JURTI, Vol.10 No.2, Juni 2026, ISSN: 2579-8790

Analisis Perbandingan Kinerja gRPC Go dan Node.js pada Layanan *Microtransactions* Berbasis Arsitektur *Microservices*

Ariya Panna*[1], Bryan Adiputra Onggo[2], Master Edison Siregar[3]
[1,2,3] Informatika, Fakultas Sains dan Teknologi, Universitas Pradita, Scientia Business Park, Jl. Gading Serpong Boulevard No.1 Tower 1, Curug Sangereng, Kec. Klp. Dua, Kabupaten Tangerang, Banten 15810, Indonesia
e-mail: ***[1]ariya.panna@student.pradita.ac.id**, [2]bryan.adiputra@student.pradita.ac.id, [3]edison.siregar@pradita.ac.id

Abstrak

Microservices dikenal sebagai salah satu arsitektur yang banyak digunakan dalam aplikasi modern dengan skala besar saat ini. gRPC sebagai salah satu implementasi dari Application Programming Interface menjadi protokol yang paling sering direkomendasikan untuk komunikasi antar services pada arsitektur berbasis microservices. Namun, performa gRPC sangat dipengaruhi oleh karakteristik runtime bahasa pemrograman yang digunakan. Studi ini bertujuan untuk menganalisis perbandingan kinerja gRPC pada implementasi bahasa pemrograman Go dan Node.js melalui studi kasus mikrotransaksi. Metode pengujian yang digunakan adalah pengujian beban yang dibagi menjadi empat skenario, dimulai dari 1.000 hingga 100.000 permintaan dengan tingkat concurrency hingga 10.000. Metrik yang diukur mencakup throughput, latensi, serta penggunaan sumber daya CPU dan RAM. Hasil memperlihatkan bahwa Go memiliki stabilitas yang lebih unggul, mencapai 1.641 requests per detik pada beban puncak, sementara Node.js stagnan di angka 174 requests per detik. Dari sisi latensi, Node.js mengalami lonjakan waktu tunggu maksimal hingga 9 detik, sementara Go tetap stabil di 2,25 detik. Studi ini juga membuktikan bahwa Go lebih efisien dalam penggunaan sumber daya dengan konsumsi RAM 26,7% lebih rendah dibandingkan Node.js pada skenario beban tinggi. Studi ini merekomendasikan pemilihan Go untuk sistem mikrotransaksi dengan intensitas concurrency tinggi, sedangkan Node.js lebih ideal untuk layanan dengan beban kerja menengah.

Kata kunci*—gRPC, Go, Node.js, Microservices, API*

1. PENDAHULUAN

Pertumbuhan pesat suatu aplikasi, baik dari sisi jumlah pengguna maupun intensitas aktivitas, menuntut sistem untuk memiliki tingkat skalabilitas dan ketersediaan yang tinggi [1]. Kebanyakan aplikasi modern saat ini mengadopsi arsitektur berbasis *microservices* karena telah terbukti dapat meningkatkan performa dan ketersediaan sistem. Hal ini bisa terjadi karena setiap service fokus menjalankan satu tugas spesifik secara independen dan tidak akan terganggu apabila ada service yang bermasalah [2], [3], [4]. Komunikasi antar *services* dilakukan dengan mekanisme yang disebut sebagai Application Programming Interface atau lebih dikenal sebagai API. Namun, tantangan baru muncul seperti yang dikatakan Khan dan Ahamad dalam penelitiannya bahwa cara antar *services* berkomunikasi berperan krusial dalam sistem *microservices* yang aman dan efisien [5].

JURTI, Vol.10 No.2, Juni 2026, ISSN: 2579-8790

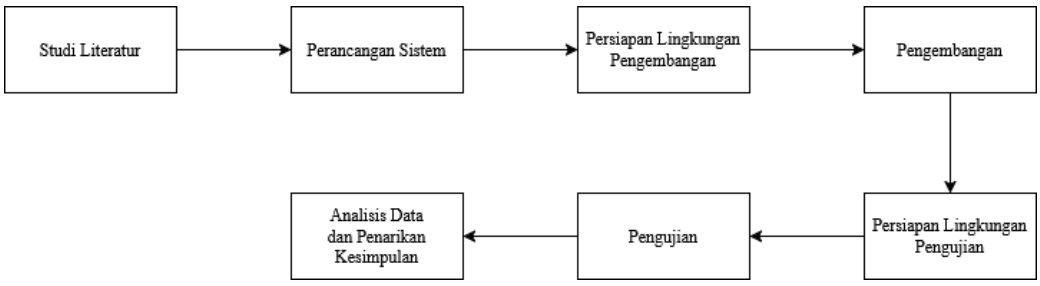
Saat ini, ada berbagai macam protokol API yang populer, antara lain REST, GraphQL, dan gRPC. Masing-masing protokol memiliki kelebihan dan kekurangan yang ditawarkan, seperti studi yang dilakukan oleh [6], [7]. Berdasarkan hasil kedua studi tersebut dijelaskan bahwa REST memiliki *error rate* tertinggi diantara protokol API lainnya, sedangkan GraphQL memiliki efisiensi penggunaan CPU paling baik namun *error rate* masih cukup tinggi. Sedangkan gRPC menunjukkan kestabilan dan direkomendasikan sebagai pilihan yang *reliable* untuk sistem *backend*. Sementara [8] menegaskan pemilihan protokol API harus kontekstual, bukan berdasarkan tren atau performa mentah saja. Dalam studinya juga dijelaskan bahwa protokol API yang ideal untuk konteks komunikasi internal seperti pada arsitektur *microservices* adalah gRPC.

Meskipun gRPC direkomendasikan dalam komunikasi antar *service* di arsitektur *microservices*, performa aktualnya sangat bergantung pada bahasa pemrograman yang digunakan untuk mengimplementasikannya. Studi yang telah dilakukan sebelumnya berupaya membandingkan performa antara Go dan Node.js, namun mayoritas protokol API yang diuji adalah REST API [9], [10]. Mengingat gRPC bekerja di atas protokol HTTP/2 dengan serialisasi biner Protocol Buffers, hasil evaluasi performa REST API tidak dapat dijadikan acuan mutlak untuk merepresentasikan kinerja gRPC. Selain itu, perbedaan fundamental antara Go dengan fitur goroutines dan Node.js dengan event loopnya diprediksi akan memberikan karakteristik performa yang berbeda saat menangani beban kerja yang intensif.

Studi ini menawarkan analisis perbandingan kinerja gRPC pada implementasi di bahasa Go dan Node.js dengan menggunakan studi kasus layanan mikrotransaksi. Pemilihan studi kasus ini didasarkan pada karakteristik mikrotransaksi yang membutuhkan latensi rendah dan konsistensi tinggi dalam menangani volume transaksi yang besar namun dengan ukuran data yang kecil. Analisis mencakup metrik *throughput*, latensi, serta penggunaan sumber daya komputasi (CPU dan RAM). Penting untuk dicatat bahwa tujuan dari studi ini bukan untuk menentukan bahasa pemrograman mana yang lebih unggul secara mutlak, melainkan untuk menyajikan data komparatif yang objektif mengenai karakteristik performa masing-masing runtime. Dengan demikian, hasil penelitian diharapkan menjadi acuan teknis bagi para arsitek sistem dalam memilih teknologi atau bahasa pemrograman yang sesuai dengan kebutuhan dari aplikasi, seperti volume transaksi yang diharapkan.

2. METODE PENELITIAN

Metode penelitian yang digunakan dalam penelitian ini terdiri dari beberapa tahapan yang disusun secara sistematis untuk memastikan proses analisis kinerja gRPC dilakukan secara terstruktur. Tahapan penelitian mencakup studi literatur, perancangan sistem, persiapan lingkungan pengembangan, pengembangan, persiapan lingkungan pengujian, pengujian, dan di tahap terakhir peneliti melakukan analisis data dan penarikan kesimpulan. Alur tahapan penelitian secara keseluruhan ditunjukkan pada Gambar 1.



Gambar 1. Alur penelitian

JURTI, Vol.10 No.2, Juni 2026, ISSN: 2579-8790

2.1. Studi Literatur

Di tahap awal ini, peneliti melakukan pengumpulan artikel dari basis data jurnal ilmiah dan analisis terhadap studi-studi terdahulu untuk memetakan posisi penelitian di dalam peta keilmuan serta memastikan kebaruan dari eksperimen yang dilakukan. Aktivitas pada tahap studi literatur tidak hanya mencakup identifikasi karakteristik teknologi dan parameter *benchmark* yang umum digunakan, tetapi juga menghasilkan pemetaan terhadap temuan-temuan studi terdahulu. Berdasarkan hasil kajian literatur, diketahui bahwa protokol gRPC menawarkan performa yang lebih stabil dan reliabel dibandingkan REST dan GraphQL untuk komunikasi internal pada arsitektur *microservices*. Namun, sebagian besar penelitian yang membandingkan performa antar bahasa pemrograman, seperti Go dan Node.js, masih berfokus pada implementasi REST API. Peneliti mengidentifikasi bahwa adanya kesenjangan penelitian berupa kurangnya analisis komparatif yang secara spesifik menganalisis kinerja gRPC pada Go dan Node.js dengan mempertimbangkan metrik seperti *throughput*, latensi, serta penggunaan sumber daya komputasi.

2.1.1. Identifikasi Karakteristik Teknologi

Peneliti mempelajari urgensi dari arsitektur berbasis *microservices* seperti latar belakang, cara kerja, kelebihan, serta kekurangan dari arsitektur *microservices*. Kemudian, peneliti juga membaca dokumentasi teknis protokol gRPC dan melakukan perbandingan dengan protokol berbasis REST untuk memahami alasan efisiensi penggunaan binary serialization di atas HTTP/2.

2.1.2. Penetapan Parameter Benchmark

Peneliti melakukan peninjauan terkait standar pengujian performa sistem yang digunakan oleh peneliti-peneliti sebelumnya untuk mengidentifikasi metrik pengujian yang paling representatif.

2.1.3. Identifikasi Kesenjangan Penelitian

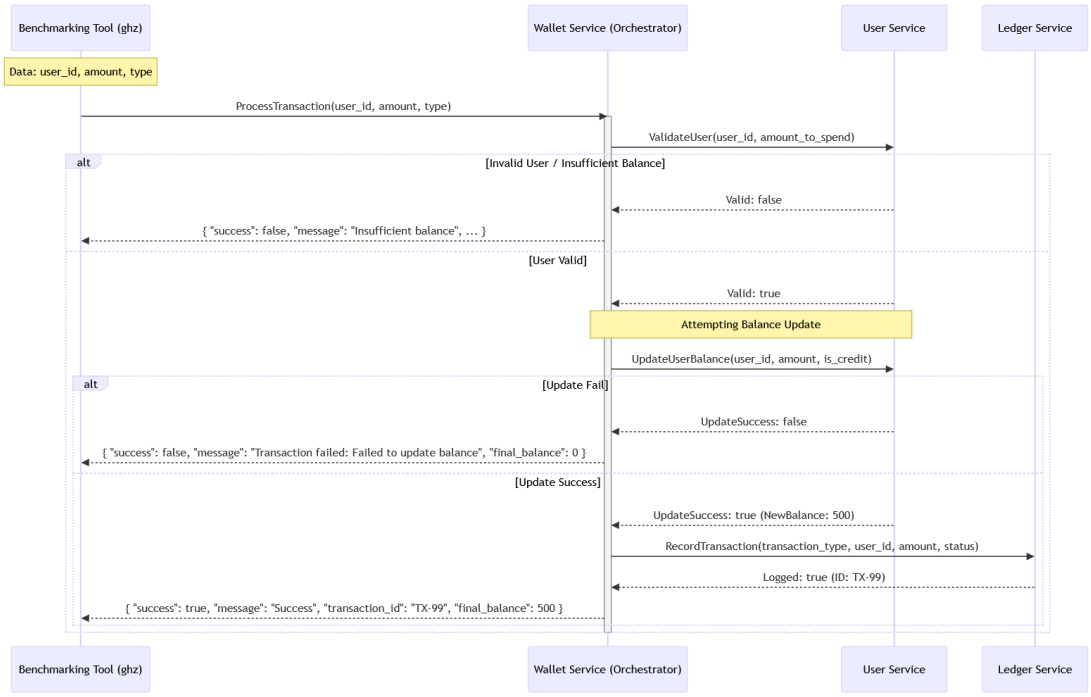
Pada tahap ini, peneliti melakukan evaluasi terhadap literatur yang telah dikumpulkan untuk menemukan celah penelitian. Tahapan-tahapan yang dilakukan meliputi pemetaan fokus protokol API, peninjauan kedalaman studi terdahulu, dan merumuskan nilai kebaruan dari penelitian ini.

2.2. Perancangan Sistem

Tahap perancangan sistem dilakukan sebagai acuan untuk memfasilitasi proses implementasi, memastikan kelancaran komunikasi antar *services*, serta menghasilkan sistem yang modular [1], [11]. Perancangan sistem dimulai dengan membuat sequence diagram untuk menggambarkan alur proses sistem seperti yang divisualisasikan pada Gambar 2.

Di setiap *service*, sistem dibangun dengan arsitektur berlapis untuk tujuan pemisahan tanggung jawab antara *handler*, logika bisnis, dan akses data. Kemudian tahapan selanjutnya adalah perancangan kontrak komunikasi antara *client* dengan *server* menggunakan format file .proto yang berisikan fungsi yang bisa dipanggil, struktur data untuk *request* dan *response* beserta dengan tipe datanya. Kontrak komunikasi ini dibutuhkan karena gRPC dirancang untuk bekerja menggunakan Protocol Buffers sebagai Interface Definition Language (IDL) [8], [12].

JURTI, Vol.10 No.2, Juni 2026, ISSN: 2579-8790



Gambar 2. Desain sistem

File .proto yang sudah didefinisikan kemudian dijadikan sebagai single source of truth saat pengimplementasian gRPC di Go dan Node.js.

2.3. Persiapan Lingkungan Pengembangan

Tahap ini berguna mempersiapkan infrastruktur untuk tahap berikutnya yakni pengembangan dengan mempersiapkan ekosistem perangkat lunak yang diperlukan selama proses pembuatan service. Persiapan meliputi instalasi SDK dan Runtime Environment untuk kedua bahasa pemrograman yang akan diuji yakni Go dan Node.js. Untuk sisi Go, digunakan Go Compiler versi stabil terbaru yakni versi 1.25.5. Sedangkan sisi Node.js, digunakan Node.js Long Term Support (LTS) saat ini yaitu versi 24.12.0. Kemudian karena gRPC bergantung pada Protocol Buffers (Protobuf), diperlukan instalasi compiler untuk file .proto. Selain itu, ditambahkan plugin spesifik untuk masing-masing bahasa, yaitu protoc-gen-go dan protoc-gen-go-grpc untuk lingkungan Go, serta @grpc/proto-loader untuk lingkungan Node.js. Dengan demikian, memungkinkan sistem untuk menghasilkan kode otomatis berdasarkan file .proto yang telah dirancang sebelumnya. Lalu dalam proses implementasinya, peneliti menggunakan dua jenis IDE untuk mendukung produktivitas dan efisiensi waktu selama proses pengembangan. Peneliti menggunakan JetBrains GoLand untuk pengembangan sistem berbasis Go karena keunggulannya dalam fitur auto-complete dan auto-import. Sementara itu, Visual Studio Code digunakan untuk implementasi sistem berbasis Node.js karena fleksibilitasnya dalam manajemen depedensi dan ekosistem plugin yang ringan.

2.4. Pengembangan

Tahapan ini merupakan tahapan inti dari penelitian, dimana proses translasi rancangan sistem menjadi fungsionalitas program. Peneliti memastikan bahwa kode yang ditulis memiliki struktur yang setara antara dua bahasa pemrograman yang diuji agar memberikan hasil pengujian yang objektif. Seluruh kode program disetor di sebuah repository Github yang bertujuan untuk menghindari konflik versi kode, mempermudah manajemen source code dan kolaborasi antar peneliti [13], [14].

JURTI, Vol.10 No.2, Juni 2026, ISSN: 2579-8790

2.1.4. Generasi Kode dari Kontrak

Langkah pertama yang dilakukan dalam tahap pengembangan adalah kompilasi file .proto yang telah dibuat, kemudian menghasilkan kode perantara secara otomatis. Langkah ini bertujuan untuk memastikan standarisasi komunikasi, seperti memiliki nama fungsi dan tipe data yang sama persis baik di Go dan Node.js.

2.1.5. Implementasi Arsitektur Berlapis

Setelah implementasi kontrak di masing-masing bahasa pemrograman telah diimplementasikan, langkah selanjutnya sesuai dengan rancangan adalah menggunakan pola arsitektur berlapis. Lapisan *handler* bertugas menangani penerimaan koneksi gRPC dan *unmarshalling* pesan yang dikirim dari client. Lapisan logika atau *service* bertugas menjalankan logika bisnis utama masing-masing service. Terakhir, lapisan data atau repositori bertanggung jawab dengan segala urusan pengaksesan dan manipulasi data [15].

2.1.6. Penyelarasan Logika Bisnis

Proses ini diperlukan untuk sinkronisasi algoritma pemrosesan sekaligus memastikan bahwa beban kerja komputasi yang diterima antara Go dan Node.js sama. Setiap langkah, mulai dari permintaan datang dari *client*, diproses oleh lapisan *service*, manipulasi data pada lapisan repositori, sampai respon kembali ke *client* dibuat identik.

2.5. Persiapan Lingkungan Pengujian

Setelah tahap pengembangan selesai, tahap persiapan lingkungan untuk pengujian dilakukan untuk menciptakan kondisi pengujian yang terisolasi dan terkendali. Tujuan dilakukannya persiapan ini agar perbedaan performa yang tercatat murni berdasarkan karakteristik bahasa pemrograman, bukan pengaruh dari latar belakang sistem operasi atau perangkat keras. Peneliti menggunakan Docker dan Docker Compose untuk membungkus masing-masing *service*. Penggunaan Docker mendukung infrastruktur sistem yang efisien, fleksibel, dan skalabel [16]. Peneliti juga menetapkan batasan sumber daya pada setiap kontainer, yakni dengan konfigurasi pembatasan penggunaan CPU 0.5 core dan penggunaan memori di 512 MB. Lalu untuk alat penguji, peneliti menyiapkan ghz sebagai alat utama untuk melakukan *load testing* gRPC. Pemilihan alat ini dilandaskan pada tujuan dibangunnya alat tersebut, yakni sebagai benchmarking tool khusus gRPC dan mendukung protokol HTTP/2 secara *native*. Untuk meminimalisir variabel pengganggu, seluruh layanan dijalankan di lingkungan perangkat keras yang identik dengan spesifikasi prosesor Intel Core i3-2120 (2 core, 4 thread), memori 4 GB DDR3, dan sistem operasi Debian 13 trixie.

2.6. Pengujian

Tahap berikutnya setelah lingkungan pengujian siap adalah eksekusi pengujian. Tahap pengujian dilakukan menggunakan metode load testing untuk mengukur batas kemampuan sistem. Pengujian dilakukan dengan 4 skenario berbeda yang merepresentasikan tingkat beban dari rendah hingga ekstrem seperti yang ditunjukkan pada Tabel 1.

Tabel 1. Skenario Pengujian

Skenario	Total Requests	Concurrency	Tujuan
1	1.000	1.00	Mengukur performa pada beban rendah.
2	10.000	1.000	Mengukur performa pada beban menengah.
3	50.000	5.000	Mengukur stabilitas sistem saat beban tinggi.
4	100.000	10.000	Mengukur titik jenuh sistem.

Prosedur eksekusi pengujian dilakukan secara bergantian antara implementasi Node.js dan Go dan dilakukan jeda waktu di setiap perpindahan skenario untuk memastikan memori telah

JURTI, Vol.10 No.2, Juni 2026, ISSN: 2579-8790

dibersihkan sepenuhnya. Saat pengujian berlangsung, pemanfaatan sumber daya setiap *service* dicatat secara *real-time* menggunakan perintah docker stats untuk mendapatkan nilai konsumsi puncak.

2.7. Analisis Data dan Penarikan Kesimpulan

Data yang dikumpulkan dari tahap pengujian kemudian dikumpulkan dan diolah untuk menarik kesimpulan yang objektif. Proses analisis dilakukan melalui beberapa tahapan, antara lain:

1. **Rekapitulasi Data:** Menggabungkan hasil data pemanfaatan sumber daya dari docker stats dengan metrik performa dari ghz.
2. **Visualisasi Data:** Mentransformasikan data numerik ke bentuk grafik garis dan batang menggunakan pustaka Python, yakni matplotlib untuk memudahkan observasi tren perbandingan hasil pengujian antara Node.js dan Go di setiap skenario.
3. **Analisis Komparatif:** Melakukan evaluasi efisiensi bahasa pemrograman dengan membandingkan rasio *throughput* terhadap penggunaan sumber daya. Analisis juga mencakup kestabilan sistem dengan mengamati lonjakan latensi pada kondisi beban ekstrem,
4. **Penarikan Kesimpulan**: Merumuskan hasil akhir berdasarkan perbandingan data yang diperoleh untuk menentukan karakteristik performa masing-masing *runtime* dalam menangani protokol gRPC pada arsitektur *microservices.*

3. HASIL DAN PEMBAHASAN

Implementasi sistem dilakukan berdasarkan desain sistem yang telah dirancang pada Gambar 2. Sistem dibangun menggunakan arsitektur *microservices* dengan 3 layanan utama, antara lain *User Service, Wallet Service,* dan *Ledger Service* yang saling berkomunikasi menggunakan protokol gRPC. Setiap layanan diimplementasikan menggunakan dua runtime berbeda, yakni Go dan Node.js dengan struktur kode yang diselaraskan untuk memastikan kesetaraan logika bisnis. File .proto yang telah dirancang kemudian digunakan sebagai acuan utama dalam menghasilkan kode stub secara otomatis pada kedua bahasa pemrograman. Setiap layanan diimplementasikan dengan arsitektur berlapis yang terdiri dari *handler, service,* dan *repository* untuk memisahkan tanggung jawab antar komponen.

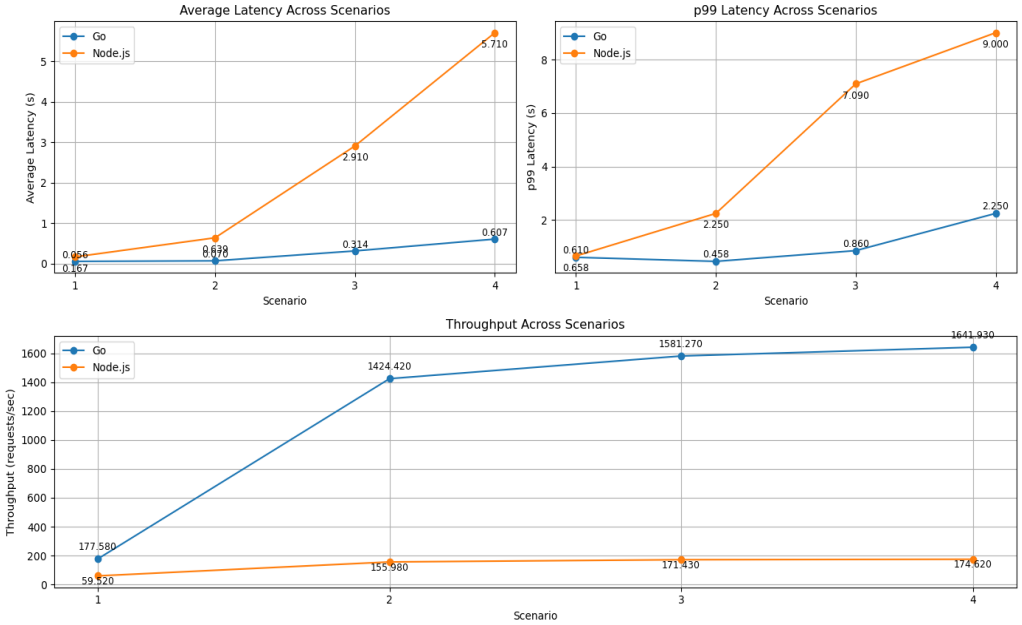
Sebelum tahap pengujian performa dilakukan, masing-masing layanan telah melalui tahap pengujian fungsional untuk memastikan seluruh *endpoint* gRPC dapat berjalan dengan baik. Proses verifikasi dilakukan menggunakan alat pengujian API, yakni *Bruno*. Dengan dilakukannya proses verifikasi fungsional setiap layanan, dapat dipastikan bahwa setiap layanan mampu menerima dan merespons permintaan sesuai dengan kontrak yang didefinisikan. Hasil pengujian performa sistem berdasarkan skenario beban yang telah ditentukan disajikan pada Gambar 3.

Hasil pengujian pada Gambar 3 menunjukkan performa sistem selama pengujian dengan berbagai skenario load. Skenario 1 hingga 4 mewakili peningkatan jumlah request dan concurrency, yakni 1.000 *requests*/100 *concurrency* hingga 100.000 *requests*/10.000 *concurrency*. Peningkatan *concurrency* menyebabkan kenaikan latensi baik di Node.js maupun Go. Node.js menunjukkan kenaikan yang lebih tajam pada skenario 3 dan 4, sedangkan Go menunjukkan peningkatan lebih bertahap. Saat memasuki skenario 3, rata-rata latensi di Node.js mencapai hampir 5,71 detik per *request* pada skenario beban puncak, sementara Go tetap stabil di bawah 1 detik.

Hal yang lebih krusial terlihat pada metrik persentil ke-99 (p99), metrik ini mendeskripsikan batas waktu tunggu maksimal yang dialami oleh 99% *request*. Node.js menunjukkan waktu tunggu hingga 9 detik. Berbeda dengan Go yang mampu menjaga batas waktu respon tetap terkendali di angka 2,25 detik. Hal ini menunjukkan adanya hambatan antrian

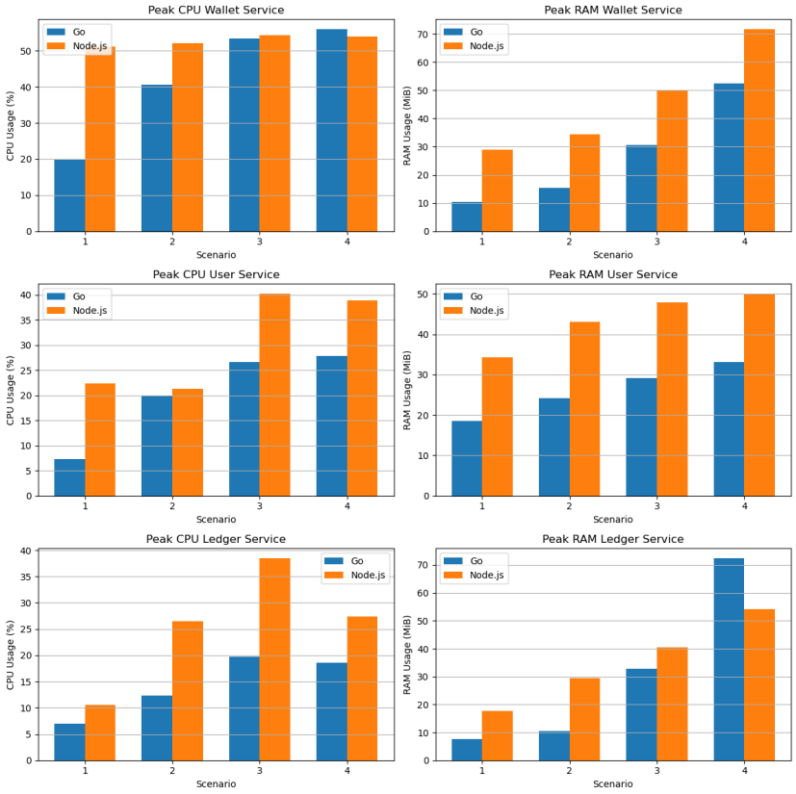
JURTI, Vol.10 No.2, Juni 2026, ISSN: 2579-8790

pemrosesan *request* yang signifikan di Node.js, di mana sebagian besar permintaan akan merasakan keterlambatan yang sangat tidak ideal.



Gambar 3. Grafik rata-rata latensi, persentil ke-99, dan *throughput*

Dari sisi *throughput*, Node.js relatif stagnan pada skenario tinggi, sedangkan Go memperlihatkan peningkatan *throughput* yang lebih progresif. Node.js hanya mampu menangani *request* kurang dari 200 *requests* dalam 1 detik, sementara Go mampu menangani lebih dari 1000 *requests* setiap detiknya mulai dari skenario 2.



Gambar 4. Grafik perbandingan penggunaan CPU dan RAM puncak

JURTI, Vol.10 No.2, Juni 2026, ISSN: 2579-8790

Gambar 4 mengilustrasikan perbandingan penggunaan CPU dan RAM puncak antara Node.js dan Go pada masing-masing *service*. Secara umum, Go cenderung lebih efisien dalam pemanfaatan CPU dan RAM, sedangkan Node.js menggunakan RAM lebih tinggi untuk service yang sama. Peningkatan penggunaan sumber daya pada semua *service* disebabkan oleh peningkatan load dari skenario 1 hingga 4. Node.js menunjukkan penggunaan CPU yang relatif stabil di Wallet Service, namun peningkatan tajam terjadi pada User dan Ledger Service di skenario tinggi. Di sisi lain, Go menunjukkan peningkatan CPU sedikit demi sedikit pada User dan Wallet Service, sementara di Ledger Service terjadi lonjakan pada RAM di skenario 4.

4. KESIMPULAN

Berdasarkan hasil pengujian, dapat ditarik kesimpulan antara lain:

- **Latensi:** Go mengalami peningkatan latensi yang gradual dibanding Node.js, meskipun load meningkat secara signifikan. Node.js menunjukkan lonjakan latensi pada skenario tinggi, mencerminkan keterbatasan event loop tunggal dalam menghadapi concurrency ekstrem.
- **Throughput:** *Throughput* Go meningkat seiring bertambahnya concurrency, sementara Node.js relatif stagnan pada skenario tinggi.
- **Pemanfaatan Sumber Daya:** Pemanfaatan CPU dan RAM untuk Wallet dan User Service lebih efisien pada Go, sedangkan Node.js menggunakan RAM lebih tinggi untuk service yang sama.
- **Kesesuaian Skala:** Node.js masih dapat diandalkan untuk layanan dengan load ringan hingga menengah, sedangkan Go lebih sesuai untuk kebutuhan skala besar berkat fitur goroutines yang efisien.
- **Keterbatasan Penelitian:** Skenario pengujian terbatas pada empat skenario load sehingga performa untuk load diluar skenario ini tidak dianalisis. Analisis hanya terbatas pada CPU dan RAM, metrik lain seperti network latency, I/O disk, dan lain-lain tidak diukur. Fokus sistem adalah pada layanan microtransactions dengan payload kecil.

5. SARAN

Berdasarkan hasil studi ini, peneliti memberikan beberapa saran untuk pengembangan studi lanjutan, antara lain:

- Pengujian dan analisis dapat diperluas dengan skenario load yang lebih ekstrem atau service dengan studi kasus berbeda dengan tujuan mendapatkan gambaran performa yang lebih komprehensif.
- Analisis pemanfaatan sumber daya dapat ditambahkan metrik lain seperti network latency, I/O disk supaya evaluasi performa lebih lengkap.

DAFTAR PUSTAKA

[1] D. Irawan and Fauzi, “Implementasi Arsitektur Microservices pada Pengembangan Aplikasi Absensi Web Terdistribusi,” *bit-Tech*, vol. 7, no. 3, pp. 1118–1127, Apr. 2025, doi: 10.32877/bt.v7i3.2401.

[2] M. Niswar, R. A. Safruddin, A. Bustamin, and I. Aswad, “Performance evaluation of microservices communication with REST, GraphQL, and gRPC,” *International Journal of Electronics and Telecommunications*, vol. 70, no. 2, pp. 429–436, Jun. 2024, doi: 10.24425/ijet.2024.149562.

[3] Surya Prabha Busi, “Understanding Microservices Architecture: A Comprehensive Guide,” *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, vol. 11, no. 1, pp. 1440–1447, Jan. 2025, doi: 10.32628/cseit251112144.

[4] L. M. Alchuluq and F. Nurzaman, "Analisis Pada Arsitektur Microservice Untuk Layanan Bisnis Toko Online," 2021.

[5] I. Khan and M. K. Ahamad, "Enhancing Security and Performance of gRPC-Based Microservices using HTTP/3 and AES-256 Encryption." [Online]. Available: https://www.jisem-journal.com/

[6] S. Chandra and A. Farisi, "Comparative Analysis of RESTful, GraphQL, and gRPC APIs: Perfomance Insight from Load and Stress Testing," *Jurnal Sisfokom (Sistem Informasi dan Komputer)*, vol. 14, no. 1, pp. 81–85, Jan. 2025, doi: 10.32736/sisfokom.v14i1.2315.

[7] R. Putra Fhonna, Y. Afrillia, V. Ilhadi, A. H. Arif, and R. A. Selian, "Performance Analysis of API Protocol Models as Recommendations for Developers in Application Development," *JINAV: Journal of Information and Visualization*, vol. 5, no. 2, pp. 2746–1440, 2024, doi: 10.35877/454RI.jinav3041.

[8] R. SIKORA and A. POSTOLIUK, "Comparative Analysis of The Effectiveness of Architectural Styles Rest, GRAPHQL, And GRPC For Scalable Microservice Systems," *Herald of Khmelnytskyi National University. Technical sciences*, vol. 355, no. 4, pp. 560–568, Aug. 2025, doi: 10.31891/2307-5732-2025-355-79.

[9] H. Ardiansyah and A. Fatwanto, "Comparison of Memory usage between REST API in Javascript and Golang," *MATRIK : Jurnal Manajemen, Teknik Informatika dan Rekayasa Komputer*, vol. 22, no. 1, pp. 229–240, Nov. 2022, doi: 10.30812/matrik.v22i1.1325.

[10] F. Pratama and A. Farisi, "Analisis Perbandingan Kinerja Backend API Menggunakan PHP, Golang, dan JavaScript Performance Analysis of Backend API Using PHP, Golang, and Node JS."

[11] M. Zukhruf Ain, R. Ardiansyah, S. Anggun Pratama, M. Akbar, and N. Trezandy Lapatta, "Comparative Performance Analysis of GRPC and Rest API Under Various Traffic Conditions and Data Sizes Using a Quantitative Approach," 2025. [Online]. Available: http://jurnal.polibatam.ac.id/index.php/JAIC

[12] I. G. Buzhin, A. Yu. Derevyankin, V. M. Antonova, A. P. Perevalov, and Y. B. Mironov, "Comparative Analysis of the REST and gRPC Used in the Monitoring System of Communication Network Virtualized Infrastructure," *T-Comm*, vol. 17, no. 4, pp. 50–55, 2023, doi: 10.36724/2072-8735-2023-17-4-50-55.

[13] A. Nizar Azzamy *et al.*, "Penggunaan Aplikasi GitHub sebagai Wadah dalam Membangun Relasi dan Mengembangkan Skill bagi Para Mahasiswa Ilmu Komputer UNNES," 2025. [Online]. Available: http://jurnalilmiah.org/journal/index.php/majemuk

[14] D. Swasono Rahmad and A. Rama Syarif, "Perancangan dan Implementasi Platform Manajemen Repositori Berbasis Git Menggunakan Teknologi Open Source," 2025.

[15] L. Y. Contreras Rivas, E. López Domínguez, Y. Hernández Velázquez, S. Domínguez Isidro, M. A. Medina Nieto, and J. De La Calleja, "A Layered Software Architecture for the Development of Smart Mobile Distributed Systems Oriented to the Management of Emergency Cases," *Applied Sciences (Switzerland)*, vol. 15, no. 7, Apr. 2025, doi: 10.3390/app15073664.

[16] U. Bani Saleh FTID, Z. Mutaqin Subekti, K. Mukiman, M. Lutfi Sulthon Auliya Sulistiyono, and R. Eka Putra, "Rancang Bangun Infrastruktur Web Server Berbasis Docker Pada Ubuntu Server," 2024.