Load Test pada *Microservice* yang menerapkan CQRS dan *Event Sourcing*

Difa Al Fansha[1], Muhammad Yusril Helmi Setyawan[2], Mohamad Nurkamal Fauzan[3]
Program Studi Teknik Informatika, Politeknik Pos Indonesia
Jl. Sariasih No.54 Sarijadi, Bandung, 40151, Jawa Barat, Indonesia
Email: [1]difaal21@gmail.com, [2]yusrilhelmi@poltekpos.ac.id, [3]m.nurkamal.f@poltekpos.ac.id

Abstract. Load Test on Microservice implementing CQRS and Event Sourcing. *In developing an application, determining the architecture is a very important job. This research implements microservice architecture with CQRS pattern and event sourcing on OpenAPI, API-driven and event-driven integration between services. Applying the right architecture can make the performance of the application faster. There are two test activities that are useful to find out the difference in the frequency of requests for API-driven and event-driven that can affect response time, error rate and throughput, as well as analyze which architecture has better performance. The test is carried out using a load test technique using the JMeter tool. This study proves that microservices with CQRS and Event Sourcing patterns have 3.7% faster performance compared to API-driven, and communication between services has no effect on error rate and throughput.*
Keywords: *microservice, load test, CQRS, event sourcing, software architecture*

Abstrak. *Dalam pengembangan sebuah aplikasi, penentuan arsitektur merupakan pekerjaan yang sangat penting. Penelitian ini mengimplementasikan arsitektur microservice dengan pola CQRS dan event sourcing pada OpenAPI, integrasi antar service berbasis API-driven dan event-driven. Penerapan arsitektur yang tepat dapat membuat performansi dari aplikasi menjadi lebih cepat. Terdapat dua aktivitas pengujian yang berguna untuk mengetahui perbedaan frekuensi request pada API-driven dan event-driven yang dapat mempengaruhi response time, error rate dan juga throughput, serta menganalisis arsitektur mana yang memiliki performa yang lebih baik. Pengujian dilakukan dengan teknik load test yang menggunuakan tool JMeter. Penelitian ini membuktikan bahwa microservice dengan pola CQRS dan Event Sourcing memiliki performansi lebih cepat 3,7% dibandingkan dengan API-driven, serta komunikasi antar service tidak berpengaruh pada error rate dan throughput.*
Kata Kunci: *microservice, load Test, CQRS, event sourcing, software arsitektur*

1. Pendahuluan

Dalam pembuatan sebuah aplikasi, salah satu yang dipertimbangkan adalah arsitektur perangkat lunak [1]. Penerapan arsitektur perangkat lunak yang tepat dapat membawa manfaat bagi aplikasi [2]. *Microservice* merupakan salah satu contoh arsitektur perangkat lunak yang memecah aplikasi menjadi beberapa bagian kecil dan aplikasi kecil tersebut adalah *service* [3]. Tidak ada aturan baku dalam pemecahan aplikasi, tetapi biasanya pemecahan dilakukan berdasarkan proses bisnis [4]. Dengan memecah aplikasi, *developer* dapat menggunakan teknologi dan menerapkan pola yang berbeda pada setiap *service* sesuai dengan kebutuhan. Pada *microservice* terdapat pola yang bernama *Command Query Responsibility Segregation* (CQRS) [5]. CQRS memisahkan proses *command* dan juga *query* [6]. *Command* bertugas untuk menangani proses modifikasi data, sedangkan *query* bertanggung jawab untuk proses membaca data [7]. CQRS sangat cocok dikombinasikan dengan *Event Sourcing* [8]. Komunikasi pada *event sourcing* menggunakan *event-driven* dan setiap perubahan disimpan didalam *message broker* sebagai *event* [9].

Penelitian ini akan memecah sebuah *service* menjadi dua, yaitu *command service* dan *query service*, serta menerapkan database per *service* agar tidak saling ketergantungan [10]. Kegunaan kedua *service* ini mengikuti aturan CQRS. Komunikasi kedua *service* menggunakan *API-driven* dan *event-driven* dengan tujuan mengetahui komunikasi mana yang memiliki performa yang lebih baik. Implementasi arsitektur perangkat lunak yang baik dapat

mempengaruhi performansi dari *service*. Pengujian dilakukan dengan lingkungan *client to server* menggunakan teknik *load test* dan teknik ini akan mengukur *response time* dari berbagai kondisi [11]. Salah satu *tool* yang digunakan untuk *load test* adalah JMeter [12]. JMeter akan melakukan request pada server, lalu *output* yang dihasilkan akan menjadi data untuk dianalisis agar dapat membuktikan komunikasi mana yang lebih cepat. Terdapat dua aktivitas yang dilakukan dalam pengujian. Aktivitas pertama dilakukan *request* sebanyak 200 dalam jangka waktu 60 detik dan aktivitas kedua dilakukan sebanyak 1000 request selama 60 detik. Perbedaan frekuensi *request* pada aktivitas pertama dan kedua bertujuan untuk mengetahui apakah terdapat pengaruh *response time, error rate*, dan *throughput* terhadap komunikasi antar *service*. *Response time* merupakan jumlah waktu server mengirimkan *response* terhadap *response client. Error rate* adalah rata-rata *error* yang terjadi ketika proses *request* dilakukan, sedangkan *throughput* adalah banyaknya data yang dapat diproses dalam satuan waktu.

2. Tinjauan Pustaka

2.1. *Load Test*

Load test berguna sebagai *performance testing* serta memastikan perangkat lunak bekerja dengan semestinya. Teknik ini akan mengukur *response time* dari berbagai kondisi [11]. *Output* dari *load test* dapat dianalisis untuk mengetahui masalah performansi yang terjadi pada aplikasi [13].

2.2. *Microservice*

Microservice memecah aplikasi menjadi bagian-bagian kecil yang saling berintegrasi [14]. Mengimplementasikan arsitektur *microservice* mempunyai banyak manfaat, seperti masing-masing *service* dapat menggunakan teknologi yang berbeda sesuai dengan kebutuhannya [15]. Saat menangani *request* yang jumlahnya cukup besar, arsitektur *microservice* memiliki perfomansi yang lebih baik daripada monolitik [16].

2.3. *Command Query Responsibility Segregation* (CQRS)

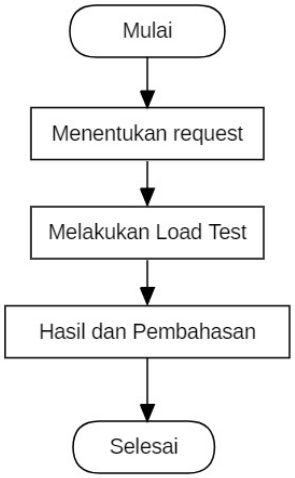
CQRS memisahkan proses *command* dan *query*, sehingga performansi *service* akan lebih baik. CQRS sangat ideal untuk aplikasi yang memerlukan proses modifikasi data dan informasi yang besar [17]. Database pada *command* dan *query* harus berbeda dan *command* akan mengisolasi dari operasi *query*. Biasanya CQRS dikombinasikan dengan *event sourcing* [18].

2.4. *Event Sourcing*

Event sourcing menangani operasi di data yang berdasarkan oleh urutan kejadian [19] dan data tersebut disimpan di tempat penyimpanan yang bernama *message broker*. Ketika perubahan terjadi, perubahan tersebut akan disimpan pada *message broker*.

3. Metodologi Penelitian

Mendefinisikan langkah-langkah yang dikerjakan dalam pengujian untuk mengetahui dampak dari perbedaan arsitektur perangkat lunak, adapun langkah-langkah tersebut mengacu pada *flowchart* berikut:



Gambar 1. Metodologi Penelitian

Langkah-langkah pekerjaan dalam pengujian yang ditunjukkan pada Gambar 1 dijelaskan sebagai berikut:

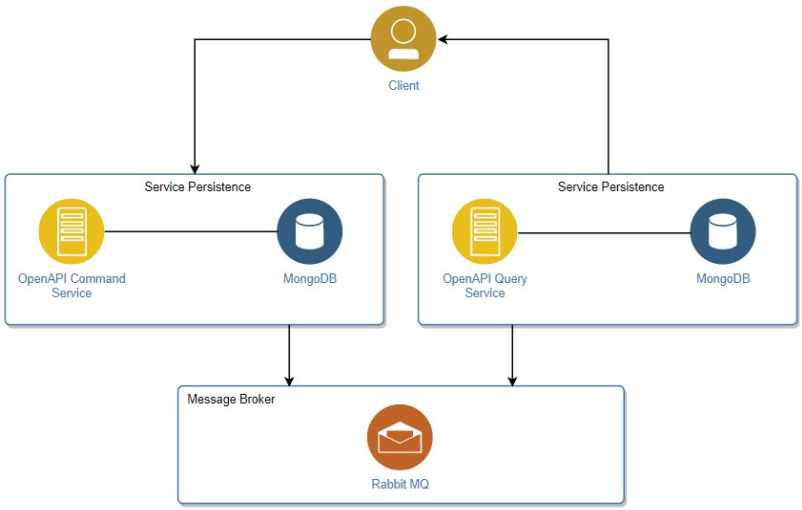
3.1. Menentukan Request

HTTP *request* yang dilakukan oleh *client* akan diarahkan pada obyek penelitian. Objek pada penelitian ini adalah OpenAPI. OpenAPI merupakan *service* utama pada aplikasi dokumentasi API. Aplikasi dokumentasi API ini menerapkan OpenAPI *Specification* (OAS) dalam pembuatan dokumen API. Pembuatan dokumen API dilakukan dengan cara menulis *code* atau *coding*. Aplikasi yang menerapkan OAS disebut sebagai OpenAPI *service*.

1) Arsitektur Perangkat Lunak

Penelitian ini memecah OpenAPI *service* menjadi dua bagian sesuai kaidah arsitektur *microservice* dan CQRS. Integrasi antar *service* menggunakan *event-driven* dan *API-driven*. OpenAPI *service* dibangun menggunakan teknologi *express* sebagai *framework* nodeJS dan MongoDB sebagai *database*-nya.

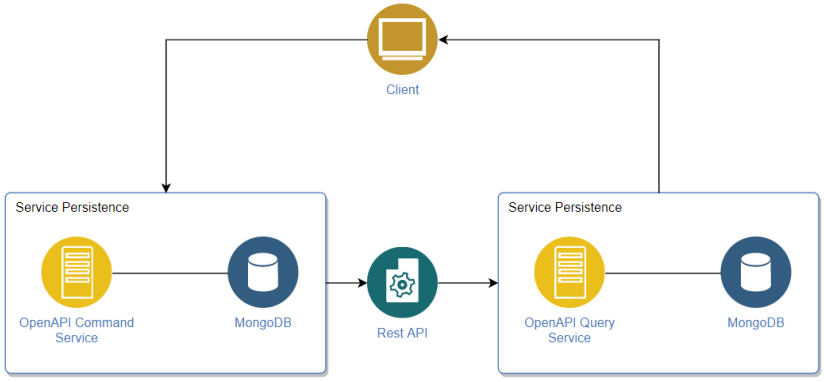
Gambar 2 menjelaskan kombinasi arsitektur CQRS dan *event sourcing* dengan komunikasi *event-driven*.



Gambar 2. Kombinasi CQRS dan *Event Sourcing*

Pada arsitektur CQRS dan *event sourcing, command service* berperan sebagai *producer*. Ketika *user* melakukan perubahan data, data tersebut akan di*publish* oleh *command service* ke *message broker* sebagai *event* [20] dan disimpan dalam *database*. Lalu *query service* bekerja sebagai *consumer*, yang berguna untuk *subscribe* atau menerima data. Setelah data tersebut di diterima, data akan dimasukkan kedalam *database*. Komunikasi *event-driven* ini bersifat

asynchronous [21]. Karena bersifat *asynchronous*, *command service* dapat mengembalikan *response* segera mungkin ketika sudah mengirimkan data pada *message broker* [19]. Gambar 3 menjelaskan penerapan API-driven pada pola CQRS.



Gambar 3. Aristektur CQRS dengan API-*driven*

Komunikasi pada arsitektur CQRS menggunakan REST API yang bersifat *synchronous* [22]. Ketika user melakukan *request* berupa modifikasi data, *command service* akan menyimpan data tersebut kedalam *database* dan melakukan API *call* ke *query service*. Lalu, *query service* akan menyimpan ke *database*, sehingga data akan *up to date*. Karena komunikasi ini bersifat *synchronous*, *client* akan menunggu proses API *call* dari *command* ke *query service* selesai, lalu bisa mendapatkan *response*.

2) *Endpoint*

Tiga target *endpoint* dilakukan pada pengujian dengan sampel HTTP *method* GET dan PUT. *Method* GET akan mendapatkan *source code* dari pembuatan dokumen API dan *method* PUT akan melakukan *update* pada *source code*. Pada aplikasi dokumentasi API, setiap melakukan perubahan pada *source code*, sistem akan secara otomatis melakukan proses *get source code*. *Method* POST digunakan untuk menambahkan dokumen, sedangkan *method* DELETE digunakan untuk menghapus dokumen yang merupakan wadah penampung informasi *source code*. Ketika dokumen dihapus, secara otomatis *source code* juga akan terhapus dari *database*, serta penamaan dokumen harus unik.

Secara *default,* JMeter akan dieksekusi secara *parallel*. Karena itu penelitian ini mengecualikan *method* POST dan DELETE. Apabila memakai keempat *method* tersebut yang diharuskan sesuai dengan alur, menyebabkan eksekusi pada JMeter tidak lagi *parallel* melainkan berurutan. Karena penamaan dokumen harus unik, pembuatan dokumen dengan cara *coding* akan lebih memiliki banyak proses yang menggunakan *method* GET dan PUT. Tabel 1 berisi informasi mengenai *request* yang digunakan untuk mendapatkan *source code* pada *thread group* API-*driven*.

Tabel 1. GET *Source Code Thread Group* API-*driven*

Path		
Endpoint	/v1/source-code/{documentName}	
Method	GET	
Parameter		
Type	***Name***	***Value***
Path	documentName	api-driven
Request		
Send	202 bytes	
Response		
Size per Request	***Number of Request***	
9077 bytes	200	
9077 bytes	1000	

Tabel 2 berisi informasi mengenai *request* yang berguna untuk mendapatkan *source code* pada thread group *event-driven*.

Tabel 2. GET *Source Code Thread Group Event-driven*

Path		
Endpoint	/v1/source-code/{documentName}	
Method	GET	
Parameter		
Type	***Name***	***Value***
Path	documentName	event-driven
Request		
Send	202 bytes	
Response		
Size per Request		***Number of Request***
9077 bytes		200
9077 bytes		1000

Hanya ada satu *method* GET. Tetapi, terdapat perbedaan *path parameter* dalam *thread group* API-*driven* dan *thread group event-driven*. *Method* GET ini berguna untuk mendapatkan *source code* sesuai dengan *value* pada *path parameter*.

Tabel 3 berisikan informasi mengenai *request* yang dilakukan oleh *client* yang berguna untuk memperbarui *source code* pada *thread group* API-*driven*.

Tabel 3. *Update Source Code Thread Group* API-*driven*

Path		
Endpoint	/v1/source-code/{documentName}/api-driven	
Method	PUT	
Parameter		
Type	***Name***	***Value***
Path	documentName	api-driven
Request		
Send	8965 bytes	
Response		
Size per Request		***Number of Request***
362 bytes		200
362 bytes		1000

Tabel 4 menjelaskan mengenai *request* yang dilakukan oleh *client*, *request* ini berguna untuk memperbarui *source code* pada *thread group event-driven*.

Tabel 4. *Update Source Code Thread Group Event-driven*

Path		
Endpoint	/v1/source-code/{documentName}/event-driven	
Method	PUT	
Parameter		
Type	***Name***	***Value***
Path	documentName	event-driven
Request		
Send	8965 bytes	
Response		
Size per Thread		***Number of Thread***
362 bytes		200
362 bytes		1000

Dua buah *endpoint* dengan *method* PUT berguna untuk memperbarui *source code* via API-*driven* dan juga *event-driven*. Perbedaan *endpoint* tersebut bisa dilihat pada Tabel 3 dan Tabel 4.

3.2. Melakukan *Load Test*

Pengujian dilakukan menggunakan *tool* JMeter dengan teknik *load test*. Terdapat dua buah aktivitas. Aktivitas pertama dan kedua memiliki jumlah *sample request* yang berbeda. Tujuannya ialah mengetahui perbedaan jumlah frekuensi *request* dapat mempengaruhi kinerja *event-driven* dan API-*driven*, serta menganalisa komunikasi yang memiliki performa yang lebih cepat. JMeter Ramp-up per periode diisi dengan 60 detik. Aktivitas pertama berjumlah 200 dan aktivitas kedua berjumlah 1000. *Load test* diatur selama satu menit untuk mengetahui kemampuan *service* dalam menangani *request* yang menggunakan *server* Heroku. Spesifikasi *server* ditampilkan pada Tabel 5.

Tabel 5. Spesifikasi *Server*

Spesifkasi	Keterangan
RAM	512 MB
Always On	*Yes*
Slug	500 MB
Bandwidth	2 TB / bulan
Number Apps	Max 5 *apps* (tanpa kartu debit)

3.3. Hasil dan Pembahasan

Hasil merupakan keluaran atau *output* yang berasal dari *tool* JMeter. *Output* tersebut akan dijadikan data yang nantinya akan dianalisis untuk mengetahui arsitektur yang memiliki *response time* yang lebih cepat. *Output* JMeter berupa *summary report* yang berisi rata-rata waktu menjalankan semua *request*, *error rate* terjadinya *error* ketika *request*, dan *throughput* data yang dapat diproses dalam satuan waktu.

4. Hasil dan Diskusi

Pada JMeter dibuat dua buah *thread group* yang dikelompokkan berdasarkan komunikasi. *Thread group* pertama menangani komunikasi API-*driven* dan *thread group* kedua bertanggung jawab untuk komunikasi *event-driven*.

Informasi mengenai *summary report* yang didapatkan dari JMeter berisikan *average* atau rata-rata waktu menjalankan serangkaian request, *error rate* (jumlah terjadinya *error* ketika melakukan *request* dalam persen), dan *throughput* (jumlah *request* yang dapat diproses dalam hitungan detik).

Tabel 6. *Summary Report* API-*driven* Aktivitas Pertama

Label	*Sample*	*Average* (ms)	*Error Rate* (%)	*Throughput*
Update Source Code	100	710	*0*	1,7/sec
Get Source Code	100	598	0	1,7/sec

Tabel 6 menunjukan laporan aktivitas pertama *thread group* API-*driven.* Pengujian aktivitas pertama tersebut dilakukan dengan *sample request* sebanyak 200 dengan rata-rata waktu eksekusi semua *request* 654 ms. *Error rate* sebesar 0% yang artinya tidak ada error sedikit pun dalam memproses total data sebanyak 3,4 per detik.

Tabel 7. *Summary Report* Event-*driven* Aktivitas Pertama

Label	*Sample*	*Average* (ms)	*Error Rate* (%)	*Throughput*
Update Source Code	100	*650*	*0*	1,7/sec
Get Source Code	100	611	0	1,7/sec

Tabel 7 menunjukan laporan aktivitas pertama *thread group event-driven*. Pengujian aktivitas pertama tersebut dilakukan dengan *sample request* sebanyak 200 dengan rata-rata waktu eksekusi semua *request* 630 ms. Error rate sebesar 0%, yang artinya tidak ada *error* sedikit pun dalam memproses total data sebanyak 3,4 per detik.

Dari Tabel 6 dan Tabel 7 dapat diambil kesimpulan bahwa *event-driven* memiliki rata-rata *response time* 3,7% lebih cepat daripada API-*driven*, sedangkan *error rate* dan *throughput* tidak terlihat adanya perbedaan. Proses *get source code* pada API-*driven* memiliki *response time* yang lebih baik, sedangkan proses *update source code* dan *response time* dari komunikasi *event-driven* lebih unggul.

Tabel 8. Summary Report API-driven Aktivitas Kedua

Label	*Sample*	*Average* (ms)	*Error Rate* (%)	*Throughput*
Update Source Code	500	725	*0*	8,3/sec
Get Source Code	500	597	0	8,3/sec

Tabel 8 menunjukan laporan aktivitas kedua *thread group* API-*driven.* Pengujian aktivitas kedua tersebut dilakukan dengan *sample request* sebanyak 1000 dengan rata-rata waktu eksekusi semua *request* 661 ms. *Error rate* sebesar 0%, yang artinya tidak ada *error* sedikit pun dalam memproses total data sebanyak 16,6 per detik.

Tabel 9. *Summary Report Event-driven* Aktivitas Kedua

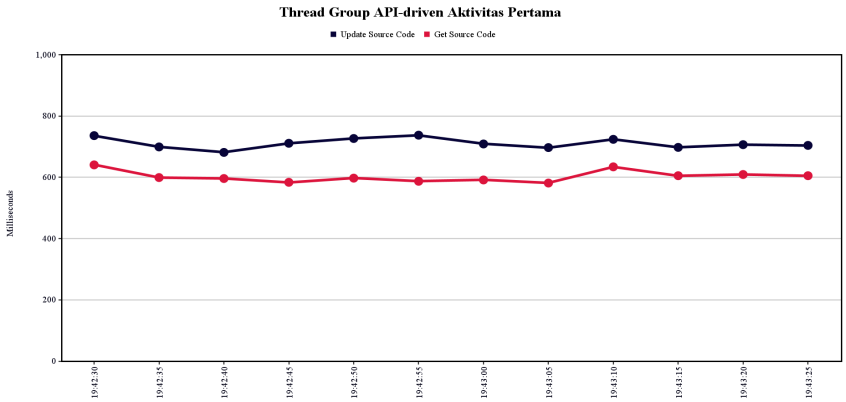
Label	*Sample*	*Average* (ms)	*Error Rate* (%)	*Throughput*
Update Source Code	500	650	*0*	8,3/sec
Get Source Code	500	599	0	8,3/sec

Tabel 9 menunjukan laporan aktivitas kedua *thread group event-driven.* Pengujian aktivitas kedua tersebut dilakukan dengan *sample request* sebanyak 1000 dengan rata-rata waktu eksekusi semua *request* 625 ms. *Error rate* sebesar 0% yang artinya tidak ada *error* sedikit pun dalam memproses total data sebanyak 16,6 per detik.

Dari Tabel 8 dan Tabel 9 dapat diambil kesimpulan bahwa *event-driven* memiliki rata-rata *response time* 5,4% lebih cepat dari pada API-*driven* dengan jumlah *request* sebesar 1000 per menit, sedangkan *error rate* dan *throughput* tidak menunjukkan perbedaan. HTTP *method* GET pada API-*driven* memiliki *response time* yang lebih cepat, tetapi HTTP *method* PUT pada *event-driven* menujukkan performa yang lebih baik.

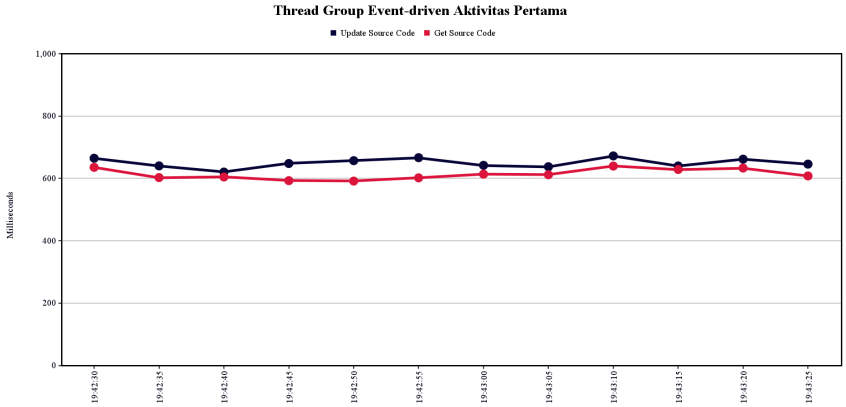
HTTP *method* GET tidak memerlukan komunikasi API-*driven* maupun *event-driven*, karena *method* GET dapat mengambil data secara langsung dari *database*. Berbeda dengan *method* PUT, PUT harus menggunakan API-*driven* atau *event-driven* untuk mengirimkan data *pada* service yang lain, agar data pada *database service* lain itu sinkron. Terjadi kenaikan rata-rata *response time* sebesar 1,05% pada aktivitas kedua API-*driven* yang mempunyai *request* yang lebih banyak dari aktivitas pertama. Sedangkan pada aktivitas kedua, pada *event-driven* terjadi penurunan rata-rata *response time* sebesar 0,79% dari aktivitas pertama.

Berikut grafik *response time* pada JMeter. Sumbu x pada grafik menandakan waktu terjadinya *request* dengan format HH:mm:ss, sedangkan sumbu y melambangkan *response time* dalam *miliseconds*.



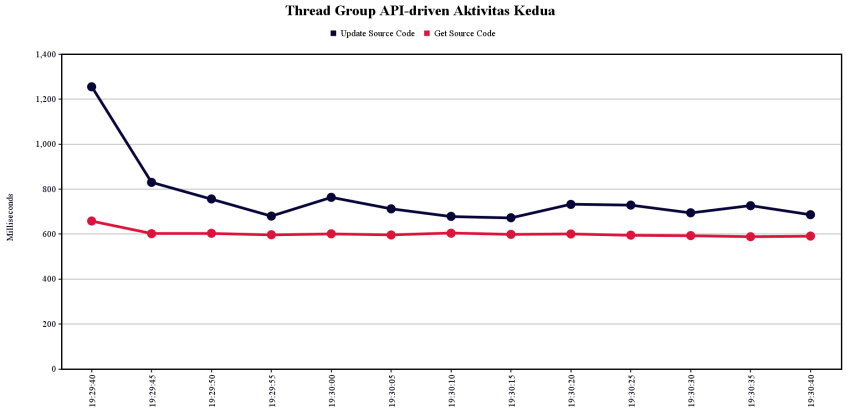
Gambar 4. Grafik *Thread Group* API-*driven* Aktivitas Pertama

Grafik pada Gambar 4 menunjukkan bahwa *update source code* berada memiliki *response time* yang lebih lama dibandingkan dengan *get source code*, tetapi kedua *request* cukup stabil.



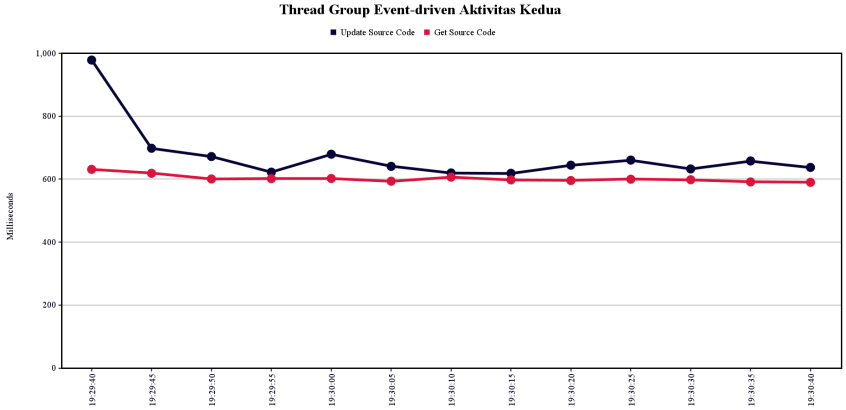
Gambar 5. Grafik Thread Group Event-driven Aktivitas Pertama

Line pada Gambar 5 memperlihatkan kedua *request* tidak terlalu berbeda dan cukup stabil, tetapi *update source* masih tetap memiliki *response time* yang lebih lama dibandingkan dengan *get source code*.



Gambar 6. Grafik *Thread Group* API-*driven* Aktivitas Kedua

Line get source code pada Gambar 6 cenderung stabil dibandingkan dengan *update source code*. Awal *request* pada *update source code* memiliki *request* yang tinggi dan terjadi penurunan yang signifikan, tetapi penurunan tersebut masih diatas *line get source code*.



Gambar 7. Grafik *Thread Group Event-driven* Aktivitas Kedua

Sesuai Gambar 6, pada *line update source code* terdapat penurunan yang signifikan, kemudian mulai stabil. Bahkan, perbedaan *response time request* antara keduanya tidak terlalu mencolok. *Get source code* dari awal sampai akhir tetap stabil.

Gambar 4 sampai dengan Gambar 6 menunjukan *event-driven* menampilkan performansi yang lebih cepat dibandingkan API-*driven*. Pada *thread group event-driven,* grafik Gambar 5 dan Gambar 7 menunjukan bahwa perbedaan kedua *request* tidak terlalu berbeda, bahkan terdapat garis yang bersentuhan. Sedangkan pada API-*driven,* terdapat area kosong pada tengah-tengah garis *get source code* dan *update source code*. Sehingga, memperjelas bahwa *response time* dari *get source code* lebih cepat.

5. Kesimpulan dan Saran

Setelah dilakukan pengujian pada aktivitas pertama dan kedua, *event-driven* memiliki 3,7% lebih cepat daripada API-*driven*, sedangkan aktivitas kedua memiliki rata-rata *response time* lebih cepat sebanyak 5,4%. Perubahan frekuensi *request* pada aktivitas pertama ke aktivitas kedua hanya mempengaruhi *response time*, tidak halnya dengan *error rate* dan *throughput*.

Penelitian ini telah membuktikan proses *command* lebih cepat lebih cepat 0,094% dengan menggunakan arsitektur CQRS dan *event soucing* dengan komunikasi *event-driven* dari pada API-*driven.* Sedangkan proses *query* data pada API-*driven* lebih cepat lebih cepat 0,012% dari pada *event-driven*. Tetapi ini tidak ada kaitannya dengan komunikasi antar *service*, karena proses *query* langsung mengambil data dari *database* tanpa komunikasi antar *service*.

Referensi

[1] H. Suryotrisongko, D. P. Jayanto, and A. Tjahyanto, "Design and Development of Backend Application for Public Complaint Systems Using Microservice Spring Boot," *Procedia Comput. Sci.*, vol. 124, pp. 736–743, Nov. 2017, doi: 10.1016/j.procs.2017.12.212.

[2] A. Akbulut and H. G. Perros, "Software Versioning with Microservices through the API Gateway Design Pattern," *2019 9th Int. Conf. Adv. Comput. Inf. Technol. ACIT 2019 - Proc.*, pp. 289–292, June. 2019, doi: 10.1109/ACITT.2019.8779952.

[3] and M. O. Takanori Ueda, Takuya Nakaike, "Workload Characterization for Microservices," no. c, pp. 85–94, Sept. 2016.

[4] F. Rademacher, S. Sachweh, and A. Zundorf, "Differences between model-driven development of service-oriented and microservice architecture," *Proc. - 2017 IEEE Int. Conf. Softw. Archit. Work. ICSAW 2017 Side Track Proc.*, pp. 38–45, Apr. 2017.

[5] A. Debski, B. Szczepanik, and M. Malawski, "In Search for a Scalable & Reactive Architecture of a Cloud Application : CQRS and Event Sourcing," pp. 1–8, Sept. 2016.

[6] G. Maddodi and S. Jansen, "Responsive Software Architecture Patterns for Workload Variations: A Case-study in a CQRS-based Enterprise Application," *CEUR Workshop Proc.*, vol. 2047, no. 2005, p. 30, 2017.

[7] Z. Long, "Improvement and Implementation of a High Performance CQRS Architecture," *Proc. - 2017 Int. Conf. Robot. Intell. Syst. ICRIS 2017*, pp. 170–173, Oct. 2017.

[8] Y. Zhong, W. Li, and J. Wang, "Using event sourcing and CQRS to build a high performance point trading system," *ACM Int. Conf. Proceeding Ser.*, pp. 16–19, July. 2019.

[9] R. Laigner *et al.*, "From a Monolithic Big Data System to a Microservices Event-Driven Architecture," *Proc. - 46th Euromicro Conf. Softw. Eng. Adv. Appl. SEAA 2020*, no. i, pp. 213–220, Aug. 2020, doi: 10.1109/SEAA51224.2020.00045.

[10] S. Suresh Kumar and P. M. Mallikarjuna Shastry, "Database-per-service for e-learning system with micro-service architecture," *Proc. 2017 Int. Conf. Smart Technol. Smart Nation, SmartTechCon 2017*, pp. 705–708, 2018.

[11] D. I. Permatasari, "Pengujian Aplikasi menggunakan metode Load Testing dengan Apache JMeter pada Sistem Informasi Pertanian," *J. Sist. dan Teknol. Inf.*, vol. 8, no. 1, p. 135, Jan. 2020, doi: 10.26418/justin.v8i1.34452.

[12] R. Abbas, Z. Sultan, and S. Nazir B, "Comparative Analysis of Automated Load Testing Tools: Apache JMeter, Microsoft Visual Studio (TFS), LoadRunner, Siege," *Int. Conf.*

Commun. Technol., vol. 7, no. 6, pp. 117–130, Apr. 2017.
[13] R. Gao, Z. M. Jiang, C. Barna, and M. Litoiu, "A Framework to Evaluate the Effectiveness of Different Load Testing Analysis Techniques," *Proc. - 2016 IEEE Int. Conf. Softw. Testing, Verif. Validation, ICST 2016*, pp. 22–32, Apr. 2016.
[14] R. Xu, W. Jin, and D. Kim, "Microservice security agent based on API gateway in edge computing," *Sensors (Switzerland)*, vol. 19, no. 22, pp. 1–17, Nov. 2019.
[15] K. Gos and W. Zabierowski, "The Comparison of Microservice and Monolithic Architecture," *2020 IEEE 16th Int. Conf. Perspect. Technol. Methods MEMS Des. MEMSTECH 2020 - Proc.*, pp. 150–153, Apr. 2020.
[16] R. Mufrizal and D. Indarti, "Refactoring Arsitektur Microservice Pada Aplikasi Absensi PT. Graha Usaha Teknik," *J. Nas. Teknol. dan Sist. Inf.*, vol. 5, no. 1, pp. 57–68, Apr. 2019.
[17] S. Han and J.-I. Choi, "Data Processing System Using CQRS Pattern and NoSQL in V2X Environment," *Int. J. Smart Device Appl.*, vol. 8, no. 1, pp. 1–8, Jun. 2020.
[18] L. H. N. Villaça, L. G. Azevedo, and F. Baio, "Query strategies on polyglot persistence in microservices," *Proc. ACM Symp. Appl. Comput.*, pp. 1725–1732, Apr. 2018.
[19] S. O. Diakov, T. E. Zubrei, and A. S. Samoidiuk, "Application of Event Sourcing and CQRS in Distributed Systems," no. 34, pp. 16–22, 2019.
[20] R. Boncea, A. Zamfiroiu, and I. Bacivarov, "A scalable architecture for automated monitoring of microservices," *Acad. Econ. Stud. Econ. Informatics*, vol. 18, no. 1, pp. 13–22, 2018, [Online].
[21] A. Akbulut and H. G. Perros, "Performance Analysis of Microservice Design Patterns," *IEEE Internet Comput.*, vol. 23, no. 6, pp. 19–27, Nov. 2019, doi: 10.1109/MIC.2019.2951094.
[22] M. Stefanko, "The Saga Pattern in a Reactive Microservices Environment," no. Icsoft, pp. 483–490, Jan. 2019.