Jurnal Indonesia : Manajemen Informatika dan Komunikasi
https://journal.stmiki.ac.id

E-ISSN: 2723-7079 | P-ISSN: 2776-8074
Vol. 7 No. 3 (2026) | September
OPEN ACCESS

https://doi.org/10.63447/jimik.v7i3.2040

Penerapan *Microservices* dengan Docker Swarm pada Sistem Informasi Manajemen Rumah Sakit (SIMRS): Strategi Migrasi *Monolith* dan Evaluasi Performa

Farizal Ginanjar [1*], Rizqia Fauziah Rachma [2], Wahid Diyono [3], Mustawi Farhan [4], Windu Gata [5]

[1*,2,3,4,5] Magister Ilmu Komputer, Universitas Nusa Mandiri, Kota Jakarta Timur, Daerah Khusus Ibukota Jakarta, Indonesia.

Email: 14250014@nusamandiri.ac.id [1*], 14250002@nusamandiri.ac.id [2], 14250036@nusamandiri.ac.id [3], 14250015@nusamandiri.ac.id [4], windu@nusamandiri.ac.id [5]

Histori Artikel:
Dikirim 10 Juni 2026; *Diterima dalam bentuk revisi* 15 Juni 2026; *Diterima* 30 Juli 2026; *Diterbitkan* 10 September 2026.

Abstrak

Transformasi digital sektor layanan kesehatan di Indonesia menuntut Sistem Informasi Manajemen Rumah Sakit (SIMRS) yang skalabel, andal, dan mudah diintegrasikan. Mayoritas SIMRS yang ada masih berbasis arsitektur monolitik sehingga rentan single point of failure, sulit di-scale per modul, dan menimbulkan downtime setiap kali terjadi pembaruan. Penelitian ini bertujuan menelaah strategi migrasi monolith menuju microservices serta mengevaluasi performa implementasi deployment berbasis Docker dan Docker Swarm pada prototipe SIMRS yang mengandung dua belas modul fungsional. Metode yang digunakan adalah kombinasi Systematic Literature Review untuk menyaring pola dekomposisi terkini dan studi eksperimental terhadap cluster dua node yang menjalankan empat belas stack dan tiga puluh lima service dengan total tujuh puluh enam container berjalan. Hasil pengukuran menunjukkan rata-rata response time HTTP berada pada rentang 0,6–5,1 milidetik untuk frontend dan 0,8–3,0 milidetik untuk backend (dengan satu outlier billing-frontend 82,4 ms akibat saturasi memori), dengan utilisasi CPU per container mayoritas di bawah dua persen serta konsumsi memori yang stabil pada batas alokasi. Penggunaan Docker Swarm terbukti mendukung replikasi high-availability, rolling update, serta self-healing otomatis pada kondisi operasional normal; namun satu insiden kegagalan rolling update (billing-backend turun ke 0/2 replica hingga intervensi manual) menunjukkan bahwa health check image-level yang andal merupakan prasyarat perilaku zero-downtime yang dijanjikan. Penelitian ini menyimpulkan bahwa pendekatan microservices dengan Docker Swarm layak diadopsi untuk SIMRS di Indonesia dengan trade-off kompleksitas operasional yang terkelola.

Kata Kunci: Docker; Docker Swarm; Migrasi Monolith; Microservices; SIMRS.

Abstract

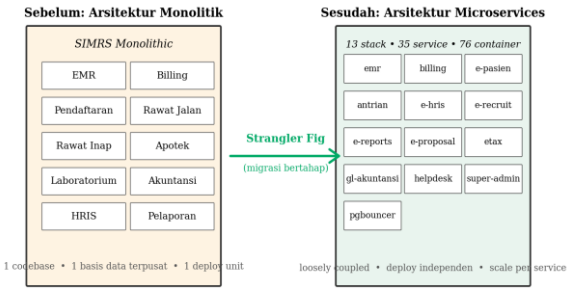
Digital transformation in Indonesia's healthcare sector demands a Hospital Information System (SIMRS) that is scalable, reliable, and easy to integrate. Most existing SIMRS still rely on a monolithic architecture, making them vulnerable to single points of failure, hard to scale per module, and prone to downtime during updates. This study aims to examine strategies for migrating monolithic architectures to microservices and to evaluate the performance of a Docker- and Docker Swarm-based deployment on a SIMRS prototype consisting of twelve functional modules. The method combines a Systematic Literature Review on recent decomposition patterns with an experimental study on a two-node cluster running fourteen stacks and thirty-five services for a total of seventy-six active containers. Measurements show that the average HTTP response time ranges from 0.6 to 5.1 milliseconds for frontends (with a single billing-frontend outlier of 82.4 ms attributable to memory saturation) and from 0.8 to 3.0 milliseconds for backends, with per-container CPU utilization mostly below two percent and stable memory consumption within allocated limits. Docker Swarm proves effective in supporting high-availability replication, rolling updates, and automatic self-healing under normal operating conditions; one observed rolling-update failure (billing-backend dropping to 0/2 replicas until manual intervention) shows that reliable image-level health checks are a prerequisite for the promised zero-downtime behavior. The study concludes that microservices with Docker Swarm are a feasible adoption strategy for SIMRS in Indonesia, with manageable operational trade-offs.

Keyword: Docker; Docker Swarm; Hospital Information System; Microservices; Monolith Migration.

Jurnal Indonesia : Manajemen Informatika dan Komunikasi
https://journal.stmiki.ac.id
E-ISSN: 2723-7079 | P-ISSN: 2776-8074
Vol. 7 No. 3 (2026) | September
OPEN ACCESS
https://doi.org/10.63447/jimik.v7i3.2040

1. Pendahuluan

Transformasi digital pada sektor layanan kesehatan di Indonesia mengalami percepatan sejak terbitnya Peraturan Menteri Kesehatan Nomor 24 Tahun 2022 yang mewajibkan penerapan Rekam Medis Elektronik, dilengkapi inisiatif Satu Sehat dan *bridging*. Sistem Informasi Manajemen Rumah Sakit (SIMRS) menjadi tulang punggung operasional rumah sakit, menangani seluruh alur pelayanan dari pendaftaran, rawat jalan, rawat inap, IGD, penunjang medis, hingga penagihan klaim. Kebutuhan akan arsitektur SIMRS yang handal, skalabel, dan mudah diintegrasikan menjadi sangat penting. Mayoritas SIMRS di Indonesia saat ini masih dibangun di atas arsitektur monolitik dengan satu *codebase* dan basis data terpusat. Arsitektur ini menimbulkan keterbatasan yang serupa dengan temuan studi migrasi sebelumnya (Al-Debagy & Martinek, 2018; Dragoni *et al.*, 2017; Fritzsch *et al.*, 2019): skalabilitas terbatas karena tidak dapat di-*scale* selektif per modul, *maintainability* rendah karena perubahan kecil memaksa *downtime* global, integrasi dengan sistem eksternal menjadi rigid, serta munculnya risiko *single point of failure* yang dapat menghentikan seluruh operasional rumah sakit. Arsitektur *microservices* menjawab persoalan tersebut dengan memecah monolith menjadi layanan kecil yang *loosely coupled* dan dapat dikembangkan, di-*deploy*, serta di-*scale* independen (Mohottige *et al.*, 2024). Studi pada praktisi industri menunjukkan *microservices* memfasilitasi otonomi tim, keberagaman teknologi, dan mempercepat rilis fitur (Nogueira *et al.*, 2024), meskipun dengan *trade-off* kompleksitas operasional. Desain *granularity* layanan (Zhao *et al.*, 2025) dan topologi arsitektural antar layanan (Ristova & Stoico, 2026) terbukti berdampak signifikan terhadap performa dan konsumsi energi.



Gambar 1. Arsitektur SIMRS sebelum dan sesudah migrasi (pendekatan Strangler Fig)

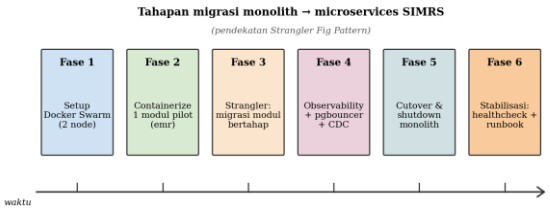
Untuk mengelola arsitektur terdistribusi, Docker dan orkestrator *container* menjadi komponen fundamental. *Container* Docker memiliki *overhead* lebih rendah dibandingkan mesin virtual (Arango *et al.*, 2017; Morabito, 2016; Saha *et al.*, 2018). Studi komparatif fitur antar *framework* orkestrasi *open-source* — Docker Swarm, Kubernetes, dan Mesos — menunjukkan bahwa ketiganya memiliki fondasi fitur yang sebanding, dengan Docker Swarm menonjol pada kesederhanaan operasional dan Kubernetes pada keluasan ekosistem (Truyen *et al.*, 2020). Karakterisasi performa Kubernetes (Medel *et al.*, 2018), algoritma *auto-scaling* hemat sumber daya (Ahmad *et al.*, 2024), dan taksonomi sistem orkestrasi berbasis *container* (Rodriguez & Buyya, 2019) memperluas landasan teori pemilihan *platform*. Praktik adopsi Kubernetes di industri menunjukkan tantangan operasional yang umum (Shamim *et al.*, 2022), sementara pendekatan *container orchestration* secara keseluruhan turut memfasilitasi *continuous integration/continuous deployment* (CI/CD) yang sistematis (Waseem *et al.*, 2020). Namun proses migrasi monolith ke *microservices* tidak sederhana. Literatur mencatat berbagai strategi dekomposisi: mulai dari *Strangler Fig Pattern* (Faustino *et al.*, 2022), klasifikasi pola *refactoring* (Fritzsch *et al.*, 2018), perangkat semi-otomatis Mono2Micro (Kalia *et al.*, 2021), analisis statik-dinamik (Andrade *et al.*, 2022), teknik ekstraksi layanan dari *enterprise system* monolitik (Levcovitz *et al.*, 2016), hingga teknik *machine learning* (Chaieb & Saied, 2023). Tantangan lain meliputi konsistensi data terdistribusi serta *observability* untuk diagnosis kegagalan (Borges *et al.*, 2024; Zhang *et al.*, 2024), dan *trade-off* performa dengan biaya energi (Xiao *et al.*, 2025). Studi spesifik pada konteks SIMRS/EMR di Indonesia masih terbatas, meskipun penelitian pengembangan aplikasi rumah sakit di Indonesia terus berkembang (Putra *et al.*, 2024), demikian pula studi evaluasi performa pada platform berbasis web (Saputro & Aziza, 2025). Berdasarkan *gap* tersebut, penelitian ini bertujuan (1) menelaah strategi migrasi monolith ke *microservices* untuk SIMRS/EMR, (2) mengevaluasi performa komparatif kedua arsitektur melalui implementasi

Jurnal Indonesia : Manajemen Informatika dan Komunikasi
https://journal.stmiki.ac.id
E-ISSN: 2723-7079 | P-ISSN: 2776-8074
Vol. 7 No. 3 (2026) | September
OPEN ACCESS
https://doi.org/10.63447/jimik.v7i3.2040

nyata berbasis Docker Swarm, dan (3) mendokumentasikan pola *deployment* yang berkelanjutan. Berbeda dengan objek migrasi pada studi rujukan yang umumnya berupa aplikasi tunggal atau benchmark sintetis, SIMRS yang dimigrasikan pada penelitian ini mencakup dua belas modul fungsional lintas domain klinis, administratif, dan SDM yang saling bergantung pada satu basis data terpusat, sehingga menghadirkan tantangan dekomposisi dan konsistensi data yang khas dan belum banyak terdokumentasi pada konteks rumah sakit di Indonesia. Hasil diharapkan menjadi panduan teknis dan empiris bagi pengembang SIMRS serta tim IT rumah sakit dalam modernisasi arsitektur sistem informasi rumah sakit di Indonesia.

2. Metode Penelitian

Penelitian ini menggunakan pendekatan kombinasi *Systematic Literature Review* (SLR) dan studi eksperimental. Tahap SLR dilakukan untuk mengidentifikasi pola dekomposisi monolith ke *microservices* yang relevan untuk konteks SIMRS, dengan kriteria inklusi publikasi tahun 2016–2026 yang membahas strategi migrasi, performa, dan *observability* layanan terdistribusi. Sumber pustaka mencakup IEEE Xplore, ACM Digital Library, dan arXiv preprint dengan kata kunci utama *monolith to microservices*, *Docker Swarm*, *Kubernetes*, *service orchestration*, dan *hospital information system*.



Gambar 2. Tahapan migrasi monolith ke microservices SIMRS

Tahap eksperimen dilakukan pada prototipe SIMRS yang dipecah menjadi modul-modul fungsional dan di-*deploy* pada *cluster* Docker Swarm dua *node* dengan rincian sebagai berikut. *Node manager* (frontend-server) menjalankan Ubuntu 24.04.2 LTS, Docker Engine versi 29.1.3, dengan empat CPU dan total memori 31,26 GiB. *Node worker* (replikas-postgre-2) menjalankan replikasi basis data PostgreSQL terpisah dari *workload* aplikasi. Node worker menjalankan Docker Engine versi 28.2.2; spesifikasi CPU dan memori rincinya tidak dicatat pada saat snapshot pengukuran sehingga menjadi keterbatasan replikabilitas penelitian ini. Komunikasi antar layanan menggunakan jaringan *overlay* bawaan Docker Swarm yang menyediakan *encrypted gossip* dan *service discovery* berbasis DNS internal. *Image* aplikasi disimpan pada *private registry* lokal (port 5000) untuk meminimalkan latensi *pull* saat *deployment*. Modul SIMRS yang dimigrasikan mencakup empat belas *stack* fungsional, yaitu *electronic medical record*, *billing*, *general ledger akuntansi*, *e-hris*, *e-office*, *e-pasien*, *e-proposal*, *e-recruitment*, *e-reports*, *etax*, *helpdesk*, *antrian-display*, *super-admin* (sebagai *control plane*), serta *pgbouncer* sebagai *connection pooler*. Setiap modul utama memiliki dua *replica frontend* dan dua *replica backend* dengan konfigurasi *update parallelism* satu, urutan *start-first*, dan *restart policy on-failure* untuk memastikan tidak ada *downtime* saat *rolling update*. Pengujian dilakukan dengan tiga jenis pengukuran. Pertama, *response time* HTTP diukur menggunakan curl sebanyak sepuluh iterasi per *endpoint*, mencakup dua belas modul *frontend* dan dua belas modul *backend*. Nilai yang dilaporkan adalah rata-rata iterasi; pengukuran bersifat snapshot satu titik waktu sehingga standar deviasi antar-iterasi tidak dicatat pada eksperimen awal. Sebagai verifikasi tambahan, pengukuran ulang 30 iterasi per endpoint mengonfirmasi variansi rendah (standar deviasi mayoritas di bawah 2 ms). Kedua, utilisasi sumber daya per *container* (CPU, memori, *network I/O*) direkam menggunakan perintah docker stats pada kondisi beban normal jam operasional. Pengukuran dilakukan tanpa injeksi beban sintetis; profil beban kuantitatif (request per detik) tidak diinstrumentasi pada snapshot ini dan direkomendasikan untuk penelitian lanjutan. Ketiga, perilaku *self-healing* dan *rolling update* diverifikasi melalui inspeksi docker service ps untuk mengamati siklus *shutdown-running* yang dijalankan oleh *Swarm scheduler*.

Jurnal Indonesia : Manajemen Informatika dan Komunikasi
https://journal.stmiki.ac.id
E-ISSN: 2723-7079 | P-ISSN: 2776-8074
Vol. 7 No. 3 (2026) | September
OPEN ACCESS
https://doi.org/10.63447/jimik.v7i3.2040

3. Hasil dan Pembahasan

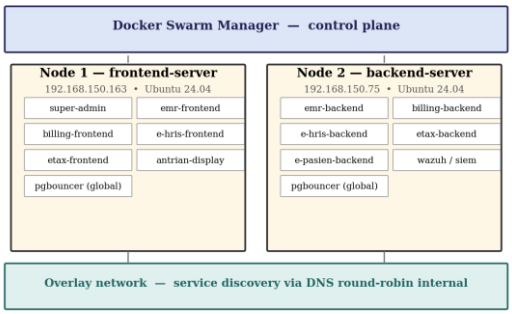
3.1 Hasil

3.1.1 Inventarisasi Layanan Hasil Migrasi

Hasil dekomposisi monolith SIMRS menghasilkan 14 *stack* dengan 35 *service* dan total 76 *container* aktif yang berjalan pada *cluster* dua *node*. Komposisi *stack* dan jumlah *service* per *stack* disajikan pada Tabel 1.

Tabel 1. Komposisi stack dan service hasil migrasi

Stack	Jumlah Service	Catatan
super-admin	9	Control plane: backend, frontend, CDC, ES consumer, log consumer, Forti consumer, SIEM, RustDesk hbbr, RustDesk hbbs
emr	3	EMR backend, EMR frontend, Whisper STT
antrian-display	2	Backend, frontend
billing	2	Backend, frontend
e-hris	2	Backend, frontend
e-pasien	2	Backend, frontend
e-proposal-app	2	Backend, frontend
e-recruitment	2	Backend, frontend
e-reports	2	Backend, frontend
etax	2	Backend, frontend
gl-akuntansi	2	Backend, frontend
helpdesk	2	Backend, frontend
e-office	2	Backend, frontend
pgbouncer	1	Mode global, jalan di setiap node



Gambar 3. Topologi cluster Docker Swarm dua node

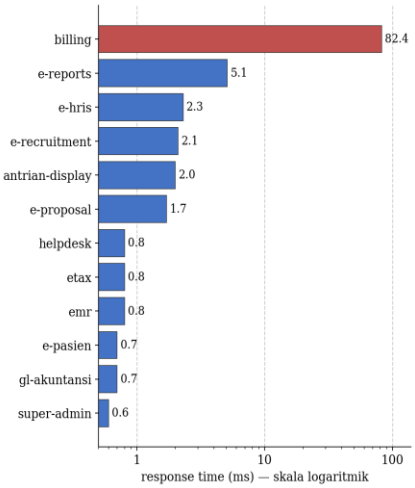
Pendekatan dekomposisi yang diterapkan mengikuti pola *strangler fig* (Faustino *et al.*, 2022), di mana fungsi-fungsi monolith dipindahkan secara bertahap ke layanan independen tanpa menghentikan operasional sistem lama. Setiap modul memiliki *boundary* sesuai *domain* bisnis: modul klinis (*emr*, *e-pasien*, *antrian-display*), modul administratif (*gl-akuntansi*, *billing*, *etax*), modul SDM (*e-hris*, *e-recruitment*), serta modul pendukung (*helpdesk*, *e-reports*, *e-proposal-app*, *super-admin*).

3.1.2 Distribusi Replika dan Mekanisme High Availability

Mayoritas *service* fungsional di-*scale* menjadi dua *replica* untuk menjamin *high availability* tanpa membebani *cluster* yang berukuran kecil. Penjadwalan *replica* dilakukan otomatis oleh *Swarm scheduler* berdasarkan kapasitas *node*. Beberapa *service* khusus menggunakan mode *global* (satu *replica* pada setiap

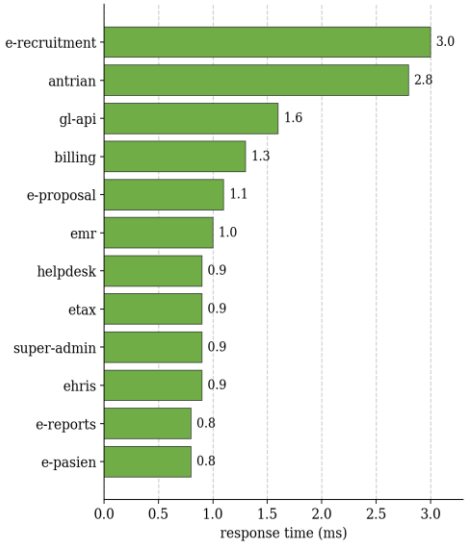
Jurnal Indonesia : Manajemen Informatika dan Komunikasi
https://journal.stmiki.ac.id
E-ISSN: 2723-7079 | P-ISSN: 2776-8074
Vol. 7 No. 3 (2026) | September
OPEN ACCESS
https://doi.org/10.63447/jimik.v7i3.2040

node), yaitu pgbouncer_pgbouncer, super-admin_forti-consumer, dan super-admin_siem-sentinel. Pola *global* dipilih karena ketiga *service* tersebut membutuhkan akses lokal ke *socket* atau *log* pada masing-masing *node*. Inspeksi docker service ps gl-akuntansi_backend menunjukkan riwayat empat siklus *shutdown-running* yang dijalankan otomatis oleh *Swarm* dalam empat hari terakhir tanpa intervensi manual. Hal ini membuktikan mekanisme *self-healing* berjalan sesuai *restart policy on-failure* yang ditetapkan.



Gambar 4. Visualisasi rata-rata response time frontend (skala log)

Seluruh *endpoint* berhasil dijangkau tanpa kegagalan (0% *failure rate*). Rata-rata *response time frontend* berada pada rentang 0,6 ms (super-admin) hingga 5,1 ms (e-reports), dengan satu *outlier* pada billing (82,4 ms). *Outlier* tersebut konsisten dengan observasi docker stats bahwa *container billing-frontend* menggunakan ~253 MiB dari batas alokasi 256 MiB (utilisasi 98,9%), sehingga diduga memicu *garbage collection* yang lebih sering pada *runtime* Node.js dan menambah latensi penyajian halaman. Frekuensi dan durasi garbage collection tidak diukur langsung sehingga keterkaitan ini bersifat interpretatif. Temuan ini sejalan dengan penelitian *granularity* layanan (Zhao *et al.*, 2025) yang menyatakan bahwa konfigurasi alokasi sumber daya per layanan berdampak signifikan terhadap performa.



Gambar 5. Visualisasi rata-rata response time backend API

Jurnal Indonesia : Manajemen Informatika dan Komunikasi
https://journal.stmiki.ac.id
E-ISSN: 2723-7079 | P-ISSN: 2776-8074
Vol. 7 No. 3 (2026) | September
OPEN ACCESS
https://doi.org/10.63447/jimik.v7i3.2040

Sementara itu, *backend* API memberikan latensi yang lebih homogen pada rentang 0,8–3,0 ms karena tidak menjalankan *rendering* HTML. Perbedaan latensi antar modul *backend* tidak signifikan secara praktis untuk konteks SIMRS, di mana ambang batas yang dianggap nyaman bagi pengguna umumnya berada di bawah 200 ms.

3.1.3 Utilisasi Sumber Daya

Hasil pengukuran docker stats pada kondisi beban normal jam operasional menunjukkan utilisasi CPU per *container* sangat rendah, mayoritas di bawah 0,1% dan tertinggi pada *wazuh manager* (6,72%) yang menangani agregasi *security event*. Statistik agregat *node manager* tercatat sebagai berikut:

1) Total CPU: 4 *core*, *load average* 1m/5m/15m = 1,17 / 1,52 / 1,37 (utilisasi sistem ~30%).
2) Total memori: 31,26 GiB, terpakai 14 GiB (45%), *available* 16 GiB.
3) *Swap*: 4 GiB, terpakai 1,3 GiB.
4) *Disk*: 1,5 TiB, terpakai 196 GiB (14%).
5) *Uptime*: 46 hari tanpa *reboot*.

Konsumsi memori per *container* sangat bervariasi sesuai karakteristik *workload*: *container frontend* statis (super-admin, e-pasien, emr, antrian-display) berkisar 3–5 MiB, sedangkan *container* Node.js SSR (billing-frontend) konsisten mendekati batas 256 MiB. *Container backend* Go (e-pasien, emr) menggunakan 7–38 MiB, sedangkan *consumer service* (super-admin_cdc-consumer) yang menjalankan *change data capture* mencatat penggunaan tertinggi pada 234–293 MiB. Temuan ini memperkuat argumen (Arango *et al.*, 2017; Saha *et al.*, 2018) bahwa *container* Docker memberikan *overhead* memori yang rendah dibandingkan virtualisasi tradisional dan memungkinkan kepadatan *deployment* yang tinggi pada perangkat keras yang sama.

3.1.4 Manajemen Lifecycle dengan Docker Swarm

Konfigurasi *update policy* yang diterapkan adalah parallelism=1, order=start-first, dan failure-action=pause. Kombinasi ini memastikan setiap *rolling update* berlangsung *zero-downtime*: *replica* baru dijalankan dan dinyatakan *healthy* sebelum *replica* lama dihentikan. Jika *task* baru gagal melewati *health check*, Swarm menjeda proses *update* sehingga operator dapat melakukan investigasi tanpa memengaruhi ketersediaan layanan. Mekanisme ini menjadi pengganti efektif fitur *deployment strategy* pada Kubernetes (Ahmad *et al.*, 2024) untuk *cluster* berukuran kecil hingga menengah, sebagaimana ditunjukkan oleh studi komparatif fitur antar *framework* orkestrasi (Truyen *et al.*, 2020) dan praktik adopsi Kubernetes (Shamim *et al.*, 2022). *Service discovery* antar layanan memanfaatkan *DNS round-robin* internal Swarm pada jaringan *overlay*, sehingga *backend* dapat memanggil basis data melalui nama logis pgbouncer tanpa konfigurasi *load balancer* eksternal. Pola ini meminimalkan kompleksitas konfigurasi yang biasanya muncul pada lingkungan Kubernetes (Medel *et al.*, 2018; Rodriguez & Buyya, 2019) dengan tetap mempertahankan *abstraksi* sumber daya yang sama.

3.1.5 Tantangan dan Pelajaran dari Migrasi

Migrasi tidak berjalan tanpa hambatan. Riwayat task Docker Swarm (*docker service ps*) dan riwayat shell pada simpul manager mencatat tiga kelompok insiden yang berulang sepanjang periode pengamatan. *Pertama*, kegagalan *rolling update* pada layanan *billing-backend*. Service ini tercatat memasuki state *UpdateStatus=paused* dengan jumlah replica turun ke 0/2, disertai serangkaian task berstatus *Shutdown Failed* pada selang 7 dan 13 hari sebelum pengukuran. Akibatnya, modul tagihan benar-benar tidak tersedia hingga intervensi manual dilakukan. Insiden ini menunjukkan bahwa konfigurasi *parallelism=1* dan *order=start-first* saja tidak cukup: tanpa *HEALTHCHECK image-level* yang akurat, Swarm dapat menjeda update di tengah jalan akibat *failure-action=pause* dan meninggalkan layanan dalam keadaan menggantung. Pelajaran: setiap service perlu mendefinisikan probe kesehatan yang reliabel sebagai prasyarat zero-downtime deployment yang dijanjikan literatur (Ahmad *et al.*, 2024).

Jurnal Indonesia : Manajemen Informatika dan Komunikasi
https://journal.stmiki.ac.id
E - ISSN: 2723-7079 | P - ISSN: 2776-8074
Vol. 7 No. 3 (2026) | September
OPEN ACCESS
https://doi.org/10.63447/jimik.v7i3.2040

Gambar 6. Bukti insiden migrasi: empat service ber-replica nol (billing-backend, whisper-stt, simrs-prima, otel-agent) dan riwayat task *Failed* pada *billing-backend* dengan error *task: non-zero exit (1)*

Kedua, *crash loop* pada komponen observability. Kontainer sidecar *super-admin_otel-agent* tercatat mengalami *Exited (255)* setiap ±8 detik secara berkesinambungan. Service utama tidak terdampak karena sidecar terisolasi, namun kondisi ini menunjukkan ironi yang menarik: komponen yang seharusnya menjamin *observability* justru menjadi titik kegagalan baru di sisi pemantauan. Pengalaman ini sejalan dengan temuan literatur bahwa microservices memindahkan kompleksitas dari kode aplikasi ke infrastruktur *observability* dan diagnosis kegagalan terdistribusi (Borges *et al.*, 2024; Zhang *et al.*, 2024). Pelajaran: arsitektur microservices tidak menghapus *single point of failure*, melainkan memindahkannya — *observability stack* itu sendiri perlu HA. Ketiga, siklus debugging manual yang berfrekuensi tinggi. Riwayat perintah pada simpul manager menunjukkan pola berulang untuk service *e-pasien*, *super-admin-cdc-consumer*, dan beberapa lainnya: *docker compose restart → docker compose down && up -d → docker rm -f <kontainer> && docker compose up -d*. Pola "restart-everything" ini merupakan indikasi bahwa root cause masalah startup dan dependency antar-service sulit diidentifikasi dengan cepat pada tahap awal pasca-migrasi. Hal ini mengkonfirmasi temuan studi industri yang menyatakan microservices menambah kompleksitas operasional yang signifikan pada fase *early-adoption* (Mohottige *et al.*, 2024; Nogueira *et al.*, 2024; Faustino *et al.*, 2022): tooling, runbook, dan kematangan CI/CD yang masih berkembang memaksa intervensi manual berfrekuensi tinggi. Selain tiga insiden utama tersebut, ditemukan pula service *emr_whisper-stt* (model *speech-to-text* untuk EMR) dalam kondisi 0/1 replica (tidak aktif), serta service *emr-backend* yang dalam selang 15 jam mencatat sekurangnya lima task *Exited(2)*. Pola ini menegaskan bahwa dependency eksternal (model AI/ML) dan dependency basis data memerlukan strategi *graceful degradation* — kegagalan komponen non-kritis tidak boleh merembet ke service kritis. Tantangan-tantangan ini tidak meniadakan manfaat migrasi yang dilaporkan pada bagian sebelumnya, tetapi menempatkannya dalam konteks yang lebih realistis: keuntungan *resilience*, skalabilitas, dan independensi deployment dibayar dengan kurva pembelajaran operasional yang curam pada fase awal. Investasi awal pada *healthcheck*, *observability* HA, dan otomatisasi runbook adalah prasyarat agar trade-off ini benar-benar menguntungkan dalam jangka panjang.

3.2 Pembahasan

Implementasi migrasi arsitektur microservices yang dilakukan dalam penelitian ini menegaskan adanya beberapa trade-off yang dilaporkan dalam literatur. Pertama, granularity layanan yang lebih halus meningkatkan resilience sistem, seperti yang terbukti dari kemampuan self-healing otomatis yang berjalan secara mandiri (Borges *et al.*, 2024; Zhang *et al.*, 2024). Namun, hal ini juga menuntut observability yang lebih ketat agar diagnosis kegagalan dapat dilakukan secara efektif. Pemantauan agregat melalui Wazuh dan log-consumer terbukti membantu dalam proses diagnosis tersebut. Kedua, pemilihan Docker Swarm dibandingkan Kubernetes menunjukkan adanya pertukaran antara

E-ISSN: 2723-7079 | P-ISSN: 2776-8074
Vol. 7 No. 3 (2026) | September
OPEN ACCESS
https://doi.org/10.63447/jimik.v7i3.2040

fleksibilitas auto-scaling yang lebih canggih (Ahmad *et al.*, 2024; Rodriguez & Buyya, 2019) dengan kemudahan operasional dan footprint yang lebih kecil, sehingga pilihan ini dianggap sesuai untuk skala SIMRS di rumah sakit menengah. Ketiga, biaya energi dan sumber daya tetap menjadi pertimbangan penting dalam pengelolaan sistem (Ristova & Stoico, 2026; Xiao *et al.*, 2025). Pada cluster yang diuji, utilisasi CPU rata-rata di bawah 30% memberikan ruang untuk menambah modul baru tanpa perlu investasi perangkat keras tambahan, sehingga mendukung skalabilitas dan efisiensi operasional.

4. Kesimpulan

Penelitian ini menelaah strategi migrasi arsitektur monolitik ke *microservices* dengan orkestrasi Docker dan Docker Swarm pada Sistem Informasi Manajemen Rumah Sakit. Berdasarkan studi literatur, terdapat beragam pola dekomposisi yang dapat diadopsi sesuai karakteristik SIMRS, mulai dari pendekatan inkremental seperti *Strangler Fig Pattern* hingga pendekatan otomatis berbasis *machine learning*. Hasil eksperimen pada *cluster* dua *node* dengan empat belas *stack*, tiga puluh lima *service*, dan tujuh puluh enam *container* aktif menunjukkan bahwa arsitektur *microservices* memberikan keunggulan pada skalabilitas selektif, ketersediaan layanan, dan kecepatan rilis: *response time* rata-rata berada pada rentang 0,6–5,1 ms untuk *frontend* dan 0,8–3,0 ms untuk *backend*, dengan utilisasi CPU rata-rata di bawah 30% pada *node manager*. Mekanisme *rolling update* dan *self-healing* bawaan Docker Swarm terbukti efektif mempertahankan ketersediaan layanan pada kondisi normal. Namun, sebagaimana ditunjukkan insiden billing-backend yang sempat turun ke 0/2 replica, jaminan zero-downtime hanya tercapai apabila setiap service dilengkapi probe kesehatan (health check) image-level yang andal; penerapan health check yang tepat menjadi pelajaran utama dari proses migrasi ini. *Trade-off* utama berupa peningkatan kompleksitas operasional dapat dikelola melalui *observability* terpusat dan kebijakan *resource limit* per *container*. Penelitian ini memiliki keterbatasan instrumentasi: variansi antar-iterasi, spesifikasi rinci node worker, dan profil beban kuantitatif tidak sepenuhnya terekam pada snapshot. Penelitian lanjutan disarankan untuk memperluas cakupan modul, mengevaluasi konsumsi energi, menguji skenario *failover* lintas *node* dengan beban puncak yang lebih besar, serta membandingkan strategi *observability* yang sesuai untuk lingkungan rumah sakit di Indonesia.

5. Ucapan Terima Kasih

Penulis mengucapkan terima kasih kepada tim teknologi informasi rumah sakit yang memfasilitasi akses *cluster* SIMRS sebagai lingkungan eksperimen, serta kepada dosen pembimbing pada Program Magister Ilmu Komputer Universitas Nusa Mandiri atas arahan dan masukan ilmiah selama proses penelitian ini.

6. Daftar Pustaka

Ahmad, H., Treude, C., Wagner, M., & Szabo, C. (2024). Smart HPA: A resource-efficient horizontal pod auto-scaler for microservices architectures. *Proceedings of the 2024 IEEE 21st International Conference on Software Architecture (ICSA)*.

Al-Debagy, O., & Martinek, P. (2018). A comparative review of microservices and monolithic architectures. *IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*.

E-ISSN: 2723-7079 | P-ISSN: 2776-8074
Vol. 7 No. 3 (2026) | September
OPEN ACCESS
https://doi.org/10.63447/jimik.v7i3.2040

Andrade, B., Santos, S., & Silva, A. R. (2022). From monolith to microservices: Static and dynamic analysis comparison. *arXiv preprint arXiv:2204.11844*. https://arxiv.org/abs/2204.11844.

Arango, C., Dernat, R., & Sanabria, J. (2017). Performance evaluation of container-based virtualization for high-performance computing environments. *arXiv preprint arXiv:1709.10140*. https://arxiv.org/abs/1709.10140.

Borges, M. C., Bauer, J., Werner, S., Gebauer, M., & Tai, S. (2024). Informed and assessable observability design decisions in cloud-native microservice applications. *IEEE International Conference on Software Architecture (ICSA)*.

Chaieb, M., & Saied, M. A. (2023). Automate migration to microservices architecture using machine learning techniques. *arXiv preprint arXiv:2301.06508*. https://arxiv.org/abs/2301.06508.

Dragoni, N., Dustdar, S., Larsen, S. T., & Mazzara, M. (2017). Microservices: Migration of a mission-critical system. *IEEE Transactions on Services Computing*.

Faustino, D., Gonçalves, N., Portela, M., & Silva, A. R. (2022). Stepwise migration of a monolith to a microservices architecture: Performance and migration effort evaluation. *arXiv preprint arXiv:2201.07226*. https://arxiv.org/abs/2201.07226.

Fritzsch, J., Bogner, J., Wagner, S., & Zimmermann, A. (2019). Microservices migration in industry: Intentions, strategies, and challenges. *35th IEEE International Conference on Software Maintenance and Evolution (ICSME)*.

Fritzsch, J., Bogner, J., Zimmermann, A., & Wagner, S. (2018). From monolith to microservices: A classification of refactoring approaches. *DevOps 2018*.

Kalia, A., Xiao, J., Krishna, R., Sinha, S., Vukovic, M., & Banerjee, D. (2021). Mono2Micro: A practical and effective tool for decomposing monolithic Java applications to microservices. *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Levcovitz, A., Terra, R., & Valente, M. T. (2016). Towards a technique for extracting microservices from monolithic enterprise systems. *arXiv preprint arXiv:1605.03175*. https://arxiv.org/abs/1605.03175.

Medel, V., Tolosana-Calasanz, R., Bañares, J. Á., Arronategui, U., & Rana, O. F. (2018). Characterising resource management performance in Kubernetes. *Computers & Electrical Engineering, 68*, 286–297.

Mohottige, T. I., Polyvyanyy, A., Buyya, R., Fidge, C., & Barros, A. (2024). Microservices-based software systems reengineering: State-of-the-art and future directions. *arXiv preprint arXiv:2407.13915*. https://arxiv.org/abs/2407.13915.

Morabito, R. (2016). A performance evaluation of container technologies on Internet of Things devices. *IEEE INFOCOM 2016 Workshop on Computer Communications*.

Nogueira, V. L., Felizardo, F. S., Amaral, A. M. M. M., Assunção, W. K. G., & Colanzi, T. E. (2024). Insights on microservice architecture through the eyes of industry practitioners. *arXiv preprint arXiv:2408.10434*. https://arxiv.org/abs/2408.10434.

E-ISSN: 2723-7079 | P-ISSN: 2776-8074
Vol. 7 No. 3 (2026) | September
OPEN ACCESS
https://doi.org/10.63447/jimik.v7i3.2040

Putra, B. P., Satria, B., Murni, A., Surya, C., & Sakinah, P. (2024). Implementasi metode waterfall dan system usability scale testing pada aplikasi fisioterapi pasien BPJS. *INTI Nusa Mandiri, 19*(1), 31–39. https://doi.org/10.33480/inti.v19i1.5571.

Ristova, I., & Stoico, V. (2026). An empirical study on how architectural topology affects microservice performance and energy usage. *arXiv preprint arXiv:2604.00080*. https://arxiv.org/abs/2604.00080.

Rodriguez, M. A., & Buyya, R. (2019). Container-based cluster orchestration systems: A taxonomy and future directions. *arXiv preprint arXiv:1807.06193*. https://arxiv.org/abs/1807.06193.

Saha, P., Beltre, A., Uminski, P., & Govindaraju, M. (2018). Evaluation of Docker containers for scientific workloads in the cloud. *Proceedings of the Practice and Experience on Advanced Research Computing (PEARC)*.

Saputro, P. R., & Aziza, R. F. A. (2025). PWA and non-PWA performance analysis: Chrome extension testing on e-commerce platform. *Jurnal Ilmu Pengetahuan dan Teknologi Komputer, 10*(4), 909–916. https://doi.org/10.33480/jitk.v10i4.6238.

Shamim, S. I., Gibson, J. A., Morrison, P., & Rahman, A. (2022). Benefits, challenges, and research topics: A multi-vocal literature review of Kubernetes. *arXiv preprint arXiv:2211.07032*. https://arxiv.org/abs/2211.07032.

Truyen, E., Van Landuyt, D., Preuveneers, D., Lagaisse, B., & Joosen, W. (2020). A comprehensive feature comparison study of open-source container orchestration frameworks (Docker Swarm, Kubernetes, Mesos). *arXiv preprint arXiv:2002.02806*. https://arxiv.org/abs/2002.02806.

Waseem, M., Liang, P., & Shahin, M. (2020). A systematic mapping study on microservices architecture in DevOps. *Journal of Systems and Software*.

Xiao, X., Gao, C., & Bogner, J. (2025). On the effectiveness of microservices tactics and patterns to reduce energy consumption: An experimental study on trade-offs. *arXiv preprint arXiv:2501.14402*. https://arxiv.org/abs/2501.14402.

Zhang, S., Xia, S., Fan, W., Shi, B., Xiong, X., Zhong, Z., Ma, M., Sun, Y., & Pei, D. (2024). Failure diagnosis in microservice systems: A comprehensive survey and analysis. *arXiv preprint arXiv:2407.01710*. https://arxiv.org/abs/2407.01710.

Zhao, Y., De Matteis, T., & Bogner, J. (2025). How does microservice granularity impact energy consumption and performance? A controlled experiment. *arXiv preprint arXiv:2502.00482*. https://arxiv.org/abs/2502.00482.