# BAB I
# PENDAHULUAN

## 1.1 Latar Belakang

Dalam konteks layanan digital modern, optimasi *backend* menjadi salah satu cara utama untuk memastikan aplikasi tetap cepat, efisien, dan menyajikan pengalaman pengguna yang baik (Azura Team, 2023). *Backend* merupakan lapisan layanan di sisi server yang mengatur aliran data, mengeksekusi aturan bisnis, dan menjaga keutuhan informasi. Lapisan ini menerjemahkan permintaan klien menjadi operasi terstruktur, sehingga proses berlangsung tertib, aman, dan dapat ditelusuri melalui pencatatan peristiwa serta penanganan kesalahan. Tanggung jawab utama *backend* mencakup penyajian antarmuka layanan, *authentication*, *authorization*, dan pemeliharaan konsistensi data.

Pengembang biasanya menggunakan *framework backend* untuk menyeragamkan cara membangun layanan, sehingga prosesnya lebih cepat, terstruktur, dan mudah dipelihara. *Framework* umumnya sudah menyediakan pengaturan rute dan *middleware*, pengelolaan data, validasi, serta fitur pendukung keamanan dan pemantauan, sehingga pengembang tidak perlu membuat semuanya dari awal. Pemilihan *framework* biasanya dipengaruhi bahasa pemrograman yang dipakai, ketersediaan pustaka, serta kemudahan *deployment*. Melalui pemilihan yang tepat, *backend* yang dibangun dapat diarahkan untuk melayani beban permintaan yang besar dan konsisten. Kemampuan ini menjadi penting ketika aplikasi digunakan pada konteks *e-commerce* dengan jumlah pengguna dan transaksi yang terus meningkat.

Sejalan dengan meningkatnya adopsi digital, perkembangan *e-commerce* menciptakan lonjakan kebutuhan akan sistem informasi dengan performa tinggi dan keandalan. Laporan perilaku konsumen *e-commerce* Indonesia mencatat bahwa nilai transaksi *e-commerce* nasional tumbuh dari sekitar Rp106 triliun pada tahun 2018 menjadi Rp476,3 triliun pada tahun 2022 (Bank Indonesia dalam Kredivo & Katadata Insight Center, 2023). Survei lain menunjukkan bahwa lebih dari 69% konsumen Indonesia berbelanja *online* setidaknya sekali dalam sebulan, yang mencerminkan tingginya aktivitas belanja digital di masyarakat (Standard Insights, 2023). Tingginya aktivitas tersebut menuntut aplikasi *e-commerce* memiliki *backend* yang mampu menangani banyak permintaan secara simultan, sambil menjaga konsistensi data dan kecepatan respons.

Secara global, lanskap *e-commerce* juga didominasi platform siap pakai. Wappalyzer memperkirakan pangsa pasar 2025 berturut-turut sekitar WooCommerce 29%, Shopify 20%, dan Magento 15%, yang menunjukkan dominasi solusi serba jadi pada pasar umum. Namun, di banyak kasus bisnis, sistem *e-commerce* perlu lebih fleksibel, serta mengintegrasikan layanan dengan sistem internal melalui pendekatan *headless* yang memisahkan *frontend* dan *backend*. Kebutuhan seperti ini tidak sepenuhnya dapat dipenuhi oleh platform siap pakai, sehingga pendekatan berbasis *framework backend* tetap diperlukan untuk merancang layanan yang fleksibel dan terintegrasi.

Dalam konteks pengembangan *backend* berbasis Node.js sendiri, terdapat beragam *framework* dengan karakteristik berbeda. Express banyak digunakan, dengan sekitar 17,8% responden survei melaporkan telah melakukan pengembangan intensif dengannya selama setahun terakhir (Stack Overflow Developer Survey, 2024). Di sisi lain, muncul *framework* generasi baru seperti Hono yang dirancang dengan fokus kecepatan dan efisiensi, serta dengan *routing* yang sangat cepat (Sarkar, 2024). Express dijadikan *baseline* karena ekosistemnya luas, sedangkan Hono relevan sebagai pembanding modern yang menekankan efisiensi. Dalam konteks pemilihan *framework*, faktor seperti pengaruh sosial, saran kolega, dan ukuran komunitas juga tercatat sebagai pendorong utama adopsi, bukan semata hasil evaluasi teknis yang terstruktur (Pano, Graziotin, & Abraha*ms*son, 2018). Kondisi ini menunjukkan perlunya kajian empiris yang lebih terarah mengenai kinerja *framework backend* pada skenario *e-commerce*.

Namun demikian, kajian empiris mengenai kinerja *framework backend* pada skenario *e-commerce* masih terbatas. Andreas (2021) membandingkan Laravel dan CodeIgniter pada prototipe *website e-commerce* PT Sinar Agung Prasadikindo, dengan fokus pada pengujian beban HTTP terhadap fitur inti. Selain itu, Saputro dan Novita (2025) juga melakukan analisis perbandingan kinerja antara *framework* Express dan Hono pada aplikasi registrasi sederhana. Studi-studi ini menunjukkan adanya perbedaan kinerja *framework* pada domain aplikasi yang berbeda, khususnya pada metrik seperti waktu respons dan *throughput*. Meskipun demikian, penelitian tersebut belum menilai penggunaan sumber daya seperti CPU dan memori, serta belum sepenuhnya merepresentasikan ekosistem Node.js yang semakin luas diadopsi dalam skala *e-commerce* yang kompleks. Selain itu, pengujian pada salah satu studi tersebut dilakukan pada lingkungan lokal (*localhost*) tanpa menggunakan server terpisah, sehingga mengabaikan aspek latensi jaringan. Ditambah lagi, penelitian tersebut belum memasukkan perbandingan penggunaan antara dua jenis *database*, yaitu SQL dan NoSQL.

Skala beban pada praktik industri menuntut perancangan yang cermat. Laporan Black Friday Cyber Monday milik Shopify mencatat beban puncak sekitar lebih dari 80 juta permintaan per menit pada *app servers* (Shopify, 2024). Namun, tantangan teknis dalam skenario ini bukan sekadar kemampuan server menerima jutaan permintaan, melainkan kemampuannya menjaga waktu respons tetap singkat di bawah ambang batas toleransi pengguna. Hal ini didefinisikan sebagai pemenuhan *Service Level Objective* (SLO). Sistem yang mampu menangani ribuan pengguna tetapi dengan latensi yang melambat drastis akan dianggap gagal memberikan layanan yang baik. Sejalan dengan panduan arsitektur modern, pemilihan *database* disesuaikan dengan karakteristik data dan pola akses. Pendekatan *purpose-built database* mendorong penggunaan lebih dari satu jenis *database* dalam satu sistem untuk menyesuaikan dengan pola beban yang berbeda (AWS, 2023). Dalam konteks *e-commerce*, praktik umum yang muncul adalah penggunaan *database* relasional untuk transaksi yang menuntut konsistensi kuat, dan *database* NoSQL untuk komponen dengan *throughput* tinggi seperti katalog dan keranjang. Metrik yang digunakan yaitu latensi termasuk persentil seperti p95 *latency*, *throughput* atau

*Request Per Second* (RPS), *error rate*, serta saturasi sumber daya yang tercermin pada penggunaan CPU dan RAM (Beyer, Jones, Petoff, & Murphy, 2017). Selain itu, ukuran *payload* berpengaruh langsung pada waktu respons dan biaya transfer, sehingga perancangan API dianjurkan menerapkan pengurangan *payload* dan *pagination* (Chrome for Developers, 2019).

Urgensi pemilihan *framework* yang tepat terlihat pada dampaknya terhadap biaya dan kualitas layanan. Pengambilan keputusan yang tidak berbasis data berisiko meningkatkan kebutuhan sumber daya komputasi, membebani biaya infrastruktur, serta mengakibatkan pelanggaran *Service Level Objective* (SLO) yang berdampak negatif pada pengalaman pengguna. Sebaliknya, pemilihan *framework* yang efisien dapat meminimalkan konsu*ms*i CPU dan memori, menjaga stabilitas latensi saat beban meningkat, serta mengendalikan biaya operasional.

Berdasarkan kebutuhan dan kesenjangan tersebut, penelitian ini memfokuskan evaluasi pada lima aspek teknis dalam skenario *e-commerce*, yaitu integrasi *database* SQL dan *NoSQL*, jenis operasi *endpoint* (baca dan tulis), tingkat beban pengguna, ukuran *payload*, serta penggunaan sumber daya, dengan pengukuran berupa *tail latency*, *throughput*, *error rate*, serta penggunaan CPU dan RAM. Secara umum, penelitian ini diharapkan memberikan dua kontribusi utama. Pertama, kontribusi akademis berupa model evaluasi *framework backend* berbasis metrik teknis yang dapat dilanjutkan pada studi sejenis. Kedua, kontribusi berupa rekomendasi bagi pengembang dan perusahaan dalam memilih *framework backend* sesuai dengan kebutuhan performa dan penggunaan daya pada sistem *e-commerce*.

### 1.2 Rumusan Masalah

Berdasarkan latar belakang yang telah diuraikan, maka rumusan masalah yang dapat diambil adalah sebagai berikut:

1. Bagaimana perbandingan performa Express dan Hono pada masing-masing tipe *database,* yaitu: Express–SQL vs Hono–SQL dan Express–NoSQL vs Hono–NoSQL?
2. Bagaimana pengaruh jenis operasi *endpoint* baca dan tulis terhadap performa masing-masing *framework*?
3. Bagaimana kapasitas penanganan beban kedua *framework* dalam mempertahankan standar performa (SLO) saat jumlah pengguna meningkat?
4. Bagaimana perbedaan performa terhadap variasi ukuran *payload* pada kedua *framework*?
5. Bagaimana penggunaan sumber daya dari Express dan Hono pada skenario pengujian tersebut?

### 1.3 Batasan Masalah

Untuk memperjelas cakupan penelitian sesuai dengan tujuan yang ingin dicapai, penelitian ini akan membatasi ruang lingkup masalah sebagai berikut:

1. *Database* dibatasi pada PostgreSQL (SQL) dan MongoDB (NoSQL).

2. *Framework* yang dibandingkan hanya Express dan Hono.
3. *Dataset* dibatasi pada Northwind (varian SQL dan NoSQL) dengan struktur data yang representatif untuk skenario *e-commerce*.
4. Lingkungan eksekusi SUT menggunakan VPS 2 vCPU, 2 GB RAM, 20 GB *storage* dengan Docker.
5. Generator beban dijalankan dari mesin lokal terpisah melalui internet publik dengan *bandwidth* ±5 Mbps
6. Pengujian beban difokuskan pada pemenuhan target kinerja, bukan *stress testing*. Mengacu pada model RAIL (Google Developers, 2020), *Service Level Objective* (SLO) untuk P95 *latency* ditetapkan lebih kecil sama dengan 1 detik.

## 1.4 Tujuan Penelitian

Berdasarkan rumusan masalah di atas, maka tujuan dari penelitian ini adalah sebagai berikut:

1. Membandingkan performa Express dan Hono saat diintegrasikan dengan SQL dan NoSQL menggunakan metrik p95 *latency*, *throughput*, dan *error rate*.
2. Menilai pengaruh jenis operasi *endpoint* baca dan tulis terhadap performa masing-masing *framework*.
3. Mengevaluasi stabilitas kinerja kedua *framework* terhadap peningkatan beban pengguna secara bertahap, ditinjau dari kepatuhan terhadap batas SLO.
4. Menganalisis dampak variasi ukuran *payload* terhadap kinerja layanan pada kedua *framework*.
5. Mengevaluasi penggunaan sumber daya (CPU dan RAM) pada berbagai skenario pengujian di atas.

## 1.5 Landasan Teori

### 1.5.1 Karakteristik Beban Kerja *E-commerce*

*E-commerce* umumnya memiliki pola akses yang tidak seimbang antara baca dan tulis. Aktivitas penelusuran dan pencarian katalog cenderung *read-heavy*, sedangkan proses menambah produk ke keranjang, *checkout*, pembayaran, dan pembaruan stok bersifat *write-heavy*. Pola ini tercermin pada daftar pola akses toko *online* seperti pengambilan produk, inventori, pelanggan, pesanan, pembayaran, dan pengiriman, yang menuntut jalur baca cepat sekaligus operasi tulis yang konsisten.

Beban lalu lintas *e-commerce* juga sangat fluktuatif dan sering mengalami lonjakan ekstrem saat promosi besar, misalnya Black Friday/Cyber Monday. Lonjakan ini menuntut kemampuan penanganan permintaan serentak tanpa mengorbankan pengalaman pengguna. Laporan resmi Shopify mencatat rekor beban dan *throughput* sistem selama periode BFCM, menegaskan sifat musiman sekaligus ekstremnya beban kerja. *(Shopify, 2024)*.

Karakter lain yang menonjol adalah sensitivitas terhadap *tail latency* dan ukuran *payload*. Praktik rekayasa kinerja merekomendasikan pengendalian ukuran data

yang dikirimkan API dipengaruhi oleh *payload,* sehingga desain API yang baik juga menganjurkan paginasi untuk koleksi data seperti daftar produk agar respons tetap ringan dan stabil. *(Chrome for Developers, 2019; Microsoft, 2025)*.

Akhirnya, variabilitas beban dan ekspektasi pengalaman pengguna menuntut pemantauan yang berfokus pada metrik inti layanan. Literatur Site Reliability Engineering merekomendasikan empat *golden signals* untuk layanan pengguna seperti latensi, *trafik/throughput, error*, dan saturasi sumber daya. Metrik ini lazim dipakai untuk mengevaluasi dan menjaga kualitas layanan pada sistem *e-commerce* berskala besar. *(Beyer et al., 2017)*.

### 1.5.2 Ekosistem Node.js pada Express dan Hono

Node.js merupakan lingkungan eksekusi (*runtime environment*) JavaScript di sisi server yang dibangun di atas mesin JavaScript V8 milik Google Chrome. Arsitektur Node.js berbeda dengan model *multi-threaded* tradisional karena menggunakan model *Single-Threaded Event Loop* dan *Non-blocking* I/O. Dalam mekanisme ini, Node.js tidak membuat *thread* baru untuk setiap permintaan yang masuk, melainkan menangani ribuan koneksi secara bersamaan dalam satu proses utama (Node.js Foundation, n.d.).

Ketika permintaan masuk melibatkan operasi *Input*/*Output* (I/O) yang lambat, seperti mengambil data dari *database* atau memanggil API eksternal, Node.js tidak akan memblokir proses untuk menunggu operasi tersebut selesai. Sebaliknya, operasi tersebut dilempar ke *background*, dan *Event Loop* segera beralih menangani permintaan lain. Setelah operasi I/O selesai, hasilnya dikembalikan ke antrean untuk diproses lebih lanjut. Karakteristik ini menjadikan Node.js sangat efisien untuk aplikasi *e-commerce* yang intensif data (*data-intensive*) dan memiliki lalu lintas tinggi (MDN Web Docs, 2025).

1. **Express**
   Express termasuk salah satu *web framework* paling populer pada ekosistem Node.js. Survei Stack Overflow (2024) menempatkannya dalam kelompok *framework* yang paling sering digunakan. Express menawarkan struktur yang ringkas dengan konsep utama *routing* dan *middleware*, sehingga pengembang dapat menyusun rantai pemrosesan permintaan secara bertahap, misalnya menambahkan *authentication*, *validation*, lalu pengiriman respons. Sifatnya yang minimalis dan dukungan paket yang sangat luas menjadikan Express sering dipakai sebagai fondasi layanan HTTP di Node.js (Express.js, n.d.; Stack Overflow, 2024).
2. **Hono**
   Hono dikembangkan sebagai *framework* yang lebih ringan dan berfokus pada kinerja. Hono mengikuti *web standards* dan dapat berjalan pada berbagai *runtime* modern seperti Cloudflare Workers, Deno, Bun, maupun Node.js, sehingga dapat digunakan pada skenario *edge* maupun server konvensional. Ukuran inti yang kecil, API yang seragam, dan orientasi pada kecepatan membuat Hono relevan dijadikan pembanding terhadap *framework* yang sudah mapan seperti Express (Hono Docs, n.d.).

### 1.5.3 Paradigma *Database* SQL Dan NoSQL

Pemilihan *database* biasanya mengikuti bentuk data dan cara data itu diakses. Panduan arsitektur seperti AWS Well-Architected *Database* (2023) menekankan prinsip *purpose-built*, yaitu setiap jenis data sebaiknya disimpan pada mesin penyimpanan yang paling sesuai agar kinerja, konsistensi, dan *throughput* tetap terjaga. Pada sistem transaksi seperti *e-commerce*, hal ini penting karena ada bagian yang sangat sensitif terhadap keakuratan seperti pada pesanan dan pembayaran, tetapi ada juga bagian yang utamanya mengejar kecepatan baca contohnya katalog dan keranjang. *Dataset* dan *endpoint* disejajarkan fungsinya, bukan disamakan skema penyimpanannya, karena model data berbeda secara prinsip.

*Database* relasional memakai *Structured Query Language (SQL)* untuk mendefinisikan, mengakses, dan mengelola data. Data disimpan dalam bentuk tabel, setiap tabel memiliki kolom dan tipe data yang jelas, dan berbagai tabel dapat dihubungkan dengan *foreign key*. PostgreSQL juga mendukung transaksi sehingga perubahan data bisa dijalankan secara andal dan konsisten (PostgreSQL Global Development Group, 2025). Model relasional seperti ini cocok untuk data yang saling berhubungan dan membutuhkan konsistensi tinggi, misalnya modul pesanan, pembayaran, pelanggan, dan pengiriman pada *e-commerce.*

Sebaliknya, *database NoSQL* dengan model *key-value* atau dokumen biasanya dioptimalkan untuk dibaca sangat cepat dan dapat diperbesar secara horizontal dengan menambah *node*. Skemanya lebih longgar sehingga tabel atau koleksi tidak harus punya kolom yang selalu sama (MongoDB, 2024). Pola seperti ini cocok untuk bagian *e-commerce* yang sering dibaca berulang-ulang oleh banyak pengguna, misalnya katalog produk, daftar kategori, atau isi keranjang.


**Gambar 1**. Perbandingan SQL dan NoSQL.
Sumber*: SQL vs NoSQL, Astera Analytics Team, 12 Maret 2025.*

### 1.5.4 Prinsip Benchmarking

*Benchmarking* bertujuan menghasilkan perbandingan kinerja sistem yang adil, terisolasi, dan representatif. Dalam konteks pengujian beban layanan *backend*, literatur *performance testing* membedakan dua model utama pemberian beban, yaitu *closed workload model* dan *open workload model*. Pada model *closed*, iterasi pengguna virtual berikutnya dijalankan setelah iterasi sebelumnya selesai, sehingga

jumlah pengguna serentak dapat dikendalikan secara eksplisit. Pendekatan ini umum digunakan ketika tujuan pengujian adalah membandingkan efisiensi beberapa sistem pada tingkat beban yang identik dan terkontrol.

Sebaliknya, pada model *open*, permintaan dikirimkan secara independen dari penyelesaian permintaan sebelumnya. Pola ini lebih mendekati karakteristik lalu lintas internet pada sistem *e-commerce* nyata, namun memiliki tingkat variabilitas yang lebih tinggi dan lebih sulit dikendalikan. Ketika sumber daya pengujian terbatas atau jaringan mengalami kejenuhan, model ini berpotensi menimbulkan bias pengukuran. Oleh karena itu, dalam studi komparatif yang berfokus pada perbandingan efisiensi sistem, model *closed* sering dipilih karena kemampuannya mengisolasi variabel dan menjaga konsistensi beban antar skenario, sementara model *open* lebih sesuai digunakan pada studi lanjutan yang menekankan realisme lalu lintas produksi.

Selain pemilihan model beban, validitas *benchmarking* sangat bergantung pada teknik mitigasi variabilitas hasil yang disebabkan oleh faktor stokastik, seperti aktivitas latar belakang, penjadwalan sistem operasi, serta mekanisme *garbage collection*. Untuk menangani ketidakpastian ini, standar industri SPEC CPU 2017 mewajibkan pengujian dilakukan dengan minimal tiga iterasi $n \geq 3$ dan pelaporan hasil menggunakan nilai tengah (median), bukan rata-rata (*mean*). Pendekatan ini didukung oleh David J. Lilja (2000) dalam *Measuring Computer Performance*, yang menyatakan bahwa rata-rata terlalu sensitif terhadap *outlier* atau gangguan sesaat pada sistem komputer, sedangkan median memberikan estimasi tendensi sentral yang lebih kekar (*robust*). Validitas penggunaan median pada jumlah sampel kecil ini juga berlandaskan pada prinsip *Exploratory Data Analysis* dari John W. Tukey (1977) mengenai *resistance*, di mana median dari tiga titik data berfungsi sebagai mekanisme *smoothing* yang efektif untuk mengeliminasi anomali ekstrem (seperti *lag* mendadak) tanpa mendistorsi kinerja representatif sistem yang sebenarnya.

### 1.5.5 Indikator dan Prosedur Pengukuran

Evaluasi kinerja layanan *backend* umumnya berfokus pada sejumlah metrik yang merepresentasikan pengalaman pengguna dan kapasitas sistem. Literatur *Site Reliability Engineering* memperkenalkan empat *golden signals*, yaitu latensi, *throughput*, tingkat kesalahan (*error rate*), dan sumber daya. Keempat metrik ini dipandang cukup representatif untuk menggambarkan kualitas layanan dan kondisi operasional sistem secara menyeluruh.

Latensi tidak hanya dilaporkan dalam bentuk nilai rata-rata, tetapi juga sebagai persentil, seperti p95 atau p99, untuk menangkap fenomena *tail latency*. Nilai persentil tinggi mencerminkan pengalaman sebagian pengguna yang menerima respons paling lambat dan sering kali menjadi faktor penentu persepsi kualitas layanan. Ketergantungan pada nilai rata-rata atau median saja dapat menutupi perilaku ekor distribusi latensi, sehingga sistem tampak stabil meskipun sejumlah permintaan mengalami keterlambatan signifikan.

*Throughput* didefinisikan sebagai jumlah permintaan yang berhasil diproses per satuan waktu, umumnya dinyatakan dalam *requests per second* (RPS). Metrik ini

merepresentasikan kapasitas sistem dalam melayani permintaan dan perlu dihitung secara konsisten pada seluruh skenario pengujian agar hasil dapat dibandingkan secara adil.

Selain metrik yang diamati dari sisi klien, evaluasi kinerja *backend* juga memerlukan pemantauan penggunaan sumber daya di sisi server, khususnya CPU dan memori. Pengamatan sumber daya ini penting untuk memahami hubungan antara perubahan latensi atau *throughput* dengan kondisi beban komputasi yang dialami sistem. Dalam lingkungan berbasis kontainer, pemantauan sumber daya umumnya dilakukan melalui mekanisme yang menyediakan data deret waktu penggunaan CPU dan memori selama pengujian, sehingga dinamika kinerja dan konsu*ms*i sumber daya dapat dianalisis secara bersamaan.

### 1.5.6 Konfigurasi Beban dan Kriteria Keberhasilan

Kriteria keberhasilan pengujian kinerja layanan *backend* umumnya ditentukan berdasarkan ambang batas latensi yang masih dapat diterima oleh pengguna. Salah satu pendekatan yang banyak digunakan adalah menetapkan batas maksimum untuk latensi persentil ke-95 (p95). Model RAIL menyatakan bahwa respons dengan latensi melebihi 1.000 milidetik cenderung menyebabkan pengguna kehilangan fokus terhadap interaksi yang sedang dilakukan. Oleh karena itu, nilai p95 di bawah ambang tersebut sering dijadikan indikator bahwa layanan masih berada pada tingkat kualitas yang dapat diterima.

Untuk mengevaluasi stabilitas kinerja sistem, beban pengujian biasanya diberikan secara bertahap. Pendekatan peningkatan beban secara progresif memungkinkan pengamatan bagaimana sistem merespons pertambahan permintaan, mengidentifikasi rentang beban di mana layanan masih stabil, serta mendeteksi titik awal terjadinya degradasi kinerja. Pemberian beban secara bertahap juga membantu menjaga keadilan perbandingan antar skenario dengan memastikan bahwa setiap sistem diuji pada kondisi beban yang sebanding.

Selain metrik utama seperti latensi, *throughput*, dan tingkat kesalahan, analisis kinerja dapat diperluas dengan metrik turunan yang mengaitkan kinerja dengan konsu*ms*i sumber daya. Salah satu pendekatan adalah menganalisis waktu CPU per permintaan, yang menggambarkan biaya komputasi rata-rata yang dibutuhkan untuk melayani satu permintaan. Pendekatan lain adalah mengukur efisiensi memori dengan mengaitkan *throughput* terhadap puncak penggunaan memori selama pengujian. Metrik-metrik turunan ini memberikan sudut pandang tambahan mengenai efisiensi sistem dan relevan dalam konteks optimasi biaya serta pemilihan teknologi *backend* yang hemat sumber daya.

# BAB II
# METODE PENELITIAN

## 2.1 Waktu dan Lokasi Penelitian

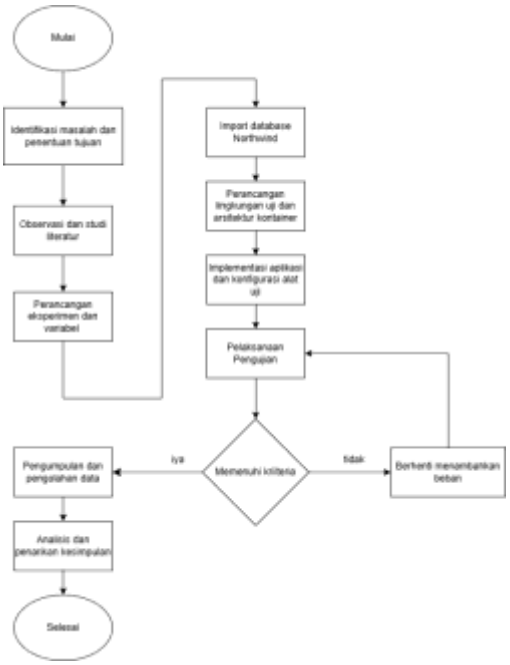
Penelitian ini dilaksanakan mulai dari minggu pertama bulan Juli 2025 hingga pertengahan Oktober 2025. Lokasi penelitian dilakukan di Lab Rekayasa Perangkat Lunak (RPL) Universitas Hasanuddin Makassar.

**Tabel 1.** Jadwal Kegiatan Penelitian

| No | Tahapan Penelitian | Juli | | | | Agustus | | | | September | | | | Oktober | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 |
| 1 | Observasi | | | | | | | | | | | | | | | | |
| 2 | Studi Literatur | | | | | | | | | | | | | | | | |
| 3 | Desain | | | | | | | | | | | | | | | | |
| 4 | Implementasi | | | | | | | | | | | | | | | | |
| 5 | Testing | | | | | | | | | | | | | | | | |
| 6 | Analisis | | | | | | | | | | | | | | | | |

## 2.2 Desain Penelitian

Penelitian ini menggunakan eksperimen komparatif berbasis *benchmark* untuk menilai perbedaan kinerja dan penggunaan sumber daya antara dua *framework backend* pada ekosistem Node.js, yaitu Express dan Hono, dalam skenario *e-commerce.*




**Gambar 2.** Diagram alur penelitian

### 2.3 Variabel Penelitian

Variabel independen adalah faktor yang sengaja diubah atau diatur, kemudian diamati dampaknya. Pada studi ini, variabel independen direkayasa melalui pemilihan teknologi, jenis operasi, tingkat beban pengguna, dan ukuran *payload* yang telah ditetapkan dalam rancangan uji.

Variabel dependen adalah hasil yang diukur karena adanya perubahan variabel independen. Pada konteks *benchmarking backend*, variabel dependen berupa metrik performa dan penggunaan sumber daya, misalnya *throughput*, p95 *latency*, *error rate*, serta konsu*ms*i CPU dan memori.

Variabel kontrol adalah kondisi yang diseragamkan agar pengaruh variabel independen terhadap variabel dependen dapat diamati secara adil. Kontrol meliputi versi perangkat lunak, konfigurasi sistem, spesifikasi perangkat keras, dan parameter pengujian yang konsisten.

#### 2.3.1 Variabel Independen

Variabel independen dalam penelitian ini meliputi pemilihan *framework backend,* jenis operasi *endpoint*, tingkat beban pengguna, dan ukuran *payload*

#### 2.3.2 Variabel Dependen

- *Throughput* (*req/s*): jumlah permintaan yang diproses per detik.
- p95 *latency* (*ms*): waktu respons persentil ke-95 (indikator *tail latency*).
- *Error rate* (%): proporsi respons gagal terhadap total permintaan.
- CPU App: waktu CPU kumulatif kontainer aplikasi.
- CPU DB: waktu CPU kumulatif kontainer *database.*
- Mem App: penggunaan memori puncak kontainer aplikasi.
- Mem DB: penggunaan memori puncak kontainer *database.*
- Metrik turunan *resource*:
  1. **Waktu CPU per permintaan (*ms/req*)** adalah rata-rata waktu CPU gabungan aplikasi dan *database* yang dihabiskan untuk melayani satu permintaan, semakin rendah semakin efisien.
  2. **Rasio Efisiensi Memori (*Rps/GiB*)** merupakan rasio antara *throughput* dan total puncak penggunaan memori gabungan. Semakin tinggi nilainya, semakin efisien penggunaan memori.

#### 2.3.3 Variabel Kontrol

1. Lingkungan eksekusi memakai VPS yang sama: 2 vCPU, 2 GB RAM, 20 GB *storage*.
2. Docker 28.4.0 dan Docker Compose v2.39.4.
3. *Database* dikunci pada PostgreSQL 16.10 dan MongoDB 7.0.24.
4. *Reverse proxy* yang dipakai satu jenis, yaitu Nginx 1.27.5.
5. Aplikasi dikembangkan dengan Node.js 20.15.0.
6. Pengujian menggunakan Apache JMeter v5.6.3

7. Seluruh skenario JMeter dijalankan dengan pola kenaikan jumlah *threads* yang sama, dengan parameter *ramp-up* 10 detik dan durasi pengujian 60 detik (termasuk fase *ramp-up*).

### 2.4 *Dataset* Dan Skenario Data

Penelitian menggunakan *dataset* Northwind karena strukturnya sudah menggambarkan alur dasar *e-commerce*, yaitu ada daftar produk yang dijual, data pelanggan yang melakukan pemesanan, pesanan yang berisi detail item, pemasok yang menyediakan barang, serta informasi pengiriman pesanan ke pelanggan. *Dataset* ini juga tersedia dalam dua bentuk, yakni relasional (SQL) dan dokumen (NoSQL), sehingga bisa dipakai untuk membandingkan kedua pendekatan *database* tersebut.

Skala *dataset* Northwind tergolong menengah, di mana setiap entitas umumnya berisi ratusan hingga beberapa ribu baris atau dokumen. Ukuran ini dinilai cukup untuk menguji operasi baca dan tulis pada sistem *e-commerce* tanpa menimbulkan beban ekstrem yang dapat mendominasi hasil pengujian. Pada beberapa konversi atau repositori, jumlah data antara varian SQL dan NoSQL dapat sedikit berbeda, misalnya akibat penyesuaian tipe data atau penggabungan detail ke dalam struktur dokumen. Meskipun demikian, struktur dan semantik data tetap sebanding sehingga perbandingan kinerja dapat dilakukan secara adil.

Untuk pengujian variasi ukuran *payload*, struktur data tambahan disiapkan secara terpisah, yaitu dalam bentuk tabel tambahan pada SQL dan koleksi tambahan pada NoSQL. Pendekatan ini memungkinkan pengaturan ukuran data yang dikirimkan tanpa memodifikasi isi *dataset* Northwind utama. Seluruh pengujian menjaga kesetaraan isi data antara varian SQL dan NoSQL. *Dataset* SQL dimuat dari berkas skrip northwind.sql, sedangkan varian NoSQL dimuat dari berkas JSON untuk setiap koleksi.

#### 2.4.1 Northwind Sql

Northwind menampilkan relasi yang jelas antar entitas, misalnya keterkaitan *orders* dengan *order_details*, *customers*, *shippers*, dan *employees*. Karakter relasional ini membuat *Northwind* cocok untuk mempraktikkan operasi baca sederhana serta *query* kompleks yang memerlukan JOIN. Pada varian relasional, jumlah baris entitas inti adalah *orders* 830, *order_details* 2.155, *customers* 91, *products* 77, *employees* 9, *suppliers* 29, *shippers* 6, *categories* 8, disertai tabel region dan *territories*. Angka tersebut mencerminkan *dataset* berukuran moderat yang dipakai untuk studi dan uji kinerja.

#### 2.4.2 Northwind Nosql

Varian dokumen yang digunakan dirujuk dari repositori *jasny/mongodb-northwind*. Beberapa penyesuaian dilakukan, misalnya kunci utama disimpan dalam *field* _id, baris detail pesanan disisipkan ke dalam dokumen *orders.details*, dan detail pembelian disisipkan ke dalam dokumen *purchase_orders.details*. Jumlah datanya lebih kecil daripada varian SQL, seperti *customers* berisi 23 dokumen, *employees*

tetap 9, *suppliers* 10, *shippers* 3, *inventory_transaction_types* 4, dan koleksi transaksi persediaan berada pada rentang sekitar 36 sampai 135 entri. Koleksi seperti *orders* dan *purchase_orders* juga memuat daftar dokumen yang konsisten dengan contoh tersebut. Meskipun jumlah dan struktur entitas pada varian dokumen lebih ringkas, perbandingan tetap dapat dilakukan secara adil karena fokus pengujian berada pada perilaku *endpoint* yang setara, misalnya membaca satu order lengkap atau mengambil daftar produk, bukan pada kesamaan mutlak jumlah baris di seluruh *database.* Perbedaan ukuran *dataset* dinyatakan sejak awal, sehingga interpretasi hasil uji tetap berada dalam konteks masing-masing varian.

### 2.5 Lingkungan Uji

Pengujian memisahkan *System Under Test* (SUT) dari generator beban. SUT terdiri dari dua pasangan uji yaitu Express pada SQL vs Hono pada SQL, dan Express pada NoSQL vs Hono pada NoSQL. Seluruh layanan dijalankan dalam kontainer di VPS. Generator beban menggunakan Apache JMeter di laptop penguji melalui jaringan publik menuju VPS. Pemisahan ini mencegah persaingan sumber daya antara SUT dan alat uji, menekan risiko *bottleneck* pada mesin uji, dan menjaga konsistensi pengukuran.

#### 2.5.1 Spesifikasi Perangkat Keras Dan Sistem Operasi

a. Mesin Lokal (Load Generator & Dev) – Laptop
   - OS Windows 11 Home
   - Perangkat HP, Intel Core i5 Gen-13, RAM 16 GB
   - Perangkat lunak utama Apache JMeter (pengujian beban), Visual Studio Code (pengembangan), Brave browser (uji fungsional singkat & pemantauan *resource*)

b. SUT – VPS (*Host* Docker)
   - Spesifikasi 2 vCPU, 2 GB RAM, 20 GB *storage*
   - Menjalankan beberapa kontainer untuk layanan aplikasi dan *database*

c. Kondisi Jaringan

   Selama pengujian, beban dikirim dari Apache JMeter di laptop penguji melalui jaringan lokal, melewati penyedia layanan internet, lalu masuk ke jaringan publik dan mencapai VPS. Di VPS, Nginx berperan sebagai *reverse proxy* yang menerima koneksi dari klien, meneruskan permintaan ke aplikasi *backend*, serta membantu menata lalu lintas dan antrean koneksi agar aliran tetap stabil. Permintaan diarahkan ke alamat IP publik VPS. Sepanjang sesi pengujian, *bandwidth* jaringan efektif berada pada kisaran sekitar 5 Mbps berdasarkan pengamatan sebelum eksekusi setiap skenario.


**Gambar 3.** Alur Jaringan Testing.
*Sumber: BotPenguin, Nginx*

### 2.5.2 Spesifikasi Server Dan Alokasi Resource

Lingkungan pengujian dijalankan pada VPS dengan spesifikasi 2 *vCPU*, 2 GB *RAM*, dan 20 GB *storage*. Alokasi sumber daya pada tiap kontainer aplikasi dan *database* dibatasi, karena sebagian kapasitas VPS disisihkan untuk kontainer pemantauan yang digunakan selama pengujian, yaitu Prometheus, cAdvisor, dan Grafana. Dengan cara ini, setiap layanan utama seperti aplikasi, *database*, dan *reverse proxy* diuji pada batas yang jelas dan konsisten, sementara pemantauan tetap berjalan.

Nilai pembatasan sumber daya ditetapkan untuk memastikan setiap layanan berjalan dalam kondisi terbatas yang realistis dan konsisten antar skenario, serta untuk mencegah satu komponen mendominasi sumber daya VPS selama pengujian.

Penting untuk dicatat bahwa seluruh pengujian dalam penelitian ini dilakukan secara eksklusif pada *runtime environment* Node.js. Meskipun dokumentasi resmi Hono menyatakan bahwa *framework* ini dapat mencapai kinerja optimal pada *runtime* modern seperti Bun, Deno, atau *Cloudflare Workers*, penelitian ini membatasi lingkup pengujian pada ekosistem Node.js. Pembatasan ini dilakukan untuk menjaga validitas komparasi seimbang.

**Tabel 2.** Kontainer dan alokasi *resource*

| Komponen | CPU limit | Memori limit | I/O disk | Konfigurasi lain |
|---|---|---|---|---|
| Aplikasi Express, Hono | 0.45 | 320m | 10 MB/s | DB_POOL_MAX: "20", JSON_LIMIT: 2mb |
| *Database* PostgreSQL dan MongoDB | 1.00 | 1.1g | 30 MB/s | — |
| *Reverse proxy* (nginx) | 0.05 | 64m | — | Mount./nginx/default.conf, port publik 80:80 |

### 2.6 Perangkat Dan Instrumentasi

#### 2.6.1 Apache JMeter

Apache JMeter digunakan sebagai *load generator* untuk mengirimkan beban HTTP menuju titik masuk (*entry point*) Nginx yang kemudian meneruskan permintaan ke layanan aplikasi. Instrumen ini sekaligus merekam metrik sisi klien, meliputi *throughput* (*req/s*), latensi (termasuk p95), *error rate*, dan jumlah sampel per skenario. Rencana uji (*Test Plan*) disusun menggunakan model *closed workload* (*Thread* Group), *HTTP Request Sampler* untuk setiap *endpoint*, serta *HTTP Header Manager* yang seragam. Pengujian dijalankan menggunakan antarmuka grafis (GUI) JMeter untuk memudahkan pemantauan *real-time*. Untuk meminimalkan *overhead* di sisi klien akibat penggunaan GUI, elemen pengujian (seperti *listeners*) dibuat sesederhana mungkin. Seluruh data hasil pengujian disimpan dalam berkas CSV untuk kemudian diolah guna mendapatkan nilai ringkasan p95, *throughput*, dan *error rate*.

#### 2.6.2 Pemantauan Sumber Daya

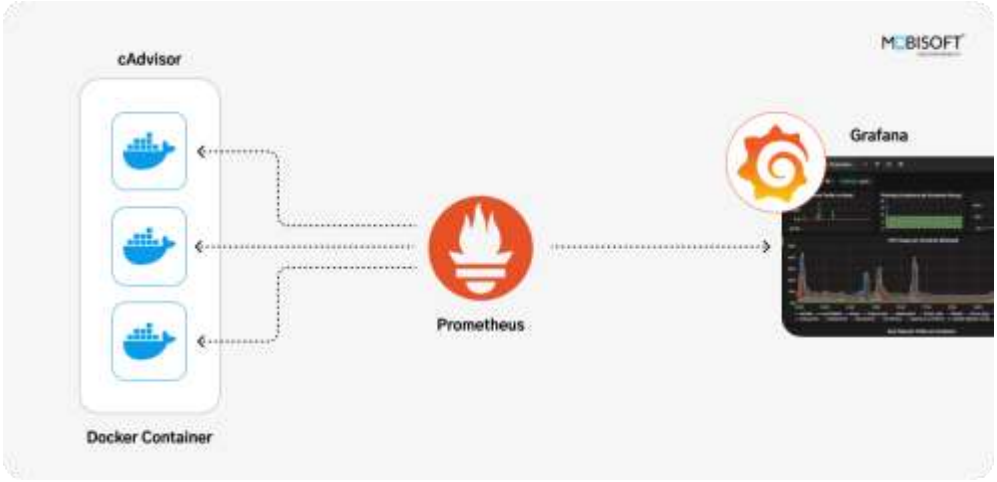
Pemantauan sumber daya sisi server (*server-side monitoring*) dilakukan untuk melengkapi metrik sisi klien yang diperoleh dari pengujian beban. Pendekatan ini memungkinkan analisis hubungan antara perubahan metrik kinerja, seperti *throughput* dan latensi, dengan kondisi penggunaan sumber daya komputasi pada aplikasi dan *database.* Dengan demikian, evaluasi efisiensi *framework* tidak hanya didasarkan pada waktu respons, tetapi juga pada biaya sumber daya yang dikeluarkan untuk melayani permintaan.

Pemantauan dilakukan terhadap seluruh kontainer layanan aplikasi dan *database* dengan fokus pada penggunaan CPU dan memori. Proses pemantauan berjalan bersamaan dengan eksekusi pengujian beban agar data waktu antara metrik kinerja dan metrik sumber daya dapat disejajarkan secara akurat. Pendekatan ini memastikan bahwa perubahan performa dapat dikaitkan langsung dengan kondisi sumber daya pada interval waktu yang sama.

Pengambilan metrik sumber daya dilakukan melalui mekanisme pemantauan kontainer, di mana cAdvisor berperan dalam mengumpulkan data penggunaan CPU dan memori dari setiap kontainer. Data tersebut kemudian dikumpulkan secara berkala dan disimpan sebagai deret waktu oleh sistem pemantauan, serta divisualisasikan menggunakan Grafana untuk keperluan inspeksi dan analisis lanjutan. Alur ini memungkinkan observasi perubahan penggunaan sumber daya sepanjang durasi pengujian secara konsisten.

Dalam analisis, CPU diperlakukan sebagai sumber daya berbasis kerja (*work-based resource*), sehingga penggunaan CPU aplikasi dan *database* diakumulasikan dalam satuan waktu CPU selama jendela pengamatan yang sama. Sebaliknya, memori diperlakukan sebagai sumber daya berbasis kapasitas (*capacity-based resource*), sehingga penggunaan memori direpresentasikan oleh nilai puncak selama jendela pengamatan, bukan akumulasi lintas waktu. Pendekatan ini mencerminkan karakteristik masing-masing sumber daya dalam konteks evaluasi kinerja sistem.

Metrik mentah yang dicatat meliputi penggunaan CPU dan memori pada kontainer aplikasi dan *database.* Berdasarkan metrik tersebut, indikator turunan seperti waktu CPU per permintaan dan efisiensi memori dihitung pada tahap analisis. Perhitungan dilakukan dengan mengaitkan data konsu*ms*i sumber daya server dengan metrik performa yang diperoleh dari pengujian beban, sehingga hubungan antara kinerja layanan dan efisiensi penggunaan sumber daya dapat dianalisis secara kuantitatif.




**Gambar 4.** Alur pemantauan Kontainer Menggunakan cAdvisor, Prometheus, dan Grafana.
*Sumber: Mobisoft Infotech, artikel oleh Pranay Lande, 21 Februari 2025*

### 2.7 Rancangan Skenario Uji

Rancangan skenario uji dibuat agar setiap kombinasi diuji pada pola beban yang sama sehingga hasilnya dapat dibandingkan langsung. Beban *e-commerce* umumnya didominasi permintaan membaca data, namun operasi tulis tetap disertakan untuk menilai konsistensi dan latensi. Detail aturan beban, tingkat variasi jumlah pengguna, ukuran *payload*, pengendalian bias, dan prosedur pengumpulan data dijelaskan pada poin berikut.

1. **Operasi Read/Write**
   Skenario operasi baca dirancang untuk merepresentasikan variasi beban kerja nyata, mulai dari pengambilan data sederhana hingga akses data individual. Simulasi penelusuran katalog dilakukan melalui *endpoint* GET /products yang menguji efisiensi *query filtering* dan mekanisme paginasi standar. Selain itu, pola akses data acak (*random access*) ke baris tertentu dievaluasi menggunakan *endpoint* GET /customers/_at/:idx, yang merepresentasikan permintaan data pelanggan secara individual.

   Untuk mengukur kinerja pada beban komputasi yang lebih berat, *endpoint* GET /orders/_at/:idx digunakan sebagai representasi halaman detail pesanan. Berbeda dengan oerasi baca sebelumnya, *endpoint* ini melibatkan eksekusi *multi-table JOIN* yang menghubungkan tabel *orders*,

*order_details*, dan *products*, serta melakukan transformasi data relasional menjadi struktur *JSON array* langsung di lapisan basis data.

2. **Variasi Beban Pengguna**
   Tingkat pengguna dinaikkan bertahap dua kali dari beban sebelumnya. Nilai awal 20 *thread* dipilih dengan mengacu pada rekomendasi ukuran koneksi maksimum pada pustaka *node-postgres*, yang menyarankan batas kisaran 20–25 koneksi aktif untuk menjaga efisiensi dan stabilitas akses basis data pada aplikasi Node.js (node-postgres documentation, 2024). Parameter *ramp-up* dan durasi uji dibuat sama pada setiap tingkat. Kenaikan ke tingkat berikutnya hanya dilakukan selama nilai p95 berada pada atau di bawah ambang yang ditetapkan (SLO).
3. **Ukuran *Payload***
   Pengujian dampak ukuran *payload* dilakukan dengan skema pelipatgandaan secara bertahap, dimulai dari 50 KB dan seterusnya selama menjaga standar kinerja, *Service Level Objective* (SLO) ditetapkan dengan batas latensi p95 maksimal 1.000 *ms*. Peningkatan ukuran *payload* hanya akan dilanjutkan selama sistem mampu mempertahankan latensi di bawah ambang batas tersebut.
4. **Urutan Eksekusi Dan Pengendalian Bias**
   Urutan skenario diacak dalam blok agar efek pemanasan *cache* atau degradasi sementara tidak selalu menimpa kombinasi yang sama. Pendekatan ini mengacu pada prinsip *randomized block design* untuk mengurangi bias urutan eksekusi (NIST/SEMATECH).
5. **Prosedur Pengumpulan Data**
   Pengumpulan data dilakukan dengan prosedur yang terstandar untuk menjamin konsistensi hasil pengujian dan perbandingan antar skenario. Seluruh tahapan dirancang agar setiap kombinasi *framework* dan *database* diuji pada kondisi yang setara, sehingga perbedaan kinerja yang diamati dapat dikaitkan langsung dengan variabel yang diteliti, bukan dengan perbedaan lingkungan atau konfigurasi pengujian.

   Tahap sebelum pengujian meliputi verifikasi versi perangkat lunak yang digunakan, inisialisasi *dataset* Northwind pada PostgreSQL dan MongoDB, serta sinkronisasi waktu antara *host load generator* dan *host* server. Sinkronisasi waktu dilakukan untuk memastikan keselarasan pencatatan waktu antara metrik sisi klien dan sisi server. Selain itu, instrumen pemantauan sumber daya diaktifkan sebelum beban diberikan agar data penggunaan CPU dan memori terekam secara utuh sejak awal pengujian.

   Eksekusi pengujian beban dilakukan menggunakan Apache JMeter dengan mekanisme replikasi sebanyak tiga kali untuk setiap skenario pengujian. Pendekatan ini mengacu pada praktik pelaporan kinerja yang merekomendasikan pengulangan pengujian guna mengurangi pengaruh variasi sesaat. Seluruh pengujian dijalankan sesuai dengan skenario yang telah ditentukan, termasuk fase *ramp-up*, untuk menjaga konsistensi antar percobaan.

Alur pelaksanaan setiap replikasi pengujian terdiri dari tiga tahap utama. Pertama, beban diberikan sesuai skenario, dan seluruh metrik kinerja seperti *throughput*, latensi p95, dan tingkat kesalahan dicatat secara penuh sejak awal hingga akhir eksekusi. Kedua, setelah satu kali eksekusi selesai, diberikan jeda waktu (*cooldown*) untuk memungkinkan sistem kembali ke kondisi stabil sebelum replikasi berikutnya dimulai. Ketiga, setelah diperoleh tiga set data dari replikasi pengujian, nilai median dari metrik utama digunakan sebagai hasil akhir yang dilaporkan.

Secara paralel dengan pengujian beban, pemantauan sumber daya server dilakukan terhadap kontainer aplikasi dan *database*. Data penggunaan CPU dan memori dikumpulkan secara berkala dan disimpan sebagai deret waktu, kemudian divisualisasikan untuk keperluan analisis. Pada tahap analisis akhir, pencatatan waktu dari metrik kinerja dan data pemantauan sumber daya disejajarkan untuk mengaitkan perubahan performa aplikasi dengan kondisi penggunaan sumber daya server pada interval waktu yang sama.

## 2.8 Metode Analisis Data

### 2.8.1 Perhitungan Metrik

Analisis data dilakukan dengan menggabungkan dua kelompok metrik, yaitu metrik sisi klien dan metrik sisi server. Dari sisi klien (*load generator* Apache JMeter), data yang dicatat meliputi *throughput* (permintaan per detik), latensi persentil ke-95 (p95), dan *error rate*. Dari sisi server, data yang dicatat melalui Prometheus dan cAdvisor meliputi penggunaan CPU dan memori pada kontainer aplikasi dan *database.* Seluruh perhitungan dilakukan berdasarkan akumulasi data dari awal hingga akhir durasi eksekusi uji beban pada setiap skenario, sehingga perbandingan kinerja antar *framework* dan *database* dilakukan secara setara.

1. **Waktu Layanan**

   Pada aspek penggunaan CPU, efisiensi layanan dianalisis menggunakan pendekatan matematis yang merujuk pada *utilization law*, yang menyatakan bahwa utilitas sumber daya (U) merupakan hasil kali antara *throughput* $(X)$ dan waktu layanan $(S)$ yaitu $(U = X \times S)$. Persamaan tersebut dapat dibalik untuk memperoleh waktu layanan $S = U/X$.

   Dalam konteks penelitian ini, nilai utilitas direpresentasikan sebagai total waktu CPU gabungan yang digunakan oleh kontainer aplikasi dan *database* selama jendela pengamatan. Nilai *throughput* pada persamaan tersebut direpresentasikan oleh jumlah permintaan yang berhasil diproses (*samples*). Dengan demikian, waktu CPU rata-rata per permintaan dihitung dengan membagi total waktu CPU gabungan dengan jumlah permintaan, kemudian dikonversi ke satuan milidetik. Nilai ini merepresentasikan biaya komputasi CPU rata-rata yang dibutuhkan untuk melayani satu permintaan, sehingga semakin kecil nilainya, semakin efisien eksekusi layanan.

$$W_{CPU} = \frac{1000 \times (CPU_{App} + CPU_{BD})}{Samples}$$

- $W_{CPU}$: waktu CPU rata-rata per permintaan (*ms/req*)
- $CPU_{App}$*:* total waktu CPU kontainer aplikasi selama pengujian, (detik CPU)
- $CPU_{DB}$: total waktu CPU kontainer *database* selama pengujian, (detik CPU)
- $Samples$: Jumlah permintaan yang berhasil diproses
- 1000: faktor konversi dari detik ke milidetik.

**2. Efisiensi Memori (*Throughput* per *Peak* Memory)**

Berbeda dengan CPU yang diperlakukan sebagai sumber daya berbasis kerja, memori dianalisis sebagai sumber daya berbasis kapasitas. Efisiensi memori diukur untuk mengetahui seberapa besar *throughput* yang dapat dihasilkan sistem terhadap kapasitas memori maksimum yang digunakan selama pengujian. Metrik ini merepresentasikan densitas kinerja sistem terhadap sumber daya memori.

Perhitungan dilakukan dengan membagi rata-rata *throughput* dengan nilai puncak penggunaan memori gabungan kontainer aplikasi dan *database* selama jendela pengamatan. Mengingat data mentah pemantauan memori diperoleh dalam satuan *Mebibyte* (MiB), nilai tersebut dikonversi terlebih dahulu ke *Gibibyte* (GiB) agar rasio yang dihasilkan lebih representatif terhadap standar kapasitas server modern. Pada metrik ini, semakin besar nilainya, semakin efisien penggunaan memori sistem.

$$W_{MEM} = \frac{Throughput}{\left(\frac{MEM_{App} + MEM_{BD}}{1024}\right)}$$

- $W_{MEM}$: Efisiensi memori (*Rps/GiB*).
- $Throughput$: Rata-rata permintaan tiap detik selama durasi pengujian (*req/s*).
- $MEM_{App}$*:* Nilai puncak penggunaan memori kontainer aplikasi (MiB).
- $MEM_{DB}$: Nilai puncak penggunaan memori kontainer *database* (MiB).
- 1024: Faktor konversi dari *Mebibyte* (MiB) ke *Gibibyte* (GiB).

### 2.8.2 Perbandingan Dan Interpretasi

Tahap ini menjelaskan cara membaca dan menafsirkan hasil uji dengan memperhatikan hubungan antar variabel, bukan hanya membandingkan nilai *throughput* atau latensi p95 secara terpisah. Karena penelitian ini memfokuskan perbandingan dua *framework* pada dua jenis *database*, urutan perbandingan dibatasi agar setiap faktor dievaluasi pada kondisi yang setara.

Ruang lingkup perbandingan mencakup dua aspek utama. Pertama, perbandingan antar *framework* pada jenis *database* yang sama untuk menilai perbedaan kinerja layanan. Kedua, analisis lintas variasi peningkatan beban pengguna dan ukuran *payload* untuk mengamati sensitivitas kinerja sistem terhadap perubahan beban.

Interpretasi hasil dilakukan secara bertahap. Tahap pertama diawali dengan pemeriksaan pemenuhan *Service Level Objective* (SLO), yaitu apakah nilai latensi p95 berada pada atau di bawah ambang 1.000 *ms* untuk setiap skenario pengujian sesuai dengan batas persepsi pengguna. Selanjutnya, pada skenario yang memenuhi kriteria kinerja, dilakukan perbandingan performa murni berdasarkan *throughput* dan stabilitas latensi terhadap peningkatan beban.

Setelah layanan dinyatakan memenuhi target kinerja, analisis dilanjutkan pada aspek efisiensi penggunaan sumber daya. Apabila nilai p95 dari dua layanan relatif serupa tetapi rasio permintaan terhadap waktu CPU gabungan aplikasi dan *database* menunjukkan perbedaan yang signifikan, maka dapat disimpulkan bahwa efisiensi penggunaan CPU kedua layanan berbeda. Sebaliknya, apabila *throughput* meningkat namun latensi p95 melampaui ambang SLO, kondisi tersebut diinterpretasikan sebagai indikasi penurunan kualitas pengalaman pengguna.

Untuk memperkuat interpretasi, temuan dari metrik sisi klien kemudian dicek silang dengan metrik sisi server, khususnya penggunaan CPU dan memori pada layanan aplikasi dan *database.* Pendekatan ini memastikan bahwa kesimpulan yang diambil tidak hanya didasarkan pada kinerja permukaan, tetapi juga mempertimbangkan biaya sumber daya yang mendasarinya.