Journal of Information Technology and Computer Science (INTECOMS)
Volume 8 Nomor 3, Tahun 2025
e-ISSN : 2614-1574
p-ISSN : 2621-3249

OPTIMIZING DATA CONSISTENCY IN MICROSERVICE ARCHITECTURE USING THE SAGA PATTERN AND EVENT-DRIVEN APPROACH

OPTIMALISASI KONSISTENSI DATA PADA ARSITEKTUR MICROSERVICE DENGAN SAGA PATTERN DAN PENDEKATAN EVENT-DRIVEN

Arif Nurdiansyah[1], Esa Fauzi[2]
Universitas Widyatama[1,2]
nurdiansyah.arif@widyatama.ac.id

ABSTRACT

An event-driven microservice architecture offers flexibility and scalability but introduces significant challenges in maintaining data consistency across distributed services. Traditional ACID transactions are not viable in such environments, leading to potential data integrity issues during partial failures. This research proposes and evaluates the use of the Saga orchestration pattern as a solution to this problem. A comparative study was conducted by developing two systems in Golang: a baseline system using synchronous inter-service communication and a second system implementing an event-driven Saga pattern with Redis Streams as the event bus. The systems, utilizing MySQL, MongoDB, Redis, and Elasticsearch, were subjected to various failure scenarios. The results demonstrate that the synchronous system consistently produced data inconsistencies during partial failures, while the Saga-based system successfully maintained data integrity by executing compensating transactions, thus restoring the system to a consistent state. The study concludes that the Saga pattern is an effective strategy for optimizing data consistency and reliability in complex microservice architectures.

***Keywords**:Microservices, Saga Pattern, Event-Driven Architecture, Data Consistency, Saga Orchestration*

ABSTRAK

Arsitektur microservice yang digerakkan oleh peristiwa (event-driven) menawarkan fleksibilitas dan skalabilitas, namun memperkenalkan tantangan signifikan dalam menjaga konsistensi data di seluruh layanan terdistribusi. Transaksi ACID tradisional tidak dapat diterapkan dalam lingkungan semacam itu, yang menyebabkan potensi masalah integritas data selama kegagalan parsial. Penelitian ini mengusulkan dan mengevaluasi penggunaan pola Saga dengan strategi orkestrasi sebagai solusi untuk masalah ini. Sebuah studi komparatif dilakukan dengan mengembangkan dua sistem menggunakan Golang: sistem dasar yang menggunakan komunikasi sinkron antar-layanan dan sistem kedua yang mengimplementasikan pola Saga yang digerakkan oleh peristiwa dengan Redis Streams sebagai event bus. Sistem yang menggunakan MySQL, MongoDB, Redis, dan Elasticsearch ini diuji dalam berbagai skenario kegagalan. Hasilnya menunjukkan bahwa sistem sinkron secara konsisten menghasilkan inkonsistensi data selama kegagalan parsial, sementara sistem berbasis Saga berhasil menjaga integritas data dengan mengeksekusi transaksi kompensasi, sehingga mengembalikan sistem ke keadaan yang konsisten. Studi ini menyimpulkan bahwa pola Saga adalah strategi yang efektif untuk mengoptimalkan konsistensi dan keandalan data dalam arsitektur microservice yang kompleks.

Kata Kunci: Microservices, Saga Pattern, Event-Driven Architecture, Konsistensi Data, Orkestrasi Saga

PENDAHULUAN

Pengembangan perangkat lunak modern semakin beralih ke arsitektur *microservice* karena skalabilitas, fleksibilitas, dan kemudahan dalam penerapan independen yang ditawarkannya (Balalaie et al., 2016). Pendekatan ini diadopsi oleh banyak perusahaan teknologi terkemuka, terutama dalam domain e-commerce yang menuntut skalabilitas dan keandalan tinggi (Hasselbring & Steinacker, 2017), seperti Amazon, Netflix, dan Spotify (Di Francesco et al., 2019). Namun, arsitektur ini juga memperkenalkan tantangan signifikan dalam menjaga konsistensi data di seluruh layanan yang terdistribusi (Fauzi & Iriani, 2020).

Untuk mengatasi masalah ini, pola-pola baru diperlukan karena transaksi atomik (ACID) tradisional yang terdistribusi, seperti *Two-Phase Commit* (2PC), tidak praktis di lingkungan *microservice*. Protokol 2PC bersifat *blocking*—ia mengunci semua sumber daya selama proses *commit*, yang secara langsung melanggar prinsip *loose coupling* dan dapat menurunkan ketersediaan sistem secara drastis (Gray, 1981). Sebagai solusi, Saga *pattern* diperkenalkan sebagai model untuk mengelola konsistensi dalam transaksi jangka panjang (*Long Lived Transactions*) tanpa menggunakan mekanisme *locking* (Garcia-Molina & Salem, 1987). Sebuah saga adalah urutan dari beberapa transaksi lokal, di mana setiap transaksi memiliki transaksi kompensasi yang berfungsi untuk membatalkan dampaknya jika terjadi kegagalan di langkah selanjutnya.

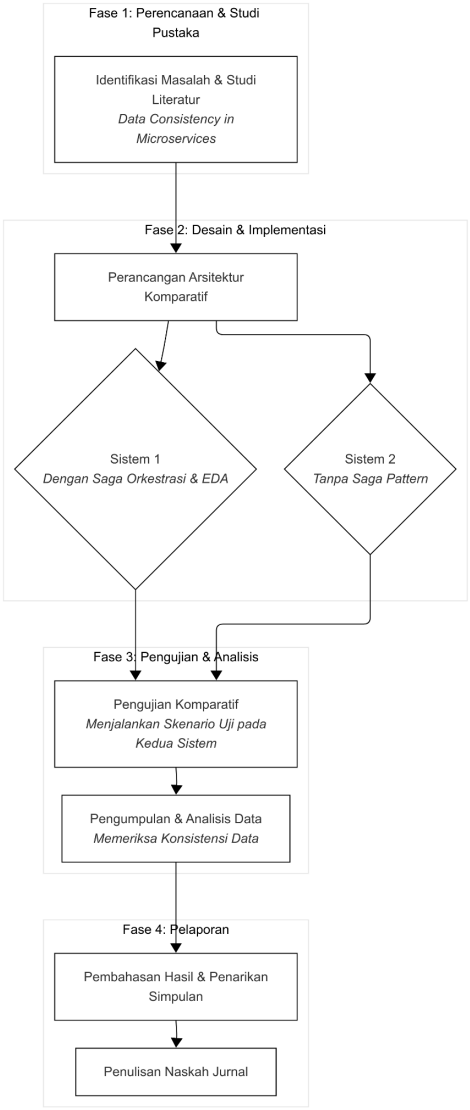
Terdapat dua strategi utama untuk mengoordinasikan saga: koreografi dan orkestrasi (Cerny et al., 2017). Dalam koreografi, setiap layanan mempublikasikan *event* dan layanan lain akan bereaksi terhadap *event* tersebut. Pendekatan ini sangat *decoupled* tetapi alur kerjanya sulit dilacak. Sebaliknya, dalam orkestrasi, sebuah komponen orkestrator terpusat secara eksplisit mengelola alur, mengirimkan perintah ke setiap layanan, dan menangani kegagalan serta kompensasi. Penelitian ini memilih pendekatan orkestrasi karena memberikan visibilitas dan kontrol yang lebih baik terhadap proses bisnis yang kompleks.

Rumusan masalah dalam penelitian ini berfokus pada *gap* teknis yang ingin diselesaikan, yaitu ketiadaan sebuah mekanisme yang terbukti andal untuk menjamin hasil akhir "semua berhasil atau semua dibatalkan" pada sistem yang bersifat *loosely coupled* dan menggunakan *polyglot persistence*.

Tujuan penelitian ini adalah untuk merancang, mengimplementasikan, dan mengevaluasi secara komparatif efektivitas dari Saga *pattern* dengan strategi orkestrasi. Untuk mencapai tujuan ini, dua sistem yang berbeda dirancang dan diimplementasikan menggunakan Golang, yang dikenal karena performa dan dukungan konkurensinya (Alchuluq & Nurzaman, 2021). Sistem usulan menggunakan Redis Streams sebagai *event bus* yang andal karena fitur persistensi dan *consumer group*-nya (Nugraha, 2022). Kedua sistem ini kemudian diuji secara komparatif untuk mengevaluasi kemampuannya dalam menjaga konsistensi data.

Kontribusi ilmiah dari penelitian ini bukanlah pada penemuan Saga *pattern* itu sendiri, melainkan pada aspek penerapan dan evaluasi empirisnya. Kebaruan dari pendekatan ini adalah penyediaan cetak biru (*blueprint*) implementasi yang konkret dari Saga orkestrasi menggunakan tumpukan teknologi modern, dan analisis komparatif berbasis bukti yang secara langsung membandingkan keadaan data akhir antara sistem berbasis Saga dan sistem sinkron dalam skenario kegagalan yang identik.

METODE



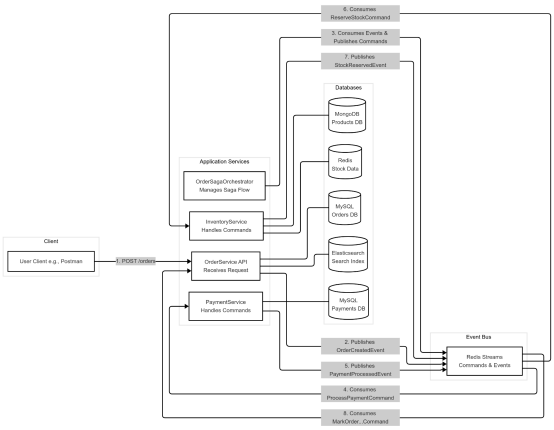
Gambar 1. Diagram Alir Metode Penelitian

Penelitian ini menggunakan pendekatan metode campuran (*mixed methods*) yang menggabungkan analisis kualitatif terhadap perilaku sistem dengan pengukuran kuantitatif terhadap performa. Pendekatan berbasis studi kasus untuk membandingkan dua implementasi sistem secara mendalam. Pendekatan ini memungkinkan analisis terperinci terhadap perilaku setiap arsitektur dalam skenario kegagalan yang terkontrol. Model pengembangan yang digunakan untuk membangun kedua sistem mengikuti alur SDLC (Software Development Life Cycle) Waterfall, yang mencakup tahapan analisis, desain, implementasi, dan pengujian (Alchuluq & Nurzaman, 2021). Alur penelitian ini secara keseluruhan dijelaskan pada gambar di atas.

Desain Sistem

Dua sistem yang merepresentasikan arsitektur berbeda dikembangkan untuk memproses alur pemesanan e-commerce. Keduanya dibangun menggunakan bahasa pemrograman Golang dengan framework Echo. Untuk menunjukkan fleksibilitas dalam pengelolaan data, kedua sistem mengadopsi model *polyglot persistence* yang memanfaatkan MySQL untuk data transaksional (pesanan, pembayaran), MongoDB untuk katalog produk, Redis untuk manajemen stok *real-time*, dan Elasticsearch untuk pengindeksan data.

Arsitektur untuk Sistem 1, yang menjadi fokus utama penelitian ini, diilustrasikan pada Gambar 2.



Gambar 2. Arsitektur Sistem dengan Saga Orkestrasi

1. **Sistem 1: Saga/EDA Orchestration System:** Seperti terlihat pada Gambar 2, sistem ini mengimplementasikan Saga pattern dengan strategi orkestrasi. Strategi ini dipilih karena kemampuannya mengelola proses bisnis yang kompleks secara terpusat (Cerny et al., 2017). Layanan `OrderSagaOrchestrator` berfungsi sebagai koordinator yang mengirimkan perintah (*commands*) dan menerima peristiwa balasan (*reply events*) dari layanan

partisipan (`OrderService`, `PaymentService`, `InventoryService`) melalui Redis Streams. Jika terjadi kegagalan, orchestrator bertanggung jawab memicu perintah kompensasi untuk memulihkan konsistensi.

2. **Sistem 2: Synchronous "No Saga" System:** Sistem ini berfungsi sebagai *baseline* perbandingan. Layanan berkomunikasi melalui pemanggilan HTTP sinkron secara berantai. Tidak ada mekanisme kompensasi otomatis jika terjadi kegagalan di tengah alur.

Logika Orkestrasi Saga

Inti dari sistem satu adalah `OrderSagaOrchestrator` yang berfungsi sebagai mesin status (*state machine*) terpusat. Orkestrator ini mendengarkan *event* yang dipublikasikan oleh layanan partisipan dan, berdasarkan status saga saat ini, memutuskan tindakan selanjutnya, baik itu melanjutkan ke langkah berikutnya atau memulai proses kompensasi. Logika proses utama dari orkestrator dapat diringkas dalam **Algoritma 1**.

ALGORITMA 1: Logika Proses Saga Orkestrator

```
1: PROCEDURE handleEvent(event)
2:   // Muat state saga berdasarkan orderId dari payload event
3:   saga <- loadSagaState(event.orderId)
4:
5:   SWITCH event.type:
6:     CASE "OrderCreatedEvent":
7:       saga <- createNewSaga(event.payload)
8:       updateSagaState(saga, "AWAITING_PAYMENT")
9:       publishCommand("payment_commands", "ProcessPaymentCommand", saga.payload)
10:
11:    CASE "PaymentProcessedEvent":
12:      IF saga.state == "AWAITING_PAYMENT" THEN
13:        updateSagaState(saga, "AWAITING_STOCK")
14:        publishCommand("inventory_commands", "ReserveStockCommand", saga.payload)
15:      END IF
16:
17:    CASE "StockReservedEvent":
18:      IF saga.state == "AWAITING_STOCK" THEN
19:        updateSagaState(saga, "COMPLETED")
20:        publishCommand("order_mgmt_commands", "MarkOrderAsCompletedCommand",
saga.payload)END IF
22:
23:    CASE "StockReservationFailedEvent": // Memulai alur kompensasi
24:      IF saga.state == "AWAITING_STOCK" THEN
25:        updateSagaState(saga, "COMPENSATING_PAYMENT")
26:        publishCommand("payment_commands", "RefundPaymentCommand", saga.payload)
27:      END IF
28:
29:    CASE "PaymentRefundedEvent": // Konfirmasi kompensasi berhasil
30:      IF saga.state == "COMPENSATING_PAYMENT" THEN
31:        updateSagaState(saga, "COMPENSATED")
32:        publishCommand("order_mgmt_commands", "MarkOrderAsFailedCommand", saga.payload)
33:      END IF
34:    // ...CASE LAIN UNTUK KOMPENSASI KEGAGALAN LAIN NYA
35:
36: END PROCEDURE
```

Algoritma 1: Logika Proses Saga Orkestrator

Seperti yang diilustrasikan pada **Algoritma 1**, orkestrator bertindak berdasarkan jenis *event* yang diterima dan status saga saat ini. Misalnya, saat menerima PaymentProcessedEvent (baris 11), orkestrator hanya akan melanjutkan proses ke tahap reservasi stok jika status saga sedang AWAITING_PAYMENT (baris 12). Hal ini mencegah pemrosesan duplikat atau di luar urutan. Jika terjadi kegagalan seperti StockReservationFailedEvent (baris 23), orkestrator tidak menghentikan proses, melainkan mengubah alur untuk memulai kompensasi dengan mengirimkan RefundPaymentCommand (baris 26). Pendekatan terstruktur ini adalah inti dari bagaimana saga menjaga konsistensi data.

Strategi Pengujian

Kedua sistem diuji menggunakan serangkaian skenario yang identik. Skenario-skenario ini dirancang untuk mensimulasikan kondisi normal ("Happy Path") dan berbagai kondisi kegagalan parsial. Salah satu skenario kegagalan utama adalah mematikan salah satu layanan (misalnya, InventoryService) untuk sementara waktu setelah langkah sebelumnya (pembayaran) berhasil. Hal ini dilakukan untuk menguji resiliensi dan kemampuan pemulihan (*recovery*) sistem. Setelah setiap pengujian, keadaan data di semua database (MySQL, Redis, MongoDB) diperiksa secara manual untuk dievaluasi tingkat konsistensinya.

Logika Kompensasi dalam Orkestrasi Saga

Secara teknis, proses kompensasi diatur oleh OrderSagaOrchestrator. Ketika menerima *event* kegagalan (misalnya, StockReservationFailedEvent), orkestrator akan memeriksa status saga dan mengirimkan perintah kompensasi (misalnya, RefundPaymentCommand) ke layanan yang relevan untuk membatalkan transaksi sebelumnya dan menjaga integritas data.

Desain Eksperimen dan Pengujian

Eksperimen dirancang secara terstruktur dengan langkah-langkah:

persiapan, eksekusi, observasi, dan evaluasi. Pengujian dipicu menggunakan *tools* seperti Postman dan Apache JMeter, sementara observasi dilakukan melalui analisis log. Skenario uji mencakup *happy path* dan berbagai *failure case* (misalnya, layanan inventaris mati pasca-pembayaran). Kriteria evaluasi utama adalah konsistensi data, waktu respons, dan waktu pemulihan.

HASIL DAN PEMBAHASAN

Bagian ini memaparkan desain arsitektur, detail implementasi, hasil pengujian komprehensif, serta pembahasan dari temuan yang didapatkan.

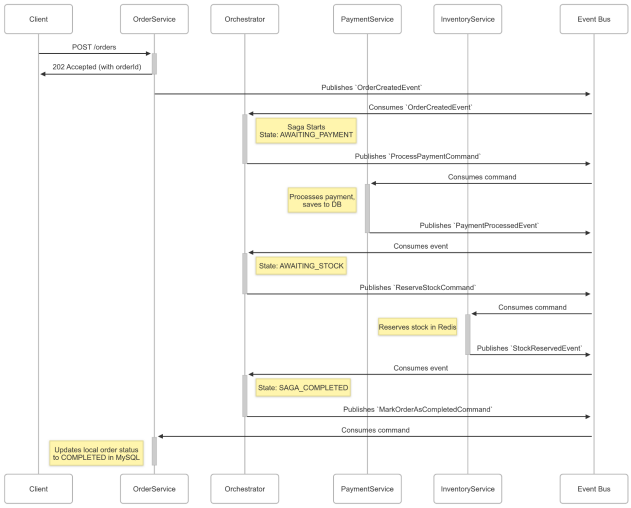
Proses Pengujian

Sebagai contoh, berikut adalah proses langkah demi langkah untuk menjalankan Skenario Kegagalan Inventaris pada Sistem 1.

1. **Persiapan:** Seluruh layanan microservice (OrderSagaOrchestrator, OrderService, PaymentService, InventoryService) dan infrastruktur database dijalankan. Kondisi kegagalan disiapkan dengan menambahkan produk baru melalui endpoint POST /products pada InventoryService dengan payload yang mengatur initial_stock menjadi 5 unit untuk item itemSaga002. Langkah ini sekaligus menyimpan detail produk di MongoDB dan stok awal di Redis.
2. **Eksekusi:** Sebuah permintaan API dikirim ke OrderService untuk memesan 10 unit itemSaga002, jumlah yang melebihi stok tersedia. OrderService memberikan respons HTTP 202 Accepted dan mencatat orderId serta sagaId.
3. **Observasi Alur (Forward):** Alur saga diamati melalui log. OrderSagaOrchestrator menerima OrderCreatedEvent, mengirim ProcessPaymentCommand, dan menerima PaymentProcessedEvent dari PaymentService yang berhasil. Kemudian, orkestrator mengirim ReserveStockCommand.
4. **Observasi Kegagalan & Kompensasi:** Log InventoryService menunjukkan bahwa reservasi gagal karena stok tidak mencukupi, lalu mempublikasikan StockReservationFailedEvent. Orkestrator menerima kegagalan ini dan memulai alur kompensasi dengan mengirim RefundPaymentCommand ke PaymentService. Setelah PaymentService mengonfirmasi refund dengan PaymentRefundedEvent, orkestrator mengirim MarkOrderAsFailedCommand ke OrderService untuk menyelesaikan saga.
5. **Verifikasi Akhir:** Pemeriksaan database pasca-pengujian dilakukan untuk memvalidasi keadaan akhir sistem yang konsisten.

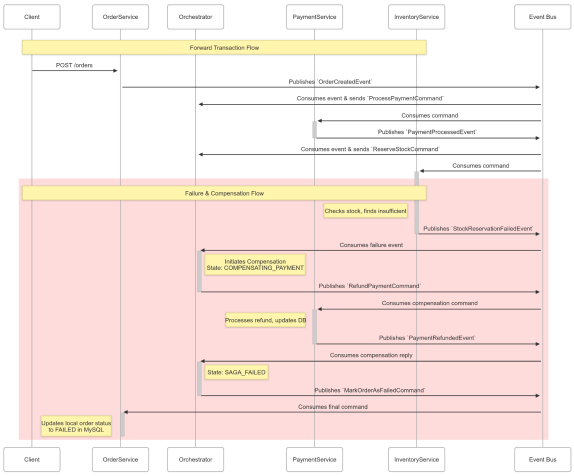
Hasil pengujian komparatif secara jelas menunjukkan perbedaan fundamental dalam penanganan konsistensi data antara kedua sistem.

Pada skenario **Happy Path**, di mana semua layanan berfungsi dengan baik, kedua sistem berhasil memproses pesanan dan menghasilkan keadaan data akhir yang konsisten. Alur pesan pada Sistem 1 untuk skenario ini diilustrasikan pada **Gambar 3**, menunjukkan urutan perintah dan peristiwa yang terkoordinasi oleh Orchestrator.



Gambar 3. Diagram Urutan Saga yang Berhasil (Happy Path)

Perbedaan signifikan baru terlihat pada skenario kegagalan. Mari kita analisis skenario di mana pembayaran berhasil, tetapi reservasi stok gagal. **Gambar 4** mengilustrasikan alur kegagalan dan kompensasi pada Sistem 1.



Gambar 4. Diagram Urutan Kompensasi Saga akibat Kegagalan Stok

Seperti yang terlihat pada **Gambar 4**, ketika StockReservationFailedEvent diterima, Orchestrator tidak berhenti, melainkan memulai alur pemulihan. Ia mengirimkan RefundPaymentCommand untuk membatalkan pembayaran yang sudah berhasil. Hasil pengujian ini dirangkum dalam Tabel 1.

Tabel 1. Perbandingan Hasil Sistem pada Skenario Kegagalan Stok Tidak Cukup

Elemen Pengamatan	Sistem 2 (No SAGA)	Sistem 1 (SAGA/EDA)
Status Order Akhir	FAILED_INVENTORY	FAILED
Status Pembayaran	**SUCCESS**	**REFUNDED**
Stok Inventaris	Tidak Berubah	Tidak Berubah
Konsistensi Data	Inkonsisten	Konsisten (Setelah Kompensasi)

Temuan ini secara jelas menunjukkan kelemahan Sistem 2. Sistem tersebut menciptakan inkonsistensi data di mana pelanggan ditagih biayanya meskipun pesanan tidak dapat dipenuhi. Sebaliknya, Sistem 1, sesuai dengan konsep asli Saga (Garcia-Molina & Salem, 1987), berhasil menjaga integritas data. Kemampuan untuk mengeksekusi transaksi kompensasi adalah kunci untuk menjaga konsistensi dalam sistem terdistribusi, yang pada akhirnya mengarah pada keadaan yang konsisten atau *eventually consistent* (Dragoni et al., 2017; Vogels, 2009). Kompleksitas tambahan dalam merancang Sistem 1 terbukti memberikan keuntungan besar dalam hal keandalan sistem.

Analisis Kuantitatif: Performa dan Latensi

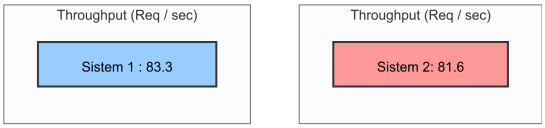
Untuk melengkapi analisis kualitatif, uji beban dilakukan menggunakan Apache JMeter. Hasil pengujian performa dan penanganan kegagalan dari kedua sistem disajikan dalam **Tabel 2**.

Tabel 2. Hasil Uji Performa dan Kegagalan dengan JMeter

Skenario Pengujian	Sistem	Throughput (req/sec)	Avg Response Time (ms)	Error Rate (%)

Happy Path	Sistem 2 (No SAGA)	81.6	234	0%
Happy Path	Sistem 1 (SAGA / EDA)	83.3	209	0%
Failure Path	Sistem 2 (No SAGA)	82.6	221	100%
Failure Path	Sistem 1 (SAGA / EDA)	83.5	209	0%

Data kuantitatif ini memberikan dua wawasan penting. Pertama, pada skenario *Happy Path*, Sistem 1 (Saga**)** menunjukkan performa yang sedikit lebih unggul dengan throughput lebih tinggi dan waktu respons rata-rata yang lebih rendah. Hal ini menunjukkan efisiensi dari arsitektur *non-blocking* di mana OrderService dapat merespons klien dengan cepat. Perbandingan *throughput* diilustrasikan pada **Gambar 5**.



Gambar 5. Perbandingan Throughput Sistem pada Skenario Happy Path

Wawasan kedua, dan yang paling krusial, datang dari skenario *Failure Path*. **Sistem 2** menghasilkan **Error Rate 100%**, karena setiap kegagalan di layanan hilir (inventaris) langsung dikembalikan sebagai respons error HTTP ke klien. Sebaliknya, **Sistem 1** mempertahankan **Error Rate 0%**. Dari sudut pandang klien, setiap permintaan diterima dengan sukses (HTTP 202 Accepted). Kegagalan dan proses kompensasi yang kompleks terjadi sepenuhnya di latar belakang, dikoordinasikan oleh saga tanpa membebani klien. Ini secara kuantitatif membuktikan bahwa arsitektur Saga memberikan pengalaman pengguna yang jauh lebih andal dan resilien.

Pembahasan Trade-off dan Keterkaitan dengan Literatur

Meskipun unggul dalam konsistensi dan resiliensi, penerapan Saga Pattern pada Sistem 1 memperkenalkan beberapa *trade-off*. Sistem 1 secara inheren lebih kompleks karena memerlukan komponen tambahan seperti *event bus* dan orkestrator, yang membuat proses *debugging* dan pelacakan menjadi lebih menantang (Di Francesco et al., 2019). Temuan ini sejalan dengan studi kasus industri yang ada, di mana keandalan yang ditawarkan oleh pola Saga dianggap sebagai pertukaran yang sepadan dengan peningkatan kompleksitas, terutama pada sistem skala besar seperti e-commerce (Hasselbring & Steinacker, 2017).

SIMPULAN

Penelitian ini secara empiris membuktikan bahwa penerapan Saga Pattern dengan strategi orkestrasi dalam arsitektur *microservice* secara efektif mengoptimalkan konsistensi data saat terjadi kegagalan parsial. Melalui studi kasus komparatif, ditunjukkan bahwa pendekatan sinkron gagal menjaga integritas data, sementara sistem berbasis Saga berhasil memulihkan keadaan yang konsisten melalui eksekusi transaksi kompensasi secara otomatis.

Selain keunggulan dalam konsistensi, analisis kuantitatif juga menunjukkan bahwa arsitektur *event-driven* pada sistem Saga memberikan keuntungan performa yang signifikan, termasuk *throughput* yang lebih tinggi dan waktu respons awal yang jauh lebih cepat bagi klien dibandingkan dengan pendekatan sinkron yang bersifat *blocking*. Meskipun demikian, keunggulan ini diimbangi dengan *trade-off* berupa peningkatan kompleksitas arsitektur dan tantangan dalam hal *observability* sistem yang terdistribusi.

Keterbatasan Penelitian

Penelitian ini memiliki beberapa keterbatasan. Pertama, pengujian dilakukan dalam lingkungan laboratorium yang terkontrol dan tidak merepresentasikan

beban kerja pada skala produksi yang sebenarnya. Kedua, implementasi Saga Orchestrator yang dibangun belum mencakup mekanisme *timeout* dan kebijakan *retry* otomatis untuk menangani layanan yang tidak merespons.

Saran untuk Penelitian Selanjutnya
Berdasarkan temuan dan keterbatasan tersebut, penelitian selanjutnya dapat difokuskan pada beberapa area. Pertama, melakukan implementasi dan evaluasi mekanisme *timeout* pada orkestrator untuk meningkatkan resiliensi sistem. Kedua, melakukan uji performa pada skala yang lebih besar untuk menganalisis *overhead* dari *event bus* dan orkestrator pada kondisi beban tinggi. Terakhir, melakukan studi komparatif antara strategi Orkestrasi dengan Koreografi menggunakan tumpukan teknologi yang sama untuk memberikan perbandingan yang adil mengenai kompleksitas implementasi keduanya.

DAFTAR PUSTAKA

Alchuluq, L. M., & Nurzaman, F. (2021). Analisis pada arsitektur microservice untuk layanan bisnis toko online. *TEKINFO*, *22*(2), 61-68.

Andaru, A. (2018). Pengertian database secara umum. *OSF Preprints*. https://doi.org/10.31219/osf.io/zpu8e

Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016). Microservices architecture enables DevOps: Migration to a cloud-native architecture. *IEEE Software*, *33*(3), 42–52.

Baresi, L., Garriga, M., & De Renzis, A. (2017). Microservices identification through interface analysis. In F. De Paoli, E. Pimentel, & E. B. Johnsen (Eds.), *Lecture Notes in Computer Science: Vol. 10465. Service-Oriented and Cloud Computing* (pp. 19–33). Springer.

Cerny, T., Donahoo, M. J., & Trnka, M. (2017). Contextual understanding of microservice architecture: Current and future directions. *Applied Computing Review*, *17*(4), 29–45.

Di Francesco, P., Lago, P., & Malavolta, I. (2019). Architecting with microservices: A systematic mapping study. *The Journal of Systems and Software*, *150*, 77–97.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering* (pp. 195-216). Springer, Cham.

Fauzi, E., & Iriani, Y. (2020). Architecture of monitor api to handle integration data problem in database-microservice. *PalArch's Journal of Archaeology of Egypt/Egyptology*, *17*(10), 1422-1428.

Febrian, T. (2021). *Analisis performansi web services pada arsitektur microservices pada domain kasus learning management system di Seskoau*. [Disertasi doktoral, Universitas Komputer Indonesia].

Gadge, S., & Kotwani, V. (2018). *Microservice architecture: API gateway considerations* [White paper]. GlobalLogic.

Garcia-Molina, H., & Salem, K. (1987). Sagas. *ACM SIGMOD Record*, *16*(3), 249–259.

Gray, J. (1981). The transaction concept: Virtues and limitations. In *Proceedings of the Seventh International Conference on Very Large Data Bases* (pp. 144-154).

Hasselbring, W., & Steinacker, G. (2017). Microservice architectures for scalability, agility and reliability in E-commerce. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)* (pp. 243-246). IEEE.

Nugraha, F. (2022). *Implementasi event-driven architecture untuk meningkatkan kinerja pada microservice*. [Disertasi doktoral, Universitas Siliwangi].

Soldani, J., & Tamburri, D. A. (2021). The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, *174*, 110893. https://doi.org/10.1016/j.jss.2020.110893

Tijms, S., van der Aalst, W. M., & van den Heuvel, W. J. (2020). A pattern-based comparison of decentralized approaches for business process execution. *Information and Software Technology*, *125*, 106318. https://doi.org/10.1016/j.infsof.2020.106318

Vogels, W. (2009). Eventually consistent. *Communications of the ACM*, *52*(1), 40–44.

Voloshina, I. (2020). *Applying microservices pattern to monolithic software* [Bachelor's thesis, Metropolia University of Applied Sciences]. Theseus. http://www.theseus.fi/handle/10024/348638