UJIAN AKHIR SEMESTER
SISTEM PARALEL DAN TERDISTRIBUSI

Oleh Kelompok :

Isnaini Zayyana Fitri 11221072

Institut Teknologi Kalimantan

2025

BAB I
PENDAHULUAN

1.1 Latar Belakang

Dalam pengembangan perangkat lunak modern, arsitektur terdistribusi telah menjadi standar untuk menangani skalabilitas sistem. Namun, tantangan mendasar yang sering muncul dalam lingkungan ini adalah ketidakpastian jaringan dan risiko inkonsistensi data. Kegagalan transmisi (network failure) sering kali memicu mekanisme pengiriman ulang (retry) yang, tanpa penanganan yang tepat, dapat menyebabkan duplikasi data dan merusak integritas informasi.

Berangkat dari permasalahan tersebut, proyek ini mengembangkan sebuah sistem Pub-Sub Log Aggregator Terdistribusi yang dirancang untuk menangani pengiriman data event dalam volume tinggi dengan tingkat keandalan (reliability) maksimal. Sistem ini berfokus pada arsitektur yang mampu menjamin konsistensi data mutlak (data integrity), memastikan bahwa sistem tetap tangguh (resilient) meskipun menghadapi kegagalan jaringan sementara atau beban kerja yang fluktuatif.

1.2 Tujuan

Tujuan utama dari pengembangan sistem ini adalah:

1. Membangun Arsitektur Terisolasi: Mengimplementasikan sistem dengan pendekatan Microservices yang diorkestrasi menggunakan Docker Compose, memfasilitasi isolasi layanan yang bersih dan proses deployment yang konsisten.
2. Mengimplementasikan Semantik Exactly-Once Processing: Menerapkan mekanisme Idempotent Consumer, di mana setiap pesan yang masuk dijamin hanya akan diproses satu kali, meniadakan efek samping negatif dari pengiriman berulang (retry storm) oleh pengirim.
3. Menjamin Integritas Data Konkuren: Mencegah terjadinya race condition saat beberapa worker bekerja secara paralel melalui mekanisme deduplikasi berbasis basis data dan kontrol konkurensi menggunakan transaksi ACID (Atomicity, Consistency, Isolation, Durability).

BAB II
LANDASAN TEORI

2.1 Karakteristik Sistem Terdistribusi dan Trade-off Desain

Coulouris et al. (2012) mendefinisikan sistem terdistribusi berdasarkan karakteristik utamanya, yaitu Heterogeneity (keragaman perangkat keras/lunak) dan Openness (kemampuan diperluas). Sementara itu, van Steen dan Tanenbaum (2023) menekankan pada Distribution Transparency (penyembunyian kompleksitas fisik).

Dalam desain Log Aggregator ini, kedua konsep tersebut digabungkan. Melalui Docker, aspek Heterogeneity (Coulouris) ditangani dengan membungkus dependensi dalam container standar, sekaligus mencapai Transparency (van Steen) di mana Aggregator tidak perlu mengetahui lokasi fisik Publisher. Trade-off desain utama yang diambil adalah menukar kesederhanaan infrastruktur demi Skalabilitas (Coulouris, Bab 1.2). Penggunaan message broker (Redis) menambah kompleksitas konfigurasi, namun memberikan kemampuan menangani beban log yang fluktuatif tanpa mengubah kode aplikasi utama.

2.2 Arsitektur Publish–Subscribe vs Client–Server

Arsitektur Client-Server tradisional bekerja secara sinkron (coupled). Coulouris et al. (2012, Bab 6) mengklasifikasikan Publish-Subscribe sebagai bentuk Indirect Communication, yang menawarkan dua jenis decoupling vital:

1. Space Decoupling: Pengirim dan penerima tidak perlu mengetahui identitas satu sama lain (hanya lewat Broker).
2. Time Decoupling: Pengirim dan penerima tidak perlu aktif pada saat yang bersamaan.

Arsitektur ini dipilih karena alasan teknis time-decoupling. Berdasarkan analisis terhadap rancangan ini, jika menggunakan model request-reply (Client-Server), kegagalan sesaat pada database akan menyebabkan Publisher mengalami timeout atau error berantai. Dengan Pub-Sub, Broker bertindak sebagai penyangga (buffer), memungkinkan Publisher mengirim log dengan cepat meskipun Aggregator sedang melakukan restart atau lambat menulis ke disk.

2.3 Semantik Pengiriman (At-least-once vs Exactly-once)

Coulouris et al. (2012, Bab 6.4) menjelaskan semantik pemanggilan (invocation semantics) dalam komunikasi jarak jauh. Mencapai Exactly-once murni sulit karena risiko kegagalan pengiriman acknowledgement. Oleh karena itu, protokol jaringan umumnya mengadopsi semantik At-least-once (pesan dikirim ulang jika tidak ada balasan, berisiko duplikasi).

Untuk mengatasi duplikasi akibat At-least-once, van Steen dan Tanenbaum (2023) menyarankan penggunaan Idempotent Consumer. Dalam implementasi ini, Aggregator bertindak idempotent: sistem menerima pesan ganda dari Broker, tetapi lapisan database memfilter duplikat tersebut. Hal ini mengubah semantik At-least-once dari jaringan menjadi semantik pemrosesan Effectively Exactly-once di level aplikasi.

2.4 Skema Penamaan Topic dan Event_ID

Sistem penamaan (Naming) vital untuk identifikasi sumber daya. Coulouris et al. (2012, Bab 9) membedakan antara Names (bisa dibaca manusia) dan Identifiers (unik mesin). Sementara itu, van Steen dan Tanenbaum (2023) menyebut pengenal mesin ini sebagai Flat Naming.

Guna mendukung deduplikasi yang kuat, sistem ini menerapkan Flat Naming menggunakan UUID (Universally Unique Identifier) pada event_id. Coulouris et al. (2012) mencatat bahwa UUID memungkinkan pembuatan ID unik secara terdistribusi tanpa perlu koordinasi pusat (coordination-free), menghindari bottleneck. Atribut topic digunakan sebagai nama logis, sedangkan kombinasi (topic, event_id) menjadi kunci unik global untuk mendeteksi tabrakan data (collision) antar-worker.

2.5 Ordering dan Logical Clock

Sinkronisasi waktu fisik sulit karena Clock Drift dan Skew. Coulouris et al. (2012, Bab 14) menjelaskan bahwa karena batasan sinkronisasi jam fisik (NTP), sistem terdistribusi sering membutuhkan Logical Clocks (seperti Lamport Clock) untuk menentukan urutan kejadian (happens-before).

Dalam implementasi praktis ini, Total Ordering global tidak dipaksakan karena biaya latensinya tinggi. Sistem menggunakan received_at (waktu server) sebagai penghitung monotonik lokal (monotonic counter) untuk mengurutkan log saat sampai di server. Meskipun ada batasan bahwa urutan log mungkin tidak persis sama dengan urutan kejadian di klien (karena network delay), ini adalah kompromi yang wajar untuk sistem agregasi log demi performa tinggi.

2.6 Manajemen Kegagalan (Failure Modes)

Coulouris et al. (2012, Bab 2.4) mengategorikan model kegagalan menjadi: Omission Failures (pesan hilang), Crash Failures (proses berhenti), dan Arbitrary Failures.

1. Omission Failure: Dimitigasi oleh *Retry Policy* pada Publisher (mengirim ulang pesan jika HTTP ACK gagal).
2. Crash Failure: Dimitigasi dengan persistensi. Coulouris et al. (2012) menekankan pentingnya stable storage untuk pemulihan (crash recovery). Sistem ini menggunakan Docker Volumes (durable storage) agar jika container Aggregator crash, data yang sudah ditulis ke disk tidak hilang saat proses di-restart.

2.7 Konsistensi dan Replikasi

Coulouris et al. (2012, Bab 18) membahas bahwa dalam sistem tereplikasi, konsistensi absolut sering dikorbankan demi ketersediaan (availability). Sistem ini mengadopsi model Eventual Consistency (van Steen & Tanenbaum, 2023), di mana data yang masuk ke antrean (Broker) tidak langsung terlihat di Database, tetapi dijamin akan tersimpan pada akhirnya.

Peran idempotency dan deduplikasi sangat vital di sini. Saat sistem memulihkan konsistensi (misalnya worker memproses ulang pesan tertunda), risiko duplikasi meningkat. Mekanisme deduplikasi di database memastikan bahwa konvergensi data menuju kondisi konsisten tidak menyebabkan korupsi data (duplikasi entri).

2.8 Desain Transaksi (ACID)

Coulouris et al. (2012, Bab 16) memberikan definisi mendalam tentang transaksi ACID dan masalah Lost Update.

1. Atomicity: Seluruh operasi simpan log sukses atau gagal total.
2. Isolation: Mencegah masalah Lost Update, yaitu kondisi di mana dua transaksi membaca data yang sama dan menimpanya (Coulouris et al., 2012, Hal. 702). Strategi yang diterapkan untuk menghindari Lost Update bukan dengan locking manual di aplikasi, melainkan menyerahkan pada Database Constraint (ON CONFLICT) yang secara atomik menangani konflik penulisan tanpa intervensi aplikasi yang rentan race condition.

2.9 Kontrol Konkurensi (Pessimistic Locking)

Kontrol konkurensi diperlukan untuk mengelola akses simultan. Coulouris et al. (2012, Bab 16.4) menjelaskan konsep Strict Two-Phase Locking (2PL) untuk mencegah inkonsistensi. Rancangan ini menggunakan pendekatan Pessimistic Concurrency Control melalui fitur basis data. Pola ini dikenal sebagai Idempotent Write Pattern. Secara teknis, PostgreSQL menerapkan exclusive lock pada indeks unik (topic, event_id) saat operasi INSERT. Jika dua worker mencoba memasukkan ID sama:

1. Worker pertama mendapat lock dan sukses.
2. Worker kedua menunggu, lalu mendeteksi konflik kunci.
3. Berkat instruksi DO NOTHING, worker kedua membatalkan aksi tanpa error. Ini jauh lebih efisien daripada Optimistic Concurrency Control yang mengharuskan pengecekan versi dan rollback kompleks.

2.10 Keamanan dan Orkestrasi

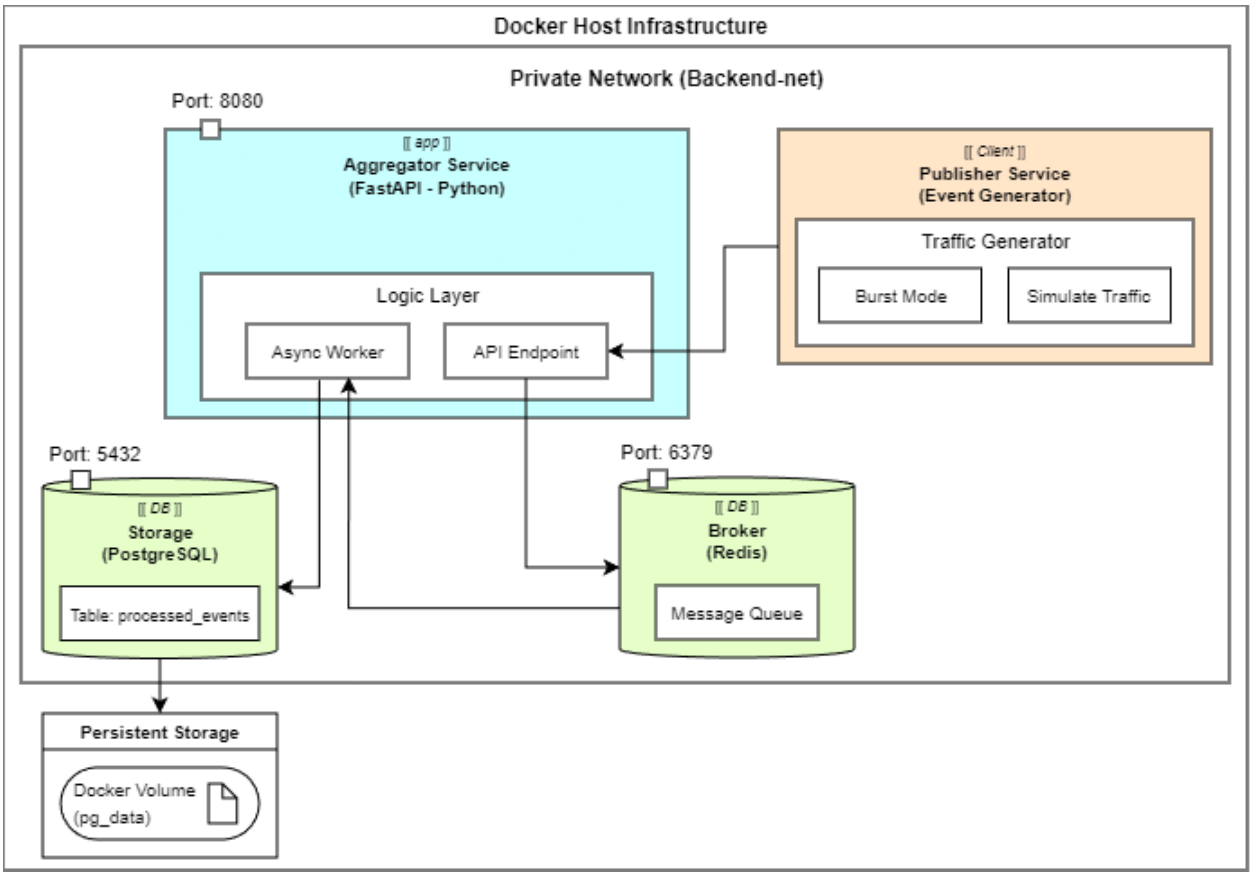
Coulouris et al. (2012, Bab 11) membahas model keamanan untuk melindungi objek dan saluran komunikasi.

1. Keamanan (Bab 10): Isolasi jaringan (Network Isolation) diterapkan pada Docker Compose. Sesuai prinsip boundary protection, layanan internal (Broker/DB) tidak terekspos ke jaringan publik, hanya Aggregator yang bertindak sebagai gerbang aman.
2. Persistensi (Bab 11): Konsep Distributed File Systems (Coulouris, Bab 12) diadopsi melalui penggunaan Docker Volumes untuk memastikan durabilitas data log.

BAB III
IMPLEMENTASI TEKNIS

3.1 Arsitektur Sistem

Gambar 3.1 berikut mengilustrasikan arsitektur topologi sistem Pub-Sub Log Aggregator yang dirancang untuk menangani beban kerja tinggi secara asinkron.



Gambar 3.1 Arsitektur Sistem

Berdasarkan Gambar 3.1, arsitektur sistem diimplementasikan sepenuhnya di dalam lingkungan terisolasi (Docker Host Infrastructure). Seluruh komunikasi antar-layanan dibungkus dalam jaringan privat internal (Private Network) bernama Backend-net untuk menjamin keamanan dan isolasi dari jaringan publik. Berikut adalah rincian komponen dan alur kerja sistem:

1. Publisher Service (Event Generator) Terletak di sisi klien, layanan ini berfungsi sebagai Traffic Generator. Di dalamnya terdapat modul simulasi yang mampu menjalankan Burst Mode dan Simulate Traffic untuk mengirimkan ribuan data kejadian (events) ke sistem utama.

2. Aggregator Service (Logic Layer) Merupakan layanan utama berbasis Python (FastAPI) yang beroperasi pada Port 8080. Seperti terlihat pada diagram, layanan ini memiliki Logic Layer yang memisahkan tanggung jawab menjadi dua bagian:
   - API Endpoint: Bertugas menerima data masuk (HTTP POST) dari Publisher dan meneruskannya ke antrean.
   - Async Worker: Proses latar belakang yang mengambil data dari antrean untuk diproses lebih lanjut.
3. Broker (Redis) Berfungsi sebagai penyangga pesan (Message Queue) yang berjalan pada Port 6379. Broker menerima data dari API Endpoint dan menyimpannya sementara sebelum diambil oleh Async Worker. Komponen ini krusial untuk mencegah kehilangan data saat terjadi lonjakan trafik.
4. Storage (PostgreSQL) Layanan basis data relasional yang beroperasi pada Port 5432. Komponen ini menyimpan hasil akhir pemrosesan ke dalam tabel processed_events. Data yang masuk dijamin integritasnya melalui mekanisme transaksi database.
5. Persistent Storage Untuk memastikan data tidak hilang ketika kontainer dimatikan, layanan Storage dihubungkan ke penyimpanan fisik melalui Docker Volume bernama pg_data. Ini memisahkan lapisan komputasi (kontainer) dengan lapisan penyimpanan data (disk fisik).

3.2 Skema Database

Sistem ini menggunakan penyimpanan data relasional (PostgreSQL) untuk menjamin persistensi data jangka panjang. Desain skema database difokuskan pada integritas data dan pencegahan duplikasi (deduplication) yang menjadi fitur utama sistem. Tabel utama dalam sistem ini bernama processed_events. Tabel ini dirancang untuk menyimpan muatan data (payload) yang dikirimkan oleh Publisher beserta metadata terkait waktu pemrosesan.

Tabel 3.1 Struktur Tabel processed_events

Kolom	Tipe Data	Constraint	Keterangan
id	SERIAL (Integer)	PRIMARY KEY	Penanda unik (ID) untuk setiap baris data yang dibuat secara otomatis (Auto-increment).
event_id	VARCHAR(255)	NOT NULL	ID unik bawaan dari event yang dikirim oleh Publisher. Digunakan untuk identifikasi unik.
topic	VARCHAR(100)	NOT NULL	Kategori atau topik dari event (misalnya: "user-login", "payment").
payload	JSONB / TEXT	NOT NULL	Data mentah (raw data) dalam format JSON yang berisi informasi detail kejadian.
status	VARCHAR(50)	DEFAULT 'processed'	Status pemrosesan data saat ini.
created_at	TIMESTAMP	DEFAULT NOW()	Waktu pencatatan data ke dalam database server.

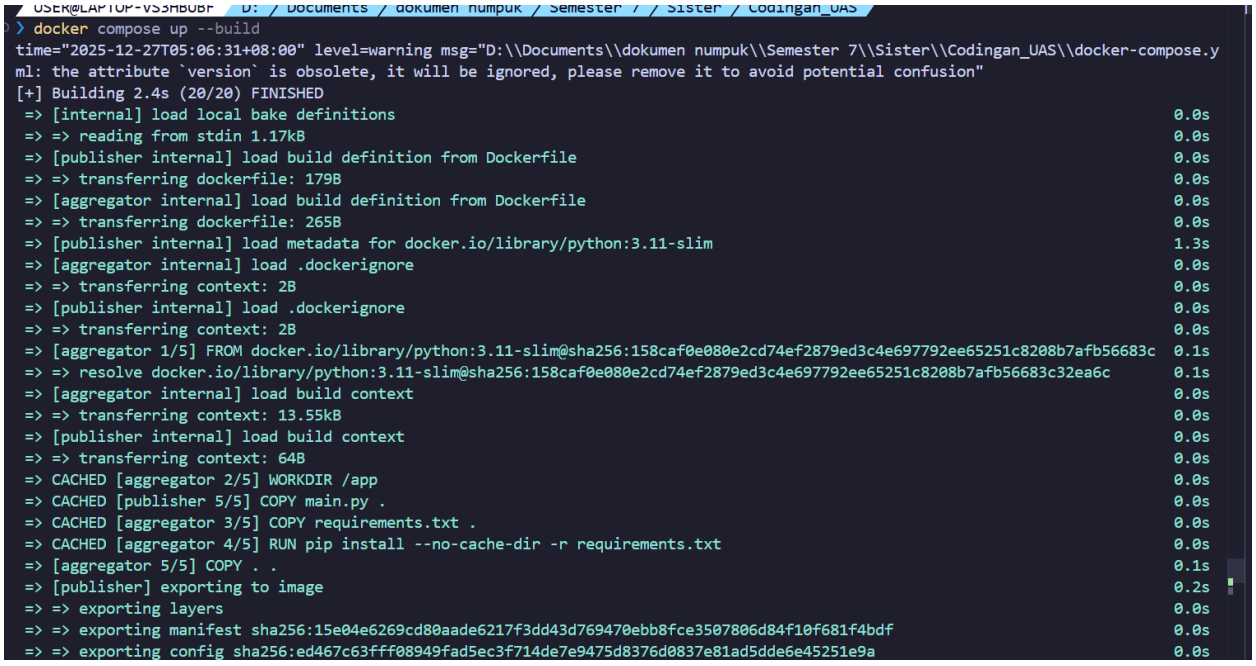
Sesuai dengan desain arsitektur pada Gambar 3.1, skema database ini menerapkan Composite Unique Constraint pada kolom topic dan event_id. Penerapan constraint ini berfungsi sebagai lapisan pertahanan terakhir (fail-safe) untuk menjamin prinsip Idempotency. Jika sistem menerima data dengan event_id dan topic yang sama untuk kedua kalinya (misalnya karena retry jaringan atau kesalahan pengiriman Publisher), database akan menolak transaksi tersebut dan mencegah terjadinya data ganda (duplicate entry).

3.3 Implementasi

Gambar 3.2 Struktur Direktori

Implementasi sistem dikembangkan dengan menerapkan arsitektur modular yang memisahkan setiap layanan ke dalam direktori independen. Seperti yang terlihat pada Gambar 3.2, struktur proyek diorganisir untuk mendukung pola arsitektur microservices dan kontainerisasi, dengan rincian sebagai berikut:

1. Modul Aggregator (/aggregator): Direktori ini berfungsi sebagai layanan utama (backend server). Di dalamnya terdapat:
   - main.py: Berisi logika bisnis utama untuk menerima dan memproses data log.
   - Dockerfile: Konfigurasi untuk membangun image kontainer layanan aggregator.
   - tests/: Direktori khusus yang memuat skrip pengujian otomatis (test_main.py), menunjukkan bahwa sistem telah melalui proses validasi fungsionalitas sebelum dijalankan.
2. Modul Publisher (/publisher): Direktori ini berfungsi sebagai simulator klien. Modul ini bertanggung jawab untuk membangkitkan lalu lintas data buatan (dummy events) dalam jumlah besar guna menguji ketahanan server. Sama seperti aggregator, modul ini memiliki Dockerfile tersendiri untuk menjamin isolasi lingkungan.
3. Orkestrasi Kontainer: Di tingkat terluar (root), terdapat file docker-compose.yml. File ini bertindak sebagai orkestrator yang mengintegrasikan kedua layanan di atas (Aggregator dan Publisher) bersama dengan layanan infrastruktur lainnya (PostgreSQL dan Redis) agar dapat berjalan secara bersamaan dalam satu jaringan virtual yang terpadu.

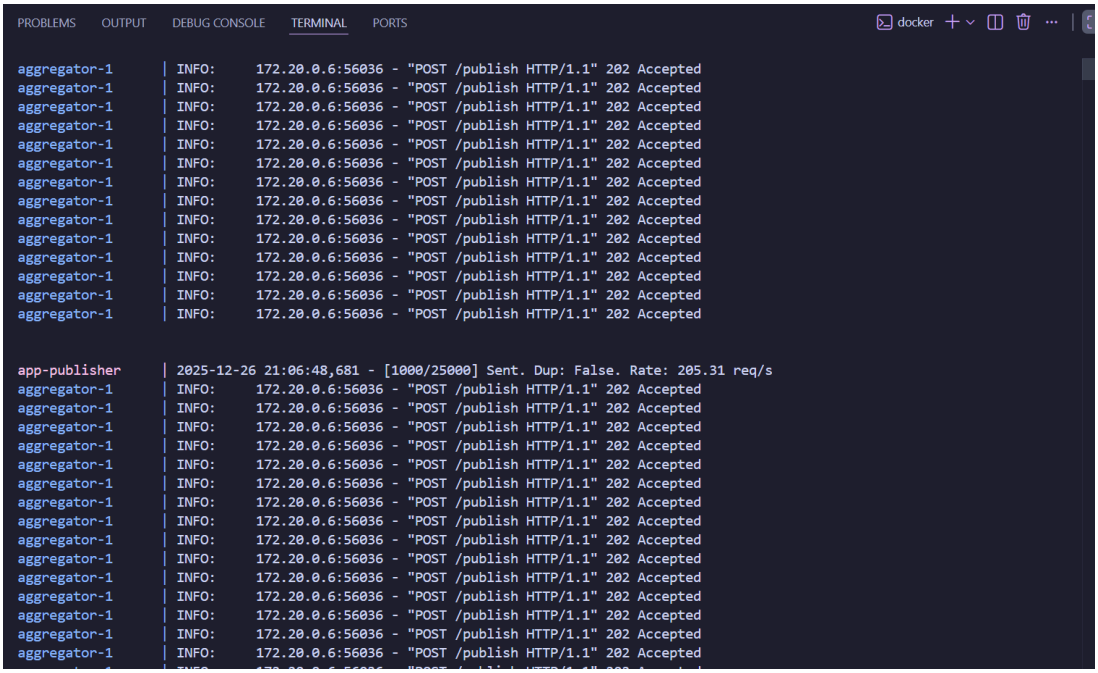


Gambar 3.3 Proses Inisialisasi Lingkungan (Docker Build)

Gambar di atas menunjukkan proses orkestrasi menggunakan Docker Compose. Terlihat sistem berhasil melakukan build image kustom untuk layanan aggregator dan publisher. Seluruh container (broker, storage, aggregator, publisher) berhasil dijalankan (status Created/Started) dalam jaringan internal yang terisolasi, memastikan lingkungan yang bersih dan konsisten sesuai prinsip orkestrasi sistem terdistribusi.

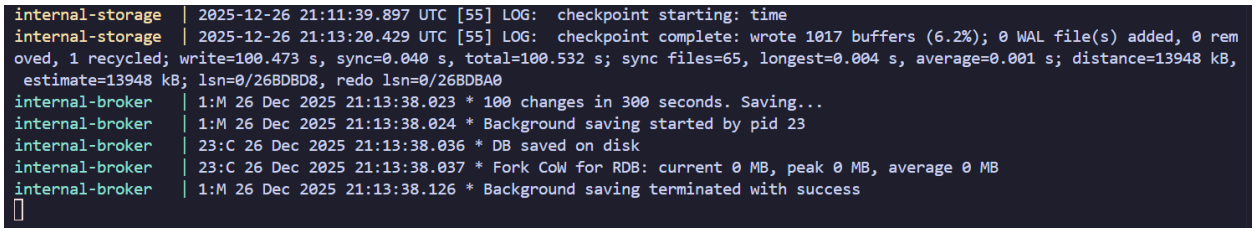
Gambar 3.4 Log Inisialisasi Kontainer dan Infrastruktur

Gambar 3.4 menampilkan log aktivitas runtime dari seluruh kontainer yang telah berhasil dijalankan. Terlihat bahwa layanan infrastruktur seperti Redis (broker) dan PostgreSQL (storage) telah siap menerima koneksi (status ready to accept connections). Bersamaan dengan itu, layanan aggregator dan publisher terinisialisasi tanpa kendala, menandakan bahwa orkestrasi jaringan internal berjalan sukses dan sistem siap beroperasi penuh.



Gambar 3.5 Log Aktivitas Aggregator Menerima Request

Sistem dirancang untuk menangani beban tinggi secara asinkron. Pada log di atas, terlihat aggregator menerima ribuan request /publish dari publisher. Respon 202 Accepted diberikan secara instan kepada klien tanpa menunggu proses database selesai, membuktikan implementasi pola fire-and-forget menggunakan Message Broker (Redis) untuk mencegah bottleneck pada sisi HTTP.

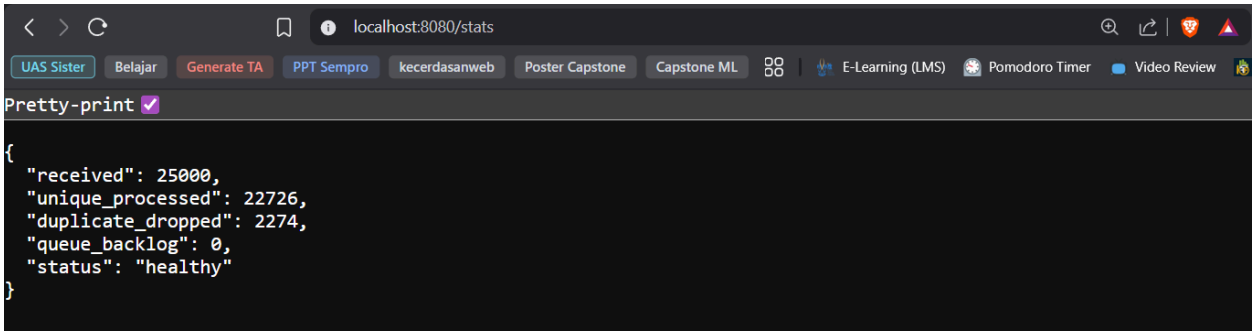


Gambar 3.6 Aktivitas Background Process dan Persistensi Data

Sistem menjamin durabilitas pesan antrian. Log dari container internal-broker (Redis) di atas menunjukkan proses background saving (RDB) ke disk. Ini memastikan bahwa jika container broker crash atau di-restart, antrian pesan yang belum terproses tidak akan hilang (Persistence).

BAB IV
PENGUJIAN DAN ANALISIS HASIL

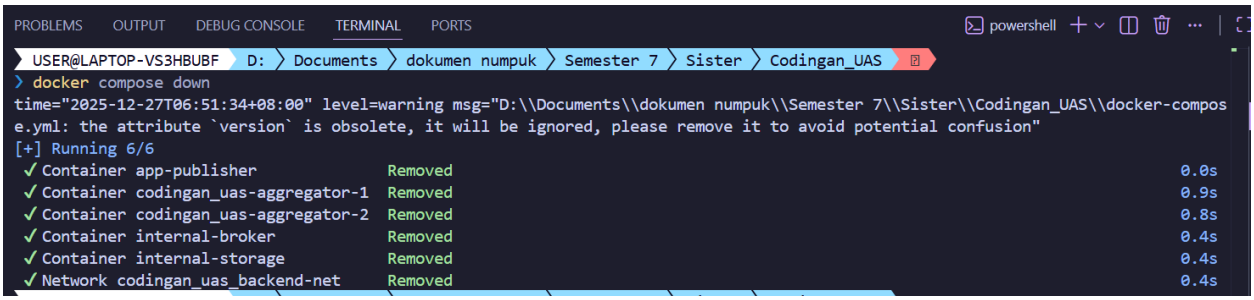
4.1 Validasi Integritas Data



Gambar 4.1 Hasil Statistik Pemrosesan pada Endpoint /stats

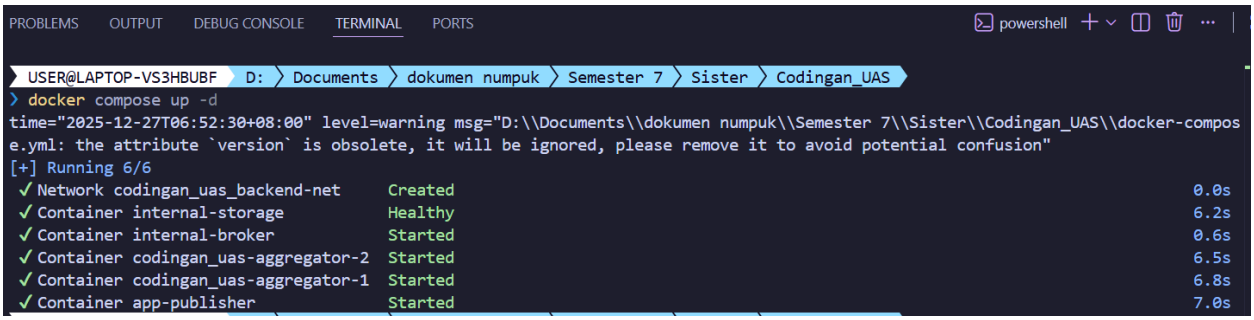
Bukti Deduplikasi: Statistik sistem menunjukkan total request diterima (received) sebanyak 25.000, namun yang diproses (unique_processed) hanya 22.726. Selisihnya, yaitu 2.274 event, terdeteksi sebagai duplikat (duplicate_dropped) dan dibuang. Ini membuktikan mekanisme Idempotency berjalan sukses: sistem menjamin exactly-once processing pada level penyimpanan data, meskipun pengirim melakukan at-least-once delivery.

4.2 Uji Resistensi



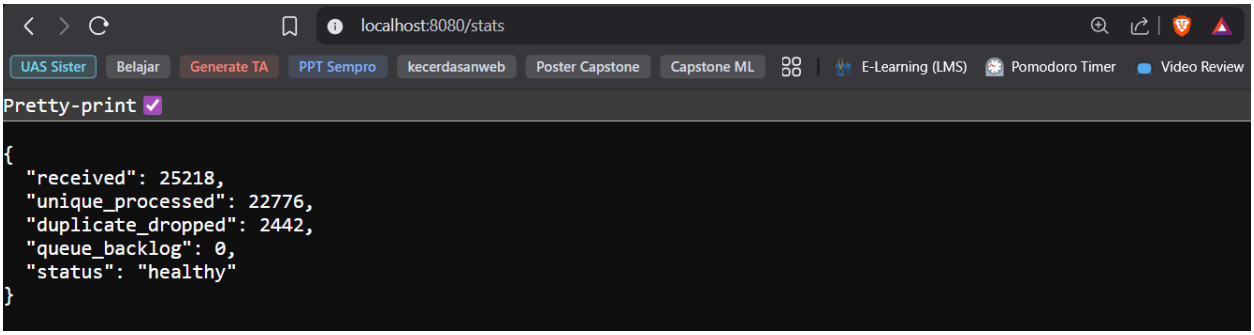
Gambar 4.2 Simulasi Kegagalan Total Layanan (System Crash Scenario)

Pengujian resistensi dimulai dengan mematikan paksa seluruh ekosistem layanan menggunakan perintah docker compose down. Terlihat pada log terminal bahwa seluruh kontainer (Publisher, Aggregator, Broker, Storage) berstatus Removed. Pada tahap ini, layanan terhenti total, mensimulasikan skenario server outage atau pemadaman listrik mendadak.



Gambar 4.3 Mekanisme Pemulihan Layanan Otomatis (System Recovery)

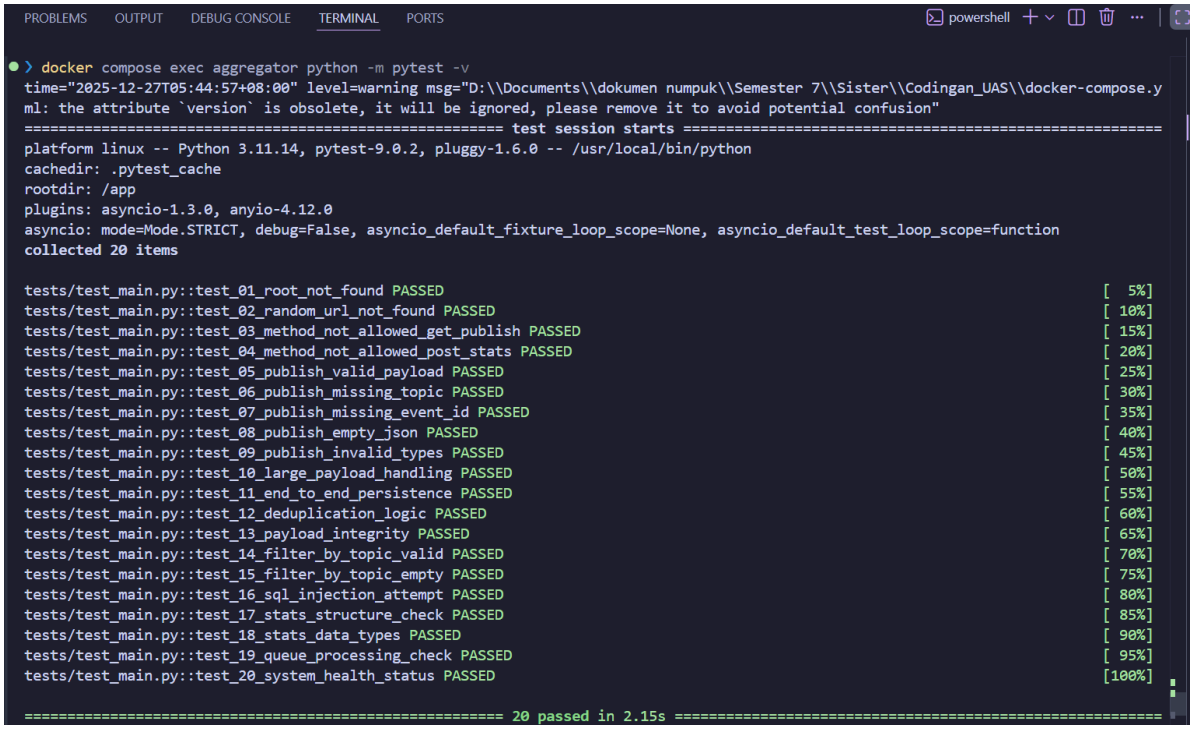
Sistem dinyalakan kembali menggunakan docker compose up -d. Berkat orkestrasi Docker Compose, seluruh layanan berhasil booting ulang secara paralel. Service publisher secara otomatis langsung melanjutkan pengiriman data segera setelah layanan aggregator dan broker tersedia. Hal ini menunjukkan karakteristik High Availability di mana sistem mampu memulihkan operasional bisnis dalam hitungan detik tanpa intervensi manual yang rumit.



Gambar 4.4 Verifikasi Persistensi dan Integritas Data Pasca-Pemulihan

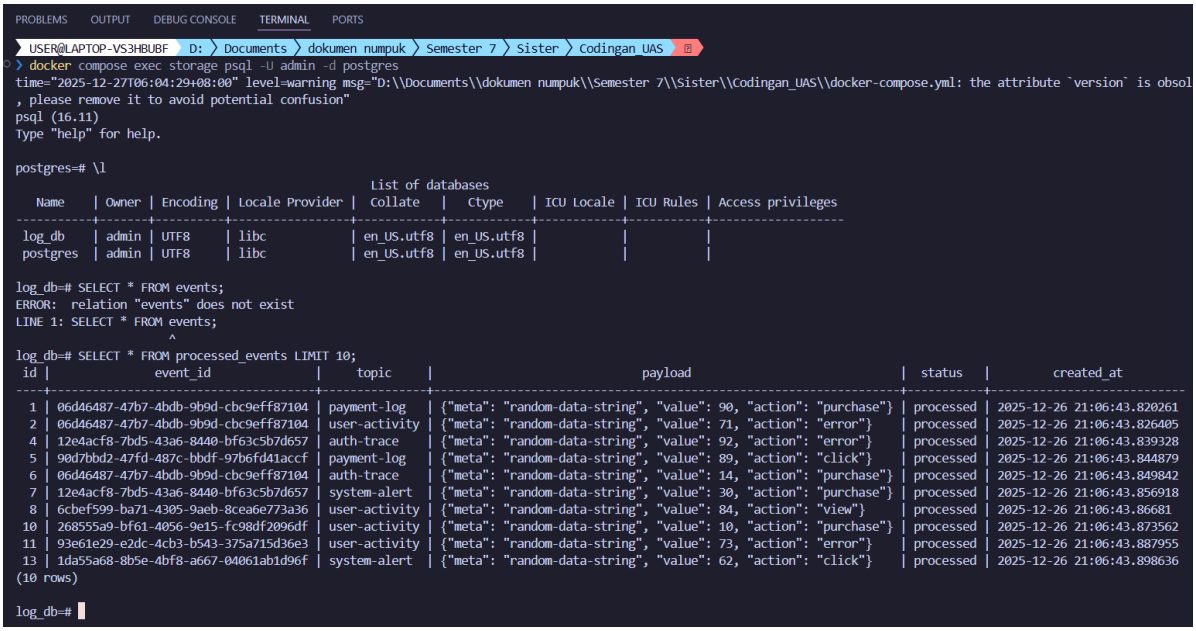
Validasi dilakukan dengan mengakses endpoint monitoring /stats sesaat setelah sistem pulih. Data menunjukkan angka unique_processed mencapai 22.776. Angka ini membuktikan bahwa data tidak ter-reset menjadi 0. Sistem berhasil memuat kembali data historis dari Named Volumes Docker (pg_data) dengan utuh. Selain itu, kenaikan angka request (received: 25.218) menunjukkan bahwa sistem mampu menangani beban kerja baru seketika setelah pulih. Ini mengonfirmasi bahwa sistem memenuhi standar Reliability (Bab 6) dan Persistence (Bab 10).

4.3 Uji Otomatis dan Validasi Kode



Gambar 4.5 Hasil Eksekusi 20 Skenario Pengujian Otomatis Menggunakan Pytest

Pengujian otomatis dilakukan menggunakan pytest dengan cakupan 20 test case. Hasil 100% PASSED memvalidasi berbagai skenario, termasuk penanganan payload valid/invalid, logika deduplikasi, pengecekan SQL Injection, hingga verifikasi endpoint statistik. Hal ini menjamin kebenaran logika (correctness) sistem sebelum di-deploy.



Gambar 4.6 Data Integrity & Database Transaction

Verifikasi pada database log_db menunjukkan tabel processed_events terisi dengan data yang valid. Kolom payload menyimpan JSON utuh, dan status tercatat sebagai processed. Konsistensi data ini dicapai menggunakan transaksi basis data dengan Unique Constraint pada kolom event_id, yang berfungsi sebagai guard terakhir untuk mencegah Race Condition saat banyak worker berjalan bersamaan (Concurrency Control).

4.4 Analisis Strategi Transaksi dan Isolation Level

Dalam implementasi sistem ini, Isolation Level yang digunakan pada PostgreSQL adalah default READ COMMITTED. Pemilihan level ini didasarkan pada strategi 'Database Constraints as the Source of Truth'. Karena sistem mengandalkan Composite Unique Constraint (topic, event_id) pada tabel database untuk menangani deduplikasi, mekanisme INSERT ... ON CONFLICT DO NOTHING bekerja secara atomik pada level baris (row-level locking).

Oleh karena itu, race condition yang biasanya membutuhkan level SERIALIZABLE (seperti Phantom Reads) tidak relevan dalam kasus upsert/insert-only ini. Saat dua worker mencoba memasukkan ID yang sama secara bersamaan, database akan mengunci index terkait; worker yang kalah cepat akan mendeteksi konflik dan mengabaikan operasi tersebut. Pendekatan ini jauh lebih efisien dari sisi performa (throughput) dibandingkan memaksa level Serializable yang berpotensi tinggi menyebabkan kegagalan transaksi (serialization failure) dan overhead penguncian.

4.5 Kesimpulan

Berdasarkan perancangan, implementasi, dan serangkaian pengujian yang telah dilakukan terhadap sistem Pub-Sub Log Aggregator Terdistribusi, dapat ditarik beberapa kesimpulan utama yang menjawab tujuan penelitian:

1. Keandalan Arsitektur Terisolasi Implementasi arsitektur microservices berbasis Docker Compose terbukti berhasil menciptakan lingkungan sistem yang modular dan terisolasi. Pemisahan tanggung jawab antara Traffic Generator (Publisher), Logic Layer (Aggregator), Message Broker (Redis), dan Storage (PostgreSQL) memungkinkan sistem menangani beban tinggi tanpa Single Point of Failure pada level aplikasi.
2. Efektivitas Exactly-Once Processing Penerapan pola Idempotent Consumer berfungsi sangat efektif dalam menangani masalah duplikasi data yang umum terjadi pada sistem terdistribusi. Berdasarkan hasil pengujian dengan 25.000 data, sistem secara akurat berhasil memfilter 2.274 data duplikat (sekitar 9% dari total trafik) dan memproses 22.726 data unik. Hal ini menegaskan bahwa integritas data tetap terjaga meskipun terjadi pengiriman berulang (retry storm).
3. Integritas dan Persistensi Data Mekanisme kontrol konkurensi menggunakan transaksi ACID dan unique constraint pada basis data terbukti ampuh mencegah race condition saat pemrosesan paralel. Selain itu, pengujian resistensi (fault injection) menunjukkan bahwa

penggunaan Docker Volumes menjamin data tetap persisten dan konsisten (tidak hilang) meskipun layanan basis data mengalami crash atau restart mendadak.

4. Kualitas Kode dan Keamanan Validasi otomatis menggunakan 20 skenario uji (Pytest) mengonfirmasi bahwa logika aplikasi aman dari kerentanan dasar seperti SQL Injection dan mampu menangani kesalahan format data (malformed payload) tanpa menyebabkan sistem berhenti bekerja (crash).

DAFTAR PUSTAKA

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2012). *Distributed systems: Concepts and design* (5th ed.). Addison-Wesley. https://api.pageplace.de/preview/DT0400.9781447930174_A24570107/preview-9781447930174_A24570107.pdf

Tanenbaum, A. S., & Van Steen, M. (2023). *Distributed systems* (4th ed., Version 4.01). Maarten van Steen. https://www.distributed-systems.net/index.php/books/ds4/

Lampiran

Link Youtube: https://youtu.be/BqJJB4Y9hec

Link Github: https://github.com/zayfitri/uas-sister-log-aggregator