. Pada tahun yang sama, peneliti melanjutkan pendidikan Diploma Empat (D4) di Politeknik Negeri Jakarta jurusan Teknik Informatika dan Komputer dengan program studi Teknik Multimedia dan Jaringan.

LAMPIRAN

Lampiran 1 - File docker-compose.yml

```yaml
x-minio-env: &minio_env
  MINIO_DISTRIBUTED_MODE_ENABLED: "yes"
  MINIO_DISTRIBUTED_NODES: "minio1,minio2,minio3,minio4"
  MINIO_ROOT_USER_FILE: /run/secrets/minio_access_key
  MINIO_ROOT_PASSWORD_FILE: /run/secrets/minio_secret_key

services:
  flask:
    image: zocc/fe-sirip-hiu
    ports:
      - "5000:5000"
    networks:
      - app-net
    deploy:
      replicas: 2
      placement:
        constraints: [node.role == worker]
    secrets:
      - api_base_url
      - secret_key
      - jwt_algorithm
      - jwt_secret_key
    environment:
      API_BASE_URL_FILE: /run/secrets/api_base_url
      SECRET_KEY_FILE: /run/secrets/secret_key
      JWT_ALGORITHM_FILE: /run/secrets/jwt_algorithm
      JWT_SECRET_KEY_FILE: /run/secrets/jwt_secret_key

  fastapi:
    image: zocc/api-sirip-hiu
    ports:
      - "8000:8000"
    networks:
      - app-net
    deploy:
      replicas: 2
      placement:
        constraints: [node.role == worker]
    secrets:
      - mysql_user
      - mysql_password
      - mysql_host
      - mysql_port
      - mysql_name
      - minio_bucket_name
      - minio_host
      - minio_port
      - jwt_algorithm
      - jwt_expiration_time
      - jwt_secret_key
      - minio_expiration_time
      - minio_use_ssl
      - minio_access_key
      - minio_secret_key
    environment:
      MYSQL_USER_FILE: /run/secrets/mysql_user
      MYSQL_PASSWORD_FILE: /run/secrets/mysql_password
      MYSQL_HOST_FILE: /run/secrets/mysql_host
      MYSQL_PORT_FILE: /run/secrets/mysql_port
      MYSQL_NAME_FILE: /run/secrets/mysql_name
```

(lanjutan)

```
      MINIO_HOST_FILE: /run/secrets/minio_host
      MINIO_PORT_FILE: /run/secrets/minio_port
      MINIO_ACCESS_KEY_FILE: /run/secrets/minio_access_key
      MINIO_SECRET_KEY_FILE: /run/secrets/minio_secret_key
      MINIO_BUCKET_NAME_FILE: /run/secrets/minio_bucket_name
      MINIO_USE_SSL_FILE: /run/secrets/minio_use_ssl
      MINIO_EXPIRATION_TIME_FILE: /run/secrets/minio_expiration_time

      JWT_SECRET_KEY_FILE: /run/secrets/jwt_secret_key
      JWT_ALGORITHM_FILE: /run/secrets/jwt_algorithm
      JWT_EXPIRATION_TIME_FILE: /run/secrets/jwt_expiration_time

  minio1:
    image: minio/minio
    hostname: minio1
    command: server --console-address ":9001" http://minio{1...4}/data{1...2}
    ports:
      - "9000:9000"
      - "9001:9001"
    volumes:
      - minio1_data1:/data1
      - minio1_data2:/data2
    environment: *minio_env
    secrets:
      - minio_access_key
      - minio_secret_key
    networks:
      - app-net
    deploy:
      restart_policy:
        delay: 10s
        max_attempts: 10
        window: 60s
      placement:
        constraints: [node.hostname == vm-dockerswarmworker1]

  minio2:
    image: minio/minio
    hostname: minio2
    command: server --console-address ":9003" http://minio{1...4}/data{1...2}
    ports:
      - "9002:9000"
      - "9003:9001"
    volumes:
      - minio2_data1:/data1
      - minio2_data2:/data2
    environment: *minio_env
    secrets:
      - minio_access_key
      - minio_secret_key
    networks:
      - app-net
    deploy:
      restart_policy:
        delay: 10s
        max_attempts: 10
        window: 60s
      placement:
        constraints: [node.hostname == vm-dockerswarmworker1]
```

(lanjutan)

```
  minio3:
    image: minio/minio
    hostname: minio3
    command: server --console-address ":9005" http://minio{1...4}/data{1...2}
    ports:
      - "9004:9000"
      - "9005:9001"
    volumes:
      - minio3_data1:/data1
      - minio3_data2:/data2
    environment: *minio_env
    secrets:
      - minio_access_key
      - minio_secret_key
    networks:
      - app-net
    deploy:
      placement:
        constraints: [node.hostname == vm-dockerswarmworker2]

  minio4:
    image: minio/minio
    hostname: minio4
    command: server --console-address ":9007" http://minio{1...4}/data{1...2}
    ports:
      - "9006:9000"
      - "9007:9001"
    volumes:
      - minio4_data1:/data1
      - minio4_data2:/data2
    environment: *minio_env
    secrets:
      - minio_access_key
      - minio_secret_key
    networks:
      - app-net
    deploy:
      placement:
        constraints: [node.hostname == vm-dockerswarmworker2]

  minio-init:
    image: minio/mc
    networks:
      - app-net
    restart: on-failure
    volumes:
      - ./minio/init-minio.sh:/init-minio.sh:ro
    entrypoint: /bin/sh /init-minio.sh
    secrets:
      - minio_access_key
      - minio_secret_key
      - minio_bucket_name

networks:
  app-net:
    driver: overlay

volumes:
  minio1_data1:
    driver: local
  minio1_data2:
    driver: local
  minio2_data1:
    driver: local
  minio2_data2:
    driver: local
  minio3_data1:
    driver: local
```

(lanjutan)

```
  minio3_data2:
    driver: local
  minio4_data1:
    driver: local
  minio4_data2:
    driver: local

secrets:
  api_base_url:
    external: true
  secret_key:
    external: true

  mysql_user:
    external: true
  mysql_password:
    external: true
  mysql_host:
    external: true
  mysql_port:
    external: true
  mysql_name:
    external: true

  minio_bucket_name:
    external: true
  minio_host:
    external: true
  minio_port:
    external: true
  minio_access_key:
    external: true
  minio_secret_key:
    external: true
  minio_use_ssl:
    external: true
  minio_expiration_time:
    external: true

  jwt_algorithm:
    external: true
  jwt_expiration_time:
    external: true
  jwt_secret_key:
    external: true
```

Lampiran 2 – Summary JMeter untuk Pengujian Keepalived

timeStamp	elapsed	label	response(	responseI	threadName	dataType	success	failureMe	bytes	sentBytes	grpThread	allThreads	URL	Latency	IdleTime	Connect
1748943057362	305	HTTP Request	200	OK	Thread Group 1-1	text	TRUE		482	178102	1	1	http://192.168.1.20/api/v1/sirip-hiu	305	0	12
1748943057667	293	HTTP Request	200	OK	Thread Group 1-1	text	TRUE		482	178078	1	1	http://192.168.1.20/api/v1/sirip-hiu	293	0	0
1748943057961	269	HTTP Request	200	OK	Thread Group 1-1	text	TRUE		482	178094	1	1	http://192.168.1.20/api/v1/sirip-hiu	269	0	0
1748943058230	159	HTTP Request	200	OK	Thread Group 1-1	text	TRUE		482	178086	1	1	http://192.168.1.20/api/v1/sirip-hiu	159	0	0
1748943058389	213	HTTP Request	200	OK	Thread Group 1-1	text	TRUE		482	178126	1	1	http://192.168.1.20/api/v1/sirip-hiu	213	0	0
1748943058603	148	HTTP Request	200	OK	Thread Group 1-1	text	TRUE		482	178086	1	1	http://192.168.1.20/api/v1/sirip-hiu	148	0	0
1748943058751	164	HTTP Request	200	OK	Thread Group 1-1	text	TRUE		482	178110	1	1	http://192.168.1.20/api/v1/sirip-hiu	164	0	0
1748943058915	158	HTTP Request	200	OK	Thread Group 1-1	text	TRUE		482	178062	1	1	http://192.168.1.20/api/v1/sirip-hiu	158	0	0
1748943059073	147	HTTP Request	200	OK	Thread Group 1-1	text	TRUE		482	178078	1	1	http://192.168.1.20/api/v1/sirip-hiu	147	0	0
1748943059220	163	HTTP Request	200	OK	Thread Group 1-1	text	TRUE		482	178086	2	2	http://192.168.1.20/api/v1/sirip-hiu	163	0	0
1748943059363	174	HTTP Request	200	OK	Thread Group 1-2	text	TRUE		482	178094	1	1	http://192.168.1.20/api/v1/sirip-hiu	174	0	10
1748943059537	148	HTTP Request	200	OK	Thread Group 1-2	text	TRUE		482	178102	1	1	http://192.168.1.20/api/v1/sirip-hiu	148	0	0
1748943059685	160	HTTP Request	200	OK	Thread Group 1-2	text	TRUE		482	178054	1	1	http://192.168.1.20/api/v1/sirip-hiu	160	0	0
1748943059846	183	HTTP Request	200	OK	Thread Group 1-2	text	TRUE		482	178070	1	1	http://192.168.1.20/api/v1/sirip-hiu	183	0	0
1748943060029	183	HTTP Request	200	OK	Thread Group 1-2	text	TRUE		482	178126	1	1	http://192.168.1.20/api/v1/sirip-hiu	183	0	0
1748943060212	171	HTTP Request	200	OK	Thread Group 1-2	text	TRUE		482	178070	1	1	http://192.168.1.20/api/v1/sirip-hiu	171	0	0
1748943060383	159	HTTP Request	200	OK	Thread Group 1-2	text	TRUE		482	178054	1	1	http://192.168.1.20/api/v1/sirip-hiu	159	0	0
1748943060542	189	HTTP Request	200	OK	Thread Group 1-2	text	TRUE		482	178070	1	1	http://192.168.1.20/api/v1/sirip-hiu	189	0	0
1748943060731	140	HTTP Request	200	OK	Thread Group 1-2	text	TRUE		482	178118	1	1	http://192.168.1.20/api/v1/sirip-hiu	140	0	0
1748943060871	145	HTTP Request	200	OK	Thread Group 1-2	text	TRUE		482	178078	1	1	http://192.168.1.20/api/v1/sirip-hiu	145	0	0
1748943061362	244	HTTP Request	200	OK	Thread Group 1-3	text	TRUE		482	178134	1	1	http://192.168.1.20/api/v1/sirip-hiu	244	0	51
1748943061606	159	HTTP Request	200	OK	Thread Group 1-3	text	TRUE		482	178070	1	1	http://192.168.1.20/api/v1/sirip-hiu	159	0	0
1748943061765	193	HTTP Request	200	OK	Thread Group 1-3	text	TRUE		482	178118	1	1	http://192.168.1.20/api/v1/sirip-hiu	193	0	0
1748943061958	173	HTTP Request	200	OK	Thread Group 1-3	text	TRUE		482	178102	1	1	http://192.168.1.20/api/v1/sirip-hiu	173	0	0
1748943062131	170	HTTP Request	200	OK	Thread Group 1-3	text	TRUE		482	178086	1	1	http://192.168.1.20/api/v1/sirip-hiu	170	0	0
1748943062301	174	HTTP Request	200	OK	Thread Group 1-3	text	TRUE		482	178070	1	1	http://192.168.1.20/api/v1/sirip-hiu	174	0	0
1748943062475	149	HTTP Request	200	OK	Thread Group 1-3	text	TRUE		482	178134	1	1	http://192.168.1.20/api/v1/sirip-hiu	149	0	0
1748943062624	146	HTTP Request	200	OK	Thread Group 1-3	text	TRUE		482	178126	1	1	http://192.168.1.20/api/v1/sirip-hiu	146	0	0
1748943062770	152	HTTP Request	200	OK	Thread Group 1-3	text	TRUE		482	178126	1	1	http://192.168.1.20/api/v1/sirip-hiu	152	0	0
1748943062922	147	HTTP Request	200	OK	Thread Group 1-3	text	TRUE		482	178062	1	1	http://192.168.1.20/api/v1/sirip-hiu	147	0	0
1748943063364	183	HTTP Request	200	OK	Thread Group 1-4	text	TRUE		482	178062	1	1	http://192.168.1.20/api/v1/sirip-hiu	183	0	7
1748943063546	183	HTTP Request	200	OK	Thread Group 1-4	text	TRUE		482	178094	1	1	http://192.168.1.20/api/v1/sirip-hiu	183	0	0
1748943063729	150	HTTP Request	200	OK	Thread Group 1-4	text	TRUE		482	178054	1	1	http://192.168.1.20/api/v1/sirip-hiu	150	0	0
1748943063879	964	HTTP Request	Non HTTP	Non HTTP	Thread Group 1-4	text	FALSE		3000	0	1	1	http://192.168.1.20/api/v1/sirip-hiu	0	0	0
1748943064843	252	HTTP Request	200	OK	Thread Group 1-4	text	TRUE		482	178062	1	1	http://192.168.1.20/api/v1/sirip-hiu	252	0	7
1748943065095	155	HTTP Request	200	OK	Thread Group 1-4	text	TRUE		482	178086	1	1	http://192.168.1.20/api/v1/sirip-hiu	155	0	0
1748943065250	204	HTTP Request	200	OK	Thread Group 1-4	text	TRUE		482	178086	2	2	http://192.168.1.20/api/v1/sirip-hiu	204	0	0
1748943065454	252	HTTP Request	200	OK	Thread Group 1-4	text	TRUE		482	178094	2	2	http://192.168.1.20/api/v1/sirip-hiu	252	0	0
1748943065362	489	HTTP Request	200	OK	Thread Group 1-5	text	TRUE		482	178070	2	2	http://192.168.1.20/api/v1/sirip-hiu	489	0	10
1748943065706	238	HTTP Request	200	OK	Thread Group 1-4	text	TRUE		482	178126	2	2	http://192.168.1.20/api/v1/sirip-hiu	238	0	0
1748943065851	171	HTTP Request	200	OK	Thread Group 1-5	text	TRUE		482	178094	2	2	http://192.168.1.20/api/v1/sirip-hiu	171	0	0
1748943065944	157	HTTP Request	200	OK	Thread Group 1-4	text	TRUE		482	178134	2	2	http://192.168.1.20/api/v1/sirip-hiu	157	0	0
1748943066022	161	HTTP Request	200	OK	Thread Group 1-5	text	TRUE		482	178078	1	1	http://192.168.1.20/api/v1/sirip-hiu	161	0	0
1748943066183	152	HTTP Request	200	OK	Thread Group 1-5	text	TRUE		482	178078	1	1	http://192.168.1.20/api/v1/sirip-hiu	152	0	0
1748943066335	158	HTTP Request	200	OK	Thread Group 1-5	text	TRUE		482	178110	1	1	http://192.168.1.20/api/v1/sirip-hiu	158	0	0
1748943066493	163	HTTP Request	200	OK	Thread Group 1-5	text	TRUE		482	178062	1	1	http://192.168.1.20/api/v1/sirip-hiu	163	0	0
1748943066656	147	HTTP Request	200	OK	Thread Group 1-5	text	TRUE		482	178118	1	1	http://192.168.1.20/api/v1/sirip-hiu	147	0	0
1748943066803	146	HTTP Request	200	OK	Thread Group 1-5	text	TRUE		482	178134	1	1	http://192.168.1.20/api/v1/sirip-hiu	146	0	0
1748943066949	159	HTTP Request	200	OK	Thread Group 1-5	text	TRUE		482	178126	1	1	http://192.168.1.20/api/v1/sirip-hiu	159	0	0

(lanjutan)

1748943067108	141	HTTP Request	200	OK	Thread Group 1-5	text	TRUE		482	178086	1	1	http://192.168.1.20/api/v1/sirip-hiu	141	0	0
1748943067362	190	HTTP Request	200	OK	Thread Group 1-6	text	TRUE		482	178094	1	1	http://192.168.1.20/api/v1/sirip-hiu	190	0	10
1748943067552	168	HTTP Request	200	OK	Thread Group 1-6	text	TRUE		482	178054	1	1	http://192.168.1.20/api/v1/sirip-hiu	168	0	0
1748943067720	161	HTTP Request	200	OK	Thread Group 1-6	text	TRUE		482	178126	1	1	http://192.168.1.20/api/v1/sirip-hiu	161	0	0
1748943067881	147	HTTP Request	200	OK	Thread Group 1-6	text	TRUE		482	178070	1	1	http://192.168.1.20/api/v1/sirip-hiu	147	0	0
1748943068028	143	HTTP Request	200	OK	Thread Group 1-6	text	TRUE		482	178078	1	1	http://192.168.1.20/api/v1/sirip-hiu	143	0	0
1748943068171	169	HTTP Request	200	OK	Thread Group 1-6	text	TRUE		482	178126	1	1	http://192.168.1.20/api/v1/sirip-hiu	169	0	0
1748943068340	142	HTTP Request	200	OK	Thread Group 1-6	text	TRUE		482	178054	1	1	http://192.168.1.20/api/v1/sirip-hiu	142	0	0
1748943068483	157	HTTP Request	200	OK	Thread Group 1-6	text	TRUE		482	178086	1	1	http://192.168.1.20/api/v1/sirip-hiu	157	0	0
1748943068640	167	HTTP Request	200	OK	Thread Group 1-6	text	TRUE		482	178070	1	1	http://192.168.1.20/api/v1/sirip-hiu	167	0	0
1748943068807	150	HTTP Request	200	OK	Thread Group 1-6	text	TRUE		482	178062	1	1	http://192.168.1.20/api/v1/sirip-hiu	150	0	0
1748943069361	319	HTTP Request	200	OK	Thread Group 1-7	text	TRUE		482	178078	1	1	http://192.168.1.20/api/v1/sirip-hiu	319	0	8
1748943069680	190	HTTP Request	200	OK	Thread Group 1-7	text	TRUE		482	178054	1	1	http://192.168.1.20/api/v1/sirip-hiu	190	0	0
1748943069870	169	HTTP Request	200	OK	Thread Group 1-7	text	TRUE		482	178126	1	1	http://192.168.1.20/api/v1/sirip-hiu	169	0	0
1748943070039	156	HTTP Request	200	OK	Thread Group 1-7	text	TRUE		482	178054	1	1	http://192.168.1.20/api/v1/sirip-hiu	156	0	0
1748943070195	149	HTTP Request	200	OK	Thread Group 1-7	text	TRUE		482	178134	1	1	http://192.168.1.20/api/v1/sirip-hiu	149	0	0
1748943070344	154	HTTP Request	200	OK	Thread Group 1-7	text	TRUE		482	178062	1	1	http://192.168.1.20/api/v1/sirip-hiu	154	0	0
1748943070498	147	HTTP Request	200	OK	Thread Group 1-7	text	TRUE		482	178126	1	1	http://192.168.1.20/api/v1/sirip-hiu	147	0	0
1748943070645	137	HTTP Request	200	OK	Thread Group 1-7	text	TRUE		482	178078	1	1	http://192.168.1.20/api/v1/sirip-hiu	137	0	0
1748943070782	153	HTTP Request	200	OK	Thread Group 1-7	text	TRUE		482	178070	1	1	http://192.168.1.20/api/v1/sirip-hiu	153	0	0
1748943070935	142	HTTP Request	200	OK	Thread Group 1-7	text	TRUE		482	178078	1	1	http://192.168.1.20/api/v1/sirip-hiu	142	0	0
1748943071362	186	HTTP Request	200	OK	Thread Group 1-8	text	TRUE		482	178094	1	1	http://192.168.1.20/api/v1/sirip-hiu	186	0	25
1748943071548	151	HTTP Request	200	OK	Thread Group 1-8	text	TRUE		482	178054	1	1	http://192.168.1.20/api/v1/sirip-hiu	151	0	0
1748943071699	147	HTTP Request	200	OK	Thread Group 1-8	text	TRUE		482	178078	1	1	http://192.168.1.20/api/v1/sirip-hiu	147	0	0
1748943071846	145	HTTP Request	200	OK	Thread Group 1-8	text	TRUE		482	178078	1	1	http://192.168.1.20/api/v1/sirip-hiu	145	0	0
1748943071991	138	HTTP Request	200	OK	Thread Group 1-8	text	TRUE		482	178070	1	1	http://192.168.1.20/api/v1/sirip-hiu	138	0	0
1748943072129	138	HTTP Request	200	OK	Thread Group 1-8	text	TRUE		482	178070	1	1	http://192.168.1.20/api/v1/sirip-hiu	138	0	0
1748943072267	151	HTTP Request	200	OK	Thread Group 1-8	text	TRUE		482	178094	1	1	http://192.168.1.20/api/v1/sirip-hiu	151	0	0
1748943072418	142	HTTP Request	200	OK	Thread Group 1-8	text	TRUE		482	178070	1	1	http://192.168.1.20/api/v1/sirip-hiu	142	0	0
1748943072560	153	HTTP Request	200	OK	Thread Group 1-8	text	TRUE		482	178102	1	1	http://192.168.1.20/api/v1/sirip-hiu	153	0	0
1748943072713	139	HTTP Request	200	OK	Thread Group 1-8	text	TRUE		482	178126	1	1	http://192.168.1.20/api/v1/sirip-hiu	139	0	0
1748943073361	257	HTTP Request	200	OK	Thread Group 1-9	text	TRUE		482	178086	1	1	http://192.168.1.20/api/v1/sirip-hiu	257	0	41
1748943073618	164	HTTP Request	200	OK	Thread Group 1-9	text	TRUE		482	178054	1	1	http://192.168.1.20/api/v1/sirip-hiu	164	0	0
1748943073782	152	HTTP Request	200	OK	Thread Group 1-9	text	TRUE		482	178070	1	1	http://192.168.1.20/api/v1/sirip-hiu	152	0	0
1748943073934	143	HTTP Request	200	OK	Thread Group 1-9	text	TRUE		482	178054	1	1	http://192.168.1.20/api/v1/sirip-hiu	143	0	0
1748943074077	140	HTTP Request	200	OK	Thread Group 1-9	text	TRUE		482	178070	1	1	http://192.168.1.20/api/v1/sirip-hiu	140	0	0
1748943074217	147	HTTP Request	200	OK	Thread Group 1-9	text	TRUE		482	178054	1	1	http://192.168.1.20/api/v1/sirip-hiu	147	0	0
1748943074364	149	HTTP Request	200	OK	Thread Group 1-9	text	TRUE		482	178134	1	1	http://192.168.1.20/api/v1/sirip-hiu	149	0	0
1748943074514	135	HTTP Request	200	OK	Thread Group 1-9	text	TRUE		482	178078	1	1	http://192.168.1.20/api/v1/sirip-hiu	135	0	0
1748943074649	162	HTTP Request	200	OK	Thread Group 1-9	text	TRUE		482	178054	1	1	http://192.168.1.20/api/v1/sirip-hiu	162	0	0
1748943074811	150	HTTP Request	200	OK	Thread Group 1-9	text	TRUE		482	178086	1	1	http://192.168.1.20/api/v1/sirip-hiu	150	0	0
1748943075362	198	HTTP Request	200	OK	Thread Group 1-10	text	TRUE		482	178070	1	1	http://192.168.1.20/api/v1/sirip-hiu	198	0	39
1748943075560	155	HTTP Request	200	OK	Thread Group 1-10	text	TRUE		482	178110	1	1	http://192.168.1.20/api/v1/sirip-hiu	155	0	0
1748943075715	157	HTTP Request	200	OK	Thread Group 1-10	text	TRUE		482	178102	1	1	http://192.168.1.20/api/v1/sirip-hiu	157	0	0
1748943075872	147	HTTP Request	200	OK	Thread Group 1-10	text	TRUE		482	178134	1	1	http://192.168.1.20/api/v1/sirip-hiu	147	0	0
1748943076019	142	HTTP Request	200	OK	Thread Group 1-10	text	TRUE		482	178126	1	1	http://192.168.1.20/api/v1/sirip-hiu	142	0	0
1748943076161	151	HTTP Request	200	OK	Thread Group 1-10	text	TRUE		482	178086	1	1	http://192.168.1.20/api/v1/sirip-hiu	151	0	0
1748943076312	160	HTTP Request	200	OK	Thread Group 1-10	text	TRUE		482	178126	1	1	http://192.168.1.20/api/v1/sirip-hiu	160	0	0
1748943076472	138	HTTP Request	200	OK	Thread Group 1-10	text	TRUE		482	178110	1	1	http://192.168.1.20/api/v1/sirip-hiu	138	0	0
1748943076610	166	HTTP Request	200	OK	Thread Group 1-10	text	TRUE		482	178094	1	1	http://192.168.1.20/api/v1/sirip-hiu	166	0	0
1748943076776	152	HTTP Request	200	OK	Thread Group 1-10	text	TRUE		482	178110	1	1	http://192.168.1.20/api/v1/sirip-hiu	152	0	0

Lampiran 3 – Docker*file* layanan FastAPI

```
#
FROM python:3.10-slim

# Install curl dan tar
RUN apt-get update && apt-get install -y curl tar && rm -rf /var/lib/apt/lists/*

# Install wait4x binary
RUN curl -LO https://github.com/wait4x/wait4x/releases/latest/download/wait4x-linux-amd64.tar.gz \
    && tar -xzf wait4x-linux-amd64.tar.gz -C /usr/local/bin/ \
    && chmod +x /usr/local/bin/wait4x \
    && rm wait4x-linux-amd64.tar.gz

#
WORKDIR /app

#
COPY ./requirements.txt /app/requirements.txt

#
RUN pip install --no-cache-dir --upgrade -r /app/requirements.txt

#
COPY . .

# Set default command to wait for MySQL and run FastAPI
CMD ["sh", "-c", "wait4x tcp 192.168.1.20:3306 --timeout 60s --interval 2s && uvicorn app.main:app --host 0.0.0.0 --port 8000"]
```

Lampiran 4 – Docker*file* Layanan Flask

```
#
FROM python:3.10-slim

#
WORKDIR /app

#
COPY ./requirements.txt /app/requirements.txt

#
RUN pip install --no-cache-dir --upgrade -r /app/requirements.txt

#
COPY . .

#
CMD ["gunicorn", "--bind", "0.0.0.0:5000", "app.main:app"]
```

Lampiran 5 – Kunjungan ke tempat Pengusaha Sirip Hiu dan Loka PSPL

(lanjutan)