Jurnal Ilmiah Sistem Informasi dan Ilmu Komputer
Volume. 6 Nomor. 2 Juli 2026
p-ISSN: 2827-8135; e-ISSN: 2827-7953; Hal 68-88
DOI: https://doi.org/10.55606/juisik.v6i2.2397
Tersedia: https://journal.sinov.id/index.php/juisik

Mitigasi Bottleneck Tingkat Koneksi pada Kubernetes HPA menggunakan Metrik RPS untuk Layanan Nginx Reverse Proxy

I Gusti Putu Rangga Adithia Putra[1*], Kadek Yota Ernanda Aryanto[2], I Ketut Purnamawan[3]
[1-3]Teknik Informatika, Universitas Pendidikan Ganesha, Indonesia
**Penulis Korespondensi: rangga.adithia26@gmail.com*

Abstract. *Default CPU metrics in the Kubernetes Horizontal Pod Autoscaler (HPA) often fail to detect connection bottlenecks in I/O-bound workloads, such as Nginx reverse proxies, causing service failures despite low CPU utilization. This study aims to evaluate and compare the effectiveness of CPU-based HPA against custom Requests Per Second (RPS) metrics in mitigating connection-level bottlenecks. An experimental approach was conducted using a two-tier architecture (Nginx and a backend with a 500ms delay) under ramp-up and sustained high-load traffic scenarios. The results demonstrated that the CPU-based HPA stagnant with low CPU usage, producing HTTP 5xx error rates up to 57.4% during ramp-up and stagnating at 36% under sustained load due to its inability to detect operating system-level network queues. Conversely, the RPS-based HPA proactively mitigated these bottlenecks by executing timely scaling, suppressing error rates below 10% during ramp-up and significantly restoring service stability. These findings imply that utilizing RPS metrics successfully decouples autoscaling triggers from hardware resource saturation levels, making it a highly reliable metric for maintaining high availability in connection-limited microservice architecture.*

Keywords: *Autoscaling; Bottleneck; Horizontal Pod Autoscaler; Kubernetes; Requests Per Second.*

Abstrak. Metrik CPU bawaan pada Kubernetes *Horizontal Pod Autoscaler* (HPA) sering kali gagal mendeteksi hambatan koneksi pada beban kerja *I/O-bound*, seperti *reverse proxy* Nginx, yang menyebabkan kegagalan layanan meskipun utilisasi CPU rendah. Penelitian ini bertujuan mengevaluasi dan membandingkan efektivitas konfigurasi HPA berbasis CPU dengan metrik kustom *Requests Per Second* (RPS) dalam menangani hambatan tingkat koneksi. Pengujian dilakukan secara eksperimental menggunakan arsitektur dua lapis (Nginx dan *backend* dengan penundaan 500ms) di bawah skenario pengujian lalu lintas bertahap (*ramp-up*) dan tinggi berkelanjutan (*sustained high-load*). Hasil penelitian menunjukkan bahwa HPA berbasis CPU stagnan dengan penggunaan resource yang rendah, menghasilkan tingkat *error* HTTP 5xx hingga 57,4% pada skenario *ramp-up* dan stagnan pada 36% di skenario beban berkelanjutan akibat ketidakmampuannya mendeteksi antrean jaringan di tingkat sistem operasi. Sebaliknya, HPA berbasis RPS berhasil secara proaktif memitigasi hambatan tersebut dengan mengeksekusi penskalaan yang tepat waktu, menekan tingkat *error* di bawah 10% pada beban bertahap dan memulihkan stabilitas layanan secara signifikan. Temuan ini mengimplikasikan bahwa penggunaan metrik RPS sukses memisahkan pemicu penskalaan otomatis dari tingkat saturasi sumber daya perangkat keras, menjadikannya metrik yang jauh lebih andal untuk menjaga ketersediaan layanan pada arsitektur yang dibatasi oleh koneksi.

Kata Kunci: *Autoscaling; Bottleneck; Horizontal Pod Autoscaler; Kubernetes; Requests Per Second.*

1. LATAR BELAKANG

Cloud Computing dan *containerization* seperti Docker telah menjadi elemen esensial dalam dunia komputasi modern untuk mendistribusikan aplikasi secara cepat dan konsisten (Malallah et al., 2023; *What Is Cloud Computing? | Microsoft Azure.*, n.d.). Namun, fluktuasi beban kerja layanan dunia nyata yang dinamis mengharuskan penggunaan alat orkestrasi kontainer seperti Kubernetes untuk menjaga ketersediaan layanan (Raj et al., 2023). Kubernetes memfasilitasi manajemen skala otomatis melalui *Horizontal Pod Autoscaler* (HPA) yang secara *default* menggunakan metrik *CPU Usage* untuk menentukan kapan *scaling* harus dilakukan (Gao et al., 2017; *Kubernetes Documentation.*, n.d.; Šimon et al., 2023). Akan

Naskah Masuk: 16 April 2026; Revisi: 17 Mei 2026; Diterima: 07 Juni 2026; Tersedia: 24 Juni 2026

tetapi, metrik CPU sering kali gagal mencerminkan beban operasional aktual pada aplikasi yang terikat I/O (*I/O-bound*), seperti layanan web yang menggunakan Nginx sebagai *reverse proxy (Huotari, 2025; Zhou & Yong, 2024)*. Pada arsitektur layanan microservice, *proxy* sering kali bergantung pada sistem eksternal seperti layanan API pihak ke-3 yang dapat menimbulkan *delay*, sehingga memaksa *proxy* untuk menahan koneksi aktif. Akibatnya, layanan dapat mengalami kelebihan beban karena kehabisan batas koneksi (*worker_connections*) dan menghasilkan *error* HTTP 5xx, meskipun utilitas CPU pod terlihat stabil dan rendah (Guedes et al., 2024; Zhou & Yong, 2024). Hal ini menunjukkan kelemahan kritis: CPU bukanlah indikator yang memadai untuk degradasi layanan akibat hambatan koneksi *I/O-bound*.

Untuk mengatasi keterbatasan tersebut, HPA dapat diperluas menggunakan metrik kustom tingkat aplikasi seperti *Requests Per Second* (RPS). Berbeda dengan metrik CPU yang bersifat reaktif terhadap konsumsi sumber daya internal, RPS memberikan sinyal proaktif terkait intensitas *request* yang masuk. Ketika terjadi lonjakan *request*, RPS dapat segera memicu HPA untuk melakukan Autoscaling (*scale up*) sebelum batas sistem pada *worker_connection* di *reverse proxy* habis (Guedes et al., 2024; Huotari, 2025; Zhou & Yong, 2024). Meskipun HPA banyak digunakan, sebagian besar penelitian sebelumnya hanya berfokus pada Autoscaling berbasis CPU/memori (Ahmad et al., 2024; Nilsen, 2023a) atau metrik tingkat aplikasi lainnya (Camilo & Campos, 2024; Zhou & Yong, 2024). Penelitian Zhou & Yong (2024) berhasil menyoroti dilema hambatan I/O pada Nginx dan membuktikan bahwa metrik *error* HTTP 5xx kustom yang mereka buat mengungguli metrik CPU.

Namun, solusi tersebut secara inheren masih bersifat reaktif karena mengandalkan kegagalan permintaan pengguna sebelum sistem memulai pemulihan. Hingga saat ini, masih sangat sedikit penelitian yang secara khusus mengeksplorasi penggunaan RPS sebagai metrik Autoscaling proaktif untuk mengantisipasi saturasi koneksi sebelum menyebabkan terjadinya *error*. Berdasarkan celah penelitian tersebut, penelitian ini bertujuan untuk mengukur dan membandingkan kinerja konfigurasi HPA berbasis CPU dan berbasis RPS dalam menangani beban kerja *I/O-bound*. Evaluasi dilakukan berdasarkan metrik penggunaan CPU, jumlah pod, rata-rata waktu respons (latensi), dan tingkat *error* HTTP 5xx di bawah pola pengujian beban lalu lintas yang bervariasi, yakni beban bertahap (*ramp-up*) dan beban tinggi berkelanjutan (*sustained high load*). Melalui pengujian ini, efektivitas Autoscaling otomatis berbasis RPS diharapkan dapat diukur secara komprehensif dalam mempertahankan stabilitas dan performa layanan pada kondisi sistem yang terbatas koneksi dibandingkan dengan Autoscaling metrik CPU bawaan.

2. KAJIAN TEORITIS

Komputasi awan memberikan fleksibilitas melalui elastisitas, yaitu kemampuan sistem untuk menyesuaikan diri dengan variasi beban kerja secara otonom (Gawande & Gorthi, 2024; Noor et al., 2025; Raj et al., 2023). Pendekatan ini didukung secara luas oleh teknologi kontainerisasi, seperti Docker, yang mengemas aplikasi beserta dependensinya menjadi unit yang ringan dan lebih efisien dibandingkan mesin virtual konvensional (Jóhann Jónsson, 2020; Josh Campbell, n.d.). Untuk mengelola kontainer dalam skala besar, arsitektur layanan mikro menggunakan Kubernetes sebagai alat orkestrasi (Lorido-Botran et al., 2014; Molleti, 2022). Kubernetes mengabstraksi kontainer ke dalam unit penyebaran terkecil yang disebut *pod*, serta mengatur penjadwalan dan replikasinya di seluruh *node* pekerja (*worker nodes*) (Nilsen, 2023b).

Salah satu mekanisme utama Kubernetes untuk menjaga ketersediaan layanan adalah Autoscaling otomatis melalui *Horizontal Pod Autoscaler* (HPA). Secara bawaan (*default*), HPA memantau metrik pemanfaatan sumber daya perangkat keras seperti CPU atau memori, dan secara dinamis menambah atau mengurangi jumlah replika *pod* untuk mencapai target utilisasi yang ditentukan (*HPA Documentation*, n.d.). Namun, Kubernetes juga menyediakan kemampuan *Custom Metrics Autoscaling* yang memungkinkan perluasan fungsi HPA menggunakan metrik spesifik aplikasi, seperti RPS. Autoscaling ini diimplementasikan dengan memanfaatkan *Prometheus Adapter* sebagai jembatan untuk mengekspos data dari Prometheus ke dalam Kubernetes Metrics API, sehingga sistem dapat beradaptasi lebih cepat terhadap pola lalu lintas permintaan yang berfluktuasi tajam (Molleti, 2022).

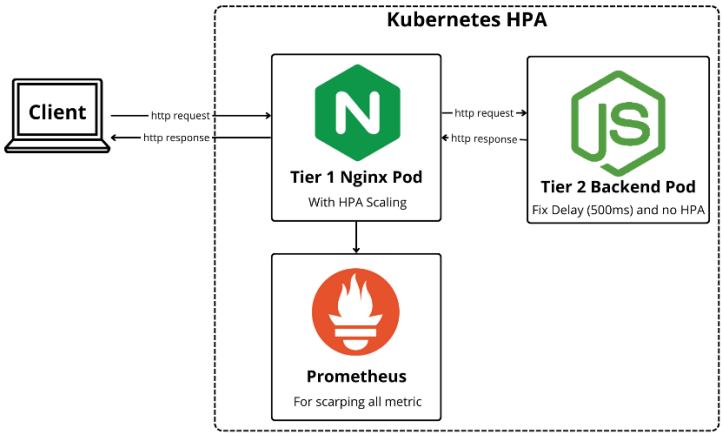
Beberapa penelitian sebelumnya telah mengevaluasi kinerja dan efisiensi HPA dalam menangani beban kerja dinamis. Penelitian oleh (Nilsen, 2023a) serta (Wirawan King et al., 2024) membandingkan performa HPA dengan *Vertical Pod Autoscaler* (VPA), di mana HPA terbukti lebih cepat dan efisien dalam merespons lonjakan beban. (Camilo & Campos, 2024) juga membandingkan HPA dengan mekanisme autoscaling berbasis *event* (KEDA) dan menemukan bahwa HPA memberikan pemrosesan pesan yang lebih stabil. Terkait penggunaan metrik kustom, penelitian (Zhou & Yong, 2024) mengevaluasi penerapan kode status HTTP 5xx pada HPA untuk mengatasi keterbatasan metrik CPU pada layanan *reverse proxy* Nginx. Mereka membuktikan bahwa metrik bawaan CPU tidak memadai saat terjadi kehabisan koneksi (*connection exhaustion*), dan metrik kustom HTTP 5xx berhasil meningkatkan ketersediaan layanan secara signifikan.

Meskipun penelitian (Zhou & Yong, 2024) membuktikan keunggulan metrik kustom dalam menangani hambatan terikat-I/O (*I/O-bound bottlenecks*), solusi yang diajukan secara inheren masih bersifat reaktif karena mengandalkan rasio *error* atau permintaan yang gagal. Pemanfaatan metrik tingkat aplikasi yang bersifat proaktif, seperti RPS, diyakini mampu mendeteksi intensitas lalu lintas aktual sebelum layanan mencapai batas koneksi kritisnya. Dengan demikian, penerapan HPA berbasis metrik RPS diproyeksikan mampu memberikan stabilitas dan tingkat respons layanan yang lebih superior dibandingkan HPA berbasis CPU, khususnya dalam memitigasi risiko kegagalan pada skenario layanan yang rentan terhadap hambatan pembatasan koneksi (*connection-limited bottlenecks*).

3. METODE PENELITIAN

Desain Sistem

Untuk mengevaluasi efektivitas *scaling* berbasis RPS dalam memitigasi *bottleneck* tingkat koneksi pada Nginx, sistem dirancang dengan arsitektur dua lapis (*two-tier*) yang memisahkan tanggung jawab penanganan *request* dan pemrosesan respons. Pemisahan arsitektural ini memungkinkan isolasi degradasi kinerja yang murni disebabkan oleh batas koneksi, terlepas dari tingkat utilisasi CPU atau memori. Hanya layanan lapis pertama yang dikenakan autoscaling otomatis (HPA) dan pengukuran kinerja, sedangkan layanan *backend* dipertahankan statis untuk memastikan efek yang diamati murni berasal dari lapisan *frontend*.



Gambar 1. Gambaran umum desain sistem.

Lapis pertama (*front-end*) terdiri dari *deployment* Nginx yang bertindak sebagai *reverse proxy*. *Proxy* ini menangani seluruh lalu lintas masuk dan meneruskannya ke layanan *backend*. *Pod* Nginx secara sengaja dikonfigurasi dengan kapasitas koneksi yang sangat dibatasi melalui nilai worker_connections yang rendah, serta pembatasan sumber daya CPU pada 500m.

Batasan ini berfungsi untuk mereproduksi kondisi di mana *pod* mengalami kelebihan beban akibat limitasi koneksi sebelum mencapai ambang batas penggunaan CPU atau memori.

Lapis kedua (*back-end*) adalah layanan *upstream* ringan berbasis Node.js yang bertugas mengembalikan respons dengan penundaan buatan yang tetap (*fixed delay*) sebesar 500ms. Layanan ini di- di-*deploy* di dalam klaster Kubernetes yang sama namun tidak dikenakan Autoscaling otomatis, dan diberikan alokasi CPU serta memori yang besar agar tidak menjadi faktor penghambat selama pengujian. Akibat adanya penundaan tetap sebesar 500ms tersebut, setiap koneksi akan tertahan (tetap terbuka) lebih lama di *reverse proxy* Nginx. Ketika tingkat permintaan (RPS) meningkat, akumulasi koneksi aktif serentak akan melonjak tajam hingga menyentuh batas kapasitas, memaksa Nginx untuk menolak koneksi baru yang berujung pada lonjakan latensi dan *error* HTTP 5xx, terlepas dari kestabilan penggunaan CPU.

Baseline Testing

Metode ini terinspirasi dari Nilsen (2023a), sebelum melakukan eksperimen Autoscaling otomatis yang utama, pengujian *baseline* (garis dasar) dilakukan untuk menentukan kapasitas beban lalu lintas maksimal yang dapat ditangani oleh sistem sebelum kinerjanya mulai mengalami degradasi. Langkah ini sangat penting untuk menetapkan nilai referensi ambang batas (*threshold*) RPS yang akan digunakan pada konfigurasi HPA berbasis RPS. Pengujian dilakukan dengan meningkatkan beban permintaan secara bertahap menggunakan tools k6, dimulai dari 50 RPS dan ditingkatkan secara inkremental sebesar 50 RPS setiap satu menit hingga mencapai puncak 500 RPS, dengan total durasi pengujian selama 10 menit. Pada setiap tingkat beban, sistem dipantau berdasarkan latensi, penggunaan CPU, dan tingkat *error*. Tujuan utamanya adalah untuk mengamati kapan sistem mulai menunjukkan tanda-tanda degradasi, terutama ketika latensi meningkat atau *error* HTTP 5xx mulai muncul, bahkan jika penggunaan CPU masih terlihat stabil. Titik ini dikenal sebagai titik infleksi (*knee point*), di mana sistem tidak lagi mampu mengimbangi permintaan yang masuk. Nilai RPS tepat sebelum titik kegagalan inilah yang kemudian dipilih sebagai ambang batas Autoscaling otomatis, sehingga memastikan HPA dapat bereaksi lebih awal sebelum hambatan tingkat koneksi (*connection-level bottlenecks*) menyebabkan kegagalan layanan.

Konfigurasi *Horizontal Pod Autoscaler* (HPA)

Konfigurasi HPA diterapkan menggunakan dua strategi metrik, yaitu utilitas CPU dan RPS. Autoscaling otomatis secara eksklusif difokuskan pada *deployment* Nginx di lapisan *frontend* yang bertindak sebagai *bottleneck* utama. Layanan *backend* dikecualikan dari Autoscaling dan diberikan alokasi sumber daya yang lebih tinggi agar tidak menjadi faktor

pembatas, sehingga dampak pemilihan metrik terhadap kinerja *rontend* dapat diisolasi secara objektif. Untuk mensimulasikan kondisi tekanan, setiap *pod* Nginx dialokasikan batasan 500 *millicores* (500m) CPU dan 256Mi memori. Pengaturan ini dipadukan dengan nilai *worker_connections* yang rendah guna mereproduksi keadaan di mana *pod* mengalami kelebihan beban akibat limitasi koneksi sebelum ambang batas perangkat keras CPU tercapai. Baik HPA berbasis CPU maupun RPS diterapkan dengan parameter ukuran replika yang identik, yakni batas minimum 1 *pod* dan maksimum 10 *pod*. Nilai target ambang batas Autoscaling (*scaling threshold*) yang pasti untuk kedua konfigurasi ini tidak ditetapkan di awal, melainkan ditentukan secara presisi setelah hasil dari *baseline testing* diperoleh.

Pengujian Beban (*Load Testing*)

Penelitian ini mengadaptasi metodologi pengujian beban dari (Wirawan King et al., 2024) guna mengevaluasi setiap konfigurasi HPA dalam mengelola degradasi kinerja yang berkaitan dengan koneksi. Jumlah pengguna konkuren didistribusikan menggunakan alat pengujian k6 untuk mengobservasi kemunculan hambatan (*bottleneck*) serta kecepatan reaksi HPA. Seluruh metrik dipantau menggunakan Prometheus pada interval satu menit dan divisualisasikan melalui dasbor Grafana. Pengujian ini dibagi menjadi dua skenario utama dengan durasi masing-masing 10 menit:

Ramp-up Load

Skenario ini dirancang untuk menyimulasikan lonjakan lalu lintas yang progresif. Beban permintaan ditingkatkan secara bertahap setiap menitnya, dimulai dari 50 RPS pada menit pertama hingga perlahan naik dan mencapai puncaknya sebesar 500 RPS pada menit ke-

Sustained High-Load

Skenario ini bertujuan untuk menguji stabilitas mekanisme Autoscaling di bawah tekanan yang konstan. Pengujian diawali dengan beban awal sebesar 50 RPS, kemudian sistem langsung diberikan beban kejut stabil sebesar 500 RPS yang dipertahankan dari menit ke-2 hingga menit ke-10.

Metrik Evaluasi

Metrik evaluasi yang digunakan dalam penelitian ini diadaptasi dari penelitian sebelumnya (Camilo & Campos, 2024; Nilsen, 2023a; Wirawan King et al., 2024) dan disesuaikan dengan tujuan serta jenis hambatan spesifik yang dieksplorasi. Evaluasi ini berfokus pada pemantauan hasil kinerja dan utilisasi sumber daya selama proses Autoscaling otomatis. Daftar lengkap metrik yang digunakan beserta deskripsinya disajikan pada Tabel 1.

Tabel 1. Metrik Evaluasi.

No	Metrik	Deskripsi
1	Jumlah Replika *Pod*	Menunjukkan bagaimana sistem menyesuaikan skala sebagai respons terhadap beban. Peningkatan jumlah *pod* yang lebih cepat atau adaptif mencerminkan daya tanggap yang lebih baik terhadap hambatan (*bottlenecks*).
2	Rata-rata Waktu Respons (ms)	Mengukur rata-rata waktu penyelesaian permintaan. Peningkatan latensi adalah gejala utama kelebihan beban koneksi dan penundaan antrean pada *reverse proxy*.
3	Tingkat *Error* (%)	Mengukur persentase permintaan yang gagal, khususnya *error* HTTP 5xx, yang secara khas menandakan bahwa *proxy* Nginx tidak dapat lagi menerima koneksi baru.
4	Penggunaan CPU (%)	Membantu memverifikasi apakah Autoscaling dipicu oleh kehabisan sumber daya aktual atau apakah sumber daya justru tetap menganggur meskipun terjadi degradasi kinerja.

4. HASIL DAN PEMBAHASAN

Lingkungan dan Implementasi Sistem

Seluruh eksperimen dalam penelitian ini dilakukan secara terisolasi menggunakan *virtual machine* (VM) pada Google Cloud Platform (GCP) dengan tipe *instance* e2-standard-2 (2 vCPU, 8 GB RAM, 40 GB *Persistent Disk*) yang berjalan di atas sistem operasi Ubuntu 24.04 LTS. Pemilihan lingkungan *cloud* ini bertujuan untuk meminimalkan interferensi proses latar belakang dari perangkat keras lokal dan menyerupai arsitektur *worker node* pada lingkungan *production* yang sebenarnya.

Untuk infrastruktur utamanya, klaster Kubernetes *single-node* disimulasikan menggunakan Minikube v1.35.0 dengan Docker sebagai *container runtime*. Layanan Nginx di-*deploy* sebagai *reverse proxy* di lapisan *frontend* yang sekaligus menjadi target utama dari HPA. Simulasi lalu lintas HTTP untuk pengujian beban dieksekusi menggunakan alat k6 berbasis JavaScript. Sementara itu, pemantauan dan pengumpulan metrik sistem ditarik menggunakan Prometheus dan divisualisasikan secara *real-time* melalui Grafana. Khusus

untuk implementasi HPA berbasis metrik kustom (RPS), *Prometheus Adapter* diintegrasikan sebagai jembatan untuk mengekspos metrik tingkat aplikasi dari Prometheus ke dalam *Kubernetes Custom Metrics API*.

Hasil Pengujian dan Evaluasi

Hasil Pengujian Baseline Testing

Hasil pengujian *baseline* menunjukkan bahwa sistem mempertahankan kinerja stabil pada tingkat *request* yang rendah. Seperti pada tabel 2. Saat beban 50 RPS, sistem memproses permintaan secara efisien dengan rata-rata waktu respons p50 sekitar 530ms, utilisasi CPU rata-rata 31,7%, dan tingkat *error* 0,0%. Degradasi layanan mulai terlihat saat beban mencapai 100 RPS, yang ditandai dengan kemunculan *error* HTTP 5xx sebesar 0,7% dan utilisasi CPU mencapai 47,3%. *Bottleneck* kritis mulai terjadi pada 150 RPS ketika tingkat *error* melonjak tajam menjadi 23,6%, meskipun latensi p50 tetap stabil di angka 530ms pada awal lonjakan lalu lintas ini.

Berdasarkan observasi tersebut, nilai 80 RPS dipilih sebagai ambang batas (*threshold*) target untuk konfigurasi HPA berbasis RPS. Mengingat sistem mulai menjatuhkan koneksi (*drop connections*) dan menghasilkan *error* pada 100 RPS, penetapan *threshold* di angka 80 RPS memberikan margin keamanan (*safety margin*) yang krusial. Margin ini memastikan mekanisme *autoscaling* memiliki waktu yang cukup untuk menyediakan dan menginisialisasi *pod* baru selama fase *ramp-up* sebelum sistem menyentuh *bottleneck* kritis di titik 100 RPS. Untuk memastikan keadilan dalam analisis komparatif, *threshold* HPA berbasis CPU harus dikonfigurasi agar terpicu pada tingkat beban kerja yang ekuivalen dengan 80 RPS. Berdasarkan data yang diperoleh, utilisasi CPU berada di angka 31,7% pada 50 RPS dan 47,3% pada 100 RPS. Untuk menentukan utilisasi CPU yang tepat dan berkorespondensi dengan target 80 RPS tersebut, digunakan formula interpolasi linier berikut:

$$CPU_{target} = CPU_{50} += \left(\frac{RPS_{target} - RPS_{50}}{RPS_{100} - RPS_{50}}\right) \times (CPU_{100} - CPU_{50}) \quad \textbf{(I)}$$

Dengan menyubstitusikan nilai *baseline* $(RPS_{target} = 80, CPU_{50} = 31.7\%, CPU_{100} = 47.3\%$, diperoleh hasil kalkulasi sebagai berikut:

$$CPU_{target} = 31.7 += \left(\frac{80 - 50}{100 - 50}\right) \times (47.3 - 31.7) = 41.06\% \quad \textbf{(II)}$$

Untuk memberikan nilai konfigurasi praktis sambil mempertahankan ekuivalensi tersebut, target utilisasi CPU ditetapkan sebesar 41% untuk konfigurasi HPA berbasis CPU. Pendekatan matematis ini memastikan bahwa kedua metrik Autoscaling dievaluasi di bawah kondisi tekanan *baseline* yang identik, sehingga memberikan landasan yang kuat dan adil untuk membandingkan daya tanggap serta efektivitasnya.

Tabel 2. Hasil Uji *Baseline Testing*.

RPS	p50 Latency (ms)	p99 Latency (ms)	Error Rate (%)	Avg CPU Usage (%)
50	529.97	597.64	0.0	31.7
100	529.70	677.46	0.7	47.3
150	529.71	577.47	23.6	56.9
200	529.91	568.83	45.0	61.5
250	530.18	570.37	57.2	65.5
300	530.60	616.34	65.0	69.1
350	531.83	603.42	70.4	73.6
400	542.24	1265.67	74.0	75.7
450	534.24	786.43	77.3	81.7
500	544.92	1083.43	79.5	85.7

Hasil Pengujian HPA Berbasis CPU pada Skenario Ramp-up Load

Pada fase awal pengujian dengan target utilisasi 41%, HPA berbasis CPU mampu menangani beban secara efisien. Ketika beban mencapai 100 RPS, utilisasi CPU menyentuh 47,3% dan berhasil memicu Autoscaling menjadi 2 *pod* pada titik 150 RPS. Tindakan awal ini sukses mendistribusikan beban kerja, menurunkan penggunaan CPU ke 36,1%, dan menjaga tingkat *error* di angka yang sangat rendah, yaitu 0,8%. Namun, kelemahan Autoscaling ini mulai terlihat saat lalu lintas meningkat ke 200 RPS. Sistem mengalami degradasi mendadak yang ditandai dengan lonjakan latensi p50 hingga 923,74ms dan tingkat *error* sebesar 11,1%, meskipun utilisasi CPU hanya tercatat 44,8%.

Temuan paling kritis terjadi di antara rentang 250 RPS hingga 400 RPS. Pada periode ini, layanan mengalami gangguan parah dengan lonjakan *error* HTTP 5xx dari 32,7% hingga mencapai puncaknya di 57,4%. Rata-rata utilisasi CPU tidak mencerminkan kegagalan tersebut, metrik ini justru stagnan di kisaran 39,7% hingga 49,6% dan sering kali turun di bawah target 41%. Fenomena ini mengilustrasikan terjadinya *bottleneck* tingkat koneksi. Karena permintaan masuk ditolak dengan cepat atau mengantre akibat batas koneksi pada *reverse proxy* Nginx, CPU tidak memprosesnya secara aktif dan cenderung stabil. Akibat representasi metrik yang tidak akurat tersebut, HPA gagal mendeteksi tekanan aktual pada sistem dan mempertahankan jumlah replika pada 2 *pod* selama fase degradasi yang paling berat. HPA baru berhasil melakukan Autoscaling menjadi 3 *pod* pada beban 450 RPS (sedikit menurunkan tingkat *error* ke 44,3%), namun reaksi yang sangat terlambat ini pada akhirnya gagal mempertahankan ketersediaan layanan. Hasil ini secara kuat mengonfirmasi hipotesis

bahwa utilisasi CPU adalah indikator yang lambat dan tidak dapat diandalkan untuk Autoscaling beban kerja *I/O-bound* yang dibatasi oleh kapasitas koneksi.

Tabel 3. Kinerja HPA Berbasis CPU pada Skenario *Ramp-up Load.*

RPS	p50 Latency (ms)	p99 Latency (ms)	Error Rate (%)	Avg CPU Usage (%)	Pod Replica
50	535.28	568.08	0.0	32.7	1
100	536.67	572.11	0.0	47.3	1
150	559.45	893.13	0.8	36.1	2
200	923.74	1417.34	11.1	44.8	2
250	530.04	1626.25	32.7	43.9	2
300	528.75	1919.87	51.7	39.7	2
350	531.65	2235.85	57.4	42.8	2
400	535.34	2093.57	53.1	49.6	2
450	539.17	2829.34	44.3	39.8	3
500	533.63	2843.35	42.9	41.5	3

Hasil Pengujian HPA Berbasis RPS pada Skenario Ramp-up Load

Kinerja sistem menunjukkan hasil yang sangat adaptif saat menggunakan metrik RPS dengan ambang batas target 80 RPS per *pod*. Ketika beban meningkat dan melampaui target pada 100 RPS, HPA mendeteksi lonjakan lalu lintas secara akurat dan proaktif menginisiasi *pod* kedua. Pada titik 150 RPS, beban kerja terdistribusi secara efektif pada 2 replika dengan tingkat *error* 1,0% dan latensi p50 stabil di 538,01ms. Saat beban dinaikan antara 200 hingga 250 RPS, sistem mengalami masa transisi singkat dengan sedikit peningkatan *error* (6,6% dan 7,3%) selama *pod* baru sedang diinisialisasi untuk menangani antrean koneksi.

Namun, berbeda dengan konfigurasi CPU yang mengalami *blind spot* dan stagnan pada 2 *pod* di fase ini, metrik RPS mampu membaca volume permintaan masuk yang sebenarnya. Akibatnya, HPA mengeksekusi Autoscaling proaktif secara berkelanjutan, meningkatkan jumlah replika menjadi 3 *pod* pada 250 RPS dan 4 *pod* pada 300 RPS. Autoscaling proaktif ini menghasilkan pemulihan yang sangat sukses, penambahan *pod* keempat secara drastis menekan tingkat *error* turun menjadi 0,1% dengan latensi p99 stabil pada 753,72ms. Hal ini sangat kontras dengan HPA berbasis CPU yang lumpuh pada 2 *pod* dengan tingkat *error* 51,7% di titik beban 300 RPS yang sama.

Sepanjang fase beban terberat (350 hingga 500 RPS), metrik RPS membuktikan keunggulannya dalam menjaga ketersediaan layanan (*high availability*). HPA secara mulus

meningkatkan skala *deployment* hingga mencapai 6 *pod* pada beban puncak 500 RPS. Dengan mencocokkan jumlah replika secara dinamis terhadap jumlah absolut permintaan masuk, sistem menahan *error* HTTP 5xx mendekati batas nol (hanya memuncak di 0,9% pada 500 RPS) dan menjaga latensi p50 stabil di kisaran 530ms. Secara krusial, utilisasi CPU rata-rata selama proses *ramp-up* tidak pernah melampaui 44,6%. Temuan ini secara definitif membuktikan hipotesis utama penelitian: RPS adalah indikator yang jauh lebih akurat dan responsif untuk Autoscaling otomatis layanan web *I/O-bound*. Dengan memisahkan pemicu Autoscaling dari saturasi sumber daya *hardware*, HPA berbasis RPS berhasil mencegah *bottleneck* tingkat koneksi yang sebelumnya menyebabkan kegagalan sistem pada HPA berbasis CPU.

Tabel 4. Kinerja HPA Berbasis RPS pada Skenario *Ramp-up Load.*

RPS	p50 Latency (ms)	p99 Latency (ms)	Error Rate (%)	Avg CPU Usage (%)	Pod Replica
50	532.91	558.51	0.0	29.3	1
100	533.51	560.35	0.0	44.2	1
150	538.01	789.93	1.0	31.7	2
200	556.05	1174.29	6.6	41.9	2
250	528.78	1901.08	7.3	44.6	3
300	529.21	753.72	0.1	41.7	4
350	527.55	597.80	0.0	35.1	4
400	528.05	614.62	0.0	36.8	5
450	529.92	652.71	0.0	34.7	5
500	549.32	855.94	0.9	32.1	6

Hasil Pengujian HPA Berbasis CPU pada Skenario Sustained High-Load

Pengujian beban *sustained high-load* menyimulasikan lonjakan lalu lintas ekstrem secara tiba-tiba dari *baseline* 50 RPS langsung menuju 500 RPS. Pada awal pengujian (50 RPS), sistem beroperasi normal dengan 1 *pod* (utilisasi CPU 31,7% dan tingkat *error* 0,0%). Namun, saat beban melonjak drastis, sistem mengalami *initial shock* yang sangat parah. Karena metrik tingkat *hardware* membutuhkan waktu untuk diagregasi oleh *metrics-server*, HPA tidak dapat bereaksi seketika. Hal ini menyebabkan *pod* tunggal kelebihan beban secara masif, menghasilkan tingkat *error* HTTP 5xx yang fatal sebesar 67,4% dan lonjakan latensi p99 hingga 1749,57ms. Meskipun HPA pada akhirnya mendeteksi lonjakan CPU dan melakukan

Autoscaling bertahap hingga mencapai 4 *pod*, reaksi yang tertunda ini membiarkan layanan terdegradasi selama beberapa menit pertama.

Temuan paling krusial dalam skenario ini terjadi tepat ketika sistem mencapai 4 replika *pod*. Pada tahap ini, beban yang terdistribusi berhasil menurunkan utilisasi CPU rata-rata kembali ke kisaran 35,2% hingga 44,3%. Karena tingkat utilisasi ini berada di sekitar target ambang batas konfigurasi HPA, sistem menginterpretasikan status tersebut sebagai kondisi optimal dan menghentikan seluruh tindakan Autoscaling lebih lanjut. Padahal, dengan 500 RPS yang didistribusikan ke 4 *pod* (sekitar 125 RPS per *pod*), masing-masing *pod* telah melampaui batas toleransi kapasitas koneksi (*worker_connections*) seperti yang dibuktikan pada pengujian *baseline* (>100 RPS). Akibatnya, tingkat *error* stagnan secara permanen di kisaran 34,0% hingga 36,8% di sepanjang sisa waktu pengujian. Hasil ini secara definitif membuktikan bahwa utilisasi CPU adalah metrik yang tidak cocok untuk memitigasi *bottleneck* koneksi, HPA berbasis CPU menciptakan ilusi stabilitas sumber daya yang sepenuhnya menutupi kenyataan bahwa lebih dari sepertiga permintaan pengguna secara aktif mengalami kegagalan akibat saturasi koneksi.

Tabel 5. Kinerja HPA Berbasis CPU pada Skenario *Sustained High-Load*

RPS	p50 Latency (ms)	p99 Latency (ms)	Error Rate (%)	Avg CPU Usage (%)	Pod Replica
50	533.92	562.65	0.0	31.7	1
500	546.96	1749.57	67.4	59.2	1
500	610.11	2245.01	64.5	60.4	2
500	629.38	3685.42	55.6	66.6	2
500	576.46	7723.67	36.8	35.2	4
500	668.59	4867.26	36.8	39.3	4
500	872.14	4968.91	34.1	44.3	4
500	892.67	4718.01	35.1	41.4	4
500	809.61	3159.43	35.8	43	4
500	862.15	4224.35	34.0	44	4

Hasil Pengujian HPA Berbasis RPS pada Skenario Sustained High-Load

Mirip dengan konfigurasi CPU, sistem berbasis RPS juga mengalami *initial shock* yang parah saat beban melonjak secara instan dari 50 ke 500 RPS. *Pod* tunggal yang aktif langsung

mengalami kelebihan beban, menghasilkan tingkat *error* HTTP 5xx awal sebesar 50,5% dan lonjakan latensi p99 menjadi 579,55ms. Degradasi awal ini wajar terjadi secara teoretis karena mekanisme *autoscaling* reaktif secara inheren membutuhkan jeda waktu (*window*) untuk mengumpulkan metrik, memicu *event*, dan menginisialisasi kontainer baru secara fisik.

Namun, divergensi krusial antara kedua metrik autoscaling ini terlihat sangat jelas pada fase pemulihan selanjutnya. Berbeda dengan HPA berbasis CPU yang terjebak pada ekuilibrium semu (*false equilibrium*) di 4 *pod* (dengan tingkat *error* stagnan 36%), HPA berbasis RPS berhasil melewati hambatan tersebut. Ketika jumlah replika mencapai 4 *pod*, utilisasi CPU rata-rata turun menjadi 38,6%, angka yang sebelumnya menipu *autoscaler* berbasis sumber daya *hardware*. Akan tetapi, metrik RPS secara akurat mengidentifikasi bahwa distribusi beban (sekitar 125 RPS per *pod*) masih jauh melebihi target aman 80 RPS, sehingga HPA RPS terus melanjutkan tindakan Autoscalingnya secara agresif.

Sistem secara berkelanjutan menyediakan sumber daya tambahan hingga mencapai puncaknya di 9 *pod*. Hasilnya, tingkat *error* menunjukkan tren pemulihan yang progresif dan sangat signifikan, turun dari 50,5% menjadi rata-rata 11,6% hingga 15,1% pada tahap akhir pengujian. Patut digarisbawahi bahwa ketika sistem mencapai 9 *pod*, utilisasi CPU rata-rata justru anjlok hingga hanya menyentuh 24,4%. Observasi akhir ini menjadi bukti bagi hipotesis penelitian: pada beban kerja yang dibatasi oleh kapasitas koneksi (*I/O bound*), utilisasi CPU terlepas sepenuhnya (*decoupled*) dari ketersediaan layanan aktual. Dengan mengandalkan metrik RPS, *autoscaler* berhasil mempertahankan representasi nyata dari *bottleneck* koneksi dan mengeksekusi upaya pemulihan berkelanjutan yang secara struktural gagal dicapai oleh metrik CPU.

Tabel 6. Kinerja HPA Berbasis RPS pada Skenario *Sustained High-Load.*

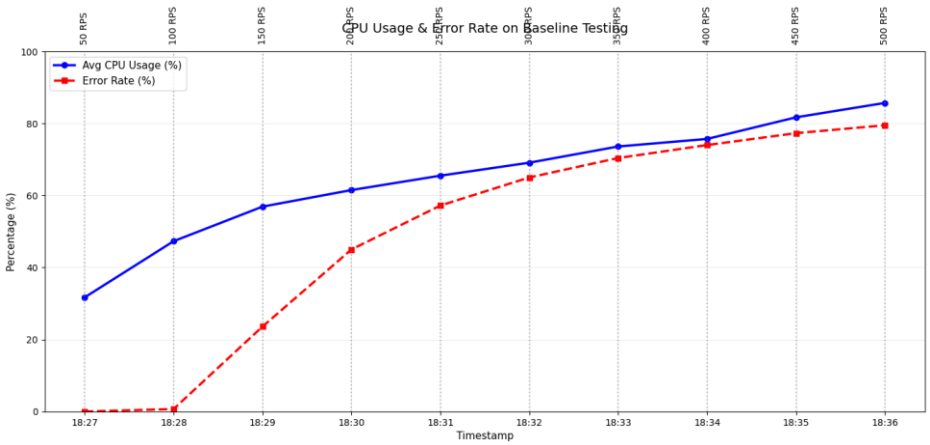
RPS	p50 Latency (ms)	p99 Latency (ms)	Error Rate (%)	Avg CPU Usage (%)	Pod Replica
50	529.98	552.34	0.0	29.4	1
500	534.49	579.55	50.5	59	2
500	690.55	1099.86	49.1	38.6	4
500	879.29	1735.55	29.5	37.6	5
500	856.72	687.60	22.0	27.6	7
500	906.87	665.27	20.2	28.3	7
500	893.52	735.23	12.9	27.8	8
500	891.67	806.77	11.6	26.7	8
500	924.71	834.14	13.8	26.5	9

500	1000.45	1578.94	15.1	24.4	9

Analisis *Error* dan *Bottleneck*

Connection-Level Bottleneck di Nginx Reverse Proxy

Untuk memahami kegagalan Autoscaling pada konfigurasi berbasis CPU, penting untuk mengidentifikasi karakteristik *bottleneck* pada kinerja sistem. Keterbatasan utama yang membatasi ketersediaan layanan dalam eksperimen ini bukanlah kapasitas komputasi (CPU) pada *pod*, melainkan kapasitas penanganan koneksi di lapisan *reverse proxy*. Keterbatasan ini secara sengaja diinduksi melalui arsitektur sistem yang menerapkan penundaan pemrosesan (*processing delay*) tetap sebesar 500ms pada layanan *backend*. Karena Nginx beroperasi sebagai *reverse proxy*, ia dipaksa untuk menahan setiap koneksi masuk tetap terbuka minimal selama setengah detik sembari menunggu respons dari *backend*. Seiring dengan meningkatnya laju permintaan (RPS), akumulasi jumlah koneksi aktif yang terjadi secara bersamaan melonjak dengan cepat, sehingga menguras batas worker_connections yang dikonfigurasi pada *pod* Nginx.



Gambar 2. *CPU Usage* dan *Error Rate* saat *Baseline Testing.*

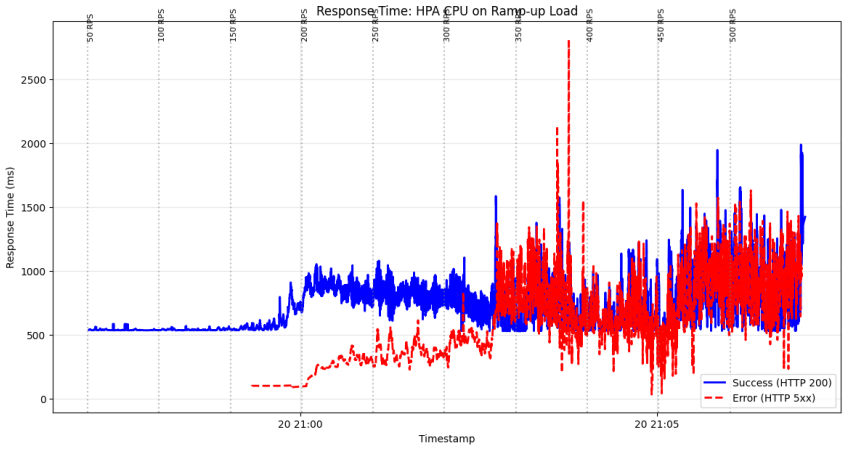
Bukti terjadinya *bottleneck* tingkat koneksi ini terlihat secara jelas pada hasil pengujian *baseline* (Gambar 5). Di bawah beban ringan 50 RPS, *pod* tunggal Nginx beroperasi stabil dengan tingkat *error* 0,0% dan utilisasi CPU rata-rata 31,7%. Namun, sistem menyentuh ambang batas kritis saat beban mencapai 100 RPS. Pada titik inilah *error* HTTP 5xx mulai muncul (0,7%), yang mengindikasikan bahwa antrean koneksi telah jenuh dan *proxy* terpaksa menolak permintaan baru yang masuk.

Fakta bahwa batas koneksi ini terlepas sama sekali (*decoupled*) dari saturasi sumber daya perangkat keras menjadi semakin jelas ketika beban dinaikkan ke 150 RPS. Pada tahap

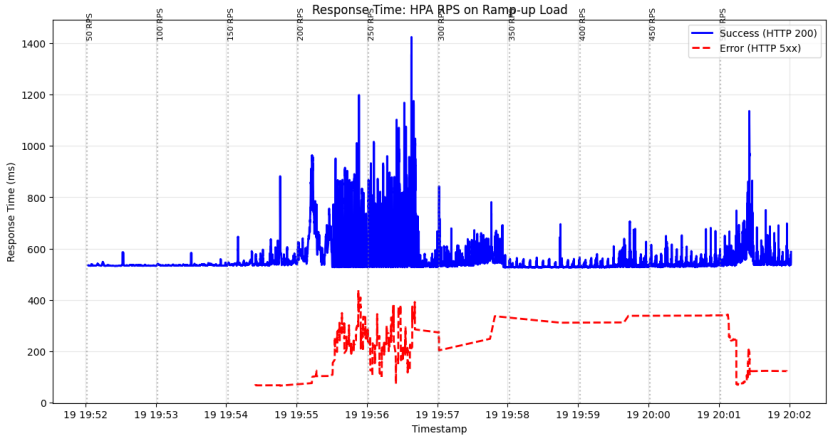
ini, layanan mengalami degradasi parah dengan lonjakan tingkat *error* hingga 23,6%, padahal rata-rata utilisasi CPU hanya berada di angka 56,9%, masih sangat jauh dari titik saturasi absolut. Hal ini mengonfirmasi bahwa degradasi layanan terjadi murni karena sistem kehabisan ketersediaan slot koneksi untuk menerima permintaan baru, bukan karena kekurangan daya komputasi prosesor. Oleh karena itu, mengandalkan metrik CPU sebagai pemicu Autoscaling di bawah kondisi beban kerja *I/O-bound* yang spesifik ini akan menciptakan ketidaksesuaian yang fundamental antara persepsi kesehatan sistem dan ketersediaan layanan yang sebenarnya.

Mekanisme Fast-Fail dan Kongesti TCP Backlog

Evolusi degradasi layanan dalam eksperimen ini dapat divisualisasikan melalui pelacakan waktu respons berkelanjutan yang terbagi ke dalam dua fase. Fase pertama, "*fast-fail*", terjadi ketika Nginx kehabisan kapasitas *worker_connections* dan langsung menolak permintaan baru (HTTP 5xx) tanpa meneruskannya ke *backend*. Karena melewati *delay* buatan 500ms di *backend*, respons *error* ini terkirim sangat cepat (<500ms). Fase kedua, "kongesti *TCP backlog*", terjadi saat lalu lintas terlampau masif sehingga permintaan mulai mengantre di tingkat sistem operasi (di dalam *TCP listen backlog* pada *kernel*) sebelum *reverse proxy* sempat menolaknya. Antrean tingkat OS ini secara artifisial menaikan latensi *error* jauh melampaui ambang batas 500ms.

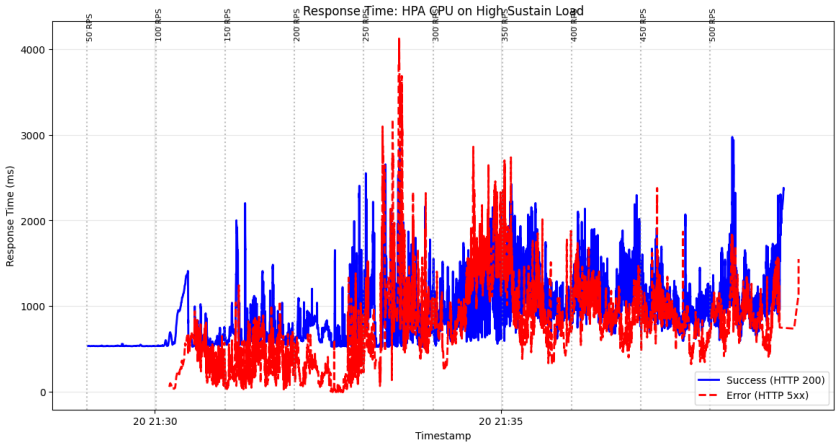


Gambar 3. Latensi HPA berbasis CPU pada skenario *Ramp-up Load.*

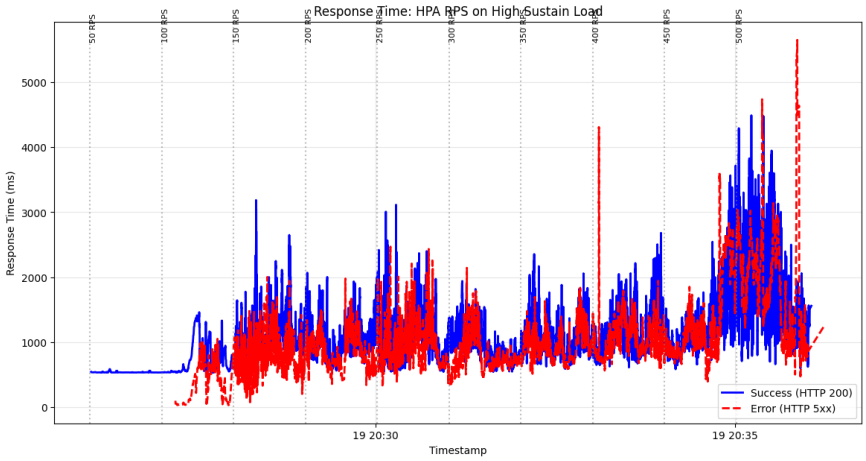


Gambar 4. Latensi HPA berbasis RPS pada skenario *Ramp-up Load*.

Pada skenario beban bertahap, konfigurasi HPA berbasis CPU (Gambar 6) secara jelas menangkap kedua fase degradasi tersebut. Antara 150 hingga 250 RPS, latensi *error* konsisten berada di kisaran 100-400ms, membuktikan bahwa *proxy* berada pada fase *fast-fail*. Namun, saat beban melampaui 300 RPS, sistem memasuki kongesti *TCP backlog* yang ditandai dengan menlonjaknya latensi *error* hingga melampaui 1000ms dan memuncak di atas 2500ms. Sebaliknya, konfigurasi HPA berbasis RPS pada skenario yang sama (Gambar 7) menunjukkan ketahanan yang baik. Berkat autoscaling yang proaktif, latensi *error* muncul sangat jarang dan tidak pernah melampaui ambang batas 500ms. Sistem berhasil memitigasi *bottleneck* pada tahap *fast-fail* dan sepenuhnya terhindar dari kongesti *TCP backlog*.



Gambar 5. Latensi HPA berbasis CPU pada skenario *Sustain High Load*.



Gambar 6. Latensi HPA berbasis RPS pada skenario *Sustain High Load.*

Kelemahan metrik CPU semakin terekspos pada skenario beban *sustained high-load*. Pada konfigurasi CPU (Gambar 8), grafik menunjukkan degradasi yang masif dan persisten. Karena HPA menghentikan autoscaling di 4 *pod*, sistem terkunci permanen dalam kondisi kongesti *TCP backlog* dengan fluktuasi latensi *error* yang sering kali menembus 2000ms hingga ekstrem di atas 4000ms. Di sisi lain, HPA berbasis RPS (Gambar 9) memang mengalami *initial shock* dengan lonjakan latensi mencapai 5000ms. Namun, perbedaan krusial terletak pada lintasan pemulihannya. Karena metrik RPS terus membaca tekanan 500 RPS terlepas dari rendahnya utilisasi CPU, HPA mendorong sistem untuk terus menambah skala hingga 9 *pod*. Penyediaan sumber daya yang berkelanjutan ini pada akhirnya berhasil memecah kebuntuan antrean jaringan, yang selaras dengan data penurunan tingkat *error* secara progresif menjadi 11,6%.

Keterbatasan Metrik CPU dalam Mendeteksi Bottleneck I/O-Bound

Kegagalan HPA berbasis CPU dalam mengeksekusi Autoscaling berkaitan erat dengan karakteristik komputasi pada fase *fast-fail* dan kongesti *TCP backlog*. Pada fase *fast-fail*, Nginx menolak koneksi karena kapasitas *worker_connections* telah habis. Proses pengembalian *error* statis HTTP 5xx dan pemutusan koneksi ini merupakan operasi yang sangat ringan karena melewati logika *proxying* yang kompleks dan waktu tunggu *backend* 500ms. Akibatnya, proses penolakan cepat ini hampir tidak mengonsumsi siklus prosesor, sehingga CPU *pod* sebagian besar tetap stagnan meskipun volume permintaan yang gagal sangat tinggi. Selanjutnya, saat sistem memburuk dan memasuki fase kongesti *TCP backlog*, permintaan yang masuk akan mengantre pada tingkat kernel sistem operasi (OS). Karena *worker* Nginx telah sepenuhnya jenuh, mereka tidak dapat mengeksekusi panggilan sistem (*system call*) accept() untuk menarik koneksi baru dari antrean tersebut. Mengingat metrik CPU hanya mengukur beban komputasi

aktif dari *pod*, metrik ini menjadi sepenuhnya buta terhadap kongesti jaringan masif di tingkat OS ini. Keterputusan struktural ini menjadi bukti definitif bahwa metrik sumber daya *hardware* seperti utilisasi CPU adalah indikator yang tidak cocok secara fundamental untuk memicu autoscaling otomatis pada kondisi hambatan *I/O-bound* yang terikat oleh kapasitas koneksi.

Responsivitas Proaktif Autoscaling Berbasis RPS

Keterbatasan fundamental dari konfigurasi berbasis CPU, baik ketidakmampuannya mendeteksi kongesti jaringan tingkat OS maupun kerentanannya terhadap ilusi stabilitas, berhasil diatasi secara efektif dengan mengadopsi RPS sebagai metrik autoscaling. Berbeda dengan metrik berbasis *hardware* yang reaktif terhadap konsumsi sumber daya internal, RPS memberikan sinyal proaktif dengan mengukur volume absolut trafik masuk secara langsung. Pendekatan ini memungkinkan *autoscaler* untuk mendeteksi tekanan tingkat koneksi secara instan, sehingga terhindar dari *blind spots* yang disebabkan oleh mekanisme *fast-fail* dan kongesti *TCP backlog*. Keunggulan proaktif ini terekspos sangat jelas pada fase pemulihan skenario beban *sustained high-load*. Pada saat HPA berbasis CPU berhenti *scaling* sistem di angka 4 *pod* akibat utilisasi CPU yang turun ke tingkat stagnan (40%), konfigurasi RPS juga mengalami penurunan utilisasi CPU menjadi 38,6% di titik 4 *pod* yang sama. Namun, konfigurasi RPS tidak tertipu oleh rendahnya utilitas prosesor tersebut. Karena metrik RPS terus memantau laju permintaan aktual, sistem secara presisi menyadari bahwa beban terdistribusi sebesar 125 RPS per *pod* masih jauh melampaui ambang batas operasional aman, yakni 80 RPS. Digerakkan oleh metrik yang peka terhadap lalu lintas ini (*traffic-aware*), HPA RPS secara berkelanjutan memaksa tindakan *scale-up* hingga menyediakan 9 *pod*. Autoscaling yang agresif dan akurat ini secara efektif memecah kebuntuan saturasi antrean koneksi, menekan tingkat *error* secara progresif turun ke angka 11,6%, dan memulihkan stabilitas layanan. Pada akhirnya, keberhasilan ini mendemonstrasikan bahwa penggunaan RPS sukses memisahkan (*decouple*) pemicu autoscaling otomatis dari jejak konsumsi sumber daya hardware aplikasi. Dengan melakukan autoscaling berdasarkan permintaan tingkat koneksi absolut alih-alih beban komputasi, metrik RPS secara proaktif memitigasi *bottleneck I/O-bound* dan sukses mempertahankan ketersediaan tinggi (*high availability*) pada lingkungan *reverse proxy* Nginx.

4. KESIMPULAN DAN SARAN

Penelitian ini menyimpulkan bahwa konfigurasi HPA berbasis metrik RPS, secara signifikan mengungguli HPA berbasis CPU dalam memitigasi bottleneck tingkat koneksi pada lingkungan reverse proxy Nginx. Hasil eksperimen secara tegas membuktikan bahwa utilisasi CPU adalah metrik yang cacat secara fundamental untuk menangani beban kerja I/O-bound, karena gagal menangkap fenomena penolakan koneksi fast-fail dan kongesti TCP backlog di tingkat sistem operasi. Pada skenario beban bertahap (ramp-up), HPA berbasis CPU mengalami kegagalan Autoscaling yang memadai sehingga menghasilkan tingkat error HTTP 5xx puncak sebesar 57,4%. Sebaliknya, HPA RPS secara proaktif menskalakan deployment hingga 6 replika, menekan error di bawah 10% (memuncak hanya di 7,2%), dan menjaga latensi rata-rata stabil di kisaran 530ms. Pada skenario beban *sustained high-load*, HPA CPU terjebak dalam masalah stabilitas semu di 4 pod dengan tingkat error stagnan pada 35-36%, sedangkan HPA RPS secara agresif melakukan Autoscaling hingga 9 pod dan sukses memulihkan tingkat error turun menjadi 11,6%. Penggunaan metrik RPS terbukti berhasil memisahkan (decouple) pemicu Autoscaling dari saturasi perangkat keras, memungkinkan sistem bereaksi langsung terhadap volume absolut lalu lintas dan menjaga ketersediaan layanan secara efektif.

Meskipun memberikan hasil yang konklusif, penelitian ini memiliki keterbatasan karena dilakukan menggunakan klaster single-node Minikube lokal dengan penundaan pemrosesan backend statis dan skenario beban yang dibatasi selama 10 menit. Berdasarkan keterbatasan tersebut, penelitian selanjutnya sangat disarankan untuk mereplikasi pengujian ini di lingkungan cloud multi-node yang terdistribusi secara nyata (seperti Google Kubernetes Engine atau Amazon EKS) guna mengevaluasi dampak latensi jaringan antar-node terhadap kecepatan agregasi metrik. Selain itu, pengujian menggunakan beban kerja dinamis dan lalu lintas harian dunia nyata berdurasi panjang (diurnal traffic) perlu dilakukan untuk memberikan wawasan yang lebih komprehensif terkait stabilitas jangka panjang dan efisiensi biaya. Sebagai pengembangan arsitektur lebih lanjut, penelitian mendatang direkomendasikan untuk mengeksplorasi pendekatan Hybrid Autoscaling yang menggabungkan HPA berbasis RPS dengan Vertical Pod Autoscaler (VPA). Kombinasi ini diharapkan dapat secara dinamis mengoptimalkan jumlah replika Nginx sekaligus batas sumber daya individu pod (seperti *worker_connections* dan memori) guna memaksimalkan efisiensi klaster secara keseluruhan.

DAFTAR REFERENSI

Ahmad, H., Treude, C., Wagner, M., & Szabo, C. (2024). Smart HPA: A resource-efficient horizontal pod auto-scaler for microservice architectures. In *2024 IEEE 21st International Conference on Software Architecture (ICSA)* (pp. 46–57). IEEE. https://doi.org/10.1109/ICSA59870.2024.00013

Camilo, J., & Campos, N. (2024). *Evaluating the transition from HPA to KEDA in an event-driven architecture deployed in Kubernetes*.

Gao, R., Xie, X., & Guo, Q. (2017). K-TAHP: A Kubernetes load balancing strategy based on TOPSIS+AHP. *XX*.

Gawande, S., & Gorthi, A. (2024). Containerization and Kubernetes: Scalable and efficient cloud-native applications. *International Journal of Innovative Science and Research Technology (IJISRT)*, 435–439. https://doi.org/10.38124/ijisrt/IJISRT24NOV314

Guedes, E. A. C., Silva, L. E. T., & Maciel, P. R. M. (2024). *Performability analysis of I/O bound application on container-based server virtualization cluster*.

HPA Documentation. (n.d.). *Horizontal Pod Autoscaler*. Retrieved March 29, 2025, from https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

Huotari, H. (2025). *NGINX as an API gateway*.

Jónsson, A. J. (2020). *Hybrid auto-scaling for an asynchronous computationally intensive application* [Degree project, Computer Science and Engineering].

Campbell, J. (n.d.). *Kubernetes vs. Docker: A comprehensive guide to containerization*. Atlassian. Retrieved March 29, 2025, from https://www.atlassian.com/microservices/microservices-architecture/kubernetes-vs-docker

Kubernetes Documentation. (n.d.). *Kubernetes documentation*. Retrieved February 26, 2025, from https://kubernetes.io/docs/home/

Lorido-Botran, T., Miguel-Alonso, J., & Lozano, J. A. (2014). A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing, 12*(4), 559–592. https://doi.org/10.1007/s10723-014-9314-7

Malallah, H. S., Qashi, R., Abdulrahman, L. M., Omer, M. A., & Yazdeen, A. A. (2023). Performance analysis of enterprise cloud computing: A review. *Journal of Applied Science and Technology Trends, 4*(1), 1–12. https://doi.org/10.38094/jastt401139

Molleti, R. (2022). Kubernetes advanced auto scaling techniques. *Journal of Mathematical & Computer Applications, 1*(4), 1–4. https://doi.org/10.47363/JMCA/2022(1)E126

Nilsen, J. (2023). *Performance evaluation of Kubernetes autoscaling strategies on GKE clusters*.

Noor, J., Faysal, M. D. B., Amin, M. D. S., Tabassum, B., Khan, T. R., & Rahman, T. (2025). Kubernetes application performance benchmarking on heterogeneous CPU architecture: An experimental review. *High-Confidence Computing, 5*(1), Article 100276. https://doi.org/10.1016/j.hcc.2024.100276

Raj, P., Vanga, S., & Chaudhary, A. (2023). Kubernetes architecture, best practices, and patterns. In *Cloud-native computing: How to design, develop, and secure microservices*

and event-driven applications (pp. 49–70). Wiley. https://doi.org/10.1002/9781119814795.ch3

Šimon, M., Huraj, L., & Búčik, N. (2023). A comparative analysis of high availability for Linux container infrastructures. *Future Internet, 15*(8), Article 253. https://doi.org/10.3390/fi15080253

Microsoft Azure. (n.d.). *What is cloud computing?* Retrieved February 26, 2025, from https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-cloud-computing/

Wirawan King, R. Z., Hari Trisnawan, P., & Yahya, W. (2024). Perbandingan metode autoscaling vertical pod autoscaler dan horizontal pod autoscaler Kubernetes pada Google Cloud Platform. *Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer*.

Zhou, H., & Yong, C. H. (2024). Implement HPA for Nginx service using custom metrics under Kubernetes framework. *IEEE Access, 12*, 189722–189734. https://doi.org/10.1109/ACCESS.2024.3509876