Penerapan Microservices dan Amazon Elastic Container Service untuk Mendukung Scalability

Antonius Tanuwijaya, Henry Novianus Palit, Agustinus Noertjahyana
Program Studi Informatika Fakultas Teknologi Industri Universitas Kristen Petra
Jl. Siwalankerto 121 – 131 Surabaya 60236
Telp. (031) – 2983455, Fax. (031) – 8417658
E-mail: antoniustanuwijaya.atx98@gmail.com, hnpalit@petra.ac.id, agust@petra.ac.id

ABSTRAK

Pada era teknologi yang semakin berkembang memberikan dampak terhadap peningkatan jumlah pengguna terhadap suatu sistem, sehingga beban kerja yang diterima *server* juga meningkat. Kondisi ini dialami pada PT. X dimana *server* tidak dapat menangani beban kerja yang terus meningkat dari waktu ke waktu, hal ini membuat *server* menjadi *overload* dan lambat dalam memberikan respon hingga berakhir pada kondisi *server down* dan tidak dapat diakses oleh *user*. Penelitian ini mencoba memberikan solusi terhadap masalah yang dihadapi PT. X dengan menerapkan sistem *microservices* dan Amazon *Elastic Container Service*. Dengan menerapkan *microservices* maka setiap *services* akan dipecah menjadi independen dan dapat meringankan beban kerja *server*. Selain itu dengan kombinasi Amazon ECS maka proses *scaling* akan menjadi lebih efektif hanya pada *service* yang mengalami kondisi *overload* sehingga proses *scaling* dapat menyesuaikan kondisi beban kerja pada *server* saat itu juga. Proses *scaling* akan membuat sistem dapat menambah atau mengurangi jumlah *task* maupun *server* yang berjalan tanpa kekurangan atau kelebihan penggunaan *resources*.Berdasarkan hasil analisis pengimplementasian *microservices* dan Amazon ECS pada sistem PT. X, dapat disimpulkan bahwa sistem *scalable microservices* menghasilkan *average response time* yang lebih rendah dengan selisih 805.56% dari *unscalable microservices* dan 38% dari monolitik, lalu *deviation* yang dihasilkan lebih rendah 902.22% dari *unscalable microservices* dan 216.87% dari monolitik, kemudian *throughput* yang dihasilkan lebih tinggi 22018.61 *request/minutes* dari *unscalable microservices* dan 24524.16 *request/minutes* dari monolitik. Untuk perbandingan *maximum concurrent user* antara sistem *scalable microservices*, *unscalable microservices*, dan monolitik sebesar 2000:1454:28. Selain itu penggunaan CPU pada sistem *scalable microservices* lebih rendah 20%-21% khususnya pada saat *login*, *generate access token*, dan *get schedule* jika dibandingkan dengan sistem *unscalable microservices* karena adanya sistem pembagian beban kerja dengan *task replica*. Selain itu penggunaan *resources* dapat menyesuaikan dengan kondisi beban kerja secara dinamis dan efisien.

Kata Kunci: Amazon *Elastic Container Service, microservices, scalability*

ABSTRACT

The technology age under development provides the impact of increasing the number of users in a system may also increase the workload received by the server. This condition is experienced in PT. X, where the server cannot handle the growing workload over time, this makes the server overloaded and slow in response until it gets to the server condition is down and unreachable by the user. This research tried to provide solutions to the problems faced by PT. X by applying a system of microservices and Amazon Elastic Container Service. By applying microservices then all services will be split into independent and can ease the workload of the server. Moreover, with the combination of Amazon ECS then the process of scaling will be more effective only on the service that is experiencing an overload condition so that the process of scaling can adjust the conditions of the workload on the server at that time. The scaling process will allow the system to increase or decrease the number of tasks performed without a lack or excessive use of resources. Based on analysis of the implementation of microservices and the Amazon ECS on the PT. X system, It can be concluded that the scalable microservices system produces a lower average response time with a difference of 805.56% compared to unscalable microservices and 38% compared to monolithic, then the resulting deviation is 902.22% lower than unscalable microservices and 216.87% lower than monolithic, then the resulting throughput is higher by 22018.61 requests/minutes from unscalable microservices and 24524.16 requests/minutes from monolithic. For a maximum concurrent user comparison between a scalable microservices system, an unscalable microservices, and monolithic of 2000:1454:28. In addition, the CPU usage of scalable microservices systems is 20%-21% lower, especially at login, generate access tokens, and get schedules when compared to unscalable microservices systems, due to workload sharing system with replication tasks. Additionally, the use of resources can adjust to workload conditions dynamically and efficiently.

Keywords: Amazon *Elastic Container Service, microservices, scalability*

1. PENDAHULUAN

Dalam sebuah arsitektur monolitik semua fungsionalitas dienkapsulasi menjadi aplikasi tunggal sehingga tiap *services* tidak bisa berjalan secara mandiri dan berjalan dalam satu *server* aplikasi yang sama. Proses pemeliharaan pada sistem monolitik dilakukan dalam satu *server* aplikasi, sehingga proses pemeliharaan dan pengembangan akan menjadi kompleks jika dilakukan dalam jangka waktu yang panjang, selain itu juga perubahan yang dilakukan pada sebagian kecil pada arsitektur ini akan mempengaruhi seluruh bagian dalam aplikasi. Proses *scaling* dalam arsitektur monolitik ini memiliki sebuah keterbatasan dan sulit dilakukan, dan biasanya proses *scaling* yang bisa dilakukan disini adalah *horizontal scale* jika *vertical scale* sudah mencapai batas [12]. Sehingga ketika melakukan *vertical scaling* hingga mencapai batas, kemudian baru dilakukan *horizontal scaling* maka akan membuang banyak *resources* karena penggunaan *resources* tidak dapat disesuaikan secara otomatis mengikuti beban kerja.

Pada penelitian kali ini akan menggunakan sebuah objek PT. X yang bergerak di bidang jasa kesehatan. Perusahaan ini memberikan jasa sebagai perantara proses *booking* jadwal konsultasi antara pihak rumah sakit dengan pasien dimana sistemnya masih menerapkan arsitektur monolitik. Arsitektur monolitik akan memberikan kemudahan dalam jangka waktu yang pendek bagi PT. X dalam proses pembuatan dan *deploying*, namun akan semakin kompleks jika digunakan untuk pengembangan fitur untuk jangka waktu yang panjang. Berdasarkan penelitian yang dilakukan oleh Makhloufi sebelumnya pada sistem monolitik terhadap jumlah *maximum concurrent user* yang dapat ditangani oleh *server* pada sistem monolitik menunjukkan hanya sekitar 200-300 *user* dari 600 *user* saja yang *request*-nya dapat ditangani oleh server [11]. Selain itu sistem booking pada PT. X cukup kompleks sehingga dapat membebani kinerja *server* jika diakses oleh ribuan *concurrent user* dan akan berdampak pada kondisi *server* yang *overload*.

Masalah-masalah yang ada dalam perusahaan ini dapat diatasi dengan menerapkan sebuah sistem arsitektur *microservices*. Arsitektur *microservices* ini sendiri berarti setiap layanannya bersifat independen, sehingga sistem lebih *fault tolerance* dalam menangani kegagalan suatu *service* [7]**.** Hal tersebut menunjukkan dengan menerapkan *microservices* maka setiap *service* disini memiliki infrastruktur sendiri dan setiap *service* bisa jadi berada dalam *server* yang berbeda sehingga jika salah satu layanan mengalami *server down*, maka *service* yang lain masih tetap bisa diakses oleh *user*. Dengan mengubah sistem arsitektur monolitik ke sistem arsitektur *microservices* menunjukkan bahwa arsitektur *microservices* dapat memenuhi kebutuhan skalabilitas dalam penggunaan *resources* masing-masing *services*, lebih mudah dalam pemeliharaan dan pengujian karena tiap *services* terpisah satu sama lain [6]. Sehingga *microservices* disini dapat membantu perusahaan dalam melakukan pengembangan serta pemeliharaan sistem. Selain itu dengan menerapkan sistem *microservices* pada PT. X akan mempermudah proses *scalability* karena tiap *service* sudah terpisah ke *container* yang berbeda sehingga proses *scaling-out* dapat dilakukan pada beberapa *services* yang mengalami peningkatan beban kerja.

Sistem Amazon ECS disini berperan sebagai pengatur *scaling* terhadap *container microservice*. Dengan adanya sistem *elastic container* maka *microservices* yang diterapkan dapat menyesuaikan penggunaan *resources* diantara *container* yang berjalan sehingga penggunaan *resource* lebih efisien dan menjadi lebih *scalable* [3]. Hal yang perlu diatur dalam Amazon ECS berupa *threshold*/*metric* berupa CPU *utilization* atau *memory utilization*, dan jumlah *container* yang ingin ditambahkan. Sehingga proses *scale-in* dan *scale-out* akan berjalan secara otomatis ketika sudah melebihi batas *metric* yang ditetapkan dalam sistem *auto scaling* [9]. Amazon ECS ini berfungsi sebagai pendukung dari sistem *microservices* yang akan dibuat pada sistem *booking* PT. X, sehingga jumlah *instance* dan *task* yang berjalan dapat disesuaikan dengan beban kerja terhadap suatu *services* dari sistem *microservices*, sehingga dapat meningkatkan performa, *response time server* terhadap *request user*, dan mengurangi resiko terjadinya *server down*.

Pada penelitian ini, konfigurasi dan penerapan *microservices* serta Amazon ECS pada PT. X bertujuan untuk menguji, membandingkan dan memperbaiki sistem perusahaan agar memudahkan pihak *developer* untuk mengembangkan dan memelihara sistem dalam jangka waktu yang panjang.

2. DASAR TEORI

2.1 Microservices

Microservices merupakan sebuah arsitektur dimana setiap modulnya diimplementasikan dan dioperasikan kedalam bagian yang kecil dan juga bersifat independen sehingga lebih cepat dalam pembuatan, pengembangan, pengoperasian, dan penskalaannya [8]. Sehingga disini setiap *service* akan berdiri secara independen baik dari segi fungsi maupun *database*. Dalam arsitektur *microservices*, jika salah satu *service* mengalami *down*, maka *services* yang lain masih dapat berjalan, karena tiap *services* dan *database* terpisah dalam *server* dan *container* yang berbeda. Selain itu arsitektur *microservices* ini sangat cocok untuk infrastruktur pada *cloud*, karena memberikan keuntungan dari segi elastisitas yang disediakan *cloud* dan menyediakan *resources* dengan cepat [4]. Sehingga *microservices* dan *cloud* dapat diintegrasikan untuk mendukung proses penskalaan bagi *microservices*.

Manfaat yang diperoleh dari *microservices* ini yaitu basis kode yang sederhana untuk karena setiap *services* yang terpisah, dapat memperbarui serta melakukan penskalaan pada suatu *service* secara terisolasi, dan memungkinkan setiap *services* dibuat dengan bahasa yang berbeda dengan memanfaatkan *middleware* [1]. Selain itu dengan mengimplementasikan *microservices* ini cukup berpengaruh besar dalam dalam suatu sistem yang cukup kompleks karena dengan menerapkan *microservices* ini maka sistem akan lebih mudah untuk proses penskalaan, pemeliharaan dan pengembangan karena sudah terbagi pada *services* yang kecil dan hal ini tidak akan mengganggu *services* yang lain.

2.2 Container

Container adalah sebuah unit *software* yang yang mengemas semua kode beserta dependensinya sehingga sebuah aplikasi dapat dijalankan dengan cepat dan dapat diandalkan dari satu lingkungan komputasi ke komputasi lainnya [5]. Dapat dikatakan *container* disini sebagai wadah untuk menampung suatu aplikasi sehingga dapat dijalankan pada sebuah sistem dengan lebih sederhana. Selain itu *container* juga mengandalkan sebuah isolasi virtual untuk menjalankan aplikasi tanpa mengganggu *host* OS ataupun *container* lain. *Container* disini sifatnya fleksibel dan dapat berjalan diatas sistem operasi tanpa *hypervisor* sehingga penggunaan *resource* akan disesuaikan dengan kebutuhan. *Container* disini dapat berjalan di mesin yang sama dan bisa saling berbagi sistem operasi dengan *container* lain sehingga dapat mengurangi penggunaan *memory*, sedangkan pada VM yang terjadi sebaliknya sehingga menggunakan lebih banyak *memory*.

2.3 Scalability

Scalability merupakan kemampuan suatu layanan untuk meningkatkan kapasitas *resources* dalam menangani peningkatan beban kerja [10]. Penyesuaian penggunaan *resources* tidak berkekurangan yang menyebabkan penurunan performa ataupun berlebihan yang menyebabkan adanya peningkatan biaya dan membuang *resources*, sehingga dengan adanya *scalability* ini dapat menyediakan *resources* sesuai dengan kebutuhan beban kerja pada saat itu. Untuk menerapkan sebuah proses *scalability* akan membutuhkan sebuah *metric* sebagai *threshold* untuk menentukan apakah sistem perlu menambahkan ataupun mengurangi penggunaan *resources* baik *container* maupun *server* dan hal ini akan menjaga keseimbangan sebuah sistem jika dalam suatu waktu jumlah *request* dari *user* meningkat ataupun menurun.

2.4 Amazon Elastic Container Service

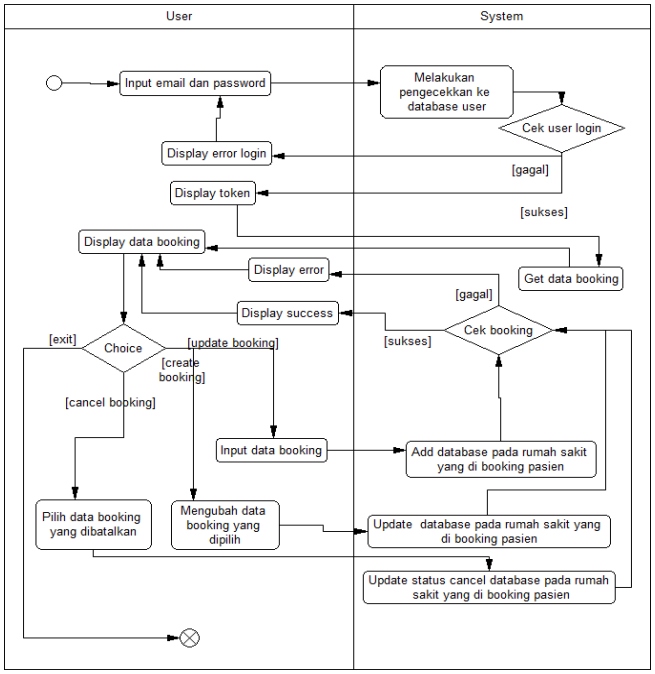
Amazon *Elastic Container Service* merupakan sebuah layanan penskalaan *container* untuk sebuah aplikasi termasuk *microservices*. Dalam Amazon ECS ini terdapat fitur yang dinamakan *auto scaling*, dimana hal ini memungkinkan *user* untuk menetapkan *metric* dalam melakukan *scale-in* maupun *scale-out* [9]. Dalam implementasinya sendiri *user* dapat menetapkan sebuah *threshold* berupa CPU atau *memory utilization* dan juga jumlah *container* yang akan ditambahkan ketika melebihi *threshold* tersebut serta batas penambahan jumlah *container*, sehingga proses *scaling* oleh ECS ini menjadi efisien dan dapat memenuhi permintaan *user* terhadap suatu *services* dalam jumlah yang besar tanpa mengurangi performa. Objek dasar yang dimiliki Amazon *Elastic Container Service* menurut Amazon [2], yaitu:

1. *Task*: tempat untuk mendefinisikan satu atau beberapa *container* dalam suatu aplikasi.
2. *Service*: merupakan sebuah konfigurasi pada Amazon ECS untuk menjalankan beberapa *task* secara bersamaan dalam sebuah *cluster*.
3. *Clusters*: Tempat untuk mengelompokkan *task* ataupun *services*. Dan umumnya jika *user* menggunakan mode peluncuran EC2 maka *cluster* akan menjadi tempat pengelompokkan *instance* dari *container* dan *Capacity Provider*.
4. *Container Agent:* Agen yang ada di dalam *container* tiap *instance* dan memiliki tugas untuk mengirimkan informasi penggunaan *resource* terhadap *tasks* menuju Amazon ECS. Dan Agen ini memiliki tugas juga untuk menjalankan dan menghentikan *tasks*.

3. DESAIN SISTEM

3.1 Desain Alur Aplikasi

Pada bagian ini menunjukkan sebuah proses yang terjadi pada *user* dan sistem pada saat melakukan *booking* konsultasi. Untuk alur proses *booking* konsultasi akan ditunjukkan pada Gambar 1.



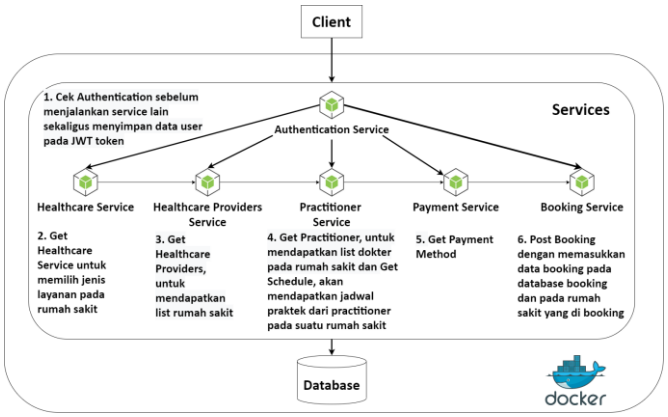
Gambar 1. *Activity Diagram* Sistem *Booking*

3.2 Desain Arsitektur Sistem

Pada bagian ini akan menjelaskan mengenai arsitektur monolitik yang dibangun oleh PT. X dan arsitektur *microservices* yang akan dibangun sehingga nantinya dapat diketahui proses kerja antara dua sistem. Hal ini bertujuan agar pembaca memahami perbedaan proses pemanggilan API dari suatu *service* antara dua arsitektur tersebut.

3.2.1 Desain Arsitektur Sistem Monolitik

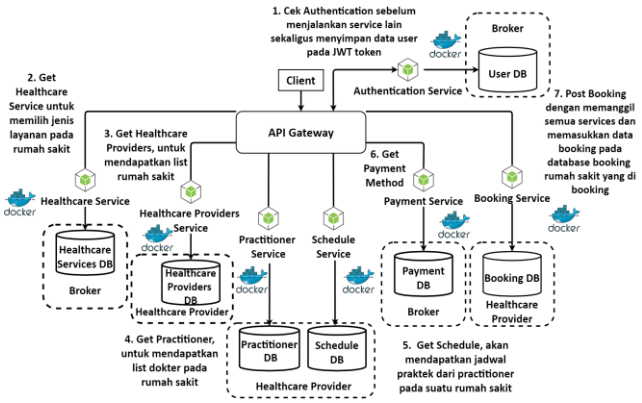
Sistem PT. X masih menerapkan arsitektur monolitik dimana semua *services* beserta *database* berada dalam satu *container* yang sama. Sehingga dapat dikatakan sistemnya disini tidak independen baik dari segi fungsi maupun *database*. Selain itu proses komunikasi antar satu *service* terhadap *service*s yang lain dilakukan secara langsung antar *services*. Selain itu ketika *user* melakukan *request*, maka diteruskan secara langsung ke *service* yang dituju. Sistem yang diterapkan sekarang memiliki beberapa resiko, salah satunya ketika salah satu *service down*, maka seluruh *services* dalam sistem juga akan terpengaruh dan tidak dapat diakses. Untuk penjelasan lebih lanjut akan ditunjukkan pada Gambar 2.



Gambar 2. Arsitektur Sistem Monolitik Perusahaan

3.2.2 *Desain Arsitektur Sistem Microservices*

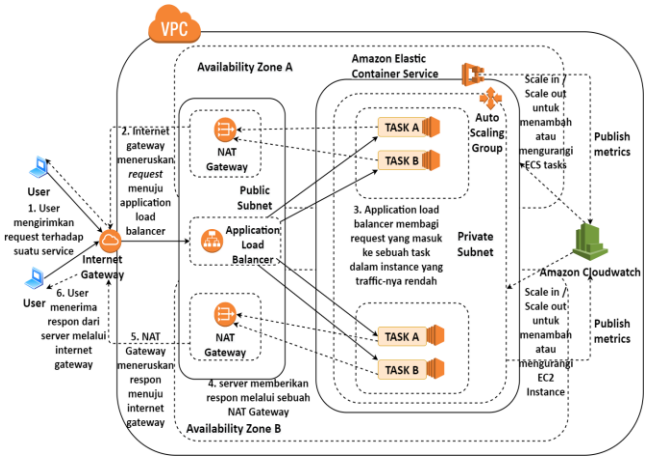
Sistem arsitektur *microservices* yang diusulkan disini dibuat sesuai dengan *services* yang ada pada sistem PT. X sebelumnya. *Microservices* disini memisahkan tiap *services* dan *database* masing-masing ke *container* yang berbeda dan diletakkan ke *server* yang berbeda. Selain itu sistem *microservices* juga memiliki sebuah *gateway* yang berfungsi sebagai perantara komunikasi antara satu *service* terhadap *services* lainnya. Selain itu terjadi pemecahan *service* antara *schedule service* dan *practitioner service*, hal ini dikarenakan penyesuaian dengan proses bisnis yang ada dalam aplikasi. Sistem *microservices* ini bersifat independen baik dari segi fungsi maupun *database* sehingga dapat mengurangi resiko kegagalan keseluruhan *services* untuk diakses ketika salah satu *service* mengalami *down*. Untuk proses komunikasinya dari *user* ke suatu *service* sendiri harus melewati sebuah *API gateway* yang nantinya akan meneruskan *request user* ke *service* yang dituju. Selain itu kepemilikan *database* terbagi menjadi 2 antara healthcare provider dan broker, dimana data-data seperti rumah sakit, layanan rumah sakit, dokter, jadwal praktek dokter dan booking akan disimpan oleh pihak rumah sakit, sedangkan metode pembayaran, data pasien, dan layanan kesehatan secara umum akan disimpan pada sistem broker. Untuk penjelasan lebih lanjut akan ditunjukkan pada Gambar 3.



Gambar 3. Arsitektur Sistem *Microservices*

3.2.3 *Desain Arsitektur pada Amazon Web Services*

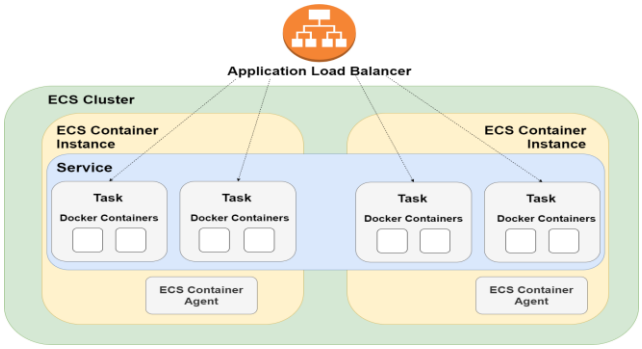
Desain arsitektur ini menggambarkan jalannya sebuah sistem yang dimulai dari sebuah *user* yang mengakses sebuah *service* melalui *Internet Gateway*. Kemudian *Internet Gateway* disini meneruskan *request* dari *user* menuju *Application Load Balancer* yang berada dalam sebuah *public subnet* dan berfungsi untuk membagi *traffic request* yang masuk ke sistem menuju *task* dalam suatu *instance* yang berada di *Availability Zone* berbeda dan berada dalam sebuah *private subnet* dengan tujuan menyeimbangkan sistem, agar tidak hanya salah satu *server* saja yang diakses sehingga menyebabkan *overload*. Setelah itu dari *request* yang masuk ini akan selalu di monitor oleh Amazon CloudWatch, karena jumlah *request* yang masuk akan mempengaruhi *metric* yang sudah ditetapkan, sehingga nantinya ketika sistem sudah tidak berjalan sesuai dengan *metric* yang ditetapkan maka akan dilakukan *scale-in* maupun *scale-out* antara *instance* atau *task* secara otomatis. Ketika sebuah *request* sudah diterima oleh salah satu *server*, maka *server* akan memberikan sebuah respon menuju NAT *Gateway* karena *server* berada dalam *private subnet* sehingga tidak dapat diakses dan memberikan respon secara langsung melalui *Internet Gateway*. Kemudian NAT *Gateway* akan meneruskan respon menuju *Internet Gateway* dan setelah diterima *Internet Gateway* maka respon akan ditampilkan ke *user*. Keseluruhan sistem arsitektur disini menggunakan VPC (*Virtual Private Cloud*) yang sama untuk menentukan jangkauan IP *address* (*Internet Protocol Address*), *subnet*, *Availability Zone*, NAT *Gateway*, dan *Internet Gateway*. Untuk penjelasan lebih lanjut mengenai arsitektur komunikasi dalam sistem akan ditunjukkan pada Gambar 4.



Gambar 4. Arsitektur Sistem Aplikasi

3.2.4 *Desain Arsitektur pada Amazon ECS*

Pada arsitektur Amazon ECS sendiri memiliki beberapa komponen dasar yaitu ECS cluster yang memungkinkan untuk mengelompokkan dan mengatur *task* ataupun sebuah *service* yang berada dalam beberapa ECS *container instance*. Kemudian di dalam setiap ECS *container instance* terdapat sebuah ECS *container agent* yang disini bertugas memberikan sebuah informasi penggunaan *resource* pada *container instance*, hal tersebut mempengaruhi keputusan Amazon ECS yang sudah terhubung dengan Amazon CloudWatch dalam memantau batas *metric* yang akan mempengaruhi apakah *task* masih perlu dijalankan atau dihentikan. Kemudian ada komponen *service* yang menjadi tempat untuk menjalankan *task*, dan bisa jadi satu *service* bisa berjalan dalam *container instance* yang berbeda karena dampak dari proses *scalability*. Dan kemudian ada sebuah *task* merupakan tempat untuk menjalankan *container*, selain itu satu *task* bisa memiliki satu atau banyak *container* di dalamnya. Dan komponen terakhir yaitu *Application Load Balancer* yang berfungsi untuk mengatur *traffic* yang masuk terhadap suatu *task* pada *service* untuk diarahkan ke *instance* yang berbeda sehingga sistemnya menjadi seimbang. Untuk penjelasan arsitektur *container* pada Amazon ECS akan ditunjukkan pada Gambar 5.



Gambar 5. Arsitektur Amazon ECS

4. PENGUJIAN SISTEM

Pengujian yang dilakukan terhadap sistem *microservices* akan terhubung dengan Amazon ECS sehingga dapat mendukung proses *scalability*. Pengujian ini akan membandingkan penggunaan *CPU*, *response time*, *throughput*, *deviation*, dan *maximum concurrent user* antara *scalable microservices*, *unscalable microservices*, dan sistem monolitik dari PT. X dengan menggunakkan *tools* Apache Jmeter dan akan diatur urutan *request* seperti saat melakukan proses *booking* konsultasi.

4.1 Pengujian Maximum Concurrent User

Berikut ini merupakan hasil perbandingan *maximum concurrent user* yang dapat mengakses *server* dalam waktu yang bersamaan antara sistem *unscalable microservices*, *scalable microservices*, dan monolitik dari 3 *scenario* yang ditunjukkan pada Tabel 1.

Tabel 1. Hasil pengujian *maximum concurrent user*

Sistem	Concurrent User (orang)		
	500	**1000**	**2000**
Monolithic	87	94	28
Unscalable microservices	500	1000	1454
Scalable Microservice	500	1000	2000

Berdasarkan perbandingan jumlah *maximum concurrent user* dari tabel diatas, dapat dilihat bahwa jumlah *maximum concurrent user* yang dapat ditangani oleh sistem *scalable microservices* jauh lebih tinggi dan stabil hingga berhasil melakukan *booking* konsultasi jika dibandingkan dengan sistem *unscalable microservices* dan monolitik yang mengalami ketidakstabilan sehingga jumlah *concurrent user* yang berhasil melakukan *booking* konsultasi semakin menurun khususnya pada *scenario* 2000 *concurrent user*. Sehingga dengan adanya proses *scale-out* terhadap *task patient service* dan *schedule service* pada sistem *scalable microservices* memberikan dampak yang besar bagi sistem dalam menangani peningkatan beban kerja tanpa menghasilkan sebuah respon *error*. Dari tabel diatas maka dapat disimpulkan jumlah perbandingan *maximum concurrent user* antara sistem *scalable microservices*, *unscalable microservices*, dan monolitik yaitu sebesar 2000:1454:28, angka perbandingan ini didapat dari jumlah *user* yang berhasil melakukan *booking* konsultasi pada tiap sistem di *scenario* 2000 *concurrent user*.

4.2 Pengujian Penggunaan CPU

Berikut ini merupakan hasil perbandingan penggunaan *CPU* antara sistem *unscalable microservices* dan *scalable microservices* dari pengujian *scenario* 2000 *concurrent user* yang ditunjukkan pada Tabel 2.

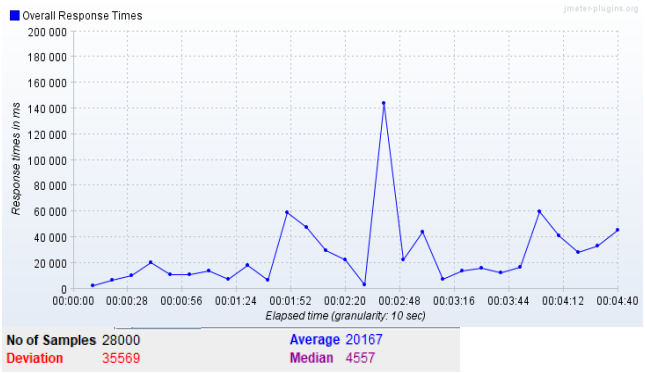
Tabel 2. Hasil pengujian *CPU* pada *unscalable microservices*

Request	CPU Usage %	
	Scalable Microservices	Unscalable Microservices
Login	29.41%	50.92%
Generate Access Token		
Get Healthcare Service	6.83%	9.14%
Get Healthcare Provider	9.92%	14.01%
Get Practitioner	6.44%	7.97%
Get Schedule	29.32%	50.48%
Get Payment	6.36%	7.69%
Post Booking	5.31%	10.94%

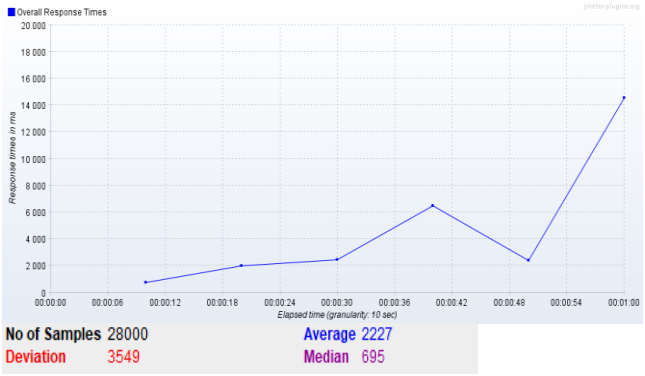
Berdasarkan kedua tabel diatas, maka dapat dilihat bahwa jumlah penggunaan *CPU* tiap *services* pada sistem *scalable microservices* jauh lebih rendah jika dibandingkan dengan *unscalable microservices*. Perbedaan penggunaan *CPU* yang paling signifikan terlihat pada saat melakukan *login*, *generate access token*, dan *get schedule* yang memiliki perbedaan antara 20-21%, sedangkan penggunaan *CPU* pada *service* lainnya cenderung lebih stabil dan serupa pada kedua sistem. Perbedaan yang cukup signifikan pada ketiga *request* tersebut dikarenakan adanya *task replica patient service* dan *schedule service* yang terbentuk sehingga penggunaan *CPU* pada kedua *services* tersebut menjadi lebih rendah pada sistem *scalable microservices* karena adanya sistem pembagian beban kerja antara *task* utama dan *task replica* sehingga penggunaan *CPU* pada tiap *task* menjadi lebih rendah.

4.3 Pengujian Response Time dan Deviation

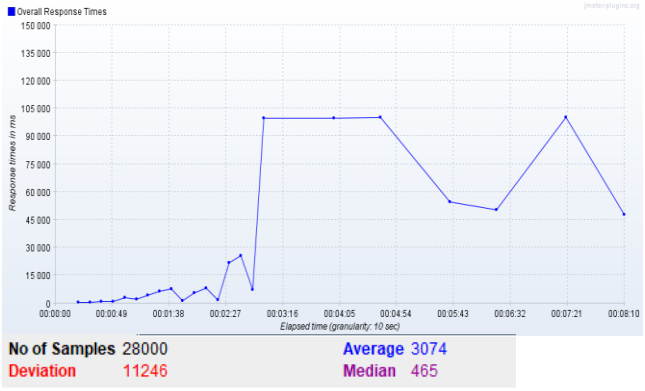
Pada bagian ini akan menjelaskan perbandingan rata-rata *response time* dan *deviation* antara sistem monolitik, *unscalable microservices*, dan *scalable microservices*. Untuk perbandingan grafik *average response time* dan *deviation* masing-masing sistem dari gabungan keseluruhan *scenario* dapat dilihat pada Gambar 6, Gambar 7, dan Gambar 8.



Gambar 6. *Average response time* dan *deviation* keseluruhan *scenario* pada sistem *unscalable microservices*



Gambar 7. *Average response time* dan *deviation* keseluruhan *scenario* pada sistem *scalable microservices*



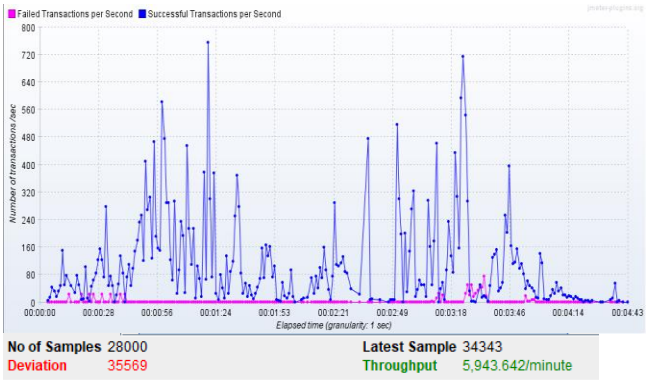
Gambar 8. *Average response time* dan *deviation* keseluruhan *scenario* pada sistem monolitik

Berdasarkan ketiga gambar diatas maka dapat dilihat bahwa *average response time* yang dihasilkan oleh sistem *scalable microservices* jauh lebih rendah jika dibandingkan dengan sistem *unscalable microservices* yang memiliki selisih 805.56% dan monolitik dengan selisih 38%. Sedangkan untuk *deviation* pada *scalable microservices* juga jauh lebih rendah jika dibandingkan dengan *unscalable microservices* yang memiliki selisih 902.22% dan monolitik dengan selisih 216.87%. Dengan tingkat *average response time* dan *deviation* yang lebih rendah, menunjukkan

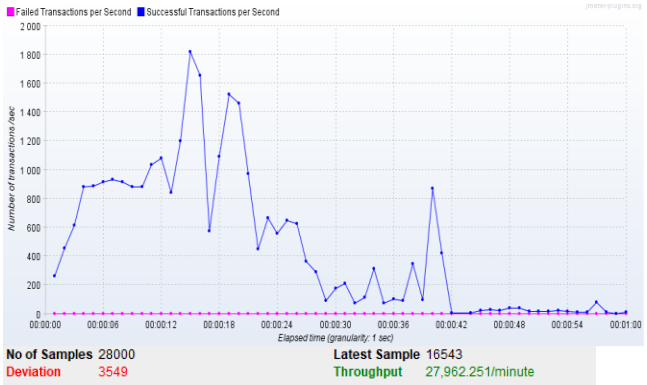
bahwa sistem *scalable microservices* dapat memberikan respon menuju *user* dengan lebih cepat dan stabil mendekati rata-rata *response time*, hal ini terjadi dikarenakan faktor *scale-out* pada *task patient service* dan *schedule service*, sehingga beban kerja khususnya pada saat melakukan *login, generate access* token, dan *get schedule* dibagi dengan *task replica* pada *server replica* dan membuat pemrosesan *request* dapat berlangsung dengan lebih cepat tanpa harus membuat *request* yang masuk saling menunggu. Selain itu tingkat *deviation* dan *average response time* pada sistem monolitik jauh lebih rendah jika dibandingkan dengan *unscalable microservices* dikarenakan tingginya tingkat *error* karena *server* pada sistem monolitik mengalami *overload*, sehingga *server* mengembalikan respon *error* dengan lebih cepat dan menghasilkan sebuah *response time* yang hampir serupa sehingga dapat dikatakan tingkat variasi *response time* menjadi lebih rendah. Lalu total waktu *testing* pada sistem *scalable microservices* berlangsung lebih cepat hanya sekitar 1 menit, sedangkan *unscalable microservices* sekitar 4 menit, dan monolitik sekitar 8 menit, dengan waktu *testing* yang cukup rendah pada sistem *scalable microservices* ini menandakan bahwa kemampuan *server* dalam menangani dan memberikan respon terhadap *request user* semakin cepat.

4.4 Pengujian Throughput

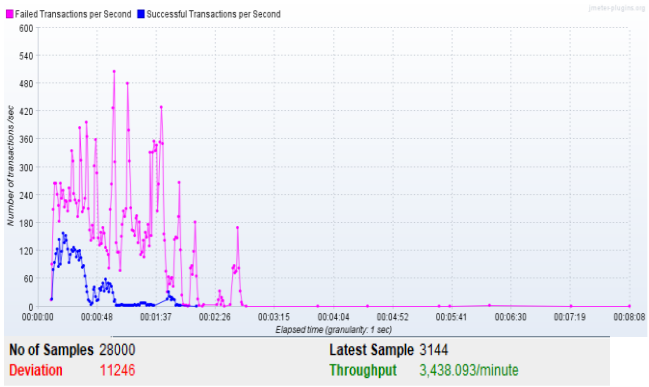
Pada bagian ini akan menjelaskan perbandingan *throughput* antara sistem monolitik, *unscalable microservices*, dan *scalable microservices* untuk mengukur kinerja *server* dalam menangani jumlah *request* detiknya. Untuk perbandingan grafik *throughput* masing-masing sistem dari gabungan keseluruhan *scenario* dapat dilihat pada Gambar 9, Gambar 10, dan Gambar 11.



Gambar 9. *Throughput* keseluruhan *scenario* pada sistem *unscalable microservices*



Gambar 10. *Throughput* keseluruhan *scenario* pada sistem *scalable microservices*



Gambar 11. *Throughput* keseluruhan *scenario* pada sistem monolitik

Berdasarkan ketiga gambar diatas maka dapat dilihat bahwa *throughput* yang dihasilkan oleh sistem *scalable microservices* jauh lebih lebih tinggi jika dibandingkan dengan sistem *unscalable microservices* yang memiliki selisih 22018.61 *request/minutes* dan monolitik dengan selisih 24524.16 *request/minutes*. Dengan tingkat *throughput* yang lebih tinggi, menunjukkan bahwa sistem *scalable microservices* dapat memproses *request* tiap detiknya dengan lebih tinggi dan stabil jika dibandingkan kedua sistem lainnya, selain itu respon yang dikembalikan dari proses *request* tersebut berupa respon kerberhasilan tanpa adanya respon kegagalan dari ketiga *scenario* tersebut. Hal ini dapat terjadi karena adanya faktor *scale-out* pada *task patient service* dan *schedule service*, selain itu pembagian beban kerja dapat dilakukan pada *task* utama dan *task replica* sehingga pemrosesan *request* tiap detiknya dapat dilakukan dengan lebih cepat karena sistem pembagian beban kerja tersebut. Walaupun terjadi peningkatan beban kerja yang tidak seimbang dengan penambahan *task replica schedule service* pada *scenario* 2000 *concurrent user* di sistem *scalable microservices*, namun tingkat *throughput* yang dihasilkan masih jauh lebih tinggi dari kedua sistem lainnya. Sehingga dapat dikatakan sistem *scalable microservices* memiliki performa yang jauh lebih baik dan cepat dalam memproses *request* yang masuk tiap detiknya.

5. KESIMPULAN DAN SARAN

5.1 Kesimpulan

- Sistem *scalable microservices* memiliki tingkat *throughput* yang lebih tinggi jika dibandingkan dengan sistem *unscalable microservices* dengan perbedaan 22018.61 *request/minutes* dan monolitik dengan perbedaan 24524.16 *request/minutes* dari keseluruhan *scenario*. Selain itu jika terjadi peningkatan beban kerja sebanyak 2x lipat dengan diimbangi penambahan jumlah *task replica* pada *patient service* dan *schedule service* sebanyak 2x lipat maka akan menghasilkan peningkatan *throughput* sebesar 100%.
- *Average response time* pada sistem *scalable microservices* cukup rendah jika dibandingkan dengan sistem monolitik dengan perbedaan lebih rendah 38%, sedangkan dengan *unscalable microservices* memiliki perbedaan lebih rendah 805.56% dari keseluruhan *scenario*. Setiap penambahan beban kerja sebanyak 2x lipat dengan diimbangi penambahan jumlah *task replica* pada *patient service* dan *schedule service* sebanyak 2x lipat maka dapat menjaga kestabilan tingkat *average response time*.

- Tingkat *deviation* pada sistem *scalable microservices* cukup rendah jika dibandingkan dengan sistem monolitik dengan perbedaan lebih rendah 216.87%, sedangkan dengan *unscalable microservices* memiliki perbedaan lebih rendah 902.22% dari keseluruhan *scenario*. Setiap penambahan beban kerja sebanyak 2x lipat dengan diimbangi penambahan jumlah *task replica* pada *patient service* dan *schedule service* sebanyak 2x lipat maka dapat menjaga kestabilan tingkat *deviation*.
- Untuk tingkat perbandingan *maximum concurrent user* khususnya pada *scenario* 2000 *concurrent user* maka dapat terlihat perbandingan pada sistem *scalable microservices*, *unscalable microservices*, dan monolitik sebesar 2000:1454:28. Dengan penambahan 1 *task replica* dari *schedule service* dan *patient service* dapat menangani penambahan 1000-1500 *concurrent user*. Selain itu penggunaan *CPU* pada sistem *scalable microservices* lebih rendah 20-21% dari *unscalable microservices* khususnya pada saat melakukan *login*, *generate access token*, dan *get schedule*.

5.2 Saran

- Menerapkan proses *sharding* dan *replication* pada *database* MongoDB agar dapat meningkatkan *throughput* karena adanya pembagian tugas *read* pada *slave node* dan *write* pada *master node*, selain itu dapat mencegah hilangnya data jika salah satu *node* mengalami *down* atau *crash* karena setiap *node* menyimpan data yang sama.
- Pengujian dapat dilakukan dengan jenis *load balancer* yang berbeda untuk menentukan jenis *load balancer* mana yang lebih baik dalam membagi *traffic* yang masuk.
- Sistem monolitik dapat di-*deploy* di *cloud* agar hasil pengujian menjadi seimbang karena menggunakan infrastruktur yang sama serta *resources* seperti *CPU* dapat diperbesar agar dapat mengimbangi sistem *microservices* dengan beberapa *server*.

6. DAFTAR PUSTAKA

[1] Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M., & Steinder, M. 2015. Performance Evaluation of Microservices Architectures Using Containers. *2015 IEEE 14th International Symposium on Network Computing and Applications*, 27–34. DOI=https://doi.org/10.1109/NCA.2015.49.

[2] Amazon Web Services Inc. 2019. *What is Amazon Elastic Container Service?* Amazon Elastic Container Service Developer Guide. URI=https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html.

[3] Cusack, G., Nazari, M., Goodarzy, S., Oberai, P., Rozner, E., Keller, E., & Han, R. 2019. Efficient Microservices with Elastic Containers. *Proceedings of the 15th International Conference on Emerging Networking EXperiments and Technologies*, 65–67. DOI=https://doi.org/10.1145/3360468.3368180.

[4] Di Francesco, P., Lago, P., & Malavolta, I. 2019. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, *150*(February), 77–97. DOI=https://doi.org/10.1016/j.jss.2019.01.001.

[5] Docker. 2020. *What is a Container? | App Containerization | Docker*. Docker.Com. URI=https://www.docker.com/resources/what-container.

[6] Habibullah, S., Liu, X., Tan, Z., Zhang, Y., & Liu, Q. 2019. Reviving Legacy Enterprise Systems with Micro service-Based Architecture with in Cloud Environments. *8th International Conference on Soft Computing, Artificial Intelligence and Applications*, 173–186. DOI=https://doi.org/10.5121/csit.2019.90713.

[7] Haselböck, S., Weinreich, R., & Buchgeher, G. 2017. Decision guidance models for microservices. *Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems*, *Part F1305*, 1–10. DOI=https://doi.org/10.1145/3123779.3123804.

[8] Jamshidi, P., Pahl, C., Mendonca, N. C., Lewis, J., & Tilkov, S. 2018. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software*, *35*(3), 24–35. DOI=https://doi.org/10.1109/MS.2018.2141039.

[9] Klinaku, F., Frank, M., & Becker, S. 2018. CAUS. *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 93–98. DOI=https://doi.org/10.1145/3185768.3186296.

[10] Lehrig, S., Eikerling, H., & Becker, S. 2015. Scalability, Elasticity, and Efficiency in Cloud Computing. *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*, *May*, 83–92. DOI=https://doi.org/10.1145/2737182.2737185.

[11] Makhloufi, Y. 2015. *Moving from Monolithic to Distributed Architecture*. (Thesis, Kungliga Tekniska Högskolan School of Information and Communication Technology). URI=https://pdfs.semanticscholar.org/2c74/a9466704e3d764a0bb0880e676ccc350712e.pdf.

[12] Ponce, F., Marquez, G., & Astudillo, H. 2019. Migrating from monolithic architecture to microservices: A Rapid Review. *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, *2019-Novem*(September), 1–7. DOI=https://doi.org/10.1109/SCCC49216.2019.8966423.