Perancangan Sistem Monitoring Performa Aplikasi Menggunakan Opentelemetry dan Grafana Stack

Guntoro Yudhy Kusuma[1], Unan Yusmaniar Oktiawati[1,*]
[1]Departemen Teknik Elektro dan Informatika, Sekolah Vokasi, Universitas Gadjah Mada;
guntoro.y.k@mail.ugm.ac.id
[*]Korespondensi: unan_yusmaniar@ugm.ac.id;

Abstract – The increasingly massive use of digital technology requires that the application architecture be designed to have high availability and reliability. This is because when an application cannot be accessed, it will cause no small loss to the organization. Therefore, the development and operation teams must be able to detect when their system is not working well. For that, we need a system that can monitor application performance. In this research, a system is developed to collect telemetry data, namely metrics and traces from an online donation backend application based on the REST API. OpenTelemetry produces telemetry as an open-source telemetry instrumentation tool. Then the telemetry data is collected by the OpenTelemetry Collector which is then stored on the backend of each telemetry. Metrics are sent to Prometheus and traces are sent to Jaeger. The data metrics collected are throughput, request latency, and error rate which are visualized using the Grafana dashboard. The test results show that the monitoring system can collect real-time metrics data with an average delay of 13,8 seconds. The system can also detect when an anomaly occurs in the app and sends notifications via Slack. In addition, the trace data collected can be used to simplify the debugging process when an error occurs in the application. However, the implementation of OpenTelemetry in a REST API-based backend application to monitor metrics and traces has a negative impact on the performance of the application itself, which can reduce the number of request throughput with an average decrease of 23.32% and increase request latency with an average increase of 22.80%.

Keywords - monitoring, APM, system observability, OpenTelemetry, Grafana

Intisari – Semakin masifnya penggunaan teknologi digital mengharuskan arsitektur aplikasi didesain agar memiliki ketersediaan dan keandalan yang tinggi. Hal ini karena ketika sebuah aplikasi tidak dapat diakses, akan menyebabkan kerugian yang tidak sedikit bagi organisasi. Oleh karena itu tim *developer* maupun *operation* harus bisa mendeteksi ketika sistem mereka sedang tidak baik-baik saja. Untuk itulah diperlukan sebuah sistem yang dapat memonitor performa aplikasi. Pada penelitian kali ini dikembangkan sebuah sistem yang dapat mengumpulkan data telemetri yaitu *metrics* dan *traces* dari sebuah aplikasi *backend* donasi *online* yang berbasis *REST API*. Telemetri tersebut dihasilkan oleh *OpenTelemetry* sebagai alat instrumentasi telemetri yang *open-source*. Kemudian data telemetri tersebut dikumpulkan oleh *OpenTelemetry Collector* yang selanjutnya disimpan pada *backend* masing-masing telemetri. *Metrics* dikirimkan menuju Prometheus dan *traces* dikirimkan ke *Jaeger*. Data *metrics* yang dikumpulkan adalah *throughput, request latency,* dan *error rate* yang divisualisasikan menggunakan *Grafana dashboard*. Hasil pengujian menunjukkan bahwa sistem monitoring dapat mengumpulkan data *metrics* secara *realtime* dengan waktu tunda rata-rata 13,8 detik. Sistem juga dapat mendeteksi ketika terjadi anomali pada aplikasi dan mengirimkan pemberitahuan melalui *Slack*. Selain itu, data *traces* yang dikumpulkan dapat digunakan untuk mempermudah proses *debugging* ketika terjadi kesalahan pada aplikasi. Namun, implementasi *OpenTelemetry* dalam aplikasi *backend* berbasis *REST API* untuk memonitor metrics dan *traces* memberikan dampak negatif pada performa aplikasi itu sendiri yaitu dapat menurunkan jumlah *request throughput* dengan penurunan rata-rata 23,32% dan menaikkan *request latency* dengan kenaikan rata-rata 22,80%.

Kata kunci – *monitoring, APM, system observability, OpenTelemetry, Grafana*

I. PENDAHULUAN

Semakin masifnya penggunaan teknologi digital mengharuskan arsitektur aplikasi didesain agar memiliki ketersediaan dan keandalan yang tinggi. Hal ini karena ketika sebuah aplikasi tidak dapat diakses, akan menyebabkan kerugian yang tidak sedikit bagi organisasi. Sebagai contoh, *Amazon* menemukan setiap 100 milidetik latensi membuat mereka kehilangan 1% dalam penjualan. *Google* menemukan tambahan 5% dalam waktu pembuatan halaman hasil pencarian menurunkan lalu lintas sebesar 20%. Sebuah pialang saham dapat kehilangan pendapatan sebesar *$4* per milidetik jika platform perdagangan elektronik mereka tertinggal 5 milidetik dari pesaingnya [1].

Untuk memastikan bahwa sistem aplikasi memberikan kualitas layanan yang diharapkan, sangat penting untuk memiliki informasi terkini tentang sistem untuk mendeteksi masalah dan menyelesaikannya secara efektif [2]. Informasi yang diperlukan tersebut berupa data telemetri *metrics*, *traces*, dan *logs* yang dikumpulkan dari berbagai komponen pada sebuah sistem seperti infrastrukutur, sistem operasi, *database*, dan aplikasi [3].

Pada penelitian kali ini dikembangkan sebuah sistem *monitoring* yang akan mengumpulkan telemetri berupa *metrics* dan *traces* dari sebuah backend sistem donasi online yang berbasis *REST API*. Telemetri tersebut dihasilkan oleh *OpenTelemetry* sebagai alat instrumentasi telemetri yang *open-source* dan kemudian dikumpulkan oleh *OpenTelemetry Collector*. Selanjutnya, telemetri berupa *metrics* dikirimkan menuju alat *monitoring* dan *alerting Prometheus* yang kemudian dapat divisualisasikan menggunakan *Grafana* secara *realtime*. Telemetri berupa *traces* akan diteruskan oleh *OpenTelemetry Collector* menuju ke *Jaeger* yang diintegrasikan pada *Grafana*. Sistem *monitoring* menggunakan data *metrics* yang dikumpulkan untuk mendeteksi terjadinya anomali pada aplikasi kemudian memberikan peringatan melalui *Slack*. Kemudian, data *traces*

yang dikumpulkan dapat digunakan untuk mempermudah proses *debugging* ketika terjadi kesalahan pada aplikasi.

II. DASAR TEORI

A. *Observability* dan *Monitoring*

Menurut JJ Tang pada *blogpost* di situs *devops.com*, dalam industri TI secara khusus, *observability* dapat didefinisikan sebagai penggunaan *logs*, *metrics*, dan *traces* untuk memahami keadaan sistem perangkat lunak yang kompleks. Sedangkan *monitoring* adalah proses pengumpulan data dari sebuah sistem. Di industri TI, *monitoring* adalah bagaimana tim mengumpulkan *logs*, *metrics*, dan *traces* dan memindahkannya ke tempat di mana mereka dapat dianalisis [4].

Di sini dapat dipahami bahwa *observability* dan *monitoring* adalah dua hal yang berbeda tetapi saling berkaitan. *Observability* berfokus pada proses menginterpretasi dan memahami data, sedangkan *monitoring* hanyalah proses pengumpulan data tersebut. Dengan adanya *monitoring* kita mendapatkan peringatan ketika terjadi sesuatu yang salah pada sistem. Kemudian dengan *observability* kita dapat mengetahui mengapa sesuatu tersebut bisa salah dan bagaimana untuk memperbaikinya.

B. *Application Performance Management (APM)*

*Application Perfromance Managemen*t (*APM*) adalah disiplin operasi inti TI yang bertujuan untuk mencapai tingkat kinerja yang memadai selama operasi. APM terdiri dari metode, teknik, dan alat untuk i) terus memantau keadaan sistem aplikasi dan penggunaannya, serta untuk ii) mendeteksi, mendiagnosis, dan menyelesaikan masalah terkait kinerja menggunakan data yang dipantau. Perlu ditekankan bahwa dalam konteks *APM*, gagasan kinerja mencakup seperangkat properti non-fungsional yang komprehensif (misalnya, ketepatan waktu, penggunaan sumber daya, keandalan, ketersediaan, dan keamanan) dan bahkan mungkin aspek fungsional [2].

C. *OpenTelemetry*

OpenTelemetry adalah seperangkat *API*, *SDK*, perkakas, dan integrasi yang dirancang untuk pembuatan dan pengelolaan data telemetri seperti *logs*, *metrics*, dan *traces*. Proyek ini menyediakan implementasi *vendor-agnostic* yang dapat dikonfigurasi untuk mengirim data telemetri ke *backend* pilihan seperti *Jaeger* dan *Prometheus*. Data telemetri dibutuhkan untuk memperkuat observabilitas sebuah sistem. Secara tradisional, data telemetri disediakan oleh vendor-vendor komersial maupun *open-source*. Namun, terdapat kurangnya standardisasi dari berbagai vendor tersebut sehingga menyebabkan kurangnya portabilitas data dan kesulitan dalam memelihara instrumentasi. Oleh karena itu, *OpenTelemetry* hadir untuk menyelesaikan masalah tersebut dengan menyediakan satu solusi yang *vendor-agnostic*.
Beberapa hal yang diberikan oleh *OpenTelemetry*:

- Pustaka instrumentasi vendor-agnostik tunggal per bahasa pemrograman dengan dukungan untuk instrumentasi otomatis dan manual.
- Biner kolektor tunggal (*OpenTelemetry Collector*) yang dapat digunakan dalam berbagai cara termasuk sebagai agen atau *gateway*.
- Implementasi ujung ke ujung untuk menghasilkan, memancarkan, mengumpulkan, memproses, dan mengekspor data telemetri.
- Kontrol penuh terhadap data kita dengan kemampuan untuk mengirimkannya ke berbagai tujuan secara paralel melalui konfigurasi.
- Konvensi semantik standar terbuka untuk memastikan pengumpulan data vendor-agnostik.
- Kemampuan untuk mendukung berbagai format propagasi konteks secara paralel untuk membantu migrasi seiring berkembangnya standar
- Dengan dukungan terhadap beragam protokol komersial maupun *open-source* membuat *OpenTelemetry* mudah untuk diadopsi [5].

D. *Prometheus*

Prometheus adalah sebuah alat *monitoring* dan *alerting* bersifat *open source* yang pada awalnya dibangun untuk *SoundCloud*. Sejak dimulai pada tahun 2012, banyak perusahaan dan organisasi telah mengadopsi *Prometheus*, dan proyek ini memiliki komunitas pengembang dan pengguna yang sangat aktif. Sekarang *Prometheus* merupakan proyek *open-source* mandiri dan dikelola secara independen dari perusahaan mana pun. Untuk menekankan hal ini, dan untuk memperjelas struktur tata kelola proyek, *Prometheus* bergabung dengan *Cloud Native Computing Foundation* pada tahun 2016 sebagai proyek yang di-*hosting* kedua setelah *Kubernetes*.

Prometheus mengumpulkan dan menyimpan metriknya sebagai data deret waktu (*time series data*), yaitu informasi metrik disimpan dengan stempel waktu saat direkam, bersama pasangan nilai kunci opsional yang disebut label. Fitur-fitur utama *Prometheus* adalah:

- Sebuah data model multidimensi dengan data deret waktu (*time series data*) yang diidentifikasi menggunakan nama metrik dan pasangan *key/value*.
- *PromQL*, bahasa *query* yang fleksibel untuk memanfaatkan data model multidimensi tersebut.
- Tidak bergantung pada penyimpanan terdistribusi, *node server* tunggal bersifat otonom.
- Pengumpulan data *time series* dilakukan dengan model *pull* melalui protokol *HTTP*.
- Dukungan model *push* untuk pengumpulan *data time series* melalui *intermediary gateway*.
- Target yang akan diambil datanya diatur melalui konfigurasi statis atau menggunakan *service discovery*.
- Dukungan berbagai mode grafik dan *dashboard* [6].

E. *Jaeger*

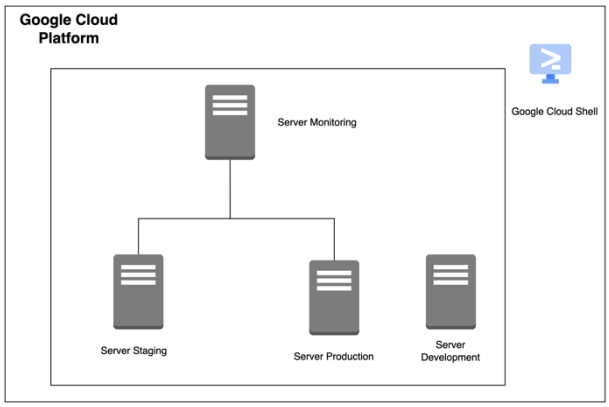
Jaeger yang merupakan *backend* penelusuran (*traces*) terdistribusi yang bersifat *open-source*. *Jaeger* dapat digunakan untuk memantau dan melakukan *troubleshooting* pada sistem terdistribusi. *Jaeger* ini juga mendukung data telemetri *traces* yang dihasilkan oleh *OpenTelemetry API* [7].

F. *Grafana Stack*

Grafana adalah *stack* observabilitas lengkap yang memungkinkan untuk memantau dan menganalisis *metrics*, *logs*, dan *traces*. *Grafana* memungkinkan untuk melakukan *query*, memvisualisasikan, membuat peringatan, dan memahami data telemetri di manapun data tersebut disimpan. *Grafana* mendukung berbagai sumber data seperti *Prometheus*, *Graphite*, *InfluxDB*, *ElasticSearch*, *MySQL*, *PostreSQL*, *dll* [8].

III. METODOLOGI

A. Perancangan *Server*



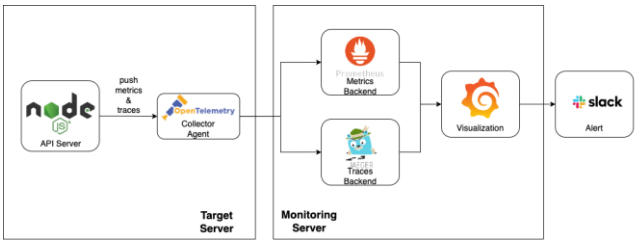
Gambar 1. Perancangan *Server*

Pada Gambar 1 merupakan perancangan *server* yang digunakan untuk lingkungan pengujian pada penelitian ini. Terdapat empat buah *Virtual Machine* dengan sistem operasi *Ubuntu 20.04* yang terpasang pada *Google Cloud Platform*. Masing-masing *VM* tersebut adalah satu *VM* sebagai *server monitoring*, dua *VM* sebagai *server staging* dan *production* yang akan menjadi target *monitoring*, dan satu *VM* sebagai *server development* yang digunakan untuk memasang aplikasi tanpa implementasi *OpenTelemetry* di dalamnya.

B. Perancangan Sistem Monitoring

Sistem *monitoring* terdiri dari dua komponen yaitu *Server Monitoring* dan Target *Monitoring*. Untuk target *monitoring* sendiri terdapat dua *server* yaitu *server staging* dan *server production*. Pada masing-masing target server terdapat beberapa *backend* sistem donasi *online* berbasis *REST API* yang dibangun menggunakan *NodeJS* yang sudah diintegrasikan dengan *OpenTelemetry API*. Selain itu, pada masing-masing target *server* juga terdapat *OpenTelemetry Collector* sebagai agen yang akan mengumpukan *metrics* dan *traces* kemudian mengirimkannya ke *Prometheus* dan *Jaeger*.

Kemudian pada *server monitoring* terpasang *server Prometheus Server* yang bertugas untuk menyimpan dan mengolah data telemetri berupa *metrics*, *Jaeger Server* yang bertugas untuk menyimpan dan mengolah data telemetri berupa *traces*, dan *Grafana Server* yang bertugas untuk visualisasi data. Terakhir terdapat *Slack* yang menjadi saluran peringatan ketika sistem *monitoring* mendeteksi adanya anomali pada data-data telemetri.



Gambar 2. Rancangan Sistem *Monitoring*

C. Skenario Pengujian

Pengujian yang digunakan dalam penelitian ini terdapat 4 metode. Metode yang digunakan adalah pengujian fungsionalitas *monitoring* dan metrics, fungsi *monitoring error rate*, pendeteksian anomali, serta pengujian performa.

1) Pengujian fungsionalitas *monitoring* data *metrics*

a. Pengujian fungsi monitoring *throughput* dan *request latency*

Pengujian ini bertujuan untuk menguji apakah sistem *monitoring* dapat mengumpulkan data *metric* berupa *throughput* dan *request latency* dan mengukur *delay* pengirimannya. Pada pengujian ini dilakukan total 10 kali percobaan *load test* terhadap *server REST API* pada *endpoint GET /v1/campaigns*. Percobaan dilakukan pada *server staging* dan *production* di mana setiap *server* mendapat bagian 5 percobaan. *Load test* dilakukan menggunakan alat bantu *Vegeta* yang dijalankan melalui *Google Cloud Shell* untuk memberikan simulasi pemanggilaan *endpoint* tersebut selama 3 menit dengan *request rate* sebesar 10 *rps*, 15 *rps*, 20 *rps*, 25 *rps*, dan 30 *rps*.

Perintah untuk menjalankan *load test* adalah sebagai berikut:

- *Production*
  echo "GET http://34.128.121.13:5000/v1/campaigns" | vegeta attack -duration=3m -rate=10 | vegeta report
- *Staging*
  echo "GET http://34.128.100.99:5000/v1/campaigns" | vegeta attack -duration=3m -rate=10 | vegeta report

b. Pengujian fungsi *monitoring error rate*

Pada pengujian ini dilakukan empat kali *load test* pada *server REST API staging* dan *production*. *Load test* dilakukan dengan cara melakukan pemanggilan *endpoint GET /v1/campaigns* untuk simulasi *request* berhasil sebanyak 10 *rps* selama 5 menit. Kemudian melakukan pemanggilan

endpoint GET /v1/error untuk simulasi *request* gagal karena *Internal Server Error* sebanyak 5 *rps* selama 5 menit.

2) Pengujian fungsionalitas *monitoring data traces*

Pengujian ini dilakukan dengan cara mematikan *server database* pada *server production* dan *staging* kemudian dilakukan uji coba *login* menggunakan *Postman* ke *REST API production* dan *staging*. Dikarenakan *server database* mati sehingga pengguna yang melakukan *login* akan mendapatkan respons *Internal Server Error*. Pengujian dikatakan berhasil jika sistem *monitoring* dapat menunjukkan bahwa penyebab pengguna tidak bisa melakukan *login* adalah aplikasi gagal terhubung ke *server database*.

3) Pengujian Deteksi Anomali

a. Error Rate

Pengujian ini bertujuan untuk menguji kemampuan sistem dalam mendeteksi *error rate* lebih dari 80%. Nilai *error rate* merupakan perbandingan dari jumlah *request* yang mendapakan *response* 500 dibandingkan dengan total seluruh *request*. Pada pengujian ini dilakukan pemanggilan sebanyak 10 *rps* ke *endpoint GET /v1/campaigns* pada *REST API production* dan *staging* selama 10 menit. Setelah 2 menit berjalan maka *server database* pada masing-masing *environment* akan dimatikan sehingga akan mendapatkan *response* 500 *Internal Server Error*. Hal ini menyebabkan *error rate* menjadi semakin meningkat. Pengujian dikatakan berhasil apabila sistem dapat mengirimkan notifikasi peringatan menuju *Slack* ketika *error rate* bernilai lebih dari 80% selama minimal 30 detik.

b. Throughput

Pengujian ini bertujuan untuk menguji kemampuan sistem dalam mendeteksi adanya penurunan rata-rata *throughput* dalam satu menit terakhir lebih dari 50% dibandingkan dengan rata-rata *throughput* 5 menit kebelakang. Pada pengujian ini dilakukan *load test* ke *endpoint GET /v1/campaigns* pada *REST API production* dan *staging* dengan ketentuan sebagai berikut:

- 10 *rps* selama 5 menit
- 4 *rps* selama 2 menit
- 10 *rps* selama 5 menit

Pengujian dikatakan berhasil apabila sistem dapat mengirimkan notifikasi peringatan menuju *Slack* ketika *throughput* turun lebih dari 50% yang disebabkan oleh perubahan jumlah *rps* dari 10 *rps* menjadi 4 *rps*. Selain itu, sistem juga harus mampu memberikan pemberitahuan apabila jumlah *rps* kembali normal yang disebabkan oleh jumlah rps kembali menjadi 10 *rps* selama 5 menit.

c. Request Latency

Pada pengujian ini dilakukan pemanggilan sebanyak 20 *rps* ke *endpoint GET /v1/campaigns* pada *REST API production* dan *staging* selama 2 menit. Pengujian dikatakan berhasil apabila sistem dapat mengirimkan notifikasi peringatan menuju Slack ketika *request latency* bernilai lebih dari 80 milidetik.

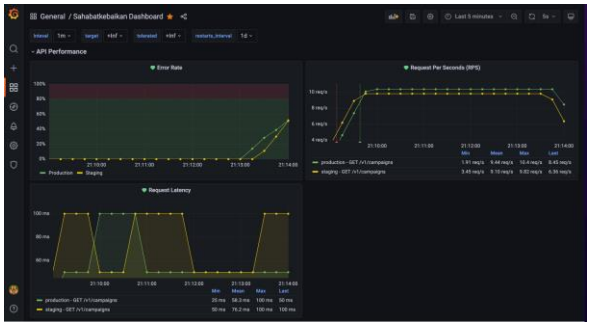
4) Pengujian Performa Aplikasi

Pengujian ini bertujuan untuk mengetahui bagaimana pengaruh implementasi monitoring *metrics* dan *traces* menggunakan *OpenTelemetry* terhadap performa dari aplikasi itu sendiri. Pada topologi akan ditambahkan satu buah *development server* yang berisi *backend* sistem donasi berbasis *REST API* tanpa implementasi *OpenTelemetry* di dalamnya. Pengujian *load test* dilakukan sebanyak 5 kali untuk *server production* dan *development*. Pengujian pertama dilakukan dengan melakukan pemanggilan ke *endpoint GET /v1/campaigns* menggunakan satu buah *user* (*thread*), pengujian kedua menggunakan dua buah *user* (*thread*), dan seterusnya hingga pengujian kelima menggunakan lima buah *user* (*thread*). Setiap pengujian akan berlangsung selama 5 menit.

IV. HASIL DAN PEMBAHASAN

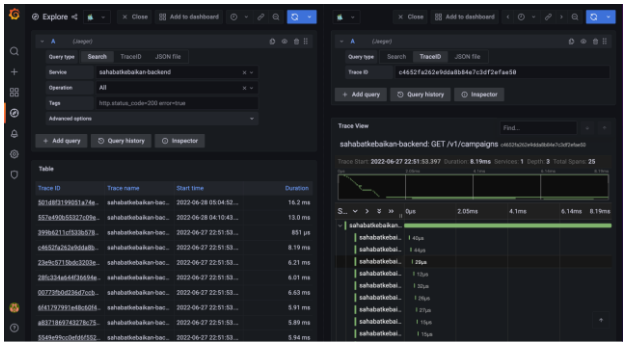
A. Tampilan Antar Muka

Berikut ini adalah tampilan antarmuka dashboard *Grafana* sistem *monitoring* yang telah dibuat. Dalam *dashboard* ini terdapat panel grafik *Error Rate* untuk memantau persentase jumlah *request error* per *endpoint*, panel *Request Per Second (RPS)* untuk memantau jumlah *request* per *endpoint* setiap detiknya, dan panel *Request Latency* untuk memantau *latency* setiap *request* per *endpoint*.



Gambar 3. Tampilan *dashboard Grafana*

Kemudian berikut ini adalah tampilan antarmuka *Jaeger* yang diintegrasikan pada *Grafana* melalui menu *Explore*. Menu ini dapat digunakan untuk membaca data-data *traces* yang telah dikumpulkan.



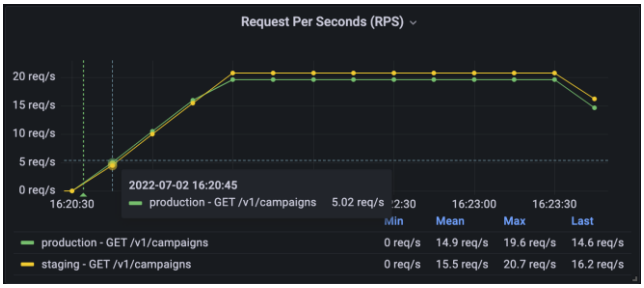
Gambar 4. Tampilan *Jaeger* di *Grafana*

B. Hasil Pengujian Sistem

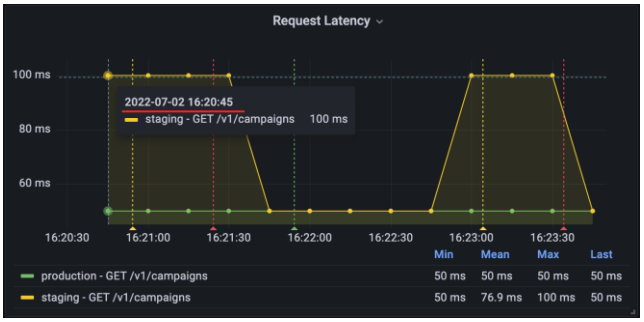
1) Pengujian *Monitoring* Data Metrics

a. Pengujian *Monitoring Throughput* dan *Request Latency*

Selama proses *load test* berlangsung, dilakukan pemantuan pada *dashboard* Grafana untuk panel *Request Per Seconds (RPS)* dan *Request Latency*. Melalui panel ini juga dilihat kapan *metrics* pertama kali diterima untuk masing-masing pengujian. Gambar 3 dan Gambar 4 adalah contoh tampilan dari pemantauan melalui *dashboard Grafana*.



Gambar 5. Contoh tampilan pemantauan panel *Request Per Seconds (RPS)*



Gambar 6. Contoh pemantauan panel *Request Latency*

Dari hasil pengujian *load test* dan pemantauan pada *dashboard Grafana* serta memperhatikan laporan *load test* dari *Vegeta*, diperoleh data sebagai berikut:

Tabel 1. Hasil pengujian *monitoring* data *metrics*

No	*Env*	Rate (rps)	Waktu Mulai *Test*	Waktu Metric Terbaca	Delay (s)
1	*Staging*	10	02/07/2022 15:52:51	02/07/2022 15:53:15	24
2	*Production*	10	02/07/2022 15:52:52	02/07/2022 15:53:00	8
3	*Staging*	15	02/07/2022 16:06:23	02/07/2022 16:06:30	7
4	*Production*	15	02/07/2022 16:06:23	02/07/2022 16:06:30	7
5	*Staging*	20	02/07/2022 16:20:27	02/07/2022 16:20:45	18
6	*Production*	20	02/07/2022 16:20:27	02/07/2022 16:20:45	18
7	*Staging*	25	02/07/2022 16:38:25	02/07/2022 16:38:45	20
8	*Production*	25	02/07/2022 16:38:25	02/07/2022 16:38:45	20
9	*Staging*	30	02/07/2022 16:45:37	02/07/2022 16:45:45	8
10	*Production*	30	02/07/2022 16:45:38	02/07/2022 16:45:45	8
			Rata-rata		13,8

Dari Tabel 1 hasil pengujian *throughput* di atas dapat dilihat bahwa sistem *monitoring* berhasil mengumpulkan data *metrics* berupa *throughput* dan *request latency* dan menampilkannya ke *dashboard Grafana* secara *realtime* dengan *delay* rata-rata sebesar 13,8 detik.

b. Pengujian *Monitoring Error Rate*

Selama proses *load test* berlangsung, dilakukan pemantuan pada dashboard *Grafana* untuk panel *Error Rate*. Berikut ini adalah hasil dari pemantauan melalui *dashboard Grafana*.

Gambar 7. Hasil pemantauan panel *Error Rate*

Dari hasil pengujian *load test* dan pemantauan pada *dashboard Grafana*, diperoleh data sebagai berikut:

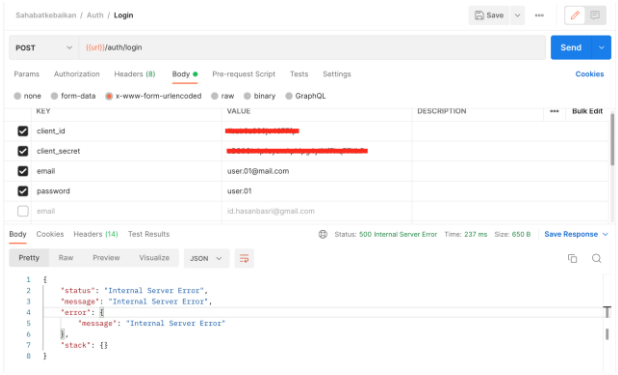
Tabel 2. Hasil pengujian *Error Rate*

No	*Env*	*Endpoint*	Durasi	*Rate*	Status Perubahan
1	*Staging*	GET /v1/campaigns	5 menit	10 *rps*	Berhasil terbaca
	Staging	GET /v1/error	5 menit	5 *rps*	
2	*Production*	GET /v1/campaigns	5 menit	10 *rps*	Berhasil terbaca
	Production	GET /v1/error	5 menit	5 *rps*	

Dari hasil tersebut dapat disimpulkan bahwa sistem *monitoring* dapat mengumpulkan data *metrics* berupa *error rate* dan mampu menampilkannya secara *realtime* pada *dashboard Grafana*.

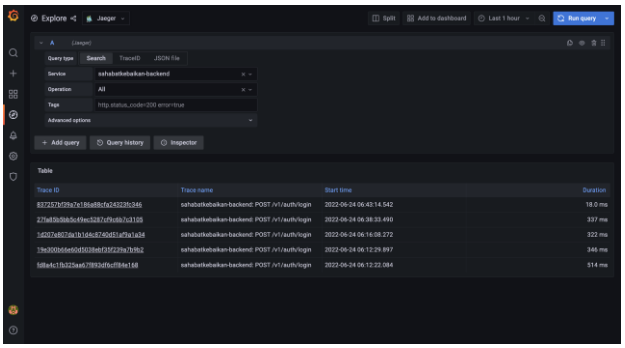
c. Pengujian *Monitoring* Data *Traces*

Pengguna melakukan *request* ke *endpoint POST /auth/login* dan mendapatkan respons *500 Internal Server Error* seperti pada gambar di bawah ini.



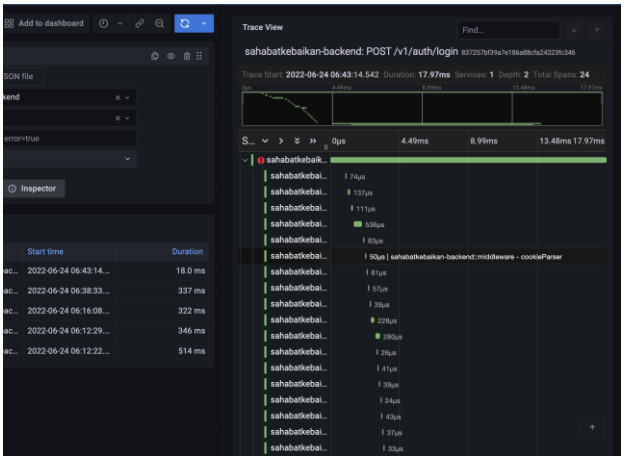
Gambar 8. Pemanggilan *Login* pada *API production*

Untuk mengetahui penyebab dari *error* tersebut adalah dengan mengecek *traces request* melalui menu *Explore Grafana* dan memilih *datasource Jaeger* seperti ditunjukkan pada gambar berikut.



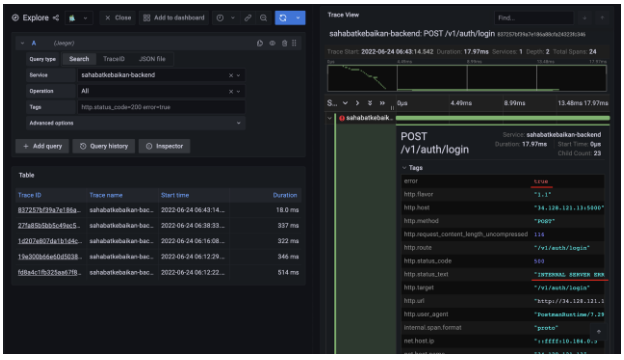
Gambar 9. Melihat daftar traces pada pengujian API *production*

Pada Gambar 9 menampilkan daftar *traces request* yang terjadi dalam satu jam terakhir. Kemudian dipilih *trace* yang memiliki *start time* sama dengan saat pelakukan pengujian yaitu *trace* yang paling terakhir dengan *Id 837257bf39a7e186a88cfa24323fc346*. Ketika diklik pada pada *trace* tersebut maka muncul detail *trace* berupa *spans* yang menunjukkan *trace* tersebut melewati fungsi apa saja seperti yang ditunjukkan pada Gambar 10 di bawah ini.



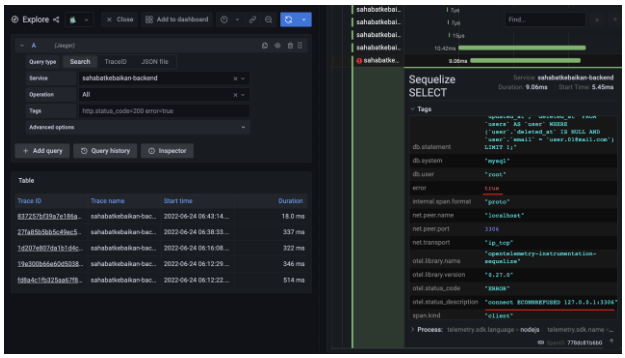
Gambar 10. Melihat detail *trace* pada pengujian *API production*

Gambar di bawah ini merupakan detail dari *span* paling luar pada *request* dengan *trace Id 837257bf39a7e186a88cfa24323fc346*. Di dalamnya terdapat berbagai *tags* yang menjadi informasi penting tentang *trace* ini. *Tag error* bernilai *true* artinya *trace* ini mengembalikan *error*. Kemudian terdapat *tags http.status_code* yang menunjukkan kode *error*-nya yaitu 500, dan *tags http.status_text* yang menunjukkan teks *error*-nya yaitu *Internal Server Error*.



Gambar 11. Melihat *tags* pada detail *trace* pada pengujian *production*

Setelah ditelurusi lebih lanjut pada span-span di dalam *trace* ini, diketahui bahwa penyebab *error* tersebut adalah pada *library Sequelize* yang mengalami kegagalan dalam melakukan *query SELECT*. Kegagalan ini dikarenakan tidak dapat terhubung ke *server database*.



Gambar 12. Melihat penyebab *error* pada *span Sequelize* pada pengujian *production*

Dengan demikian sistem *monitoring* berhasil menemukan sumber masalah penyebab pengguna tidak dapat *login* ke aplikasi *production* yaitu aplikasi gagal terhubung ke *server database*.

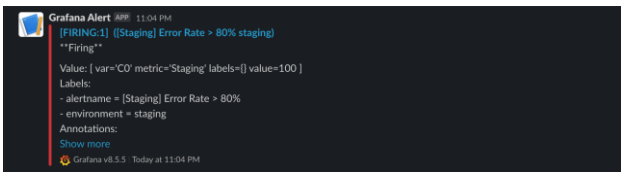
3) Pengujian Deteksi Anomali

a. Deteksi Anomali *Error Rate*

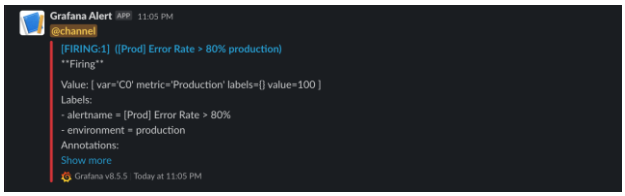
Pada pengujian ini dilakukan pemanggilan sebanyak 10 *rps* ke *endpoint GET /v1/campaigns* pada *REST API production* dan *staging* selama 10 menit. Setelah 2 menit berjalan maka *server database* akan dimatikan sehingga akan mendapatkan *response* 500 *Internal Server Error*. Hal ini menyebabkan *error rate* menjadi semakin meningkat seperti yang ditunjukkan oleh Gambar 13 pada pukul 23:03:00. Ketika *error rate* mencapai lebih dari 80% selama 30 detik maka akan memicu *alert* yang dikirimkan menuju *Slack*.

Gambar 13. Melihat panel *Error Rate* pada pengujian deteksi anomali

Berikut ini adalah tangkapan layar notifikasi yang dikirimkan menuju *Slack* baik untuk *server production* dan *staging*.



Gambar 14. Peringatan anomali *error rate* pada *API staging* terkirim ke *Slack*



Gambar 15. Peringatan anomali *error rate* pada *API production* terkirim ke *Slack*

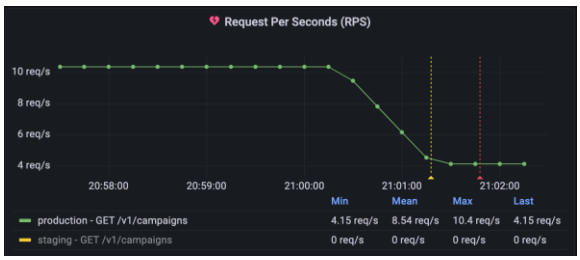
Sekitar 2 menit kemudian *server database* untuk masing-masing *server* dinyalakan kembali. Hal ini membuat pemanggilan *endpoint* mendapatkan respons 200 sehingga *error rate* perlahan menurun seperti yang ditunjukkan oleh Gambar 13 pada pukul 23:05:45. Setelah *error rate* berada di bawah 80 selama 30 detik maka kondisi akan menjadi normal kembali. Kemudian sistem akan mengirimkan notifikasi menuju *Slack* bahwa keadaan sudah kembali normal.

Gambar 16. Pemberitahuan anomali *error rate API staging* sudah kembali normal terkirim ke *Slack*

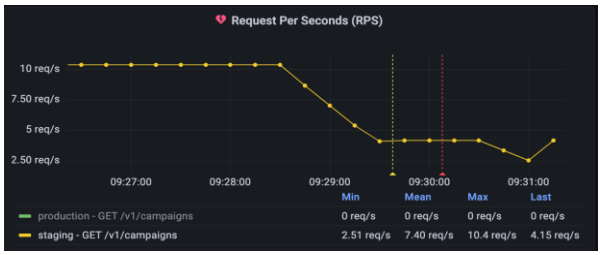
Gambar 17. Pemberitahuan anomali *error rate API production* sudah kembali normal terkirim ke *Slack*

b. Deteksi Anomali *Throughput*

Pada pengujian ini dilakukan pemanggilan ke *endpoint GET /v1/campaigns* pada *REST API production* dan *staging* sebanyak 10 *rps* selama 5 menit. Setelah itu langsung dilanjutkan dengan pemanggilan sebanyak 4 *rps* selama 2 menit. Hal ini mengakibatkan jumlah *rps* turun sebesar 60% seperti yang ditunjukkan pada gambar berikut ini.

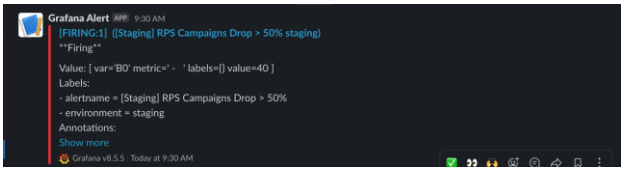


Gambar 18. Melihat panel *Request Per Seconds (RPS)* pada pengujian deteksi anomali *request drop API production*



Gambar 19. Melihat panel *Request Per Seconds (RPS)* pada pengujian deteksi anomali *request drop API staging*

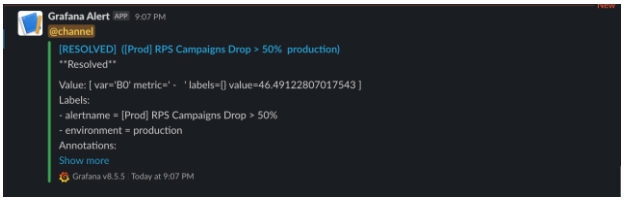
Setelah keadaan ini berlangsung selama 30 detik sistem mengirimkan peringatan menuju *Slack* seperti yang ditunjukkan pada gambar berikut.



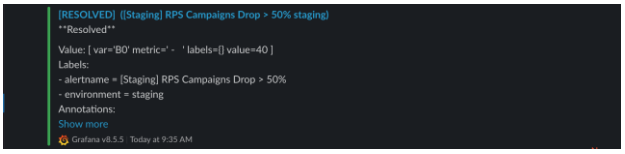
Gambar 20. Peringatan anomali *request drop* pada *API staging* terkirim ke *Slack*

Gambar 21. Peringatan anomali *request drop* pada *API production* terkirim ke *Slack*

Setelah pemanggilan sebanyak 4 *rps* selama 2 menit kemudian dilanjutkan pemanggilan sebanyak 10 *rps* selama 5 menit. Hal ini menyebabkan keadaan menjadi normal kembali dan sistem berhasil mengirimkan pemberitahuan menuju *Slack*.

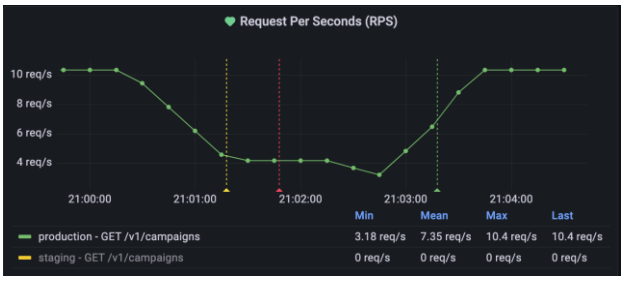


Gambar 22. Pemberitahuan anomali *request drop API staging* sudah kembali normal terkirim ke *Slack*

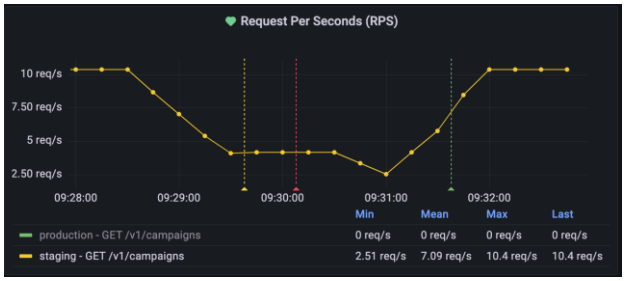


Gambar 23. Pemberitahuan anomali *request drop API staging* sudah kembali normal terkirim ke *Slack*

Ketika dilihat secara keseluruhan dalam panel *Request Per Seconds (RPS)* maka tampilannya seperti gambar berikut.



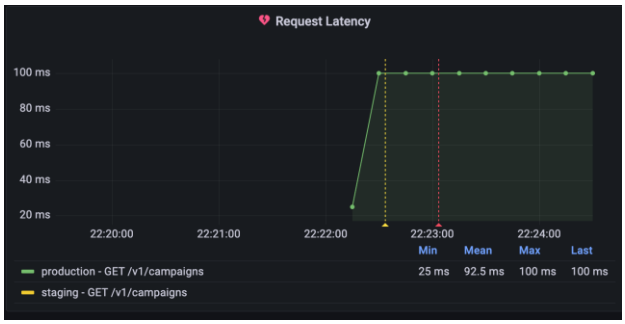
Gambar 24. Tampilan grafik keseluruhan pada pengujian *RPS Drop production*



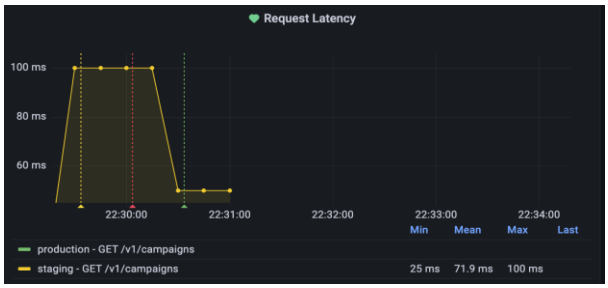
Gambar 25. Tampilan grafik keseluruhan pada pengujian *RPS Drop staging*

c. *Request Latency*

Pada pengujian ini dilakukan pemanggilan sebanyak 20 *rps* ke *endpoint GET /v1/campaigns* pada *REST API production* dan *staging* selama 2 menit. *Request latency* rata-rata sebesar 100 milidetik baik pada *server production* dan *staging* seperti yang ditunjukkan pada grafik berikut ini.

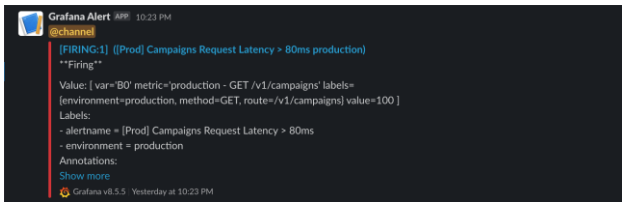


Gambar 26. Melihat panel *Request Latency* pada pengujian deteksi anomali latensi > 80 *ms* pada *API production*

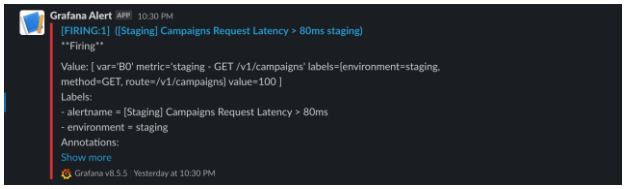


Gambar 27. Melihat panel *Request Latency* pada pengujian deteksi anomali latensi > 80 *ms* pada *API staging*

Sistem *monitoring* berhasil memberikan peringatan melalui *Slack* seperti yang ditunjukkan pada gambar berikut.



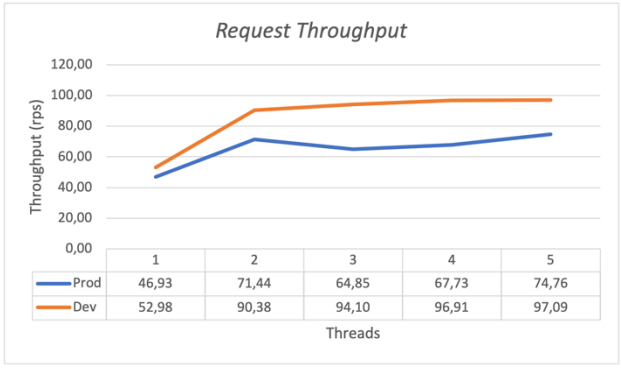
Gambar 28. Peringatan anomali latensi >80 *ms* pada *API production* terkirim ke *Slack*



Gambar 29. Peringatan anomali latensi >80 *ms* pada *API staging* terkirim ke *Slack*

4) Pengujian Performa Aplikasi

Gambar 30. menunjukkan *request throughput* dari total 10 pengujian yang dilakukan pada *server* aplikasi *production* dan *development*.



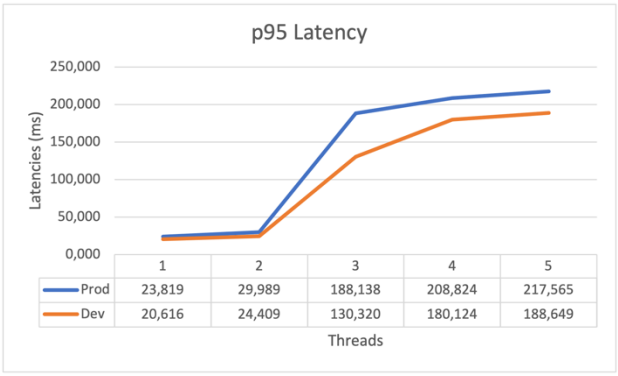
Gambar 30. Hasil *throughput* pada uji performa aplikasi

Seperti yang terlihat pada Gambar 30 implementasi *OpenTelemetry* untuk *monitoring metrics* dan *traces* pada *server production* memberikan dampak negatif pada performa aplikasi. Dari 5 kali pengujian dengan setiap pengujian menggunakan jumlah *virtual user (threads)* yang berbeda semuanya menunjukkan jumlah *request throughput* pada *server production* lebih sedikit daripada *server development*. Berikut ini adalah presentase penurunan jumlah *request throughput server production* dibandingkan dengan *server development*.

Tabel 3. Penurunan *request throughput* pada *server production*

Thread	**Penurunan (%)**
1	11,42
2	20,96
3	31,08
4	30,11
5	23,00
Rata-rata	23,31

Gambar 31 menunjukkan hasil *request latency* dari total 10 pengujian yang dilakukan pada *server* aplikasi *production* dan *development*.



Gambar 31. Hasil *Request Latency* pada uji performa aplikasi

Dari gambar tersebut diketahui bahwa latensi pada *server production* selalu lebih tinggi dibandingkan dengan *server development*. Hal ini menunjukkan bahwa implementasi *OpenTelemetry* untuk *monitoring metrics* dan *traces* pada *server production* juga memberikan pengaruh terhadap performa aplikasi itu sendiri dalam hal *request latency*. Berikut ini adalah presentase kenaikan latensi *server production* dibandingkan dengan *server development*.

Tabel 4. Kenaikan latensi pada *server production*

Thread	**Kenaikan (%)**
1	15,54
2	22,86
3	44,37
4	15,93
5	15,33
Rata-rata	22,80

Dari data yang ditunjukkan oleh Gambar 30, Gambar 31, Tabel 3, dan Tabel 4 dapat diambil kesimpulan bahwa implementasi *OpenTelemetry* untuk *monitoring metrics* dan *traces* memberikan pengaruh negatif pada performa aplikasi

yaitu dapat menurunkan jumlah *request throughput* dengan penurunan rata-rata 23,32% dan menaikkan *request latency* dengan kenaikan rata-rata 22,80%. Hal ini dapat diterima karena dengan adanya implementasi *OpenTelemetry* pada aplikasi artinya terdapat tambahan pekerjaan yang berjalan di setiap *request* yang terjadi yaitu pekerjaan untuk mengirimkan data *metrics* dan *traces* menuju ke *server monitoring*. Perlu adanya penelitian lebih lanjut untuk melakukan optimasi pengiriman agar penurunan performa yang disebabkan oleh implementasi *OpenTelemetry* menjadi semakin kecil.

V. SIMPULAN

Berdasarkan hasil penelitian yang diperoleh dari penelitian "Perancangan Sistem *Monitoring* Performa Aplikasi Menggunakan *OpenTelemetry* dan *Grafana Stack*", dapat ditarik kesimpulan sebagai berikut:

- *OpenTelemetry* dapat digunakan untuk menginstrumentasi dan mengumpulkan data telemetri berupa *metrics* dan *traces* yang dapat diproses dan ditampilkan melalui *dashboard Grafana*.
- Sistem *monitoring* mampu memantau *data metrics* dua aplikasi berbasis *REST API* secara *realtime* dengan *delay* rata-rata 13,8 detik.
- Melalui *Grafana Alert Manager*, sistem *monitoring* mampu memberikan peringatan melalui *Slack* ketika terjadi anomali pada data *metrics* yang dikumpulkan.
- Sistem *monitoring* dapat digunakan sebagai alat *debugging* yang efektif untuk menemukan akar masalah yang terjadi pada aplikasi berbasis *REST API* dengan menggunakan data *traces* yang disimpan pada *Jaeger*.
- Implementasi *OpenTelemetry* dalam aplikasi *backend* berbasis *REST API* untuk memonitor *metrics* dan *traces* memberikan dampak negatif pada performa aplikasi itu sendiri yaitu dapat menurunkan jumlah *request throughput* dengan penurunan rata-rata 23,32% dan menaikkan *request latency* dengan kenaikan rata-rata 22,80%.

REFERENSI

[1] Y. Einav. "Amazon Found Every 100ms of Latency Cost them 1% in Sales". [Online]. Available: https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales (accessed January 31, 2022).

[2] A. Van Hoorn and S. Siegl. "Application Performance Management: Measuring and Optimizing the Digital Customer Experience". *SIGS DATACOM GmbH. Troisdrof*. 2019.

[3] G. Kim, J. Humble, P. Debois and J. Willis. *The Devops Handbook How to Create World Class Agility, Reliability, and Security in Technology Organizations*. 2nd ed. Page 360. 2021

[4] J.J. Tang. "Observability Vs. Monitoring: What's the Difference?". [Online]. Available: https://devops.com/observability-vs-monitoring-whats-the-difference/ (accessed January 31, 2022).

[5] OpenTelemetry. [Online]. Available: https://opentelemetry.io/docs/concepts(accessed January 31, 2022).

[6] Prometheus. [Online]. Available: https://prometheus.io/docs/introduction/overview/(accessed January 31, 2022).

[7] Jaeger Overview. [Online]. Available: https://www.jaegertracing.io/docs/1.35/ (accessed January 31, 2022).

[8] Grafana. [Online]. Available: https://grafana.com/docs/grafana/latest/basics/ (accessed January 31, 2022).