BAB 3
PELAKSANAAN KERJA MAGANG

3.1 Kedudukan dan Koordinasi

3.1.1 Kedudukan

Selama program magang di PT Pintarnya Solusi Teknologi, penempatan dilakukan pada divisi *Engineering* sebagai Full Stack Engineer dengan tanggung jawab utama mengimplementasikan *observability tools* pada sistem *microservice* berbasis Golang. Dalam struktur organisasi, posisi magang berada di bawah arahan Supervisor yang menjabat sebagai Senior Software Engineer di dalam divisi *Engineering* perusahaan.

3.1.2 Koordinasi

Selama kegiatan magang, koordinasi dilakukan secara langsung dengan Supervisor yang bertanggung jawab terhadap proyek yang diberikan. Koordinasi ini bertujuan untuk memastikan setiap langkah implementasi *observability tools* berjalan sesuai dengan kebutuhan perusahaan. Supervisor memberikan arahan terkait rancangan arsitektur, pemilihan teknologi, serta praktik terbaik dalam penerapan *monitoring* dan *observability*. Selain itu, komunikasi rutin dilakukan untuk membahas progres pekerjaan, kendala teknis yang dihadapi, serta solusi yang dapat diterapkan. Dengan pola koordinasi yang terfokus pada satu pihak, kegiatan magang dapat berjalan lebih efektif dan menghasilkan bimbingan yang mendalam mengenai penerapan *observability tools* sebagai *proof-of-concept* untuk mendukung rencana perusahaan beralih ke arsitektur *microservice* berbasis *Golang*.

3.2 Tugas yang Dilakukan

Selama menjalani kegiatan magang di PT Pintarnya Solusi Teknologi, tugas yang dikerjakan berkaitan dengan pengembangan dan peningkatan *observability* pada sistem berbasis *microservice*, khususnya *microservice* berbasis Golang. Adapun tugas-tugas yang dilakukan selama masa magang adalah sebagai berikut:

1. Mengimplementasikan *observability tools* seperti Prometheus, Loki, Jaeger, dan Grafana untuk memantau performa sistem, melakukan *tracing request*

antar *microservice*, serta menyediakan visualisasi metrik yang membantu tim dalam proses *debugging* dan analisis.

2. Menerapkan *Inbox-Outbox pattern* pada *microservice* berbasis Golang untuk memastikan konsistensi data dan keandalan komunikasi antar layanan, terutama dalam skenario transaksi yang melibatkan lebih dari satu sistem.
3. Melakukan konfigurasi dan integrasi *observability tools* dengan *microservice* yang ada, termasuk *setup dashboard* di Grafana, konfigurasi *alerting*, serta integrasi *log management* menggunakan Loki.
4. Melakukan pengujian (*testing*) terhadap sistem observability yang dikembangkan, meliputi validasi metrik, *tracing*, serta identifikasi dan perbaikan *bug* pada integrasi observability maupun penerapan *Inbox-Outbox pattern*.
5. Berkoordinasi dengan tim pengembang untuk menyelaraskan pekerjaan, melakukan integrasi modul observability ke dalam sistem secara keseluruhan, serta memastikan pola *Inbox-Outbox* berjalan sesuai standar arsitektur yang diterapkan perusahaan.
6. Mendokumentasikan pekerjaan yang dilakukan sebagai bagian dari pelaporan dan pengembangan proyek berkelanjutan, termasuk dokumentasi konfigurasi *observability tools* dan penerapan *Inbox-Outbox pattern*.

Pelaksanaan kerja magang dijelaskan melalui pembagian mingguan yang disusun pada Tabel 3.1, yang memuat rincian pekerjaan yang dilakukan setiap minggu selama masa kerja magang.

Tabel 3.1. Pekerjaan yang dilakukan tiap minggu selama pelaksanaan kerja magang

Minggu Ke -	Pekerjaan yang dilakukan
1	Mengikuti proses *onboarding*, mengenal lingkungan kerja, serta melakukan konfigurasi proyek lokal dan *repository*.
2	Mempelajari serta memahami alur kerja *Git workflow*.
3	Mendalami konsep arsitektur *microservice* dan prinsip observability.
4	Mempelajari berbagai *observability tools* seperti Prometheus, Grafana, Loki, dan Jaeger.
5	Menerapkan Prometheus untuk pengumpulan *metrics*.
6	Melakukan konfigurasi dan pembuatan *dashboard* pada Grafana.
7	Melakukan integrasi antara Grafana dengan Prometheus.
8	Melakukan konfigurasi Loki untuk *log aggregation*.
9	Melanjutkan proses integrasi Loki dalam sistem *log aggregation*.
10	Mengembangkan *shared library microservice* untuk integrasi dengan Jaeger.
11	Menerapkan Jaeger untuk *distributed tracing*.
12	Menerapkan pola *Inbox-Outbox* dalam sistem.
13	Melanjutkan penerapan pola *Inbox-Outbox*.
14	Melaksanakan pengujian sistem serta melakukan perbaikan *bug* pada keseluruhan proyek.
15	Melanjutkan pengujian sistem dan perbaikan berdasarkan masukan dari rekan kerja.
16	Menyusun dokumentasi serta melakukan *review* menyeluruh terhadap proyek.

3.3 Uraian Pelaksanaan Magang

Dalam pelaksanaan magang, tugas utama berkaitan dengan rencana perusahaan untuk beralih ke arsitektur *microservice* berbasis Golang. Mengingat transisi tersebut masih dalam tahap perencanaan, diberikan tugas untuk menyusun sebuah *proof-of-concept (PoC) monitoring* dan *observability* yang nantinya dapat diimplementasikan ketika *microservice* telah dibangun. *Proof-of-concept* ini

mencakup perancangan sistem pemantauan kinerja layanan, pengumpulan *log*, serta pelacakan distribusi *request* menggunakan berbagai *observability tools* seperti Prometheus, Grafana, Loki, dan Jaeger. Dengan adanya PoC ini, perusahaan diharapkan memiliki fondasi yang kuat untuk memastikan setiap *microservice* dapat dipantau secara efektif, sehingga memudahkan proses identifikasi masalah, analisis performa, dan peningkatan keandalan sistem di masa mendatang.

3.3.1 Metrics Collection dengan Prometheus dan Grafana

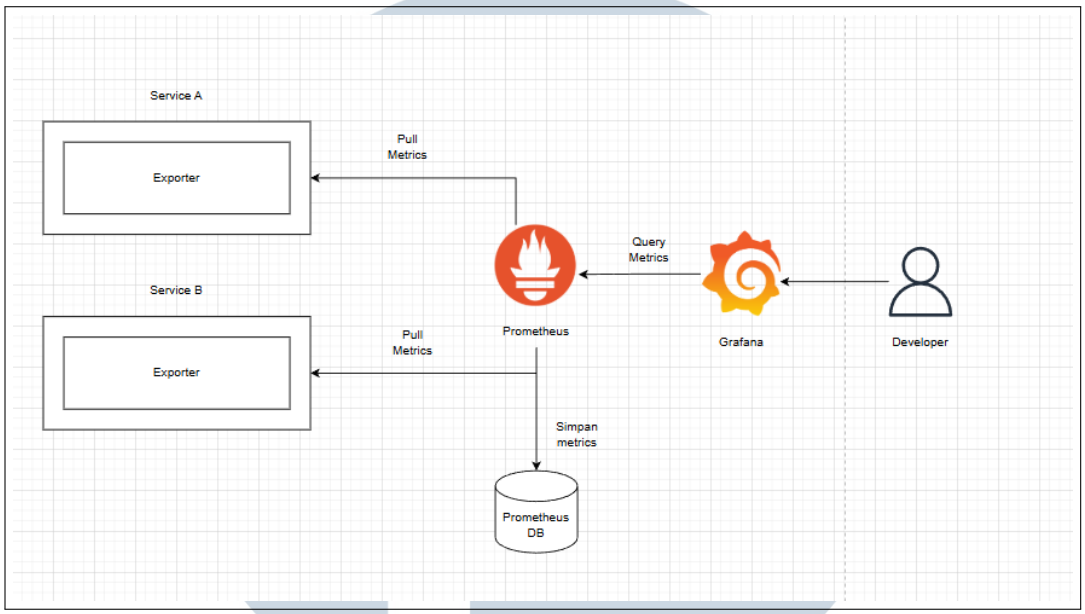
Dalam tahap ini, perancangan dan implementasi sistem *metrics collection* dilakukan menggunakan Prometheus dan Grafana. Prometheus digunakan untuk melakukan *scraping* terhadap *metrics* yang diekspos oleh aplikasi maupun *exporter*, kemudian menyimpannya dalam bentuk *time-series database* [6]. Data tersebut kemudian diolah dan divisualisasikan melalui Grafana dalam bentuk *dashboard* interaktif [3]. Dengan adanya integrasi ini, perusahaan dapat memantau performa sistem secara *real-time*, seperti penggunaan CPU, memori, jumlah *request*, maupun tingkat *error*. *Proof-of-concept* ini menjadi dasar penting agar ketika arsitektur *microservice* berbasis Golang mulai dibangun, sistem *observability* sudah siap digunakan untuk mendukung *monitoring* yang komprehensif.

A Teknologi yang Dipakai

Berikut adalah teknologi yang dipakai, yaitu:

- **Prometheus**: Prometheus adalah *open-source monitoring system* yang berfungsi untuk melakukan *metrics scraping* dari berbagai sumber. Data yang dikumpulkan disimpan dalam bentuk *time-series database* sehingga dapat dianalisis menggunakan bahasa *query* khusus, yaitu PromQL. Prometheus mendukung integrasi dengan berbagai *exporter* (misalnya Node Exporter, cAdvisor) untuk mengumpulkan data dari sistem operasi, *container*, maupun aplikasi.[6]
- **Grafana**: Grafana adalah platform visualisasi data yang digunakan untuk membuat *dashboard* interaktif. Grafana dapat terhubung dengan Prometheus sebagai *data source*, sehingga *metrics* yang dikumpulkan dapat divisualisasikan dalam bentuk grafik, tabel, maupun *alerting system*. Keunggulan Grafana adalah fleksibilitas dalam membuat *dashboard* yang mudah dipahami oleh tim operasional maupun *developer*.[3]

B Arsitektur Metrics Collection

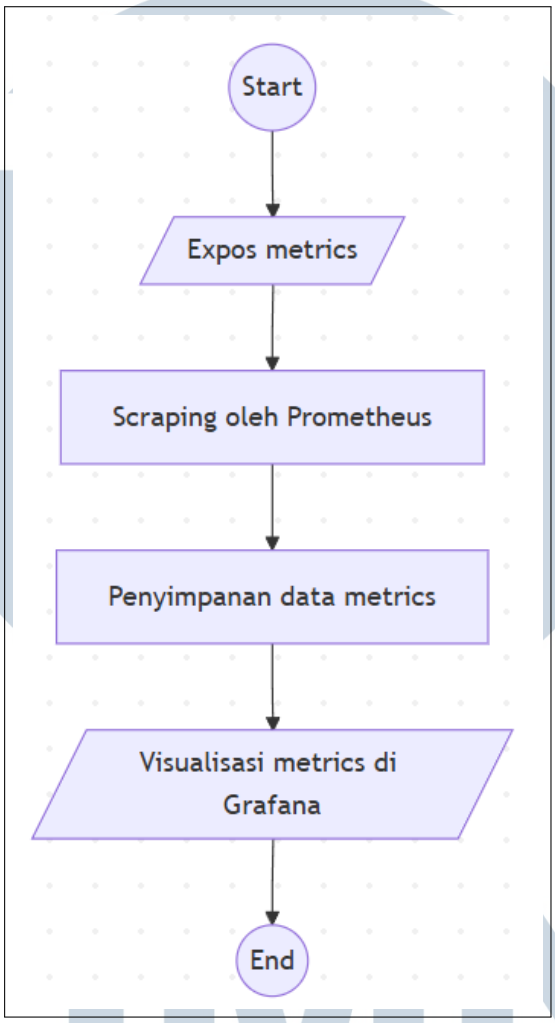


Gambar 3.1. Arsitektur *metrics collection* menggunakan Prometheus dan Grafana

Gambar 3.1 menunjukkan arsitektur *metrics collection* yang terdiri dari beberapa komponen utama:

- **Service**: Meng-*expose* endpoint *metrics* (biasanya /metrics) yang dapat di-*scrape* oleh Prometheus.
- **Exporter**: Digunakan untuk mengumpulkan *metrics* dari sistem atau aplikasi.
- **Prometheus Server**: Bertugas melakukan *scraping*, menyimpan data dalam *time-series database*, dan menyediakan API untuk *query*.
- **Grafana**: Terhubung ke Prometheus sebagai *data source* untuk menampilkan *metrics* dalam bentuk *dashboard*.

C Alur Metrics Collection



Gambar 3.2. *Flowchart metrics collection* menggunakan Prometheus dan Grafana

Gambar 3.2 menunjukkan alur proses *metrics collection* yang dapat dijelaskan dalam beberapa tahap:

1. Ekspos *metrics*: Aplikasi atau sistem menyediakan *endpoint metrics* (misalnya /metrics) atau menggunakan *exporter*.
2. *Scraping* oleh Prometheus: Prometheus melakukan *scraping* secara periodik.
3. Penyimpanan Data *Metrics*: *Metrics* yang dikumpulkan disimpan dalam *database* internal Prometheus.

4. Visualisasi *Metrics*: Grafana akan melakukan *query* terhadap Prometheus menggunakan PromQL dan menampilkan *metrics* dalam bentuk grafik, tabel, atau indikator lain sesuai kebutuhan.

D Hasil Implementasi

3.3.2 Dashboard Monitoring di Grafana

Gambar 3.3. *Dashboard* Grafana

Gambar 3.3 menunjukkan tampilan dashboard *monitoring* yang dibangun menggunakan Grafana. Dashboard ini menampilkan berbagai metrik performa sistem dalam rentang waktu 5 menit terakhir, dengan pembaruan data setiap 10 detik. Dashboard ini membantu tim pengembang dan operasional dalam memantau kesehatan layanan secara *real-time*, mengidentifikasi anomali, serta melakukan analisis performa berdasarkan metrik kuantitatif. Visualisasi yang interaktif dan terstruktur ini menjadi bagian penting dalam sistem *observability* yang mendukung arsitektur *microservice*.

A Black Box Testing Prometheus dan Grafana

Tabel 3.2. Black Box Testing Prometheus dan Grafana

No	Langkah Uji	Input	Output yang Diharapkan	Status
1	Ekspos endpoint /metrics	Endpoint aktif	Metrics ditampilkan di Prometheus	Lulus
2	Buat panel baru di Grafana	Query PromQL	Grafik muncul sesuai query	Lulus
3	Ubah interval scraping	Interval 10s	Data diperbarui setiap 10 detik	Lulus
4	Filter metrics dengan label	label job=”api”	Data sesuai label ditampilkan	Lulus

Tabel 3.2 menunjukkan hasil *black box testing* terhadap integrasi Prometheus dan Grafana. Pelaksanaan pengujian dilakukan secara kolaboratif oleh Supervisor yang menjabat sebagai Senior Software Engineer.

3.3.3 Log Aggregation dengan Loki

Dalam tahap ini, *log aggregation* diimplementasikan menggunakan Loki yang terintegrasi dengan Promtail dan Grafana. Promtail bertugas mengumpulkan *log* dari aplikasi atau sistem, menambahkan label *metadata*, lalu mengirimkannya ke Loki untuk disimpan. Loki kemudian menyimpan *log* dalam bentuk *chunks* dengan *indexing metadata* sehingga pencarian *log* menjadi lebih efisien. Hasil *log* dapat di-*query* menggunakan LogQL dan divisualisasikan melalui Grafana. Dengan adanya sistem ini, perusahaan dapat melakukan analisis *log* secara terpusat, memudahkan proses *debugging*, serta mengidentifikasi pola *error* atau anomali yang terjadi pada aplikasi.[4]

A Teknologi yang Dipakai

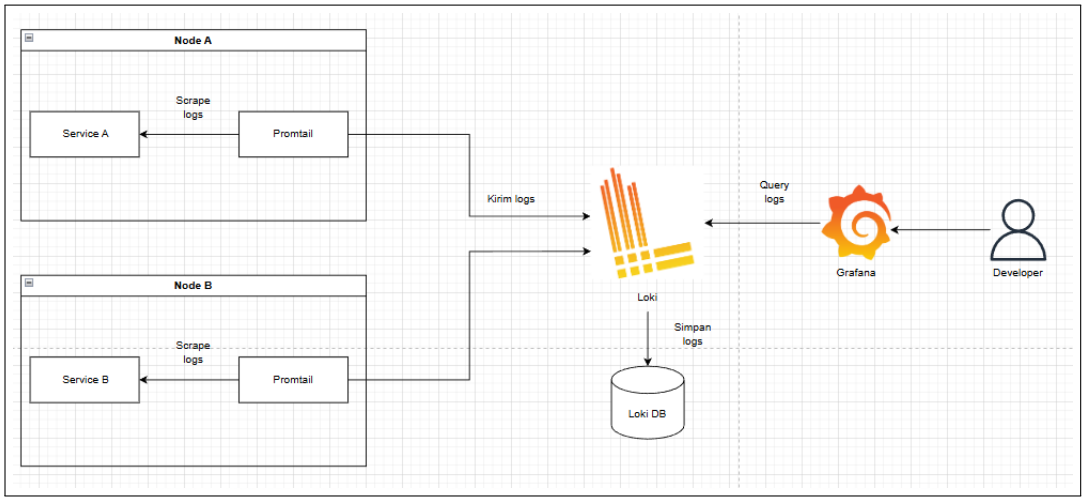
Berikut adalah teknologi yang dipakai, yaitu:

- **Loki**: Loki adalah sistem *log aggregation open-source* yang dikembangkan oleh Grafana Labs. Loki dirancang untuk mengumpulkan, menyimpan, dan

melakukan *query* terhadap *log* dengan cara efisien. Berbeda dengan sistem *log* tradisional, Loki hanya melakukan *indexing* pada *metadata* (seperti label, nama aplikasi, atau *namespace*), sehingga lebih hemat sumber daya dan cocok untuk skala besar.[4]

- **Promtail**: Promtail adalah agen *log* yang berjalan di *server* atau *container*. Tugas utama Promtail adalah membaca *log* dari *file*, `stdout`/`stderr`, atau sistem *logging* lain, kemudian menambahkan label *metadata* sebelum mengirimkannya ke Loki. Dengan Promtail, proses pengumpulan *log* menjadi otomatis dan terintegrasi dengan baik.[8]
- **Grafana (Integrasi dengan Loki)**: Grafana digunakan sebagai antarmuka visual untuk menampilkan *log* yang dikumpulkan oleh Loki. Dengan integrasi ini, pengguna dapat melakukan pencarian *log* menggunakan bahasa *query* LogQL dan menampilkan hasilnya dalam *dashboard* yang sama dengan *metrics* dari Prometheus.[3]

B Arsitektur Log Aggregation



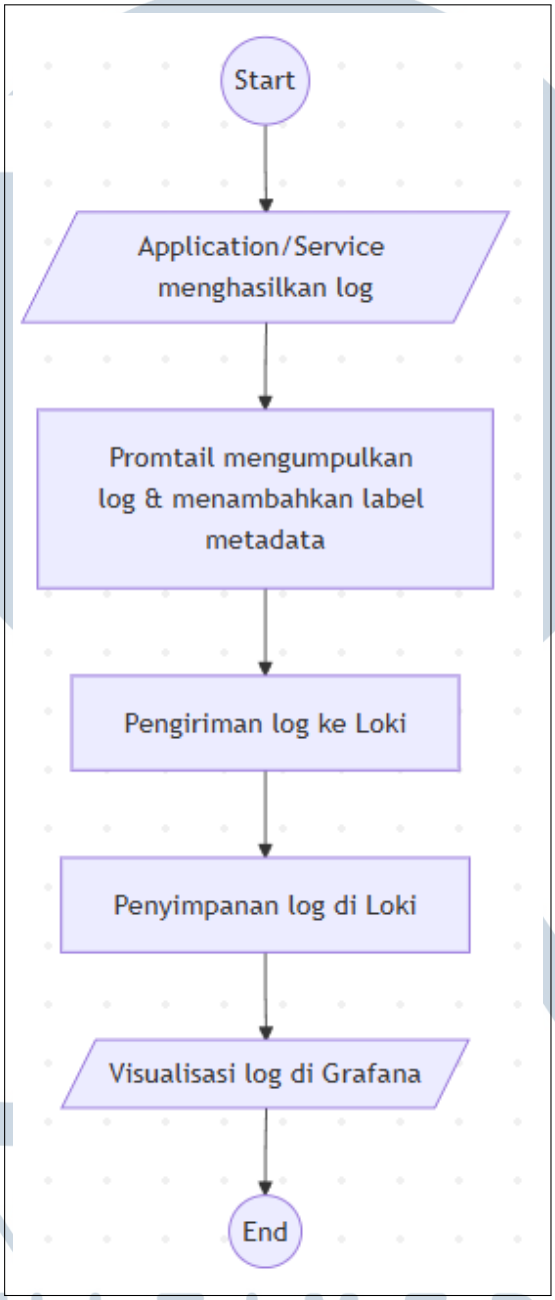
Gambar 3.4. Arsitektur *log aggregation* menggunakan Loki

Gambar 3.4 menunjukkan arsitektur *log aggregation* dengan Loki yang terdiri dari beberapa komponen utama:

- **Service**: Menghasilkan *log* (stdout/stderr atau file log).

- **Promtail**: Agen yang mengumpulkan *log* dari aplikasi atau sistem, menambahkan *metadata label*, lalu mengirimkannya ke Loki.
- **Loki Server**: Menyimpan *log* dalam bentuk *chunks* dan *metadata label*.
- **Grafana**: Menampilkan *log* melalui query LogQL dan menyediakan *dashboard* terpadu.

C Alur Log Aggregation



Gambar 3.5. *Flowchart log aggregation* menggunakan Loki

Gambar 3.5 menunjukkan alur proses *log aggregation* yang dapat dijelaskan dalam beberapa tahap:

1. **Log Generation**: Aplikasi atau sistem menghasilkan *log* (misalnya request, error, atau aktivitas sistem).

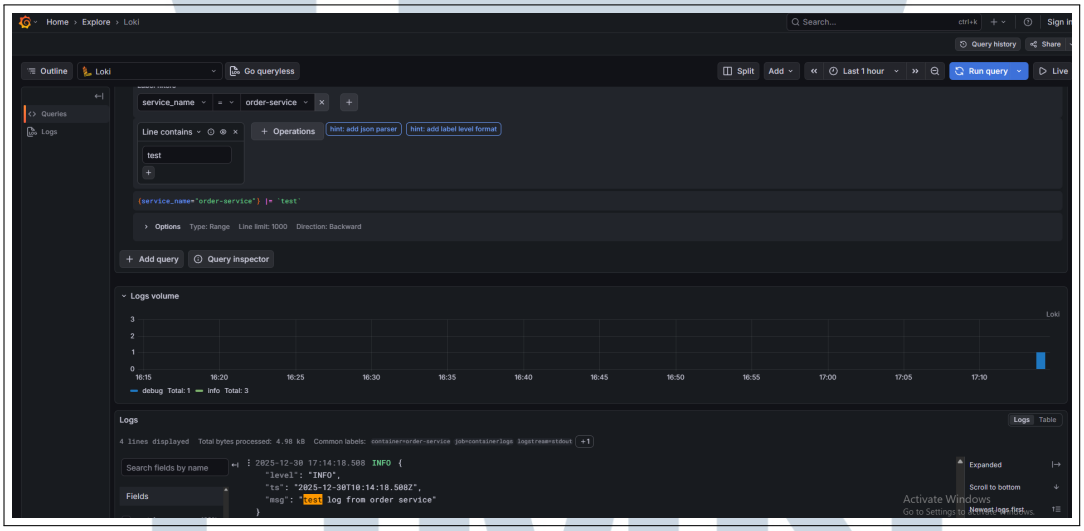
2. **Log Collection oleh Promtail**: Promtail membaca *log* dari file atau output aplikasi, menambahkan label metadata (misalnya nama aplikasi, namespace).

3. **Pengiriman ke Loki**: Promtail mengirimkan *log* ke Loki untuk disimpan.

4. **Penyimpanan di Loki**: Loki menyimpan *log* dalam bentuk *chunks* dan melakukan *indexing metadata*.

5. **Visualisasi di Grafana**: Grafana akan melakukan *query* terhadap Loki menggunakan LogQL dan menampilkan *log* dari service yang terintegrasi.

D Hasil Implementasi



Gambar 3.6. *Query log* menggunakan Loki

Gambar 3.6 memperlihatkan proses pencarian *log* menggunakan Loki melalui antarmuka Grafana Explore. *Query* yang digunakan adalah ”test”, yang bertujuan untuk menampilkan *log* dari Order Service yang mengandung kata ”test”. Hasil pencarian ditampilkan dalam bentuk grafik volume *log* serta daftar *log* mentah yang sesuai dengan kriteria. Tampilan ini membantu tim pengembang dalam melakukan analisis *log* secara cepat dan terarah, terutama untuk keperluan *debugging* dan validasi aktivitas sistem.

E Black Box Testing Loki

Tabel 3.3. Black Box Testing Loki

No	Langkah Uji	Input	Output yang Diharapkan	Status
1	Kirim log dari aplikasi	Log INFO	Log tampil di Grafana	Lulus
2	Query log berdasarkan label	Label service=”order”	Log relevan ditampilkan	Lulus
3	Query log berdasarkan keyword log	”test”	Log relevan ditampilkan	Lulus
4	Kirim log ERROR	Log dengan level ERROR	Log ditandai sebagai error	Lulus
5	Kirim log dengan timestamp	Log dengan waktu tertentu	Log muncul sesuai waktu	Lulus

Tabel 3.3 menunjukkan hasil *black box testing* terhadap integrasi *log aggregation* menggunakan Loki. Pelaksanaan pengujian dilakukan secara kolaboratif oleh Supervisor yang menjabat sebagai Senior Software Engineer.

3.3.4 Distributed Tracing dengan Jaeger

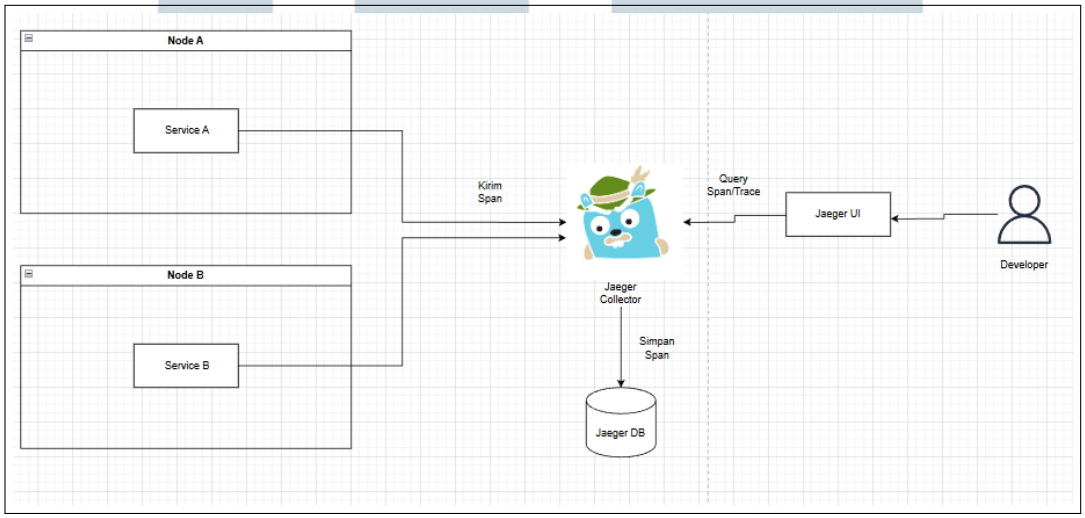
Dalam tahap ini, *distributed tracing* diimplementasikan menggunakan Jaeger. *Distributed tracing* bekerja dengan merekam jejak eksekusi sebuah request dalam bentuk *trace*, yaitu rangkaian aktivitas yang menggambarkan perjalanan request dari satu layanan ke layanan lain. Setiap aktivitas di dalam *trace* direpresentasikan sebagai *span*, yang berisi informasi detail seperti nama operasi, waktu mulai, durasi, serta metadata tambahan. *Tracing* ini memungkinkan perusahaan untuk melacak alur request yang melewati berbagai layanan dalam arsitektur *microservice*.[2]

A Teknologi yang Dipakai

Berikut adalah teknologi yang dipakai, yaitu

- **Jaeger**: Jaeger adalah *open-source distributed tracing system* yang dikembangkan oleh Uber Technologies. Jaeger digunakan untuk memantau dan memecahkan masalah pada sistem mikroservis dengan cara melacak alur permintaan *(request)* yang melewati berbagai layanan. Dengan Jaeger, developer dapat mengetahui *latency*, *bottleneck*, dan dependensi antar layanan.[2]

B Arsitektur Distributed Tracing

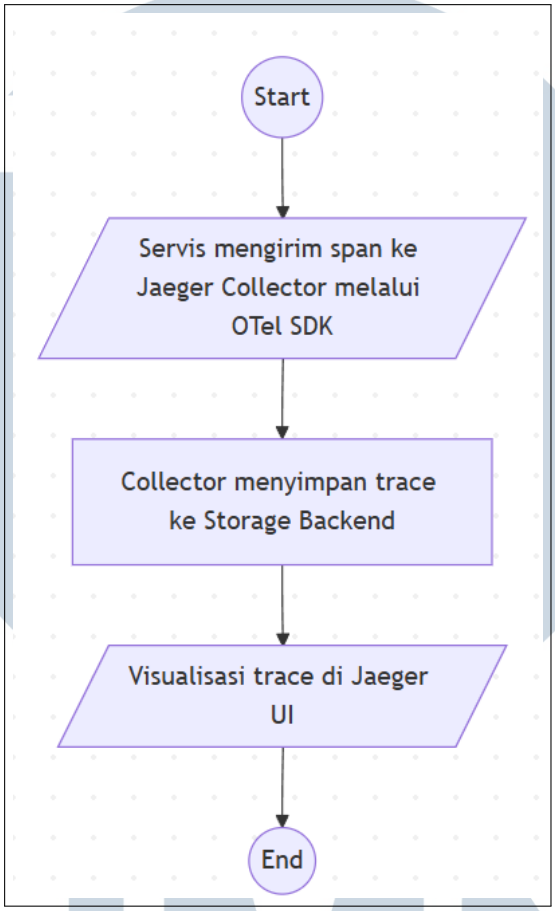


Gambar 3.7. Arsitektur *distributed tracing* menggunakan Jaeger

Gambar 3.7 menunjukkan arsitektur *distributed tracing* dengan Jaeger yang terdiri dari beberapa komponen utama:

- **Service**: Layanan *microservice* yang di-instrumentasi menggunakan OpenTelemetry SDK. SDK ini bertugas menghasilkan data *tracing* berupa *span* dan *context propagation* dari setiap request yang melewati layanan..
- **Jaeger Collector**: Jaeger Collector menerima data *tracing (span)* dari aplikasi atau agen, memprosesnya, lalu menyimpannya ke *storage backend* untuk dapat di-*query* dan divisualisasikan.
- **Jaeger UI**: Antarmuka visual untuk melakukan pencarian *trace* dan menampilkan alur permintaan secara end-to-end yang memungkinkan untuk melihat *latency*, dependensi antar layanan, serta *bottleneck* dalam sistem.

C Alur Distributed Tracing

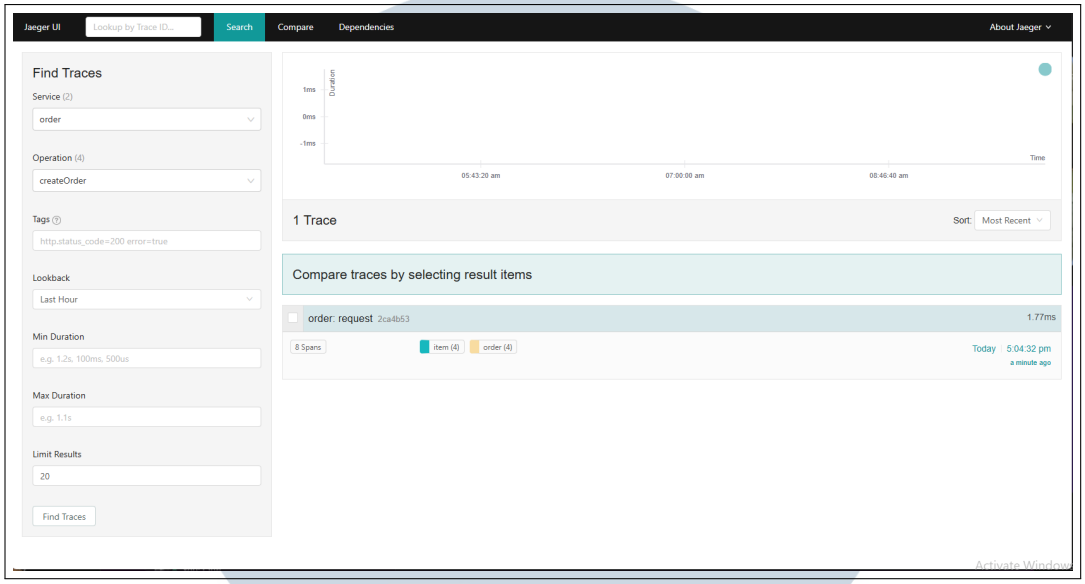


Gambar 3.8. *Flowchart distributed tracing* menggunakan Jaeger

Gambar 3.8 menunjukkan alur proses *distributed tracing* yang dapat dijelaskan dalam beberapa tahap:

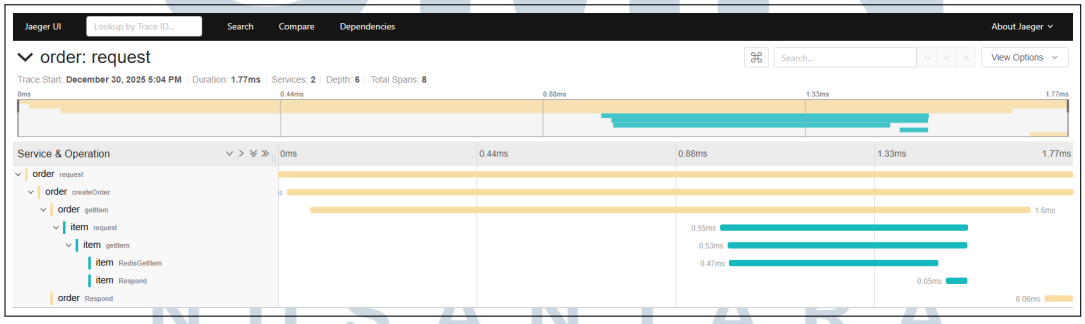
1. **Pengiriman Span**: Aplikasi *microservice* di-instrumentasi menggunakan OpenTelemetry SDK untuk menghasilkan data *tracing* berupa *span* yang akan dikirim ke Jaeger Collector.
2. **Penyimpanan Span**: Collector bertugas menerima dan memproses data *tracing* yang kemudian akan disimpan ke dalam *database*.
3. **Visualisasi di Jaeger UI**: Data *tracing* yang tersimpan dapat di-*query* melalui Jaeger Query dan divisualisasikan di Jaeger UI. Developer dapat melihat alur lengkap perjalanan *request*, detail *latency* di setiap *span*, serta hubungan antar layanan.

D Hasil Implementasi



Gambar 3.9. Daftar *traces* pada Jaeger UI

Gambar 3.9 menunjukkan tampilan utama dari antarmuka pengguna Jaeger, sebuah *observability tool* yang digunakan untuk melakukan *distributed tracing* pada sistem berbasis *microservice*. Pada halaman ini, pengguna dapat melakukan pencarian *trace* berdasarkan parameter seperti nama service, jenis operasi, tags, durasi minimum dan maksimum, serta periode waktu pencarian *(lookback)*. Tampilan ini menjadi titik awal dalam proses analisis alur permintaan antar layanan, yang sangat penting untuk *debugging* dan pemantauan performa sistem secara menyeluruh.



Gambar 3.10. Detail *trace* pada Jaeger UI

Gambar 3.10 menampilkan hasil pencarian *trace* untuk suatu operasi pada Order Service. *Trace* tersebut memiliki beberapa *span* yang mencakup interaksi

antara dua layanan, yaitu Order Service dan Item Service. Visualisasi ini menunjukkan struktur hierarki dan durasi masing-masing *span*. Informasi seperti waktu mulai, durasi total, kedalaman *trace*, serta jumlah *span* ditampilkan secara rinci, sehingga memudahkan proses identifikasi *bottleneck* dan analisis performa antar layanan.

E Black Box Testing Jaeger

Tabel 3.4. Black Box Testing Jaeger

No	Langkah Uji	Input	Output yang Diharapkan	Status
1	Kirim request antar layanan	Request HTTP antar service	Trace muncul di UI dengan span lengkap	Lulus
2	Kirim request dengan delay	Request lambat	Span menunjukkan durasi tinggi	Lulus
3	Kirim request berantai	Request antar 2 service	Semua span memiliki trace ID yang sama	Lulus
4	Cari trace dengan nama service tertentu	Nama service valid	Trace ditemukan sesuai filter	Lulus

Tabel 3.4 menunjukkan hasil *black box testing* terhadap integrasi *distributed tracing* menggunakan Jaeger. Pelaksanaan pengujian dilakukan secara kolaboratif oleh Supervisor yang menjabat sebagai Senior Software Engineer.

3.3.5 Inbox-Outbox Pattern

Inbox-Outbox Pattern adalah sebuah pola arsitektur yang digunakan untuk menjamin konsistensi data dan keandalan komunikasi antar layanan dalam sistem terdistribusi. Pola ini biasanya diterapkan ketika sebuah aplikasi perlu melakukan operasi *database* sekaligus mengirimkan pesan ke sistem lain. Dengan adanya *outbox*, setiap perubahan data yang terjadi di *database* akan dicatat terlebih dahulu, kemudian pesan dikirim secara asinkron melalui *message broker*. Hal ini mencegah terjadinya inkonsistensi akibat kegagalan pengiriman pesan atau transaksi yang

tidak selesai. Sementara *inbox* digunakan untuk memastikan bahwa pesan yang diterima dari sistem lain hanya diproses sekali *(idempotent processing)*, sehingga menghindari duplikasi.[5]

A Teknologi yang Dipakai

Berikut adalah teknologi yang dipakai, yaitu

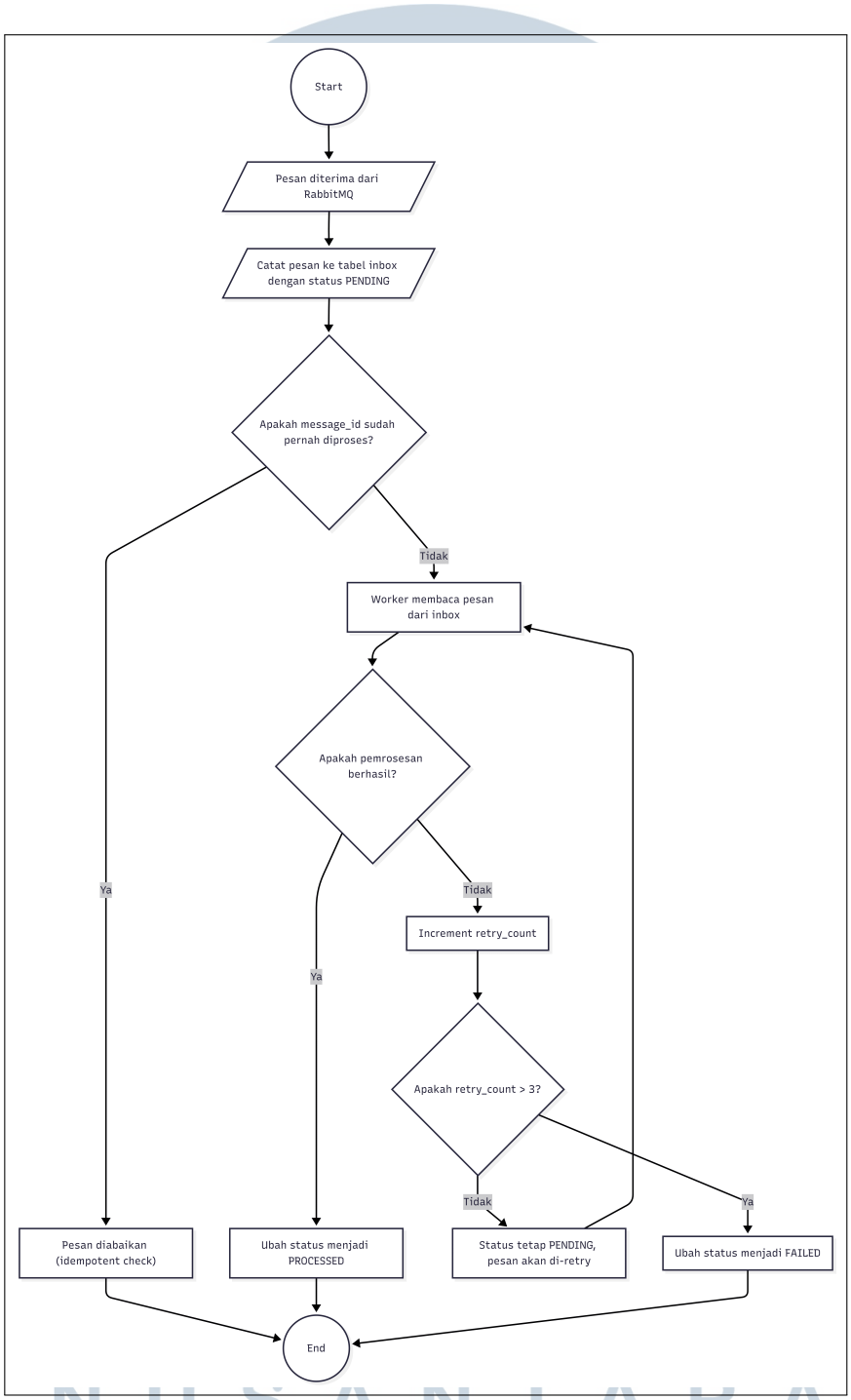
- **PostgreSQL**: Postgres digunakan sebagai *database* utama untuk menyimpan data aplikasi sekaligus mencatat transaksi dalam tabel *outbox*. Dengan memanfaatkan fitur *transactional consistency* dari Postgres, setiap perubahan data dan pencatatan pesan ke *outbox* dapat dilakukan dalam satu transaksi atomik. Hal ini memastikan bahwa data tidak akan hilang atau terduplikasi ketika terjadi kegagalan sistem.[9]
- **RabbitMQ**: RabbitMQ berperan sebagai *message broker* yang menerima pesan dari *outbox* dan mendistribusikannya ke layanan lain. RabbitMQ mendukung komunikasi asinkron antar *microservice*, sehingga pesan dapat dikirim dengan aman dan di-retry jika terjadi kegagalan. Dalam implementasi *inbox-outbox pattern*, RabbitMQ memastikan bahwa pesan yang sudah dicatat di *database* dapat dikirim ke sistem tujuan tanpa kehilangan. developer dapat mengetahui *latency*, *bottleneck*, dan dependensi antar layanan.[10]
- **Golang**: Golang digunakan sebagai bahasa pemrograman untuk mengimplementasikan logika aplikasi, termasuk mekanisme pembacaan *outbox*, pengiriman pesan ke RabbitMQ, serta pemrosesan pesan di *inbox*. Dalam implementasi ini, *worker pool pattern* digunakan untuk membaca data dari *outbox* dan *inbox* secara paralel. *Worker pool* memungkinkan beberapa *worker goroutines* berjalan bersamaan untuk memproses pesan, sehingga meningkatkan *throughput* dan efisiensi sistem. Dengan pendekatan ini, aplikasi dapat menangani beban pesan yang tinggi secara lebih stabil dan terukur, sekaligus memastikan setiap pesan diproses secara konsisten dan idempoten.[11]

B Tabel Inbox

Tabel ini digunakan untuk menyimpan pesan yang diterima dari sistem lain melalui *message broker*. Tujuannya adalah memastikan setiap pesan yang masuk diproses sekali saja *(idempotent processing)* dan dicatat untuk keperluan audit maupun *retry* jika terjadi kegagalan.

1. **id (SERIAL, PK)**: Identitas unik untuk setiap record pesan yang diterima.
2. **sender_id (VARCHAR, NOT NULL)**: Identitas pengirim pesan, misalnya nama layanan atau sistem sumber.
3. **message_id (UUID, UNIQUE, NOT NULL)**: Identitas unik pesan untuk mencegah pemrosesan ganda.
4. **event_type (VARCHAR)**: Jenis event yang diterima, misalnya *InventoryUpdated*.
5. **payload (JSONB)**: Isi data pesan dalam format JSON.
6. **status (message_status, DEFAULT 'PENDING')**: Status pemrosesan pesan, misalnya *PENDING*, *PROCESSED*, atau *FAILED*.
7. **retry_count (INT, DEFAULT 0)**: Jumlah percobaan pemrosesan ulang jika terjadi kegagalan.
8. **exchange (VARCHAR, DEFAULT 'orders')**: Nama exchange di RabbitMQ tempat pesan berasal.
9. **routing_key (VARCHAR)**: Routing key yang digunakan untuk mengarahkan pesan ke layanan penerima.
10. **error (TEXT)**: Informasi error jika pemrosesan pesan gagal.
11. **locked_at (TIMESTAMP)**: Waktu ketika pesan sedang diproses oleh worker.
12. **locked_by (VARCHAR)**: Identitas worker yang sedang memproses pesan.
13. **created_at (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP)**: Waktu pesan pertama kali diterima.
14. **updated_at (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP)**: Waktu terakhir pesan diperbarui.

C Alur Inbox

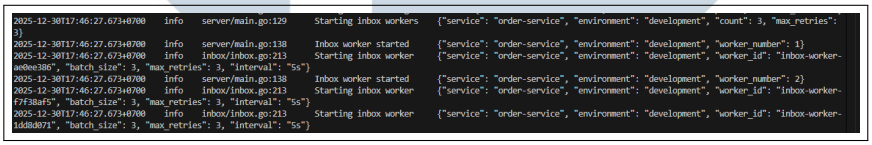


Gambar 3.11. *Flowchart inbox pattern*

Gambar 3.11 menampilkan alur pemrosesan pesan menggunakan *Inbox Pattern*. Alur *Inbox* dimulai ketika sebuah pesan diterima dari RabbitMQ dan

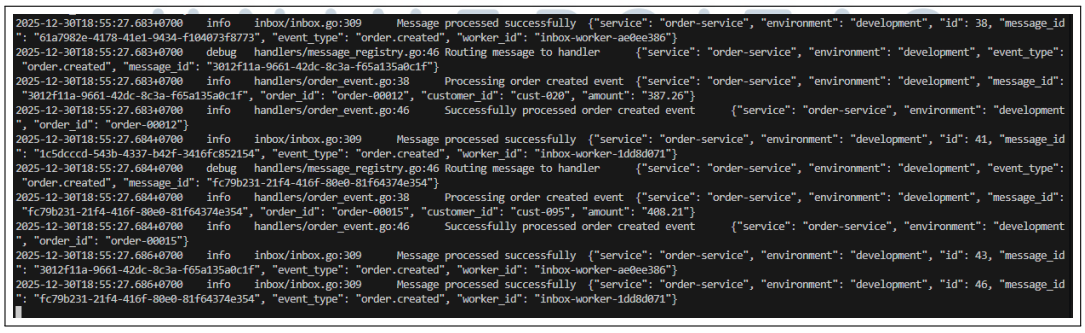
dicatat ke dalam tabel *inbox* dengan status awal `PENDING`. Sebelum diproses, sistem melakukan pengecekan idempotensi berdasarkan `message_id` untuk memastikan bahwa pesan yang sama tidak diproses lebih dari sekali. Jika pesan sudah pernah diproses sebelumnya, maka pesan tersebut akan diabaikan. Jika belum, *worker* akan membaca pesan dari *inbox* dan mencoba memprosesnya. Jika pemrosesan berhasil, status pesan akan diperbarui menjadi `PROCESSED`. Namun, jika pemrosesan gagal, sistem akan menambahkan nilai `retry_count` dan mencoba memproses ulang pesan hingga maksimal tiga kali. Apabila setelah tiga kali percobaan pesan tetap gagal diproses, maka status pesan akan diubah menjadi `FAILED`. Dengan mekanisme ini, *inbox* menjamin bahwa setiap pesan yang diterima dari sistem lain diproses secara konsisten, aman dari duplikasi, dan memiliki keandalan tinggi dalam menghadapi kegagalan.

D Hasil Implementasi Inbox



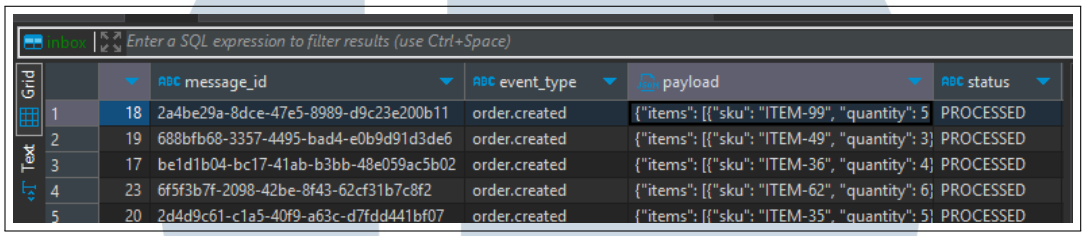
Gambar 3.12. Inisialisasi *inbox worker*

Gambar 3.12 menunjukkan proses inisialisasi *inbox worker*. *Log* mencatat bahwa sistem memulai beberapa *worker* untuk memproses pesan masuk secara paralel. Inisialisasi ini merupakan bagian dari implementasi *Inbox-Outbox pattern* yang bertujuan untuk memastikan setiap pesan yang diterima dapat diproses secara konsisten dan tidak hilang, sehingga mendukung integritas komunikasi antar layanan.



Gambar 3.13. *Log* pemrosesan pesan oleh *inbox worker*

Gambar 3.13 menampilkan *log* aktivitas dari *inbox worker* yang berhasil menerima dan memproses pesan masuk. Setiap pesan memiliki atribut seperti `message_id`, `event_type`, dan `worker_id`, yang menunjukkan bahwa event yang diterima telah diproses sesuai dengan alur bisnis. Proses ini memastikan bahwa pesan yang masuk dari *queue* atau *exchange* dapat ditangani dengan baik oleh sistem.



Gambar 3.14. Hasil *message* di tabel *outbox* setelah diproses

Gambar 3.14 memperlihatkan isi dari *inbox table* setelah pesan berhasil diproses. Setiap baris merepresentasikan satu event dengan informasi lengkap seperti `message_id`, `event_type`, `payload`, `status`, dan `retry_count`. Seluruh pesan dalam tabel memiliki status `PROCESSED` dan nilai `retry_count` sebesar 0, yang menandakan bahwa pesan telah diterima dan diproses tanpa kesalahan.

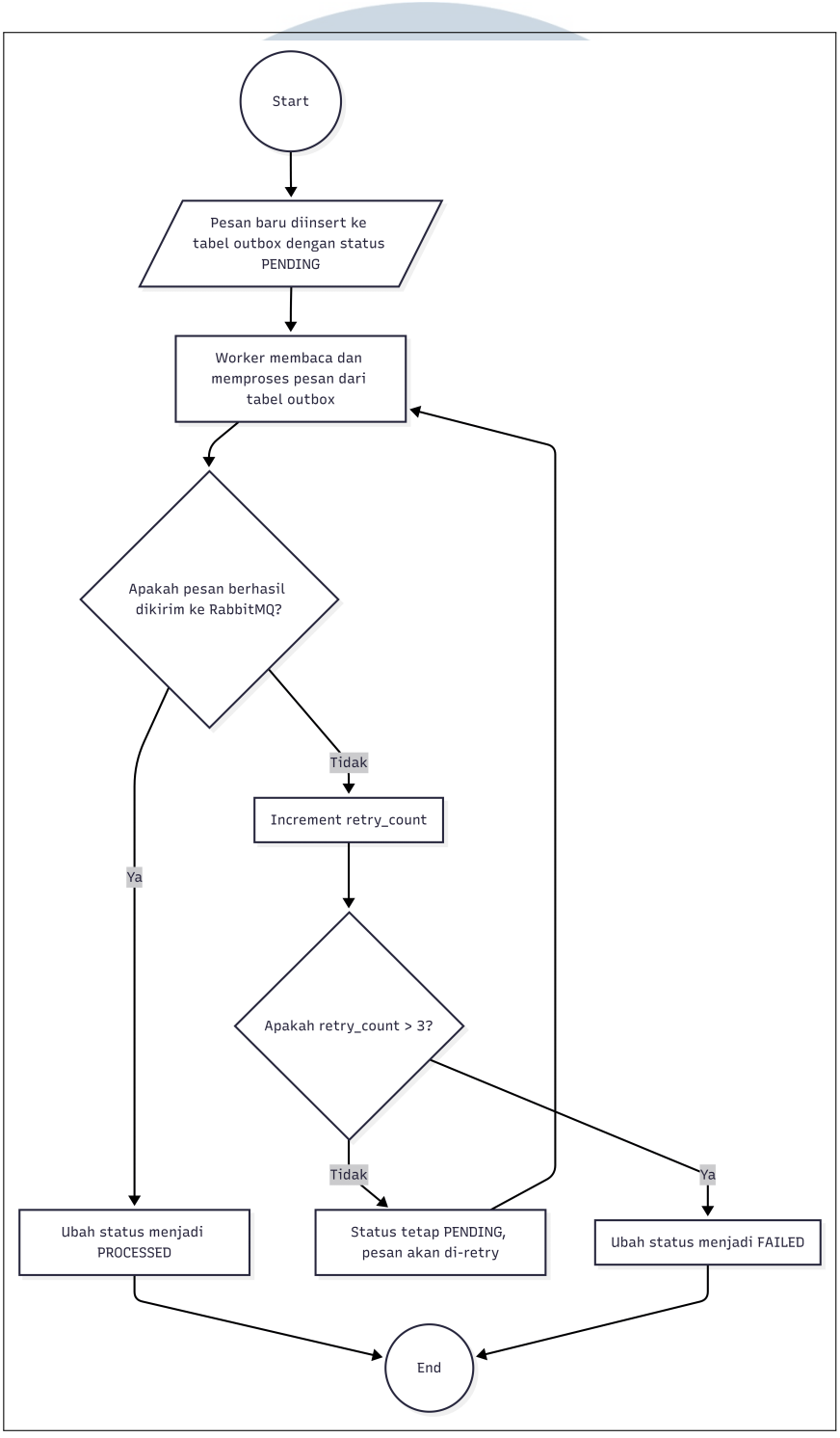
E Tabel Outbox

Tabel ini digunakan untuk menyimpan pesan yang akan dikirim ke sistem lain melalui *message broker* (RabbitMQ). Tujuannya adalah memastikan setiap perubahan data yang terjadi di aplikasi tercatat terlebih dahulu sebelum dikirim, sehingga mendukung konsistensi dan keandalan sistem.

1. **id (SERIAL, PK)**: Identitas unik untuk setiap record pesan.
2. **message_id (UUID, UNIQUE, NOT NULL)**: Identitas unik pesan agar tidak terjadi duplikasi.
3. **event_type (VARCHAR)**: Jenis event yang terjadi, misalnya *OrderCreated*.
4. **payload (JSONB)**: Isi data pesan dalam format JSON.
5. **status (message_status, DEFAULT 'PENDING')**: Status pesan, misalnya *PENDING*, *SENT*, atau *FAILED*.

6. **retry_count (INT, DEFAULT 0)**: Jumlah percobaan pengiriman ulang jika terjadi kegagalan.
7. **exchange (VARCHAR, DEFAULT 'orders')**: Nama exchange di RabbitMQ tempat pesan akan dikirim.
8. **routing_key (VARCHAR)**: Routing key untuk menentukan tujuan pesan.
9. **error (TEXT)**: Informasi error jika pengiriman pesan gagal.
10. **locked_at (TIMESTAMP)**: Waktu ketika pesan sedang diproses oleh worker.
11. **locked_by (VARCHAR)**: Identitas worker yang sedang memproses pesan.
12. **created_at (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP)**: Waktu pesan pertama kali dibuat.
13. **updated_at (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP)**: Waktu terakhir pesan diperbarui.

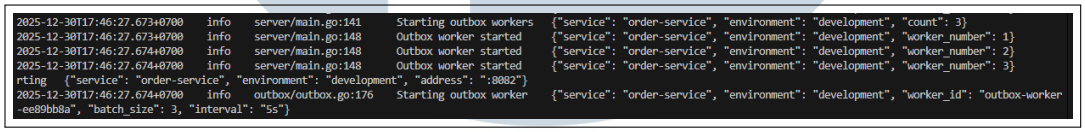
F Alur Outbox



Gambar 3.15. *Flowchart outbox pattern*

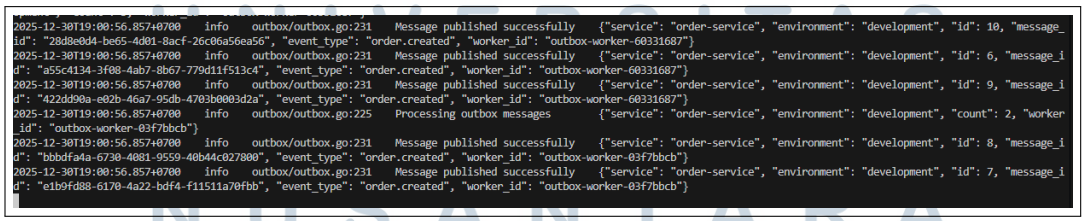
Gambar 3.15 menampilkan alur pemrosesan pesan menggunakan *Outbox Pattern*. Alur *Outbox* dimulai ketika aplikasi mencatat sebuah pesan baru ke dalam tabel *outbox* dengan status awal `PENDING`. *Worker* kemudian membaca pesan tersebut dan mencoba mengirimkannya ke RabbitMQ. Jika pengiriman berhasil, status pesan akan diperbarui menjadi `SENT` sebagai tanda bahwa pesan telah terkirim dengan baik. Namun, jika pengiriman gagal, sistem akan menambahkan nilai *retry_count* dan mencoba mengirim ulang pesan. Proses *retry* ini dilakukan hingga maksimal tiga kali. Apabila setelah tiga kali percobaan pesan tetap gagal dikirim, maka status pesan akan diubah menjadi `FAILED`. Dengan mekanisme ini, *Outbox* memastikan bahwa setiap perubahan data yang terjadi di aplikasi tidak hilang begitu saja, melainkan selalu dicatat dan diusahakan untuk dikirim ke sistem tujuan secara konsisten.

G Hasil Implementasi Outbox



Gambar 3.16. Inisialisasi *outbox worker*

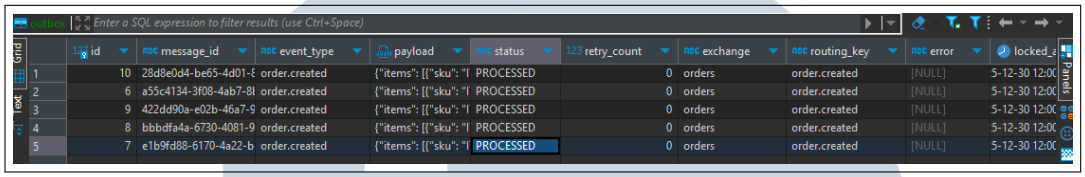
Gambar 3.16 menunjukkan proses inisialisasi *outbox worker* pada layanan dalam lingkungan pengembangan. *Log* mencatat bahwa sistem memulai tiga *worker* secara paralel, masing-masing ditandai dengan nomor urut dan informasi lingkungan. Inisialisasi ini merupakan bagian dari implementasi *Inbox-Outbox pattern* yang bertujuan untuk memproses pesan secara asinkron dari *outbox table*, sehingga menjamin keandalan pengiriman event antar layanan tanpa mengganggu transaksi utama.



Gambar 3.17. *Log* pemrosesan pesan oleh *outbox worker*

Gambar 3.17 menampilkan *log* aktivitas dari *outbox worker* yang berhasil memproses dan menerbitkan pesan dari *outbox table*. Setiap pesan memiliki atribut

seperti `message_id`, `event_type`, dan `worker_id`, yang menunjukkan bahwa event `order.created` telah berhasil dikirim oleh masing-masing *worker*. Proses ini menunjukkan bahwa pesan dikirim secara terjamin dan dapat ditelusuri melalui *log*.



	id	message_id	event_type	payload	status	retry_count	exchange	routing_key	error	locked_a
1	10	28d8e0d4-be65-4d01-8	order.created	{"items": [{"sku": "I	PROCESSED	0	orders	order.created	[NULL]	5-12-30 12:0(
2	6	a55c4134-3f08-4ab7-8l	order.created	{"items": [{"sku": "I	PROCESSED	0	orders	order.created	[NULL]	5-12-30 12:0(
3	9	422dd90a-e02b-46a7-9	order.created	{"items": [{"sku": "I	PROCESSED	0	orders	order.created	[NULL]	5-12-30 12:0(
4	8	bbbdfa4a-6730-4081-9	order.created	{"items": [{"sku": "I	PROCESSED	0	orders	order.created	[NULL]	5-12-30 12:0(
5	7	e1b9fd88-6170-4a22-b	order.created	{"items": [{"sku": "I	PROCESSED	0	orders	order.created	[NULL]	5-12-30 12:0(

Gambar 3.18. Hasil *message* di tabel *outbox* setelah diproses

Gambar 3.18 memperlihatkan isi dari *outbox table* setelah pesan berhasil diproses. Setiap baris merepresentasikan satu event dengan informasi lengkap seperti `message_id`, `event_type`, `payload`, `status`, dan `retry_count`. Seluruh pesan dalam tabel memiliki status `PROCESSED` dan nilai `retry_count` sebesar 0, yang menandakan bahwa pesan telah dikirim tanpa kesalahan dan tidak memerlukan pengulangan.

H Black Box Testing Inbox-Outbox

Tabel 3.5. Black Box Testing Inbox-Outbox

No	Langkah Uji	Input	Output yang Diharapkan	Status
1	Simpan data dan kirim event	Payload JSON	Record muncul di tabel outbox	Lulus
2	Jalankan worker outbox	Status PENDING	Pesan terkirim, status jadi PROCESSED	Lulus
3	Jalankan worker inbox	Status PENDING	Pesan terproses, status jadi PROCESSED	Lulus
4	Simulasikan error RabbitMQ	Status PENDING	retry_count bertambah	Lulus
5	Simulasikan error PostgreSQL	Status PENDING	retry_count bertambah	Lulus
6	Simulasikan error berulang	retry_count = 4	Status berubah jadi FAILED	Lulus
7	Terima pesan dari RabbitMQ	Payload JSON	Record muncul di tabel inbox	Lulus
8	Kirim pesan dengan message_id sama	Duplikat message_id	Pesan kedua diabaikan (idempotent check)	Lulus

Tabel 3.5 menunjukkan hasil *black box testing* terhadap implementasi *inbox-outbox pattern* menggunakan Golang. Pelaksanaan pengujian dilakukan secara kolaboratif oleh Supervisor yang menjabat sebagai Senior Software Engineer.

3.4 Kendala dan Solusi yang Ditemukan

3.4.1 Kendala yang Ditemukan

Selama pelaksanaan kerja magang, terdapat sejumlah kendala dan tantangan yang dihadapi, antara lain sebagai berikut:

1. **Keterbatasan pemahaman terhadap teknologi baru**: Pada awal pelaksanaan magang, belum sepenuhnya familiar dengan teknologi yang digunakan, seperti Golang, RabbitMQ, serta *observability tools* (Prometheus, Grafana, Loki, dan Jaeger). Hal ini menyebabkan adanya keterlambatan dalam memahami konsep dasar dan praktik terbaik yang diperlukan untuk implementasi.

2. **Pengerjaan paralel dengan proyek lain**: Selain mengerjakan proyek *observability*, terdapat pula keterlibatan dalam pengerjaan proyek lain yang sedang berjalan di perusahaan. Kondisi ini menimbulkan tantangan dalam manajemen waktu dan fokus, karena harus membagi perhatian antara beberapa pekerjaan sekaligus.

3. **Bug pada tahap implementasi**: Dalam proses implementasi, beberapa *bug* ditemukan, baik berupa kesalahan logika, konfigurasi yang tidak sesuai, maupun *error* saat integrasi antar komponen. *Bug* ini menghambat jalannya pengembangan dan memerlukan waktu tambahan untuk melakukan *debugging* serta perbaikan.

3.4.2 Solusi yang Diterapkan

Untuk mengatasi kendala dan tantangan yang dihadapi selama pelaksanaan kerja magang, berikut ini adalah solusi yang diterapkan:

1. **Melakukan pembelajaran mandiri dan mentoring**: Pembelajaran mandiri dilakukan melalui dokumentasi resmi, tutorial, serta diskusi dengan Senior Software Engineer. Dengan adanya mentoring, pemahaman terhadap teknologi yang digunakan dapat diperoleh lebih cepat dan diterapkan dalam proyek.

2. **Menerapkan manajemen waktu dan prioritas**: Untuk mengatasi pekerjaan yang dibarengi dengan proyek lain, dibuat jadwal kerja harian dan ditetapkan prioritas tugas. Dengan cara ini, pekerjaan dapat diselesaikan sesuai target tanpa mengorbankan kualitas hasil.

3. ***Bug* pada tahap implementasi**: Setiap *bug* yang ditemukan ditangani dengan proses *debugging* yang terstruktur, seperti melakukan *logging*, *tracing*, serta pengujian ulang dengan berbagai skenario. *Testing* dilakukan

secara berulang untuk memastikan bahwa perbaikan yang diterapkan benar-benar menyelesaikan masalah dan tidak menimbulkan *bug* baru.