

Analisis Performansi dan Skalabilitas Migrasi Arsitektur Monolitik ke Microservices Berbasis Cloud Infrastructure: Studi Kasus Sistem Tryout Ujian Daring

Dian Wahyuningsih¹, Luqman Affandi²

¹Program Studi Teknologi Informasi, STMIK PPKIA Pradnya Paramita, Indonesia

²Jurusan Teknologi Informasi, Politeknik Negeri Malang, Indonesia

¹dian.wahyuningsih@stimata.ac.id, ²laffandi@polinema.ac.id

Abstrak

Arsitektur monolitik banyak dipilih pada tahap awal pengembangan sistem informasi karena kesederhanaan proses pengembangan dan deployment-nya. Namun, seiring meningkatnya jumlah pengguna simultan, arsitektur ini kerap mengalami keterbatasan skalabilitas, khususnya pada sistem yang memiliki pola beban tidak merata seperti sistem tryout ujian daring yang diakses ribuan peserta secara bersamaan pada rentang waktu tertentu. Penelitian ini bertujuan menganalisis performansi dan skalabilitas migrasi arsitektur monolitik ke microservices berbasis cloud infrastructure pada studi kasus sistem tryout. Metode penelitian dilakukan dengan membangun dua prototipe aplikasi tryout yang identik secara fungsional, satu berarsitektur monolitik dan satu berarsitektur microservices (terdiri atas auth-service, question-service, attempt-service, scoring-service, dan API Gateway), lalu melakukan pengujian beban (*load testing*) menggunakan autocannon pada level *concurrency* 5, 10, 20, dan 50 koneksi simultan untuk skenario baseline dan alur bisnis multi-hop, serta tambahan *concurrency* 100 khusus pada skenario pengujian skalabilitas horizontal, mencakup pengukuran *throughput*, *response time*, *latency* p99, dan perilaku skalabilitas horizontal. Hasil pengujian menunjukkan bahwa pada beban rendah hingga menengah, arsitektur monolitik memberikan *throughput* dan *latency* yang lebih baik dibandingkan microservices karena tidak adanya overhead komunikasi antar-jaringan (*network hop*); setiap konfigurasi diuji tiga kali pengulangan (*repeated trials*) setelah *warm-up* untuk meningkatkan reliabilitas pengukuran, dan pada *concurrency* 50, monolitik mencapai rata-rata 8.326,7 request/detik dibandingkan 4.515,8 request/detik pada microservices. Pada endpoint yang melibatkan panggilan antar-layanan (*attempt/submit*, *concurrency* 20), microservices menunjukkan penurunan *throughput* signifikan (rata-rata 1.136,4 req/detik) dibandingkan monolitik (2.215,2 req/detik) akibat dua kali *network round-trip*. Pengukuran pemanfaatan CPU pada level proses menunjukkan bahwa microservices secara konsisten menggunakan CPU jauh lebih tinggi (hingga mendekati saturasi 1 vCPU) dibandingkan monolitik pada beban yang sama. Pengujian skalabilitas horizontal question-service menunjukkan bahwa manfaat penambahan instance sangat bergantung pada ketersediaan sumber daya komputasi (CPU core) pada infrastruktur cloud yang digunakan; temuan dan kesimpulan penelitian ini berlaku pada lingkungan eksperimen single-vCPU yang digunakan dan tidak digeneralisasikan langsung terhadap infrastruktur cloud multi-core/multi-node. Penelitian ini memberikan kontribusi berupa data empiris kuantitatif yang dapat menjadi pertimbangan bagi pengembang dalam menentukan titik migrasi yang tepat dari arsitektur monolitik ke microservices.

Kata kunci: arsitektur monolitik, *microservices*, skalabilitas, *cloud infrastructure*, *load testing*, sistem *tryout*

1. Pendahuluan

Sistem tryout ujian daring merupakan salah satu jenis aplikasi web yang memiliki karakteristik beban akses tidak merata (*bursty traffic*): jumlah pengguna dapat melonjak drastis pada periode tertentu, misalnya menjelang jadwal ujian nasional atau seleksi masuk perguruan tinggi, kemudian menurun tajam di luar periode tersebut. Karakteristik ini menuntut arsitektur sistem yang mampu beradaptasi secara elastis terhadap perubahan beban.

Pada tahap awal pengembangan, sebagian besar

tim pengembang memilih arsitektur monolitik karena kesederhanaannya: seluruh modul aplikasi autentikasi, manajemen bank soal, sesi pengerjaan ujian, dan penilaian digabungkan dalam satu basis kode dan satu unit *deployment*. Pendekatan ini mempercepat proses pengembangan awal, namun beberapa penelitian menunjukkan bahwa arsitektur monolitik lebih sesuai untuk aplikasi dengan jumlah pengguna kecil, sedangkan penambahan beban yang signifikan cenderung menurunkan performanya secara drastis.

Studi mengenai evaluasi performansi *platform* pendaftaran lomba nasional melaporkan keterbatasan

performa yang nyata ketika sebuah *website* berarsitektur monolitik diakses oleh lebih dari 1.000 pengguna secara bersamaan, dengan penurunan performa dipicu oleh proses *query* yang kompleks serta penggunaan resource statis yang tidak efisien (Suryotrisongko, 2017). Temuan tersebut dikonfirmasi secara lebih luas oleh studi komparatif berskala besar yang mengevaluasi performa dan skalabilitas arsitektur monolitik versus *microservice*, yang menyimpulkan bahwa *microservice* secara konsisten unggul pada skenario beban tinggi meskipun memerlukan kompleksitas infrastruktur tambahan (Blinowski et al., 2022). Studi lain pada sistem informasi akademik perguruan tinggi menemukan bahwa arsitektur monolitik yang telah berjalan lama menghadapi kendala skalabilitas, kesulitan pemeliharaan, dan ketergantungan antar-modul yang tinggi, sehingga migrasi ke *microservices* dengan skor kesiapan tinggi dinilai layak untuk mengatasi kendala tersebut (Ramadhan Putra et al., 2025). Penelitian pada sistem Dana Pensiun Lembaga Keuangan (DPLK) turut mengonfirmasi bahwa performa aplikasi web *microservices* secara umum lebih unggul dibandingkan aplikasi monolitik dalam menangani beban akses yang besar (Munawar & Hodijah, 2018).

Namun demikian, migrasi ke *microservices* bukan tanpa konsekuensi. Sebuah tinjauan literatur sistematis terhadap studi migrasi monolitik-ke-*microservice* mengidentifikasi bahwa tantangan utama meliputi kompleksitas komunikasi antarlayanan, konsistensi data terdistribusi, dan kebutuhan orkestrasi *container* yang matang (Hassan et al., 2024). Tinjauan literatur lain yang berfokus pada pola *deployment* dan komunikasi *microservice* turut menegaskan bahwa pemilihan pola komunikasi (*synchronous* vs. *asynchronous*) sangat memengaruhi performansi keseluruhan sistem (Karabey Aksakalli et al., 2021). Beberapa penelitian mengenai pola migrasi *microservices* menunjukkan bahwa proses migrasi idealnya dilakukan secara bertahap dan terukur, misalnya dengan menerapkan *Microservices Migration Pattern* serta mengevaluasi kesiapan arsitektur menggunakan instrumen seperti *Microservices Scorecard* sebelum implementasi penuh dilakukan (Ramadhan Putra et al., 2025).

Kesenjangan (*research gap*) yang mendasari penelitian ini adalah minimnya data empiris kuantitatif yang membandingkan secara langsung performansi dan skalabilitas kedua arsitektur pada studi kasus yang identik secara fungsional. Sejumlah penelitian terdahulu telah melakukan perbandingan serupa pada domain lain, seperti aplikasi berbasis kontainerisasi Docker (Dirgantara et al., 2024) dan aplikasi dengan volume transaksi tinggi (Marieska et al., 2025), namun sebagian besar berfokus pada satu titik beban tertentu tanpa analisis granular pada level *endpoint* dan pola komunikasi (*single-service* vs. *multi-hop*), serta belum menyertakan pengujian skalabilitas horizontal secara langsung. Penelitian ini melengkapi

Tabel 1. Matriks komparatif cakupan analisis penelitian terdahulu vs. penelitian ini

Penelitian	E	M	H
Blinowski et al. (2022)	–	–	–
Munawar & Hodijah (2018)	–	–	–
Dirgantara et al. (2024)	–	–	–
Marieska et al. (2025)	–	–	–
Sanjaya & Murnawan (2024)	–	–	–
Ramadhan Putra et al. (2025)	–	–	–
Fauziah & Surantha (2026)	–	Sebagian	–
Penelitian ini	✓	✓	✓

kesenjangan tersebut khususnya untuk domain sistem *tryout* ujian daring yang memiliki pola beban unik (*bursty traffic*). Dibandingkan dengan penelitian-penelitian terdahulu yang umumnya melaporkan keunggulan *microservices* secara agregat pada satu titik beban tertentu (Blinowski et al., 2022; Munawar & Hodijah, 2018; Sanjaya & Murnawan, 2024), kontribusi utama penelitian ini terletak pada tiga hal: (1) analisis performansi granular pada level *endpoint* dan pola komunikasi (*single-service* vs. *multi-hop*), bukan hanya pada level aplikasi secara keseluruhan; (2) pengujian skalabilitas horizontal secara langsung dengan variasi jumlah instance, yang jarang disertakan pada studi sejenis; dan (3) penyediaan data empiris kuantitatif berjenjang (*concurrency* 5 hingga 50 untuk pengujian *baseline* dan alur bisnis, serta hingga 100 khusus pada pengujian skalabilitas horizontal), diperkuat dengan prosedur *repeated trials* serta pengukuran CPU dan memori pada level proses untuk mendukung reliabilitas dan interpretasi kausal hasil, yang dapat dijadikan acuan praktis bagi pengembang dalam menentukan titik migrasi yang tepat dari arsitektur monolitik ke *microservices*, khususnya untuk sistem dengan pola beban tidak merata seperti sistem *tryout* ujian daring.

Untuk memperkuat argumen *research gap* di atas, Tabel 1 merangkum cakupan analisis pada penelitian-penelitian terdahulu yang paling relevan, dibandingkan dengan cakupan analisis pada penelitian ini.

Keterangan: E = analisis performansi pada level *endpoint* individual (bukan hanya level aplikasi agregat); M = analisis pola komunikasi *single-service* vs. *multi-hop* antar-layanan; H = pengujian skalabilitas horizontal (variasi jumlah instance) secara langsung dan terkontrol. Tanda “–” menunjukkan bahwa aspek tersebut tidak dilaporkan atau bukan fokus utama pada studi terkait, sejauh dapat ditelusuri dari publikasi yang dirujuk, bukan penilaian mutlak terhadap keseluruhan metodologi studi tersebut. Sebagaimana terlihat pada Tabel 1, seluruh penelitian pembandingan melaporkan perbandingan performa pada level aplikasi secara agregat, dan hanya Fauziah dan Surantha (2026) yang menyentuh perbedaan beban kerja pada level komunikasi antar-komponen (*middleware*),

namun tidak sampai pada analisis *endpoint* individual maupun pengujian skalabilitas horizontal secara terkontrol. Kekosongan inilah yang secara spesifik diisi oleh penelitian ini melalui tiga kontribusi yang telah disebutkan sebelumnya: analisis granular per-*endpoint* dan pola komunikasi (single-service vs. multi-hop), pengujian skalabilitas horizontal langsung dengan variasi jumlah instance, serta data empiris berjenjang yang disertai repeated trials dan pengukuran pemanfaatan sumber daya (CPU dan memori) untuk mendukung interpretasi kausal atas hasil yang diperoleh.

Berdasarkan latar belakang tersebut, penelitian ini bertujuan untuk: (1) merancang dan mengimplementasikan dua prototipe sistem tryout yang identik secara fungsional dengan arsitektur monolitik dan microservices; (2) melakukan pengujian performansi (*response time*, *throughput*, *latency*) pada kedua arsitektur menggunakan metode *load testing* pada berbagai tingkat *concurrency*; (3) menganalisis perilaku skalabilitas horizontal arsitektur microservices; dan (4) merumuskan rekomendasi titik migrasi yang tepat berdasarkan data empiris yang diperoleh.

Arsitektur Monolitik dan Microservices. Arsitektur monolitik adalah pendekatan pengembangan perangkat lunak di mana seluruh komponen aplikasi antarmuka pengguna, logika bisnis, dan akses data digabungkan dalam satu basis kode dan dijalankan sebagai satu unit proses tunggal. Sebaliknya, arsitektur microservices memecah aplikasi menjadi kumpulan layanan kecil yang independen, masing-masing bertanggung jawab atas satu fungsi bisnis spesifik, memiliki basis data sendiri, dan berkomunikasi melalui protokol jaringan seperti REST API atau gRPC.

Analisis terhadap penelitian-penelitian terdahulu menunjukkan bahwa arsitektur monolitik lebih cocok digunakan untuk aplikasi dengan jumlah pengguna kecil, sedangkan arsitektur *microservice* menawarkan skalabilitas dan reliabilitas yang lebih baik untuk aplikasi dengan beban akses tinggi (Sanjaya & Murnawan, 2024). Studi komparatif berskala besar turut menegaskan bahwa *microservice* secara umum unggul dalam metrik skalabilitas meski monolitik tetap kompetitif pada beban rendah dengan kompleksitas operasional yang jauh lebih rendah (Blinowski et al., 2022). Perbandingan karakteristik arsitektural antara kedua pendekatan mencakup *coupling* antar-modul, unit *deployment*, dan strategi penskalaan tim pengembang dibahas secara komprehensif dalam studi analisis desain arsitektur perangkat lunak skalabel (Kamisetty et al., 2023), serta pada studi perbandingan langsung antara microservices dan monolith dari aspek pemeliharaan dan kompleksitas pengembangan (Sastrawan & Hendra Suputra, 2023).

Perbandingan Performansi Empiris Monolitik vs. Microservices. Perbandingan performansi empiris antara arsitektur monolitik dan microservices

telah banyak dilakukan pada berbagai domain aplikasi. Studi berbasis containerisasi Docker melaporkan bahwa microservices unggul pada *throughput* dalam kondisi beban tinggi, sementara monolitik masih unggul pada *latency* respons untuk beban rendah (Dirgantara et al., 2024). Penelitian pada aplikasi dengan volume transaksi tinggi menemukan pola yang konsisten: microservices menunjukkan keunggulan signifikan seiring peningkatan beban, sementara monolitik lebih unggul pada kondisi beban rendah karena tidak adanya *overhead* komunikasi jaringan (Marieska et al., 2025). Temuan serupa turut dilaporkan pada studi perbandingan performa aplikasi transaksi, yang mengonfirmasi bahwa titik impas (*break-even point*) antara kedua arsitektur sangat bergantung pada karakteristik beban spesifik aplikasi yang diuji (Lauwren & Setianto, 2024). Pola inilah yang menjadi acuan utama dalam merancang skenario pengujian bertingkat pada penelitian ini (*concurrency* 5 hingga 100).

Pola dan Strategi Migrasi Microservices. Proses migrasi dari sistem monolitik ke microservices umumnya dilakukan secara bertahap melalui identifikasi kandidat layanan berdasarkan batas domain bisnis (*bounded context*) dan tabel basis data yang digunakan. Penelitian pada Sistem Informasi Akademik (SIKAD) di sebuah institut teknologi menerapkan enam tahapan migrasi berdasarkan Microservices Migration Pattern, mencakup analisis subsistem monolitik terhadap fungsi bisnis, hingga evaluasi kesiapan rancangan menggunakan Microservices Scorecard, dan memperoleh skor kesiapan 81,7% yang dikategorikan sebagai kesiapan tinggi (Ramadhan Putra et al., 2025). Tinjauan literatur sistematis mengenai migrasi monolitik ke *microservice* merangkum tahapan umum yang meliputi identifikasi *bounded context*, dekomposisi basis data, dan strategi *stranglerfig* untuk migrasi bertahap tanpa mengganggu layanan yang sedang berjalan (Hassan et al., 2024), sementara tinjauan pola *deployment* dan komunikasi *microservice* menekankan pentingnya pemilihan granularitas layanan yang tepat sejak tahap perancangan (Karabey Aksakalli et al., 2021).

Studi penerapan arsitektur microservices asyn-chronous pada aplikasi *smart village* menunjukkan bahwa pola komunikasi *non-blocking* dapat mengurangi *latency* kumulatif dibandingkan pola *synchronous request-response* pada alur bisnis yang melibatkan banyak layanan (Asri et al., 2022). Pada domain perbankan, evaluasi performa *middleware* selama proses transisi dari arsitektur monolitik ke microservices menemukan bahwa pemilihan teknologi *middleware* turut memengaruhi signifikansi *overhead* migrasi, khususnya pada transaksi yang melibatkan banyak layanan sekaligus (Fauziah & Surantha, 2026), sebuah temuan yang relevan dengan hasil pengujian *endpoint* attempt/submit pada penelitian ini (lihat subbab 3.2).

Skalabilitas dan *Auto-scaling* pada Infrastruk-

tur *Cloud*. Skalabilitas horizontal pada arsitektur *microservices* umumnya direalisasikan melalui mekanisme *auto-scaling* pada *platform* orkestrasi *container* seperti Kubernetes. Penelitian mengenai *tuning* Horizontal Pod Autoscaler (HPA) menunjukkan bahwa parameter ambang batas (*threshold*) CPU/memori yang tidak tepat dapat menyebabkan efek berkebalikan, yaitu penurunan performa akibat *overhead* penjadwalan ulang pod yang terlalu sering (Augustyn et al., 2024). Tinjauan komprehensif terhadap teknik *auto scaling* pada lingkungan *cloud* dan *edge/fog computing* mengelompokkan strategi penskalaan menjadi pendekatan reaktif dan prediktif, dengan masing-masing memiliki *trade-off* antara responsivitas dan stabilitas sistem (Dogani et al., 2023). Pendekatan berbasis data (*data-driven*) untuk *auto-scaling* aplikasi *cloud* menunjukkan bahwa prediksi beban yang akurat dapat mengurangi latensi penskalaan dibandingkan pendekatan berbasis ambang batas statis (Swethasri Kavuri, 2024), sejalan dengan skema prediksi beban yang dapat dikustomisasi untuk Kubernetes (Mondal et al., 2023).

Tinjauan literatur mengenai *load-balancing* dan *auto-scaling* pada arsitektur *container microservices* turut mengidentifikasi bahwa keterbatasan sumber daya komputasi fisik merupakan salah satu faktor penghambat utama efektivitas *auto-scaling* (Rabiu et al., 2022), sebuah temuan yang secara langsung relevan dengan hasil pengujian skalabilitas horizontal pada penelitian ini (subbab 3.3). Studi penggabungan Kubernetes Cluster Autoscaler dengan Auto Scaling Group pada penyedia layanan *cloud* menunjukkan peningkatan efisiensi penskalaan dinamis dibandingkan penggunaan salah satu mekanisme saja (Romanov et al., 2025). Algoritma penjadwalan (*scheduling*) pada Kubernetes turut memengaruhi distribusi beban antar-*node*, sebagaimana dirangkum dalam survei komprehensif mengenai algoritma penjadwalan Kubernetes (Senjab et al., 2023). Pendekatan hybrid yang menggabungkan model prediktif dan reaktif untuk *auto-scaling* aplikasi kontainerisasi dilaporkan mampu menyeimbangkan kecepatan respons dan efisiensi biaya sumber daya dibandingkan pendekatan tunggal (Vu et al., 2022).

Metode Pengujian Performansi (Load Testing). *Load testing* merupakan teknik *performance testing* yang digunakan untuk mengukur respons sistem terhadap berbagai kondisi beban, dengan tujuan menentukan bagaimana perilaku perangkat lunak ketika diakses oleh banyak pengguna secara bersamaan (Permatasari et al., 2020). Parameter yang umum diukur dalam pengujian ini meliputi *throughput*, *response time*, *latency*, *error rate*, dan *resource utilization* (CPU serta memori), sebagaimana diterapkan dalam pengujian web *server* pada layanan *cloud computing* menggunakan Apache Jmeter (Fandy et al., 2022). Studi serupa pada pengujian performa aplikasi BANDUNGTANGINAS menggunakan JMeter turut mengonfirmasi bahwa *response time* dan

throughput merupakan indikator utama dalam menilai kesiapan suatu aplikasi menghadapi beban produksi (Alam & Dewi, 2022).

Penelitian evaluasi performa *website* menggunakan skenario *virtual user* berjenjang dari 50 hingga 5000 pengguna melaporkan bahwa sistem dapat berjalan stabil hingga 200 pengguna simultan tanpa error, namun *response time* meningkat tajam melampaui 60 detik dan error mulai muncul ketika beban melampaui 300 pengguna simultan, yang mengindikasikan tercapainya batas kapasitas performa *server* (Annisa et al., 2026). Studi serupa pada pengujian *website* politeknik menemukan bahwa optimasi konfigurasi *server* dapat meningkatkan kapasitas pengujian dari 50 pengguna menjadi 300 pengguna pada layanan *shared hosting*, dan hingga 1000 pengguna pada layanan VPS hosting (Setiawan & Enda, 2025). Prinsip pengukuran yang konsisten pada studi-studi tersebut *throughput*, *response time*, *latency*, dan *resource utilization* (Alam & Dewi, 2022; Annisa et al., 2026; Permatasari et al., 2020; Setiawan & Enda, 2025) diadopsi dalam penelitian ini menggunakan tool *auto-cannon* sebagai alat *load testing* berbasis Node.js.

Studi Terdahulu tentang Performansi Migrasi. Penelitian pada sistem informasi Dana Pensiun Lembaga Keuangan (DPLK) mengidentifikasi kandidat *microservice* melalui enam tahapan migrasi dan menyimpulkan bahwa performa aplikasi web *microservice* lebih baik dibanding aplikasi web monolitik karena mampu menerima load yang lebih besar (Munawar & Hodijah, 2018). Penelitian mengenai resiliensi sistem informasi berbasis *microservice* turut melaporkan bahwa pendekatan *microservice* memiliki performa lebih baik dibandingkan pendekatan monolitik, khususnya ketika dikombinasikan dengan teknologi orkestrasi *container* dan *caching* terdistribusi (Suryotrisongko, 2017).

Berbeda dengan temuan-temuan tersebut yang cenderung menyimpulkan keunggulan *microservices* secara umum, penelitian ini memberikan perspektif tambahan dengan mengukur performansi pada level *endpoint* dan pola komunikasi spesifik (*single call* vs. *multi-hop* antar-layanan), sehingga dapat menunjukkan kondisi di mana monolitik justru masih unggul, khususnya pada beban rendah-menengah dan pada alur bisnis yang melibatkan banyak panggilan antar service.

Arsitektur Event-Driven untuk Optimasi Komunikasi Antar-Layanan. Untuk mengurangi *overhead* komunikasi sinkron (*chatty communication*) antar-layanan sebagaimana teridentifikasi pada bagian hasil penelitian ini (subbab 3.2), sejumlah studi merekomendasikan penerapan arsitektur *event-driven* sebagai alternatif komunikasi HTTP request-response langsung. Penelitian pada sistem e-commerce melaporkan bahwa penerapan message broker RabbitMQ yang dikombinasikan dengan kontainerisasi Docker mampu meningkatkan skalabilitas sistem melalui komunikasi asynchronous, sehingga men-

gurangi ketergantungan pada pola request-response langsung antar-layanan dan menurunkan *latency* kumulatif pada alur bisnis multi-service (Ramadhan Putra et al., 2025).

2. Metode

Penelitian ini menggunakan pendekatan eksperimental kuantitatif dengan membangun dua prototipe aplikasi sistem tryout yang identik secara fungsional namun berbeda arsitektur, kemudian membandingkan performansi keduanya melalui pengujian beban terkontrol pada infrastruktur komputasi yang sama.

2.1 Objek Penelitian: Sistem Tryout

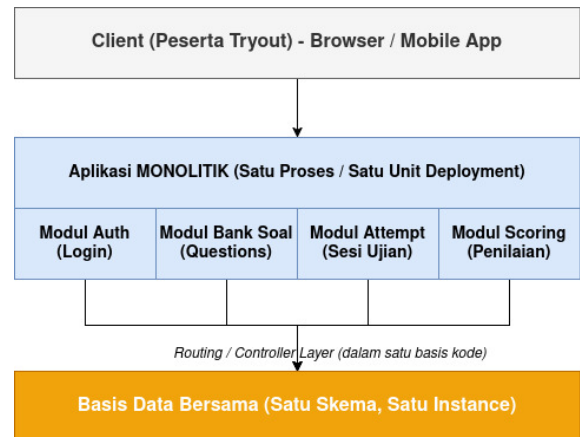
Sistem tryout yang dijadikan studi kasus memiliki empat modul fungsional utama: (1) Autentikasi pengguna (login), (2) Manajemen bank soal (pengambilan daftar soal pilihan ganda), (3) Manajemen sesi pengerjaan (attempt) meliputi memulai ujian dan mengirim jawaban, dan (4) Penilaian otomatis (scoring) yang menghitung skor berdasarkan jawaban yang dikirimkan.

2.2 Implementasi Prototipe

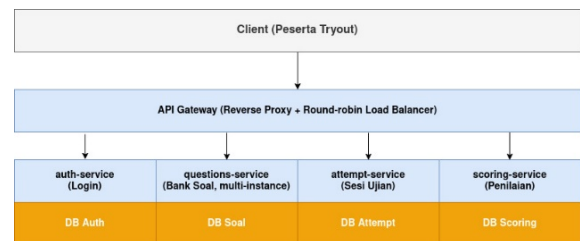
Prototipe monolitik diimplementasikan sebagai satu proses *server* tunggal yang menangani seluruh modul di atas menggunakan satu basis data bersama, sebagaimana diilustrasikan pada Gambar 1. Prototipe *microservices* diimplementasikan sebagai lima proses independen: *auth-service*, *question-service*, *attempt-service*, *scoring-service*, dan API Gateway sebagai titik masuk tunggal yang meneruskan (*proxy*) permintaan ke layanan yang sesuai, termasuk mendukung *load balancing* round-robin ke beberapa *instance question-service* untuk keperluan pengujian skalabilitas horizontal (Gambar 2). Alur bisnis *attempt/submit* pada arsitektur *microservices* melibatkan komunikasi HTTP antar-layanan (*attempt-service* memanggil *scoring-service*), sehingga merepresentasikan skenario multi-hop yang umum terjadi pada arsitektur terdistribusi.

Gambar 1 seluruh modul berjalan dalam satu proses dan berbagi satu basis data, sedangkan pada Gambar 2, API Gateway 4 layanan independen dan basis data terpisah per layanan (*database-per-service*).

Kedua prototipe diimplementasikan menggunakan *runtime* Node.js dengan pola REST API dan struktur layering (*routing-handler-data* layer) yang merepresentasikan pola Model-View-Controller (MVC) sebagaimana lazim digunakan pada *framework* seperti Laravel. Pemilihan Node.js sebagai lingkungan implementasi dilakukan karena keterbatasan akses jaringan pada lingkungan eksperimen terhadap repositori paket Composer/Packagist yang

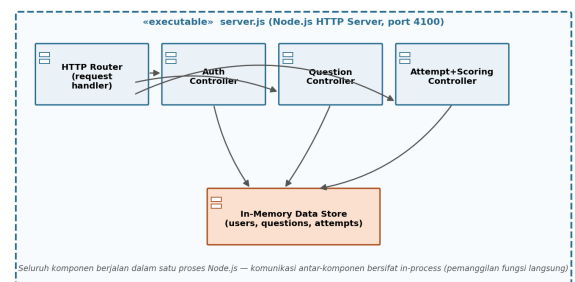


Gambar 1. Arsitektur monolitik sistem tryout



Gambar 2. Arsitektur *microservices* sistem

UML Component Diagram — Prototipe Monolitik



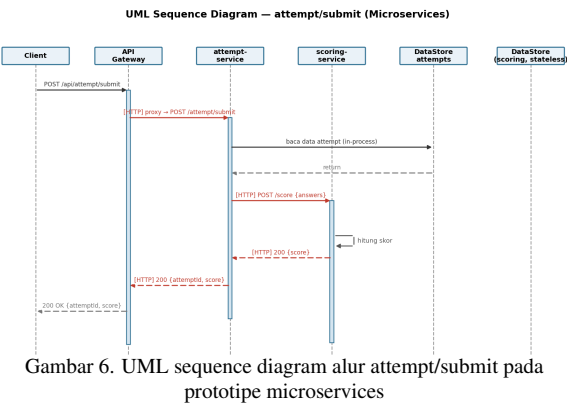
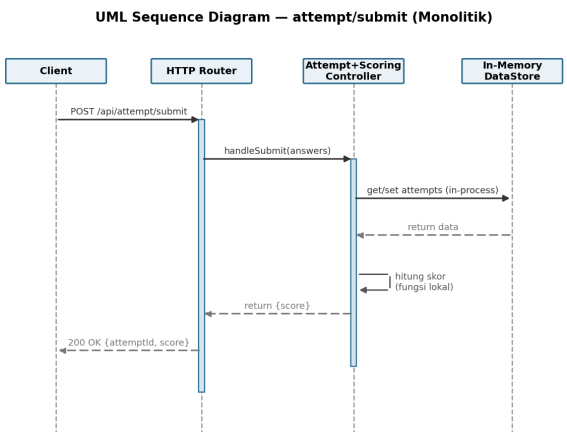
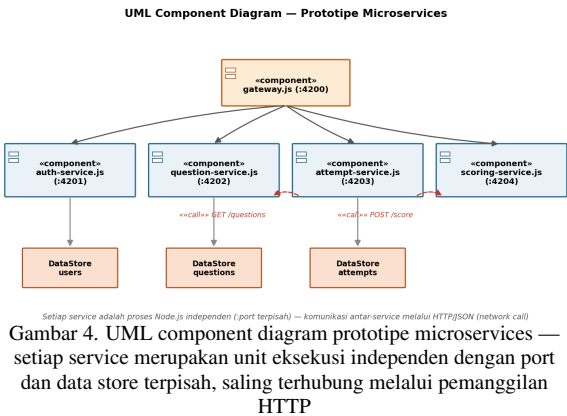
Gambar 3. UML component diagram prototipe monolitik — seluruh *controller* berada dalam satu unit eksekusi (*server.js*) dan mengakses data store yang sama secara in-process

dibutuhkan Laravel, sehingga logika bisnis, pola routing, dan pola komunikasi antar-layanan direplikasi secara setara tanpa bergantung pada *framework* tersebut. Keterbatasan ini menjadi salah satu batasan penelitian yang diuraikan pada bagian 2.6.

2.3 Pemodelan UML Prototipe

Untuk memperjelas struktur internal dan pola interaksi masing-masing prototipe, penelitian ini menyertakan dua jenis diagram UML: komponen diagram untuk menggambarkan struktur statis komponen perangkat lunak, dan sequence diagram untuk menggambarkan interaksi dinamis pada alur bisnis *attempt/submit* yang paling banyak menunjukkan perbedaan performansi antar-arsitektur (lihat subbab 3.2).

Perbedaan struktural yang tampak pada Gambar 3 dan Gambar 4 secara langsung memengaruhi pola interaksi saat *runtime*. Gambar 5 dan Gambar 6



menggambarkan sequence diagram untuk alur bisnis attempt/submit pada masing-masing arsitektur, yang menjadi dasar analisis pada subbab 3.2.

Pada Gambar 5, total 1 proses tanpa network round-trip antar modul dengan rata-rata latensi 8,92 milidetik dan c=20 dan seluruh interaksi bersifat in-process (pemanggilan fungsi).

Pada Gambar 6 melibatkan tiga network hop HTTP (gateway→attempt, attempt→scoring, dan jalur respons) dengan rata-rata latensi 23,42 milidetik dan c=20 yang berarti 2,6x lebih lambat dari monolitik.

Sequence diagram pada Gambar 6 secara eksplisit menunjukkan sumber tambahan latency pada

Tabel 2. Skenario pengujian beban

No	Skenario	Endpoint	Concurrency	Durasi
1	Baseline throughput bank soal	GET /questions	5, 10, 20, 50	4 detik/skenario
2	Alur bisnis multi-hop (submit + scoring)	POST /attempt/submit	20	4 detik
3	Skalabilitas horizontal (1 vs 2 instance)	GET /questions (via gateway)	50, 100	4 detik/skenario

arsitektur microservices: setiap panah bertanda [HTTP] merepresentasikan satu round-trip jaringan yang tidak ada pada versi monolitik (Gambar 5). Visualisasi ini mengonfirmasi secara struktural mengapa endpoint attempt/submit pada Tabel 5 (subbab 3.2) menunjukkan penurunan throughput yang jauh lebih besar dibandingkan endpoint bank soal pada Tabel 4 (subbab 4.1).

2.4 Skenario Pengujian

Tabel 2 merangkum skenario pengujian beban yang dijalankan pada kedua prototipe.

Pengujian dilakukan menggunakan tool load testing autocannon, dengan parameter yang diukur meliputi: rata-rata throughput (request/detik), latency rata-rata, latency persentil ke-99 (p99), total request yang berhasil diproses, serta jumlah error/timeout.

Perlu ditegaskan bahwa level concurrency yang diuji tidak seragam pada seluruh skenario di Tabel 2. Skenario 1 (baseline bank soal) dan skenario 2 (alur bisnis multi-hop) hanya diuji pada rentang concurrency 5, 10, 20, dan 50 (skenario 1) serta 20 (skenario 2), sedangkan concurrency 100 hanya digunakan pada skenario 3 (pengujian skalabilitas horizontal). Dengan demikian, perbandingan pada rentang concurrency 5 hingga 100 tidak dimaksudkan sebagai perbandingan tunggal yang seragam di seluruh skenario, melainkan sebagai gabungan data dari tiga skenario dengan cakupan concurrency yang berbeda-beda sebagaimana dirangkum pada Tabel 2, dan setiap perbandingan angka pada bagian Hasil dan Pembahasan selalu mengacu pada skenario serta level concurrency yang sama untuk kedua arsitektur.

Prosedur Pengulangan (Repeated Trials), Warm-up, dan Pengukuran Sumber Daya. Untuk meningkatkan reliabilitas pengukuran, setiap konfigurasi pengujian pada Tabel 2 (kombinasi arsitektur, endpoint, dan level concurrency) tidak dijalankan satu kali, melainkan melalui prosedur repeated trials sebagai berikut: (1) satu kali eksekusi warm-up autocannon berdurasi 2 detik pada konfigurasi yang sama untuk menstabilkan proses Node.js (JIT warm-up, alokasi memori awal, koneksi TCP) sebelum pengukuran, hasil warm-up ini dibuang dan

Tabel 3. Spesifikasi lingkungan eksperimen

Komponen	Spesifikasi
CPU	Intel(R) Xeon(R) @ 2.10GHz, 1 core (vCPU)
Memori	3,9 GiB RAM
Runtime	Node.js v22.22.2
Load testing tool	autocannon
Pola komunikasi	REST API / HTTP JSON

tidak dimasukkan ke dalam data; (2) dilanjutkan tiga kali pengulangan ($n=3$) pengujian aktual berdurasi 4 detik per pengulangan dengan jeda 500 milidetik antar-pengulangan untuk memberi waktu proses *server* kembali ke kondisi idle; (3) nilai yang dilaporkan pada Tabel 4, Tabel 5, dan Tabel 6 merupakan rata-rata (mean) dari tiga pengulangan tersebut, dilengkapi simpangan baku (standar deviasi, SD) pada teks pembahasan untuk menggambarkan variabilitas antar-pengulangan. Selama setiap pengulangan, pemanfaatan CPU pada level proses (seluruh proses Node.js yang terlibat, dihitung dari selisih waktu CPU pada `/proc/[pid]/stat` sebelum dan sesudah pengujian dibagi durasi wall-clock) serta penggunaan memori (Resident Set Size/RSS) turut direkam untuk mendukung interpretasi kausal atas hasil performansi, sebagaimana dirangkum pada Tabel 7 (subbab 3.5). Prosedur ini dijalankan pada mesin dan sesi yang sama dengan spesifikasi pada Tabel 3 untuk menjaga konsistensi kondisi eksperimen antar-pengulangan.

2.5 Lingkungan Eksperimen

Pada Tabel 3 merangkum spesifikasi lingkungan eksperimen yang digunakan pada kedua prototipe.

Seluruh pengujian dijalankan pada mesin yang sama untuk kedua arsitektur guna mengeliminasi bias perbedaan spesifikasi perangkat keras.

2.6 Batasan Penelitian

Penelitian ini memiliki beberapa batasan yang perlu diperhatikan dalam menginterpretasikan hasil: (1) prototipe diimplementasikan menggunakan Node.js sebagai representasi pola arsitektur REST/MVC, bukan implementasi Laravel/PHP secara langsung, karena keterbatasan akses jaringan eksperimen terhadap repositori paket; (2) lingkungan eksperimen hanya memiliki satu CPU core, sehingga hasil pengujian skalabilitas horizontal dipengaruhi oleh keterbatasan sumber daya komputasi dan tidak sepenuhnya merepresentasikan lingkungan *cloud* production dengan multi-core/multi-node; (3) basis data yang digunakan bersifat in-memory untuk menyederhanakan pengukuran murni terhadap *overhead* arsitektural, bukan *overhead* I/O basis data sesungguhnya; dan (4) pengujian dilakukan dalam durasi singkat (4 detik per skenario) untuk efisiensi eksperimen, bukan pengujian daya tahan (soak test)

jangka panjang.

3. Hasil dan Pembahasan

3.1 Perbandingan *Throughput* pada *Endpoint* Bank Soal

Tabel 4 menunjukkan hasil pengujian *throughput* dan *latency* pada *endpoint* GET /questions untuk kedua arsitektur pada berbagai tingkat concurrency.

Data pada Tabel 4 menunjukkan pola yang konsisten pada seluruh tingkat concurrency: arsitektur monolitik secara konsisten menghasilkan *throughput* lebih tinggi dibandingkan *microservices*, dengan selisih yang melebar seiring naiknya concurrency: berkisar 16%–21% pada concurrency 5–20, kemudian melebar tajam menjadi 45,8% pada concurrency 50 (rata-rata tiga pengulangan: monolitik $8.326,67 \pm 72,11$ request/detik berbanding *microservices* $4.515,75 \pm 90,15$ request/detik). Sebagaimana dibahas lebih lanjut pada subbab 3.5 (Tabel 7), pelebaran selisih ini konsisten dengan pemanfaatan CPU *microservices* yang naik jauh lebih cepat (dari 33,4% menjadi 80,4%) dibandingkan monolitik (12,5% menjadi 31,9%) pada rentang concurrency yang sama, sehingga *microservices* mulai mendekati batas kapasitas CPU pada beban tinggi, sedangkan monolitik masih memiliki kapasitas CPU tersisa. Temuan ini konsisten dengan karakteristik arsitektural *microservices* yang memperkenalkan *overhead* komunikasi jaringan melalui API Gateway, meskipun untuk *endpoint* sederhana yang hanya melibatkan satu layanan *backend*. Pola ini sejalan dengan studi berbasis containerisasi Docker yang juga melaporkan keunggulan monolitik pada beban rendah (Dirgantara et al., 2024), maupun studi pada aplikasi transaksi bervolume tinggi yang menemukan monolitik lebih unggul pada kondisi beban rendah karena ketiadaan *overhead* komunikasi jaringan (Marieska et al., 2025). Namun demikian, temuan ini berbeda dengan simpulan umum pada studi komparatif berskala besar yang menyatakan *microservice* secara konsisten unggul pada skenario beban tinggi (Blinowski et al., 2022), perbedaan ini kemungkinan disebabkan oleh rentang concurrency yang diuji pada penelitian ini pada skenario *baseline* (hingga 50 koneksi simultan; concurrency 100 hanya diuji pada skenario skalabilitas horizontal, lihat subbab 2.4) belum mencapai titik beban tinggi yang diamati pada studi tersebut.

Dari sisi *latency* p99 pada Tabel 4, *microservices* menunjukkan variabilitas antar-pengulangan (SD) yang jauh lebih tinggi dibandingkan monolitik, terutama pada concurrency 5 di mana rata-rata *latency* p99 mencapai 15,33 ms (SD $\pm 8,26$ ms; rentang 9–27 ms pada tiga pengulangan) dibandingkan 7,67 ms (SD $\pm 3,09$ ms) pada monolitik. Besarnya SD ini menegaskan pentingnya prosedur repeated trials pada subbab 2.4: satu kali pengujian tunggal pada konfigurasi ini berisiko memberikan gambaran *latency* p99 yang

Tabel 4. *Throughput* dan *latency endpoint* GET /questions

Concurrency	Arsitektur	Throughput (req/s)	Latency avg (ms)	Latency p99 (ms)	Total Request	Error
5	Monolitik	1.024,08	4,28	7,67	4.096	0
5	Microservices	849,00	5,38	15,33	3.396	0
10	Monolitik	2.033,08	4,39	7,33	8.131	0
10	Microservices	1.702,75	5,38	7,67	6.811	0
20	Monolitik	3.902,67	4,58	6,33	15.608	0
20	Microservices	3.093,83	5,97	10,00	12.373	0
50	Monolitik	8.326,67	5,52	8,67	33.306	0
50	Microservices	4.515,75	10,58	17,67	18.061	0

jauh menyimpang dari kondisi rata-rata. Variabilitas yang lebih tinggi pada *microservices* kemungkinan disebabkan oleh proses proxying pada API Gateway yang menambahkan satu hop jaringan tambahan dengan waktu tunggu yang lebih rentan terhadap fluktuasi penjadwalan proses pada mesin 1 vCPU, sebelum permintaan diteruskan ke *question-service*, sebuah pola yang juga diamati pada studi perbandingan performa aplikasi transaksi yang menyimpulkan bahwa titik impas (break-even point) antar-arsitektur sangat bergantung pada karakteristik beban spesifik aplikasi yang diuji (Lauwren, 2024).

3.2 Perbandingan pada Alur Bisnis Multi-Hop (attempt/submit)

Endpoint attempt/submit merepresentasikan alur bisnis nyata di mana *attempt-service* perlu memanggil *scoring-service* untuk menghitung skor peserta. Tabel 5 menunjukkan hasil pengujian pada *endpoint* ini dengan *concurrency* 20.

Hasil ini menunjukkan penurunan *throughput* yang jauh lebih signifikan (rata-rata tiga pengulangan: 48,7%, dari monolitik $2.215,17 \pm 17,92$ request/detik menjadi *microservices* $1.136,42 \pm 86,27$ request/detik) dibandingkan *endpoint* bank soal pada *concurrency* yang sebanding (16–21% pada *concurrency* 5–20) ketika alur bisnis melibatkan komunikasi antar-layanan berganda. Pengukuran CPU pada Tabel 7 turut menunjukkan bahwa *microservices* menggunakan CPU jauh lebih tinggi pada *endpoint* ini (65,0%) dibandingkan monolitik (16,9%), memperkuat bahwa penurunan *throughput* bukan semata akibat *latency* jaringan per permintaan, melainkan juga akibat beban komputasi tambahan dari proses-proses layanan yang saling memanggil. *Latency* p99 pada *microservices* meningkat rata-rata sekitar 3,6 kali lipat ($38,67 \pm 23,63$ ms vs $10,67 \pm 0,47$ ms) dibandingkan monolitik; simpangan baku p99 yang besar pada *microservices* ($\pm 23,63$ ms) turut mengindikasikan bahwa *latency* ekor (tail *latency*) pada alur multi-hop jauh lebih tidak stabil antar-pengulangan dibandingkan monolitik, sehingga estimasi tunggal (bukan repeated trials) berisiko tidak mewakili kondisi rata-rata. Temuan ini menegaskan

bahwa dampak performansi migrasi ke *microservices* tidak seragam di seluruh *endpoint*, melainkan sangat bergantung pada jumlah *network-hop* yang terlibat dalam satu alur bisnis. Semakin banyak layanan yang harus saling berkomunikasi untuk menyelesaikan satu permintaan, semakin besar pula *overhead latency* kumulatif yang timbul, sebuah fenomena yang dikenal dalam literatur sebagai “*chatty communication*” antar-*microservices*. Sebagai pembanding, evaluasi performa *middleware* pada domain perbankan turut melaporkan bahwa signifikansi *overhead* migrasi meningkat pada transaksi yang melibatkan banyak layanan sekaligus (Fauziah & Surantha, 2026) sejalan dengan tinjauan pola *deployment* dan komunikasi *microservice* yang menekankan pentingnya pemilihan granularitas layanan sejak tahap perancangan untuk meminimalkan *chatty communication* (Karabey Ak-sakalli et al., 2021). Kedua studi tersebut memperkuat interpretasi bahwa selisih performansi pada Tabel 5 bukan sekadar konsekuensi migrasi arsitektur secara umum, melainkan konsekuensi langsung dari jumlah panggilan antar-layanan pada alur bisnis yang spesifik.

3.3 Pengujian Skalabilitas Horizontal

Salah satu keunggulan teoretis utama *microservices* adalah kemampuan penskalaan independen per layanan. Pengujian ini membandingkan *throughput question-service* dengan 1 instance dibandingkan dengan 2 instance yang di-load balance secara round-robin oleh API Gateway. Tabel 6 menunjukkan hasil pengujian tersebut.

Bertentangan dengan ekspektasi umum bahwa penambahan instance akan meningkatkan *throughput* secara linear, hasil pengujian (rata-rata tiga pengulangan) justru menunjukkan *throughput* yang relatif tidak jauh berbeda, bahkan sedikit lebih rendah, pada skenario 2 instance dibandingkan 1 instance dengan *concurrency* yang sama: pada *concurrency* 50, 1 instance mencapai $4.327,83 \pm 390,98$ req/s dibandingkan $3.904,92 \pm 424,95$ req/s pada 2 instance, dan pada *concurrency* 100, 1 instance mencapai $4.466,67 \pm 56,58$ req/s dibandingkan $4.163,17 \pm 182,69$ req/s pada 2 instance, selisih pada kedua

Tabel 5. *Throughput* dan *latency endpoint attempt/submit* (concurrency 20)

Arsitektur	Throughput (req/s)	Latency avg (ms)	Latency p99 (ms)	Total Request	Network hop
Monolitik	2.215,17	8,53	10,67	8.859	0 (in-process)
Microservices	1.136,42	17,18	38,67	4.546	2 hop

Tabel 6. Hasil uji skalabilitas horizontal question-service

Skenario	Concurrency	Throughput (req/s)	Latency avg (ms)	Latency p99 (ms)
1 instance	50	4.327,83	11,15	33,67
1 instance	100	4.466,67	21,89	33,00
2 instance	50	3.904,92	12,46	34,33
2 instance	100	4.163,17	23,57	39,67

level concurrency tersebut (sekitar 7–10%) berada dalam rentang satu simpangan baku sehingga tidak dapat disimpulkan berbeda secara meyakinkan hanya dari data *throughput* semata. Untuk mendukung interpretasi kausal atas temuan ini, penelitian ini turut mengukur pemanfaatan CPU pada level proses (subbab 2.4), yang dirangkum pada Tabel 7 (subbab 3.5): pemanfaatan CPU pada seluruh konfigurasi 1 maupun 2 instance berada pada kisaran 79,5%–83,8%, mendekati saturasi 1 vCPU yang tersedia pada lingkungan eksperimen (Tabel 3). Data pengukuran CPU langsung ini mengindikasikan bahwa keterbatasan infrastruktur eksperimen berupa satu CPU core (1 vCPU) merupakan faktor pembatas utama tidak terealisasinya manfaat penambahan instance, sebab menjalankan proses tambahan (instance kedua *question-service* beserta proses API Gateway, *auth-service*, *attempt-service*, dan *scoring-service*) secara bersamaan pada satu core yang sama membuat seluruh proses berbagi kapasitas CPU yang tetap, sehingga kapasitas pemrosesan total sistem pada dasarnya tidak bertambah meski jumlah instance bertambah. Mekanisme rinci di balik saturasi CPU tersebut, yaitu apakah didominasi oleh context-switching *overhead* antar-proses, kontensi penjadwalan (*scheduling contention*), atau kombinasi keduanya, tetap bersifat interpretasi, karena penelitian ini mengukur CPU utilization dan memori pada level proses (agregat), bukan jumlah context switch pada level sistem operasi secara langsung (misalnya melalui *vmstat/pidstat*); dengan demikian, context-switching *overhead* sebaiknya dipahami sebagai penjelasan mekanisme yang paling mungkin dan konsisten dengan data CPU yang tersedia, bukan sebab yang telah dibuktikan secara langsung. Temuan ini konsisten dengan tinjauan literatur mengenai *load-balancing* dan *auto-scaling* pada arsitektur *container microservices*, yang mengidentifikasi keterbatasan sumber daya komputasi fisik sebagai salah satu faktor penghambat utama efektivitas *auto-scaling* (Rabiu et al., 2022). Fenomena serupa juga sejalan dengan temuan bahwa penalaan parameter *Horizontal Pod Autoscaler* (HPA) yang tidak tepat termasuk ketiadaan alokasi core yang memadai per-pod dapat menyebabkan *overhead* penjadwalan ulang yang kon-

traproduktif terhadap performa agregat (Augustyn et al., 2024).

Temuan ini merupakan kontribusi penting penelitian ini: skalabilitas horizontal *microservices* bukanlah keunggulan otomatis, melainkan sangat bergantung pada ketersediaan sumber daya komputasi fisik pada infrastruktur *cloud* yang mendasarinya. Pada infrastruktur *cloud* production dengan alokasi CPU core yang memadai per-*container* umumnya menggunakan orkestrasi seperti Kubernetes dengan mekanisme Cluster Autoscaler yang digabungkan dengan Auto Scaling Group (Romanov et al., 2025) serta algoritma penjadwalan (*scheduling*) yang mendistribusikan beban antar-*node* secara optimal (Senjab et al., 2023), penambahan instance *question-service* secara independen diperkirakan akan memberikan peningkatan *throughput* yang lebih linear, sejalan dengan temuan penelitian terdahulu yang melaporkan performa *microservice* lebih baik dalam menerima load besar ketika didukung sumber daya *cloud* yang memadai (Munawar & Hodi-jah, 2018). Kondisi pada penelitian ini justru menggarisbawahi pentingnya perencanaan kapasitas (*capacity planning*) yang matang sebelum migrasi ke *microservices* dilakukan, karena tanpa dukungan infrastruktur yang tepat, migrasi berpotensi menurunkan performansi alih-alih meningkatkannya.

3.4 Pemanfaatan CPU dan Memori Selama Pengujian

Sebagaimana dijelaskan pada subbab 2.4, setiap pengulangan pengujian turut merekam pemanfaatan CPU pada level proses (seluruh proses Node.js yang terlibat) dan penggunaan memori (RSS). Tabel 7 merangkum rata-rata kedua metrik tersebut untuk seluruh skenario pada Tabel 4, Tabel 5, dan Tabel 6, guna mendukung interpretasi kausal atas selisih performansi yang diamati dan mengonfirmasi (atau membatasi) klaim penyebab yang sebelumnya bersifat dugaan. (Keterangan kolom: C = concurrency; Mono = Monolitik.)

Data pada Tabel 7 menunjukkan pola yang konsisten dan secara langsung mendukung interpretasi pada subbab 3.1–3.3. Pada skenario *baseline*, peman-

Tabel 7. Rata-rata pemanfaatan CPU dan memori per skenario pengujian

Skenario	Konfigurasi	C	CPU (%)	Mem (MB)
Baseline	Monolitik	5	12,5	61,0
Baseline	Microservices	5	33,4	267,5
Baseline	Monolitik	10	12,1	64,6
Baseline	Microservices	10	39,3	275,4
Baseline	Monolitik	20	19,3	71,5
Baseline	Microservices	20	57,6	291,9
Baseline	Monolitik	50	31,9	85,7
Baseline	Microservices	50	80,4	302,1
Multi-hop	Monolitik	20	16,9	87,8
Multi-hop	Microservices	20	65,0	342,1
Scaling	1 instance	50	80,2	297,7
Scaling	1 instance	100	79,5	312,6
Scaling	2 instance	50	83,8	353,2
Scaling	2 instance	100	81,9	363,8

faatan CPU pada arsitektur *microservices* meningkat jauh lebih cepat seiring kenaikan concurrency (12,5% menjadi 80,4% dari concurrency 5 ke 50) dibandingkan monolitik (12,5% menjadi 31,9%), sejalan dengan fakta bahwa *microservices* menjalankan lima proses Node.js (*Gateway*, *auth-service*, *question-service*, *attempt-service*, *scoring-service*) yang seluruhnya bersaing memperebutkan satu core CPU, sementara monolitik hanya menjalankan satu proses. Pola ini menjelaskan mengapa selisih *throughput* antara kedua arsitektur pada Tabel 4 melebar tajam pada concurrency 50 (45,8%) dibandingkan pada concurrency rendah (16–21%): pada beban rendah, CPU belum menjadi faktor pembatas bagi *microservices*, namun pada beban tinggi *microservices* mulai mendekati saturasi CPU sementara monolitik masih memiliki kapasitas CPU tersisa. Penggunaan memori (RSS) pada *microservices* juga secara konsisten sekitar 3–4 kali lebih tinggi dibandingkan monolitik akibat *overhead* lima proses Node.js terpisah, meskipun beban memori absolut ini masih jauh di bawah kapasitas RAM 3,9 GiB pada Tabel 3 sehingga tidak menjadi faktor pembatas utama.

Pada skenario skalabilitas horizontal, pemanfaatan CPU berada pada kisaran 79,5%–83,8% pada seluruh konfigurasi, baik dengan 1 maupun 2 instance *question-service*. Kondisi CPU yang sudah mendekati saturasi 1 vCPU inilah yang secara langsung mendukung penjelasan bahwa tidak diperolehnya peningkatan *throughput* yang proporsional saat instance ditambah (Tabel 6) disebabkan oleh keterbatasan sumber daya komputasi fisik, bukan oleh cacat pada mekanisme *load balancing* round-robin pada API Gateway. Perlu dicatat bahwa pengukuran ini merupakan CPU utilization dan memori pada level proses (agregat), bukan penghitungan langsung terhadap jumlah context switch pada level sistem operasi (misalnya melalui *vmstat*/*pidstat*), sehingga

istilah “context-switching *overhead*” pada pembahasan subbab 3.3 tetap perlu dipahami sebagai interpretasi mekanisme yang paling mungkin di balik saturasi CPU tersebut, bukan sebagai sebab yang telah dibuktikan secara langsung melalui penghitungan context switch. Dengan data CPU utilization pada Tabel 7, klaim kausal pada subbab 3.3 kini didukung oleh bukti pengukuran langsung (CPU mendekati saturasi), sekalipun mekanisme rinciannya (context-switching vs. penjadwalan proses vs. kontensi I/O event loop) tetap bersifat interpretatif dan terbuka untuk divalidasi lebih lanjut menggunakan instrumentasi sistem operasi yang lebih rinci pada penelitian selanjutnya.

3.5 Pembahasan Umum dan Implikasi

Secara keseluruhan, hasil penelitian ini memberikan gambaran yang lebih nuansir (*nuanced*) dibandingkan generalisasi umum “*microservices* selalu lebih baik dari monolitik”. Tiga temuan utama dapat dirangkum sebagai berikut:

- Pada beban rendah hingga menengah dan alur bisnis sederhana (*single-service*), arsitektur monolitik unggul dalam *throughput* dan *latency* karena tidak ada *overhead* komunikasi jaringan.
- *Overhead microservices* meningkat signifikan pada alur bisnis yang melibatkan banyak network hop antar-layanan, sehingga desain *granularitas* atau tingkat kerincian layanan dan minimalisasi komunikasi antar-service (misalnya melalui pola *asynchronous messaging/event-driven*) menjadi krusial dalam migrasi.
- Manfaat skalabilitas *horizontal microservices* sepenuhnya bergantung pada kecukupan sumber daya komputasi infrastruktur *cloud*; tanpa alokasi CPU/memori yang memadai per-instance, penambahan instance dapat kontraproduktif.

Temuan-temuan ini sejalan dengan pandangan bahwa keputusan migrasi idealnya tidak dilakukan sekaligus di seluruh sistem, melainkan dimulai dari identifikasi satu fitur yang paling sering mengalami *bottleneck* sebagai *pilot project*, sebelum diperluas ke modul lain secara bertahap. Untuk kasus sistem *tryout*, modul yang paling berpotensi diuntungkan dari migrasi *microservices* adalah *attempt-service* dan *scoring-service*, karena modul inilah yang mengalami lonjakan beban paling ekstrem saat jadwal ujian berlangsung serentak, sementara modul seperti manajemen bank soal (yang relatif statis dan jarang berubah) dapat tetap dipertahankan dalam arsitektur monolitik atau di-cache secara agresif tanpa perlu dipisah menjadi layanan tersendiri.

Perspektif bertahap ini juga selaras dengan studi analisis desain arsitektur perangkat lunak skalabel yang menekankan bahwa keputusan *granular-*

itas layanan dan strategi penskalaan tim pengembang perlu disesuaikan dengan karakteristik beban modul, bukan diterapkan secara seragam di seluruh sistem (Kamisetty et al., 2023). Sementara itu, studi perbandingan langsung antara *microservices* dan monolitik dari aspek pemeliharaan menyimpulkan bahwa kompleksitas operasional tambahan pada *microservices* baru sepadan dengan manfaatnya ketika beban akses benar-benar tinggi dan tim pengembang cukup besar untuk mengelola layanan secara independen (Sastrawan & Hendra Suputra, 2023), sebuah pertimbangan yang relevan mengingat pada penelitian ini justru arsitektur monolitik masih unggul pada beban rendah-menengah. Dengan demikian, temuan penelitian ini memperkuat pandangan bahwa migrasi ke *microservices* bukan keputusan biner, melainkan trade-off yang perlu dipertimbangkan per-modul berdasarkan pola beban, kompleksitas alur bisnis, dan kapasitas infrastruktur yang tersedia.

4. Kesimpulan

Penelitian ini berhasil membangun dua prototipe sistem *tryout* yang identik secara fungsional dengan arsitektur monolitik dan *microservices*, serta melakukan pengujian performansi dan skalabilitas secara empiris menggunakan *load testing* pada berbagai tingkat *concurrency*. Hasil penelitian menunjukkan bahwa arsitektur monolitik memberikan *throughput* 16%–49% lebih tinggi (rata-rata tiga pengulangan) dan *latency* lebih rendah pada beban rendah-menengah dibandingkan *microservices* untuk *end-point* tunggal, dengan selisih performansi yang semakin besar (48,7%) pada alur bisnis yang melibatkan komunikasi antar-layanan berganda; pengukuran pemanfaatan CPU pada level proses mengonfirmasi bahwa pelebaran selisih ini konsisten dengan lebih cepatnya *microservices* mendekati saturasi CPU pada beban tinggi. Pengujian skalabilitas *horizontal* juga mengungkap temuan penting bahwa manfaat penambahan *instance* layanan sangat bergantung pada ketersediaan sumber daya CPU pada infrastruktur *cloud* yang digunakan, dan tidak dijamin memberikan peningkatan performa linear tanpa perencanaan kapasitas yang tepat. Seluruh temuan kuantitatif di atas berlaku secara spesifik pada lingkungan eksperimen yang digunakan pada penelitian ini, yaitu mesin dengan satu CPU core (1 vCPU) sebagaimana dirinci pada Tabel 3, dan tidak dimaksudkan sebagai generalisasi terhadap infrastruktur *cloud* secara umum yang pada praktiknya dapat memiliki alokasi CPU multi-core/multi-node, mekanisme *auto-scaling*, serta orkestrasi *container* yang berbeda. Validasi lebih lanjut pada lingkungan *cloud* multi-core/multi-node diperlukan sebelum kesimpulan mengenai skalabilitas horizontal ini digeneralisasikan di luar konteks eksperimen single-vCPU tersebut.

Berdasarkan temuan ini, dapat disimpulkan bahwa migrasi ke *microservices* sebaiknya dilakukan secara selektif dan bertahap, dimulai dari modul-modul yang paling sering mengalami *bottleneck* beban tinggi, serta didukung oleh perencanaan infrastruktur *cloud* yang memadai (jumlah core, orkestrasi *autoscaling* menggunakan mekanisme seperti *Kubernetes Cluster Autoscaler* (Romanov et al., 2025), dan strategi *caching*) agar manfaat skalabilitas dapat direalisasikan secara nyata. Untuk mengatasi *overhead* komunikasi antar-layanan pada alur bisnis multi-hop sebagaimana ditemukan pada subbab 3.2, penerapan arsitektur event-driven menggunakan *message broker* seperti *RabbitMQ* (Ramadhan Putra et al., 2025) dapat menjadi strategi mitigasi yang relevan untuk penelitian dan implementasi selanjutnya. Penelitian selanjutnya disarankan untuk menguji kedua arsitektur pada infrastruktur *cloud multi-node* yang sesungguhnya (misalnya menggunakan *Kubernetes* dengan alokasi multi-core dan HPA yang dikonfigurasi secara tepat (Augustyn et al., 2024; Mondal et al., 2023)), menggunakan implementasi *framework* produksi (seperti Laravel dengan PHP-FPM, sebagaimana pendekatan evaluasi *middleware* pada domain perbankan (Fauziah & Surantha, 2026)), basis data *non-in-memory*, serta pengujian daya tahan (*soak test*) dalam durasi lebih panjang untuk memvalidasi temuan pada kondisi yang lebih merepresentasikan lingkungan produksi.

Daftar Pustaka

- Alam, E. N., & Dewi, F. (2022): *PERFORMANCE TESTING ANALYSIS OF BANDUNGTANGINAS APPLICATION WITH JMETER*, International Journal of Innovation in Enterprise System, 6(2), 146–155. <https://doi.org/10.25124/ijies.v6i01.165>
- Annisa, A. R., Ardiana, M., Rahayu, P. N., Brian, T., & Abu Jami'in, M. (2026): *Performance Evaluation of Web Applications Using JMeter Load Testing for Server Capacity and Response Efficiency*, In Journal of Applied Informatics and Computing (JAIC) (Vol. 10, Number 1). <http://jurnal.polibatam.ac.id/index.php/JAIC>
- Asri, S. A., Gede, N., Astawa, A., Agung, G., Sunaya, M., Riyan, M., Nugroho, A., & Setiawan, W. (2022): *Implementation of Asynchronous Microservices Architecture on Smart Village Application*, 12(3).
- Augustyn, D. R., Wyciślik, Ł., & Sojka, M. (2024): *Tuning a Kubernetes Horizontal Pod Autoscaler for Meeting Performance and Load Demands in Cloud Deployments*, Applied Sciences (Switzerland), 14(2). <https://doi.org/10.3390/app14020646>
- Blinowski, G., Ojdowska, A., & Przybyłek, A. (2022): *Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation*,

- IEEE Access, 10, 20357–20374. <https://doi.org/10.1109/ACCESS.2022.3152803>
- Dirgantara, D. P., Kusumo, D. S., & Utomo, R. G. (2024): *DOCKER-BASED MONOLITHIC AND MICROSERVICES ARCHITECTURE PERFORMANCE COMPARISON*, J. Tek. Inform. (JUTIF), 5(2), 357–365.
- Dogani, J., Namvar, R., & Khunjush, F. (2023): *Auto-scaling techniques in container-based cloud and edge/fog computing: Taxonomy and survey*, Computer Communications, 209, 120–150.
- Fandy, Rosmasari, & Putra, G. M. (2022): *Pengujian Kinerja Web Server Atas Penyedia Layanan Elastic Cloud Compute (EC2) Pada Amazon Web Services (AWS)*, Adopsi Teknologi Dan Sistem Informasi (ATASI), 1(1), 21–35. <https://doi.org/10.30872/atasi.v1i1.45>
- Fauziah, R., & Surantha, N. (2026): *Evaluating Middleware Performance in the Transition from Monolithic to Microservices Architecture for Banking Applications*, Electronics (Switzerland), 15(1). <https://doi.org/10.3390/electronics15010221>
- Hassan, H., Abdel-Fattah, M. A., & Mohamed, W. (2024): *Migrating from Monolithic to Microservice Architectures: A Systematic Literature Review*, In IJACSA International Journal of Advanced Computer Science and Applications (Vol. 15, Number 10). www.ijacsa.thesai.org
- Kamisetty, A., Narsina, D., Rodriguez, M., Kothapalli, S., Chandra, J., & Gummadi, S. (2023): *Microservices vs. Monoliths: Comparative Analysis for Scalable Software Architecture Design*, Engineering International, 11(2).
- Karabey Aksakalli, I., Çelik, T., Can, Ahmet. B., & Tekinerdoğan, B. (2021): *Deployment and communication patterns in microservice architectures: A systematic literature review*, Journal of Systems and Software, 180, 111014.
- Lauwren, A. J., & Setianto, Y. b D. (2024): *Microservice and monolith performance comparison in transaction application*, Proxies: Jurnal Informatika, 5(2), 86–105.
- Marieska, M. D., Yunanta, A., Aulia, H., Utami, A. S., & Rizqie, M. Q. (2025): *Performance Comparison of Monolithic and Microservices Architectures in Handling High-Volume Transactions*, Jurnal RESTI, 9(3), 594–600. <https://doi.org/10.29207/resti.v9i3.6183>
- Mondal, S. K., Wu, X., Kabir, H. M. D., Dai, H. N., Ni, K., Yuan, H., & Wang, T. (2023): *Toward Optimal Load Prediction and Customizable Autoscaling Scheme for Kubernetes*, Mathematics, 11(12). <https://doi.org/10.3390/math11122675>
- Munawar, G., & Hodijah, A. (2018): *Analisis Model Arsitektur Microservice Pada Sistem Informasi DPLK*, Sinkron: Jurnal dan Penelitian Teknik Informatika, 3(1), 232–238.
- Permatasari, D. I., Ardani, M., Ma’ulfa, A. Y., Ilhami, N., Pratama, S. G., Astuti, S. R. D., & Nufalita, N. W. (2020): *Pengujian Aplikasi Menggunakan Metode Load Testing dengan Apache Jmeter pada Sistem Informasi Pertanian*, Jurnal Sistem Dan Teknologi Informasi (JustIN), 8(1), 135–139.
- Rabiu, S., Yong, C. H., Mashita, S., & Mohamad, S. (2022): *A Cloud-Based Container Microservices: A Review on Load-Balancing and Auto-Scaling Issues*, In International Journal of Data Science (Vol. 3, Number 2).
- Ramadhan Putra, K., Rizal Nurjaman, A., & Haniifah, A. F. (2025): *Perancangan dan Evaluasi Arsitektur Microservices Menggunakan Microservices Migration Pattern dan Microservices Scorecard*, In Jurnal Aplikasi Teknologi Informasi dan Manajemen (JATIM) (Vol. 6, Number 2).
- Romanov, O., Mankivskiy, V., Romanov, A., Nesterenko, M., & Pidpaly, O. (2025): *Performance assessing of dynamic scaling of containerized applications when using Kubernetes cluster autoscaler and auto scaling group together*, Advanced Smart Information and Communication Technology and Systems.
- Sanjaya, & Murnawan. (2024): *Pemanfaatan Arsitektur Microservice Untuk Peningkatan Performansi Website Lomba Nasional Kreativitas Mahasiswa*, Jurnal Teknologi Informasi Dan Ilmu Komputer (JTIIK), 11(2), 377–384. <https://doi.org/10.25126/jtiik20241128307>
- Sastrawan, G. G., & Hendra Suputra, I. P. G. (2023): *Perbandingan Antara Arsitektur Perangkat Lunak Microservices dan Monolith*, JELIKU (Jurnal Elektronik Ilmu Komputer Udayana), 11(4), 751–756.
- Senjab, K., Abbas, S., Ahmed, N., & Khan, A. ur R. (2023): *A survey of Kubernetes scheduling algorithms*, In Journal of Cloud Computing (Vol. 12, Number 1). Springer Science and Business Media Deutschland GmbH. <https://doi.org/10.1186/s13677-023-00471-1>
- Setiawan, H., & Enda, D. (2025): *Pengujian Load Testing Website Jurusan Teknik Informatika Politeknik Negeri Bengkalis*, JEKIN - Jurnal Teknik Informatika, 5(1), 1–12. <https://doi.org/10.58794/jekin.v5i1.820>
- Suryotrisongko, H. (2017): *Arsitektur Microservice untuk Resiliensi Sistem Informasi*.
- Swethasri Kavuri. (2024): *Integrating Kubernetes Autoscaling for Cost Efficiency in Cloud Services*, International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 10(5), 480–502. <https://doi.org/10.32628/cseit241051038>
- Vu, D. D., Tran, M. N., & Kim, Y. (2022): *Predictive Hybrid Autoscaling for Containerized Applications*, IEEE Access, 10, 109768–109778. <https://doi.org/10.1109/ACCESS.2022.3214985>