Progresif: Jurnal Ilmiah Komputer
https://ojs.stmik-banjarbaru.ac.id/index.php/progresif/index
Jl. Ahmad Yani, K.M. 33,5 - Kampus STMIK Banjarbaru Loktabat – Banjarbaru,
e-mail: puslit.stmikbjb@gmail.com
e-ISSN: 2685-0877

Perancangan *Failure-Aware Self-Healing Workflow* pada Sistem *Event-Driven* untuk Pemulihan Proses Layanan Digital Secara Otomatis

DOI: http://dx.doi.org/10.35889/progresif.v22i3.3977

Dimaz Ardawan[1*], Denny Kurniadi[2], Khairi Budayawan[3], Randi Proska Sandra[4]
Informatika, Universitas Negeri Padang, Padang, Indonesia
*e-mail *Corresponding Author:* dimazardawan@student.unp.ac.id

Abstract

Digital service systems built on event-driven architecture relied on infrastructure-level self-healing, leaving workflow-layer failures handled reactively and processes abandoned mid-execution even when infrastructure remained available. This study designed a failure-aware self-healing workflow that detected, classified, and recovered digital service failures automatically based on event context, treating classification as mandatory before recovery. A prototype was built following the Prototyping Model, integrating n8n as workflow orchestrator and Apache Kafka as event broker inside Docker, with recovery governed by a JavaScript finite state machine, tested on five failure scenarios and two baseline conditions, each run for 30 iterations. The system classified all failures correctly across 150 iterations, reaching 100% recovery accuracy and a 100% success rate where fallback was available. Self-healing lowered Recovery Time by 29.90% against a no-recovery baseline, though a static-recovery baseline reached a lower time with 0% success, showing recovery speed and correctness can move in opposite directions.

Keywords: *Failure-Aware; Self-Healing; Event-Driven Architecture; Workflow Orchestration; Mean Time to Recovery (MTTR)*

Abstrak

Sistem layanan digital berbasis arsitektur *event-driven* umumnya mengandalkan *self-healing* pada tingkat infrastruktur, sementara kegagalan pada lapisan orkestrasi *workflow* ditangani secara reaktif melalui pemantauan manual sehingga proses bisnis dapat terhenti di tengah eksekusi meskipun infrastrukturnya masih berjalan normal. Penelitian ini merancang failure-aware *self-healing workflow* yang mendeteksi, mengklasifikasikan, dan memulihkan kegagalan layanan digital secara otomatis berdasarkan konteks *event*, dengan klasifikasi sebagai tahap wajib sebelum pemulihan dijalankan. Prototipe dibangun mengikuti Prototyping Model, mengintegrasikan n8n sebagai orkestrator *workflow* dan Apache Kafka sebagai *event broker* di dalam Docker, dengan keputusan pemulihan dikendalikan finite *state* machine berbasis JavaScript, diuji pada lima skenario kegagalan dan dua *baseline*, masing-masing 30 iterasi. Sistem mengklasifikasikan seluruh kegagalan secara tepat pada 150 iterasi, mencapai *recovery accuracy* 100% dan *success rate* 100% pada skenario dengan jalur *fallback*. *Self-healing* menurunkan MTTR sebesar 29,90% dibandingkan *baseline* tanpa pemulihan, meskipun *baseline* pemulihan statis mencatat waktu lebih rendah namun dengan *success rate* 0%, menunjukkan kecepatan dan ketepatan pemulihan dapat bergerak berlawanan arah.

Kata kunci*: Failure-Aware; Self-Healing; Arsitektur Event-Driven; Orkestrasi Workflow; Mean Time to Recovery (MTTR)*

1. Pendahuluan

Arsitektur *event-driven* menjadi tulang punggung layanan digital berskala besar yang memungkinkan setiap *microservice* berkomunikasi melalui pertukaran *event*, dengan *message broker* seperti Apache Kafka memberikan independensi tinggi antar layanan [1]. Independensi ini justru berisiko ketika banyak layanan berlangganan pada satu *event stream* yang sama, sebab

ketidaktersediaan satu komponen dapat merambat dan mengganggu bagian *workflow* lain yang sebenarnya tidak bergantung langsung padanya.

Bahaya terbesar muncul pada kegagalan sebagian atau *partial failure*, sebab sistem terdistribusi menjalankan setiap layanan pada ruang proses terisolasi sehingga kegagalan satu layanan tidak serta-merta terlihat oleh layanan lain yang sedang menunggu respons. Survei industri terhadap sistem *microservice* produksi mengidentifikasi *fault propagation* antar-layanan dan *silent failure* sebagai kategori kegagalan yang paling sulit dideteksi maupun diselesaikan [2], dengan gejala yang kerap baru muncul lama setelah pemicu sebenarnya terjadi.

Keterlambatan ini membawa konsekuensi terukur terhadap *Mean Time to Recovery* atau MTTR yang menjadi indikator baku keandalan operasional. Kajian Alnafessah et al. [3] terhadap riset DevOps menunjukkan bahwa penanganan kegagalan yang sekadar mengandalkan monitoring pasif dan investigasi manual cenderung menghasilkan respons yang keliru terhadap akar penyebab masalah, sejalan dengan temuan bahwa organisasi tanpa mekanisme diagnosis otomatis lebih rentan mengalami pemulihan yang lambat dan berbiaya tinggi [4].

Sistem *self-healing* dirancang menjawab pola reaktif tersebut melalui umpan balik otomatis yang memungkinkan sistem memulihkan diri tanpa campur tangan manusia, namun sebagian besar implementasi praktis masih beroperasi pada tingkat infrastruktur yang hanya memulihkan ketersediaan komponen fisik semata [5], tanpa menyentuh proses bisnis yang sedang berjalan karena *state workflow* akan hilang sepenuhnya atau terpaksa dimulai ulang dari awal.

Riset *fault tolerance* pada tingkat *workflow* mulai dikembangkan untuk menutup sebagian celah tersebut [6], namun umumnya masih menetapkan satu strategi pemulihan tunggal yang kaku dan seragam tanpa mempertimbangkan penyebab spesifik kegagalan, sehingga terbukti kesulitan beradaptasi begitu kondisi operasional berubah secara dinamis di lapangan [7]. Upaya *self-healing* pada arsitektur *microservice* otonom juga masih terkonsentrasi pada fase operasional infrastruktur [8].

Berbagai keterbatasan ini memunculkan tiga celah penelitian, yaitu *self-healing* tingkat infrastruktur yang mengabaikan *state workflow*, pendekatan *fault tolerance* yang merespons berbagai kegagalan secara seragam, serta minimnya eksplorasi pemulihan berbasis klasifikasi menggunakan *orchestrator* dan *message broker open-source* di luar platform berlisensi tertutup.

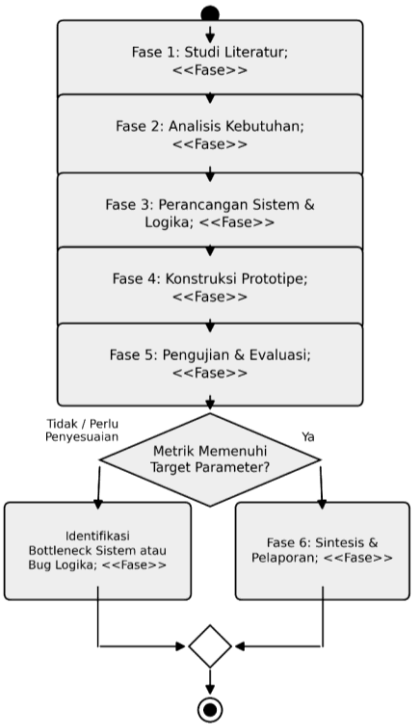
Penelitian ini menjawab ketiga celah tersebut dengan merancang *failure-aware self-healing workflow* yang menempatkan klasifikasi kegagalan sebagai prasyarat utama sebelum memulai pemulihan, dilandaskan pada bukti empiris bahwa diagnosis kegagalan yang memperhitungkan konteks *workflow* mampu melokalisasi akar masalah jauh lebih akurat dibandingkan sekadar memantau ketersediaan layanan [9], sementara pengayaan sinyal diagnostik gabungan juga terbukti meningkatkan akurasi identifikasi jenis kegagalan secara signifikan [10]. Klasifikasi akurat menjadi mutlak diperlukan karena strategi yang keliru terhadap jenis kegagalan justru berisiko memperpanjang waktu pemulihan. Cakupan penelitian ini berfokus murni pada lapisan orkestrasi *workflow* tanpa melibatkan manajemen tingkat infrastruktur, dengan empat kontribusi utama berupa arsitektur terintegrasi korelasi *event*, mekanisme pemulihan adaptif, prototipe *open-source* yang dapat direproduksi, serta evaluasi kuantitatif terhadap metrik keandalan pemulihan

2. Metodo*log*i

Penelitian ini menggunakan pendekatan kuantitatif eksperimental yang distrukturkan melalui *Prototyping Model* Pressman [11], dibangun secara nyata lalu diuji melalui serangkaian skenario terkendali untuk mengukur metrik evaluasi di dalam lingkungan Docker..

2.1 Alur Penelitian

Alur penelitian bergerak sistematis dari analisis masalah hingga sintesis temuan sebagaimana divisualisasikan pada Gambar 1, yang juga menyertakan jalur umpan balik dari fase evaluasi menuju fase sebelumnya untuk penyempurnaan apabila metrik hasil pengujian belum memenuhi target, guna memfasilitasi perbaikan *bottleneck* sistem atau *bug log*ika secara berkelanjutan.



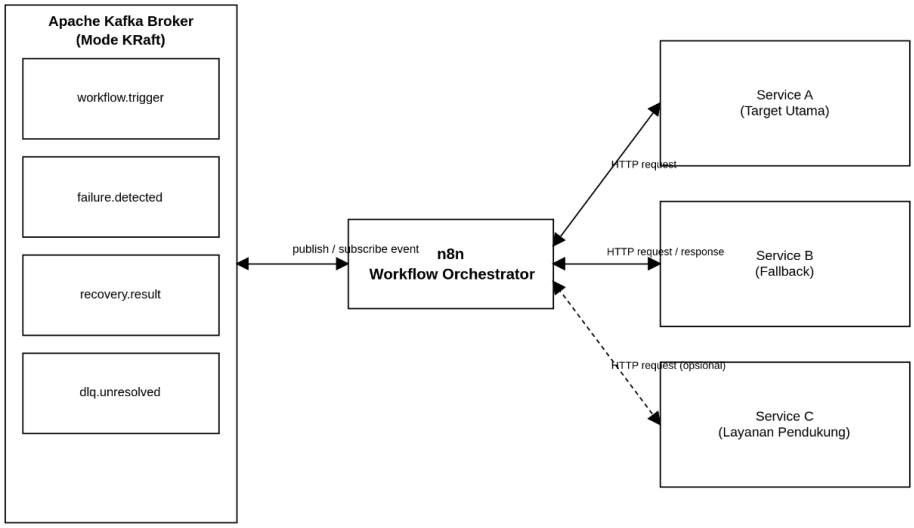
Gambar 1. Diagram Alur Penelitian

2.1.1 Fase Analisis Kebutuhan

Fase ini mengidentifikasi permasalahan kesenjangan antara *self-healing* tingkat infrastruktur dan kebutuhan pemulihan orkestrasi *workflow*. Kebutuhan fungsional menetapkan kemampuan sistem untuk mendeteksi kegagalan secara otomatis, mengklasifikasikan kategori kegagalan, dan mengeksekusi strategi pemulihan adaptif. Kebutuhan non-fungsional memastikan reproduktifitas eksperimen yang terkunci versinya, keterukuran empat parameter evaluasi, serta penggunaan komponen *open-source* yang sepenuhnya terjangkau.

2.1.2 Fase Perancangan Sistem

Fase perancangan menghasilkan empat keluaran yang saling berkaitan erat. Sistem mengintegrasikan Apache Kafka sebagai *event broker* dan n8n sebagai orchestrator *workflow* beserta tiga layanan REST API di dalam Docker Compose. Skema arsitektur disajikan pada Gambar 2.



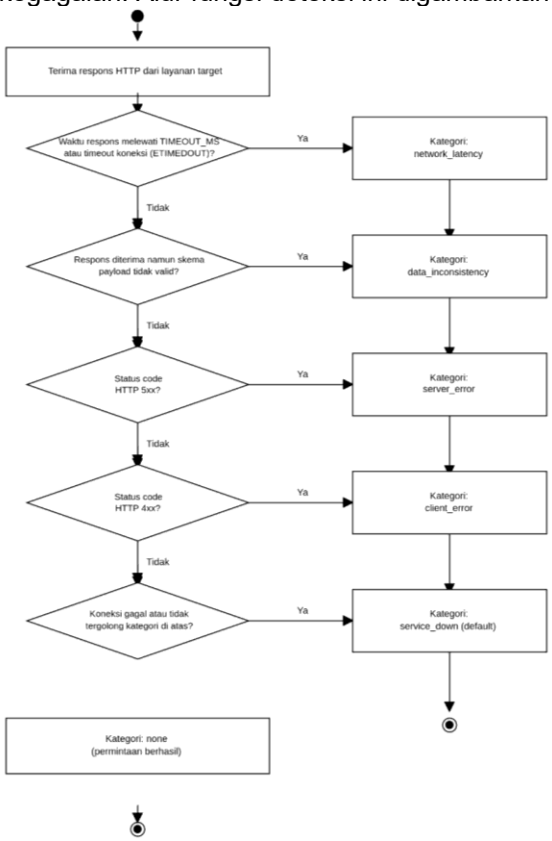
Gambar 2. Model Arsitektur Sistem

Injeksi kegagalan dieksekusi menggunakan perangkat Pumba yang terbukti efektif menguji ketahanan aplikasi *cloud-native* [12]. Rincian lima skenario kegagalan dan dua kondisi *baseline* dirangkum pada Tabel 1.

Tabel 1. Skenario Kegagalan dan Kondisi *Baseline*

Kode	Kondisi	Deskripsi	Strategi Pemulihan
S1	*Service down*	Kontainer Service A tidak dapat dijangkau (*connection refused*)	*Retry* 3x (*backoff* 5–15–45s) lalu *fallback* ke Service B
S2	*Network latency*	Respons melampaui ambang 5000 ms	*Retry* 2x dengan interval diperpanjang lalu *fallback* ke Service B
S3	*Data inconsistency*	Respons sukses (HTTP 2xx) tapi *payload* tidak sesuai skema JSON	Eskalasi langsung ke *dead letter queue*
S4	*Server error*	Layanan mengembalikan HTTP 5xx	*Retry* 3x (*backoff* 5–15–45s) lalu *fallback* ke Service B
S5	*Cascade failure*	Kegagalan bertingkat pada layanan utama dan layanan *fallback* secara bersamaan	Eskalasi ke *dead letter queue*
Baseline A	Tanpa *self-healing*	Tidak ada mekanisme pemulihan apa pun, sistem menunggu hingga *timeout*	*Timeout* tetap 120 detik
Baseline B	*Static recovery*	*Retry* seragam 3x (*backoff* 5–15–45s) untuk seluruh jenis kegagalan tanpa klasifikasi terlebih dahulu	*Retry* seragam lalu langsung ke DLQ

Keputusan pemulihan dikendalikan oleh Finite *State* Machine berbasis JavaScript yang ditanamkan dalam n8n. Sistem menganalisis status kode HTTP dan validitas skema respons untuk menetapkan jenis kegagalan. Alur fungsi deteksi ini digambarkan pada Gambar 3.



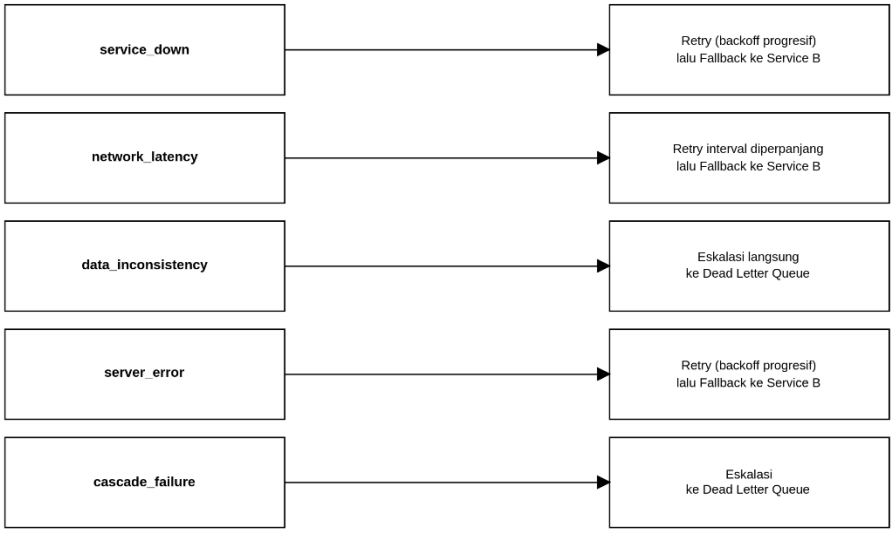
Gambar 3. Model *Failure Detection*

Sistem kemudian memetakan klasifikasi tersebut menuju strategi pemulihan spesifik yang sesuai. Aturan keputusan pemetaan ini dijabarkan pada Tabel 2.

Tabel 2. Aturan Keputusan yang Memetakan Jenis Kegagalan ke Strategi Pemulihan

Jenis Kegagalan	Sinyal yang Diamati	Strategi Terpilih
Service down	*Connection refused* / HTTP status 0	*Retry* lalu *fallback*
Network latency	*Response time* melewati ambang 5000 ms	*Retry* dengan interval diperpanjang lalu *fallback*
Data inconsistency	HTTP 2xx dengan *payload* tidak valid	Eskalasi ke DLQ (tidak ada *fallback* relevan)
Server error	HTTP status 5xx	*Retry* lalu *fallback*
Cascade failure	Kegagalan pada layanan utama dan *fallback* secara bersamaan	Eskalasi ke DLQ

Struktur *classification-first* memastikan sistem benar-benar memahami konteks sebelum bertindak. Strategi pemulihan dasar direpresentasikan pada Gambar 4.



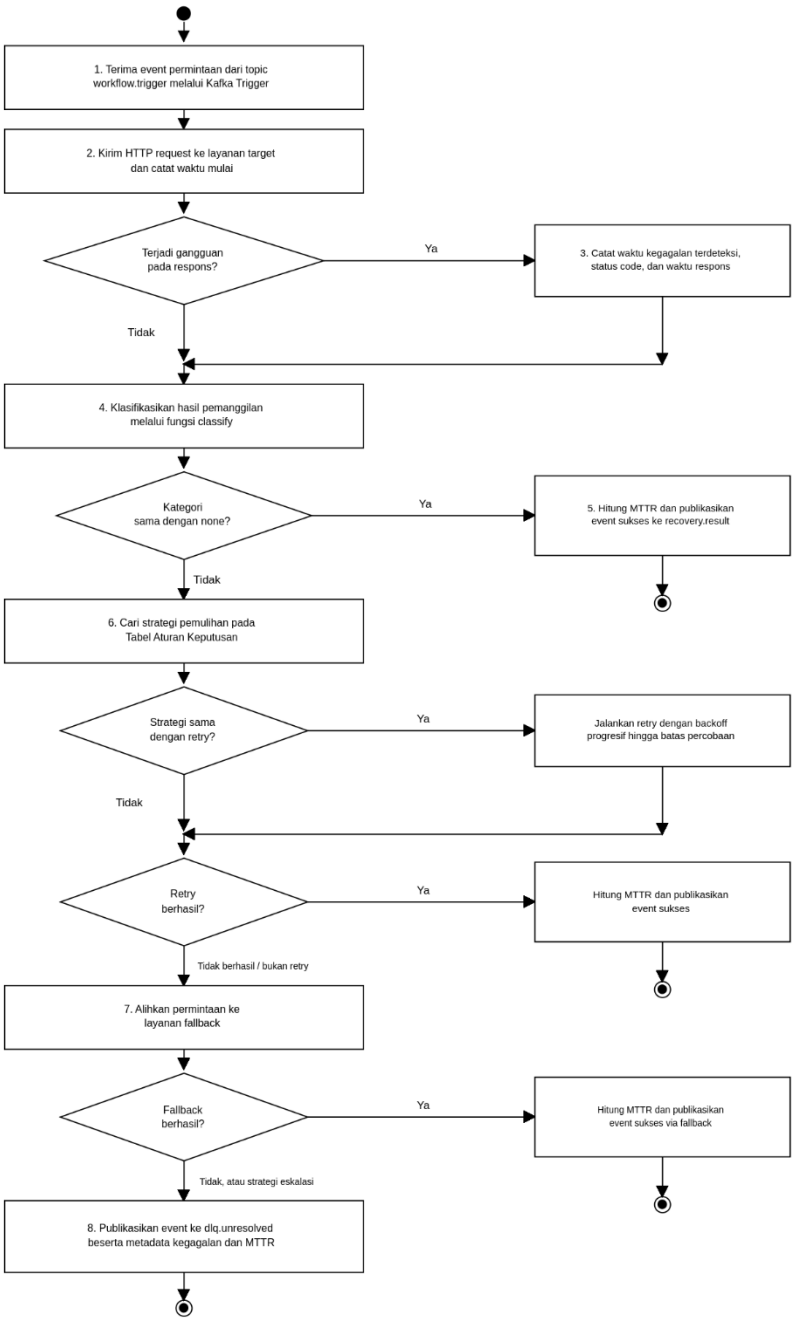
Gambar 4. Strategi *Self-Healing*

Integrasi seluruh proses tersebut tertuang dalam algoritma *workflow self-healing* yang tersusun dari delapan tahapan berurutan:

1. Sistem menerima *event* permintaan dari *topic workflow.trigger* melalui node Kafka Trigger, yang menjadi titik masuk setiap siklus eksekusi.
2. Sistem mengirimkan permintaan HTTP ke layanan target dan mencatat waktu mulai proses sebagai acuan awal perhitungan durasi.
3. Apabila terjadi gangguan pada respons yang diterima, sistem mencatat waktu kegagalan terdeteksi beserta *status code* dan waktu respons yang menyertainya.
4. Hasil pemanggilan tersebut diklasifikasikan melalui fungsi *classify* ke salah satu kategori kegagalan pada Tabel 2, atau ke kategori *none* apabila permintaan berhasil dan skema respons dinyatakan valid.
5. Apabila kategori bernilai *none*, sistem menghitung MTTR dan langsung mempublikasikan *event* sukses ke *topic* recovery.result, menandai berakhirnya siklus eksekusi.
6. Apabila kategori bernilai selain *none*, sistem mencari strategi pemulihan yang berpadanan pada Tabel 2, kemudian menjalankan *retry* dengan *backoff progresif* apabila strategi yang dipilih adalah *retry*, hingga batas percobaan tercapai
7. Apabila strategi yang dipilih adalah *fallback*, atau seluruh percobaan *retry* telah habis tanpa hasil, permintaan dialihkan ke layanan *fallback* yang telah ditentukan pada Model Arsitektur Sistem.

8. Apabila strategi yang dipilih adalah eskalasi, atau layanan *fallback* turut mengalami kegagalan, sistem mempublikasikan *event* ke *topic* dlq.unresolved beserta metadata kegagalan, sementara pada setiap titik akhir yang valid sistem menghitung MTTR sebagai selisih antara waktu berakhir dengan waktu mulai atau waktu kegagalan terdeteksi, sebelum mempublikasikan metrik akhir ke *topic* recovery.result.

Kedelapan langkah algoritma tersebut divisualisasikan dalam bentuk diagram alir pada Gambar 5 untuk memperjelas percabangan keputusan pada setiap tahap eksekusi.



Gambar 5. Algoritma *Workflow Self-Healing*

2.1.3 Fase Konstruksi

Fase ini mewujudkan rancangan konseptual menjadi prototipe utuh. Keempat komponen sistem dikemas dalam satu berkas Docker Compose dengan versi terkunci untuk konsistensi, topik Kafka dibuat pada *broker* mode KRaft, *workflow* orkestrasi dikonstruksi secara visual pada

n8n beserta komponen validasi JavaScript yang terhubung dengan Kafka Publisher, dan seluruh layanan diuji terisolasi sebelum pengujian *end-to-end*.

2.1.4 Fase Pengujian

Fase pengujian memverifikasi bahwa kelima jenis kegagalan pada Tabel 1 dapat diinjeksikan terkendali, dipantau konsisten, dan diukur dengan indikator jelas. Skenario yang diuji mencakup *service down*, *network latency*, *data inconsistency*, *server error*, dan *cascade failure*, ditambah *baseline* tanpa *self-healing* dan *baseline* pemulihan statis sebagai pembanding.

Injeksi kegagalan dijalankan menggunakan Pumba, *chaos testing tool* untuk Docker yang menginjeksikan kegagalan pada level kontainer tanpa modifikasi kode layanan, dipilih karena ringan, *time-bounded* melalui parameter durasi eksplisit, dan dapat direproduksi ulang melalui skrip yang terdokumentasi. Karena Pumba tidak dapat memanipulasi *payload* HTTP maupun kode status aplikasi, dua *endpoint* simulasi kustom disiapkan untuk melengkapi kategori kegagalan tersebut.

Pada S1, *service down* disimulasikan melalui pumba kill dengan sinyal SIGTERM pada kontainer rest-api-service-a tanpa parameter durasi, sehingga kontainer tetap mati hingga direstart manual. Pada S2, *network latency* disimulasikan melalui pumba netem delay dengan waktu tunda 8000 milidetik dan *jitter* 500 milidetik selama 90 detik, sengaja melampaui ambang 5000 milidetik pada fungsi *classify*. Pada S3, *data inconsistency* disimulasikan melalui *endpoint* /simulate/inconsistency yang mengembalikan skema JSON tidak valid. Pada S4, *server error* disimulasikan melalui *endpoint* /simulate/servererror yang mengembalikan status HTTP 5xx. Pada S5, *cascade failure* disimulasikan melalui dua perintah pumba kill paralel pada Service A dan Service B, mensimulasikan layanan utama dan *fallback* gagal bersamaan.

Performa sistem diukur melalui empat indikator, yaitu MTTR sebagai selisih waktu kegagalan terdeteksi dengan berakhirnya pemulihan, *response time* sebagai total waktu satu siklus hingga keputusan awal, *success rate* sebagai proporsi iterasi berhasil, dan *recovery accuracy* sebagai proporsi kegagalan yang berhasil diklasifikasikan tepat.

Parameter yang dipantau tiap iterasi mencakup *status code* HTTP, waktu respons, validitas skema *payload* JSON, serta jumlah *retry* sebelum berpindah ke *fallback* atau eskalasi, dengan aktivitas keempat *topic* Kafka turut dipantau agar tidak terjadi penumpukan data. Waktu deteksi kegagalan tidak dilaporkan sebagai metrik berdiri sendiri, melainkan komponen *timestamp* dalam perhitungan MTTR.

Ketepatan respons ditentukan melalui pencocokan catatan jenis kegagalan yang diinjeksikan sebagai *ground truth* dengan *field* rule_applied pada *log* n8n dan Kafka; keputusan dinyatakan tepat apabila aturan yang aktif sesuai jenis kegagalan yang diinjeksikan. Kasus manifestasi ambigu didokumentasikan sebagai catatan, bukan langsung dinilai salah.

Setiap iterasi mengikuti langkah identik, yaitu Pumba atau *endpoint* simulasi menginjeksikan kegagalan, *event* dipublikasikan ke *topic workflow.trigger*, *workflow* n8n mengeksekusi algoritma *self-healing* sesuai Subbab 2.1.2, dan hasil direkam sebagai keluaran terstruktur berisi identitas layanan, jenis kegagalan, strategi, jumlah *retry*, waktu mulai-berakhir, serta MTTR yang dianalisis statistik sesuai Subbab 2.3.

Apabila metrik belum memenuhi target, penyebabnya ditelusuri kembali ke fase konstruksi atau perancangan untuk diperbaiki, dan pengujian diulang hingga metrik tercapai atau penyebabnya dapat dijelaskan, sebagaimana dibahas pada Subbab 3.6 untuk MTTR yang melampaui ambang 60 detik pada tiga dari lima skenario.

2.2 Lingkungan Implementasi

Reproduktifitas sistem dipertahankan dengan mengunci versi n8n pada 2.15.0 [13] dan Apache Kafka pada versi 4.2 [14], dengan keseluruhan *stack* komunikasi berbasis JSON diorkestrasi melalui satu berkas lingkungan terpusat.

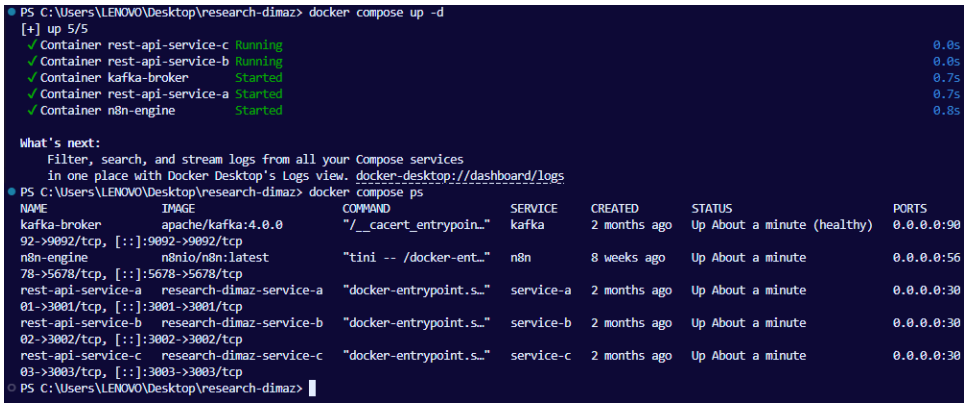
2.3 Analisis Statistik

Pengolahan nilai MTTR memanfaatkan pustaka Python, dengan uji normalitas Shapiro-Wilk dan kesetaraan varians Levene membuktikan data melanggar asumsi parametrik, sehingga uji non-parametrik Mann-Whitney U diterapkan untuk membandingkan kelompok *baseline* terhadap skenario utama.

3. Hasil dan Pembahasan

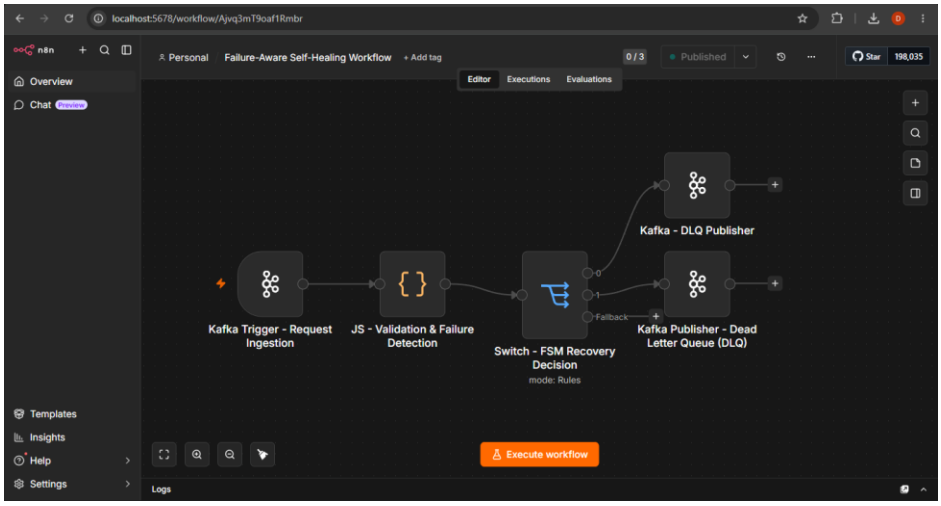
3.1 Hasil Konstruksi Prototipe

Hasil pengujian ini memvalidasi kelancaran seluruh mekanisme eksekusi. Kondisi operasional kelima kontainer lingkungan Docker dikonfirmasi berjalan optimal seperti terlihat pada Gambar 6.



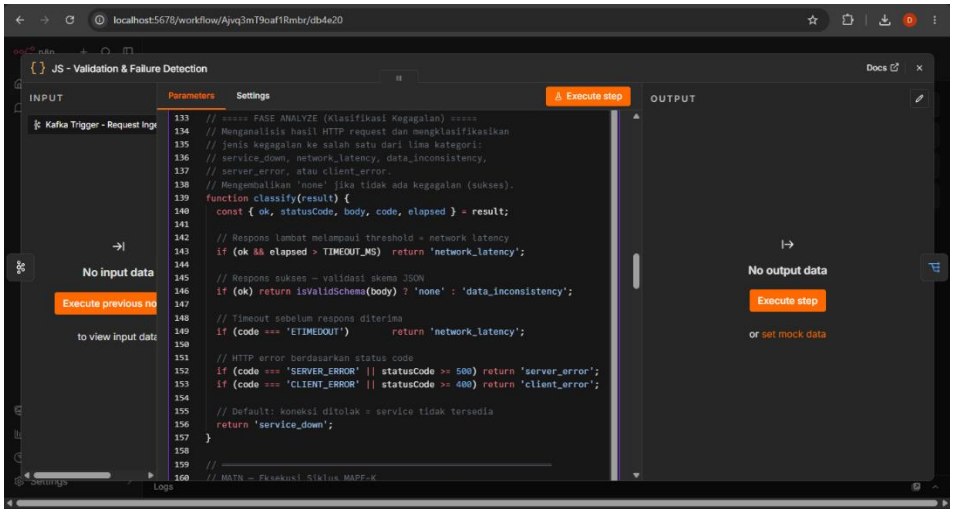
Gambar 6. Verifikasi Lingkungan Docker Compose

Struktur *workflow* orkestrasi yang matang diperlihatkan pada Gambar 7 yang memuat seluruh node fungsional untuk penerimaan, validasi, penentuan, hingga pencatatan *event* pemulihan.



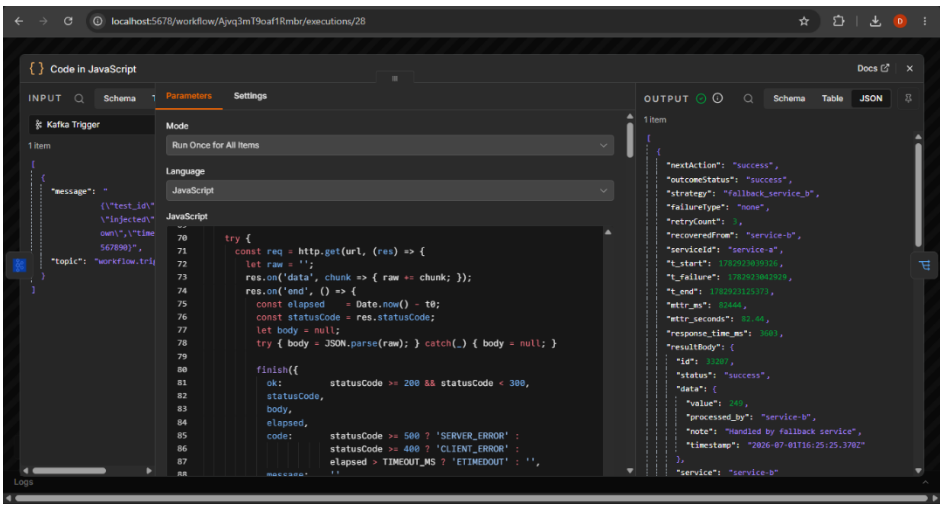
Gambar 7. Implementasi *Workflow Self-Healing* pada n8n

Fungsi klasifikasi yang mendasari kecerdasan sistem divalidasi kinerjanya pada Gambar 8. Evaluasi kode menunjukkan klasifikasi dieksekusi teguh pada karakteristik respons aktual dan bukan sekadar asumsi statis awal.



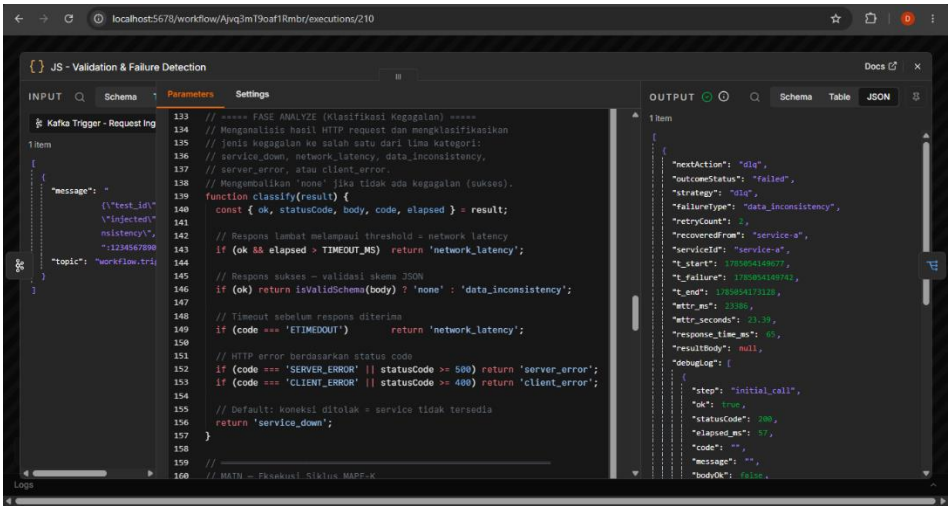
Gambar 8. Implementasi *Log*ika *Failure Detection*

Eksekusi yang berakhir pada keberhasilan *fallback* tergambar pada Gambar 9. Sistem memulihkan kegagalan layanan utama dengan mengalihkan permintaan ke layanan cadangan secara terukur.



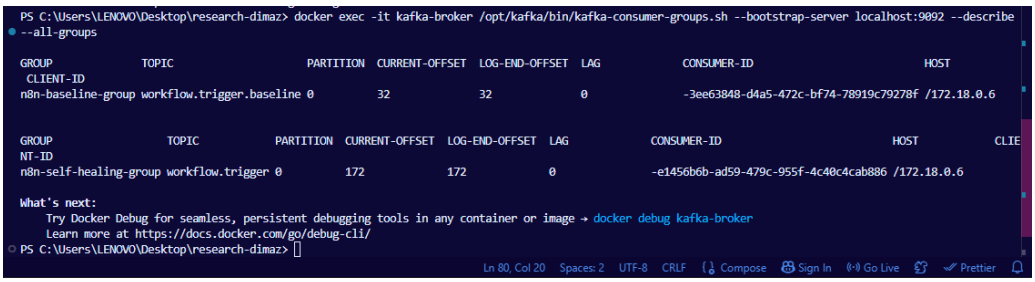
Gambar 9. Eksekusi *Self-Healing* - Jalur *Fallback* Berhasil

Skenario eskalasi kegagalan yang berujung pada *dead letter queue* divisualisasikan pada Gambar 10. Keputusan pengalihan otomatis ini menegaskan bahwa sistem merespons masalah secara sangat akurat sesuai jenis kerusakannya.



Gambar 10. Eksekusi *Self-Healing* - Jalur Eskalasi DLQ

Kondisi offset pesan di dalam Kafka yang bebas dari antrean tertunda ditampilkan pada Gambar 11 yang memperlihatkan keandalan proses penyaluran data *real-time*.



Gambar 11. Status Consumer Group, Offset, dan Lag pada *Topic* Kafka

Tampilan eksekusi perintah kafka-consumer-groups.sh pada Gambar 11 mengonfirmasi bahwa consumer group n8n-*self-healing*-group secara aktif mengonsumsi *event* permintaan dari *topic workflow.trigger*. Nilai Lag yang bernilai 0 dengan Current Offset sebesar 172 yang setara dengan *Log* End Offset membuktikan bahwa seluruh *event* pesan Kafka yang dikirimkan berhasil diproses secara penuh oleh *workflow* n8n tanpa mengalami penumpukan atau penundaan antrean data yang tertahan pada *broker*.

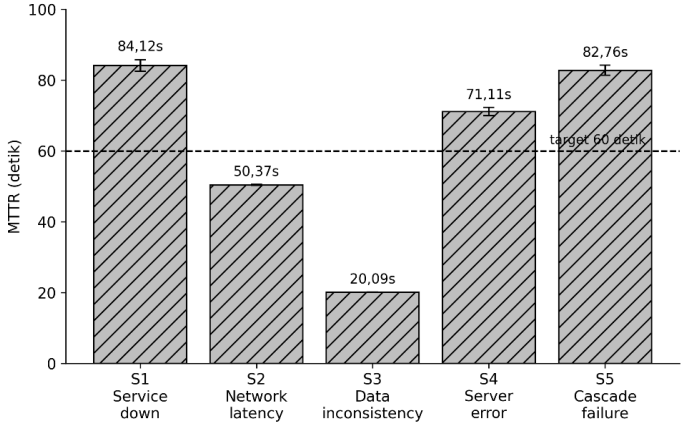
3.2 Hasil Skenario Kegagalan

Pengujian kelima skenario utama diulang secara ketat hingga mencapai 150 iterasi. Ringkasan perolehan kinerja secara menyeluruh dirangkum pada Tabel 3.

Tabel 3. Ringkasan Hasil Skenario Kegagalan (N=30 per skenario)

Skenario	MTTR Rerata (dtk)	SD MTTR (dtk)	*Success rate* (%)	*Response time* Rerata (ms)	*Recovery accuracy* (%)
S1 – *Service down*	84.12	1.66	100	3808.13	100
S2 – *Network latency*	50.37	0.11	100	5053.93	100
S3 – *Data inconsistency*	20.09	0.06	100	5.33	100
S4 – *Server error*	71.11	1.16	100	5.20	100
S5 – *Cascade failure*	82.76	1.43	100	3763.73	100

Nilai rerata waktu pemulihan beserta deviasi standar dari seluruh skenario dipetakan terhadap ambang batas 60 detik pada Gambar 12.



Gambar 12. Rerata MTTR per Skenario Kegagalan (S1–S5) terhadap Ambang Target 60 Detik

Akurasi pemulihan yang mencapai angka sempurna membuktikan mekanisme validasi skema merespons masalah dengan kecerdasan yang diharapkan. Keberhasilan penyelesaian iterasi tanpa kebocoran *event* membuahkan *success rate* 100% yang selaras dengan setiap skenario tersedia. Metrik *response time* mengungkap kecepatan *round-trip* rute pemulihan yang bervariasi bergantung pada ketersediaan *fallback* maupun operasi eskalasi lokal. Performa sistem pada skenario S4 sengaja dicatat sebagai pertanyaan terbuka untuk audit simulator lanjutan di luar ranah penelitian ini.

3.3 Evaluasi Hasil Pengujian

Sintesis metrik menelaah empat dimensi performa sistem yang sangat lazim digunakan pada riset *self-healing* tingkat lanjut.

3.3.1 *Recovery Time*

Rentang pemulihan berfluktuasi antara 20,09 hingga 84,12 detik, dengan skenario berjeda interval *retry* lebih panjang terbukti melampaui target waktu pemulihan secara wajar akibat desain kehati-hatian sistem.

3.3.2 *Availability*

Ketepatan klasifikasi yang berujung pada diagnosis benar menembus angka mutlak 100%, menjadi pembeda konseptual terbesar dibandingkan kebijakan statis konvensional.

3.3.3 *Reliability*

Ketepatan klasifikasi yang berujung pada diagnosis benar menembus angka mutlak 100 persen. Keakuratan pengambilan keputusan inilah yang melahirkan pembeda konseptual terbesar dibandingkan kebijakan statis konvensional.

3.3.4 *Throughput*

Kapasitas penyelesaian siklus merespons langsung terhadap jalur eksekusi spesifik skenario. Hasil estimasi *Throughput* secara komprehensif dirangkum pada Tabel 4.

Tabel 4. Estimasi *Throughput* per Skenario Kegagalan

Skenario	*Response time* Rerata (ms)	*Throughput* Estimasi (req/detik)
S1 – *Service down*	3.808,13	0,26
S2 – *Network latency*	5.053,93	0,20
S3 – *Data inconsistency*	5,33	187,62
S4 – *Server error*	5,20	192,31
S5 – *Cascade failure*	3.763,73	0,27

Perbedaan drastis yang diamati menyoroti panjangnya rute jaringan yang wajib ditempuh pada skenario *fallback* sungguhan dibandingkan operasi eskalasi lokal.

3.3.5 Analisis Perbandingan

Seluruh dimensi performa disatukan pada Tabel 5 untuk memperlihatkan korelasi pola yang tersembunyi.

Tabel 5. Sintesis Multi-Metrik: *Recovery Time*, *Availability*, *Reliability*, dan *Throughput*

Kondisi	*Recovery Time* (dtk)	*Availability* (%)	*Reliability* (%)	*Throughput* Estimasi (req/dtk)
S1 – *Service down*	84,12	100	100	0,26
S2 – *Network latency*	50,37	100	100	0,20
S3 – *Data inconsistency*	20,09	100	100	187,62
S4 – *Server error*	71,11	100	100	192,31
S5 – *Cascade failure*	82,76	100	100	0,27
Baseline A (tanpa *self-healing*)	120,00	0	0	tidak bermakna (tidak pernah merespons)
Baseline B (pemulihan statis)	28,02	0	0	tidak sebanding (ada pada catatan Subbab 3.3.4)

Pola sintesis menegaskan bahwa kecepatan pemulihan dan kualitas ketepatan pemulihan kerap bergerak saling berlawanan arah dan memerlukan kompromi cermat.

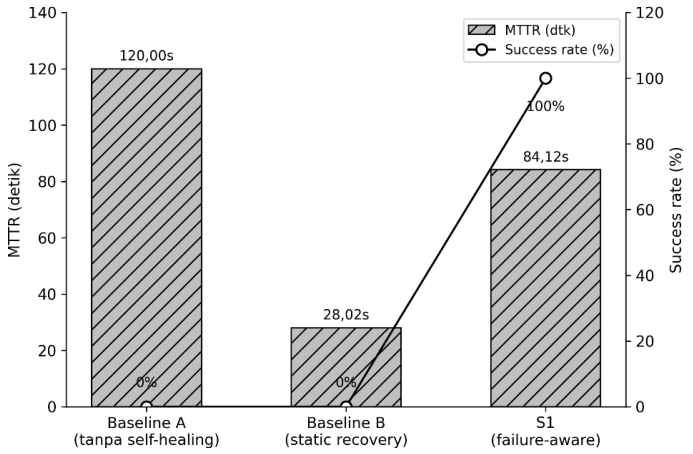
3.4 Perbandingan *Baseline*

Tingkat keandalan skenario pemulihan utama dijajarkan secara langsung terhadap kedua kelompok pengontrol pada Tabel 6

Tabel 6. Kondisi *Baseline* Dibandingkan terhadap Skenario S1

Kondisi	MTTR Rerata (dtk)	*Success rate* (%)	*Recovery accuracy* (%)	Strategi
Baseline A – No *self-healing*	120.00	0	0	Menunggu hingga *timeout* tetap
Baseline B – *Static recovery*	28.02	0	0	*Retry* seragam 3x lalu langsung ke DLQ
S1 – *Failure-aware recovery*	84.12	100	100	*Retry* lalu *fallback* berdasarkan klasifikasi

Trade-off paling radikal ditunjukkan oleh kondisi *Baseline* B yang justru mengalami kegagalan performa mutlak di balik kecepatan pemulihannya. Pola berbanding terbalik tersebut divisualisasikan berdampingan secara jelas pada Gambar 13.



Gambar 13. MTTR dan *Success rate* pada *Baseline* A, *Baseline* B, dan Pemulihan Failure-Aware (S1)

3.5 Pengujian Statistik Inferensial

Ketidaksesuaian distribusi normalitas menuntut implementasi uji non-parametrik yang terbukti sangat efektif menelaah pola hipotesis penelitian ini.

Tabel 7. Hasil Uji Mann-Whitney U

Perbandingan	Mean A	Mean B	Statistik U	p-value	Keputusan
Baseline A vs S1	120.00s	84.12s	900.00	< 0.001	Tolak H0
Baseline B vs S1	28.02s	84.12s	0.00	1.000	Gagal tolak H0

Perbandingan terhadap *Baseline* A mendukung klaim utama penelitian. Penurunan MTTR 29,90% signifikan pada p<0,001, menunjukkan *self-healing* mempercepat pemulihan dibandingkan tanpa mekanisme otomatis, meski masih di bawah target 50% yang ditetapkan sejak awal, sebagaimana dibahas pada Subbab 3.6.

Perbandingan terhadap *Baseline* B lebih kompleks. Statistik U=0,00 dengan p=1,000 menunjukkan setiap iterasi *Baseline* B memiliki MTTR lebih rendah dibandingkan setiap iterasi S1, berkebalikan dari hipotesis awal, namun berdampingan dengan *success rate* 0% pada Tabel 6, maknanya berubah signifikan.

Uji proporsi satu sampel membandingkan *recovery accuracy* 100% pada 150 iterasi terhadap ambang tebakan acak 20%. Hasil Z=24,49 dengan p<0,0001 mengonfirmasi klasifikasi bekerja jauh melampaui tebakan acak, diperoleh dari pencocokan *ground truth* dan rule_applied sesuai Subbab 2.1.4.

3.6 Pembahasan

Baseline B mencapai MTTR lebih rendah bukan karena lebih efisien, melainkan karena menyerah lebih cepat. Begitu tiga *retry* seragamnya gagal, sistem langsung eskalasi ke *dead letter queue* tanpa strategi alternatif, sehingga clock-nya berhenti lebih awal. S1 memerlukan waktu lebih lama karena mengklasifikasikan kegagalan secara tepat sebagai kegagalan yang memerlukan *fallback*, lalu mengeksekusi *fallback* tersebut hingga layanan pulih. Kondisi ini sejalan dengan moda kegagalan yang dijelaskan Nikolic et al. [7] pada kebijakan pemulihan tetap, di mana strategi yang ditetapkan sekali pada waktu desain tidak dapat menyesuaikan diri ketika kondisi nyata menyimpang dari yang diantisipasi. Perbandingan *success rate* 0% pada *Baseline* B dengan 100% pada *classification-first* mengubah argumen tersebut dari teori menjadi bukti terukur.

S1, S4, dan S5 melampaui ambang 60 detik, masing-masing 84,12, 71,11, dan 82,76 detik, berbagi penyebab yang sama, yaitu tiga *retry* dengan *backoff progresif* lima, lima belas, lalu empat puluh lima detik yang saja sudah berjumlah 65 detik. Ini pilihan desain disengaja untuk menghindari kesalahan klasifikasi terhadap kegagalan transien yang akan pulih sendiri, sekaligus menghindari *thundering herd* akibat beralih ke *fallback* sebelum layanan utama sempat pulih. S2 dan S3 memenuhi ambang 60 detik karena skema *retry*-nya lebih sedikit atau intervalnya lebih pendek, menyiratkan target seragam 60 detik mungkin terlalu kaku untuk kegagalan yang memerlukan jendela verifikasi berbeda.

Perbandingan dengan Camunda Platform 8 pada penelitian Rasheedh et al. [6] relevan diangkat di sini. Aydın dan Çebi [15] menunjukkan bahwa platform semacam ini umumnya bergantung pada jalur pemulihan statis tanpa mekanisme klasifikasi *runtime*. Desain *Baseline* B mencerminkan *log*ika statis serupa, dan kegagalan totalnya menghadapi *service down* yang sebenarnya memerlukan *fallback* membuktikan secara empiris apa yang sebelumnya baru diargumentasikan secara konseptual.

Zhang et al. [16] memperkenalkan *durable execution* yang mempertahankan progres *workflow* melalui kegagalan infrastruktur, namun umumnya tidak terintegrasi nyata dengan Kafka dan memerlukan *deployment* server terpisah. Raptis et al. [17] mengukur performa Kafka secara mendalam namun tidak memperluas analisis hingga orkestrasi pemulihan di atas lapisan *broker*. Penelitian ini mengisi celah tersebut, menunjukkan n8n dan Kafka dapat berfungsi sebagai mekanisme pemulihan terkoordinasi tanpa infrastruktur tambahan *durable execution* maupun kompleksitas lisensi seperti Camunda.

Pola yang muncul adalah *trade-off* nyata, bukan kemenangan sepihak. Mekanisme *classification-first* mencapai *recovery accuracy* 100% dan *success rate* 100% pada skenario dengan *fallback* tersedia, serta pulih signifikan lebih cepat dibandingkan tanpa mekanisme otomatis, namun tidak lebih cepat dibandingkan kebijakan statis, karena kecepatan kebijakan tersebut berasal dari menyerah lebih awal, bukan menyelesaikan pemulihan. Rancangan *self-healing workflow* mendatang sebaiknya memperlakukan *trade-off* ini sebagai keputusan desain eksplisit.

4. Simpulan

Penelitian ini menjawab empat pertanyaan seputar failure-aware *self-healing* pada lapisan orkestrasi *workflow*, dibuktikan dari 150 iterasi skenario dan 60 iterasi *baseline*. Sistem berhasil mendeteksi dan mengklasifikasikan setiap kegagalan secara tepat, mencapai *recovery accuracy* 100% pada seluruh lima skenario, termasuk S3 dan S5 yang keluaran benarnya berupa eskalasi akurat ke *dead letter queue*. Mekanisme pemulihan mengadaptasi respons sesuai jenis kegagalan, memetakan *service down* dan *server error* ke *fallback*, *network latency* ke *retry* diperpanjang, serta *data inconsistency* dan *cascade failure* ke eskalasi DLQ, seluruhnya dieksekusi tepat melalui integrasi empat *topic* Kafka dengan *log*ika orkestrasi n8n.

Performa metrik keandalan tidak seragam. *Self-healing* menurunkan MTTR 29,90% dibandingkan *baseline* tanpa pemulihan, signifikan pada $p<0{,}001$, meski masih di bawah target 50%. Tiga dari lima skenario melampaui ambang MTTR 60 detik, konsekuensi desain *retry-before-fallback* yang sengaja konservatif. Yang paling menonjol, *baseline* pemulihan statis mencapai MTTR lebih rendah namun *success rate* 0%, mengonfirmasi kecepatan dan ketepatan pemulihan bukan hal yang sama.

Kontribusi utama penelitian ini adalah pembuktian empiris bahwa arsitektur *classification-first* menukar sebagian kecepatan pemulihan dengan hasil yang jauh lebih andal, dan *trade-off* ini seharusnya diperlakukan sebagai keputusan desain, bukan efek samping yang perlu dioptimalkan hingga hilang. Penelitian lanjutan yang memperluas perbandingan *baseline* ke kondisi *network latency* dan *data inconsistency* akan memperkuat validitas eksternal temuan ini.

Daftar Referensi

[1] S. Kul, I. Tashiev, A. Şentaş, and A. Sayar, "*Event*-Based *Microservice*s With Apache Kafka Streams: A *Real-time* Vehicle Detection System Based on Type, Color, and Speed Attributes," *IEEE Access*, vol. 9, pp. 83137–83148, 2021, doi: 10.1109/ACCESS.2021.3085736.

[2] X. Zhou *et al.*, "Fault Analysis and Debugging of *Microservice* Systems: Industrial Survey, Benchmark System, and Empirical Study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2021, doi: 10.1109/TSE.2018.2887384.

[3] A. Alnafessah, A. U. Gias, R. Wang, L. Zhu, G. Casale, and A. Filieri, "Quality-Aware DevOps Research: Where Do We Stand?," *IEEE Access*, vol. 9, pp. 44476–44489, 2021, doi: 10.1109/ACCESS.2021.3064867.

[4] P. Desaraju, "*Self-healing* Software Systems: AI-Driven Fault Prediction and Recovery," *World Journal of Advanced Engineering Technology and Sciences*, vol. 18, no. 3, pp. 064–071, Mar. 2026, doi: 10.30574/wjaets.2026.18.3.0120.

[5] B. Magableh and M. Almiani, "A Self Healing *Microservice*s Architecture: A Case Study in Docker Swarm Cluster," in *Advances in Intelligent Systems and Computing*, Springer Verlag, 2020, pp. 846–858. doi: 10.1007/978-3-030-15032-7_71.
[6] J. A. Rasheedh, G. N. Ahmed, S. H. Abdul Cader, and K. Nirmala, "*Fault tolerance* Enhancement in *Microservice*s using Orchestration Process with Agile," in *2024 8th International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, 2024, pp. 406–413. doi: 10.1109/I-SMAC61858.2024.10714589.
[7] J. Nikolic, N. Jubatyrov, and E. Pournaras, "*Self-healing* Dilemmas in Distributed Systems: *Fault correction* vs. *Fault tolerance*," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 2728–2741, Sep. 2021, doi: 10.1109/TNSM.2021.3092939.
[8] A. Bucchiarone, C. Guidi, I. Lanese, N. Bencomo, and J. Spillner, "A MAPE-K Approach to Autonomic *Microservice*s," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, 2022, pp. 100–103. doi: 10.1109/ICSA-C54293.2022.00025.
[9] T. Wang, W. Zhang, J. Xu, and Z. Gu, "*Workflow*-Aware Automatic Fault Diagnosis for *Microservice*-Based Applications With Statistics," *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2350–2363, 2020, doi: 10.1109/TNSM.2020.3022028.
[10] Y. Pan, M. Ma, X. Jiang, and P. Wang, "DyCause: Crowdsourcing to Diagnose *Microservice* Kernel Failure," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 6, pp. 4763–4777, 2023, doi: 10.1109/TDSC.2022.3233915.
[11] R. S. Pressman, *Software engineering: A practitioner's approach*, 7th ed. McGraw-Hill, 2010.
[12] A. Al-Said Ahmad, L. F. Al-Qora'n, and A. Zayed, "Exploring the impact of *chaos engineering* with various user loads on cloud native applications: an exploratory empirical study," *Computing*, vol. 106, no. 7, pp. 2389–2425, Jul. 2024, doi: 10.1007/s00607-024-01292-z.
[13] n8n, "n8n Documentation," docs.n8n.io. Accessed: May 06, 2026. [Online]. *Available*: [https://docs.n8n.io/]
[14] Apache Software Foundation, "Apache Kafka Documentation," kafka.apache.org. Accessed: May 06, 2026. [Online]. *Available*: [https://kafka.apache.org/documentation/]
[15] S. Aydin and C. B. Çebi, "Comparison of Choreography vs Orchestration Based Saga Patterns in *Microservice*s," in *2022 International Conference on Electrical, Computer and Energy Technologies (ICECET)*, 2022, pp. 1–6. doi: 10.1109/ICECET55527.2022.9872665.
[16] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Fault-tolerant and transactional *state*ful serverless *workflow*s," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, Nov. 2020, pp. 1187–1204. [Online]. *Available*: https://www.usenix.org/conference/osdi20/presentation/zhang-haoran
[17] T. P. Raptis, C. Cicconetti, and A. Passarella, "Efficient *topic* partitioning of Apache Kafka for high-*Reliability real-time* data streaming applications," *Future Generation Computer Systems*, vol. 154, pp. 173–188, May 2024, doi: 10.1016/j.future.2023.12.028.