Jurnal Teknik Informatika (JUTIF)
Vol. 6, No. 1, February 2025, pp. 453-468
DOI: https://doi.org/10.52436/1.jutif.2025.6.1.4040
p-ISSN: 2723-3863
e-ISSN: 2723-3871

COMPARATIVE ANALYSIS OF PERFORMANCE AND EFFICIENCY OF LOAD BALANCING ALGORITHMS ON INGRESS CONTROLLER

Ahmad Rizal Khamdani[*1], Ahmad Rofiqul Muslikh[2], Arif Saivul Affandi[3]

[1,2,3]Information System, Faculty of Information Technology, University of Merdeka Malang, Indonesia
Email: [1]ahmadkhamdani9@gmail.com, [2]rofickachmad@unmer.ac.id, [3]fandi@unmer.ac.id

(Article received: October 21, 2024; Revision: November 7, 2024; published: March 6, 2025)

Abstract

Kubernetes has become the dominant container orchestration platform in production environments, with the ingress controller playing a critical role in managing external traffic to services within the cluster. This study aims to provide recommendations for optimal load balancing algorithms for Kubernetes production environments by analyzing and comparing the performance of four algorithms namely round robin, static-rr, least connection, and random on the HAProxy ingress controller. The research method is conducted through observation using k6 and Grafana performance test tools, as well as literature studies, with measurements including total requests, throughput, latency, CPU usage, and memory at various levels of user load. The data was analyzed using descriptive statistical techniques, normality test, homogeneity test, and tests for group differences using one-way ANOVA or Kruskal-Wallis H. The results show that static-rr excels in throughput, total requests, and CPU and memory efficiency at high load, while least connection is more effective for latency at low load. Round robin and random showed stable performance at low load but less optimal at high load. The conclusion of this study is that choosing the right load balancing algorithm depends on the load characteristics and desired performance metrics, to ensure optimal Kubernetes performance under various load scenarios in production environments.

Keywords: *Container Orchestration, Ingress Controller, Kubernetes, Load Balancing, Load Testing.*

ANALISIS PERBANDINGAN PERFORMA DAN EFISIENSI ALGORITMA LOAD BALANCING PADA INGRESS CONTROLLER

Abstrak

Kubernetes telah menjadi platform orkestrasi kontainer yang dominan dalam lingkungan produksi, dengan *ingress controller* memainkan peran penting dalam mengelola lalu lintas eksternal ke layanan di dalam *cluster*. Penelitian ini bertujuan untuk memberikan rekomendasi algoritma *load balancing* yang optimal untuk lingkungan produksi Kubernetes dengan menganalisis dan membandingkan kinerja empat algoritma yaitu *round robin*, *static-rr*, *least connection*, dan *random* pada HAProxy *ingress controller*. Metode penelitian dilakukan melalui observasi menggunakan alat uji kinerja k6 dan Grafana, serta studi literatur, dengan pengukuran meliputi *total request*, *throughput*, *latency*, penggunaan CPU, dan memori pada berbagai tingkat beban pengguna. Data dianalisis menggunakan teknik statistik deskriptif, uji normalitas, uji homogenitas, dan uji beda menggunakan *ANOVA one-way* atau *Kruskal-Wallis H*. Hasil menunjukkan bahwa *static-rr* unggul dalam *throughput*, total *request*, serta efisiensi CPU dan memori pada beban tinggi, sedangkan *least connection* lebih efektif untuk *latency* pada beban rendah. *Round robin* dan *random* menunjukkan performa yang stabil pada beban rendah tetapi kurang optimal pada beban tinggi. Kesimpulan dari penelitian ini adalah pemilihan algoritma *load balancing* yang tepat bergantung pada karakteristik beban dan metrik kinerja yang diinginkan, untuk memastikan performa Kubernetes yang optimal dalam berbagai skenario beban di lingkungan produksi.

Kata kunci: *Ingress Controller, Kubernetes, Load Balancing, Load Testing, Orkestrasi Kontainer.*

1. PENDAHULUAN

Kubernetes telah menjadi salah satu platform orkestrasi *container* yang dominan dalam *cloud* dan pengembangan perangkat lunak modern. Laporan CNCF Annual Survey 2023 menyebutkan bahwa Kubernetes telah melampaui "chasm" adopsi menjadi teknologi *mainstream* global, dengan 66% organisasi pengguna *cloud* menggunakannya dalam produksi dan hanya 15% yang tidak berencana untuk menggunakannya [1]. Dengan adopsi yang terus meningkat, Kubernetes terbukti sebagai platform

kunci dalam pengelolaan *container* dan orkestrasi. Perkembangan ini sejalan dengan peran penting *cloud computing* sebagai fondasi utama dalam pengembangan dan penyajian aplikasi web yang *scalable* dan andal [2]. Kombinasi teknologi *cloud* dan Kubernetes telah menciptakan ekosistem yang kuat untuk mendukung inovasi dan efisiensi dalam pengembangan perangkat lunak modern.

Dalam ekosistem Kubernetes, terdapat komponen yang disebut sebagai *kube-proxy* yang berfungsi untuk menjaga koneksi antar *pod* serta mendistribusikan lalu lintas jaringan ke berbagai *pod* atau server *backend* di dalam *cluster* [3]. Proses pendistribusian lalu lintas yang dilakukan oleh *kube-proxy* ini dikenal sebagai *load balancing*, yang sangat penting dalam Kubernetes karena memungkinkan aplikasi untuk melakukan *scaling* secara dinamis sesuai permintaan, meningkatkan ketersediaan, serta mengoptimalkan pemanfaatan kapasitas server [4]. Melalui *load balancing* ini juga aplikasi dapat beroperasi dengan lebih efisien dan tetap responsif, terutama dalam menghadapi lonjakan lalu lintas atau beban tinggi yang dapat terjadi sewaktu-waktu [5]. Tanpa adanya mekanisme *load balancing* pada Kubernetes, aplikasi berisiko mengalami kelebihan beban pada satu *pod*, yang dapat mengakibatkan penurunan kinerja, waktu respons yang lambat, bahkan *downtime*, sehingga berpotensi merugikan pengalaman pengguna dan stabilitas layanan secara keseluruhan.

Selain *kube-proxy*, Kubernetes juga memiliki komponen lainnya yang memiliki fungsi *load balancing* yang sama, yaitu ingress dan ingress controller. Ingress berfungsi sebagai kumpulan aturan yang menentukan bagaimana lalu lintas eksternal diarahkan ke berbagai layanan internal, sementara ingress controller menerjemahkan dan menerapkan aturan-aturan ini, memastikan bahwa lalu lintas dikirim ke tujuan yang tepat sesuai dengan konfigurasi yang diinginkan [6]. Proses ini mirip dengan konsep *routing*, di mana item disampaikan melalui rute jaringan yang telah dikonfigurasi untuk mencapai titik tujuan dari suatu lokasi [7]. Perbedaan antara *kube-proxy* dan ingress controller terletak pada cakupan fungsinya. *Kube-proxy* mengelola lalu lintas internal dalam *cluster* dan melakukan *load balancing* antar *pod* di dalam cluster. Sementara itu, ingress controller menangani lalu lintas eksternal dan mengarahkan permintaan ke layanan yang sesuai berdasarkan aturan dalam konfigurasi ingress. Selain itu, ingress controller juga dapat melakukan *load balancing* seperti yang dilakukan *kube-proxy* dengan menyeimbangkan permintaan berdasarkan *endpoint* yang diambil dari Kubernetes API server.

Dalam konteks ingress controller, algoritma yang digunakan dalam *load balancing* berperan penting dalam menentukan bagaimana permintaan klien didistribusikan ke *pod* atau server *backend* di dalam *cluster*. Algoritma-algoritma ini berfungsi untuk memastikan bahwa distribusi lalu lintas terjadi secara efisien dan sesuai dengan kebutuhan sistem. Secara *default*, beberapa ingress controller, termasuk HAProxy, menggunakan algoritma *round robin* untuk mendistribusikan permintaan secara merata ke seluruh server, tanpa mempertimbangkan kapasitas server atau beban yang ada [8]. Di samping itu, terdapat juga algoritma lain seperti *least connection*, yang mengarahkan distribusi beban lalu lintas ke server dengan jumlah koneksi aktif paling sedikit [9]. Selain kedua algoritma tersebut, masih ada banyak algoritma lain yang dapat diterapkan dalam konteks *load balancing* pada ingress controller di Kubernetes, yang dapat memberikan fleksibilitas tambahan dalam mengelola lalu lintas dan menyesuaikan distribusi beban berdasarkan karakteristik serta kebutuhan sistem.

Meskipun banyak pilihan algoritma *load balancing* yang tersedia dalam ingress controller, tidak semua algoritma cocok untuk setiap jenis aplikasi atau beban kerja, dan dukungan terhadap algoritma ini bergantung pada jenis ingress controller yang digunakan. Salah satu tantangan utama adalah memastikan bahwa algoritma *load balancing* yang dipilih mampu mengoptimalkan kinerja sekaligus meminimalkan konsumsi sumber daya. Mengacu pada Usman [10] yang menyebutkan tujuh kriteria untuk algoritma penjadwalan yang efisien, dalam hal ini difokuskan pada dua kriteria yaitu minimum *makespan* dan konsumsi energi. Dengan mempertimbangkan kedua aspek tersebut, pemilihan algoritma yang tepat menjadi krusial untuk mengoptimalkan distribusi beban kerja pada sistem. Sebagai contoh, algoritma dinamis seperti *least connection* mungkin dapat mendistribusikan lalu lintas dengan lebih baik dalam kondisi beban yang berfluktuasi, sebagaimana ditemukan oleh Solehudin dkk. [11], yang menunjukkan bahwa *least connection* memiliki waktu respons rata-rata yang lebih cepat, yaitu 6,9 ms dibandingkan *round robin* yang mencapai 7,2 ms. Namun, *round robin* mungkin lebih cocok untuk beban kerja yang lebih sederhana dan terdistribusi secara merata, terutama karena algoritma ini menunjukkan penggunaan CPU yang lebih rendah, yaitu 23,7% dibandingkan *least connection* yang mencapai 24,3%. Oleh karena itu, memilih algoritma yang sesuai dengan kebutuhan skalabilitas dan karakteristik lalu lintas sangat penting untuk menghindari pembebanan berlebih pada infrastruktur, sehingga perlu mengevaluasi kesiapan teknologi sejak awal agar algoritma yang dipilih mampu memenuhi kedua aspek tersebut, terutama dalam lingkungan *microservices* yang permintaannya tinggi seperti di Kubernetes.

Penelitian sebelumnya yang dilakukan oleh Luthfi dkk. [12] menyoroti perancangan dan implementasi *load balancing* menggunakan algoritma *least connection* dan *IP hash* pada perangkat *load balancer* milik Oracle Cloud dalam lingkungan Kubernetes. Temuan mereka

menunjukkan bahwa algoritma *least connection* mampu memberikan performa yang baik dalam mendistribusikan paket permintaan dari *client*, berdasarkan pengukuran parameter *response time*, *throughput*, dan *request loss*. Penelitian oleh Rawls dan Salehi [13] juga membahas berbagai metode *load balancing* menggunakan HAProxy dan menemukan bahwa dalam lingkungan server homogen, algoritma *load balancing* yang memberikan waktu respons terendah secara berurutan adalah *random*, *least connection*, *round robin*, dan *static-rr*. Hal ini menunjukkan bahwa keempat algoritma ini memiliki performa yang serupa dalam hal kecepatan respons, meskipun terdapat sedikit variasi di antara mereka. Selain itu, penelitian yang dilakukan oleh Ibrahim dkk. [14] menunjukkan bahwa teknik *load balancing* dapat meningkatkan performa server web. Dalam studi ini, algoritma *pending job* dan *IP hash* terbukti mengungguli algoritma lain, termasuk *round robin*. Algoritma *pending job* berhasil mengurangi waktu pemrosesan hingga 18,5% dibandingkan *round robin*, sementara *IP hash* menunjukkan *latency* yang lebih rendah dan *throughput* yang lebih baik dibandingkan dengan *least connection*. Meskipun penelitian-penelitian sebelumnya memberikan kontribusi dan wawasan yang berharga terkait metode *load balancing*, sebagian besar dari penelitian tersebut berfokus pada aspek *load balancing* di luar konteks atau lingkungan Kubernetes. Walaupun terdapat penelitian yang membahas topik *load balancing* dalam konteks Kubernetes, fokusnya lebih terletak pada penggunaan perangkat *external load balancer* yang ditawarkan oleh penyedia layanan cloud. Hal ini berbeda dengan pendekatan yang diperlukan untuk memahami dinamika *load balancing* di dalam *cluster* Kubernetes itu sendiri, khususnya yang melibatkan penggunaan ingress controller.

Berdasarkan temuan-temuan dari penelitian terdahulu, penting untuk melakukan penelitian lebih lanjut yang memfokuskan pada evaluasi kinerja dan efisiensi berbagai algoritma *load balancing* dalam konteks ingress controller di Kubernetes, terutama dalam skenario penggunaan yang lebih kompleks dan bervariasi. Oleh karena itu, penelitian ini bertujuan untuk menganalisis dan membandingkan kinerja serta efisiensi dari empat algoritma *load balancing*, yaitu *round robin*, *static-rr*, *least connection*, dan *random* pada Kubernetes menggunakan HAProxy ingress controller. HAProxy dipilih sebagai ingress controller dalam penelitian ini karena dukungan untuk banyak algoritma *load balancing*, sedangkan pemilihan keempat algoritma didasarkan pada hasil penelitian sebelumnya yang menunjukkan bahwa keempat algoritma ini merupakan yang terbaik dalam hal performa. Analisis dalam penelitian ini meliputi pengujian kinerja berdasarkan jumlah *request*, *throughput*, dan *latency*, serta uji efisiensi yang mencakup penggunaan CPU dan memori. Pengujian kinerja penting untuk memberikan wawasan tentang operasi aplikasi di berbagai kondisi dan beban [15], sementara uji efisiensi mengevaluasi seberapa baik algoritma *load balancing* memanfaatkan sumber daya sistem guna memastikan performa optimal dalam lingkungan produksi. Teknik analisis data yang akan digunakan dalam penelitian ini meliputi statistik deskriptif untuk memberikan gambaran umum tentang data, uji normalitas dan uji homogenitas untuk memastikan asumsi analisis yang tepat, serta uji beda menggunakan *ANOVA one-way* atau *Kruskal-Wallis H* untuk menilai perbedaan signifikan antar algoritma *load balancing* dalam hal kinerja dan efisiensi.

Penelitian ini diharapkan dapat memberikan panduan praktis bagi pengembang dan administrator sistem dalam memilih algoritma *load balancing* yang paling sesuai untuk kebutuhan mereka pada HAProxy di Kubernetes. Dengan demikian, hasil penelitian ini tidak hanya meningkatkan pemahaman tentang kinerja algoritma *round robin*, *static-rr*, *least connection*, dan *random*, tetapi juga memberikan dasar yang kuat untuk pengambilan keputusan dalam implementasi algoritma tersebut di lingkungan produksi. Evaluasi komparatif seperti ini penting untuk meminimalkan risiko implementasi teknologi yang tidak efisien.

2. METODE PENELITIAN

2.1. Desain Penelitian

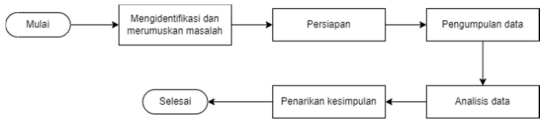
Penelitian ini menggunakan desain quasi eksperimental, yang merupakan metode eksperimen tanpa randomisasi penuh, namun masih melibatkan kontrol terhadap variabel non-eksperimental serta penggunaan kelompok kontrol sebagai kelompok komparatif untuk memahami efek perlakuan [16]. Penelitian ini juga termasuk dalam jenis penelitian kuantitatif dengan pendekatan komparatif, yang bertujuan untuk menganalisis dan membandingkan kinerja dan efisiensi algoritma *round robin*, *static-rr*, *least connection*, dan *random* pada HAProxy ingress controller berbagai metrik seperti *total request*, *throughput*, *latency*, CPU *usage*, dan memory *usage*. Desain penelitian ini memungkinkan peneliti untuk melakukan eksperimen meski tanpa randomisasi penuh, namun tetap dapat mengukur dampak perlakuan dengan akurat.

Dalam penelitian ini, tidak terdapat kelompok kontrol dan hanya ada kelompok eksperimen yang terdiri dari berbagai jenis algoritma *load balancing*, yaitu *round robin*, *static-rr*, *least connection*, dan *random*, yang diterapkan pada HAProxy ingress controller di Kubernetes. Masing-masing dari kelompok tersebut diberikan perlakuan yang sama dan diujikan dalam kondisi yang berbeda-beda. Karena semua kelompok eksperimen mendapatkan perlakuan yang serupa dan tidak ada kelompok kontrol, maka desain eksperimen yang diterapkan

adalah *counter balance* [17]. Desain ini diadopsi untuk mengatasi potensi bias urutan dan memastikan bahwa setiap algoritma *load balancing* diuji dalam berbagai kondisi secara sistematis, sehingga hasilnya lebih valid dan dapat diandalkan. Penggunaan *counter balance* juga membantu menghilangkan efek urutan yang mungkin mempengaruhi kinerja hasil pengujian.

2.2. Alur Penelitian

Penelitian ini mengikuti alur penelitian yang sistematis, dimulai dari identifikasi masalah hingga penarikan kesimpulan akhir. Setiap tahapan disusun untuk memberikan gambaran yang jelas mengenai proses eksperimen dalam menganalisis dan membandingkan kinerja dan efisiensi algoritma *load balancing* dalam konteks HAProxy ingress controller pada Kubernetes. Berikut adalah gambaran alur penelitian yang digunakan dalam penelitian ini.



Gambar 1. Alur Penelitian

Gambar 1 menunjukkan alur tahapan penelitian yang digunakan dalam studi ini dimulai dengan mengidentifikasi dan merumuskan masalah, yaitu langkah awal yang penting untuk menentukan fokus dan tujuan penelitian. Selanjutnya, tahap persiapan dilakukan untuk menyusun rencana eksperimen dan menyiapkan alat serta sumber daya yang diperlukan. Setelah persiapan, proses pengumpulan data dilaksanakan, di mana data diperoleh melalui pengujian serta observasi untuk variabel yang relevan. Setelah data terkumpul, tahap analisis data dilakukan dengan menerapkan teknik statistik untuk mengevaluasi hasil eksperimen secara menyeluruh. Akhirnya, penarikan kesimpulan dilakukan berdasarkan analisis data untuk memberikan wawasan dan rekomendasi terkait kinerja dan efisiensi dari algoritma *load balancing* yang diuji. Setiap tahap dalam alur penelitian ini dirancang untuk memastikan bahwa proses eksperimen berjalan secara sistematis dan menghasilkan temuan yang valid dan dapat diandalkan

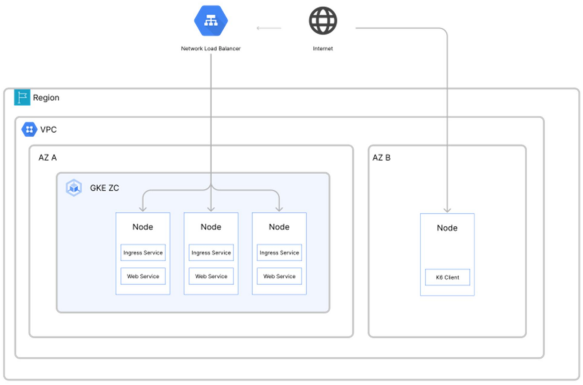
2.2.1. Identifikasi Masalah

Pada tahap ini, penelitian dimulai dengan merumuskan masalah utama mengenai perbandingan kinerja dan efisiensi empat algoritma *load balancing*, yaitu *round robin*, *static-rr*, *least connection*, dan *random*, pada berbagai tingkat beban pengguna dengan 1000, 2000, dan 3000 pengguna. Variabel bebas dalam penelitian ini adalah jenis algoritma *load balancing* dan skenario beban pengguna, yang diuji untuk melihat perbedaan performa dan efisiensi di bawah kondisi yang bervariasi.

Variabel terikat yang diukur mencakup *total request* sebagai jumlah permintaan yang berhasil diproses, *throughput* sebagai rata-rata permintaan per detik sebagai indikator performa, *latency* sebagai rata-rata waktu respons, serta penggunaan CPU dan memori yang menunjukkan efisiensi sumber daya. Penelitian ini bertujuan memberikan wawasan mengenai kinerja setiap algoritma dalam mengelola *traffic*, serta menentukan algoritma yang paling tepat digunakan dalam konteks tertentu.

2.2.2. Persiapan

Pada tahap persiapan, desain arsitektur *cloud* disusun untuk memastikan lingkungan yang optimal bagi pengujian. Desain arsitektur *cloud* penting untuk menyediakan infrastruktur yang efisien dan skalabel, yang mencakup penentuan jumlah dan jenis *node* serta pengaturan *Availability Zone* (AZ) untuk mendukung distribusi beban yang efektif. Selain desain arsitektur cloud, konfigurasi Kubernetes *cluster* juga dilakukan di *region* Jakarta melalui *Google Kubernetes Engine* (GKE), yang merupakan sistem manajemen *cluster* dan orkestrasi yang andal untuk mengoperasikan *container* Docker, yang didasarkan pada platform *open source* Kubernetes [18]. Dalam pengaturan ini, terdapat tiga *worker node* di AZ A dan satu *node* di AZ B sebagai *client* untuk k6. Ingress controller juga diinstal dan dikonfigurasi pada *cluster*, serta dilakukan pemantauan performa sistem untuk mengevaluasi kinerja *cluster* secara keseluruhan. Tahap ini juga memastikan bahwa semua komponen telah dikonfigurasi dengan benar dan siap untuk tahap pengujian. Berikut adalah gambar yang menunjukkan desain arsitektur dan konfigurasi yang telah diterapkan.



Gambar 2. Arsitektur Cloud

Gambar 2 menunjukkan gambaran arsitektur *cloud* yang peneliti gunakan, dimana pengujian akan dilakukan di GKE *Zonal Cluster*. Contoh dari aplikasi web akan di-*deploy* dalam *container* yang berjalan di *node-node* Kubernetes di *Availability Zone* (AZ) A. Permintaan dari internet akan masuk melalui *Network Load Balancer* (NLB) dan diarahkan ke Ingress Service. Ingress Service kemudian meneruskan permintaan ke layanan web yang sesuai di dalam *cluster*. Sementara itu, di AZ B terdapat sebuah node yang menjalankan k6 *client*. *Node* k6 *client* ini digunakan untuk mengirimkan *load testing* ke dalam Kubernetes *cluster* yang ada di AZ A.

Dalam tahap persiapan, identifikasi kebutuhan perangkat juga dilakukan untuk memastikan kesiapan infrastruktur dan alat yang digunakan dalam penelitian. Tabel di bawah ini merinci spesifikasi perangkat yang digunakan.

Tabel 1. Spesifikasi Kebutuhan

Perangkat	Spesifikasi	Keterangan
Local Client	OS	Windows 11
	CPU	6
	RAM	16 GB
	Tool dan Aplikasi	IBM SPSS Statistics, Microsoft Excel
K6 Client	OS	Ubuntu 20.04 Focal
	vCPU	2
	RAM	4 GB
	Tool dan Aplikasi	Speed Test, K6
Worker Node	OS	cos_containerd
	vCPU	2
	RAM	4 GB
	Tool dan Aplikasi	Ingress Controllers, Nginx Container (10 Pod)

Tabel 1 menampilkan detail tentang spesifikasi perangkat yang digunakan untuk pengujian, termasuk sistem operasi, kapasitas CPU dan RAM, serta alat dan aplikasi yang diperlukan untuk analisis data dan pelaksanaan *load testing*.

2.2.3. Pengumpulan Data

Pada tahap ini, pengumpulan data dilakukan menggunakan dua teknik utama yaitu observasi dan studi literatur. Metode observasi adalah teknik pengumpulan data yang dilakukan melalui pengamatan langsung dan pencatatan kondisi atau perilaku objek yang menjadi sasaran [19]. Dalam penelitian ini, observasi dilakukan dengan bantuan dua alat uji yaitu k6 dan Grafana. K6 merupakan alat pengujian beban yang dirancang untuk memberikan pengalaman pengujian performa yang efisien dan menyenangkan bagi pengembang, serta bersifat gratis dan *open-source* [20]. Alat ini dioptimalkan untuk penggunaan sumber daya sistem yang minimal dan mampu menjalankan berbagai jenis pengujian beban, seperti *spike*, *stress*, dan *soak tests*, di lingkungan pra-produksi dan QA, sehingga mempermudah simulasi *traffic* serta pengukuran metrik seperti *total request*, *throughput*, dan *latency* [20].

Pengujian menggunakan k6 dilakukan untuk setiap algoritma *load balancing* yaitu *round robin*, *static-rr*, *least connection*, dan *random* pada tiga tingkat beban, yaitu 1000, 2000, dan 3000 pengguna. Setiap skenario beban diuji selama 1 menit dan diulang sebanyak 10 kali untuk menghasilkan data *total request*, *throughput*, dan *latency*. Setelah semua skenario dan algoritma diuji, k6 dijalankan kembali dengan durasi 12 menit per skenario untuk mengumpulkan data rata-rata penggunaan CPU dan memori, yang dipantau melalui alat monitoring.

Monitoring performa sistem secara *real-time* dilakukan dengan Grafana, perangkat lunak *open-source* untuk visualisasi dan analitik yang memungkinkan pengguna memonitor metrik yang tersimpan [21]. Pada penelitian ini Grafana digunakan untuk mengambil data penggunaan CPU

dan memori berdasarkan pengujian berdurasi 12 menit yang sebelumnya dilakukan dengan k6, dengan 10 segmen waktu diambil pada menit ke-7 hingga menit ke-11, seperti 10:00:00, 10:00:15, dan 10:00:30. Segmen ini dipilih karena pada rentang waktu ini grafik penggunaan CPU dan memori sudah mencapai kestabilan, memberikan representasi yang lebih akurat dibandingkan menit-menit awal yang masih mengalami kenaikan. Data yang diambil dari kedua alat ini memberikan gambaran mendalam mengenai kinerja algoritma *round robin*, *static-rr*, *least connection*, dan *random* pada HAProxy ingress controller, sehingga memungkinkan evaluasi menyeluruh terhadap efektivitas dan efisiensinya dalam menangani *traffic*.

Metode studi literatur juga digunakan untuk memahami konsep dan teori yang mendasari algoritma yang diuji serta teknik pengujian performa di lingkungan Kubernetes. Kegiatan ini berhubungan dengan pengumpulan referensi data pustaka melalui pembacaan, pencatatan, dan pengolahan bahan penelitian yang relevan dengan topik dalam penelitian ini [22]. Informasi yang diperoleh mendukung perancangan metode pengujian yang sistematis dan membantu dalam interpretasi hasil penelitian sesuai konteks teoritis yang ada.

2.2.4. Analisis Data

Setelah data dikumpulkan, analisis dalam penelitian ini dilakukan menggunakan statistik deskriptif. Statistik deskriptif merupakan prosedur statistika yang berfungsi untuk mengatur, meringkas, dan menjadikan data yang diperoleh menjadi mudah untuk dipahami [23]. Dalam konteks penelitian ini, perhitungan statistik deskriptif yang digunakan hanya mencakup perhitungan minimum, maksimum, dan mean, karena hanya perhitungan tersebut yang relevan dan dibutuhkan untuk analisis performa dan pembuatan diagram perbandingan.

Selanjutnya, uji normalitas dilakukan menggunakan uji *Shapiro-Wilk*, yang direkomendasikan untuk ukuran sampel kecil, seperti 10 data per kelompok [24]. Uji ini dikembangkan oleh Samuel Shapiro dan Martin Wilk pada tahun 1965, bertujuan untuk menentukan apakah data mengikuti distribusi normal, yang merupakan asumsi dasar untuk banyak uji statistik parametrik [25]. Saat ini, uji *Shapiro-Wilk* menjadi pilihan utama karena memiliki kekuatan uji yang lebih baik dibandingkan dengan metode alternatif di berbagai rentang ukuran sampel [25]. Rumus uji *Shapiro-Wilk* yang digunakan dalam penelitian ini dapat dilihat pada persamaan (1) dan (2) berikut.

$$D = \sum_{i=1}^{n}(X_i - \bar{X})^2 \quad (1)$$

$$T_3 = \frac{1}{D}\left[\sum_{i=1}^{k} a_i (X_{n-i+1} - X_i)\right]^2 \quad (2)$$

Pada persamaan (1), D merupakan varians sampel yang dihitung sebagai jumlah kuadrat dari selisih setiap nilai data (X_i) terhadap rata-rata sampel ($\bar{X}$). Varians ini menjadi dasar dalam uji *Shapiro-Wilk* untuk mengevaluasi kenormalan distribusi data. Sementara itu, pada persamaan (2), T_3 adalah statistik uji *Shapiro-Wilk* yang mengukur kesesuaian data dengan distribusi normal, di mana a_i berfungsi sebagai koefisien bobot yang bergantung pada ukuran sampel. Nilai X_{n-i+1} dan X_i masing-masing merujuk pada nilai terbesar dan terkecil dari data yang sudah diurutkan.

Hipotesis yang diuji dalam analisis ini mencakup hipotesis nol (H_0) yang menyatakan bahwa data mengikuti distribusi normal, sedangkan hipotesis alternatif (H_a) menyatakan sebaliknya. Jika nilai p dari uji *Shapiro-Wilk* lebih besar dari 0,05, maka H_0 tidak ditolak, menunjukkan bahwa data berdistribusi normal. Sebaliknya, jika nilai p kurang dari 0,05, H_0 ditolak, dan data dianggap tidak berdistribusi normal, yang mengharuskan penggunaan teknik analisis non-parametrik.

Selain itu, dalam penelitian ini uji homogenitas varians dilakukan menggunakan *Levene's test* untuk memastikan keseragaman varians antar kelompok [26]. Uji ini penting karena banyak uji statistik, termasuk *ANOVA*, memerlukan keseragaman varians di seluruh kelompok yang dibandingkan. Uji *Levene* menggunakan analisis varian satu arah untuk memeriksa signifikansi perbedaan varians antar kelompok dengan cara mentransformasikan data melalui penghitungan selisih setiap skor dengan rata-rata kelompoknya, yang membantu dalam mengidentifikasi perbedaan varians secara lebih akurat [26]. Berikut adalah rumus uji *Levene* yang dapat dilihat pada persamaan (3).

$$\mathrm{W} = \frac{(N-k)\sum_{i=1}^{k} n_i(\bar{z}_{i.} - \bar{z}_{..})^2}{(k-1)\sum_{i=1}^{k}\sum_{j=1}^{n_i}(z_{ij} - \bar{z}_{i.})^2} \quad (3)$$

Rumus uji *Levene* yang ditunjukkan dalam persamaan (3) digunakan untuk menguji homogenitas varians antar kelompok. Dalam rumus ini, W adalah statistik uji, N adalah total pengamatan, k adalah jumlah kelompok, dan n_i adalah jumlah pengamatan dalam kelompok i. Hasil dari uji ini akan menunjukkan apakah terdapat perbedaan signifikan dalam varians antar kelompok yang dibandingkan.

Hipotesis yang diuji dalam uji *Levene* ini terdiri dari dua bagian. Hipotesis nol (H_0) menyatakan bahwa tidak ada perbedaan signifikan dalam varians antar kelompok, atau dengan kata lain, semua kelompok memiliki varians yang sama. Sebaliknya, hipotesis alternatif (H_a) menyatakan bahwa setidaknya ada satu kelompok yang variansnya berbeda dari kelompok lainnya. Pada umumnya, jika nilai p yang dihasilkan dari uji *Levene* lebih besar dari 0,05, maka hipotesis nol tidak dapat ditolak, yang menunjukkan bahwa asumsi keseragaman varians terpenuhi. Uji *Levene* ini penting untuk memastikan bahwa asumsi keseragaman varians terpenuhi sebelum melakukan

analisis lebih lanjut, seperti *ANOVA*, yang memerlukan varians yang homogen agar hasilnya valid.

Jika data memenuhi asumsi normalitas dan homogenitas, *ANOVA One-Way* digunakan untuk menguji perbedaan signifikan antar algoritma [27]. Namun, jika data tidak berdistribusi normal atau varians tidak homogen, uji *Kruskal-Wallis* diterapkan sebagai alternatif untuk membandingkan perbedaan antar kelompok [28]. Rumus kedua uji beda tersebut dapat dilihat pada persamaan (4) dan (5) berikut.

$$F = \frac{MSB}{MSW} \tag{4}$$

$$H = \frac{12}{n_T(n_T+1)} \frac{\sum_{j=1}^{k} R_j^2}{n_j} - 3(n_T + 1) \tag{5}$$

Persamaan (4) untuk *ANOVA One-Way* menghitung statistik *F*, yang membandingkan *mean square* antara (*MSB*) dan dalam kelompok (*MSW*), sementara persamaan (5) untuk uji *Kruskal-Wallis* menghitung statistik *H* untuk data tidak normal. Hipotesis nol (H_0) untuk kedua uji menyatakan bahwa tidak ada perbedaan signifikan antara rata-rata kelompok, sedangkan hipotesis alternatif (H_a) menyatakan bahwa setidaknya satu kelompok memiliki rata-rata yang berbeda. Nilai p kurang dari 0,05 digunakan sebagai ambang batas untuk memutuskan apakah hipotesis nol dapat ditolak, yang menunjukkan adanya perbedaan signifikan antara kelompok, baik dalam *ANOVA One-Way* maupun *Kruskal-Wallis*.

2.2.5. Penarikan Kesimpulan

Pada tahap penarikan kesimpulan, hasil dari analisis data dievaluasi untuk menentukan temuan utama penelitian. Kesimpulan diambil berdasarkan hasil analisis data untuk menentukan algoritma *load balancing* yang paling efisien dalam konteks HAProxy ingress controller di Kubernetes. Berdasarkan hasil pengujian, kesimpulan ini akan memberikan rekomendasi tentang algoritma *load balancing* yang optimal, serta panduan praktis untuk pemilihan algoritma yang paling sesuai dalam pengelolaan lalu lintas di lingkungan Kubernetes.

3. HASIL DAN PEMBAHASAN

Pada bagian ini, hasil dan pembahasan pengujian dari berbagai algoritma *load balancing* akan dijelaskan secara keseluruhan. Pengujian dilakukan pada tiga skenario beban, yaitu 1000, 2000, dan 3000 pengguna. Metrik utama seperti total *request*, *throughput*, *latency*, serta penggunaan CPU dan memori digunakan untuk mengevaluasi kinerja masing-masing algoritma. Analisis ini bertujuan mengidentifikasi respons setiap algoritma terhadap berbagai tingkat beban dan penggunaan sumber daya.

Untuk menganalisis kinerja dan efisiensi algoritma *load balancing*, berbagai teknik statistik diterapkan menggunakan software SPSS. Teknik yang digunakan meliputi statistik deskriptif untuk gambaran umum, uji normalitas untuk memeriksa distribusi data, dan uji homogenitas untuk menilai kesamaan varians. *ANOVA One-Way* digunakan untuk membandingkan rata-rata antar kelompok, sementara uji *Kruskal-Wallis H* diterapkan jika asumsi normalitas tidak terpenuhi, guna memberikan perbandingan kinerja yang lebih tepat.

3.1. Analisa Performa

Analisis performa algoritma *load balancing* mencakup *total request*, *throughput*, dan latency untuk mengukur kemampuan algoritma dalam menangani lalu lintas jaringan di berbagai beban pengguna. *Total request* menunjukkan bagaimana algoritma mendistribusikan permintaan ke server dan beradaptasi dengan peningkatan jumlah pengguna. Throughput mengukur efektivitas algoritma dalam mempertahankan jumlah *request per second* (RPS) saat beban meningkat. Pengukuran *latency* memberikan gambaran tentang kecepatan respons sistem dan bagaimana algoritma mempertahankan waktu respons yang rendah. Analisis ini membantu memahami kinerja algoritma dalam berbagai kondisi beban, termasuk stabilitas dan responsivitas layanan.

Tabel 2. Statistik Deskriptif Performa

Kelompok		Total Request			Throughput			Latency		
		Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
1000	Round Robin	58618	58907	59024	960.9	966.1510	968.69	9.49	11.4456	13.50
	Static RR	58997	59020	59213	967.04	968.9357	970.91	5.95	9.6118	11.17
	Least Connection	58993	59030	59136	967.24	968.4155	971.53	6.31	9.2108	10.27
	Random	58965	59018	59072	966.57	968.2549	969.93	8.51	9.8047	11.87
2000	Round Robin	113523	114879	115681	1865.48	1883.3296	1896.34	16.44	21.3212	29.58
	Static RR	112923	115169	115596	1882.36	1887.8925	1894.93	16.41	19.5560	23.33
	Least Connection	113416	115121	115580	1857.6	1887.1502	1895.04	17.16	19.7748	31.28
	Random	113584	114534	115380	1840.92	1877.3887	1890.69	17.38	22.2599	37.04
3000	Round Robin	155177	165612	167209	2687.74	2715.2999	2742.33	36.04	43.5236	51.57
	Static RR	152944	166201	167871	2684.03	2724.3022	2752.28	35.27	42.8883	54.28
	Least Connection	154662	165678	166734	2691.76	2715.4796	2733.93	36.98	42.8803	48.41
	Random	154804	164351	166890	2663.49	2693.6929	2735.62	37.18	49.5482	54.75

Tabel 2 menunjukkan statistik deskriptif perbandingan performa dari empat algoritma *load balancing* pada tiga tingkat beban pengguna yang berbeda. Analisis *total request* menunjukkan variasi performa di antara algoritma *load balancing* pada tiga tingkat beban pengguna. Pada 1000 pengguna, *least connection* mencatat rata-rata tertinggi dengan 59030 request, diikuti oleh *static-rr* dengan 59020. Saat beban meningkat ke 2000 pengguna, *static-rr* mengambil alih dengan rata-rata tertinggi 115169 request, sementara *least connection* berada di posisi kedua dengan 115121 request. Pada beban 3000 pengguna, *static-rr* mempertahankan performa terbaiknya dengan rata-rata 166201 request, diikuti oleh *least connection* dengan 165678, menunjukkan konsistensi *static-rr* dalam menangani beban tinggi.

Dalam hal *throughput*, pola yang serupa terlihat. Pada 1000 pengguna, *static-rr* memimpin dengan rata-rata 968.9357, diikuti oleh *least connection* dengan 968.4155. Ketika beban meningkat ke 2000 pengguna, *static-rr* mempertahankan keunggulannya dengan rata-rata 1887.8925, sementara *random* berada di posisi kedua dengan throughput rata-rata 1887.3887. Pada beban 3000 pengguna, *static-rr* tetap unggul dengan throughput rata-rata 2724.3022, sementara *least connection* berada di posisi kedua dengan 2715.4796. Hal ini menunjukkan kemampuan *static-rr* yang konsisten dalam mendistribusikan beban secara efisien pada berbagai tingkat pengguna.

Analisis *latency* menunjukkan variasi performa yang berbeda di antara algoritma pada berbagai tingkat beban pengguna. Pada beban 1000 pengguna, algoritma *least connection* mencatat latency terendah dengan rata-rata 9.2108 ms, sedangkan *static-rr* mengikuti dengan latency rata-rata 9.6118 ms. Saat jumlah pengguna meningkat menjadi 2000, *static-rr* mengambil alih dengan latency rata-rata 19.5560 ms, sementara *least connection* berada di posisi kedua pada 19.7748 ms. Namun, pada tingkat beban tertinggi, yaitu 3000 pengguna, *least connection* berhasil mengungguli dengan latency rata-rata 42.8803 ms, yang sedikit lebih baik dibandingkan dengan *static-rr* yang memiliki *latency* 42.8883 ms. Temuan ini mengindikasikan bahwa meskipun *static-rr* umumnya menunjukkan performa yang lebih baik pada beban sedang, *least connection* bisa menjadi pilihan yang lebih efektif dalam mengurangi *latency* pada beban yang sangat tinggi.

Selanjutnya, akan dilakukan uji beda untuk mengevaluasi signifikansi perbedaan kinerja antara algoritma load balancing yang telah dianalisis. Dua metode akan digunakan untuk mengevaluasi perbedaan antar kelompok algoritma yaitu *Anova One-Way* dan *Kruskal Wallis H*. *Anova One-Way* akan diterapkan untuk menguji perbedaan mean antar kelompok, dengan asumsi data terdistribusi normal dan memiliki varians yang homogen. Sementara itu, *Kruskal Wallis* H akan digunakan sebagai alternatif non-parametrik, yang tidak mengasumsikan normalitas distribusi data. Kedua uji ini akan memberikan pemahaman yang lebih mendalam tentang sejauh mana perbedaan performa antar algoritma *load balancing* bersifat signifikan secara statistik, sehingga dapat mendukung pengambilan keputusan yang lebih tepat dalam memilih algoritma yang paling sesuai untuk berbagai skenario beban.

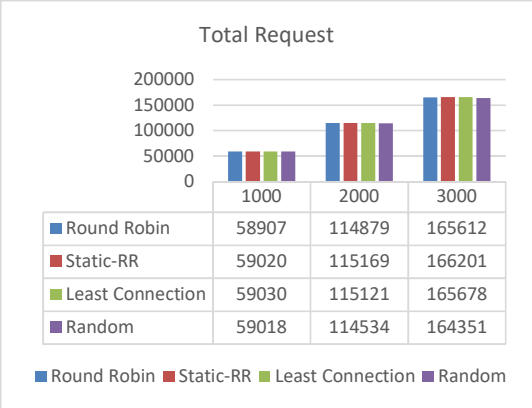
Tabel 3. Uji Signifikansi Perbedaan Performa

Jumlah Pengguna	Significance		
	Total Request	Throughput	Latency
1000	0.081	0.107	0.004
2000	0.032	0.029	0.153
3000	0.011	0.008	0.019

Tabel 3 menunjukkan hasil uji beda yang mengindikasikan bahwa performa algoritma *load balancing* bervariasi secara signifikan berdasarkan jumlah pengguna. Pada tingkat 1000 pengguna, nilai signifikansi untuk *total request* dan *throughput* masing-masing adalah 0.081 dan 0.107, menunjukkan bahwa tidak ada perbedaan yang signifikan antara algoritma dalam metrik ini. Namun, *latency* memiliki nilai signifikansi yang rendah, yaitu 0.004, yang menunjukkan perbedaan yang signifikan dalam performa *latency* antar algoritma.

Saat jumlah pengguna meningkat menjadi 2000, nilai signifikansi untuk *total request* dan *throughput* menurun menjadi 0.032 dan 0.029, mengindikasikan adanya perbedaan signifikan dalam performa antara algoritma pada kedua metrik ini. Sebaliknya, *latency* menunjukkan nilai signifikansi yang lebih tinggi, yaitu 0.153, yang menunjukkan bahwa performa *latency* tidak berbeda secara signifikan antar algoritma pada tingkat beban ini.

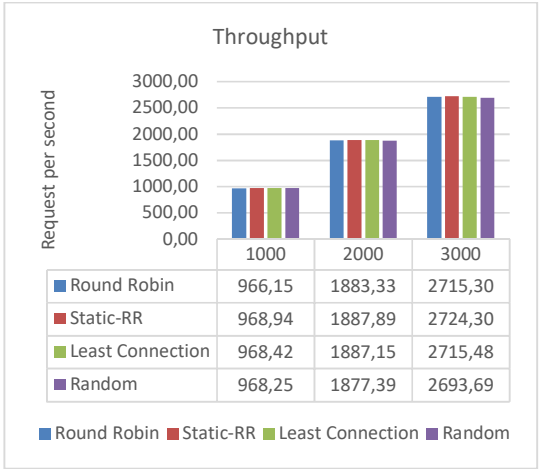
Pada tingkat beban tertinggi, yaitu 3000 pengguna, hasil menunjukkan signifikansi yang kuat dengan nilai 0.011 untuk *total request* dan 0.008 untuk *throughput*, menandakan adanya perbedaan yang signifikan di antara algoritma. Nilai signifikansi untuk *latency* juga tetap rendah pada 0.019, menunjukkan perbedaan yang signifikan, meskipun tidak sekuat pada metrik lainnya. Secara keseluruhan, hasil ini menekankan bahwa seiring dengan meningkatnya jumlah pengguna, perbedaan dalam efektivitas algoritma *load balancing* menjadi lebih jelas, terutama dalam hal *total request* dan *throughput*, yang dapat membantu dalam pengambilan keputusan untuk memilih algoritma yang tepat berdasarkan kondisi beban yang ada.



Gambar 3. Perbandingan Rata-Rata *Total Request*

Gambar 3 pada diagram tersebut menunjukkan bahwa rata-rata *total request* yang diproses oleh masing-masing algoritma menunjukkan hasil yang serupa di setiap skenario jumlah pengguna, meskipun terdapat perbedaan kecil yang signifikan. Pada tingkat 1000 pengguna, algoritma *least connection* menunjukkan performa terbaik dengan rata-rata *total request* tertinggi sebesar 59030, sedangkan *static-rr* dan *random* berada di posisi kedua dan ketiga dengan rata-rata 59020 dan 59018. Ketika jumlah pengguna meningkat menjadi 2000, *static-rr* mengambil alih dengan rata-rata 115169 *total request*, sedangkan *least connection* menduduki posisi kedua dengan 115121. Pada tingkat 3000 pengguna, *static-rr* kembali mempertahankan posisi teratas dengan rata-rata *total request* sebesar 166201, diikuti oleh *least connection* yang mencatat 165678 *request*.

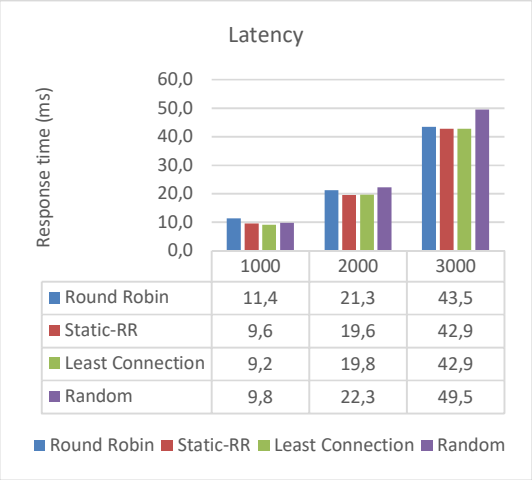
Secara keseluruhan, hasil ini menunjukkan bahwa meskipun semua algoritma dapat mengelola permintaan dengan baik, *static-rr* konsisten dalam memberikan performa terbaik pada beban tinggi. Sementara itu, *least connection* terbukti efektif dan dapat diandalkan pada skenario dengan jumlah pengguna yang lebih rendah, sehingga memberikan alternatif yang kuat dalam strategi *load balancing* pada situasi tersebut.



Gambar 4. Perbandingan Rata-Rata *Throughput*

Gambar 4 memperlihatkan diagram yang menunjukkan perbandingan *throughput* rata-rata untuk algoritma *round robin*, *static-rr*, *least connection*, dan *random* pada tiga skenario jumlah pengguna yang berbeda. Pada skenario 1000 pengguna, algoritma *static-rr* mencatat *throughput* tertinggi sebesar 968.94, diikuti oleh *least connection* dan *random* dengan nilai 968.42 dan 968.25, sedangkan *round robin* berada di posisi terakhir dengan throughput 966.15. Temuan ini menunjukkan bahwa pada tingkat beban rendah, algoritma *static-rr* mampu mengoptimalkan distribusi permintaan, memberikan performa yang lebih baik dibandingkan algoritma lainnya. Keunggulan ini dapat menjadi faktor penting dalam aplikasi yang memerlukan respons cepat dan efisien pada jumlah pengguna yang lebih sedikit, di mana performa *throughput* dapat langsung mempengaruhi pengalaman pengguna.

Ketika jumlah pengguna meningkat menjadi 2000, *static-rr* tetap unggul dengan nilai 1887.89, menunjukkan kemampuannya dalam menghadapi beban yang lebih tinggi. Sementara itu, *least connection* dan *round robin* mencatat nilai yang hampir setara, yakni 1887.15 dan 1883.33, yang mengindikasikan bahwa kedua algoritma tersebut masih dapat berfungsi dengan baik dalam situasi beban yang lebih intensif, meskipun tidak seefisien *static-rr*. Pada skenario 3000 pengguna, *static-rr* kembali menunjukkan kinerja terbaik dengan *throughput* 2724.30, diikuti oleh *least connection* dengan 2715.48, sementara *random* mencatat nilai terendah di 2693.69. Hasil ini menegaskan bahwa *static-rr* secara konsisten mendemonstrasikan kemampuan optimal dalam mendistribusikan throughput di berbagai tingkat beban, yang menjadikannya pilihan unggul dalam konteks *load balancing* untuk aplikasi yang memerlukan kinerja tinggi.



Gambar 5. Perbandingan Rata-Rata *Latency*

Gambar 5 menunjukkan perbandingan rata-rata *latency* yang dialami oleh masing-masing algoritma pada skenario dengan jumlah pengguna yang bervariasi dalam bentuk diagram. Pada skenario 1000 pengguna, *least connection* mencatat *latency* terendah dengan 9.2 ms, diikuti oleh *static-rr* dan *random* dengan 9.6 ms dan 9.8 ms, sedangkan *round robin* berada di urutan terakhir dengan 11.4 ms. Ketika jumlah pengguna meningkat menjadi 2000, *random* menunjukkan *latency* tertinggi di 22.3 ms, sementara *least connection* dan *static-rr* memiliki nilai yang hampir setara, yakni 19.8 ms dan 19.6 ms. Pada skenario 3000 pengguna, *least connection* dan *static-rr* mencatat *latency* yang sama sebesar 42.9 ms, tetapi *random* mencatat latency tertinggi dengan 49.5 ms, menunjukkan bahwa algoritma *least connection* lebih efisien dalam mengelola *latency* di berbagai tingkat beban.

Dalam praktik industri, hasil analisis performa ini memiliki implikasi penting dalam pemilihan algoritma *load balancing* berdasarkan kebutuhan aplikasi. Misalnya, untuk aplikasi *e-commerce* dengan lonjakan pengunjung, algoritma *round robin* atau *least connection* lebih cocok karena dapat menangani perubahan beban secara dinamis. Sebaliknya, algoritma *static-rr* lebih sesuai untuk aplikasi dengan *traffic* yang stabil dan dapat diprediksi, seperti aplikasi internal perusahaan yang digunakan untuk manajemen inventaris atau aplikasi berbasis ERP (*Enterprise Resource Planning*), karena mampu menjaga *throughput* yang optimal tanpa terpengaruh perubahan jumlah server atau *pod*. Bagi aplikasi yang mengutamakan latensi rendah, seperti *video streaming* atau layanan *real-time*, algoritma *least connection* lebih efektif dalam mengurangi *latency*. Pemilihan algoritma bergantung pada prioritas aplikasi, apakah itu *throughput* tinggi, latensi rendah, atau kestabilan pada beban yang konsisten.

3.2. Analisa Efisiensi

Efisiensi penggunaan CPU dan memori adalah aspek krusial dalam memastikan bahwa sumber daya komputasi dimanfaatkan secara optimal, terutama dalam lingkungan dengan beban yang terus meningkat. Pemanfaatan CPU yang berlebihan dapat mengakibatkan penurunan kinerja sistem secara keseluruhan. Oleh karena itu, memahami bagaimana setiap algoritma *load balancing* memanfaatkan sumber daya prosesor di berbagai tingkat beban menjadi langkah penting untuk menentukan efisiensi operasionalnya. Dengan mengamati pemanfaatan CPU, kita dapat menilai sejauh mana algoritma tersebut mampu menjaga performa yang stabil saat beban pengguna meningkat, serta mengidentifikasi potensi *bottleneck* yang dapat mempengaruhi responsivitas sistem.

Di sisi lain, penggunaan memori yang efisien juga sangat penting untuk memastikan stabilitas sistem, terutama dalam kondisi beban tinggi. Algoritma *load balancing* yang mampu mengelola memori dengan baik akan menghasilkan sistem yang lebih stabil dan responsif, bahkan ketika beban pengguna meningkat secara signifikan. Dalam analisis ini, perhatian diberikan pada cara tiap algoritma menangani permintaan dalam skenario yang berbeda, dengan fokus pada kemampuan mereka dalam meminimalkan penggunaan memori sambil tetap mempertahankan kinerja yang optimal. Dengan demikian, pemahaman yang mendalam tentang manajemen memori dapat membantu dalam pengembangan strategi yang lebih efektif dalam pengelolaan sumber daya.

Temuan dari evaluasi ini memberikan wawasan penting tentang efektivitas pengelolaan sumber daya oleh setiap algoritma. Dengan analisis yang komprehensif, pengembang dan administrator sistem dapat membuat keputusan yang lebih baik terkait pemilihan algoritma *load balancing* yang paling sesuai dengan kebutuhan spesifik dalam pengelolaan sumber daya. Selain itu, evaluasi ini berfungsi sebagai panduan untuk meningkatkan kinerja sistem secara keseluruhan, meminimalkan risiko implementasi teknologi yang tidak efisien, dan memastikan bahwa sistem dapat beroperasi secara optimal dalam berbagai kondisi beban.

Tabel 4. Statistik Deskriptif Efisiensi

Kelompok		CPU Usage			Memory Usage		
		Min	Mean	Max	Min	Mean	Max
1000	Round Robin	0.2579	0.2616	0.2664	0.365	0.367	0.369
	Static RR	0.2538	0.2557	0.2583	0.351	0.357	0.359
	Least Connection	0.2676	0.2699	0.2734	0.358	0.360	0.362
	Random	0.2604	0.2685	0.2762	0.359	0.363	0.367
2000	Round Robin	0.4109	0.4168	0.4201	0.366	0.367	0.369
	Static RR	0.4082	0.4106	0.4154	0.359	0.360	0.360
	Least Connection	0.4297	0.4359	0.4416	0.360	0.365	0.368
	Random	0.4251	0.4320	0.4444	0.362	0.364	0.366
3000	Round Robin	0.5333	0.5382	0.5458	0.368	0.370	0.371
	Static RR	0.5182	0.5303	0.5449	0.361	0.363	0.365
	Least Connection	0.5505	0.5600	0.5685	0.365	0.367	0.371
	Random	0.5373	0.5535	0.5750	0.365	0.367	0.370

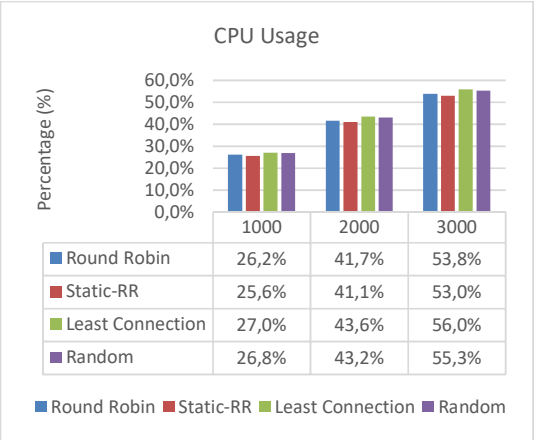
Tabel 4 menunjukkan statistik deskriptif perbandingan efisiensi sumber daya antara empat algoritma *load balancing* pada tiga tingkat beban pengguna yang berbeda. Pada 1000 pengguna, *static-rr* menunjukkan efisiensi terbaik dengan rata-rata penggunaan CPU terendah sebesar 0.2557, sementara *least connection* memiliki penggunaan tertinggi dengan 0.2699. Ketika beban meningkat ke 2000 pengguna, *static-rr* tetap paling efisien dengan rata-rata 0.4106, diikuti oleh *round robin* dengan 0.4168. Pada beban tertinggi 3000 pengguna, *static-rr* konsisten mempertahankan efisiensi terbaiknya dengan rata-rata penggunaan CPU 0.5303, sementara *least connection* menunjukkan penggunaan tertinggi dengan 0.5600. Pola ini mengindikasikan bahwa *static-rr* cenderung lebih efisien dalam penggunaan CPU di berbagai tingkat beban.

Dalam hal penggunaan memori, perbedaan antar algoritma relatif kecil namun tetap terlihat. Pada 1000 pengguna, *static-rr* kembali menunjukkan efisiensi tertinggi dengan rata-rata penggunaan memori 0.357, diikuti oleh *least connection* dengan 0.360. Saat beban meningkat ke 2000 pengguna, *static-rr* mempertahankan efisiensinya dengan penggunaan memori terendah sebesar 0.360, sementara *least connection* dan *round robin* menunjukkan penggunaan yang lebih tinggi masing-masing 0.365 dan 0.367. Pada beban 3000 pengguna, *static-rr* tetap paling efisien dengan penggunaan memori 0.363, sedangkan *round robin*, *least connection*, dan *random* memiliki penggunaan yang sama yaitu 0.370, 0.367, dan 0.367 secara berurutan. Hasil ini menunjukkan bahwa *static-rr* secara konsisten memiliki efisiensi penggunaan memori yang lebih baik di berbagai tingkat beban dibandingkan algoritma lainnya.

Tabel 5. Uji Signifikansi Perbedaan Efisiensi

Jumlah Pengguna	Significance	
	CPU Usage	Memory Usage
1000	0.000	0.000
2000	0.000	0.000
3000	0.000	0.000

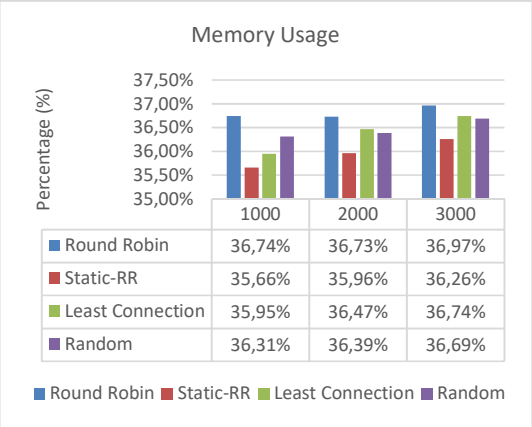
Tabel 5 menunjukkan hasil uji signifikansi terkait perbedaan efisiensi antar algoritma *load balancing*, yang mengungkapkan temuan yang konsisten dan sangat signifikan di seluruh tingkat beban pengguna. Berdasarkan diatas, baik untuk penggunaan CPU maupun memori, diperoleh nilai signifikansi sebesar 0.000 pada setiap tingkat beban yang diuji. Nilai signifikansi yang sangat rendah ini ($p < 0.05$) menunjukkan adanya perbedaan yang sangat signifikan secara statistik dalam efisiensi penggunaan sumber daya antara algoritma yang dianalisis. Konsistensi hasil pada semua tingkat beban ini memperkuat kesimpulan bahwa pemilihan algoritma *load balancing* berdampak signifikan dan terukur terhadap efisiensi penggunaan CPU dan memori. Temuan ini memberikan implikasi penting bagi pengambilan keputusan dalam memilih algoritma *load balancing* yang paling optimal, mengingat perbedaan kinerja yang signifikan dapat mempengaruhi efisiensi sistem secara keseluruhan, terutama dalam lingkungan dengan variasi beban kerja.



Gambar 6. Perbandingan Rata-Rata *CPU Usage*

Gambar 6 menampilkan perbandingan rata-rata penggunaan CPU di antara empat algoritma *load balancing* pada tiga tingkat beban pengguna yang berbeda. Pada 1000 pengguna, *static-rr* mencatat penggunaan CPU terendah sebesar 25,6%, sementara *least connection* tertinggi di 27,0%. Saat beban meningkat ke 2000 pengguna, semua algoritma menunjukkan peningkatan penggunaan CPU, dengan *static-rr* tetap paling efisien di 41,1%

dan *least connection* tertinggi di 43,6%. Pada 3000 pengguna, *static-rr* tetap unggul dengan 53,0%, sedangkan *least connection* mencatat 56,0%. *Round robin* dan *random* berada di antara kedua algoritma tersebut di semua skenario beban. Diagram ini menunjukkan bahwa *static-rr* secara konsisten paling efisien dalam penggunaan CPU.



Gambar 7. Perbandingan Rata-Rata *Memory Usage*

Gambar 7 menampilkan perbandingan penggunaan memori di antara empat algoritma *load balancing* pada tiga tingkat beban pengguna yang berbeda. Pada beban 1000 pengguna, *static-rr* menunjukkan efisiensi tertinggi dengan penggunaan memori terendah sebesar 35.66%, diikuti oleh *least connection* dengan 35.95%. *Round robin* memiliki penggunaan memori tertinggi pada 36.74%. Ketika beban meningkat ke 2000 pengguna, pola serupa terlihat dengan sedikit peningkatan penggunaan memori untuk semua algoritma. *Static-rr* tetap paling efisien dengan 35.96%, sementara *round robin* masih yang tertinggi dengan 36.73%. Pada beban maksimum 3000 pengguna, tren ini berlanjut dengan peningkatan marginal. *Static-rr* konsisten mempertahankan efisiensi tertinggi dengan 36.26%, sedangkan *round robin* tetap yang tertinggi dengan 36.97%. *Least connection* dan *random* secara konsisten berada di antara kedua ekstrem ini di semua tingkat beban. Diagram ini mengilustrasikan bahwa meskipun terjadi peningkatan penggunaan memori seiring bertambahnya beban, peningkatannya relatif kecil dibandingkan dengan penggunaan CPU. *Static-rr* secara konsisten menunjukkan efisiensi terbaik dalam pengelolaan sumber daya memori di berbagai tingkat beban.

Berdasarkan analisis efisiensi yang telah dilakukan terkait penggunaan CPU dan memori, pemilihan algoritma *load balancing* yang sesuai memiliki dampak signifikan terhadap kinerja sistem dalam pengelolaan sumber daya. Sebagai contoh, pada aplikasi pemantauan cuaca yang merupakan *memory-intensive application*, algoritma *static-rr* lebih cocok karena kemampuannya dalam mengelola memori secara efisien, menjaga kinerja sistem tetap stabil meskipun beban meningkat. Demikian pula, untuk aplikasi render grafis berbasis *cloud* yang termasuk *CPU-intensive application*, algoritma *static-rr* juga dapat lebih efisien dalam penggunaan CPU, menghindari penggunaan prosesor yang berlebihan dan memastikan sumber daya tetap optimal. Temuan ini menunjukkan bahwa *static-rr* dapat menjadi pilihan yang efektif untuk aplikasi yang memerlukan pengelolaan sumber daya yang lebih konsisten dan optimal, baik dari sisi memori maupun CPU. Namun, perlu dipertimbangkan juga bahwa *static-rr* bersifat statis, sehingga kurang fleksibel dalam menangani fluktuasi beban yang cepat.

3.3. Komparasi Penelitian Terdahulu

Untuk memperkuat temuan, hasil penelitian ini dibandingkan dengan penelitian terdahulu yang relevan dengan penelitian ini. Sebagian besar penelitian sebelumnya fokus pada perbandingan dua algoritma, seperti *round robin* dan *least connection*, dalam konteks eksternal load balancer. Selain itu, ada juga penelitian yang membahas *load balancing* di internal Kubernetes dengan metode atau komponen yang berbeda. Perbandingan yang dilakukan dalam peneltian ini bertujuan untuk mengidentifikasi persamaan dan perbedaan hasil pengujian yang berkaitan dengan algoritma *load balancing*, sehingga memungkinkan penilaian yang lebih komprehensif mengenai performa algoritma dalam konteks yang lebih beragam. Perbandingan lebih rinci antara penelitian ini dan penelitian terdahulu dapat dilihat pada Tabel 6 berikut.

Tabel 6. Komparasi Penelitian Terdahulu

Judul	**Hasil**	**Perbedaan dengan penelitian ini**
Perbandingan Algoritma Round Robin dan Algoritma Least Connection pada Haproxy untuk Load Balancing Web Server [11]	*Least connection* unggul dalam *response time*, sedangkan *round robin* lebih efisien dalam CPU *utilization*.	Penelitian ini juga meneliti penggunaan CPU, tetapi *static-rr* lebih efisien pada beban tinggi.
Cluster implementation on mini Raspberry Pi computers using Round Robin Algorithm [29]	*Round robin* menunjukkan stabilitas *response time* dan *throughput* lebih baik dibandingkan *least connection*.	Penelitian ini membahas *latency* dan *throughput*, dengan least connection unggul dalam *latency* pada beban rendah.
Load Balancing Method Performance Analysis on Haproxy and Router OS [30]	*Least connection* unggul dalam *delay* dan *jitter* dibandingkan *round robin* dan *source*.	Tidak ada pembahasan tentang *jitter* dan *delay*, fokus lebih pada *total request*, *throughput*, dan penggunaan CPU.
Load Balancing Analysis Using Round-Robin and Least Connection Algorithms for Server Service	*Round robin* lebih baik pada beban rendah, *least connection* unggul saat koneksi meningkat.	Penelitian ini tidak menekankan performa *round robin* pada beban rendah, tetapi setuju *least connection* unggul saat beban meningkat.

Response Time [31]		
Load Balancer Tuning: Comparative Analysis of HAProxy Load Balancing Methods [13]	Algoritma *random*, *leastconn*, *roundrobin*, dan *static-rr* memiliki performa serupa dengan sedikit variasi respons.	Penelitian ini menemukan *static-rr* unggul dalam *total request* dan *throughput*, dengan perbedaan signifikan di efisiensi.
Perancangan dan Implementasi Load Balancing menggunakan Algoritma Least Connection dan Ip Hash pada Kubernetes [12]	*Least connection* dan *IP hash* unggul dalam *response time*, *throughput*, dan *request loss* di Kubernetes dengan Oracle Cloud.	Fokus penelitian ini pada Kubernetes dengan HAProxy, sementara penelitian Luthfi dkk. membahas perangkat Oracle Cloud.
Experimental Setup for Investigating the Efficient Load Balancing Algorithms on Virtual Cloud [32]	Algoritma *round robin* lebih unggul dari *least connection* dalam performa pada *external load balancer* menggunakan HAProxy.	Penelitian ini menunjukkan *static-rr* unggul dalam performa dan juga mengkaji efisiensi CPU dan memori pada ingress controller di Kubernetes, yang tidak diteliti oleh Alankar dkk.
Enhancing the performance of mobile networks using Kubernetes – Load balancing traffic by utilizing workload estimation [33]	Algoritma *least load* memiliki performa terbaik, diikuti *least connection* dan *random*, sementara *round robin* memiliki performa terburuk.	Penelitian ini menggunakan Ingress Controller di Kubernetes dan membandingkan algoritma *static-rr*, sementara Laukka & Fransson menggunakan KPNG dan algoritma *least load*.
Load-Balancing of Kubernetes-Based Edge Computing Infrastructure Using Resource Adaptive Proxy [3]	Algoritma RAP meningkatkan *throughput* dan mengurangi latensi dengan mengarahkan permintaan ke *node* terbaik berdasarkan status sumber daya.	Penelitian ini menggunakan Ingress Controller, sedangkan penelitian yang dilakukan Nguyen dkk. mengembangkan RAP untuk meningkatkan kemampuan *kube-proxy* di Kubernetes.

4. DISKUSI

Hasil penelitian yang telah dilakukan menunjukkan bahwa setiap algoritma memiliki keunggulan dan kekurangan yang berbeda. *Static-rr* terbukti paling efektif dalam hal *total request*, *throughput*, dan efisiensi dalam penggunaan CPU dan memori, terutama pada beban tinggi. *Least connection*, di sisi lain, unggul dalam latensi pada beban rendah, menjadikannya pilihan yang baik untuk skenario yang memprioritaskan respons cepat. *Round robin* dan *random* cenderung kurang optimal pada beban tinggi, meskipun tetap stabil pada beban ringan. Temuan ini menegaskan bahwa pemilihan algoritma yang tepat sangat bergantung pada karakteristik beban pengguna.

Berdasarkan perbandingan dengan penelitian sebelumnya pada Tabel 6, beberapa penelitian tersebut memiliki hasil yang serupa dengan penelitian ini. Sebagai contoh, penelitian yang dilakukan oleh Solehudin [11] dan Permana [30] menunjukkan bahwa algoritma *least connection* memiliki kinerja terbaik jika dibandingkan dengan *round robin* dari segi latensi. Hasil ini serupa dengan hasil pada penelitian yang telah peneliti lakukan, tetapi yang membedakan adalah jumlah algoritma yang dibandingkan dan juga hasil akhir yang berbeda dimana algoritma *static-rr* lebih unggul dibandingkan ketiga algoritma lainnya.

Di sisi lain, penelitian oleh Laukka [33] dan Nyuyen [3] juga membahas *load balancing* dalam konteks Kubernetes, tetapi menggunakan metode yang berbeda dengan penelitian ini, yaitu melalui *kube-proxy* untuk melakukan *load balancing* antar *pod* dalam *cluster*. Berdasarkan penelitian-penelitian tersebut, peneliti mendapati bahwa terdapat metode *load balancing* lain pada internal Kubernetes selain menggunakan ingress controller yaitu menggunakan *kube-proxy*. Namun, jika dibandingkan dengan *kube-proxy*, ingress controller tidak hanya dapat melakukan *load balancing* antar *pod* tetapi juga mampu mengekspos layanan ke luar *cluster* dengan fitur *routing* yang lebih fleksibel dan opsi keamanan tambahan.

5. KESIMPULAN

Berdasarkan hasil penelitian, dapat disimpulkan bahwa algoritma *static-rr* menunjukkan kinerja terbaik dalam hal *total request*, *throughput*, serta efisiensi penggunaan CPU dan memori pada beban tinggi. Sementara itu, algoritma *least connection* lebih unggul dalam mengurangi latensi pada beban rendah. Algoritma *round robin* dan *random* stabil pada beban ringan, namun kurang efisien pada beban tinggi, terutama dalam pemanfaatan sumber daya CPU dan memori.

Hasil analisis juga mengungkapkan adanya perbedaan signifikan dalam kinerja dan efisiensi antara algoritma yang diuji. Uji beda menggunakan *ANOVA One-Way* dan *Kruskal-Wallis H* menunjukkan bahwa perbedaan *total request* dan *throughput* sangat signifikan pada beban tinggi, dengan nilai p di bawah 0.05. Perbedaan signifikan juga terlihat dalam efisiensi penggunaan CPU dan memori pada semua tingkat beban, dengan nilai p 0.00, yang menegaskan adanya perbedaan nyata dalam pemanfaatan sumber daya antara algoritma.

Berdasarkan temuan ini, disarankan agar algoritma *static-rr* digunakan untuk aplikasi dengan lalu lintas yang stabil dan dapat diprediksi, seperti aplikasi manajemen inventaris atau ERP (*Enterprise Resource Planning*) dalam lingkungan perusahaan, karena dapat menjaga *throughput* yang optimal tanpa memerlukan perubahan *pod* dinamis atau penyesuaian beban yang sering. Algoritma *least connection* lebih cocok untuk aplikasi yang memerlukan latensi rendah, seperti aplikasi perpesanan instan atau game online, yang membutuhkan respons cepat pada beban rendah. Sedangkan algoritma *round robin* dan *random* lebih tepat digunakan untuk aplikasi dengan tingkat permintaan yang merata namun tidak terlalu tinggi, seperti server email perusahaan atau portal informasi

karyawan, yang dapat berjalan baik meskipun tidak mengoptimalkan sumber daya secara intensif.

DAFTAR PUSTAKA

[1] "CNCF Annual Survey 2023," CNCF. Accessed: Aug. 21, 2024. [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2023/

[2] A. R. Ekaputra and A. S. Affandi, "Pemanfaatan layanan cloud computing dan docker container untuk meningkatkan kinerja aplikasi web," *J. of Information System and Application Development*, vol. 1, no. 2, pp. 138–147, Sep. 2023, doi: 10.26905/jisad.v1i2.11084.

[3] Q.-M. Nguyen, L.-A. Phan, and T. Kim, "Load-Balancing of Kubernetes-Based Edge Computing Infrastructure Using Resource Adaptive Proxy," *Sensors*, vol. 22, no. 8, p. 2869, Apr. 2022, doi: 10.3390/s22082869.

[4] I. Vasireddy, G. Ramya, and P. Kandi, "Kubernetes and Docker Load Balancing: State-of-the-Art Techniques and Challenges," *IJIREM*, vol. 10, no. 6, pp. 49–54, Dec. 2023, doi: 10.55524/ijirem.2023.10.6.7.

[5] A. A. J. Sinlae, M. Bagir, and M. H. Prayitno, "Analisis Perbandingan Algoritma Round-Robin dengan Least-Connection Terhadap Peningkatan Nilai Throughput Pada Layanan Web Server," *Jur. Ris. Kom.*, vol. 9, no. 5, p. 1584, Oct. 2022, doi: 10.30865/jurikom.v9i5.4995.

[6] N. S. ALFIANY, "Analisis Perbandingan Kinerja Ingress Controller KONG Dan ISTIO Pada Kubernetes Cluster," skripsi, Institut Teknologi Telkom Purwokerto, 2023. Accessed: Aug. 24, 2024. [Online]. Available: https://repository.ittelkom-pwt.ac.id/10017/

[7] R. D. Marcus, R. A. Saputro, and F. Y. Pamuji, "Optimasi Jaringan Routing Open Shortest Path First Dengan Menggunakan Multiprotocol Label Switching," *Bri*, vol. 5, no. 3, p. 612, Aug. 2020, doi: 10.28926/briliant.v5i3.486.

[8] S. D. Riskiono and D. Pasha, "ANALISIS METODE LOAD BALANCING DALAM MENINGKATKAN KINERJA WEBSITE E-LEARNING," *JTI*, vol. 14, no. 1, p. 22, Jan. 2020, doi: 10.33365/jti.v14i1.466.

[9] P. P. Desa and F. Dewanta, "ANALISIS LOAD BALANCING MENGGUNAKAN ALGORTIMA OPTIMASI KOLONI SEMUT DAN LEAST CONNECTION PADA JARINGAN SOFTWARE DEFINED NETWORK," Apr. 2021.

[10] M. Usman Sana and Z. Li, "Efficiency aware scheduling techniques in cloud computing: a descriptive literature review," *PeerJ Computer Science*, vol. 7, p. e509, May 2021, doi: 10.7717/peerj-cs.509.

[11] A. Solehudin, R. Mayasari, G. Garno, and A. Susilo Yuda Irawan, "Perbandingan Algoritma Round Robin dan Algoritma Least Connection pada Haproxy untuk Load Balancing Web Server," *sys*, vol. 2, no. 1, p. 21, Apr. 2020, doi: 10.35706/sys.v2i1.3634.

[12] H. Luthfi, R. Tulloh, and M. Iqbal, "Perancangan Dan Implementasi Load Balancing Menggunakan Algoritma Least Connection Dan Ip Hash Pada Kubernetes," Aug. 2023.

[13] C. Rawls and M. A. Salehi, "Load Balancer Tuning: Comparative Analysis of HAProxy Load Balancing Methods," Dec. 29, 2022, *arXiv*: arXiv:2212.14198. Accessed: Sep. 27, 2024. [Online]. Available: http://arxiv.org/abs/2212.14198

[14] I. M. Ibrahim *et al.*, "Web Server Performance Improvement Using Dynamic Load Balancing Techniques: A Review," *AJRCoS*, pp. 47–62, Jun. 2021, doi: 10.9734/ajrcos/2021/v10i130234.

[15] S. Pargaonkar, "A Comprehensive Review of Performance Testing Methodologies and Best Practices: Software Quality Engineering," *International Journal of Science and Research (IJSR)*, vol. 12, pp. 2008–2014, Nov. 2023, doi: 10.21275/SR23822111402.

[16] Sulastri, N. Kamila, and I. Rahmawati, "Efektivitas Teknik Mindfulness Untuk Mengatasi Insomania Pada Mahasiswa," *Journal of Psychology Today*, vol. 1, no. 4, Art. no. 4, Dec. 2023.

[17] A. Rahayu, E. Ernawati, and R. A. Rahim, "PERBANDINGAN HASIL BELAJAR MATEMATIKA DENGAN MENGGUNAKAN MODEL NUMBER HEAD TOGETHER (NHT) DAN THINK PAIR SHARE (TPS) BERBASIS MEDIA WHATSAPP," *Jtm*, vol. 1, no. 2, pp. 12–18, Jan. 2021, doi: 10.47435/jtm.v1i2.468.

[18] O. Pramadika and D. W. Chandra, "Provisioning Google Kubernetes Engine (GKE) Cluster dengan Menggunakan Terraform dan Jenkins pada Dua Environment," *jipi. jurnal. ilmiah. penelitian. dan. pembelajaran. informatika.*, vol. 8, no. 2, pp. 597–606, May 2023, doi: 10.29100/jipi.v8i2.3630.

[19] M. P. Hasibuan, R. Azmi, D. B. Arjuna, and S. U. Rahayu, "Analisis Pengukuran Temperatur Udara Dengan Metode Observasi," vol. 1, 2023.

[20] A. K. Chandrasekhar and D. A. S. Chandran, "COMPARATIVE ANALYSIS OF LOAD TESTING TOOLS," vol. 9, no. 6, 2021.

[21] D. Rahman, H. Amnur, and I. Rahmayuni, "Monitoring Server dengan Prometheus dan Grafana serta Notifikasi Telegram," *JITSI : Jurnal Ilmiah Teknologi Sistem Informasi*, vol. 1, no. 4, Art. no. 4, Dec. 2020.

[22] P. Fajar and Y. I. Aviani, "Hubungan Self-Efficacy dengan Penyesuaian Diri: Sebuah Studi Literatur," vol. 6, 2022.

[23] M. Tarigan and D. Frintiana Silaban, "Statistika Deskriptif," *JINTAN*, vol. 4, no. 2, pp. 187–195, Jul. 2024, doi: 10.51771/jintan.v4i2.859.

[24] F. Orcan, "Parametric or Non-parametric: Skewness to Test Normality for Mean Comparison," *International Journal of Assessment Tools in Education*, vol. 7, no. 2, pp. 255–265, Jun. 2020, doi: 10.21449/ijate.656077.

[25] D. S. Rini and F. Faisal, "Perbandingan Power of Test dari Uji Normalitas Metode Bayesian, Uji Shapiro-Wilk, Uji Cramer-von Mises, dan Uji Anderson-Darling," vol. 11, no. 2, 2015.

[26] R. Sianturi, "Uji homogenitas sebagai syarat pengujian analisis," *PSSA*, vol. 8, no. 1, pp. 386–397, Jul. 2022, doi: 10.53565/pssa.v8i1.507.

[27] M. Wassalwa, H. D. Siregar, K. Janani, and I. S. Harahap, "ANALISIS UJI HIPOTESIS PENELITIAN PERBANDINGAN MENGGUNAKAN STATISTIK PARAMETRIK," *Al Ittihadu*, vol. 3, no. 1, Art. no. 1, 2024.

[28] A. Indrasetianingsih, I. A. Haryanto, and P. A. Divaio, "Analisis Kruskal-Wallis untuk Mengetahui Kemampuan Literasi Siswa SMP Miftahurrohman Gresik Berdasarkan Asesmen Kompetensi Minimum," *ijmst*, vol. 2, no. 1, pp. 32–36, Feb. 2024, doi: 10.31004/ijmst.v2i1.286.

[29] E. Rohadi, A. Amalia, A. Prasetyo, M. F. Rahmat, A. Setiawan, and I. Siradjuddin, "Cluster implementation on mini Raspberry Pi computers using Round Robin Algorithm," *J. Phys.: Conf. Ser.*, vol. 1450, no. 1, p. 012068, Feb. 2020, doi: 10.1088/1742-6596/1450/1/012068.

[30] Y. Permana and Y. Afrianto, "Load Balancing Method Performance Analysis on Haproxy and Router OS," vol. 4, no. 36, 2020.

[31] T. Wira Harjanti, H. Setiyani, and J. Trianto, "Load Balancing Analysis Using Round-Robin and Least-Connection Algorithms for Server Service Response Time," *ATCSJ*, vol. 5, no. 2, pp. 40–49, Dec. 2022, doi: 10.33086/atcsj.v5i2.3743.

[32] B. Alankar, G. Sharma, H. Kaur, R. Valverde, and V. Chang, "Experimental Setup for Investigating the Efficient Load Balancing Algorithms on Virtual Cloud," *Sensors*, vol. 20, no. 24, p. 7342, Dec. 2020, doi: 10.3390/s20247342.

[33] L. Laukka and C. Fransson, "Enhancing the performance of mobile networks using Kubernetes: Load balancing traffic by utilizing workload estimation," 2023.