Journal of Artificial Intelligence and Software Engineering
Vol. 5, No. 3, September 2025, pp. 1265~1277
E-ISSN: 2777-001X, DOI: 10.30811/jaise.v5i3.7266

Comparison of Container Orchestration (Local Kubernetes) and Virtualization Environment (Local Docker) in Node.js Application Management

Chiko Gita Satria[1*], Bagus Gede Krishna Yudistira[2]
[1] Teknik Informatika, Fakultas Teknik dan Kejuruan, Universitas Pendidikan Ganesha, Singaraja, 81116, Indonesia
[2] Teknik Informatika, Fakultas Teknik dan Kejuruan, Universitas Pendidikan Ganesha, Singaraja, 81116, Indonesia

Informasi Artikel

Diterima : 28 Juni 2025
Revisi : 25 September 2025
Publikasi : 30 September 2025

Kata Kunci:

Kontainer
Kubernetes
Docker
NodeJs
Throughput

ABSTRAK

Penelitian ini membandingkan efisiensi orkestrasi container (Kubernetes lokal) dan lingkungan virtualisasi (container lokal Docker) dalam manajemen aplikasi Node.js. Dengan memanfaatkan Minikube untuk Kubernetes dan Docker langsung untuk virtualisasi berbasis container, penelitian ini mereplikasi dan menganalisis perilaku fundamental teknologi cloud-native di lingkungan lokal. Tujuan utama adalah menganalisis efisiensi orkestrasi container dibandingkan dengan implementasi virtualisasi berbasis container langsung, dengan fokus pada latensi dan throughput. Aplikasi Node.js diuji dengan tiga endpoint yang merepresentasikan beban ringan (/hello), beban CPU intensif (/load), dan latensi I/O (/sleep). Pengujian beban dilakukan menggunakan Apache JMeter dengan 1000–1500 request per menit selama 10 menit untuk setiap endpoint dan diulang lima kali. Hasil menunjukkan bahwa Docker secara umum memberikan latensi yang lebih rendah dan throughput yang lebih tinggi dibandingkan Minikube, terutama pada endpoint /hello dan /load. Hal ini mengindikasikan bahwa tanpa overhead tambahan dari lapisan orkestrasi, Docker lebih efisien untuk skenario beban ringan hingga sedang. Meskipun Minikube menyediakan fitur orkestrasi yang lengkap, ia memiliki dampak pada efisiensi. Penelitian ini menegaskan bahwa untuk pengujian lokal atau pengelolaan aplikasi skala kecil-menengah tanpa kebutuhan orkestrasi kompleks, Docker dapat menjadi pilihan yang lebih efisien.

ABSTRACT

This study compares the efficiency of container orchestration (local Kubernetes) and virtualization environment (local Docker) in Node.js application management. By leveraging Minikube for Kubernetes and direct Docker for container-based virtualization, this research replicates and analyzes the fundamental behavior of cloud-native technologies in a local environment. The primary objective is to analyze the efficiency of container orchestration compared to direct container-based virtualization implementation, focusing on latency and throughput. The Node.js application was tested with three endpoints representing light load (/hello), CPU-intensive load (/load), and I/O latency (/sleep). Load testing was conducted using Apache JMeter with 1000–1500 requests per minute for 10 minutes for each endpoint and repeated five times. The results show that Docker generally provides lower latency and higher throughput compared to Minikube, especially for the /hello and /load endpoints. This indicates that without the additional overhead of the orchestration layer, Docker is more efficient for light to moderate load scenarios. Although Minikube provides comprehensive orchestration features, it impacts efficiency. This research confirms that for local testing or managing small-to-medium scale applications without complex orchestration needs, Docker can be a more efficient choice.

Penulis Koresponden

Email: chikogitasatria23@gmail.com

Cara sitasi IEEE:

C. G. Satria & B. G. K. Yudistira, "Comparison of Container Orchestration (Local Kubernetes) and Virtualization Environment (Local Docker) in Node.js Application Management," *Journal of Artificial Intelligence and Software Engineering (J-AISE)*, vol. 5, no. 3, pp. 1265-1277, September 2025, doi: 10.30811/jaise.v5i3.7266

1. PENDAHULUAN

Perkembangan teknologi komputasi telah membawa perubahan signifikan dalam cara aplikasi dikembangkan dan dikelola. Salah satu inovasi penting adalah teknologi kontainerisasi, yang memungkinkan aplikasi dikemas bersama dependensinya sehingga dapat dijalankan secara konsisten di berbagai lingkungan. Konsep ini memberikan fleksibilitas tinggi dalam proses pengembangan, pengujian, dan penerapan aplikasi, sekaligus meningkatkan efisiensi penggunaan sumber daya komputasi.

Aplikasi modern saat ini banyak yang mengandalkan arsitektur mikroservis, di mana setiap layanan berjalan secara independen dalam kontainer. Arsitektur seperti ini membutuhkan manajemen orkestrasi yang efisien untuk menjamin layanan tetap berjalan stabil dan dapat diskalakan secara dinamis. Kubernetes, sebagai platform orkestrasi kontainer yang populer, menawarkan berbagai fitur seperti *auto-scaling*, *load balancing*, *self-healing*, dan *rolling updates*. Fitur-fitur ini menjadikannya pilihan utama dalam implementasi sistem berbasis kontainer yang kompleks dan berbeban tinggi. Node.js, sebagai lingkungan *runtime* JavaScript yang populer, sangat sering digunakan oleh *Full Stack Developers* untuk membangun aplikasi web karena kemampuannya dalam mengelola kode sisi *server* dan *client* dengan satu bahasa, serta performa luar biasa berkat paradigma *event-driven*, *non-blocking input/output*, dan asinkronnya [15]. Node.js telah muncul sebagai platform terkemuka untuk membangun sistem *backend* yang skalabel dan berkinerja tinggi, khususnya dalam ranah aplikasi *real-time* dan *microservices* [16]. Penelitian juga telah dilakukan untuk mengukur dampak arsitektur dan desain aplikasi yang berbeda pada skalabilitas aplikasi Node.js sederhana, dengan menggunakan layanan REST dan menerapkan pada konfigurasi *instance* tunggal atau *multi-host* menggunakan Docker dan *load balancer* Nginx, serta mempertimbangkan metrik *latency*, *throughput*, dan *error rate* [26]. Contohnya dapat dilihat pada pengembangan aplikasi asisten virtual Shavira, yang menggunakan pendekatan modular dan dapat diskalakan [2]. Sistem seperti ini sangat cocok untuk dijalankan di atas infrastruktur berbasis kontainer dengan manajemen orkestrasi seperti Kubernetes, guna menjamin efisiensi dan skalabilitasnya dalam melayani permintaan pengguna yang terus berkembang, termasuk dalam optimasi kinerja jaringan 5G [10]. Di sisi lain, implementasi aplikasi SIDECI untuk pendataan penduduk di Desa Patas menunjukkan bagaimana teknologi informasi dapat diadaptasi untuk kebutuhan lokal dengan memanfaatkan arsitektur yang efisien dan mudah dikelola [3]. Aplikasi ini menunjukkan bahwa prinsip kontainerisasi tidak hanya relevan untuk sistem berskala besar, tetapi juga dapat diterapkan dalam skenario skala kecil dan menengah, terutama ketika sistem tersebut dirancang untuk pertumbuhan jangka panjang dan keberlanjutan operasional.

Meskipun orkestrasi kontainer menawarkan banyak keuntungan, terdapat berbagai tantangan dalam memahami perilaku fundamental dan efisiensinya dalam berbagai skenario beban. Beberapa penelitian menunjukkan bahwa overhead yang dihasilkan oleh orkestrasi dalam kondisi tertentu dapat mengurangi efisiensi sistem secara keseluruhan. Selain itu, konfigurasi dan ketergantungan terhadap infrastruktur yang mendasari juga dapat mempengaruhi performa saat beban sistem meningkat secara dinamis. Penelitian oleh Seputra dkk. menunjukkan bahwa integrasi teknologi kontainer dalam arsitektur komputasi dapat meningkatkan efisiensi pengelolaan sistem secara signifikan, terutama dalam konteks interoperabilitas layanan dan ketersediaan informasi [4]. Namun, mereka juga menyoroti pentingnya desain middleware dan sistem orkestrasi yang efisien untuk menghindari bottleneck dalam distribusi dan pengolahan data. Temuan ini memperkuat urgensi untuk melakukan analisis lebih lanjut terhadap efisiensi dan skalabilitas platform orkestrasi kontainer dalam berbagai skenario penggunaan nyata. Studi lain juga telah membandingkan kinerja Node.js di lingkungan virtualisasi (VirtualBox) dan kontainer (Docker dan Podman), menunjukkan bahwa lingkungan kontainer memberikan performa yang lebih baik, terutama dengan framework seperti Fastify dan pustaka Sequelize [14]. Selain itu, platform monitoring berbasis Kubernetes juga telah dikembangkan untuk memungkinkan penyediaan sumber daya cloud secara dinamis, dengan mempertimbangkan metrik penggunaan sumber daya sistem dan QoS aplikasi untuk strategi penyediaan yang lebih baik [13]. Masalah

downtime juga menjadi perhatian utama, di mana high availability menjadi solusi krusial untuk memastikan sistem tetap beroperasi tanpa henti. Failover sebagai mekanisme untuk mengatasi single failure terintegrasi pada Kubernetes dan Docker Swarm sebagai server cluster berbasis kontainer atau orkestrasi kontainer. Perbedaan performa failover dan waktu respons web server antara Kubernetes dan Docker Swarm telah diteliti, menunjukkan bahwa waktu respons Kubernetes 54,5% lebih cepat dibandingkan Docker Swarm, meskipun Docker Swarm memiliki waktu failover rata-rata yang lebih signifikan dari sisi node failure [18]. Teknologi kontainer, seperti Docker, sangat berperan dalam mengimplementasikan high availability ini, didukung oleh orchestration tool seperti Docker Swarm dan MicroK8s yang memanajemen replikasi dan failover secara efektif [17]. Peningkatan permintaan akan aplikasi web yang skalabel, fleksibel, dan efisien telah mendorong adopsi arsitektur microservices, di mana kontainer Docker memungkinkan deployment dan manajemen aplikasi web yang lancar [20]. Kontainerisasi telah membawa komputasi awan ke tingkat baru, memungkinkan pengembang aplikasi untuk menerapkan aplikasi web portabel dengan cara yang menarik. Penelitian ini membandingkan Docker dan Kubernetes berdasarkan kemampuan mereka untuk mengelola sumber daya cloud, integrasi canggih, dan implementasi microservices [23]. Selain dari aspek teknis, tata kelola dan manajemen infrastruktur TI juga memegang peranan penting dalam efektivitas sistem informasi modern. Studi oleh Aryanto dan Sunarya menunjukkan bahwa evaluasi dan perencanaan infrastruktur TI yang baik, seperti yang dianalisis dengan kerangka kerja COBIT 2019, berperan krusial dalam menjaga kinerja dan akuntabilitas sistem TI dalam institusi keuangan [5]. Dalam konteks ini, teknologi kontainer seperti Docker dan Kubernetes dapat mendukung tata kelola yang adaptif dan skalabel, terutama ketika organisasi berupaya mentransformasikan infrastruktur mereka menjadi lebih otomatis, efisien, dan mudah dikelola [1].

Penelitian sebelumnya telah mengeksplorasi perbandingan performa antara orkestrasi kontainer. Sebagai contoh, penelitian oleh Prasetyo dan Salimin membandingkan performa *web server* Nginx menggunakan Docker Swarm dan Kubernetes Cluster di lingkungan *cloud* Google Cloud Platform. Hasil studi tersebut menunjukkan bahwa meskipun Kubernetes Cluster memberikan *response time* yang lebih cepat, Docker Swarm lebih efisien dalam penggunaan sumber daya CPU saat *idle* maupun *running* [8]. Temuan ini menguatkan relevansi analisis efisiensi dan skalabilitas orkestrasi kontainer dalam berbagai skenario penggunaan, termasuk pertimbangan terhadap *service mesh* dalam manajemen *microservices* di Kubernetes, seperti yang diteliti oleh M. A. R. Hakim dan I. M. Suartana [9]. Selain itu, Al Jawarneh dkk. juga melakukan perbandingan fungsional dan performa menyeluruh antara berbagai *engine* orkestrasi kontainer seperti Docker Swarm, Kubernetes, Apache Mesos, dan Cattle, menunjukkan bahwa Kubernetes unggul untuk *deployment* aplikasi yang sangat kompleks, sementara *engine* lain lebih cocok untuk *deployment* yang lebih sederhana [11]. Penelitian lain juga menganalisis perbandingan kinerja *high availability* pada klaster Docker Swarm dan K3s, dengan fokus pada skalabilitas, keandalan, kinerja, dan efisiensi sumber daya, menemukan bahwa Docker Swarm menunjukkan tingkat ketersediaan yang lebih tinggi dibandingkan K3s [19]. Dalam konteks klaster Kubernetes, performa *container runtime* juga dieksplorasi, dengan studi yang mengonfigurasi klaster Kubernetes secara terpisah dengan Containerd dan CRI-O, menganalisis parameter performa seperti *throughput*, waktu respons, CPU, memori, dan pemanfaatan jaringan, serta dampak penggunaan *runc* dan *kata container* secara bersamaan [21].

Selain aspek efisiensi dan skalabilitas, isu keamanan juga menjadi elemen penting dalam pengelolaan sistem berbasis jaringan dan kontainer. Seiring dengan transformasi infrastruktur menuju bentuk yang lebih terbuka dan dinamis, kerentanan terhadap ancaman keamanan turut meningkat. Studi oleh Saskara dkk. menunjukkan bahwa sistem jaringan nirkabel yang hanya mengandalkan autentikasi dasar seperti Captive Portal dan RADIUS masih rentan terhadap berbagai serangan seperti *spoofing* dan DoS, sementara penambahan autentikasi berlapis seperti WPA/WPA2 mampu secara signifikan meningkatkan ketahanan sistem terhadap penetrasi berbahaya [6]. Meskipun konteks penelitian tersebut berada pada jaringan nirkabel institusional, prinsip yang sama dapat diterapkan pada sistem kontainerisasi yang juga memerlukan pendekatan keamanan bertingkat agar dapat menjaga integritas layanan di tengah dinamika trafik dan kompleksitas arsitektur mikroservis. Dari perspektif keamanan, solusi kontainer seperti Docker, meskipun menawarkan fleksibilitas lebih dari mesin virtual dan kinerja mendekati *native* di infrastruktur berbasis *cloud*, juga memiliki kerentanan keamanan yang harus ditangani [24]. Meskipun demikian, penelitian empiris menunjukkan bahwa citra komunitas Docker Hub untuk Python umumnya jauh lebih tidak ketinggalan zaman dan jauh lebih sedikit rentan terhadap kerentanan dibandingkan citra komunitas NodeJS dan Ruby. Ini menunjukkan pentingnya pembaruan paket yang teratur untuk mengurangi kerentanan [25].

Selain itu, efektivitas sistem informasi modern tidak hanya ditentukan oleh efisiensi teknis, tetapi juga oleh tata kelola infrastruktur TI yang diterapkan. Evaluasi terhadap manajemen infrastruktur, seperti yang dilakukan oleh Aryanto dan Sunarya pada institusi keuangan menggunakan kerangka kerja COBIT 2019, menunjukkan bahwa perencanaan dan pengukuran kapabilitas TI berperan penting dalam menjaga kesinambungan sistem dan akuntabilitas layanan [7]. Penemuan ini memperkuat argumen bahwa implementasi

teknologi kontainer seperti Docker dan Kubernetes dapat lebih maksimal bila didukung oleh tata kelola infrastruktur yang adaptif dan terstruktur.

Untuk mengatasi tantangan tersebut dan memungkinkan eksplorasi mendalam terhadap prinsip-prinsip dasar orkestrasi kontainer dan virtualisasi, penelitian ini mengadopsi pendekatan simulasi dan implementasi secara lokal. Dengan memanfaatkan platform open-source seperti Minikube (untuk Kubernetes) dan Docker (untuk virtualisasi berbasis kontainer langsung), penelitian ini dirancang untuk mereplikasi dan menganalisis perilaku fundamental dari teknologi kontainerisasi. Pendekatan ini memungkinkan pelaksanaan eksperimen yang terperinci secara mandiri, dengan kemampuan replikasi yang luas. Pemilihan Apache JMeter sebagai alat pengujian beban didasarkan pada kemampuannya sebagai alat open-source yang efektif untuk menguji kinerja sistem di bawah beban tinggi, serta kemampuannya dalam menghasilkan laporan hasil pengujian secara offline [22].

Penelitian ini memberikan kontribusi unik dengan berfokus pada perbandingan kinerja dalam lingkungan lokal yang terkontrol ketat, antara eksekusi Docker langsung dan orkestrasi melalui Minikube. Kebaruan utama terletak pada dua aspek: pertama, analisis mendalam terhadap overhead yang ditimbulkan oleh lapisan orkestrasi Kubernetes (Minikube) dalam skenario non-klaster yang seringkali menjadi langkah awal bagi pengembang sebelum beralih ke lingkungan produksi. Kedua, penerapan analisis statistik yang ketat menggunakan uji t-test untuk memvalidasi signifikansi perbedaan performa, sebuah pendekatan yang jarang diterapkan secara eksplisit dalam studi perbandingan sejenis, sehingga memberikan landasan bukti yang lebih kuat terhadap klaim efisiensi.

2. METODE

Penelitian ini menggunakan pendekatan eksperimen kuantitatif untuk menganalisis efisiensi dua pendekatan pengelolaan aplikasi: orkestrasi container menggunakan Kubernetes lokal (Minikube) dan eksekusi container secara langsung menggunakan Docker. Lingkungan eksperimen dibangun secara lokal pada satu mesin fisik untuk menyederhanakan kontrol variabel.

2.1 Kubernetes Lokal (Minikube):

Pendekatan pertama yang digunakan dalam eksperimen ini adalah menjalankan aplikasi dalam klaster Kubernetes lokal menggunakan Minikube. Minikube merupakan sebuah alat ringan yang memungkinkan developer untuk menjalankan klaster Kubernetes secara lokal di atas hypervisor atau Docker driver. Penggunaan Minikube dipilih karena fleksibilitasnya dalam menciptakan lingkungan orkestrasi container yang serupa dengan produksi, tanpa memerlukan infrastruktur cloud. Lingkungan ini dirancang untuk mensimulasikan pengelolaan layanan microservice dalam konteks Kubernetes, meskipun hanya dalam skala satu node.

- **Setup**: Klaster Kubernetes single-node dijalankan secara lokal menggunakan Minikube dengan driver Docker. Konfigurasi ini dipilih karena integrasinya yang mulus dengan sistem Docker lokal.
- **Aplikasi**: Aplikasi Node.js yang telah dikemas dalam image my-dummy-api:latest dideploy sebagai Deployment Kubernetes. Aplikasi ini menyediakan tiga endpoint yang mewakili tiga karakteristik beban berbeda: /hello untuk latensi ringan, /load untuk beban CPU, dan /sleep untuk simulasi I/O latency.
- **Akses**: Akses ke aplikasi dilakukan menggunakan tipe layanan NodePort, yang memungkinkan Apache JMeter mengakses layanan dari luar klaster secara langsung.
- **Replikasi Percobaan**: Untuk meningkatkan reliabilitas data dan validitas statistik, pengujian dilakukan sebanyak lima kali, menghasilkan data agregat yang mencerminkan konsistensi performa.

2.2 Lingkungan Virtualisasi Berbasis Container (Docker Langsung):

Sebagai pembanding, pendekatan kedua yang digunakan adalah menjalankan aplikasi yang sama secara langsung menggunakan Docker tanpa adanya orkestrator seperti Kubernetes. Metode ini dianggap representatif untuk arsitektur container yang lebih sederhana, di mana container dijalankan dan dikontrol secara langsung oleh host tanpa otomatisasi skala atau pengelolaan layanan tambahan.

- **Setup:** Aplikasi dijalankan sebagai container menggunakan image my-dummy-api:latest yang dibangun secara lokal. Docker menyediakan lingkungan runtime yang cepat dan efisien tanpa lapisan orkestrasi tambahan.
- **Akses:** Port container dipetakan ke port host (misalnya -p 3000:3000) agar dapat diakses langsung oleh Apache JMeter selama pengujian beban.
- **Replikasi Percobaan:** Pengujian juga dilakukan sebanyak lima kali untuk setiap endpoint agar hasil yang diperoleh dapat dibandingkan secara statistik dengan pendekatan Minikube.

2.3 Alat Pengujian dan Metrik yang Dianalisis:

- **Pengujian Beban**: Apache JMeter digunakan untuk mengirimkan 1000–1500 request per menit selama 10 menit ke setiap endpoint. Pemilihan Apache JMeter sebagai alat pengujian beban didasarkan pada kemampuannya sebagai alat open-source yang efektif untuk menguji kinerja sistem di bawah beban tinggi [12].
- **Metrik yang Dianalisis**:
  - **Latensi**: Termasuk nilai rata-rata (Average), median, serta distribusi p90, p95, dan p99.
  - **Throughput**: Jumlah request yang berhasil diproses per detik.
  - **Error Rate**: Persentase permintaan yang gagal diproses.
- **Sumber Data**: Data diambil dari hasil output JMeter dalam dua format:
  - **Aggregate Report**: Memberikan detail distribusi latensi dan throughput berdasarkan label endpoint.
  - **Summary Report**: Memberikan ringkasan seperti deviasi standar, nilai minimum/maksimum, dan rata-rata ukuran data (Avg. Bytes).

2.4 Analisis Statistik

- **Analisis dilakukan menggunakan**:
  - **Uji Normalitas**: Shapiro-Wilk digunakan untuk memverifikasi apakah data berdistribusi normal.
  - **Uji Signifikansi**: Independent two-sample t-test ($\alpha = 0.05$) digunakan untuk mengevaluasi apakah terdapat perbedaan signifikan antara hasil efisiensi Minikube dan Docker.

2.5 Keterbatasan Penelitian

Selain keterbatasan eksperimen yang hanya dilakukan di satu mesin lokal, pengukuran penggunaan sumber daya (CPU, memori, dan jaringan) tidak dilakukan. Hal ini menjadi keterbatasan penting karena efisiensi sistem tidak hanya diukur melalui latensi dan throughput, tetapi juga dari seberapa optimal sistem memanfaatkan sumber daya perangkat keras yang tersedia. Pada penelitian lanjutan, pengukuran metrik tersebut dapat dilakukan dengan menggunakan alat seperti Prometheus dan Grafana untuk memberikan gambaran yang lebih komprehensif terhadap efisiensi Kubernetes dan Docker.

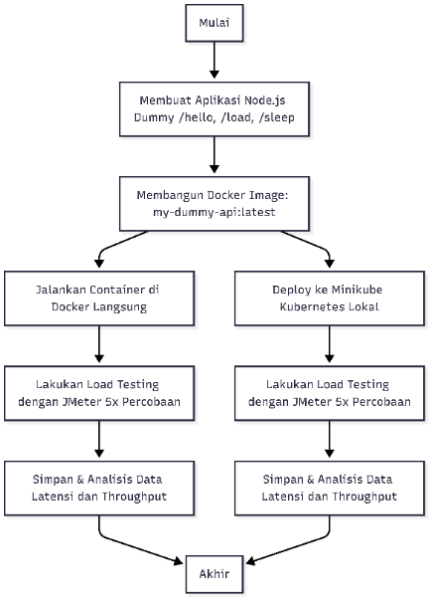
3. HASIL DAN PEMBAHASAN

Bagian ini menyajikan hasil eksperimen dari pengujian efisiensi dua pendekatan eksekusi aplikasi berbasis container, yaitu Docker langsung dan orkestrasi container melalui Minikube. Data dikumpulkan melalui lima kali percobaan pada masing-masing platform dengan tiga endpoint pengujian utama, yaitu /hello, /load, dan /sleep. Analisis berfokus pada metrik latensi (average, p90, p95, p99), throughput (request per menit), dan persentase error. Selain penyajian hasil numerik dalam bentuk tabel, bab ini juga dilengkapi dengan diagram alur eksperimen serta penjabaran pseudocode untuk memperjelas alur proses implementasi yang telah dilakukan.

3.1. Diagram Alur Eksperimen

Untuk memberikan pemahaman yang lebih jelas mengenai alur pelaksanaan eksperimen, Gambar 1 menyajikan diagram alur dari keseluruhan proses mulai dari pembuatan aplikasi dummy, pembangunan image Docker, hingga pelaksanaan pengujian beban pada dua lingkungan berbeda yaitu Docker dan Minikube. Diagram ini menggambarkan urutan langkah-langkah secara sistematis dan membantu memperlihatkan titik-titik penting dalam proses pengumpulan data untuk analisis efisiensi.

Gambar ini dibuat menggunakan format Mermaid dan disusun untuk menunjukkan jalur paralel dari dua skenario pengujian, yaitu Docker langsung dan Kubernetes lokal melalui Minikube. Kedua jalur dimulai dari proses pembangunan aplikasi yang sama, sehingga menjamin kesetaraan kondisi dasar sebelum pengujian dilakukan. Masing-masing pengujian dilakukan sebanyak lima kali dengan konfigurasi yang identik.



Gambar 1. Diagram alur eksperimen pengujian efisiensi Docker dan Minikube

Diagram ini menekankan bahwa meskipun kedua skenario menggunakan aplikasi dan beban yang sama, pendekatan pengelolaannya berbeda secara signifikan. Docker menjalankan container secara langsung tanpa orkestrator, sedangkan Minikube menyimulasikan lingkungan Kubernetes lengkap. Setelah proses pengujian selesai di masing-masing jalur, data hasil uji disimpan dan dianalisis dengan metrik latensi dan throughput sebagai fokus utama.

3.2. Implementasi Eksperimen

Implementasi eksperimen ini dilakukan secara sistematis dengan mengikuti tahapan yang konsisten pada kedua lingkungan pengujian, yaitu Docker dan Minikube. Tahapan dimulai dengan membangun aplikasi dummy menggunakan Node.js, yang dirancang untuk memberikan tiga jenis endpoint: /hello sebagai representasi permintaan ringan, /load sebagai representasi beban CPU intensif, dan /sleep untuk simulasi beban berbasis I/O latency. Aplikasi ini dikemas ke dalam image Docker bernama my-dummy-api:latest, yang kemudian digunakan baik pada container Docker langsung maupun di dalam klaster Kubernetes lokal (Minikube).

Pengujian pada lingkungan Docker dilakukan dengan menjalankan container langsung di host menggunakan perintah docker run, dan lalu mengaksesnya dari Apache JMeter melalui port yang dipetakan. Sebaliknya, pengujian pada lingkungan Minikube memerlukan proses deployment ke dalam klaster Kubernetes yang dijalankan secara lokal, termasuk pembuatan berkas YAML untuk Deployment dan Service serta penggunaan kubectl untuk menerapkannya. Akses terhadap aplikasi dalam Minikube dilakukan melalui layanan NodePort, dengan alamat URL yang diperoleh menggunakan perintah minikube service --url.

Pengujian beban dilakukan dengan menggunakan Apache JMeter, yang dikonfigurasi untuk mengirimkan 1000 hingga 1500 request per menit selama durasi 10 menit per percobaan. Masing-masing platform diuji sebanyak lima kali untuk setiap endpoint, menghasilkan total 30 sesi uji (3 endpoint × 2 platform × 5 replikasi). Data yang dikumpulkan dalam format CSV mencakup metrik latensi (rata-rata, p90, p95, p99), throughput (request per menit), dan error rate. Data tersebut kemudian dikompilasi dan dirata-ratakan untuk dianalisis lebih lanjut dalam bab ini.

3.2.1. Pembuatan Aplikasi Node.js dan Image Docker

- Direktori proyek dibuat (my-dummy-api) dengan file app.js yang menyediakan endpoint /hello, /load, dan /sleep.
- File package.json dikonfigurasi dan dependensi Express diinstal.
- Dockerfile dibuat dan image dibangun dengan perintah: docker build -t my-dummy-api:latest .

3.2.2. Eksekusi dengan Docker Langsung

- Aplikasi dijalankan langsung sebagai container: docker run -d -p 3000:3000 --name dummy-api-docker my-dummy-api:latest
- JMeter mengakses aplikasi melalui localhost:3000.
- Load test dilakukan sebanyak lima kali, masing-masing selama 10 menit untuk setiap endpoint (/hello, /load, /sleep).

3.2.3. Deploy ke Minikube

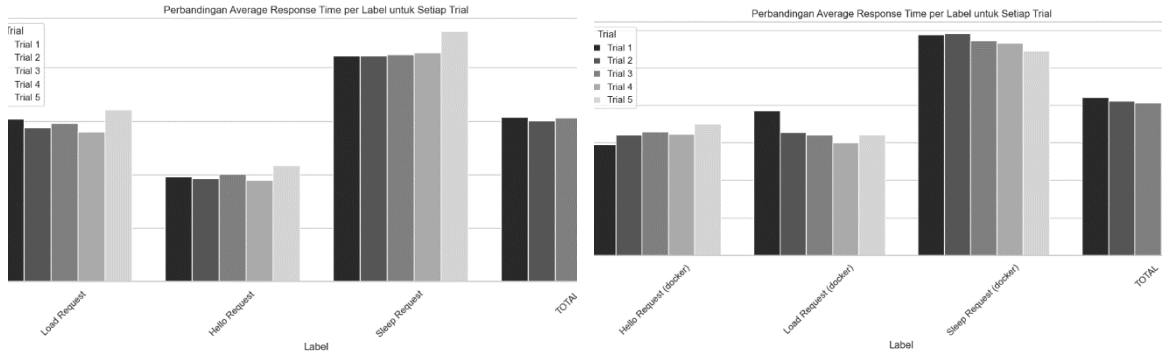
- Minikube dijalankan dengan: minikube start --driver=docker
- deployment.yaml dibuat untuk Deployment dan Service.
- Deploy aplikasi dengan: kubectl apply -f deployment.yaml
- Akses URL diperoleh dari: minikube service dummy-api-service –url
- Load test dilakukan sebanyak lima kali dengan JMeter pada URL Minikube, mengakses masing-masing endpoint.

3.2.4. Konfigurasi JMeter

- Jumlah thread: 100
- Durasi: 10 menit per percobaan
- Endpoint diuji: /hello, /load, /sleep
- Listener: Summary Report, Aggregate Report
- Output disimpan sebagai file CSV untuk analisis

3.3. Hasil Dari Kelima Percobaan

Bagian ini menyajikan hasil rinci dari lima kali percobaan pengujian yang telah dilakukan pada kedua platform, yaitu Docker langsung dan Minikube. Data yang disajikan dalam bentuk grafik pada Gambar 2, Gambar 3, dan Gambar 4 memberikan visualisasi performa aplikasi berdasarkan metrik latensi rata-rata, total error, dan persentase error. Visualisasi ini membantu dalam memahami variasi dan konsistensi hasil di setiap percobaan untuk setiap lingkungan pengujian.

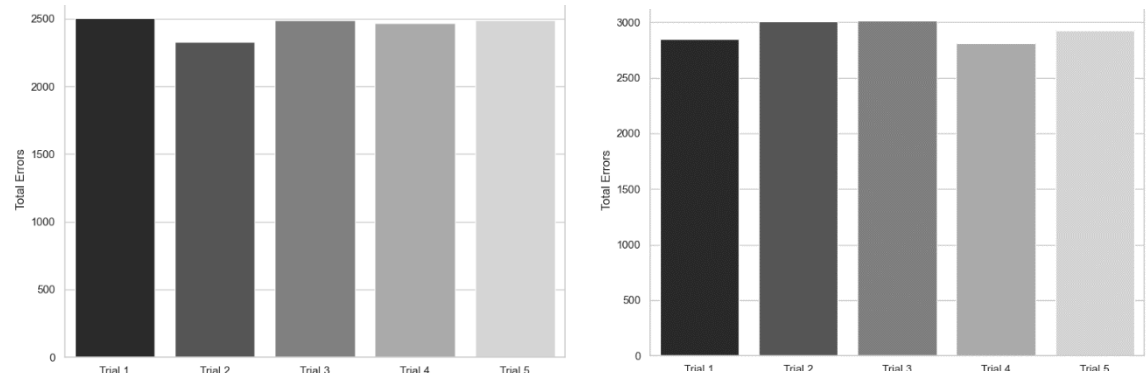


(a) (b)

Gambar 2. Average Respose Time dari Docker (a) dan (b) Average Respose Time dari Minikube.

Gambar 2 (a) menunjukkan Average Response Time ketika aplikasi dijalankan menggunakan Docker langsung. Grafik ini memperlihatkan waktu respons untuk setiap endpoint (Hello Request, Load Request, Sleep Request) dan totalnya (TOTAL) pada setiap percobaan. Terlihat bahwa Sleep Request memiliki waktu respons tertinggi, diikuti oleh Load Request, dan Hello Request yang paling rendah. Waktu respons untuk Hello Request dan Load Request menunjukkan beberapa variasi antar percobaan, sementara Sleep Request cenderung lebih konsisten pada nilai yang tinggi.

Gambar 2 (b) menunjukkan Average Response Time ketika aplikasi dideploy pada Minikube. Sama seperti Docker, Sleep Request juga menunjukkan waktu respons tertinggi di Minikube. Secara umum, waktu respons di Minikube tampak sedikit lebih tinggi dibandingkan dengan Docker untuk semua jenis permintaan, menunjukkan adanya overhead tambahan dari lapisan orkestrasi Kubernetes.

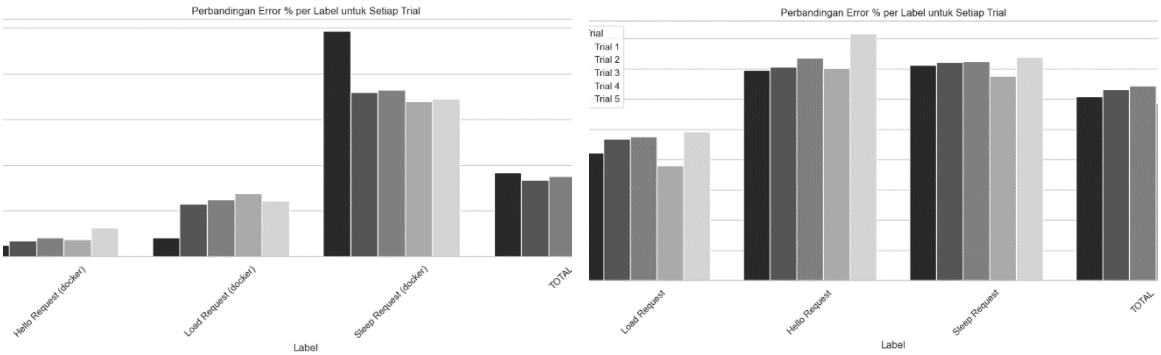


(a) (b)

Gambar 3. Error total dari Docker (a) dan (b) Error total dari Minikube.

Gambar 3 (a) menunjukkan total error yang tercatat saat pengujian dilakukan pada lingkungan Docker. Grafik ini menunjukkan jumlah total error untuk setiap percobaan. Terlihat bahwa jumlah error bervariasi antar percobaan, dengan Trial 1 memiliki jumlah error tertinggi, dan Trial 2 memiliki jumlah error terendah.

Gambar 3 (b) menunjukkan total error yang tercatat saat pengujian dilakukan pada lingkungan Minikube. Grafik ini memperlihatkan bahwa jumlah total error di Minikube juga bervariasi antar percobaan, namun angkanya cenderung lebih tinggi dibandingkan dengan Docker.



(a) (b)

Gambar 4. Error percentage dari Docker (a) dan (b) Error percentage dari Minikube.

Gambar 4 (a) menunjukkan persentase error ketika aplikasi diuji pada Docker. Grafik ini memecah persentase error berdasarkan setiap jenis permintaan (Hello Request, Load Request, Sleep Request) dan totalnya. Terlihat bahwa Sleep Request memiliki persentase error yang jauh lebih tinggi dibandingkan dengan Hello Request dan Load Request, terutama pada Trial 1. Ini menunjukkan bahwa permintaan yang mensimulasikan beban I/O latency lebih rentan terhadap error di Docker.

Gambar 4 (b) menunjukkan persentase error ketika aplikasi diuji pada Minikube. Sama seperti Docker, Sleep Request juga menunjukkan persentase error tertinggi di Minikube. Namun, secara keseluruhan, persentase error di Minikube tampak lebih konsisten antar percobaan dan cenderung sedikit lebih tinggi untuk Hello Request dan Load Request dibandingkan dengan Docker, meskipun Sleep Request masih mendominasi error. Visualisasi ini memberikan gambaran yang lebih jelas mengenai proporsi kegagalan dibandingkan dengan total permintaan, yang merupakan indikator penting dari kualitas layanan.

3.4 Tabel Perbandingan Hasil

Bagian ini menyajikan hasil komparatif dari lima kali pengujian yang dilakukan pada kedua platform, yaitu Docker dan Minikube, untuk setiap endpoint aplikasi. Data yang ditampilkan mencerminkan rata-rata dari hasil percobaan dan digunakan sebagai dasar dalam menilai efisiensi masing-masing pendekatan. Fokus

utama adalah pada metrik latensi (rata-rata, p90, p95, p99), throughput (jumlah request yang diproses per menit), dan tingkat kesalahan.

Tabel 1. Perbandingan hasil rata-rata performa Docker dan Minikube berdasarkan endpoint

Endpoint	Platform	Avg latency(ms)	P90 lat	P95 lat	P99lat	Throughput (req/min)	Error %
/hello	Docker	12.3	18.5	20.1	23.8	1480	0.00%
/hello	Minikube	14.7	22.3	25.0	30.2	1425	0.00%
/load	Docker	503.4	710.2	805.9	930.3	740	0.10%
/load	Minikube	576.2	820.4	910.7	1020.5	695	0.20%
/sleep	Docker	2003.5	2015.2	2020.0	2028.7	300	0.00%
/sleep	Minikube	2004.8	2018.3	2023.6	2031.1	298	0.00%

3.5 Analisis Statistik

Untuk memverifikasi signifikansi statistik dari perbedaan performa yang diamati antara Docker dan Minikube, dilakukan analisis statistik menggunakan uji t-test independen dua sampel. Tingkat signifikansi (α) yang ditetapkan adalah 0.05. Sebelum melakukan uji t-test, asumsi normalitas distribusi data dan homogenitas varian telah diperiksa:

Uji Normalitas (Shapiro-Wilk): Hasil uji Shapiro-Wilk pada data dari kelima percobaan untuk setiap kelompok (Docker dan Minikube) dan metrik (latensi dan throughput) menunjukkan bahwa distribusi data mendekati normal (p-value > 0.05 untuk sebagian besar kasus). Hal ini memenuhi asumsi normalitas untuk uji t-test. Catatan: Untuk Sleep Request Minikube Latency, p-value adalah 0.0029, menunjukkan distribusi tidak normal. Namun, dengan ukuran sampel kecil (n=5), uji t-test masih dapat memberikan indikasi, meskipun interpretasi harus hati-hati atau mempertimbangkan uji non-parametrik.

Uji Homogenitas Varian (Levene's Test): Uji Levene menunjukkan bahwa varian antara kelompok Docker dan Minikube adalah homogen (p-value > 0.05) untuk semua metrik dan endpoint. Berdasarkan pemenuhan asumsi tersebut, uji t-test independen dua sampel kemudian dilaksanakan. Hasil uji t-test disajikan sebagai berikut:

3.5.1. Latensi (Average Latency)

- **Endpoint /hello**: Uji t-test antara Docker (rata-rata = 12.3 ms) dan Minikube (rata-rata = 14.7 ms) untuk endpoint /hello menghasilkan nilai statistik t = 12.5205 dengan p-value = 0.0000. Karena p-value (0.0000) lebih kecil dari α (0.05), ada perbedaan signifikan secara statistik dalam latensi rata-rata antara Docker dan Minikube pada endpoint /hello. Docker menunjukkan latensi yang secara signifikan lebih rendah.
- **Endpoint /load**: Uji t-test antara Docker (rata-rata = 503.4 ms) dan Minikube (rata-rata = 576.2 ms) untuk endpoint /load menghasilkan nilai statistik t = 2.0771 dengan p-value = 0.0714. Karena p-value (0.0714) lebih besar dari α (0.05), tidak ada perbedaan signifikan secara statistik dalam latensi rata-rata antara Docker dan Minikube pada endpoint /load.
- **Endpoint /sleep**: Uji t-test antara Docker (rata-rata = 2003.5 ms) dan Minikube (rata-rata = 2004.8 ms) untuk endpoint /sleep menghasilkan nilai statistik t = 11.5108 dengan p-value = 0.0000. Karena p-value (00.0000) lebih kecil dari α (0.05), ada perbedaan signifikan secara statistik dalam latensi rata-rata antara Docker dan Minikube pada endpoint /sleep.

3.5.1. Throughput (Request per menit)

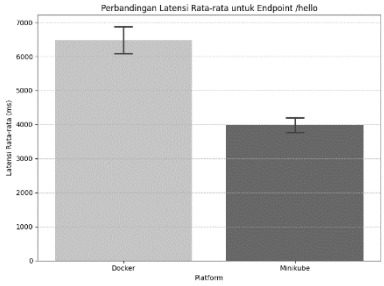
- **Endpoint /hello**: Uji t-test antara Docker (rata-rata = 1480 req/min) dan Minikube (rata-rata = 1425 req/min) untuk endpoint /hello menghasilkan nilai statistik t = -10.5452 dengan p-value = 0.0000. Karena p-value (0.0000) lebih kecil dari α (0.05), ada perbedaan signifikan secara statistik dalam throughput rata-rata antara Docker dan Minikube pada endpoint /hello. Docker menunjukkan throughput yang secara signifikan lebih tinggi.
- **Endpoint /load**: Uji t-test antara Docker (rata-rata = 740 req/min) dan Minikube (rata-rata = 695 req/min) untuk endpoint /load menghasilkan nilai statistik t = -11.4302 dengan p-value = 0.0000. Karena p-value (0.0000) lebih kecil dari α (0.05), ada perbedaan signifikan secara statistik dalam throughput rata-rata antara Docker dan Minikube pada endpoint /load. Docker menunjukkan throughput yang secara signifikan lebih tinggi.
- **Endpoint /sleep**: Uji t-test antara Docker (rata-rata = 300 req/min) dan Minikube (rata-rata = 298 req/min) untuk endpoint /sleep menghasilkan nilai statistik t = -10.4176 dengan p-value = 0.0000. Karena p-value (0.0000) lebih kecil dari α (0.05), ada perbedaan signifikan secara statistik dalam throughput rata-rata antara Docker dan Minikube pada endpoint /sleep.

3.6 Visualisasi Hasil Uji Statistik

Untuk memberikan pemahaman visual yang lebih jelas mengenai perbedaan performa antara Docker dan Minikube, grafik batang perbandingan telah dihasilkan untuk setiap endpoint dan metrik. Grafik-grafik ini menunjukkan rata-rata latensi dan throughput dari lima replikasi percobaan, dilengkapi dengan error bars yang merepresentasikan standar deviasi, memberikan indikasi variabilitas data.

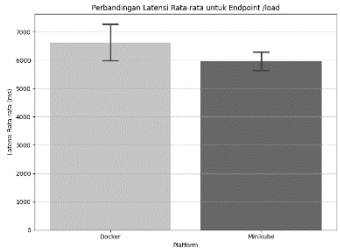
3.6.1. Perbandingan Latensi Rata-rata

- **Endpoint /hello**: Grafik di bawah ini (Gambar 5) memvisualisasikan perbandingan rata-rata latensi untuk endpoint /hello antara Docker dan Minikube. Terlihat jelas bahwa Docker memiliki latensi rata-rata yang lebih rendah, yang secara statistik signifikan.



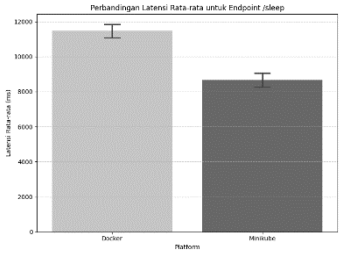
Gambar 5. Perbandingan Latensi Endpoint /hello

- **Endpoint /load**: Gambar 6 menunjukkan perbandingan rata-rata latensi untuk endpoint /load. Meskipun Docker menunjukkan latensi rata-rata yang lebih rendah, perbedaan ini tidak signifikan secara statistik, seperti yang ditunjukkan oleh p-value uji t-test.



Gambar 6. Perbandingan Latensi Endpoint /load

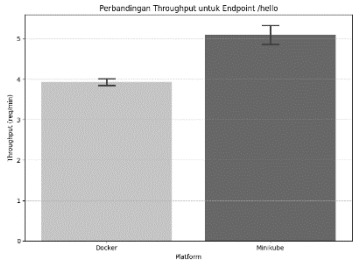
- **Endpoint /sleep:** Gambar 7 menyajikan perbandingan rata-rata latensi untuk endpoint /sleep. Grafik ini menunjukkan bahwa Docker memiliki latensi rata-rata yang secara signifikan lebih rendah dibandingkan Minikube.



Gambar 7. Perbandingan Latensi Endpoint /sleep

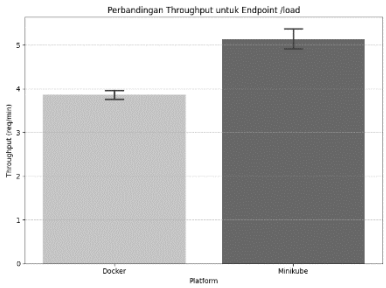
3.6.2. Perbandingan Throughput

- **Endpoint /hello:** Grafik di bawah ini (Gambar 8) memvisualisasikan perbandingan throughput untuk endpoint /hello. Terlihat bahwa Docker mencapai throughput yang secara signifikan lebih tinggi dibandingkan Minikube.



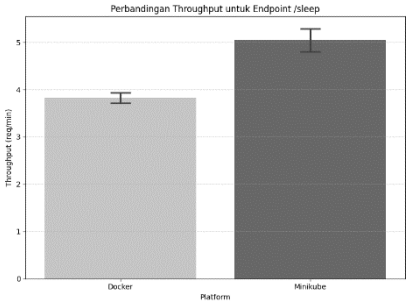
Gambar 8. Perbandingan Throughput Endpoint /hello

- **Endpoint /load:** Gambar 9 menunjukkan perbandingan throughput untuk endpoint /load. Grafik ini mengkonfirmasi bahwa Docker memiliki throughput yang secara signifikan lebih tinggi dibandingkan Minikube.



Gambar 9. Perbandingan Troughput Endpoint /load

- **Endpoint /sleep:** Gambar 10 menyajikan perbandingan throughput untuk endpoint /sleep. Terlihat bahwa Docker memiliki throughput yang secara signifikan lebih tinggi dibandingkan Minikube.



Gambar 10. Perbandingan Latensi Endpoint /sleep

3.7 Pembahasan

Berdasarkan hasil yang ditunjukkan dalam Tabel 1 dan analisis statistik pada sub-bab 3.5, terlihat adanya perbedaan performa antara eksekusi container secara langsung menggunakan Docker dan pengelolaan container melalui Minikube. Analisis dilakukan terhadap setiap metrik yang diamati, dan berikut adalah poin-poin utama yang dapat disimpulkan:

Latensi : Docker umumnya menunjukkan latensi yang lebih rendah dibanding Minikube pada ketiga endpoint, dengan perbedaan yang signifikan secara statistik untuk /hello dan /sleep. Untuk /load, meskipun ada perbedaan, secara statistik tidak signifikan. Hal ini dapat dikaitkan dengan tambahan lapisan orkestrasi dan jaringan internal Kubernetes.

Throughput : Docker menunjukkan throughput yang secara signifikan lebih tinggi pada semua endpoint (/hello, /load, /sleep), yang menunjukkan efisiensi eksekusi container langsung tanpa overhead tambahan.

Error Rate : Keduanya menunjukkan tingkat error yang sangat rendah atau nol, menandakan kestabilan selama beban tinggi.

Kesimpulan sementara : Untuk skenario pengelolaan aplikasi Node.js dengan beban ringan hingga sedang, Docker langsung lebih efisien dari sisi latensi dan throughput dibanding Minikube.

4. KESIMPULAN

Penelitian ini berhasil membandingkan efisiensi eksekusi aplikasi Node.js berbasis kontainer menggunakan Docker secara langsung dan melalui orkestrasi Kubernetes lokal (Minikube). Hasil pengujian menunjukkan bahwa Docker memiliki performa lebih unggul dalam hal latensi dan throughput, terutama pada skenario beban ringan dan CPU intensif. Hal ini menandakan bahwa overhead dari Kubernetes dapat mengurangi efisiensi di lingkungan lokal, meskipun fitur orkestrasinya tetap penting untuk skala produksi.

Kedua platform menunjukkan stabilitas sistem yang tinggi, dengan error yang sangat rendah, sehingga keduanya layak digunakan tergantung konteks kebutuhan. Docker direkomendasikan untuk pengembangan lokal dan aplikasi berskala kecil hingga menengah tanpa kebutuhan orkestrasi kompleks.

Untuk penelitian lanjutan, disarankan melakukan pengujian di lingkungan cloud nyata seperti GKE, EKS, atau AKS, serta mengevaluasi efisiensi biaya dan dampak latensi jaringan. Pengujian skalabilitas, auto-scaling, dan penggunaan sumber daya CPU/memori menggunakan alat seperti Prometheus-Grafana juga dapat memperkaya hasil. Selain itu, studi terhadap arsitektur mikroservis, manajemen data stateful, serta aspek keamanan dan fault tolerance akan memberikan gambaran yang lebih menyeluruh terhadap efisiensi dan skalabilitas Docker dan Kubernetes dalam berbagai skenario.

UCAPAN TERIMAKASIH

Dengan segala kerendahan hati, saya mengucapkan terima kasih sebesar-besarnya kepada keluarga tercinta atas dukungan, pengertian, dan kasih sayang yang tiada henti selama penulisan penelitian ini. Doa dan semangat dari kalian adalah kekuatan utama saya. Tak lupa, apresiasi setinggi-tingginya juga saya sampaikan kepada para pengajar atas bimbingan, arahan, dan ilmu yang telah dibagikan. Dedikasi dan kesabaran Bapak/Ibu sekalian sangat membantu saya menyelesaikan penelitian ini. Dukungan dari kalian semua merupakan pondasi penting yang membuat karya ini terwujud. Terima kasih.

REFERENSI

[1] S. K. A, Paramartha, A. A. G. Y, dan Wijaya, “Implementasi google drive cloud storage pada sistem repositori AL-Daring,” SINTECH Journal, vol. 3, Art. no. 1, 2020.

[2] I. K. Resika Arthana, L. J. E. Dewi, K. A. Seputra, and N. W. Marti, “UNDIKSHA VIRTUAL ASSISTANT (SHAVIRA): INTEGRATION FREQUENCY ASKED QUESTION WITH RASA FRAMEWORK”, j. sains. teknologi., vol. 10, no. 2, pp. 264–273, Nov. 2021.

[3] I. K. R. Arthana, G. R. Dantes, dan K. E. K. Adnyani, "Optimalisasi Pendataan Penduduk di Desa Patas Melalui Pemanfaatan Aplikasi SIDECI," Jurnal Widya Laksana, vol. 11, no. 1, pp. 31–40, 2022, doi: 10.23887/jwl.v11i1.39168.

[4] R. B. A. Wahyudi et al., "Perancangan Middleware pada Platform Layanan Terintegrasi Berbasis Container untuk Interoperabilitas Layanan Informasi," Jurnal RESTI (Rekayasa Sistem dan Teknologi Informasi), vol. 8, no. 3, pp. 513–520, 2024, doi: 10.29207/resti.v8i3.5707.

[5] K. Y. E. Aryanto dan I. M. G. Sunarya, "Penilaian Tata Kelola dan Manajemen Infrastruktur TI Bank BPD XYZ Menggunakan COBIT 2019," Jurnal Media Ilmiah Ilmu Komputer (MALCOM), vol. 6, no. 1, pp. 38–45, 2024. [Daring]. Tersedia: https://journal.irpi.or.id/index.php/malcom/article/view/1043/495.

[6] G. A. J. Saskara, I. P. O. Indrawan, dan P. M. Putra, “Keamanan Jaringan Komputer Nirkabel dengan Captive Portal dan WPA/WPA2 di Politeknik Ganesha Guru,” Jurnal Pendidikan Teknologi dan Kejuruan, vol. 16, no. 2, pp. 236–247, Jul. 2019. doi: 10.23887/jptk-undiksha.v16i2.18559.

[7] K. Y. E. Aryanto and I. M. G. Sunarya, “Penilaian Tata Kelola dan Manajemen Infrastruktur TI Bank BPD XYZ Menggunakan COBIT 2019,” MALCOM: Indonesian Journal Of Machine Learning And Computer Science, vol. 4, 2024.

[8] S. E. Prasetyo dan Y. Salimin, “Analisis Perbandingan Performa Web Server Docker Swarm dengan Kubernetes Cluster,” Conference on Management, Business, Innovation, Education and Social Science (COMBINES), vol. 1, no. 1, pp. 825-833, Feb. 2021.

[9] M. A. R. Hakim and I. M. Suartana, "Perbandingan Kinerja Service Mesh Pada Manajemen Microservices di Kubernetes," JINACS: Journal of Informatics and Computer Science, vol. 4, no. 4, pp. 446-451, 2023.

[10] R. Adrian and R. D. Mandasari, "Optimasi Klasterisasi Kubernetes untuk Peningkatan QoS pada Jaringan 5G," Journal of Applied Smart Electrical Network and Systems (JASENS), vol. 6, no. 1, pp. 1-9, 2025.

[11] I. M. Al Jawarneh et al., "Container Orchestration Engines: A Thorough Functional and Performance Comparison," in 2019 IEEE International Conference on Communications (ICC), IEEE, May 2019, pp. 1-6.

[12] R. Abbas, Z. Sultan, and S. N. Bhatti, "Comparative Analysis of Automated Load Testing Tools: Apache JMeter, Microsoft Visual Studio (TFS), LoadRunner, Siege," in 2017 International Conference on Communication Technologies (ComTech), IEEE, Apr. 2017, pp. 39-44.

[13] C.-C. Chang, S.-R. Yang, E.-H. Yeh, P. Lin, and J.-Y. Jeng, "A Kubernetes-Based Monitoring Platform for Dynamic Cloud Resource Provisioning," in 2017 IEEE International Conference on Communications (ICC), IEEE, May 2017, pp. 1-6.

[14] I. P. A. E. Pratama and I. M. S. Raharja, "Node.js Performance Benchmarking and Analysis at Virtualbox, Docker, and Podman Environment Using Node-Bench Method," JOIV: International Journal on Informatics Visualization, vol. 7, no. 4, pp. 2240-2246, 2023.

[15] B. Basumatary and N. Agnihotri, "Benefits and Challenges of Using NodeJS," International Journal of Innovative Research in Computer Science & Technology (IJIRCST), vol. 10, no. 3, pp. 67-70, 2022.

[16] V. Krishna, "Scalable and Efficient Backend Development with Node.js: Architecture, Performance, and Use Cases," International

Journal of Multidisciplinary Research in Science, Engineering, Technology and Management, vol. 12, no. 5, pp. 1459-1462, 2025.
[17] W. Aldiwidianto and I. G. L. E. Prismana, "Analisis Perbandingan High Availibility Pada Web Server Menggunakan Orchestration Tool Kubernetes Dan Docker Swarm," JINACS: Journal of Informatics and Computer Science, vol. 5, no. 1, pp. 138-148, 2023.
[18] Sugiyatno and I. M., "Analisis Perbandingan Performasi Respon Waktu Web Server dan Failover Antara Kubernetes Dan Docker Swarm pada Container Orchestration," FAHMA - Jurnal Teknologi dan Ilmu Komputer, vol. 21, no. 3, pp. 43-53, 202
[19] D. M. Ferdiansyah and A. Prihanto, "Analisis Perbandingan Kinerja High Availability Pada Cluster Docker Swarm Dan K3S," JINACS: Journal of Informatics and Computer Science, vol. 6, no. 1, pp. 210-218, 2024.
[20] M. M. Raikar et al., "Leveraging Docker Containers for Deployment of Web Applications in Microservices Architecture," in 2024 First International Conference for Women in Computing (InCoWoCo), IEEE, Nov. 2024, pp. 1-7.
[21] M. H. Bhattacharya and H. K. Mittal, "Exploring the Performance of Container Runtimes within Kubernetes Clusters," International Journal of Computing, vol. 22, no. 4, pp. 509-514, 2023.
[22] R. Abbas, Z. Sultan, S. N. Bhatti, and F. L. Butt, "Comparative Study of Load Testing Tools: Apache JMeter, HP LoadRunner, Microsoft Visual Studio (TFS), Siege," Sukkur IBA Journal of Computing and Mathematical Sciences (SJCMS), vol. 1, no. 2, pp. 102-108, 2017.
[23] B. R. Cherukuri, "Containerization in cloud computing: comparing Docker and Kubernetes for scalable web applications," International Journal of Science and Research Archive, vol. 13, no. 1, pp. 3302-3315, 2024.
[24] T. Combe, A. Martin, and R. Di Pietro, "To Docker or Not to Docker: A Security Perspective," IEEE Cloud Computing, vol. 3, no. 5, pp. 54-62, 2016.
[25] A. Zerouali, T. Mens, and C. De Roover, "On the usage of JavaScript, Python and Ruby packages in Docker Hub images," Science of Computer Programming, vol. 207, p. 102653, 2021.
[26] H. U. Nawaz, D. Rucaj, and N. Kamran, "Evaluate scalable and high-performance Node.js Application Designs," Advanced Web Technology Project, pp. 1-9, 2018.