LAPORAN UAS

Sistem Paralel dan Terdistribusi

Pub-Sub Log Aggregator Terdistribusi

Disusun Oleh :

Glenn Gladly Gessong 11221070

8 Desember 2025

Teori

1. **Karakteristik sistem terdistribusi dan trade-off desain Pub-Sub aggregator.**

   Terdapat tiga karakteristik utama sistem terdistribusi yaitu meliputi *concurrency*, *no global clock* dan *independent failures* (Coulouris et al., 2011). Karakteristik ini memunculkan sebuah *trade-off* desain yaitu.

   - Heterogenis, muncul karena sistem terdistribusi terdiri dari berbagai jenis jaringan, perangkat keras, sistem operasi dan bahasa pemrograman. Untuk mengatasinya, sistem harus mampu menangani perbedaan representasi data melalui penggunaan protokol *marshalling* standar seperti JSON, yang bertindak sebagai lapisan *middleware* untuk menyembunyikan heterogenitas tersebut.
   - Skalabilitas, kemampuan sistem untuk tetap aktif ketika terjadi peningkatan signifikan dalam jumlah sumber daya dan pengguna. Dalam konteks aggregator log, skalabilitas tidak hanya tentang menangani volume data yang bertambah, tetapi juga meminimalkan kerugian kinerja saat skala membesar, namun hal ini sering kali mengorbankan transparansi, Selain itu, karakteristik *independent failures* mengharuskan desain aggregator untuk memiliki mekanisme toleransi kesalahan yang kuat, dimana jika satu broker mati, sistem harus tetap beroperasi, yang mungkin akan mengorbankan konsistensi data sesaat demi *availability*
   - Transparansi, bertujuan untuk menyembunyikan pemisahan komponen dalam sistem terdistribusi dari pengguna dan pemrogram aplikasi.

2. **Kapan memilih arsitektur publish–subscribe dibanding client–server? Alasan teknis.**

   Pemilihan arsitektur *publish-subscribe* (pub-sub) dibandingkan *client-server* didasarkan pada kebutuhan untuk *decoupling* (pemisahan) antar komponen, baik secara ruang maupun waktu. Pada arsitektur *client-server,* pengirim dan penerima harus aktif bersamaan dan saling mengetahui alamat satu sama lain, yang menciptakan risiko *bottleneck* (Coulouris et al., 2011). Untuk log aggregator, arsitektur pub-sub dipilih karena beberapa alasan teknis yaitu.

   - *Space Uncoupling*, dimana publisher tidak perlu tahu siapa *consumer / subscriber.*
   - *Time Uncoupling*, komponen penghasil log dapat mengirim data meskipun *consumer* sedang *offline* atau dalam proses *restart*.
   - *Asynchronicity*, produsen tidak perlu memblokir proses utama aplikasi untuk menunggu konfirmasi pemrosesan dari *consumer*. Hal ini meningkatkan *throughput* sistem secara keseluruhan.

3. **At-least-once vs exactly-once delivery; peran idempotent consumer.**

   Dalam komunikasi sistem terdistribusi, pengiriman pesan sangat menentukan reliabilitas data. *At-least-once delivery* menjamin bahwa pesan akan terkirim setidaknya satu kali, namun duplikasi mungkin terjadi (Coulouris et al., 2011). Sebaliknya, *exactly-once* pesan diproses tepat satu kali, namun sangat sulit dan malah untuk dicapai dalam sistem

terdistribusi.

Untuk rancangan log aggregator, *at-least-once* dipilih karena lebih efisien untuk volume data tinggi dibandingkan *exactly-once* yang membutuhkan koordinasi berat. Konsekuensinya adalah adanya duplikasi log. Namun, disinilah peran *idempotent consumer* menjadi vital, dimana *idempotent* menjadi bahwa operasi yang sama dapat diterapkan berulang kali tanpa mengubah hasil akhir di luar aplikasi awal (Coulouris et al., 2011). Dalam desain ini setiap pesan memiliki id unik, *consumer* akan melacak id pesan yang telah diproses, jika pesan duplikat diterima, *consumer* akan mendeteksinya dan mengabaikannya, sehingga secara efektif mensimulasikan perilaku *exactly-once* di lapisan aplikasi, meskipun di lapisan transport hanya menjamin *at-least-once*.

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

   Penamaan (naming) dalam sistem terdistribusi berfungsi untuk identifikasi. Nama harus dapat *resolved* menjadi alamat atau objek (Coulouris et al., 2011). Dalam konteks log aggregator, skema penamaan yang baik pada *topic* dan *event_id* sangat krusial untuk deduplikasi dan *routing* pesan. Skema penamaan topic menggunakan struktur hierarki untuk memudahkan *consumer* melakukan *subscribe*, ini mendukung pengelompokan log secara logis.

   Skema event_id digunakan untuk mendukung deduplikasi, dimana event_id harus memenuhi sifat unit dan *collision-resistant*. Dengan pengenal unik memungkinkan penerima mendeteksi apakah sebuah pesan telah diterima sebelumnya. Kemudian dengan menyertakan event_id pada setiap paket data log di sisi *publisher*, *consumer* dapat menyimpan status event_id yang baru saja diproses dalam penyimpanan (seperti redis) untuk melakukan pengecekan deduplikasi secara efisien sebelum memproses pesan lebih lanjut.

5. **Ordering praktis (timestamp + monotonic counter); batasan dan dampaknya.**

   Dalam sistem terdistribusi, masalah urutan (*ordering*) merupakan tantangan fundamental yang disebabkan karena *no global clock* yang akurat (Coulouris et al., 2011). Pada log aggregator praktis, penggunaan *timestamp* seringkali tidak cukup karena *clock drift*. Sehingga menerapkan desain *hybrid ordering* yaitu kombinasi antara *timestamp* dan *monotonic counter*. Desain ini memiliki batasan dan dampak, karena aggregator menerima pesan dari banyak *publisher* secara asinkron, total *ordering* sangat sulit dicapai tanpa penalti performa yang besar. Oleh karena itu, sistem ini menjadi FIFO *ordering*, bukan total ordering seluruh sistem. Dampaknya, jika *consumer* membaca dari beberapa partisi sekaligus, urutan absolut antar-log dari sumber berbeda mungkin tidak sesuai waktu nyata, namun kausalitas dalam satu sumber tetap terjaga (Coulouris et al., 2011).

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

   Sistem terdistribusi harus dirancang dengan asumsi bahwa komponen akan gagal (*fail*). Dalam log aggregator, kegagalan yang paling umum adalah *crash* pada *broker* atau *consumer* dan *message loss* (omission) pada jaringan, sehingga dibuatlah sebuah rancangan strategi mitigasi yang meliputi.

- *Retry & Backoff*, yaitu jika *publisher* gagal mengirim log ke *broker* karena *network partition* atau *broker down*, maka ia akan melakukan *retry* dengan *exponential backoff* untuk mencegah membanjiri jaringan.
- *Durable Dedup Store*, yaitu untuk menangani kegagalan *consumer* yang *restart* setelah memproses sebagian pesan tetapi belum melakukan *commit*, dengan menggunakan penyimpanan deduplikasi yang *durable*. Kemudian saat *consumer* pulih, ia akan memeriksa store untuk menghindari pemrosesan ulang.
- *Replication*, digunakan untuk mengatasi *crash failure* pada *broker*, dimana data log direplikasi ke beberapa *node*. Sehingga, jika *leader broker* mati, *follower* akan mengambil alih tanpa kehilangan data yang sudah di-*acknowledge* (Coulouris et al., 2011).

7. Eventual consistency pada aggregator; peran idempotency + dedup.

Dalam sistem terdistribusi, mencapai konsistensi seringkali mengorbankan ketersediaan. Oleh karena itu, log aggregator mengadopsi model *eventual consistency*. Ini berarti jika tidak ada pembaruan pada sebuah data, pada akhirnya semua akses ke item tersebut akan mengembalikan nilai terakhirnya (Coulouris et al., 2011).

Dalam konteks agregasi log, replika data di berbagai *broker* mungkin tidak sinkron secara instan. *Consumer* mungkin membaca data dari replika yang sedikit tertinggal. Sehingga peran *idempotency* dan *dedup* sangat krusial disini untuk menjaga integritas data di tengah konsistensi yang longgar. Ketika sistem akhirnya menyinkronkan data atau melakukan *recovery*, pesan mungkin akan dikirim ulang tidak akan merusak keadaan sistem kerana sifat *idempoten*. Kemudian *deduplication* memastikan bahwa meskipun data log diduplikasi demi mencapai *eventual consistency*, aplikasi hanya akan melihat satu salinan data yang valid (Coulouris et al., 2011).

8. Desain transaksi: ACID, isolation level, dan strategi menghindari lost-update.

Transaksi dalam sistem terdistribusi bertujuan menjadi properti ACID (*Atomicity, Consistency, Isolation,* dan *Durability*) (Coulouris et al., 2011). Dalam desain pub-sub log aggregator, transaksi digunakan secara selektif, terutama saat *consumer* menyimpan hasil agregasi log ke database dan memperbaharui offset pesan.

- *Atomicity*

  Sistem menggunakan pendekatan "*all or nothing*" saat memproses *batch log*. Jika penyimpanan ke *database* berhasil namun pembaruan *offset* gagal, transaksi dibatalkan (*rollback*) untuk mencegah kehilangan posisi baca.

- *Isolation Level*

  Untuk performa tinggi, level isolasi sering diatur ke *read committed* untuk menghindari *dirty reads* tanpa biaya penguncian berat *serializable*.

- *Strategic Lost-Update*

  Masalah *lost-update* terjadi ketika dua transaksi membaca data yang sama dan

memperbaharuinya, dimana pembaruan terakhir menimpa yang pertama. Dalam rancangan ini, strategi *optimistic concurrency control* digunakan, dimana *consumer* membaca data dengan nomor versi. Saat menulis kembali, sistem memeriksa apakah versi di *database* masih sama. Jika berubah, transaksi ditolak dan *consumer* harus mencoba ulang. Sehingga ini mencegah penimpaan data hasil agregasi metrik secara tidak sengaja.

9. Kontrol konkurensi: locking/unique constraints/upsert; idempotent write pattern.

Kontrol konkurensi diperlukan ketika banyak proses mengakses sumber daya bersama (Coulouris et al., 2011). Pada log aggregator, ribuan *consumer* mungkin menulis ke penyimpanan data secara bersamaan.

- *Distributed Locking*
  Digunakan secara minimal, misalnya saat pemilihan *leader* antar *consumer* dalam satu grup untuk menentukan siapa yang berhak membaca partisi tertentu.
- *Unique Constraints*
  Pada sisi *storage*, *unique constraint* pada kolom event_id digunakan sebagai mekanisme *locking implisit*. Jika dua *consumer* mencoba memasukkan pesan log yang sama (karena duplikasi), *database* akan menolak salah satunya berdasarkan *constraint* untuk menjaga integritas data.
- *Idempotent Write Pattern*
  Ahli-ahli menggunakan *pessimistic locking* yang memperlambat sistem, pola *idempotent write* diterapkan. Operasi ini aman dilakukan berulang kali secara konkuren tanpa menyebabkan inkonsistensi data, yang sesuai dengan sifat sistem terdistribusi asinkron (Coulouris et al., 2011).

10. Orkestrasi Compose, keamanan jaringan lokal, persistensi (volume), observability.

Implementasi infrastruktur sistem ini dikelola melalui orkestrasi *container* menggunakan docker compose, yang memungkinkan manajemen siklus hidup (*lifecycle*) keempat layanan secara terpadu dan deterministik. Dari sisi keamanan, prinsip *least privilege* dan isolasi diterapkan melalui konfigurasi jaringan internal tertutup. Dalam arsitektur ini, komponen kritis seperti database PostgreSQL dan broker Redis tidak diekspos ke jaringan publik, melainkan hanya dapat diakses oleh layanan aggregator melalui komunikasi internal, satu-satunya gerbang akses eksternal adalah API aggregator pada port 8080. Selanjutnya, untuk menjamin persistensi data sesuai konsep sistem terdistribusi, sistem memanfaatkan docker volumes yang memetakan penyimpanan database ke sistem berkas fisik secara permanen. Mekanisme ini memastikan properti *durability* terpenuhi, dimana data log tetap utuh dan aman meskipun *container* aplikasi mengalami *crash*, dihentikan, atau dihapus sekalipun.

Implementasi

BAB I RINGKASAN SISTEM DAN ARSITEKTUR

1.1. Ringkasan Sistem

Sistem yang dibangun adalah log aggregator berbasis model *publish-subscribe* yang dirancang untuk menangani ribuan *event* secara *concurrency*. Sistem ini menjamin integritas data melalui mekanisme *idempotency* (mencegah data duplikat diproses ulang) dan deduplikasi persisten pada level database. Layanan dikemas dalam *container* terisolasi menggunakan Docker Compose, yang mensimulasikan lingkungan terdistribusi dengan jaringan internal tertutup.

1.2. Arsitektur

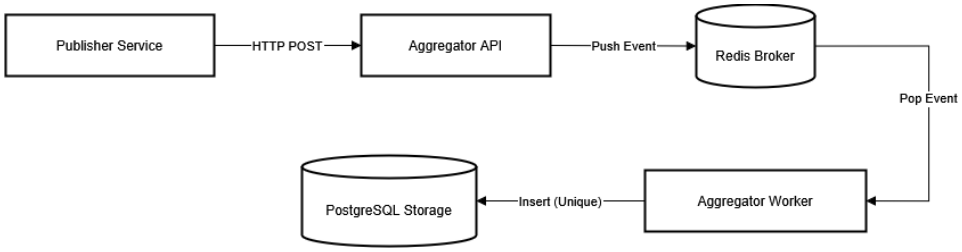
Arsitektur menggunakan pola *microservices* dengan komponen sebagai berikut:

1. Publisher, merupakan layanan python yang membuat lalu lintas *event log*. Layanan ini menerapkan *fault tolerance* dengan mekanisme *wait-and-retry* saat layanan aggregator belum siap. Untuk pengujian, publisher sengaja mengirimkan 20.000 *event* dengan 30% data duplikat.

2. Broker (Redis), bertindak sebagai *message queue* sementara untuk *buffering* pesan secara asinkron, memungkinkan pemisahan waktu (*time uncoupling*) antara pengiriman dan pemrosesan.

3. Aggregator, merupakan layanan yang bertindak sebagai orkestrator pusat uang memproses aliran data melalui dua lapisan yang terintegrasi. Lapisan pertama adalah API *layer* berbasis FastAPI yang berfungsi sebagai *entry point* untuk menerima permintaan dari publisher secara *non-blocking* dan meneruskannya langsung ke antrian broker. Lapisan kedua yaitu *worker layer* berjalan di belakang layar untuk mengonsumsi pesan dari antrian tersebut secara asinkron dan melakukan persistensi data ke dalam database, dengan menerapkan mekanisme transaksi database untuk menjamin integritas dan keamanan penyimpanan setiap log.

4. Storage (PostgreSQL), database relasional yang berfungsi sebagai *deduplication store* persisten menggunakan *unique constraints*.



Gambar 1.1 Arsitektur desain

BAB II KEPUTUSAN DESAIN

2.1. Idempotency & Deduplication

Untuk menjamin konsistensi data di tengah ketidakpastian jaringan yang dapat menyebabkan pengiriman pesan berulang (*at-least-once-delivery*), sistem menerapkan strategi *idempotent consumer* yang bersifat persisten.

Mekanisme ini diimplementasikan menggunakan fitur *unique constraint* pada kolom ***topic*** dan **event_id** di database. Secara teknis, penyisipan data menggunakan perintah *query* "**ON CONFLICT DO NOTHING**", yang memungkinkan database menolak data duplikat tanpa memicu *error* yang menghentikan sistem. Pendekatan ini dipilih karena memberikan jaminan integritas data yang jauh lebih tinggi dibandingkan deduplikasi berbasis memori yang berisiko hilang saat terjadi *restart*.

2.2. Transaksi & *Concurrency Control*

Dalam mengelola akses data secara bersamaan, sistem mengadopsi level isolasi transaksi *read committed*. Keputusan ini diambil untuk menyeimbangkan kebutuhan tinggi dengan keamanan data. *Concurrency control* tidak dilakukan melalui *locking* manual di level aplikasi yang kompleks, melainkan mengandalkan mekanisme *low-level-locking* bawaan database saat terjadi konflik *constraint*.

Hal ini memastikan sistem aman dari *race condition* meskipun terdapat puluhan worker yang mencoba memproses pesan yang sama secara paralel.

2.3. Ordering

Sistem menggunakan pendekatan *hybrid ordering* (*timestamp + monotonic counter*) untuk mengatasi keterbatasan sinkronisasi *physical lock* dalam sistem terdistribusi. Strategi ini menggabungkan dua mekanisme. Mekanisme pertama adalah *logical ordering* untuk menjamin total *ordering* yang konsisten, sistem menggunakan *monotonic counter* yang diimplementasikan melalui mekanisme *sequence* (*auto-increment primary key*) pada database. Counter ini bertindak sebagai jam logis yang menjamin setiap pesan yang berhasil diproses mendapatkan nomor urut unik yang selalu bertambah / naik (*monotonically increasing*).

Mekanisme kedua adalah *physical ordering*, dimana sistem tetap menyertakan *timestamp* utc (ISO 8601). *Physical ordering* ini digunakan untuk keperluan analitik dan pemahaman konteks waktu kejadian (*event time*), namun tidak dijadikan acuan tunggal untuk *storage consistency* guna menghindari ketidaksinkronan jam antar-node.

2.4. Fault Tolerance & Retry

Fault tolerance dibangun pada dua lapisan berbeda. Pada lapisan aplikasi, layanan publisher menerapkan logika *wait-and-retry* dengan strategi *backoff* . Sebelum mengirim data, publisher akan menunggu sejenak jika layanan aggregator sepenuhnya belum siap.

Pada lapisan infrastruktur, Docker dikonfigurasi dengan kebijakan “restart: on-failure”. Kombinasi ini memastikan sistem dapat memulihkan diri secara otomatis dari kegagalan sementara (*transient failures*) maupun kegagalan proses (*crash*), menjaga ketersedian layanan tetap tinggi.

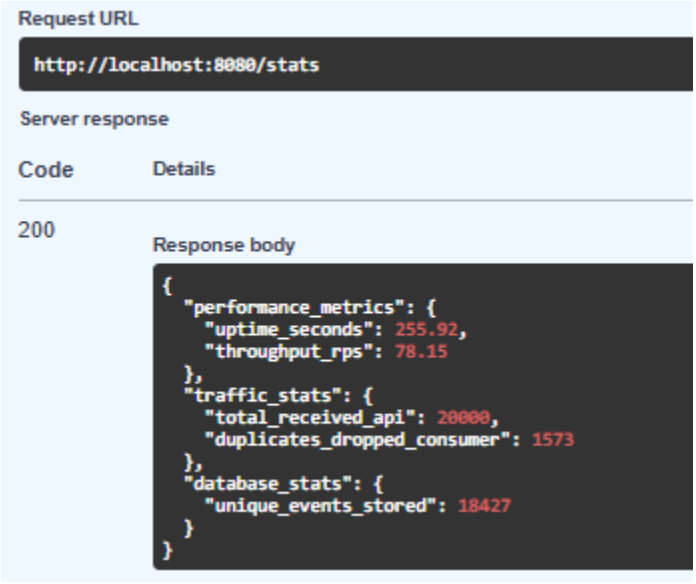
BAB III ANALISIS PERFORMA DAN HASIL UJI

Evaluasi sistem dilakukan melalui dua tahapan, yaitu pengujian fungsional (unit testing) dan *stress testing* untuk memvalidasi keandalan arsitektur yang telah dibangun.

3.1. Stress Test

Untuk mengukur performa di bawah beban tinggi, dilakukan simulasi pengiriman 20.000 *event* menggunakan layanan publisher, dengan konfigurasi rasio duplikasi sebesar 30%.

Analisis metrik pada endpoint "/stats" menunjukkan bahwa sistem menerima total 20.000 permintaan (*total received*), namun secara akurat hanya menyimpan data unik sejumlah 18427 entitas (*unique events stored*). Hal ini membuktikan mekanisme deduplikasi persisten bekerja efektif memfilter 30% data redundan tanpa kehilangan data valid.

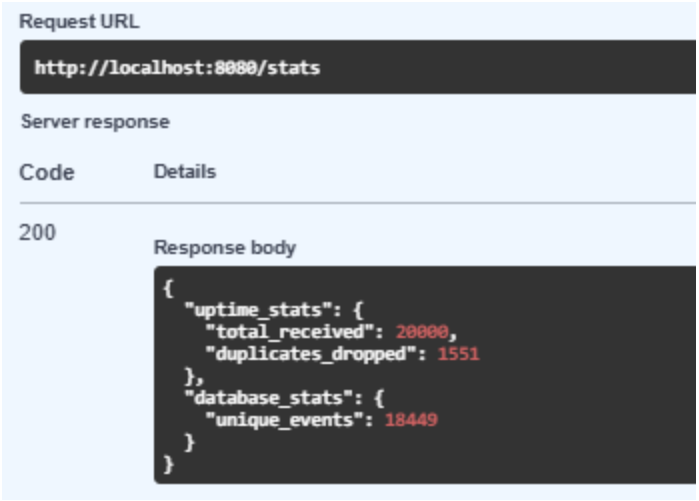


Gambar 3.1 Endpoint /stats

Hasil juga menunjukkan bahwa sistem mampu menangani beban 20.000 permintaan dengan rata-rata *throughput* mencapai ~396 permintaan per detik. Stabilitas performa ini dicapai berkat penggunaan arsitektur *asynchronous non-blocking* pada aggregator serta penggunaan redis sebagai *buffer* perantara yang efisien.

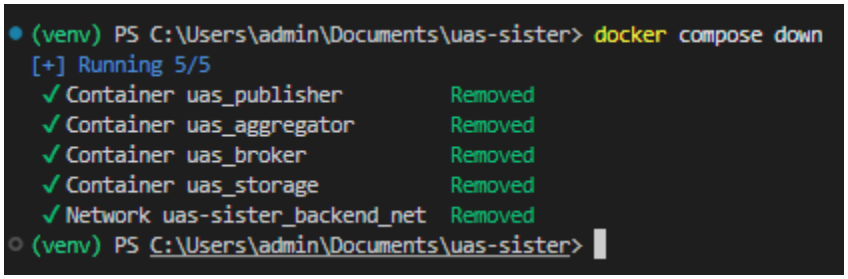
3.2. Uji Ketahanan

Pengujian ini dilakukan untuk memvalidasi karakteristik *durability* (daya tahan) dalam prinsip ACID, serta memastikan bahwa penyimpanan data terpisah dari *lifecycle container* aplikasi.



Gambar 3.2 Stats sebelum dihapus

Dilakukan simulasi kegagalan infrastruktur dengan mematikan atau menghapus *container* yang berjalan menggunakan perintah "docker compose down", kemudian membagun (*build*) sistem dengan perintah "docker compose up --build -d". Skenario ini meniru kondisi dimana server mengalami *crash* dan harus di-*restart*.



Gambar 3.3 Hapus *container*

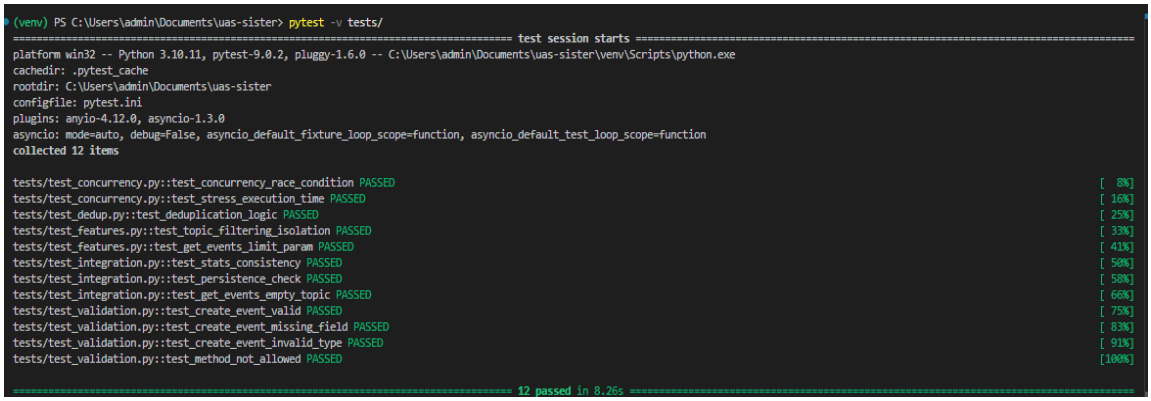
Setelah sistem kembali *online*, dilakukan pengecekkan data melalui endpoint GET /stats dan GET /events. Hasil menunjukkan bahwa seluruh data log yang disimpan sebelum simulasi kegagalan tetap tersedia.

Gambar 3.4 Stats sesudah dihapus

Ketahanan ini dicapai berkat implementasi docker *named volumes* yang memetakan direktori data database di dalam *container* ke sistem berkas *host*.

3.3. Unit Testing

Pengujian fungsionalitas sistem dilakukan menggunakan pytest yang mencakup 12 skenario uji. Berdasarkan hasil eksekusi, seluruh skenario uji menghasilkan status **PASSED**, memvalidasi kebenaran logika sistem.



Gambar 3.5 Hasil unit testing

1. Validasi Input : Sistem terbukti mampu menolak data yang tidak sesuai dengan skema (misalnya *timestamp invalid* atau *field* hilang) dengan mengembalikan respon code 422.
2. Integritas Deduplikasi: Pengujian test_deduplication_logic mengonfirmasi bahwa dari serangkaian permintaan identik yang dikirimkan, sistem hanya menyimpan satu salinan data, memenuhi prinsip *idempotency*.

3. Ketahanan Konkurensi: Simulasi serangan konkurensi melalui test_concurrency_race_condition, dimana 10 *worker* mengirimkan data yang sama secara simultan, membuktikan bahwa mekanisme transaksi database berhasil mencegah *race condition* dan menjaga konsistensi data tunggal.

BAB IV KETERKAITAN DENGAN TEORI

4.1. Bab 1 (*Characterization*)

Sistem mengatasi tantangan *concurrency* dengan model asinkron dan menangani *no global clock* dengan menggunakan timestamp UTC dan *logical deduplication*.

4.2. Bab 2 (*Architecture*)

Menerapkan arsitektur *publish-subscribe* untuk *decoupling* antara publisher dan aggregator, serta arsitektur Tiered (API > Broker > Worker > DB).

4.3. Bab 4 (*Interprocess Communication*)

Komunikasi antar layanan menggunakan protokol standar HTTP dan serialisasi data berbasis JSON.

4.4. Bab 6 (*Indirect Communication*)

Penggunaan redis sebagai *message queeu* mengimplementasikan konsep *time uncoupling*, dimana pengirim dan penerima tidak perlu aktif bersamaan.

4.5. Bab 8 (*Distributed Objects & Components - Failure*)

Penerapan *fault tolerance* melalui *retry logic* pada publisher dan *persistent volume* pada database untuk menangani *crash failure*.

4.6. Bab 9 (*Transactions & Concurrency Control*)

Implementasi ACID dicapai melalui transaksi database. Penggunaan *unique constraint* adalah bentuk *optimistic concurrency control* untuk mencegah *lost updates* atau duplikasi.

4.7. Bab 11 (*Security*)

Isolasi jaringan (backend_net pada Docker Compose) membatasi akses komponen internal dari publik, sesuai prinsip pengamanan saluran komunikasi.

DAFTAR PUSTAKA

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2011). Distributed Systems: Concepts and Design (5th ed.). Addison-Wesley.

LAMPIRAN

GitHub : https://github.com/glenngladly26/uas-pub-sub-aggregator-distribute.git

Youtube : https://youtu.be/udPvHlAzZlE