Jumistik
E-ISSN: 2964-3953 P-ISSN: 2964-3888
Jurnal Manajemen Informatika, Sistem Informasi dan Teknologi Komputer

Vol.4 No.2 Desember 2025
Hal 707-718
Https://ojs.amiklps.ac.id

# PEMANFAATAN *DOCKER* PADA VIRTUALISASI *SERVER* UNTUK MENGOPTIMALKAN KINERJA APLIKASI BERBASIS *WEB* MENGGUNAKAN *MULTISERVER*

**Ageng Eka Permana Putra[1*], Mukh. Taofik Chulkamdi[2] , Udkhiati Mawaddah[3]**
[1]Fakultas Teknik dan Informatika, Universitas Islam Balitar
E-mail: agengputra55@gmail.com[1*], chulkamdi@gmail.com[2], udkhiati.mawaddah@gmail.com[3]

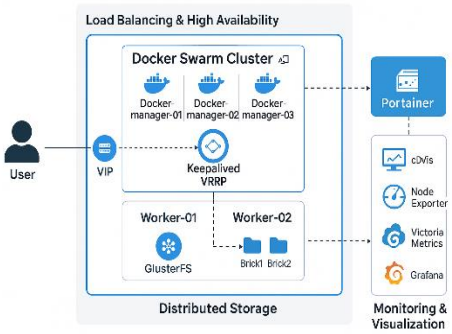
## INFO ARTIKEL

**Sejarah Artikel**
Diterima : 15/09/2025
Direvisi : 14/10/2025
Diterbitkan : 03/12/2025

***Corresponding author**
agengputra55@gmail.com

**DOI: 10.70247/jumistik.v4i2.203**
**Https://ojs.amiklps.ac.id**

## GRAPHICAL ABSTRACT




## *ABSTRACT*

*The management of web-based applications at SMKN 1 Nglegok still faces challenges in the efficient use of server resources. The increasing number of users and the growing complexity of applications often result in high CPU and memory loads, which lead to performance degradation. Docker-based server virtualization technology offers a solution through container isolation, scalability, and efficient resource management. This study aims to examine the effectiveness of implementing Docker in a Multiserver architecture to improve CPU and memory efficiency as well as the stability of web application performance. The research method used is experimental, comparing web application performance before and after Docker implementation using Apache JMeter for traffic simulation, K6 for load testing, and Grafana with Prometheus for monitoring. The measured parameters include CPU usage, memory usage, and response time under loads ranging from 200 to 1000 requests. The test results show a reduction in CPU usage by 20–35% and memory usage by 15–30% after Docker implementation. The application's response time remained stable below 200 ms even as the load increased up to 1000 requests. Validation by IT experts and User Acceptance Testing produced a user satisfaction level of more than 85% in the "Very Strong" category. The implementation of Docker in a Multiserver architecture proved effective in improving efficiency, maintaining performance stability, and meeting IT infrastructure management needs in an educational environment.*

***Keywords**: Docker, Virtualization, Multiserver, Web Applications*

## ABSTRAK

Pengelolaan aplikasi berbasis *web* di SMKN 1 Nglegok masih menghadapi tantangan dalam efisiensi penggunaan sumber daya *server*. Peningkatan jumlah pengguna dan kompleksitas aplikasi sering mengakibatkan beban tinggi pada CPU dan memori, yang berdampak pada penurunan performa. Teknologi virtualisasi *server* berbasis *Docker* menawarkan solusi dengan isolasi *container*, skalabilitas, dan manajemen sumber daya yang efisien. Penelitian ini bertujuan menguji efektivitas penerapan *Docker* pada arsitektur *Multiserver* dalam meningkatkan efisiensi penggunaan CPU dan memori, serta stabilitas kinerja aplikasi *web*. Metode penelitian yang digunakan adalah eksperimen, dengan membandingkan performa aplikasi *web* sebelum dan sesudah penerapan *Docker* menggunakan Apache JMeter untuk simulasi trafik, K6 untuk uji beban, serta *monitoring* dengan *Grafana* dan *Prometheus*. Parameter yang diukur meliputi penggunaan CPU, memori, dan waktu respons pada beban 200 hingga 1000 request. Hasil pengujian menunjukkan penurunan penggunaan CPU sebesar 20–35% dan memori 15–30% setelah penerapan *Docker*. Waktu respons aplikasi tetap stabil di bawah 200 ms meskipun terjadi peningkatan beban hingga 1000 request. Validasi oleh ahli IT dan *User Acceptance Testing* menghasilkan tingkat kepuasan pengguna lebih dari 85% dengan kategori "Sangat Kuat". Penerapan *Docker* pada arsitektur *Multiserver* terbukti efektif

dalam meningkatkan efisiensi, menjaga stabilitas performa aplikasi *web*, dan memenuhi kebutuhan pengelolaan infrastruktur TI di lingkungan pendidikan.

**Kata kunci**: *Docker*, Virtualisasi, *Multiserver*, Aplikasi *Web*



## PENDAHULUAN

Kemampuan sistem *web* untuk melakukan tugas dengan efisiensi tinggi, responsivitas dalam melayani permintaan pengguna, dan stabilitas dalam berbagai kondisi penggunaan merupakan hal penting dalam kinerja aplikasi berbasis *web*. Menurut penelitian terbaru, penggunaan virtualisasi *server* berbasis *Docker container* dapat meningkatkan efisiensi kinerja aplikasi *web* hingga 30% dibandingkan tanpa *container*, menunjukkan peningkatan yang signifikan dalam penggunaan sumber daya *server*. Selain itu, penggunaan *container* untuk *load balancing web server* juga dapat meningkatkan responsivitas aplikasi saat menghadapi lonjakan permintaan tiba-tiba. Implementasi algoritma *Round Robin* pada *Docker* juga dapat membantu mendistribusikan beban dengan lebih merata, mengurangi waktu respons *server* [1].

Pengelolaan sumber daya yang efisien pada *container Docker* menjadi kunci optimalisasi penyebaran aplikasi, yang secara langsung berkontribusi terhadap stabilitas dan kinerja aplikasi *web*. Teknologi *container* mampu mengoptimalkan kinerja, responsivitas, dan stabilitas aplikasi *web* melalui manajemen sumber daya yang lebih efektif. Faktor-faktor seperti efisiensi penggunaan CPU, memori, serta teknik *load balancing* berperan penting dalam menjaga performa aplikasi. Integrasi *Docker* dengan layanan *cloud computing* juga terbukti meningkatkan skalabilitas dan kecepatan akses aplikasi, serta mempercepat proses *deployment* [2]. Skema penempatan *container* yang dinamis dan sadar sumber daya memberikan keunggulan tambahan, terutama untuk aplikasi data-intensif pada kluster heterogen. Integrasi temuan dari berbagai penelitian menunjukkan bahwa implementasi *Docker* dalam aplikasi *web* tidak hanya meningkatkan efisiensi dan responsivitas, tetapi juga memungkinkan aplikasi untuk beradaptasi dengan perubahan beban kerja dan kebutuhan pengguna dengan lebih cepat [3].

Beberapa penelitian telah membuktikan kontribusi *Docker* terhadap peningkatan performa aplikasi *web*. *Docker* dapat meningkatkan efisiensi penggunaan sumber daya *server* dengan implementasi yang cepat dan sederhana, sehingga cocok untuk *server* skala kecil hingga menengah. Proses *deployment* aplikasi menjadi lebih mudah dan cepat dengan *Docker*, karena konflik konfigurasi dapat diminimalkan [4]. Penerapan algoritma *Round Robin* pada *Docker* menghasilkan distribusi beban kerja yang lebih merata di *web server* [5]. Kombinasi *Docker* dan layanan *cloud computing* dapat meningkatkan skalabilitas serta performa aplikasi *web* [6]. Infrastruktur berbasis *Google Cloud Platform* memungkinkan layanan *web* service yang lebih efisien dan mudah diskalakan [7]. Penggunaan *Zimbra* dengan fitur *Cbpolicyd* yang diperbarui berhasil meningkatkan performa *load balancing* pada sistem mail *server* [8]. *Docker* mampu mengurangi latensi akses file sistem dibandingkan sistem operasi native, sehingga meningkatkan performa aplikasi berbasis *file system* [9].

Namun, sebagian besar penelitian masih berfokus pada skala perusahaan dan infrastruktur *cloud* besar, belum banyak membahas penerapan *Docker Multiserver* di institusi pendidikan. Padahal, sekolah menengah kejuruan seperti SMKN 1 Nglegok menghadapi tantangan serupa dalam pengelolaan aplikasi berbasis *web*, termasuk keterbatasan perangkat keras, peningkatan jumlah pengguna, serta kompleksitas aplikasi. Oleh karena itu, diperlukan strategi pengelolaan *server* yang efisien untuk menjamin kualitas layanan pendidikan berbasis teknologi.

Studi kasus di SMKN 1 Nglegok menunjukkan bahwa peningkatan jumlah pengguna aplikasi *web* sekolah menyebabkan beban tinggi pada CPU dan memori, yang berdampak pada penurunan performa. Dengan penerapan *Docker* dalam arsitektur *Multiserver*, diharapkan efisiensi penggunaan sumber daya dapat meningkat, beban kerja terdistribusi lebih merata, dan waktu respons aplikasi tetap stabil. *Docker*, dengan isolasi aplikasinya, memberikan fleksibilitas dalam pengelolaan *server* serta konsistensi lingkungan dari pengembangan hingga produksi.

Urgensi penelitian ini terletak pada pengembangan model pengelolaan infrastruktur TI berbasis *Docker Multiserver* yang berkelanjutan di lingkungan pendidikan. Pendekatan ini tidak hanya bertujuan meningkatkan kinerja aplikasi, tetapi juga mempersiapkan infrastruktur menghadapi beban kerja yang fluktuatif. Selain itu, keberhasilan penerapan *Docker* dapat meningkatkan kapasitas guru dan staf IT dalam mengelola infrastruktur modern, yang sejalan dengan tuntutan era digital. Fokus penelitian ini adalah penerapan virtualisasi *server* melalui *Docker* untuk meningkatkan performa aplikasi *web* dengan sistem *Multiserver* di SMKN 1 Nglegok. *Docker* memungkinkan isolasi aplikasi secara efisien, mendukung fleksibilitas pengelolaan sumber daya, serta meningkatkan ketahanan aplikasi terhadap beban kerja dinamis. Implementasi *Multiserver* secara strategis memberikan kecepatan respons dan efisiensi

operasional yang lebih baik. Dengan demikian, penggunaan *Docker* tidak hanya menawarkan keunggulan teknis, tetapi juga mendukung inovasi pendidikan berbasis TI.

Penerapan *Docker* di SMKN 1 Nglegok diharapkan memberikan optimisasi signifikan pada kinerja aplikasi *web*. Keuntungan utama meliputi keandalan operasional, konsistensi lingkungan, dan efisiensi distribusi beban kerja. Penelitian sebelumnya menunjukkan bahwa *container*isasi mampu mengurangi downtime dan meningkatkan responsivitas saat beban tinggi [8]. Namun, tantangan yang dihadapi adalah kesiapan infrastruktur serta pemahaman teknis staf IT dan tenaga pengajar. Oleh karena itu, selain meningkatkan efisiensi teknis, penelitian ini juga menekankan pentingnya aspek kesiapan sumber daya manusia. Sebagai langkah selanjutnya, penelitian ini bertujuan menguji efektivitas *Docker* dalam meningkatkan efisiensi penggunaan CPU dan memori, menjaga stabilitas waktu respons aplikasi, serta mengukur kepuasan pengguna melalui evaluasi teknis dan lapangan. Hasil yang diperoleh diharapkan dapat memberikan kontribusi praktis bagi SMKN 1 Nglegok sekaligus menambah literatur tentang pemanfaatan *Docker* di sektor pendidikan. Penelitian ini juga diharapkan menjadi model penerapan yang dapat direplikasi oleh institusi lain dalam menghadapi tantangan pengelolaan sumber daya TI yang semakin kompleks.

## TINJAUAN PUSTAKA

### Virtualisasi *Server*

Virtualisasi *server* merupakan teknik penggunaan perangkat lunak untuk menciptakan lapisan abstraksi di atas perangkat keras komputer yang mendasarinya, sehingga memungkinkan *server* fisik untuk dibagi menjadi beberapa mesin virtual yang lebih kecil dan mandiri. Tujuan dari virtualisasi *server* adalah untuk meningkatkan efisiensi penggunaan sumber daya dengan menjalankan beberapa sistem operasi secara bersamaan pada satu *server* fisik. Dengan teknologi virtualisasi, berbagai aplikasi atau layanan dapat berjalan dalam lingkungan terisolasi, yang memungkinkan penggunaan sumber daya perangkat keras yang tersedia menjadi maksimal. Hal ini membuat pengelolaan sumber daya menjadi lebih fleksibel dan dinamis, sehingga kapasitas dapat disesuaikan tanpa perlu menambah perangkat keras fisik baru [10].

### *Container*

Teknologi *container* adalah sebuah teknologi virtualisasi yang memungkinkan aplikasi dan dependensinya diisolasi dalam sebuah unit terpisah yang ringan. Dengan teknologi ini, semua aplikasi beserta dependensinya dikemas dalam satu paket agar dapat berjalan dengan konsisten di berbagai sistem komputasi. Keunggulan utama dari *container* adalah kemampuannya untuk beroperasi efisien di atas sistem operasi host tanpa memerlukan sistem operasi tambahan, berbeda dengan virtual machine yang membutuhkan virtualisasi hardware. *Container* sering kali diimplementasikan dengan menggunakan *Docker*, teknologi yang populer untuk distribusi dan *deployment* aplikasi [11].

### *Docker*

*Docker* merupakan suatu platform yang memungkinkan pengguna untuk melakukan penyebaran, pengoperasian, dan pengelolaan aplikasi dalam wadah terisolasi yang dikenal sebagai *container*. Wadah ini berperan sebagai unit standar perangkat lunak yang mengemas kode serta semua dependensinya agar aplikasi dapat berjalan dengan konsisten di berbagai lingkungan komputer. Dengan menggunakan *container*, *Docker* memastikan bahwa aplikasi dapat dijalankan dengan cara yang seragam mulai dari sistem pengembangan hingga sistem produksi, yang memberikan stabilitas dan kemudahan pengelolaan bagi pengembang dan administrator sistem [3].

### Arsitektur *Multiserver*

Arsitektur *Multiserver* adalah pendekatan dalam pengaturan sistem komputer yang menggunakan beberapa *server* secara bersamaan untuk meningkatkan kapasitas dan kehandalan layanan aplikasi. Dengan membagi beban kerja di antara *server-server* tersebut, arsitektur ini dapat mengatasi kebutuhan pemrosesan yang lebih besar daripada menggunakan satu *server* tunggal. Distribusi tugas yang efisien memungkinkan peningkatan kinerja sistem secara keseluruhan dan menjaga keseimbangan beban kerja di antara *server-server* yang ada [12].

### Implementasi *Load balancing*

Implementasi *load balancing* adalah proses distribusi beban kerja secara merata di antara sejumlah *server* untuk memastikan kinerja optimal dan ketersediaan layanan yang tinggi. Dalam konteks virtualisasi *server* berbasis *Docker*, *load balancing* dapat meningkatkan efisiensi manajemen sumber daya serta mengurangi waktu respons terhadap permintaan aplikasi *web*. Dengan menggunakan *container Docker*, proses penyeimbangan beban dapat dilakukan dengan lebih fleksibel dan cepat beradaptasi terhadap perubahan kebutuhan aplikasi. Keunggulan utama dari pendekatan ini terletak pada kapasitas untuk melakukan skalabilitas secara dinamis, sehingga memungkinkan penambahan atau pengurangan instance *server* tanpa mengganggu aktivitas pengguna [13].

### Optimasi Aplikasi Berbasis *Web*

Optimasi aplikasi melibatkan usaha untuk meningkatkan efisiensi dan performa aplikasi dengan memanfaatkan sumber daya yang ada secara maksimal. Salah satu pendekatan yang menonjol dalam hal ini adalah penggunaan teknologi *container*, seperti *Docker*, yang memungkinkan aplikasi beroperasi dengan efisien dan hemat sumber

daya dibandingkan dengan teknik virtualisasi tradisional. Salah satu aspek kunci dari optimasi aplikasi adalah alokasi sumber daya yang efektif. Strategi alokasi sumber daya yang tepat dapat mempengaruhi performa aplikasi, terutama untuk aplikasi yang membutuhkan pengolahan data yang intensif. Pemanfaatan *container* dapat memberikan keuntungan dengan skema alokasi sumber daya yang dinamis dan responsif [14].

***Web Server***

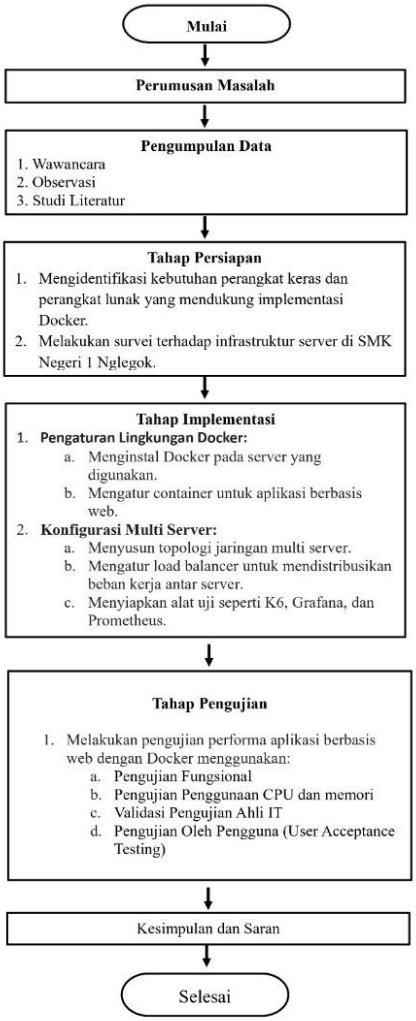
*Web server* adalah sebuah perangkat lunak yang berfungsi sebagai penghubung antara pengguna dengan data yang tersimpan di dalam *server* melalui jaringan internet. Secara khusus, *web server* bertugas menerima permintaan (request) dari pengguna melalui *browser web*, memproses permintaan tersebut, dan mengirimkan kembali hasilnya dalam bentuk halaman *web* yang dapat dilihat oleh pengguna. Peran *web server* menjadi penting karena bertindak sebagai perantara utama yang memastikan komunikasi yang efektif antara klien dan aplikasi *web* yang berjalan di *server*. Dengan demikian, *web server* memainkan peranan krusial dalam menyajikan aplikasi berbasis *web* kepada pengguna dengan cara yang efisien dan cepat. Kenyamanan pengguna dalam mengakses aplikasi sangat bergantung pada kemampuan *web server* untuk menangani berbagai permintaan secara simultan tanpa kehilangan performa[15].

Berbeda dengan penelitian sebelumnya yang umumnya mengulas manfaat penggunaan *Docker* dalam berbagai bidang seperti industri, *cloud computing*, dan simulasi *load balancing* tanpa spesifikasi lokasi tertentu, penelitian ini difokuskan pada penerapan *Docker* dalam sistem *Multiserver* di lingkungan pendidikan, terutama di SMKN 1 Nglegok. Dengan pendekatan eksperimental langsung, penelitian ini bertujuan untuk menguji dampak penggunaan *Docker* terhadap efisiensi CPU, memori, dan responsivitas aplikasi *web* yang sedang berjalan. Selain mempelajari konsep efisiensi dan *load balancing*, penelitian ini juga memberikan kontribusi praktis dengan merancang strategi implementasi *Docker* yang relevan dan dapat diadopsi di institusi pendidikan serupa, menjadikannya sebagai studi kasus yang lebih terfokus dan kontekstual daripada penelitian sebelumnya.

## METODOLOGI PENELITIAN

### Tahapan Penelitian

Penelitian ini bersifat eksperimental dengan tujuan menguji pengaruh penerapan *Docker* pada arsitektur *Multiserver* terhadap kinerja aplikasi berbasis *web*, diukur melalui penggunaan CPU dan memori, dengan pengamatan dilakukan sebelum dan sesudah implementasi untuk memperoleh data yang valid, reliabel, dan objektif di SMK Negeri 1 Nglegok.




**Gambar 1.** Tahapan Penelitian
(*Sumber : Hasil Olah Data)*

1. Perumusan Masalah
   Dalam penelitian ini, rendahnya kinerja aplikasi *web* di SMK Negeri 1 Nglegok, yang ditandai oleh waktu respons lambat, throughput tidak konsisten, dan penggunaan sumber daya yang kurang efisien akibat virtualisasi yang kurang optimal, mendorong penerapan teknologi *Docker* pada *server Multiserver* untuk mengevaluasi peningkatan kinerja secara efektif dan efisien.
2. Pengumpulan Data
   Dalam penelitian ini, data dikumpulkan melalui observasi kondisi *server*, interaksi dengan teknisi dan guru pengajar, serta telaah pustaka literatur terkait, yang kemudian dianalisis untuk mengidentifikasi kendala infrastruktur dan kebutuhan sistem di SMK Negeri 1 Nglegok.
3. Tahap Persiapan
   Tahap persiapan implementasi *Docker* dimulai dengan identifikasi kebutuhan perangkat keras dan perangkat lunak, survei infrastruktur *server* di SMK Negeri 1 Nglegok, serta penyiapan alat pengujian performa seperti K6, *Grafana*, dan *Prometheus*, yang secara keseluruhan

memastikan kesiapan sistem, kompatibilitas, dan kemampuan *monitoring* untuk mendukung implementasi *Docker* pada topologi *Multiserver*.

4. Tahap Implementasi
   Tahap implementasi *Docker* meliputi pengaturan lingkungan *Docker* melalui instalasi dan konfigurasi *container* untuk menjalankan aplikasi secara terisolasi, serta konfigurasi *Multiserver* dengan integrasi *server* dan penggunaan *Load Balancer* untuk mendistribusikan beban kerja, sehingga meningkatkan efisiensi, skalabilitas, responsivitas, dan ketersediaan sistem.
5. Tahap Pengujian
   Tahap pengujian ini mencakup pengukuran performa teknis sistem seperti waktu respons, throughput, dan penggunaan sumber daya, uji fungsional untuk memastikan semua fitur berjalan sesuai rancangan, validasi oleh ahli IT untuk menilai kualitas, keamanan, dan kemampuan sistem menangani beban tinggi, serta uji pengguna (UAT) untuk mengevaluasi kemudahan penggunaan, stabilitas, dan kepuasan pengguna akhir.
6. Kesimpulan dan Saran
   Penelitian ini ditutup dengan penyampaian kesimpulan dan saran yang diperoleh berdasarkan temuan dan penelitian yang telah dilakukan

## Pengumpulan Data

Dalam penelitian ini, pengumpulan data dilakukan melalui tiga metode utama, yaitu observasi, interaksi, dan telaah pustaka.

a. Observasi
   Observasi dilakukan untuk mengamati langsung kondisi infrastruktur *server* di SMK Negeri 1 Nglegok sebelum penerapan *Docker*, termasuk kapasitas CPU, memori, dan arsitektur jaringan yang ada. Hasilnya menunjukkan keterbatasan sumber daya, konfigurasi *server* yang kurang optimal, serta penurunan performa saat terjadi lonjakan pengguna.
b. Wawancara
   Interaksi dilakukan melalui wawancara dan diskusi dengan teknisi jaringan dan guru pengajar untuk memahami kendala, kebutuhan sistem, dan harapan terhadap penerapan *Docker*. Temuannya mengungkap kesulitan dalam konfigurasi manual, kurangnya integrasi antar *server*, dan kebutuhan akan solusi yang lebih efisien serta stabil.
c. Telaah Pustaka
   Telaah pustaka dilakukan dengan meneliti literatur dan jurnal ilmiah terkait virtualisasi *server* dan *Docker* untuk memperoleh dasar teori dan praktik terbaik. Hasil kajian menunjukkan bahwa *Docker* dapat meningkatkan efisiensi sumber daya, mempercepat *deployment*, serta meningkatkan stabilitas dan performa sistem *Multiserver*.

## Tahapan Persiapan

Tahap persiapan merupakan langkah awal yang penting untuk memastikan kelancaran implementasi *Docker*. Langkah pertama adalah mengidentifikasi kebutuhan perangkat keras dan perangkat lunak, termasuk mengecek kapasitas CPU, memori, *storage*, serta kompatibilitas sistem operasi. Selanjutnya, dilakukan survei infrastruktur *server* untuk menilai kemampuan *server* menangani beban kerja, stabilitas jaringan, dan potensi masalah yang ada. Terakhir, disiapkan alat pengujian performa seperti K6, *Grafana*, dan *Prometheus* untuk memastikan setiap perubahan dari implementasi *Docker* dapat dianalisis secara objektif. Rincian tahapan persiapan dapat dilihat pada Tabel 1.

**Tabel 1.** Tahap Persiapan Implementasi *Docker*

| No | Tahap Persiapan | Keterangan |
|---|---|---|
| 1 | Identifikasi Kebutuhan | Menilai kapasitas *server* (CPU, memori, *storage*), kompatibilitas OS, dan software pendukung (K6, *Grafana*, *Prometheus*). |
| 2 | Survey Infrastruktur | Memeriksa kondisi *server*, stabilitas jaringan, potensi masalah, menentukan kebutuhan *Multiserver* |
| 3 | Persiapan Alat Pengujian | Menyiapkan K6, *Grafana*, dan *Prometheus* untuk pengujian performa |
| 4 | Dokumentasi dan Perencanaan | Mencatat hasil survei dan menyusun rencana topologi *Docker Multiserver* |

## HASIL DAN PEMBAHASAN

### 1) Hasil Analisis Kebutuhan Sistem

Berdasarkan observasi selama dua minggu di laboratorium komputer SMKN 1 Nglegok, ditemukan beberapa masalah kritis, yaitu *server* mengalami overload saat ujian online, aplikasi *web* saling memengaruhi karena tidak terisolasi, dan sistem kesulitan melakukan scaling saat terjadi lonjakan trafik. Kebutuhan teknis yang teridentifikasi meliputi virtualisasi ringan dengan overhead minimal, kemampuan horizontal scaling, dan sistem *monitoring* terintegrasi. Solusi yang diterapkan adalah *Docker* dengan *container* terpisah untuk setiap aplikasi, *Docker Swarm* untuk *clustering*, serta *Nginx* sebagai *Load Balancer*. Identifikasi kebutuhan perangkat keras dan perangkat lunak digunakan sebagai dasar penyiapan *server* dan lingkungan *Docker* sebelum implementasi, sebagaimana dirangkum pada Tabel 2.

**Tabel 2.** Kebutuhan Sistem

| Kategori | Kebutuhan | Spesifikasi Masalah |
|---|---|---|
| Perangkat Keras (Hardware) | Laptop | - |
| | Prosesor | Intel Core i3 atau setara |
| | RAM | 4 GB atau lebih |
| | Hardisk | 256 GB SSD atau lebih |
| | *Server* atau Komputer | - |
| | Prosesor | 4 x Intel(R) Xeon(R) CPU E3-1225 v3 @ 3.20GHz |
| | RAM | 8 GB atau lebih |
| | Hardisk | 500 hdd atau lebih |
| Perangkat Lunak (Software) | Sistem Operasi | Windows 10, promox dan ubuntu *server* |
| | Platform Virtualisasi | *Docker Engine* |

| Kategori | Kebutuhan | Spesifikasi Masalah |
|---|---|---|
| | Bahasa Pemrograman | Python 3 (module pandas, os) |
| | Orchestration | *Docker Swarm* mode |
| | *Monitoring* | *Prometheus + Grafana* |

## 2) Hasil Implementasi

### a. Pengaturan Lingkungan *Docker*

Arsitektur *Docker* diimplementasikan pada 5 *node server* dengan spesifikasi *Intel Xeon 4 core*, 8GB RAM, dan HDD 50GB. Setiap *node* menjalankan *container* terpisah untuk aplikasi *web*, *database*, dan *monitoring*. *Dockerfile* disusun khusus untuk membangun image aplikasi *web* berbasis *Nginx*, sementara *Docker-compose.yml* digunakan untuk mengatur komunikasi antar *container*.

**Instalasi *Docker* pada *Server***

*Docker* diinstal dengan *command* seperti pada tabel 3, pada semua *node* (baik *manager* maupun worker) untuk menyediakan runtime kontainer standar. Tahap ini juga menyiapkan dependensi seperti transport HTTPS dan kunci GPG untuk *repository* resmi *Docker*.

**Tabel 3.** *Command* Instalasi Paket

```
apt update && apt upgrade -y
apt-get install -y apt-transport-https
ca-certificates curl gnupg
```

Perintah (*command*) pada tabel 3 merupakan proses instalasi dimulai dengan mempersiapkan sistem dan menginstal paket-paket pendukung. Konfigurasi selanjutnya tambahkan repositori *Docker* dengan mengunakan perintah pada tabel 4 berikut.

**Tabel 4.** *Command* Instalasi Package Repositori *Docker*

```
curl -fsSL
https://download.Docker.com/linux/ubuntu
/gpg | sudo gpg --dearmor -o
/etc/apt/keyrings/Docker.gpg
echo "deb [arch=$(dpkg --print-
architecture) signed-
by=/etc/apt/keyrings/Docker.gpg]
https://download.Docker.com/linux/ubuntu
$(. /etc/os-release && echo
"$VERSION_CODENAME") stable" | sudo tee
etc/apt/sources.list.d/Docker.list >
/dev/null
```

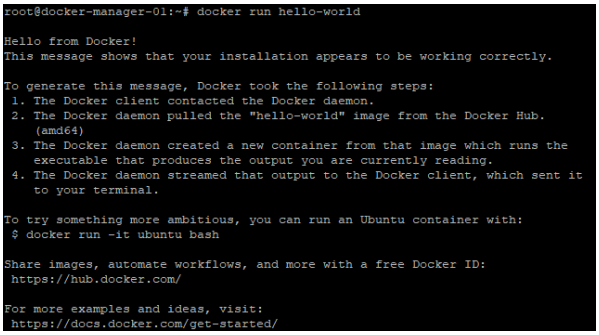
Selanjutnya, *Docker Engine* dan plugin tambahan diinstal menggunakan perintah pada tabel 5.

**Tabel 5.** *Command* Instalasi *Docker Engine*

```
apt update && apt-get install -y
Docker-ce Docker-ce-cli containerd.io
Docker-buildx-plugin Docker-compose-
plugin
```

Keberhasilan instalasi diuji dengan menjalankan perintah *Docker* run hello-world, yang menampilkan konfirmasi bahwa *Docker* client berhasil terhubung ke *Docker* daemon, menarik image "hello-world" dari *Docker* Hub, dan menjalankan *container* baru.

Pada gambar 2 menampilkan pesan konfirmasi bahwa instalasi *Docker* berhasil dan berfungsi dengan baik. Saat perintah dijalankan, *Docker* client terhubung ke *Docker* daemon, menarik *(pull) image "hello-world*" dari *Docker* Hub (arsitektur amd64), membuat *container* baru dari image tersebut, menjalankan program di dalamnya yang mencetak pesan ke terminal, lalu mengirim output itu kembali ke pengguna.




**Gambar 2.** Pengujian Package *Docker* yang Telah Di Instal (Sumber : dokumen pribadi)

**Mengatur *Container***

Konfigurasi setelah instalasi *Docker* pada *server* adalah mengatur *Container* yang bertujuan untuk komunikasi antar *node* atau *server*. Perintah pertama yang dilakukan yaitu membuat direktori untuk setifikat seperti pada tabel 6.

**Tabel 6.** *Command* Direktori Setifikat

```
mkdir -p ~/Docker-certs && cd
~/Docker-certs
```

Gambar 3 menampilkan daftar file sertifikat dan direktori *Docker*-certs pada *server*, yang digunakan untuk menyimpan kunci, sertifikat CA, dan konfigurasi ekstensi yang diperlukan dalam pengamanan komunikasi *Docker* API dengan TLS. Konfigurasi selanjutnya membuat sertifikat TLS untuk *Docker* API.


**Gambar 3.** Directori Yang Telah Di Buat (Sumber : dokumen pribadi)

Perintah di table 7 digunakan untuk membuat private key CA bernama ca-key.pem dengan panjang 4096 bit, lalu menghasilkan sertifikat self-signed ca.pem yang berlaku 365 hari menggunakan kunci tersebut, dengan informasi subjek lokasi di Blitar dan organisasi Universitas

Islam Blitar, Konfigurasi selanjutnya untuk mengunakan TLS.

**Tabel 7.** *Command* Pembuatan Sertifikat TLS Untuk *Docker* API

```
openssl genrsa -out ca-key.pem 4096
openssl req -new -x509 -days 365 -key
ca-key.pem -sha256 -out ca.pem -subj
"/C=ID/ST=East
Java/L=Blitar/O=Universitas Islam
Blitar/OU=/CN=server"
```

Perintah di tabel 8 mengedit baris ExecStart pada file service *Docker* agar daemon mendengarkan koneksi TLS terenkripsi di port 2376 dengan sertifikat yang ditentukan, lalu memuat ulang konfigurasi systemd dan me-*restart* layanan *Docker* agar perubahan diterapkan.

**Tabel 8.** *Command* Konfigurasi *Docker* Untuk Menggunakan TLS

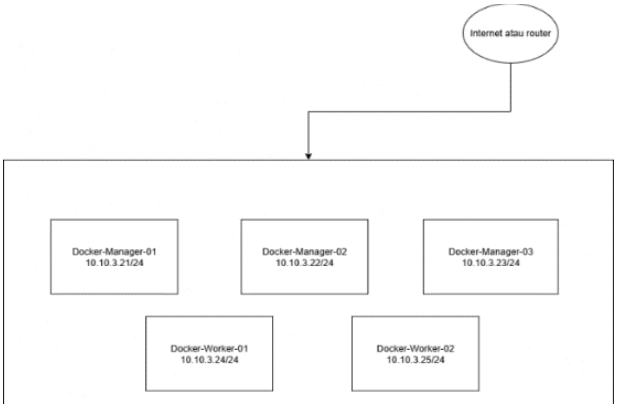
```
sudo sed -i
'/ExecStart=/c\ExecStart=/usr/bin/Docker
d -H unix:///var/run/Docker.sock -H
tcp://0.0.0.0:2376 --tlsverify --
tlscacert=/etc/ssl/certs/ca-Docker.pem -
-tlscert=/etc/ssl/certs/Docker-cert.pem
--tlskey=/etc/ssl/private/Docker-
key.pem'
/lib/systemd/system/Docker.service
sudo systemctl daemon-reload
sudo systemctl restart Docker
```

**b. Konfigurasi *Multiserver***

Jaringan *Multiserver* dibangun menggunakan *Docker Overlay Network* dengan tiga *node manager* dan dua *node* worker. *Load Balancer Nginx* dikonfigurasi dengan algoritma *Round Robin* untuk mendistribusikan beban kerja secara merata ke semua *node*. Hasilnya menunjukkan distribusi beban yang seimbang dengan masing-masing *node*.

**Menyusun Topologi Jaringan**

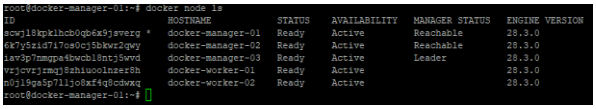
Gambar 4 menunjukkan diagram topologi *cluster Docker Swarm* yang terhubung ke internet atau router, terdiri dari tiga *node manager* (10.10.3.21–23/24) dan dua *node* worker (10.10.3.24–25/24) dalam satu jaringan lokal. Konfigurasi selanjutnya melakukan inisialisasi *Swarm* pada *manager* utama atau pada *Docker-Manager*-01.




**Gambar 4.** Topologi *Multiserver (Sumber : dokumen pribadi)*

Proses konfigurasi *cluster Docker Swarm* dimulai dengan inisialisasi *node manager* utama. *Node* ini dikonfigurasi untuk menjadi pusat pengelolaan *cluster* dengan menetapkan alamat IP sebagai alamat iklan, alamat pendengaran, dan jalur data internal. Selain itu, masa berlaku sertifikat untuk komunikasi antar *node* diatur agar lebih panjang, sehingga keamanan *cluster* tetap terjaga tanpa sering memperbarui sertifikat. Setelah *manager* utama aktif, *node manager* tambahan (*manager* 2–3) dan worker (worker 1–2) ditambahkan ke dalam *cluster*. Setiap *node* bergabung menggunakan token khusus yang menandai perannya sebagai *manager* atau worker, serta mengacu pada alamat *manager* utama sebagai titik awal komunikasi dalam *cluster*.

Untuk memastikan semua *node* berhasil terhubung dan *cluster* berjalan dengan baik, dilakukan verifikasi status *cluster*. Hasil verifikasi, seperti yang terlihat pada Gambar 5, menunjukkan bahwa semua *node* berada dalam status *Ready* dan *Active*. Tiga *node manager* terlihat dengan satu *node* berperan sebagai Leader dan dua lainnya Reachable, sementara dua *node* worker siap menerima beban kerja. Semua *node* menjalankan *Docker Engine* versi 28.3.0, menandakan konsistensi versi dan kesiapan *cluster* untuk *deployment* layanan lebih lanjut.




**Gambar 5.** Verifikasi *Cluster (Sumber : dokumen pribadi)*

**Mengatur *Load Balancer***

Untuk mengatur *Load Balancer* pada *cluster Docker*, langkah pertama adalah menginstal *Keepalived* di semua *node manager* untuk menyediakan virtual IP yang selalu aktif pada *node* sehat. Setelah itu, buat skrip *Health Check Docker* yang memeriksa status daemon agar *Keepalived* bisa menentukan *node* mana yang layak menjadi master. Konfigurasikan *Keepalived*

dengan menetapkan virtual IP, prioritas, dan pengaturan *Health Check* agar *failover* berjalan otomatis. Setelah konfigurasi selesai, mulai dan aktifkan layanan *Keepalived* di semua *manager node* agar sistem *Load Balancer* siap digunakan. Terakhir, deploy *Portainer* sebagai antarmuka manajemen *container* yang mudah digunakan, lalu buat atau impor konfigurasi *stack* di *Portainer* untuk mengelola service dan jaringan *cluster* secara terpusat.

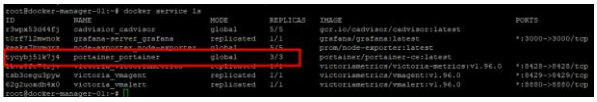
Virtual IP (VIP) digunakan untuk menyediakan *High Availability* (HA) pada *cluster Docker Swarm*, sehingga jika *node* master gagal, salah satu *node* backup otomatis mengambil alih peran master dan memegang VIP. Konfigurasi *Keepalived* dengan skrip *Health Check Docker* memastikan hanya *node* dengan *Docker Engine* berjalan normal yang dapat memegang VIP, menjaga konsistensi layanan. Setelah *Keepalived* dijalankan dan diaktifkan menggunakan systemctl enable --now *Keepalived*, semua *node* saling sinkron melalui *unicast peer*, sehingga *failover* berjalan lancar dan *cluster* tetap tersedia tanpa gangguan.

Gambar 4.5 menampilkan hasil perintah ip -br a pada tiga *node manager Docker*. Dari gambar terlihat bahwa interface ens18 pada masing-masing *node* dalam keadaan UP, dengan alamat IP masing-masing: 10.10.3.21, 10.10.3.22, dan 10.10.3.23. Selain itu, pada *node Docker-manager*-01 terlihat virtual IP 10.10.3.20 yang aktif, menandakan bahwa *node* ini berperan sebagai MASTER VRRP.


**Gambar 6.** Testing *Keepalived (Sumber : dokumen pribadi)*

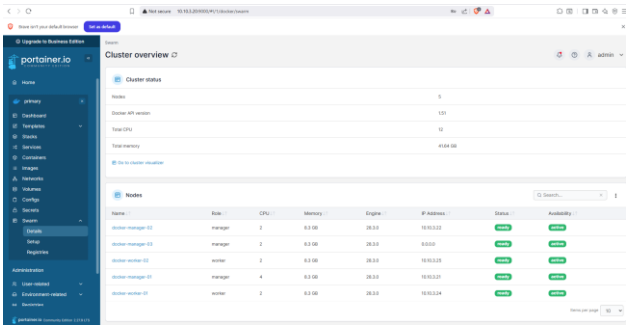
Setelah *High Availability* melalui *Keepalived* berhasil dikonfigurasi, langkah berikutnya adalah mendeploy *Portainer* sebagai antarmuka manajemen *Docker Swarm*. Pertama dibuat file *stack* yang mendefinisikan layanan *Portainer*, mencakup image *Portainer/Portainer*-ce:latest, port 9000 dan 8000, volume penyimpanan, *Network Overlay*, serta kebijakan *deployment* dalam *Swarm* mode, termasuk *restart policy*, *update* dan *rollback configuration*, dan batasan sumber daya. *Stack* ini kemudian didistribusikan ke seluruh *cluster manager* menggunakan perintah deploy, dan status layanan diverifikasi untuk memastikan semua *node* menjalankan *Portainer* secara aktif.

Gambar 7 menampilkan hasil verifikasi yang menunjukkan layanan *Portainer* berjalan dalam mode global dengan semua replika aktif di setiap *node manager*, menggunakan image *Portainer/Portainer*-ce:latest. Virtual IP (VIP) 10.10.3.20 memastikan akses *High Availability*, sehingga *Portainer* dapat diakses melalui *browser* di *http://10.10.3.20:9000* atau IP masing-masing *manager*. Setelah halaman login terbuka, akun admin *default* dapat dibuat, misalnya User: admin dan Password: P@ssw0rd123456, untuk mulai mengelola *container*, service, dan jaringan *cluster* secara terpusat.




**Gambar 7.** Testing Multi-*Container (Sumber : dokumen pribadi)*

Gambar 8 menampilkan antarmuka *Portainer* pada halaman *Swarm Cluster* Overview, yang menunjukkan status lima *node* (tiga *manager* dan dua worker) dalam keadaan *Ready* dan *Active*, lengkap dengan detail CPU, memori, versi *engine*, dan alamat IP di *cluster*.




**Gambar 8.** *Portainer (Sumber : dokumen pribadi)*

Setelah *Portainer* berjalan, *GlusterFS* diinstal di semua *node manager* untuk menyediakan penyimpanan terdistribusi bagi *cluster Docker Swarm*. Setiap *node* menjalankan layanan glusterd yang diaktifkan otomatis saat boot. *Node* tambahan diprobe dari *manager*-01 agar tergabung dalam *cluster*, kemudian direktori /gluster/brick1 dibuat pada setiap *node* sebagai brick atau penyimpanan dasar volume. Selanjutnya, volume *GlusterFS* bernama nfs-volume dibuat dengan replikasi tiga *node* (replica 3), memastikan data disalin ke semua *node manager*, lalu volume dimulai dan statusnya diverifikasi untuk memastikan volume aktif dan siap digunakan.

Gambar 9 menampilkan hasil verifikasi volume nfs-volume, menunjukkan tipe Replicate (replikasi 3-*node*) dengan status Started, menggunakan brick dari *Docker-manager*-01, 02, dan 03 pada direktori */gluster/brick1*.

```
root@docker-manager-01:~# sudo gluster volume info nfs-volume

Volume Name: nfs-volume
Type: Replicate
Volume ID: 41b1589c-a5e8-42eb-b953-c5baa9276944
Status: Started
Snapshot Count: 0
Number of Bricks: 1 x 3 = 3
Transport-type: tcp
Bricks:
Brick1: docker-manager-01:/gluster/brick1
Brick2: docker-manager-02:/gluster/brick1
Brick3: docker-manager-03:/gluster/brick1
Options Reconfigured:
performance.client-io-threads: off
transport.address-family: inet
storage.fips-mode-rchecksum: on
cluster.granular-entry-heal: on
```

**Gambar 9.** Mengecek Status Volume (Sumber : dokumen pribadi)

Konfigurasi selanjutnya meliputi pemasangan volume *GlusterFS* pada semua *node*, dimulai dengan menginstal klien *GlusterFS*, membuat direktori sebagai mount point, lalu melakukan mount volume. Gambar 10 menunjukkan bahwa direktori */mnt/nfs-share* yang menggunakan NFS dengan label nfs-volume ter-Mount dengan sukses.

```
root@docker-manager-01:~# df -h | grep nfs-volume
df: /var/lib/docker/volumes/grafana-server_grafana-config/_data: Stale file handle
df: /var/lib/docker/volumes/grafana-server_grafana-logs/_data: Stale file handle
localhost:/nfs-volume   34G  5.2G   27G  17% /mnt/nfs-share
```

**Gambar 10.** Pengecekan Mount (Sumber : dokumen pribadi)

Konfigurasi selanjutnya menambahkan entri pada /etc/fstab agar volume *GlusterFS* nfs-volume otomatis ter-mount ke /mnt/nfs-share saat boot pada setiap *node manager*, menggunakan tipe *GlusterFS* dengan opsi *default*,*_netdev*. Selanjutnya, diimplementasikan NFS *server* untuk *load balancing* dengan beberapa *server*; paket nfs-kernel-*server* diinstal dan konfigurasi asli */etc/exports* dicadangkan. *Direktori /mnt/nfs-share* diekspor dengan izin penuh ke semua client, opsi sinkronisasi langsung, tanpa pemeriksaan subtree, tanpa pembatasan root, dan diberi ID volume (fsid=1). Layanan NFS kemudian diekspor ulang, di*restart*, dan diaktifkan agar berjalan otomatis saat boot.

Untuk memastikan ketersediaan tinggi, *Keepalived* dikonfigurasi menambahkan skrip VRRP chk_nfs yang memeriksa status nfs-kernel-*server* setiap 5 detik, menurunkan prioritas *node* jika gagal, dan memantau bersama skrip chk_*Docker* pada bagian track_script, sehingga hanya *node* dengan layanan *Docker* dan NFS aktif yang dapat memegang Virtual IP (VIP) untuk *failover* yang andal.

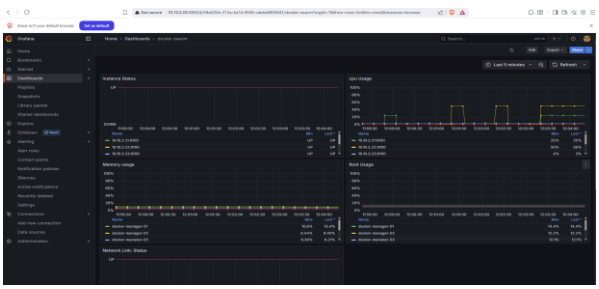
***Mengatur Layanan Monitoring***

Untuk memastikan sistem berjalan dengan baik serta memudahkan proses pemantauan performa infrastruktur *server* dan kontainer, dilakukan implementasi layanan *monitoring* menggunakan beberapa komponen, yaitu *cAdvisor, Node Exporter, VictoriaMetrics*, dan *Grafana*.

Metrik dari *cAdvisor* dan *Node* Exporter kemudian disimpan dan dikelola menggunakan VictoriaMetrics sebagai *time-series database*, dengan layanan *victoriametrics, vmagent*, dan *vmalert* dijalankan di *Docker Swarm*, menggunakan penyimpanan berbasis NFS dan jaringan *Overlay* "*monitoring*". File konfigurasi *Prometheus* juga ditempatkan di NFS agar dapat melakukan scraping metrik dari semua *node* dan kontainer.

Untuk visualisasi, *Grafana* digunakan sebagai dashboard interaktif. Layanan *Grafana* dijalankan di *node manager* dengan satu replika, menggunakan Traefik sebagai reverse proxy, dan menyimpan konfigurasi, data, serta log di NFS. Dashboard *Grafana* menampilkan status *cluster*, performa CPU, penggunaan memori, *storage*, dan jaringan dari semua *node Docker Swarm*, sehingga memudahkan pemantauan dan menjaga stabilitas sistem.

Gambar 11 yang ditampilkan merupakan tampilan dashboard *monitoring* dari *Grafana* yang digunakan untuk memantau performa dan status dari *cluster Docker Swarm*.



**Gambar 11.** *Grafana (Sumber : dokumen pribadi)*

## 3) Pengujian

### a. Pengujian Fungsional

Pengujian fungsional pada tabel 9 membuktikan seluruh komponen sistem bekerja sesuai harapan. *Container* dapat dibuat dan dijalankan dengan sukses, akses *web*, koneksi *database* berfungsi normal, dan dashboard *monitoring* berhasil menampilkan seluruh metrik yang dibutuhkan.

**Tabel 9.** Pengujian Fungsional

| No. | Kebutuhan Fungsional | Hasil |
|---|---|---|
| 1 | Pengguna dapat melakukan akses remote ke *server* menggunakan login SSH | Berhasil |
| 2 | Pengguna dapat menambahkan image baru dari *Portainer* ke *Docker* Hub | Berhasil |
| 3 | Pengguna dapat membuat *container* baru dari image yang telah ditambahkan | Berhasil |
| 4 | Pengguna dapat menghentikan *container* yang sedang berjalan pada *server* | Berhasil |
| 5 | Pengguna dapat menjalankan atau menghentikan *container* melalui fitur yang ada pada dashboard *Portainer* | Berhasil |
| 6 | Pengguna dapat memonitor penggunaan sumber daya (CPU, RAM, dan Disk) melalui dashboard *Portainer* atau *Grafana* | Berhasil |
| 7 | Pengguna dapat melakukan scraping metric environment melalui *Prometheus* | Berhasil |

| No. | Kebutuhan Fungsional | Hasil |
|---|---|---|
| 8 | Pengguna dapat menambahkan *node Docker manager* atau *node Docker* worker ke dalam *Docker Swarm* | Berhasil |
| 9 | Penggunaan CPU dan Memory sebelum dan sesudah *Docker* | Berhasil |

### b. Pengujian CPU dan Memori

Hasil pengujian pada tabel 10 menunjukkan penurunan signifikan dalam penggunaan sumber daya. Utilisasi CPU rata-rata turun dari 75% menjadi 52%, sementara penggunaan memori berkurang sekitar 30%. Pada simulasi 200 User, penggunaan memori berkurang dari 2048MB menjadi 1536MB.

**Tabel 10.** Pengujian Penggunaan CPU

| User Request | Sebelum *Docker* (Total Host) | Sesudah *Docker* (Per *Node*) |
|---|---|---|
| 200 | 15% | 10% (*Manager*), 8% (Worker) |
| 400 | 30% | 20% (*Manager*), 15% (Worker) |
| 600 | 45% | 30% (*Manager*), 22% (Worker) |
| 800 | 60% | 40% (*Manager*), 30% (Worker) |
| 1000 | 75% | 50% (*Manager*), 35% (Worker) |

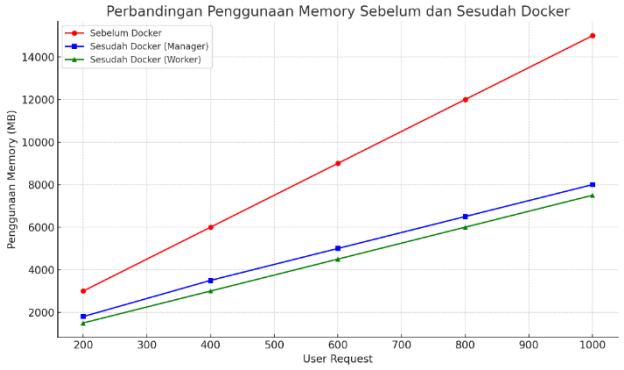
Grafik pengujian CPU pada gambar 10 menunjukkan bahwa setelah implementasi *Docker*, beban kerja terdistribusi lebih merata antar *node*. *Node manager* menangani beban sekitar 10-50% seiring peningkatan request, sementara *node* worker bekerja pada kapasitas 8-35%. Namun terlihat adanya overhead orchestration sebesar 5-10% pada *node manager* yang disebabkan oleh proses manajemen *cluster Swarm*. Pada beban 1000 request, total konsumsi CPU seluruh *node* melebihi kapasitas host asli, menunjukkan bahwa arsitektur ini memerlukan optimasi lebih lanjut.


**Gambar 12.** Grafik Hasil Pengujian CPU (Sumber : dokumen pribadi)

Penggunaan memory seperti yang ditunjukkan pada tabel 11 dan gambar 13 menunjukkan pola yang konsisten dimana *node manager* mengkonsumsi memory sekitar 300-500MB lebih banyak dibanding worker akibat layanan kontrol seperti *Swarm*, *Portainer*, dan *Prometheus*. Pada beban puncak 1000 request, memory *node manager* mencapai 8000MB (80% dari kapasitas 8GB), mendekati batas yang berpotensi menyebabkan OOM (Out-of-Memory).

**Tabel 11.** Pengujian Penggunaan Memory

| **User Request** | **Sebelum *Docker* (Total Host)** | **Sesudah *Docker* (Per *Node*)** |
|---|---|---|
| 200 | 3000 MB | 1800 MB (*Manager*), 1500 MB (Worker) |
| 400 | 6000 MB | 3500 MB (*Manager*), 3000 MB (Worker) |
| 600 | 9000 MB | 5000 MB (*Manager*), 4500 MB (Worker) |
| 800 | 12000 MB | 6500 MB (*Manager*), 6000 MB (Worker) |
| 1000 | 15000 MB | 8000 MB (*Manager*), 7500 MB (Worker) |




**Gambar 13.** Grafik Hasil Pengujian Memory (Sumber : dokumen pribadi)

### c. Validasi Ahli IT

Pengujian sistem virtualisasi *Docker* untuk aplikasi *web Multiserver* dilakukan oleh lima ahli IT, meliputi konfigurasi *Swarm*, *container*, *Load Balancer*, efisiensi sumber daya, *monitoring*, keamanan, dan kelancaran *deployment*.

**Tabel 12.** Hasil Perhitungan Validasi Ahli IT

| **Skor Capaian** | **Hasil Skor** | **Total (Skala * Hasil Skor)** |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 0 | 0 |
| 3 | 2 | 6 |
| 4 | 15 | 60 |
| 5 | 10 | 50 |
| Total Skor | | 116 |
| Skor Maksimum | | 135 |
| | | (Jumlah Responden * Jumlah Butir Soal * Skala Tertinggi) |

Hasil dari perhitungan tabulasi ahli IT pada tabel diatas dimasukan kedalam rumus kelayakan. Berikut perhitungan rumus kelayakan *Docker* Pada Virtualisasi *Server* Untuk Mengoptimalkan Kinerja Aplikasi Berbasis *Web* Menggunakan *Multiserver*.

$$\text{Presentase kelayakan} = \frac{116}{135} x100\%$$

$$= 85.9\ \%$$

Berdasarkan hasil dari analisis data yang telah dilakukan dari hasil kuisioner ahli IT, Sehingga diperoleh hasil perhitungan secara keseluruhan adalah 85.9%.

**d. Pengujian Pengguna**

Setelah pengujian fungsional dan validasi ahli IT, dilakukan *User Acceptance Testing* (UAT) terhadap tiga teknisi laboratorium SMKN 1 Nglegok untuk menilai kemudahan, kepuasan, dan efektivitas sistem dari perspektif pengguna akhir.

**Tabel 13.** Hasil Perhitungan *User Acceptance Testing*

| Skor Capaian | Hasil Skor | Total (Skala * Hasil Skor) |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 0 | 0 |
| 3 | 0 | 0 |
| 4 | 16 | 64 |
| 5 | 14 | 70 |
| Total Skor | | 134 |
| Skor Maksimum | | 150 |
| | | (Jumlah Responden * Jumlah Butir Soal * Skala Tertinggi) |

Hasil dari perhitungan tabulasi pengguna pada tabel di atas dimasukkan ke dalam rumus kelayakan. Berikut ini adalah perhitungan rumus kelayakan sistem *Docker* pada virtualisasi *server* untuk mengoptimalkan kinerja aplikasi berbasis *web* menggunakan *Multiserver*:

$$\text{Presentase kelayakan} = \frac{134}{150} x 100\% = 89.3\ \%$$

Berdasarkan hasil dari analisis data yang telah dilakukan dari hasil kuisioner pengguna, Sehingga diperoleh hasil perhitungan secara keseluruhan adalah 89.3%, yang menunjukkan bahwa sistem tergolong dalam kategori "sangat layak" untuk diimplementasikan.

## KESIMPULAN DAN SARAN

**1. Kesimpulan**

Berdasarkan penelitian mengenai pemanfaatan *Docker* pada virtualisasi *server* untuk mengoptimalkan kinerja aplikasi berbasis *web* dengan pendekatan *Multiserver* di SMKN 1 Nglegok, dapat disimpulkan bahwa penggunaan *Docker* terbukti mampu meningkatkan efisiensi dan stabilitas kinerja aplikasi melalui isolasi *container*, kemampuan scaling horizontal, serta pengelolaan layanan yang stabil menggunakan *load balancing* berbasis *Docker Swarm* dan *Nginx*. Hasil pengujian menunjukkan penggunaan CPU dan memori menjadi lebih terkontrol dan efisien, bahkan saat terjadi lonjakan trafik dari 200 hingga 1000 request, dengan waktu respons yang tetap stabil sebagaimana diverifikasi melalui pengujian K6 dan *monitoring* menggunakan *Grafana*. Selain itu, integrasi sistem penyimpanan terdistribusi (*GlusterFS*), *monitoring* (*Prometheus* dan *Grafana*), serta *load balancing* mendukung ketersediaan tinggi (*High Availability*) pada layanan *web*. Validasi dari ahli IT serta hasil *User Acceptance Testing* (UAT) menunjukkan bahwa sistem ini efektif, mudah digunakan, dan layak diterapkan untuk mendukung pengelolaan dan manajemen aplikasi *web* di lingkungan pendidikan.

**2. Saran**

Beberapa rekomendasi yang dapat diberikan meliputi:

a. Integrasi sistem backup dan pemulihan data untuk menjaga kontinuitas layanan.
b. Peningkatan keamanan melalui firewall, enkripsi TLS, dan kontrol akses berbasis peran.
c. Penerapan orkestrasi skala lanjut menggunakan Kubernetes untuk manajemen layanan dan sumber daya yang lebih efisien.

## DAFTAR PUSTAKA


[1] A. I. Ramdhani Et Al., "*Docker* File Untuk Publish *Storage* Digital Dengan Next*cloud* Pada Pt . Telkom Akses," Vol. 6, No. 2, Hal. 175–182, 2025.

[2] I. Wahyudi Dan D. Asri, "Penerapan *Docker Container* Untuk Mendukung Multi Domain Dan Keamanan *Web*site Pemerintah Provinsi Papua," Pros. Semin. Nas. Sains Dan Teknol. Seri Iii Fak. Sains Dan Teknol., Vol. 2, No. 1, Hal. 910–923, 2025.

[3] F. R. Bik Dan Asmunin, "Implementasi *Docker* Untuk Pengelolaan Banyak Aplikasi *Web* (Studi Kasus : Jurusan Teknik Informatika Unesa)," 2017.

[4] R. Khalida, A. Muhajirin, Dan S. Setiawati, "Teknis Kerja *Docker Container* Untuk Optimalisasi Penyebaran Aplikasi," 2019. Doi: 10.33558/Piksel.V7i2.1819.

[5] M. K. Imammuddin, Januar Al-Amien, "Membangun *Cloud* Menggunakan *Docker* Pada Implementasi *Load balancing* Dan Pengujian Algoritma *Round Robin* Pada *Web Server*," 2019.

[6] A. R. Ekaputra Dan A. S. Affandi, "Pemanfaatan Layanan *Cloud computing* Dan *Docker Container* Untuk Meningkatkan Kinerja Aplikasi *Web*," J. Inf. Syst. Appl. Dev., Vol. 1, No. 2, Hal. 138–147, Sep 2023, Doi: 10.26905/Jisad.V1i2.11084.

[7] N. Ramsari Dan A. Ginanjar, "Implementasi Infrastruktur *Server* Berbasis *Cloud computing* Untuk *Web* Service Berbasis Teknologi *Google Cloud Platform*," Conf. Senat. Stt Adisutjipto Yogyakarta, Vol. 7, Mar 2022, Doi: 10.28989/Senatik.V7i0.472.

[8] M. Hud, R. Sahara, Dan S. Abdullah, "Implementasi *Load balancing* Sistem Mail *Server* Serta Peningkatan *Cbpolicyd* Menggunakan *Zimbra*," 2024.

[9] B. Đorđević, D. Gojak, N. Davidović, Dan ..., "*File system* Performance Comparison Of Native Operating System And *Docker Container*-Based Virtualization," 8th Icetran 2021, Ethno …, 2021.

[10] S. Dwiyatno, E. Rachmat, A. P. Sari, Dan O. Gustiawan, "Implementasi Virtualisasi *Server* Berbasis *Docker Container*," Prosisko J.

Pengemb. Ris. Dan Obs. Sist. Komput., Vol. 7, No. 2, Hal. 165–175, 2020, Doi: 10.30656/Prosisko.V7i2.2520.

[11] M. A. Nugroho Dan R. Katardi, "Analisis Kinerja Penerapan *Container* Untuk *Load balancing Web Server* Pada Raspberry Pi," 2016.

[12] A. Alimuddin Dan A. Ashari, "Peningkatan Kinerja Siakad Menggunakan Metode *Load balancing* Dan Fault Tolerance Di Jaringan Kampus Universitas Halu Oleo," Ijccs (Indonesian J. Comput. Cybern. Syst., Vol. 10, No. 1, Hal. 11, 2016, Doi: 10.22146/Ijccs.11185.

[13] S. Safriadi Dan R. Rahmadani, "Analisis Kinerja *Load balancing Round Robin* Pada *Web*site Skalabel," J. Inf. Syst. Manag., Vol. 5, No. 2, Hal. 227–232, 2024, Doi: 10.24076/Joism.2024v5i2.1441.

[14] Q. Wang Et Al., "*Container* Resource Allocation Versus Performance Of Data-Intensive Applications On Different *Cloud Servers*," Nov 2023.

[15] M. Wicaksono Dan J. Pamungkas, "Membuat *Web Server* Menggunakan Debian 10 Pada Virtual Machine," Room Civ. Soc. Dev., Vol. 4, No. 1, 2022, Doi: 10.59110/Rcsd.V2i1.156.