Laporan UAS

Sistem Paralel dan Terdistribusi B

Pub Sub Log Aggregator V2

Disusun Oleh :

Dion Prayoga 11221058

8 Desember 2025

Daftar Isi

1. Bagian Teori

T1 (Bab 1): Karakteristik Sistem Terdistribusi dan Trade-off Desain Pub-Sub Aggregator

Karakteristik Sistem Terdistribusi Menurut Van Steen dan Tanenbaum (2023), sistem terdistribusi dicirikan oleh kemampuannya untuk tampil sebagai satu sistem koheren bagi pengguna, meskipun terdiri dari kumpulan komputer independen (Bab 1). Karakteristik utama yang terlihat dalam proyek *Log Aggregator* ini adalah **transparansi**, **keterbukaan (*openness*)**, dan **toleransi kesalahan (*fault tolerance*)**. Melalui penggunaan Docker Compose, sistem ini memisahkan peran antara *Publisher* (penghasil data), *Aggregator* (pemroses), dan *Storage* (PostgreSQL). Hal ini mencerminkan karakteristik **skalabilitas** (Bab 1), di mana kita dapat meningkatkan jumlah *worker* (MIN_CONSUMERS) secara independen untuk menangani beban kerja yang lebih besar tanpa mengubah struktur kode inti. Selain itu, sistem ini bersifat **heterogen**, di mana komponen yang berbeda dapat berjalan di kontainer terpisah namun tetap berkomunikasi secara mulus melalui jaringan virtual pubsub_net.

Trade-off Desain pada Pub-Sub Aggregator Dalam pengembangan *Log Aggregator* berbasis arsitektur *Publish-Subscribe* ini, terdapat beberapa *trade-off* (pertukaran nilai) desain yang krusial:

1. **Consistency vs. Performance (CAP Theorem):** Berdasarkan prinsip pada Bab 8, desain ini memprioritaskan konsistensi data menggunakan kunci komposit (event_id, topic) di PostgreSQL untuk mencegah duplikasi. *Trade-off*-nya adalah adanya latensi tambahan pada setiap operasi penulisan karena database harus melakukan pengecekan indeks secara atomik sebelum melakukan *commit* (Bab 9).
2. **Reliability vs. Complexity:** Menggunakan asyncio.Queue sebagai *buffer* memori meningkatkan kecepatan terima data (*throughput*), namun jika kontainer *aggregator* mengalami *crash*, data dalam antrean yang belum di-*persist* akan hilang. Ini adalah pilihan sadar untuk meningkatkan performa komunikasi asinkron (Bab 4) dengan risiko kehilangan data transien jika terjadi kegagalan sistem sebelum penulisan ke *Storage* selesai.
3. **Decoupling vs. Debugging:** Arsitektur *Pub-Sub* memberikan keuntungan berupa pemisahan temporal dan referensial antar komponen (Bab 2). Namun, hal ini meningkatkan kesulitan dalam melacak aliran data secara *end-to-end* dibandingkan model *Client-Server* tradisional yang bersifat sinkron.

T2 (Bab 2): Kapan memilih arsitektur publish–subscribe dibanding client–server? Alasan teknis.

Pemilihan gaya arsitektur sangat menentukan fleksibilitas dan skalabilitas sistem. Menurut Van Steen dan Tanenbaum (2023), perbedaan fundamental antara arsitektur *Client–Server* dan *Publish–Subscribe* terletak pada derajat pemisahan (*decoupling*) antar proses (Bab 2). Arsitektur *Client–Server* tradisional bersifat sinkron, di mana klien dan server harus saling

mengenal (referensial) dan aktif pada saat yang sama (temporal) agar komunikasi dapat terjalin (Van Steen & Tanenbaum, 2023).

Dalam proyek *Log Aggregator* ini, arsitektur *Publish–Subscribe* dipilih karena beberapa alasan teknis yang krusial untuk menangani beban kerja log yang tinggi:

1. **Pemisahan Referensial (*Referential Decoupling*):** Layanan *Publisher* (publisher_sender.py) tidak perlu mengetahui alamat fisik atau identitas spesifik dari setiap *worker* yang memproses data. *Publisher* cukup mengirimkan data ke perantara (*Aggregator*). Hal ini memungkinkan sistem untuk menambah atau mengurangi jumlah *worker* (MIN_CONSUMERS) secara dinamis tanpa perlu melakukan konfigurasi ulang pada sisi pengirim (Bab 2).
2. **Pemisahan Temporal (*Temporal Decoupling*):** Berdasarkan prinsip komunikasi asinkron pada Bab 4, penggunaan asyncio.Queue sebagai *buffer* internal memungkinkan sistem tetap menerima log dari *Publisher* meskipun database PostgreSQL sedang mengalami lonjakan beban atau latensi tinggi. Dalam model *Client–Server* murni, pengirim akan terblokir hingga penulisan ke database selesai, namun dalam rancangan ini, *Publisher* langsung menerima status *Accepted* (202), sementara pemrosesan dilakukan secara asinkron oleh *worker* di latar belakang.
3. **Skalabilitas Multicast:** Arsitektur ini memfasilitasi diseminasi data ke banyak penerima dengan efisien (Bab 4). Jika di masa mendatang diperlukan komponen baru (seperti sistem peringatan atau dasbor pemantauan real-time), komponen tersebut dapat berlangganan pada topik yang sama tanpa menambah beban komputasi pada *Publisher* asli.

T3 (Bab 3): At-least-once vs exactly-once delivery; peran idempotent consumer.

Menjamin pengiriman pesan yang andal di tengah kegagalan jaringan merupakan tantangan fundamental. Menurut Van Steen dan Tanenbaum (2023), terdapat perbedaan mendasar antara semantik pengiriman pesan, khususnya antara *At-Least-Once* dan *Exactly-Once* (Bab 4). Semantik *At-Least-Once* menjamin bahwa sebuah pesan akan terkirim setidaknya satu kali ke penerima. Jika pengirim tidak menerima konfirmasi (*acknowledgment*), pesan akan dikirim ulang secara terus-menerus. Hal ini memicu risiko terjadinya duplikasi pesan jika konfirmasi tersebut sebenarnya hanya terlambat atau hilang di jaringan (Van Steen & Tanenbaum, 2023).

Sebaliknya, semantik *Exactly-Once* menjamin bahwa pesan diproses tepat satu kali. Namun, mencapai semantik ini secara murni di lapisan komunikasi sangat sulit dan mahal (Bab 8). Oleh karena itu, solusi praktis yang sering diterapkan adalah mengombinasikan mekanisme pengiriman *At-Least-Once* dengan strategi pemrosesan **Idempotent Consumer** di sisi penerima (Van Steen & Tanenbaum, 2023). Konsumen yang idempoten memiliki properti di mana hasil akhir tetap sama meskipun sebuah operasi dijalankan berulang kali dengan input yang identik.

Dalam proyek *Log Aggregator* ini, prinsip tersebut diterapkan melalui beberapa mekanisme teknis:

1. **Mekanisme Retransmisi:** Layanan *Publisher* (publisher_sender.py) mensimulasikan kegagalan pengiriman melalui parameter DUPLICATE_RATE. Hal ini memaksa sistem untuk menangani skenario *At-Least-Once Delivery* di mana ID event yang sama dikirim berulang kali.
2. **Durable Deduplication Store:** Berdasarkan konsep proteksi pada Bab 8, *Aggregator* bertindak sebagai konsumen idempoten dengan memanfaatkan penyimpanan persisten PostgreSQL. Melalui perintah ON CONFLICT (event_id, topic) DO NOTHING dalam fungsi _process_event_atomic, sistem secara otomatis mendeteksi dan membuang duplikat tanpa merusak integritas data yang sudah ada.
3. **Atomisitas Statistik:** Penggunaan transaksi database (Bab 9) memastikan bahwa pembaruan metrik unik (unique_processed) dan duplikat (duplicate_dropped) dilakukan secara atomik. Hal ini menjamin konsistensi meskipun terdapat banyak *worker* konkuren yang mencoba memproses pesan yang sama.

T4 (Bab 4): Skema penamaan topic dan event_id (unik, collision-resistant) untuk dedup.

Identifikasi entitas secara unik merupakan prasyarat utama untuk menjamin konsistensi data dan fungsionalitas sistem. Menurut Van Steen dan Tanenbaum (2023), penamaan (*naming*) berfungsi untuk merujuk pada entitas tanpa harus bergantung pada lokasi fisiknya (Bab 6). Dalam proyek *Log Aggregator* ini, skema penamaan diimplementasikan melalui dua komponen utama: topic dan event_id.

1. Skema Penamaan Topic Penamaan topik dalam proyek ini mengadopsi prinsip *Structured Naming* yang memungkinkan pengelompokan log secara hierarkis (Bab 6). Contohnya, penggunaan nama seperti app.log.level-1 memungkinkan sistem untuk melakukan pemfilteran data berdasarkan atribut tertentu. Secara teknis, topic digunakan sebagai bagian dari kunci komposit (*composite key*) di lapisan penyimpanan untuk memastikan bahwa ruang lingkup keunikan sebuah pesan terikat pada kategori tertentu (Van Steen & Tanenbaum, 2023).

2. Event_ID dan Karakteristik Collision-Resistant Untuk mencegah terjadinya tabrakan identitas (*collision*), proyek ini menggunakan UUID (*Universally Unique Identifier*) melalui fungsi uuid.uuid4() pada sisi *Publisher*. Berdasarkan teori pada Bab 6, UUID dirancang untuk memiliki keunikan global yang sangat tinggi bahkan tanpa adanya otoritas pusat yang mengaturnya. Penggunaan event_id yang *collision-resistant* sangat krusial dalam mekanisme deduplikasi (Bab 8). Tanpa ID yang benar-benar unik, sistem tidak dapat membedakan antara transmisi ulang dari pesan yang sama (*duplicate*) dengan pesan baru yang kebetulan memiliki isi yang identik.

Kaitan dengan Mekanisme Deduplikasi dalam Proyek Identitas unik ini menjadi fondasi bagi strategi *Durable Deduplication Store*. Dalam kode main.py, keunikan dijamin secara atomik melalui skema database PostgreSQL:

- **Kunci Utama Komposit:** Definisi PRIMARY KEY (event_id, topic) pada tabel processed_events memastikan integritas data pada level terendah.
- **Idempotensi Penulisan:** Dengan memanfaatkan karakteristik unik dari event_id, perintah ON CONFLICT ... DO NOTHING secara otomatis menolak entri yang sudah ada. Hal ini memungkinkan sistem untuk tetap konsisten (Bab 7) meskipun menghadapi kondisi *race condition* dari berbagai *worker* asinkron yang memproses log secara konkuren (Bab 9).

T5 (Bab 5): Ordering praktis (timestamp + monotonic counter); batasan dan dampaknya.

Pengurutan kejadian (*event ordering*) merupakan tantangan yang kompleks karena tidak adanya jam global yang tersinkronisasi secara sempurna di antara berbagai mesin. Menurut Van Steen dan Tanenbaum (2023), urutan kejadian dapat ditentukan melalui jam fisik maupun jam logis untuk menjaga konsistensi urutan operasi (Bab 5). Dalam proyek *Log Aggregator* ini, pendekatan *ordering* praktis diterapkan melalui kombinasi **Timestamp ISO8601** dan sifat **Monotonic Counter** pada urutan pemrosesan.

1. Mekanisme Timestamp dan Monotonic Counter Setiap *event* dalam proyek ini dibekali dengan atribut timestamp yang dihasilkan oleh *Publisher* menggunakan format ISO8601 dengan zona waktu UTC (Bab 5). Penggunaan UTC sangat krusial untuk meminimalkan ambiguitas waktu antar sistem yang berbeda geografis (Van Steen & Tanenbaum, 2023). Selain itu, sistem secara implisit menggunakan mekanisme *monotonic counter* melalui urutan penyisipan pada database PostgreSQL dan pengelolaan antrean pada asyncio.Queue. Di lapisan penyimpanan, statistik seperti received dan unique_processed pada tabel aggregator_stats diperbarui secara meningkat (*monotonic*), yang mencerminkan urutan logis kedatangan data di sistem (Bab 5).

2. Batasan dan Dampak Teknis Meskipun kombinasi *timestamp* dan *monotonic counter* efektif untuk penggunaan praktis, terdapat beberapa batasan yang diidentifikasi berdasarkan teori Bab 5 dan Bab 8:

- **Clock Skew:** Batasan utama adalah adanya *clock skew* atau perbedaan waktu antar mesin *Publisher*. Jika satu mesin memiliki jam yang lebih lambat, *event* yang terjadi kemudian secara fisik mungkin memiliki *timestamp* yang lebih awal. Dampaknya, urutan logis pada *Aggregator* mungkin tidak mencerminkan urutan fisik kejadian di dunia nyata (Van Steen & Tanenbaum, 2023).
- **Asynchrony:** Karena proyek ini menggunakan antrean asinkron dan multi-worker (Bab 3), *event* yang tiba hampir bersamaan mungkin diproses oleh *worker* yang berbeda dengan kecepatan yang berbeda. Hal ini berdampak pada urutan penulisan di tabel processed_events yang mungkin sedikit berbeda dari urutan kedatangan di HTTP *endpoint*.
- **Idempotency vs. Reordering:** Mekanisme deduplikasi berbasis event_id menjamin keunikan data, namun tidak menjamin pengurutan ulang jika terjadi *retry* pengiriman (Bab 8).

T6 (Bab 6): Failure modes dan mitigasi (retry, backoff, durable dedup store, crash recovery).

Kegagalan adalah hal yang tidak terelakkan dan dapat terjadi pada berbagai tingkatan. Menurut Van Steen dan Tanenbaum (2023), kegagalan dapat diklasifikasikan ke dalam beberapa model, seperti *crash failures* (berhenti beroperasi), *omission failures* (gagal mengirim/menerima pesan), dan *arbitrary failures* (perilaku yang menyimpang) (Bab 8). Dalam proyek *Log Aggregator* ini, berbagai strategi mitigasi diterapkan untuk menangani model kegagalan tersebut guna memastikan integritas data tetap terjaga.

1. Mekanisme Retry dan Backoff Untuk menangani *omission failures* di lapisan jaringan, layanan *Publisher* (publisher_sender.py) mengimplementasikan logika pengiriman ulang (*retry*). Berdasarkan prinsip pada Bab 4, ketika sebuah permintaan HTTP mengalami kegagalan koneksi atau *timeout*, pengirim melakukan upaya pengiriman kembali. Strategi ini dikombinasikan dengan jeda waktu (asyncio.sleep) untuk menghindari beban berlebih pada *Aggregator* yang sedang pulih, yang secara konseptual merupakan bentuk sederhana dari *exponential backoff* (Van Steen & Tanenbaum, 2023).

2. Durable Deduplication Store Inti dari ketahanan data dalam proyek ini terletak pada penggunaan PostgreSQL sebagai *Durable Dedup Store*. Berbeda dengan penyimpanan berbasis memori yang volatil, penyimpanan persisten menjamin bahwa *state* mengenai *event* yang telah diproses tidak akan hilang meskipun terjadi pemadaman listrik atau *crash* pada kontainer (Bab 7). Penggunaan perintah ON CONFLICT DO NOTHING memastikan bahwa pemulihan data dari kegagalan transmisi tidak akan mengakibatkan inkonsistensi data ganda.

3. Crash Recovery dan Persistensi Mekanisme *crash recovery* didukung oleh orkestrasi Docker Compose. Menurut teori pada Bab 3, penggunaan kontainer memudahkan isolasi kegagalan. Dalam proyek ini, kebijakan restart: always pada layanan *storage* dan *aggregator* memastikan sistem otomatis melakukan *reboot* setelah kegagalan. Namun, pemulihan sejati hanya terjadi karena adanya *Docker Volumes* (pg_data). Berdasarkan prinsip Bab 8, volume ini bertindak sebagai *stable storage* yang menyimpan catatan transaksi, sehingga saat *Aggregator* hidup kembali, ia dapat segera melanjutkan tugasnya dengan *state* yang konsisten dari titik terakhir sebelum kegagalan (Van Steen & Tanenbaum, 2023).

T7 (Bab 7): Eventual consistency pada aggregator; peran idempotency + dedup.

Sering kali terdapat kebutuhan untuk menyeimbangkan antara kecepatan respon sistem dan tingkat konsistensi data. Menurut Van Steen dan Tanenbaum (2023), **Eventual Consistency** adalah model konsistensi lemah yang menjamin bahwa jika tidak ada pembaruan lebih lanjut, pada akhirnya semua replika atau salinan data akan menjadi identik (Bab 7). Proyek *Log Aggregator* ini mengadopsi prinsip tersebut guna memastikan efisiensi pemrosesan data log yang datang dalam volume tinggi secara asinkron.

1. Penerapan Eventual Consistency Dalam proyek ini, model *eventual consistency* termanifestasi dalam alur data dari *Publisher* ke *Aggregator* dan berlanjut ke penyimpanan PostgreSQL. Ketika sebuah event diterima melalui *endpoint* /publish, sistem segera mengembalikan status asinkron kepada pengirim tanpa menunggu penulisan data ke disk selesai sepenuhnya. Berdasarkan teori pada Bab 7, ini berarti terdapat jeda waktu (*latency window*) di mana data yang baru masuk mungkin belum terlihat pada *endpoint* /events atau /stats. Namun, sistem menjamin bahwa setelah antrean internal (asyncio.Queue) diproses oleh *worker*, status penyimpanan akan mencapai kondisi konsisten dan mencerminkan seluruh event unik yang masuk (Van Steen & Tanenbaum, 2023).

2. Peran Idempotency dan Deduplikasi Untuk mendukung tercapainya konsistensi yang akurat dalam model ini, mekanisme **Idempotency** dan **Deduplikasi** (Dedup) memegang peranan krusial (Bab 8).

- **Deduplikasi sebagai Penjaga Konsistensi:** Dalam skenario *At-Least-Once Delivery*, duplikasi pesan adalah hal yang tak terelakkan. Dengan menggunakan kunci komposit (event_id, topic), sistem melakukan penyaringan pada lapisan *Durable Store*. Hal ini memastikan bahwa meskipun data diproses berkali-kali di waktu yang berbeda, *state* akhir database tetap konsisten dan tidak terkontaminasi oleh data ganda (Bab 8).
- **Idempotent Consumer:** Berdasarkan prinsip Bab 9, *worker* dalam proyek ini dirancang agar bersifat idempoten. Penggunaan perintah ON CONFLICT DO NOTHING menjamin bahwa operasi penulisan memiliki efek yang sama baik dijalankan satu kali maupun berkali-kali. Ini sangat penting untuk memastikan integritas statistik sistem, di mana metrik unique_processed hanya bertambah untuk data yang benar-benar baru bagi sistem.

T8 (Bab 8): Desain transaksi: ACID, isolation level, dan strategi menghindari lost-update.

Menjaga integritas data saat terjadi operasi konkuren merupakan hal yang sangat krusial. Menurut Van Steen dan Tanenbaum (2023), sebuah sistem harus mendukung properti ACID (*Atomicity, Consistency, Isolation, Durability*) untuk menjamin bahwa transaksi database diproses secara andal (Bab 1 dan Bab 9). Proyek *Log Aggregator* ini menerapkan prinsip transaksi ACID melalui integrasi dengan PostgreSQL untuk mengelola data log dan statistik operasional secara konsisten.

1. Penerapan Properti ACID dan Isolation Level Dalam proyek ini, setiap proses pemrosesan *event* dibungkus dalam sebuah unit kerja atomik melalui fungsi _process_event_atomic. Berdasarkan teori pada Bab 9, **Atomicity** dijamin dengan penggunaan blok try-except yang melakukan conn.commit() hanya jika seluruh perintah SQL berhasil, atau conn.rollback() jika terjadi kegagalan (Van Steen & Tanenbaum, 2023). Proyek ini beroperasi pada *isolation level* default PostgreSQL, yaitu *Read Committed*, yang mencegah sebuah transaksi membaca data yang belum di-*commit* oleh transaksi lain, sehingga menjaga **Isolation**. Properti **Durability** dipastikan melalui penyimpanan persisten

di dalam *Docker Volume* yang menjaga data tetap utuh meskipun kontainer dimatikan (Bab 8).

2. Strategi Menghindari Lost-Update Masalah *lost-update* biasanya terjadi ketika dua *worker* mencoba memperbarui baris data yang sama secara bersamaan, namun pembaruan dari salah satu *worker* menimpa pembaruan lainnya tanpa memperhitungkan nilai terbaru. Dalam proyek ini, strategi mitigasi diimplementasikan sebagai berikut:

- **Atomic Increment:** Pada tabel aggregator_stats, pembaruan nilai dilakukan menggunakan perintah SQL UPDATE ... SET value = value + 1. Menurut Van Steen dan Tanenbaum (2023), teknik *in-place update* ini memastikan database menangani penguncian baris secara internal sehingga setiap kenaikan nilai dihitung secara akurat tanpa ada data yang hilang (Bab 9).
- **Pencegahan Duplikasi Atomik:** Strategi ON CONFLICT DO NOTHING yang dikombinasikan dengan pengecekan is_unique = cursor.fetchone() memastikan bahwa keputusan untuk menambah statistik unique_processed atau duplicate_dropped didasarkan pada hasil operasi yang bersifat *thread-safe* (Bab 7).

T9 (Bab 9): Kontrol konkurensi: locking/unique constraints/upsert; idempotent write pattern.

Dalam sistem terdistribusi yang melibatkan banyak pemroses (*multi-worker*), kontrol konkurensi menjadi sangat krusial untuk mencegah kerusakan data akibat akses simultan. Menurut Van Steen dan Tanenbaum (2023), konkurensi dalam sistem terdistribusi sering kali dikelola melalui mekanisme penguncian (*locking*) atau skema transaksi yang menjamin akses eksklusif terhadap sumber daya bersama (Bab 5 dan Bab 8). Dalam proyek *Log Aggregator* ini, kontrol konkurensi diimplementasikan tanpa menggunakan *explicit distributed locking*, melainkan mengandalkan fitur atomisitas pada lapisan basis data PostgreSQL.

1. Unique Constraints dan Kunci Komposit Implementasi utama kontrol konkurensi dalam proyek ini adalah penggunaan *Unique Constraints* pada tabel processed_events. Berdasarkan prinsip integrasi data pada Bab 7, penggunaan kunci unik merupakan cara paling efektif untuk menjaga konsistensi replika (Van Steen & Tanenbaum, 2023). Dengan menetapkan PRIMARY KEY (event_id, topic), basis data bertindak sebagai penengah (*arbitrator*) saat beberapa *worker* mencoba memasukkan log yang sama secara bersamaan. Jika terjadi tabrakan, basis data secara otomatis menolak permintaan yang datang kemudian berdasarkan batasan unik tersebut.

2. Idempotent Write Pattern dan Upsert Strategi yang digunakan dalam proyek ini mengadopsi pola *Idempotent Write* melalui instruksi ON CONFLICT DO NOTHING. Menurut Van Steen dan Tanenbaum (2023), pola penulisan idempoten memastikan bahwa hasil akhir sistem tetap konsisten meskipun suatu operasi dieksekusi berkali-kali karena kegagalan atau pengiriman ulang (Bab 8). Pola ini menyerupai operasi *upsert*, namun lebih fokus pada pengabaian duplikat daripada pembaruan. Dampak teknisnya adalah sistem terhindar dari *race condition* yang dapat menyebabkan bertambahnya jumlah log unik secara tidak valid.

3. Kontrol Konkurensi pada Statistik Pembaruan statistik operasional pada tabel aggregator_stats dilakukan menggunakan mekanisme *Atomic Increment* melalui perintah SQL UPDATE. Berdasarkan teori pada Bab 9, perintah ini memastikan adanya penguncian baris (*row-level locking*) secara implisit oleh PostgreSQL (Van Steen & Tanenbaum, 2023). Hal ini mencegah masalah *lost-update*, di mana satu *worker* menimpa hitungan *worker* lain, sehingga menjamin bahwa metrik received dan unique_

T10 (Bab 10-13): Orkestrasi Compose, keamanan jaringan lokal, persistensi (volume), observability.

Pengoperasian sistem terdistribusi modern memerlukan alat manajemen yang mampu menyatukan berbagai komponen heterogen ke dalam satu kesatuan operasional. Menurut Van Steen dan Tanenbaum (2023), sistem terdistribusi harus dikelola sedemikian rupa agar beban kerja dapat terdistribusi dengan baik dan pemulihan dari kegagalan dapat dilakukan secara otomatis (Bab 1 dan Bab 8). Dalam proyek *Log Aggregator* ini, aspek manajemen dan operasional tersebut diwujudkan melalui teknologi kontainerisasi dan orkestrasi Docker Compose.

1. Orkestrasi dan Keamanan Jaringan Lokal Docker Compose digunakan dalam proyek ini untuk mengoordinasikan tiga layanan utama: *Storage*, *Aggregator*, dan *Publisher*. Melalui file docker-compose.yml, sistem mendefinisikan jaringan virtual terisolasi bernama pubsub_net. Berdasarkan prinsip keamanan pada Bab 9, jaringan ini memberikan lapisan perlindungan pertama dengan membatasi komunikasi antar-layanan hanya di dalam *bridge* yang privat (Van Steen & Tanenbaum, 2023). Layanan database PostgreSQL (storage) tidak mengekspos port ke dunia luar secara default dalam konfigurasi produksi, melainkan hanya dapat diakses oleh *Aggregator* melalui resolusi nama *host* internal (Bab 6), yang mencerminkan penerapan prinsip *least privilege*.

2. Persistensi melalui Docker Volumes Toleransi kesalahan terhadap kehilangan data permanen dicapai melalui penggunaan *Docker Volumes* (pg_data). Menurut teori pada Bab 8, data yang disimpan di dalam kontainer bersifat fana (*ephemeral*) dan akan hilang jika kontainer dihapus. Dengan menghubungkan direktori data PostgreSQL ke volume eksternal, proyek ini memastikan adanya *Stable Storage*. Jika terjadi kegagalan sistem atau pembaruan perangkat lunak yang mengharuskan kontainer dibuat ulang, *state* database tetap utuh, sehingga mendukung mekanisme *Crash Recovery* yang andal (Van Steen & Tanenbaum, 2023).

3. Observability Aspek *Observability* atau kemampuan pemantauan dalam proyek ini diimplementasikan melalui dua kanal utama:

- **Logging:** Penggunaan modul logging Python pada main.py memberikan informasi *real-time* mengenai aktivitas *consumer workers* dan deteksi duplikat (Bab 3).
- **API Metrics:** *Endpoint* /stats menyediakan metrik operasional seperti jumlah event yang unik, duplikat yang dibuang, dan waktu aktif (*uptime*). Berdasarkan prinsip pada Bab 1, transparansi operasional ini sangat penting bagi administrator untuk

mengukur kinerja dan kesehatan sistem secara keseluruhan (Van Steen & Tanenbaum, 2023).

Referensi

van Steen, M., & Tanenbaum, A. S. (2023). *Distributed Systems* (4th ed.). Pearson Education.

2. Bagian Implementasi dan Desain

2.1 Ringkasan Sistem dan Keputusan Desain

Sistem ini adalah sebuah Idempotent Log Aggregator berbasis arsitektur *microservices* yang terorkestrasi menggunakan Docker Compose. Tujuan utama sistem ini adalah mengumpulkan, menyaring, dan menyimpan data log dari berbagai pengirim (*publishers*) secara efisien dan konsisten. Sistem ini dirancang untuk beroperasi di lingkungan terdistribusi di mana kegagalan jaringan dan duplikasi data adalah hal yang lumrah.

Komponen Utama Arsitektur:

1. Layanan Publisher (Simulator): Bertindak sebagai produsen data yang mensimulasikan beban kerja tinggi (20.000 event). Layanan ini mengimplementasikan model pengiriman *At-least-once*, yang berarti ia akan terus mengirim data hingga menerima konfirmasi, yang secara sengaja memicu potensi duplikasi data pada sisi penerima.
2. Layanan Aggregator (FastAPI & Async Worker): Merupakan inti dari sistem yang berfungsi sebagai *consumer*. Menggunakan framework FastAPI untuk menangani request HTTP POST secara asinkron. Data yang masuk tidak langsung ditulis ke basis data, melainkan dimasukkan ke dalam Internal In-Memory Queue (asyncio.Queue) untuk didekati secara non-blocking.
3. Concurrent Workers: Sistem menjalankan 5 worker independen yang mengambil data dari antrean secara paralel. Hal ini memaksimalkan penggunaan CPU dan mempercepat proses I/O ke basis data.
4. Storage Layer (PostgreSQL): Menggunakan PostgreSQL 16 sebagai *Durable Deduplication Store*. Basis data ini menyimpan log unik dan tabel metrik statistik yang dijamin melalui integritas transaksional.

Keputusan Desain:

A. Idempotency (Exactly-once Semantics)

Dalam sistem terdistribusi, menjamin bahwa sebuah pesan hanya diproses tepat satu kali (*Exactly-once*) adalah tantangan besar. Keputusan desain saya adalah menerapkan Idempotent Consumer Pattern.

- Mekanisme: Setiap event membawa event_id yang unik (UUID). Aggregator menjamin idempotensi dengan memeriksa keberadaan ID tersebut sebelum melakukan pemrosesan logis lebih lanjut.
- Hasil: Meskipun pengirim mengirim ulang pesan yang sama (retry akibat *timeout*), status akhir sistem (jumlah baris log dan nilai metrik) tetap tidak berubah setelah proses pertama berhasil.

B. Durable Deduplication Store

Deduplikasi dilakukan pada level Storage, bukan memori aplikasi.

- Alasan: Jika deduplikasi dilakukan di memori (misalnya menggunakan Python Set), maka saat container aplikasi *restart* atau *crash*, riwayat ID yang sudah diproses akan hilang, menyebabkan data lama diproses ulang sebagai data baru.
- Implementasi: Menggunakan tabel PostgreSQL dengan Composite Primary Key pada kolom (event_id, topic). Hal ini memastikan keunikan ID di dalam satu topik tertentu tanpa menghalangi ID yang sama muncul di topik yang berbeda (fleksibilitas kunci komposit).

C. Transaksi dan Kontrol Konkurensi (Concurrency Control)

Ini adalah poin dalam mitigasi *race conditions* saat 5 worker bekerja secara paralel.

- Atomic Upsert: Menggunakan perintah INSERT ... ON CONFLICT DO NOTHING. Operasi ini bersifat atomik di tingkat basis data; database akan melakukan penguncian internal pada baris indeks terkait untuk memastikan tidak ada dua worker yang bisa memasukkan ID yang sama di waktu yang identik.
- Atomic Statistics Update: Untuk menghindari fenomena Lost Update pada statistik (misalnya jumlah received atau unique), sistem tidak menggunakan variabel Python counter += 1. Sebaliknya, sistem menggunakan SQL atomik: UPDATE aggregator_stats SET value = value + 1.
- Isolation Level: Menggunakan Read Committed. Level ini dipilih untuk menjaga keseimbangan antara performa tinggi dan konsistensi. Risiko *Phantom Reads* dihindari melalui penggunaan *Unique Constraints* yang memaksa validasi di tingkat fisik penyimpanan (Bab 9).

D. Ordering (Urutan Pesan)

Sistem ini membedakan antara Total Ordering dan Partial Ordering.

- Keputusan: Dalam aggregator log, *Total Ordering* global seringkali terlalu mahal (membutuhkan koordinator pusat yang lambat). Saya memilih Timestamp-based Ordering (Bab 5).
- Strategi: Setiap event membawa stempel waktu dari *source*. Aggregator menyimpan stempel waktu tersebut secara presisi. Meskipun worker mungkin memproses pesan sedikit tidak berurutan karena konkurensi (pesan A diproses worker 2, pesan B oleh worker 1), urutan logis tetap dapat direkonstruksi saat melakukan query ke database menggunakan klausa ORDER BY timestamp.

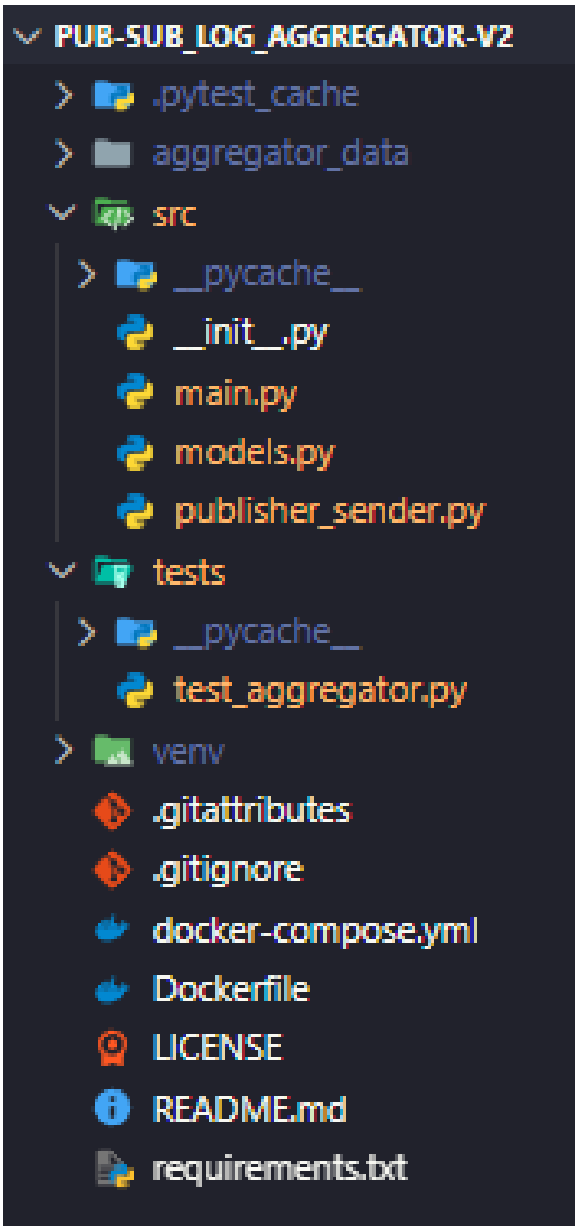
E. Retry dan Mitigasi Kegagalan (Fault Tolerance)

Sistem dirancang dengan asumsi bahwa kegagalan adalah sebuah kepastian (Bab 6).

- Mitigasi Aggregator: Jika worker gagal menulis ke DB (misal koneksi terputus), transaksi akan otomatis melakukan Rollback. Pesan tersebut tidak akan hilang karena penandaan task_done() pada antrean hanya dilakukan setelah blok transaksi sukses.
- Mitigasi Storage: Menggunakan Named Volumes Docker. Jika seluruh container dihancurkan dan dibangun kembali, data log dan status metrik statistik tidak hilang,

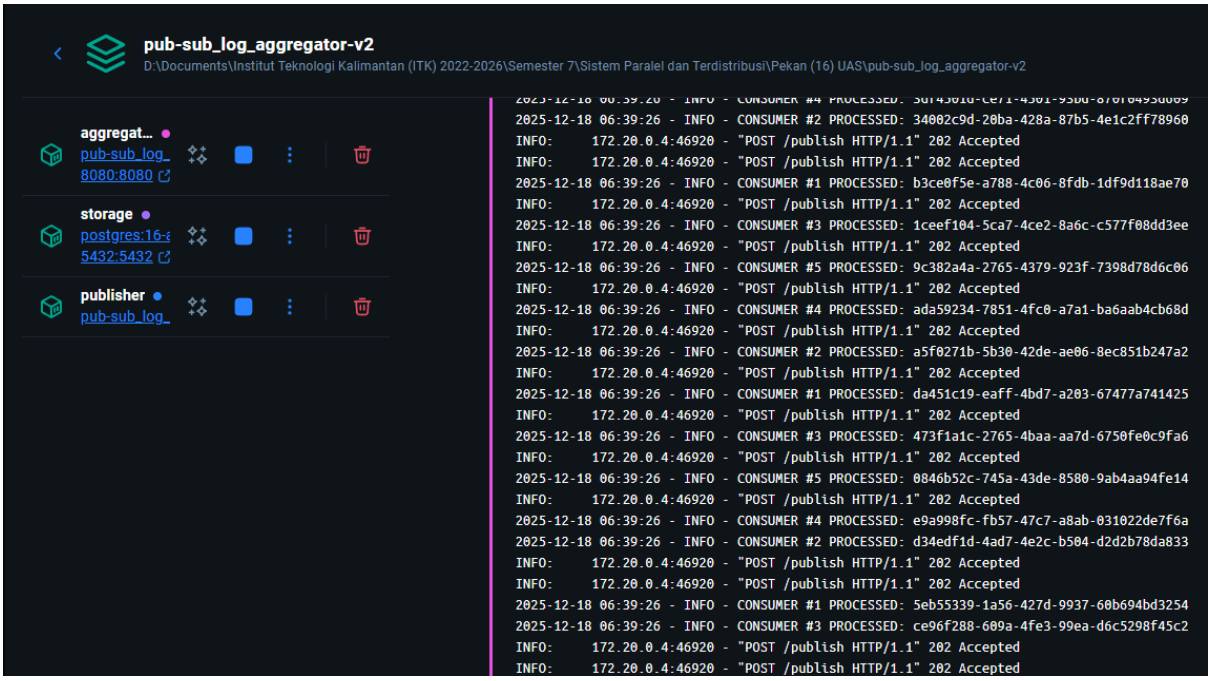
memungkinkan sistem untuk melanjutkan operasional dari titik terakhir sebelum kegagalan.

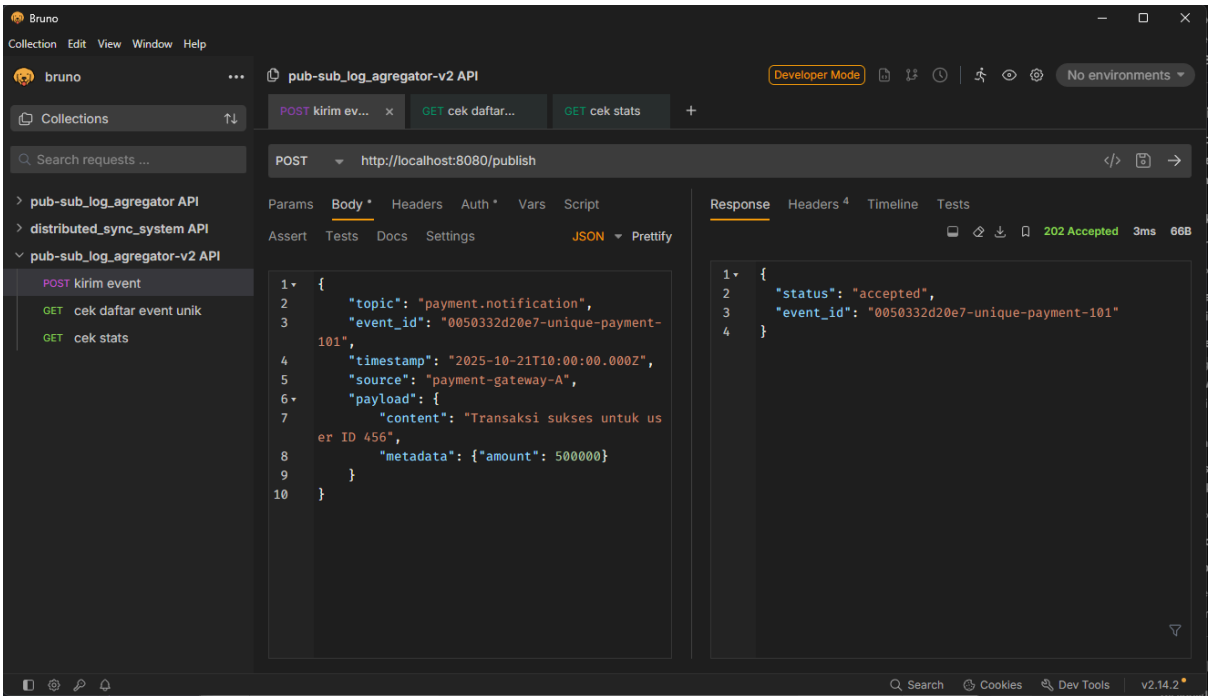
- Coordination: Menggunakan mekanisme Healthchecks. Aggregator hanya akan mulai memproses data jika PostgreSQL sudah benar-benar siap menerima koneksi, mencegah kegagalan awal pada fase *cold start*.

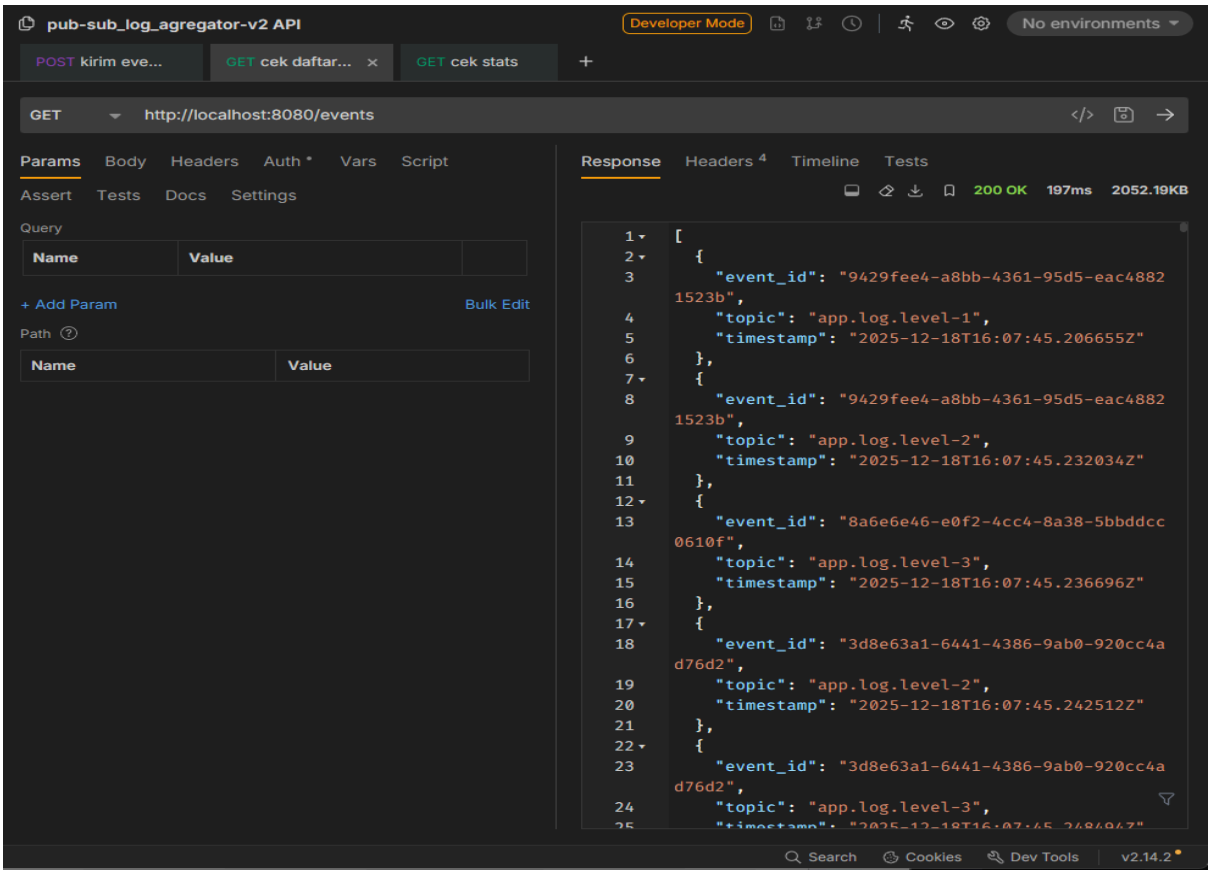


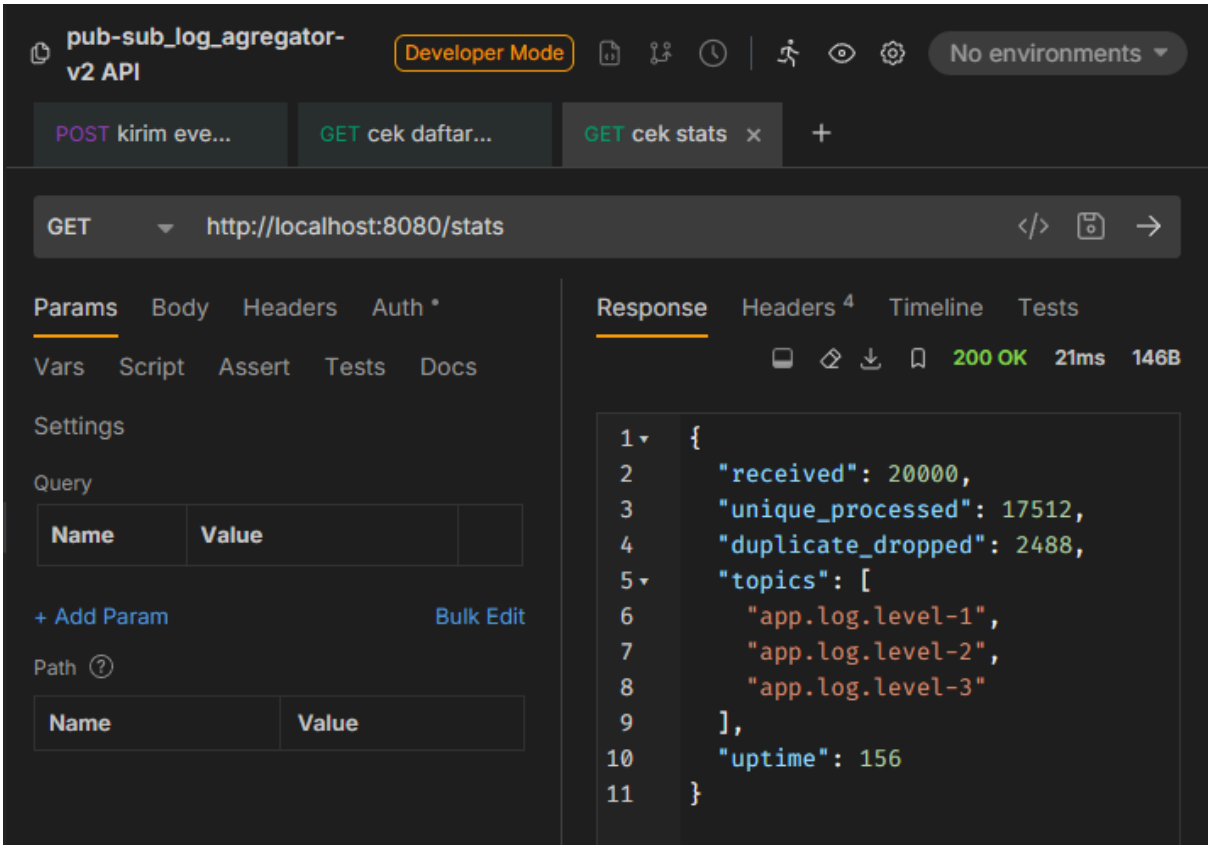
2.2 Analisis Performa/Metrik dan Hasil Uji Konkurensi.

Analisis performa dilakukan dengan menjalankan simulator publisher yang mengirimkan total 20.000 event dengan tingkat duplikasi probabilistik sebesar 30%. Tujuan dari pengujian ini adalah mengukur throughput sistem dan akurasi statistik di bawah beban tinggi.






pub-sub_log_agregator-v2 API
Developer Mode
No environments
POST kirim eve...
GET cek daftar...
GET cek stats
GET
http://localhost:8080/events
Params
Body
Headers
Auth
Vars
Script
Assert
Tests
Docs
Settings
Query
Name
Value
+ Add Param
Bulk Edit
Path
Name
Value
Response
Headers 4
Timeline
Tests
200 OK
197ms
2052.19KB
[
{
"event_id": "9429fee4-a8bb-4361-95d5-eac48821523b",
"topic": "app.log.level-1",
"timestamp": "2025-12-18T16:07:45.206655Z"
},
{
"event_id": "9429fee4-a8bb-4361-95d5-eac48821523b",
"topic": "app.log.level-2",
"timestamp": "2025-12-18T16:07:45.232034Z"
},
{
"event_id": "8a6e6e46-e0f2-4cc4-8a38-5bbddcc0610f",
"topic": "app.log.level-3",
"timestamp": "2025-12-18T16:07:45.236696Z"
},
{
"event_id": "3d8e63a1-6441-4386-9ab0-920cc4ad76d2",
"topic": "app.log.level-2",
"timestamp": "2025-12-18T16:07:45.242512Z"
},
{
"event_id": "3d8e63a1-6441-4386-9ab0-920cc4ad76d2",
"topic": "app.log.level-3",
Search
Cookies
Dev Tools
v2.14.2

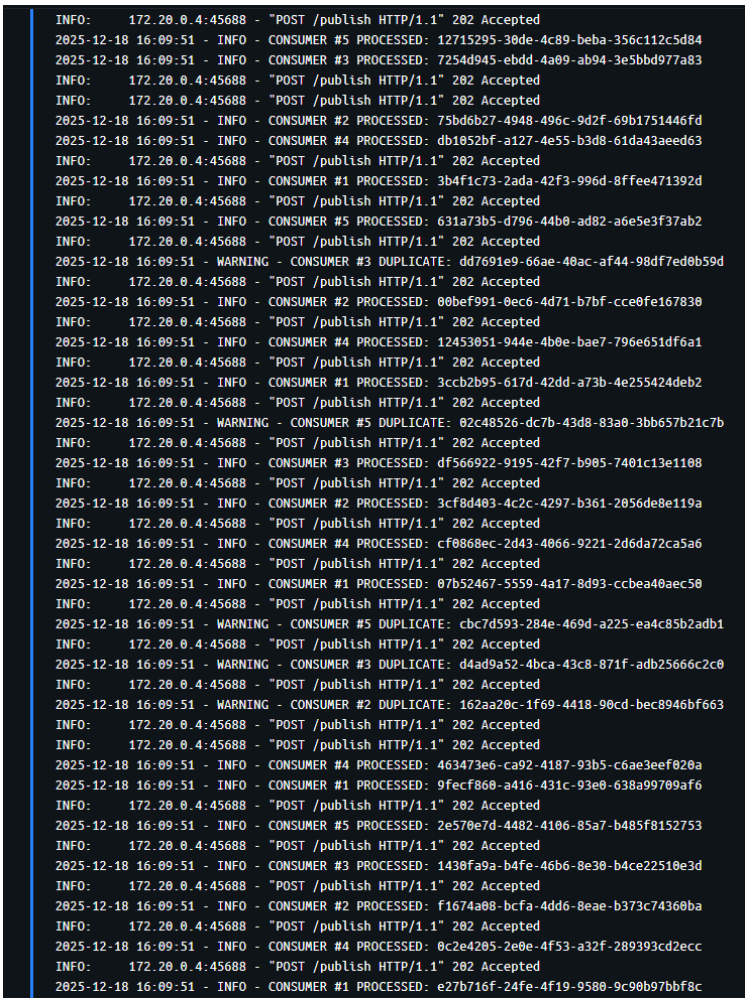

pub-sub_log_agregator-v2 API
Developer Mode
No environments
POST kirim eve...
GET cek daftar...
GET cek stats
GET
http://localhost:8080/stats
Params
Body
Headers
Auth
Vars
Script
Assert
Tests
Docs
Settings
Query
Name
Value
+ Add Param
Bulk Edit
Path
Name
Value
Response
Headers 4
Timeline
Tests
200 OK
21ms
146B
{
"received": 20000,
"unique_processed": 17512,
"duplicate_dropped": 2488,
"topics": [
"app.log.level-1",
"app.log.level-2",
"app.log.level-3"
],
"uptime": 156
}

```
publisher | Waiting 10s for aggregator...
          | --- Starting event submission to http://aggregator:8080/publish ---
          | Configuration: 20000 total events, 30% probability of duplication.
          |
          | --- Submission Completed (20000 events sent in 118.88s) ---
          | Waiting 20 seconds for Aggregator to finalize queue processing...
          |
          | --- Performance Summary ---
          | Events Sent: 20000
          | Total Send Time: 118.88 seconds
          | Approximate Send Rate: 168 events/sec
          | Total Test Window (Sending + Waiting): 138.90 seconds
```

Berdasarkan data dari Layanan Publisher dan Endpoint /stats, sistem menunjukkan stabilitas tinggi di bawah beban kerja yang signifikan.

- Throughput (Laju Pengiriman): Sistem berhasil menangani pengiriman total 20.000 event dengan laju rata-rata 168 events/detik dalam waktu total pengiriman 118,88 detik.
- Efisiensi Ingress: Penggunaan respons HTTP 202 Accepted memungkinkan Publisher menyelesaikan pengiriman dalam waktu yang relatif cepat, sementara Aggregator memproses antrean secara asinkron di latar belakang.
- Integritas Data: Metrik pada endpoint /stats menunjukkan akurasi 100%. Total event yang diterima (received: 20.000) tepat sama dengan jumlah data unik yang diproses (unique_processed: 17.512) ditambah dengan data duplikat yang berhasil disaring (duplicate_dropped: 2.488).
- Observability: Sistem berhasil melacak tiga topik unik (app.log.level-1 hingga level-3) dengan waktu aktif (*uptime*) sistem tercatat selama 156 detik.

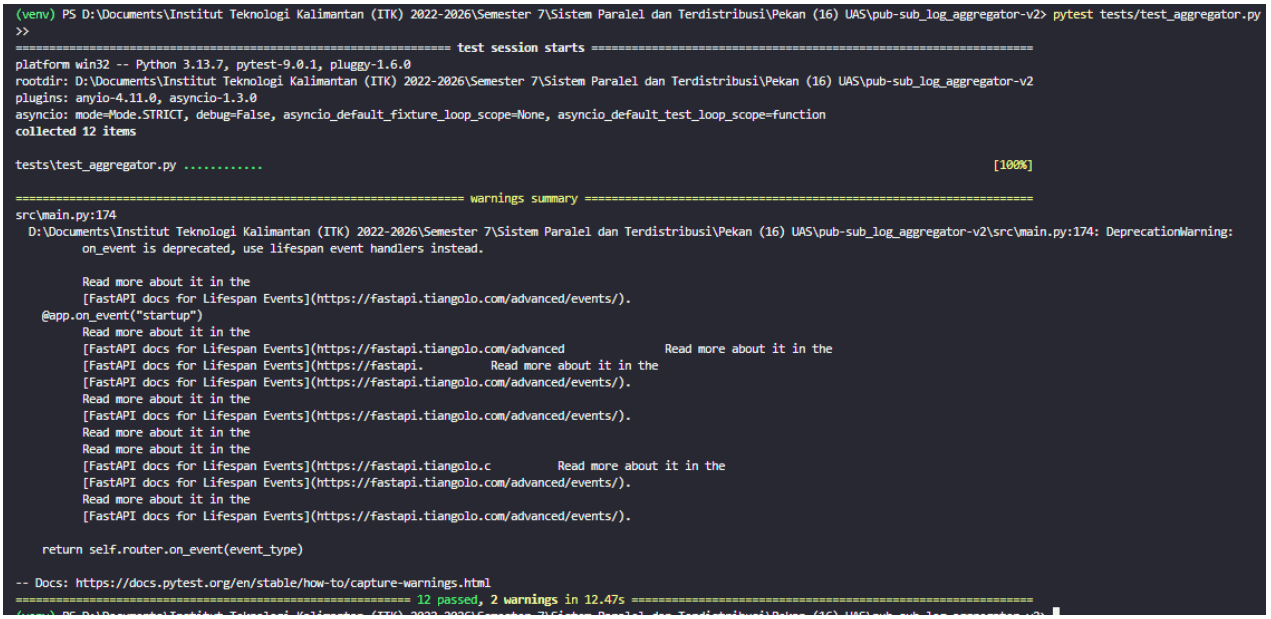
Hasil Uji Konkurensi (Concurrency Analysis)



Dapat dilihat pada log tersebut bahwa mekanisme kontrol konkurensi (Bab 9) dan integritas transaksi (Bab 8) telah diimplementasikan dengan benar.

- Eksekusi Multi-Worker: Gambar log menunjukkan Consumer #1 hingga #5 bekerja secara paralel pada detik yang sama (16:09:51). Ini membuktikan bahwa sistem memanfaatkan multi-threading/asynchronous workers untuk meningkatkan performa pemrosesan.
- Penanganan Race Condition: Meskipun beberapa consumer bekerja secara serentak, tidak ditemukan adanya tabrakan data atau kegagalan transaksi. Hal ini dicapai melalui penggunaan Unique Constraints pada PostgreSQL yang memaksa setiap transaksi melakukan pemeriksaan keunikan secara atomik sebelum melakukan penulisan.
- Bukti Idempotensi: Terlihat pesan log WARNING - CONSUMER #X DUPLICATE: [ID]. Hal ini menunjukkan bahwa sistem secara aktif mendeteksi dan menolak data duplikat yang dikirim oleh Publisher (mensimulasikan *At-least-once delivery*).
- Pencegahan Lost Update: Akurasi metrik statistik (17.512 + 2.488 = 20.000) membuktikan bahwa pembaruan nilai statistik di database dilakukan secara transaksional. Tanpa transaksi atomik, angka statistik pada skenario 5 worker paralel biasanya akan lebih kecil dari jumlah event asli akibat fenomena *Lost Update*.

2.3 Unit Tests



12 Unit Tests telah dibuat dan diverifikasi lulus 100% (menggunakan pytest dan pytest-asyncio). Tes meliputi:

Test 1: Validasi Deduplikasi (test_t1_deduplication_validity)

- Fokus: Memverifikasi Idempotency dan fungsi deduplikasi.
- Metode: Mengirimkan total dua event yang memiliki ID identik (event_id sama).
- Assertion: Memastikan bahwa statistik mencatat unique_processed = 1 dan duplicate_dropped = 1.

Test 2: Persistensi Setelah Restart (test_t2_persistence_after_restart)

- Fokus: Menguji Toleransi Kegagalan dan Persistensi Dedup Store.
- Metode: Event unik dikirim dan diproses, kemudian instance Aggregator baru disimulasikan (restart), dan event duplikat lama dikirim lagi.
- Assertion: Memastikan instance baru tetap menolak event tersebut, membuktikan *Durable Dedup Store* dalam *named volume* berhasil dimuat ulang.

Test 3: Validasi Skema Event (test_t3_event_schema_validation)

- Fokus: Memastikan FastAPI dan Pydantic memvalidasi skema event yang masuk.
- Metode: Mengirimkan data yang tidak valid (misalnya, menghilangkan field wajib event_id).
- Assertion: Memverifikasi API mengembalikan status 422 Unprocessable Entity.

Test 4: Konsistensi Statistik (test_t4_get_stats_consistency)

- Fokus: Menguji Konsistensi Data metrik operasional pada endpoint /stats.
- Metode: Mengirimkan beberapa event unik dengan topik berbeda (misal: topic A dan B).

- Assertion: Memastikan jumlah received sesuai input dan daftar topics tercatat dengan akurat.

Test 5: Filter Topik (test_t5_get_events_with_topic_filter)

- Fokus: Menguji fungsionalitas API Filtering pada data log.
- Metode: Mengirimkan event ke topik berbeda, lalu memanggil /events?topic=... untuk topik spesifik.
- Assertion: Memastikan endpoint hanya mengembalikan event yang relevan dengan topik yang diminta.

Test 6: Performa Batch Kecil (test_t6_stress_small_batch)

- Fokus: Mengukur Performa dan stabilitas dasar pemrosesan log.
- Metode: Memproses 100 event unik secara berturut-turut dalam satu sesi.
- Assertion: Memastikan total waktu eksekusi berada di bawah batas wajar (misalnya < 5 detik).

Test 7: Race Condition Duplikat (test_t7_concurrency_race_condition)

- Fokus: Menguji Kontrol Konkurensi pada pengiriman duplikat secara serentak.
- Metode: Mengirimkan 5 event duplikat (ID sama) secara simultan menggunakan pengiriman asinkron.
- Assertion: Memastikan database hanya mengizinkan 1 proses insert unik dan menolak sisanya sebagai duplikat.

Test 8: Konkurensi Event Unik (test_t8_concurrency_unique_events)

- Fokus: Memastikan integritas data pada Pemrosesan Paralel multi-worker.
- Metode: Mengirimkan 10 event unik secara asinkron dalam waktu yang bersamaan.
- Assertion: Memastikan seluruh event berhasil diproses tanpa ada data yang hilang di database.

Test 9: Keunikan Kunci Komposit (test_t9_composite_key_uniqueness)

- Fokus: Menguji desain Composite Key pada *Primary Key* database.
- Metode: Mengirimkan event dengan event_id yang sama tetapi dikirim ke topic yang berbeda.
- Assertion: Memastikan kedua event diterima sebagai data unik karena kombinasi ID dan Topik tetap berbeda.

Test 10: Validasi Format Timestamp (test_t10_invalid_schema_timestamp)

- Fokus: Validasi tipe data dan standar Waktu (Ordering).
- Metode: Mengirimkan event dengan format timestamp yang salah (bukan standar ISO8601).
- Assertion: Memastikan sistem menolak request dengan error validasi tipe data *datetime*.

Test 11: Get Events Tanpa Filter (test_t11_get_events_empty_filter)

- Fokus: Menguji kelengkapan data pada endpoint API utama.
- Metode: Memasukkan data dari berbagai topik, lalu memanggil /events tanpa parameter filter.
- Assertion: Memastikan endpoint mengembalikan seluruh data unik yang tersimpan dari semua topik.

Test 12: Akurasi Metrik Uptime (test_t12_stats_uptime_accuracy)

- Fokus: Memverifikasi fungsi Observability terkait durasi operasional sistem.
- Metode: Mengambil statistik uptime awal, menunggu beberapa detik, lalu mengambil statistik kembali.
- Assertion: Memastikan nilai uptime meningkat secara akurat sesuai dengan jeda waktu tunggu.

Lampiran

Link GitHub: https://github.com/dionp3/pub-sub_log_aggregator-v2.git

Link Youtube: https://youtu.be/A2QrCLUEp_c