KESATRIA: Jurnal Penerapan Sistem Informasi (Komputer & Manajemen)
Terakreditasi Nomor 204/E/KPT/2022 | Vol. 4, No. 4, Oktober (2023), pp. 830-840

Analisis Performansi Layanan Web Menggunakan Arsitektur Microservice Dan Monolitik

Siti Amatullah Karimah[1], Haris Hamdani Latif [2], Sidik Prabowo[3]
[1,2,3]Fakultas Informatika, Universitas Telkom, Bandung, Indonesia
E-mail: [1]karimahsiti@telkomuniversity.ac.id,
[2]harishl@students.telkomuniversity.ac.id, [3]pakwowo@telkomuniversity.ac.id

Abstract

Container-based virtualization technology is currently popularly used across all cloud platforms and is predicted to continue for the next few years. The use of this container technology will make it easier and save resources used for services. Coupled with the development of the current web architecture which is increasingly being developed and used for commercial purposes, including Microservice and Monolithic. This Microservice architecture divides its services into smaller parts based on functionality. Meanwhile, Monolithic Architecture is referred to as conventional architecture because in it services become a unified whole. For this reason, a test scenario was carried out to determine the performance of the two web architectures. In this study, load testing was carried out with the number of requests 50, 100, 500, and 1000 on Microservice and Monolithic to show scalability. The results show that the Monolithic service is superior with an average CPU usage on AWS of 83% while Microservice is at 99%. CPU in Monolithic Docker Container is 92% while Microservice is 30% for each service. For Memory Usage, Microservice gets an average of 14% while for Monolithic services it is 12%. Response Time was obtained at 1497.31 ms for Microservice and 89.02 ms for Monolithic. In testing the availability by terminating/stopping the service in the Microservice service then it is reactivated and takes 2 seconds, while in the Monolithic service it takes 3 seconds to restore the service. When the service is turned off, the Microservice service can still run normally, only the dead service will experience interference, this is inversely proportional to the Monolithic service which will be completely dead when the service is turned off.

***Keywords**: Microservice, Monolithic, Container, Virtualization, Scalability, Availability*

Abstrak

Teknologi Container-based virtualization saat ini menjadi populer digunakan di seluruh platform Cloud dan diprediksi akan terus berlanjut untuk beberapa tahun ke depan, penggunaan teknologi Container ini akan memudahkan serta dapat menghemat resource yang digunakan untuk layanan. Ditambah lagi dengan perkembangan arsitektur web saat ini yang semakin berkembang dan digunakan untuk komersial diantaranya yaitu Microservice dan Monolitik. Arsitektur Microservice ini membagi layanannya menjadi bagian-bagian yang lebih kecil berdasarkan fungsionalitas. Sedangkan Arsitektur Monolitik ini disebut sebagai arsitektur konvensional karena di dalamnya layanan menjadi satu kesatuan utuh. Untuk itu dilakukan skenario pengujian untuk mengetahui performansi dari kedua arsitektur web. Pada penelitian ini dilakukan load testing dengan jumlah request 50, 100, 500, dan 1000 pada Microservice dan Monolitik untuk menunjukan skalabilitas. Didapatkan hasil bahwa layanan Monolitik lebih unggul dengan rata – rata penggunaan CPU pada AWS sebesar 83% sedangkan Microservice di angka 99%. CPU pada Docker Container Monolitik sebesar 92% sedangkan Microservice 30% untuk setiap service. Untuk Memory Usage layanan Microservice mendapatkan hasil rata – rata 14% sedangkan untuk layanan Monolitik 12%. Response Time di dapatkan waktu 1497,31 ms untuk Microservice dan 89,02 ms untuk Monolitik. Pada pengujian

ISSN: 2720-992X

availabilitas dengan melakukan terminate/stop pada service di layanan Microservice lalu diaktifkan kembali dan membutuhkan waktu 2 detik, sedangkan pada layanan Monolitik membutuhkan waktu 3 detik untuk mengembalikan service. Pada saat service di matikan layanan Microservice masih bisa berjalan dengan normal, hanya pada service yang mati saja yang akan mengalami gangguan, hal ini berbanding terbalik pada layanan Monolitik yang akan mati total saat service dimatikan.

***Kata kunci**: Microservice, Monolitik, Container, Virtualisasi, Skalabilitas, Availabilitas*

1. Pendahuluan

Komputasi awan atau yang biasa dikenal dengan sebutan *Cloud Computing* merupakan sebuah teknologi yang relatif baru dikalangan masyarakat Indonesia dan saat ini sudah mulai popular untuk digunakan, teknologi ini bisa mengolah berbagai proses komputasi data dengan bantuan internet dalam menjalankannya [1] . Penggunaan teknologi *cloud computing* ini juga bisa meningkatkan *scalability*, *availability*, dan pengembangan kedepannya yang akan menjadi tantangan untuk kedepannya [2]. Teknologi virtualisasi yang merupakan bagian dari *cloud computing* juga menjadi populer terutama *Container-based virtualization* yang membuat aplikasi menjadi terasa ringan saat digunakan meskipun masih dalam mesin yang sama [3]. *Container* ini merupakan metode yang dipakai untuk menjalankan sebuah aplikasi ataupun web yang sifatnya tertutup dengan cara membungkus file atau *enkapsulasi* [4], container ini juga di desain agar bisa bekerja dengan kinerja tinggi dan mempermudah dalam pengembangan. Salah satu platform virtualisasi berbasis *Container* yang paling banyak digunakan adalah *Docker*. *Docker* merupakan platform yang terkenal yang biasa digunakan untuk mengemas perangkat lunak di dalam *container* dan menjalankan perangkat lunak tersebut [5]. Di samping itu evolusi arsitektur web juga ikut berkembang karena didorong oleh kebutuhan dari masyarakat yang menginginkan arsitektur web yang lebih baik lagi, saat ini ada 2 arsitektur yang mendominasi dalam pengembangan web atau aplikasi di antaranya *Monolitik* dan *Microservice* [6]. *Microservice* adalah sebuah arsitekur web yang di dalamnya terpisah-pisah secara fungsionalitasnya dan saling berkomunikasi satu dengan yang lain [7]. Hal ini berbeda dengan *Monolitik* yang dibangun menjadi satu kesatuan utuh [8]. Saat ini kedua arsitektur ini masih sering dipakai untuk mengambangkan sebuah aplikasi. Penggunaan arsitektur *Microservice* juga diyakini mempermudah proses *maintenance*, *reusability*, *scalability*, *availability*, dan *deployment*. Oleh sebab itu dilakukan penelitian untuk membandingkan antara arsitektur *Microservice* dan *Monolitik* manakah yang lebih baik secara performansinya dalam mengelola traffic yang tinggi serta bagaimana layanan itu mempertahankan *service* nya jika suatu saat terjadi gangguan baik yang disebabkan oleh kesalahan manusia maupun sistem. Di sini terdapat beberapa parameter yang kami uji diantaranya *CPU Utilization*, *Memori Usage*, *Container Creating Time*, dan *Response Time* dari layanan tersebut.

Tabel 1. Studi Terkait

No.	Judul Paper	Tahun	Sistem	Metode	*Output*
1.	Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation	2022	Container Based	Docker Container	Melakukan beberapa pembandingan layanan Microservice dan Monolitik dari aspek Scalability, Skenario Deployment hingga pengaruh penambahan jumlah resource instance terhadap layanan.

No.	Judul Paper	Tahun	Sistem	Metode	*Output*
2.	From Monolithic Systems to Microservices: A Comparative Study of Performance	2020	Container Based	Container Orchestration	Pengujian CPU Consumption, Memori Consumption, Network Transmision/Reception, Disk Writing dan Reading Speed pada layanan Monolitik dan Microservice.
3.	A Comparative Review of Microservice and Monolitik Architectures	2018	Container Based	Container Orchestration	Komparasi arsitektur Microservice dan Monolitik dengan melakukan loadtesting pada web dan melakukan pengujian throughput pada layanan tersebut.

Pada penelitian sebelumnya tentang *A Compare Review of Microservice and Monolitik Architectures*, bertujuan untuk membandingkan antara arsitektur *Microservice* dan *Monolitik* dalam berbagai skenario pengujian. Untuk skenario pengujian yang pertama dengan cara memberi *load* di bawah normal atau beban kurang dari 100 pengguna dengan hasil bahwa aplikasi *Microservice* dan *Monolitik* memiliki kinerja yang sama atau tidak terlalu terlihat perbedaannya. Sedangkan jika jumlah pengujian ditambah sampai 7000, kinerja *Microservice* bisa lebih baik, akan tetapi perbedaan kinerja antara *Microservice* dan *Monolitik* rata -rata hanya sekitar 0,87% yang menunjukkan tidak terlihat perbedaan yang signifikan. Pada pengujian *throughput* juga *Monolitik* masih lebih unggul dibandingkan dengan *Microservice*, tetapi dari segi banyaknya permintaan pada *Microservice* dapat memproses lebih banyak permintaan dibandingkan dengan *Monolitik*. Untuk skenario pengujian yang kedua yaitu dengan *concurency testing* dimana user melakukan pengujian bermasa-sama di pengujian ini diperoleh hasil dari segi *throughput Monolitik* masih lebih unggul dibandingkan dengan *Microservice* dengan perbandingan perbedaan rata-rata 6%. Pengujian *response time* juga sama pada *Monolitik* memiliki waktu *response time* yang lebih rendah dari pada *Microservice*. Pada penelitian ini juga dapat disimpulkan bahwa Aplikasi *Monolitik* lebih unggul jika digunakan untuk aplikasi sederhana yang hanya digunakan oleh beberapa orang saja dengan beberapa parameter yang telah diujikan sebelumnya [6].

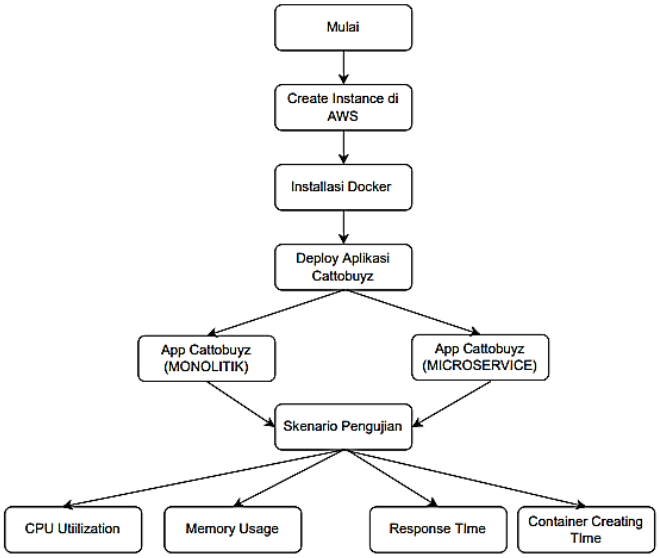
Pada penelitian sebelumnya tentang *From Monolithic Systems to Microservices: A Comparative Study of Performance*, bertujuan untuk membandingkan arsitektur *Microservice* dan *Monolitik* secara performa dengan berbagai skenario, dari segi *CPU Consumtion* aplikasi *Microservice* memakan lebih banyak *resource* dengan perbandingan 7,85% untuk *Microservice* dan 2,38% untuk *Monolitik*. Pada *Microservice* lebih banyak memakai *CPU* karena pada aplikasi *Microservice* memiliki sumber daya dan basis datanya sendiri-sendiri serta mengonsumsi sumber daya komputasi masing-masing *service* serta memiliki kerumitan dalam mengelola dan mengintegrasikan proses di dalam aplikasi *Microservice*. Namun untuk konsumsi *Memory* pada *Microservice* memperoleh hasil yang lebih banyak dengan perbandingan *Microservice* 2,88% sedangkan untuk *Monolitik* 4,52%. Secara *Network Transmission* aplikasi *Microservice* memperoleh hasil sebanyak 4,50% sedangkan *Monolitik* 3,12% sedangkan *Network Reception Microservice* diangka 10,48% kemudian untuk *Monolitik* 3,28% dalam hal ini berarti *Microservice* memperoleh hasil yang lebih baik. Pengujian selanjutnya pada *Disk Writing* aplikasi *Microservice* dan *Monolitik* memperoleh hasil yang seimbang 3,83% tetapi dari *Disk Reading* aplikasi *Monolitik* memperoleh hasil lebih tinggi dengan 4,50% sedangkan *Microservice* 3,16%. Pada penelitian ini dianggap wajar karena aplikasi perlu menangani sejumlah data yang terus berjalan di layanan *Microservice* [9].

Pada penelitian sebelumnya tentang *Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation* bertujuan untuk membandingkan aplikasi yang berarsitektur *Microservice* vs *Monolitik* dari berbagai hal. Salah satunya membandingkan performansi dari beberapa platform *cloud* dan beberapa *resource* yang dipakai dari yang paling kecil sampai yang terbesar dan didapatkan kesimpulan yang hampir sama dengan penelitian sebelumnya yaitu arsitektur *Microservice* bukanlah yang paling cocok untuk setiap konteks. Namun penggunaan aplikasi *Monolitik* bisa menjadi sebuah pilihan yang lebih baik jika ingin membuat sistem yang sederhana dan tidak mendukung penggunaan besar kemudian layanan *Microservice* akan berjalan maksimal juga jika diimbangi dengan *resource* yang besar juga [10].

Pada penelitian saya tentang Analisis Performansi Layanan Web Menggunakan Arsitektur Microservice dan Monolitik ini menjelaskan perbandingan performansi pada dua arsitektur yang berbeda. Ada 2 aspek yang diujikan diantaranya *Scalability* dan *Availability*. Pada penelitian sebelumnya sudah menjelaskan tentang pengujian pada aspek *Scalability*, tetapi belum ada yang menjelaskan pada aspek *Availability*. Aspek *Availability* ini penting pada sebuah layanan untuk menjamin ketersediaan layanan dan informasi yang bisa diakses oleh user. Pada penelitian ini dilakukan pengujian pada aspek *Scalability* dan *Availability*.

2. Metodologi Penelitian

Penelitian ini menggunakan dua arsitektur layanan yaitu *Microservice* dan *Monolitik* di bawah ini kami sajikan *flowchart* dari penelitian ini.



Gambar 1. Perancangan Pengerjaan

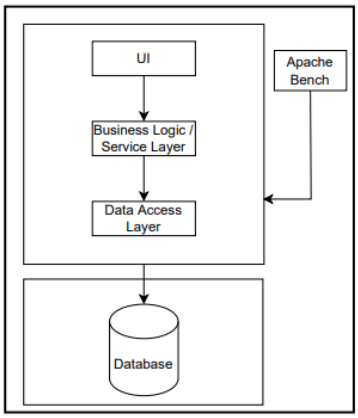
Hal pertama yang dilakukan pada penelitian ini yaitu dengan membuat *instance* pada AWS di sini membuat 3 *Instance* yaitu *Monolitik, Microservice* dan *Database*. Setelah itu melakukan installasi *Docker* pada ketiga *instance* tesebut. Selanjutnya melakukan *deploy* Aplikasi *Cattobuyz* dalam bentuk *docker image* yang berarsitektur *Microservice* dan *Monolitik* di dalam *Docker Container*. Setelah aplikasi sudah berhasil di *deploy* maka langkah selanjutnya melakukan pengujian, ada beberapa aspek yang diuji yaitu Aspek Skalabilitas dan Aspek Availabilitas, pada aspek Skalabilitas dilakukan *load testing* dengan jumlah *request* yang telah ditentukan sebelumnya, dari *load testing* tersebut didapatkan persentase dari *CPU Utilization, Memori Usage* dan *Response Time*. Aspek selanjutnya yaitu Availabilitas yaitu dengan melakukan *terminate* atau *stop* pada *service* sampai *service* mati dan tidak dapat diakses, kemudian dilakukan start *service* kembali

pada *container* lalu bisa didapatkan *Container Creating Time* dari kedua aplikasi dengan arsitektur yang berbeda.

Tabel 2. Spesifikasi Instance AWS

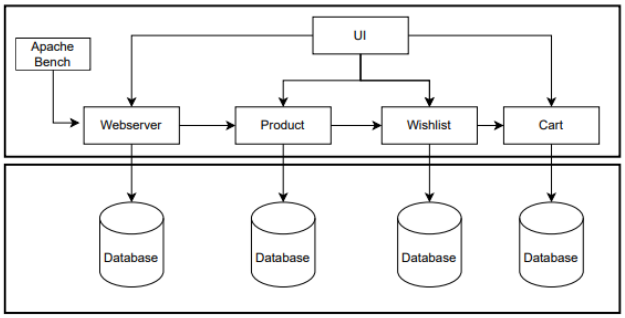
Komponen	Spesifikasi
Processor	1vCPUs, 2.5 GHz, Intel Xeon Family
RAM	1 GiB
HDD	8 GiB

Pada Tabel 2 menjelaskan *Service cloud* yang digunakan adalah AWS, pada penelitian ini dibuat 3 *Instance* yaitu *Instance Monolitik*, *Instance Microservice* dan *Instance Database*. Pada Instance *Monolitik* berisi *Container Monolitik* beserta database, pada *Instance Microservice* berisikan beberapa *Container* dari *Microservice*, yang terakhir *Instance Database* berisikan file *database* dari Aplikasi *Microservice*. Tiap instance memiliki spesifikasi yang sama.



Gambar 2. Arsitektur Aplikasi Cattobuyz Monolitik

Pada Gambar 2 menjelaskan Aplikasi dengan arsitektur *Monolitik* semua bagian dari mulai *UI*, *Business Logic* atau *Service Layer* dan *Data Access Layer* menjadi satu kesatuan aplikasi dalam 1 *container* beserta databasenya. Terdapat juga Apache Bench yang digunakan untuk melakukan *load testing* pada *container*. Aplikasi berarsitekur Monolitik ini di buat dalam 1 *instance* (*instance monolitik*).



Gambar 3. Arsitektur Aplikasi Cattobuyz Microservice

Pada Gambar 3 diketahui bahwa bahwa pada arsitektur *Microservice* aplikasi akan dipecah menjadi beberapa *service* berdasarkan fungsionalitasnya, sebagai contoh *service cart*, *service wishlist, dll*. Terdapat juga Apache Bench yang digunakan untuk melakukan *load testing* pada tiap *container / service*. Pada *Microservice* ini dibagi menjadi 2 *instance* yaitu *Instance Microservice* dan *Instance Monolitik*. Perangkat lunak yang digunakan pada penelitian ini antara lain:

1. *AWS* sebagai *instance* untuk menjalankan *service.*
2. *Docker* sebagai *virtualisasi Container*.
3. *MobaXterm* sebagai *controller Container*.
4. *Apache Bench* sebagai *benchmarking* tools *service*.
5. *Microservice* dan *Monolitik* sebagai arsitektur yang digunakan.

Aplikasi yang digunakan pada penelitian ini diberi nama "*Cattobuyz"* yang merupakan sebuah aplikasi belanja *online* yang dibuat khusus untuk pecinta hewan peliharaan. Di dalam aplikasi ini berisikan berbagai barang yang digunakan untuk kebutuhan sehari-hari hewan peliharaan terutama kucing. Di dalam aplikasi ini *user* bisa membeli sesuai barang yang tersedia sampai dengan *checkout* barang atau menaruh barang impian ke dalam *wishlist*. Aplikasi ini dikembangkan menggunakan 2 arsitektur web yang berbeda yaitu *Microservice* dan *Monolitik.*

3. Hasil dan Pemabahasan

3.1. Skenario Pengujian

Parameter yang menjadi pembanding dalam penelitian antara layanan *Microservice* dan *Monolitik* adalah dengan menggunakan 2 aspek yaitu Skalabilitas dan Availabilitas untuk performansi.

a) Skalabilitas
Untuk uji skalabilitas dengan melakukan *Load Testing* menggunakan *Request Generator* yaitu *Apache Bench* dimana *container / service* dari *Microservice dan Monolitik* akan diisi oleh *Apache Bench* dengan jumlah *request* yang sudah ditentukan. Parameter yang digunakan pada *Load Testing* dengan mengukur *CPU Utilization, Memori Usage, Response Time* dari *service* pada Layanan *Microservice* dan *Monolitik* dengan dengan memberikan masing-masing 50 *request*, 100 *request*, 500 *request* dan 1000 *request* sesuai dengan yang telah ditentukan pada *instance AWS*.

b) Availabilitas
Untuk uji selanjutnya dengan parameter *Container Creating Time* dari *Container*, ketika tidak bisa melayani *service* dan mengakibatkan *container* / *service* menjadi *terminate/stop*, pada aspek ini dibandingkan respon dari aplikasi berarsitektur *Microservice* dan *Monolitik* saat *service* dimatikan serta perbandingan waktu saat *service* diaktifkan kembali.

3.2. Evaluasi

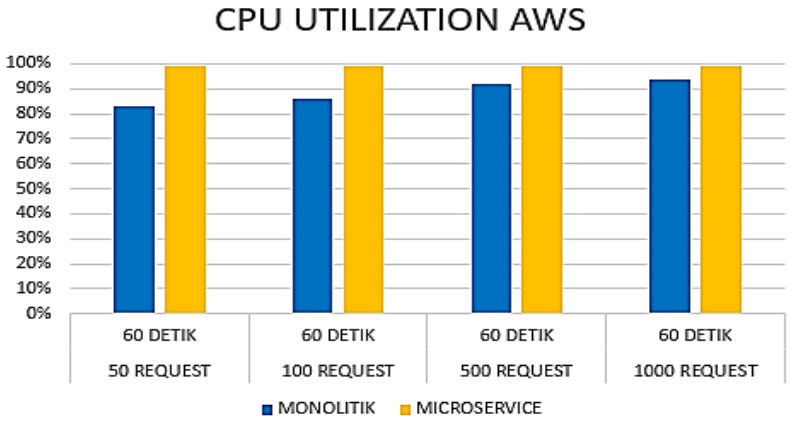
Bagian ini menampilkan hasil analisis, seperti yang telah dijelaskan sebelumnya parameter yang dipilih untuk dilakukan pengukuran adalah *Load Testing* untuk menentukan skalabilitas, waktu *Container Creating Time* untuk menentukan availabilitas dari kedua layanan *Microservice* dan *Monolitik.*

Performansi pengujian yang pertama dilakukan *Load Testing*, untuk melihat performansi skalabilitas dari *Microservice* dan *Monolitik*, disini melakukan tes pada sistem secara konstan diikuti dengan bertambahnya beban untuk melihat skalabilitas kedua arsitektur. Tujuan dari melakukan *load testing* ini adalah untuk mengetahui batas maksimal dari sebuah sistem. Sehingga bisa diketahui kualitas dari layanan yang dipakai, pada pengujian ini digunakan parameter *CPU Utilization, Memori Usage* dari *Microservice* dan *Monolitik.*

Pada pengujian selanjutnya untuk availabilitas dengan melakukan pengecekan dengan parameter *Container Creating Time*, yaitu saat *container* dari layanan mengalami *down* ataupun gangguan yang disebabkan oleh manusia maupun kesalahan dari sistem kita bisa mengetahui waktu yang diperlukan oleh sistem untuk mengembalikan layanannya kembali normal dan dilakukan cek pada aplikasi saat *service* mengalami *stop atau terminate*.

Skenario pengujian pertama pada aspek skalabilitas pada layanan *Microservice* dan *Monolitik*

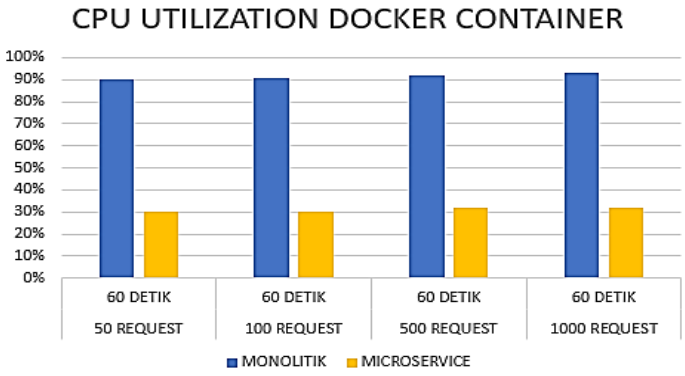
1) Skenario pengujian *load testing* dengan parameter *CPU Utilization AWS* pada Aplikasi *Cattobuyz* berarsitektur *Microservice* dan *Monolitik* dalam waktu 60 detik.



Gambar 4. Diagram CPU Utilization AWS dalam 60 detik

Pada skenario ini dilakukan pengujian dengan jumlah *request* 50, 100, 500 dan 1000 pada Aplikasi *Cattobuyz* yang berarsitektur *Microservice* dan *Monolitik* dalam waktu 60 detik. Tingkat *CPU Utilization Microservice* pada AWS cenderung di angka **99%.** Tingkat *CPU Utilization Monolitik* pada AWS berada di angka **83%** (50 *request*), **86%** (100 *request*), **92%** (500 *request*), **94%** (1000 *request*). Pada skenario ini layanan *Monolitik* masih unggul pada *CPU Utilization AWS*. Pada Gambar 4 layanan *Microservice* memakai *resource* CPU yang lebih banyak dikarenakan pada *Microservice* mengonsumsi sumber daya tiap *service* (4 *service*). Berbeda dengan *Monolitik* yang hanya ada 1 *service* beserta *databasenya*, layanan *Microservice* juga memiliki kerumitan dalam pengelolaan dan pengintegrasian tiap *service*.

2) Skenario pengujian *load testing* dengan parameter *CPU Utilization Docker Container* pada Aplikasi *Cattobuyz* berarsitektur *Microservice* dan *Monolitik* dalam waktu 60 detik.

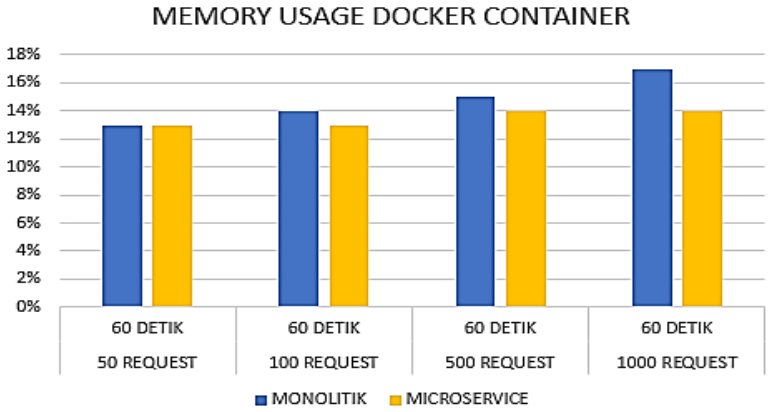


Gambar 5. Diagram CPU Utilization Docker Container dalam 60 detik

Pada skenario ini dilakukan pengujian dengan jumlah *request* 50, 100, 500 dan 1000 pada Aplikasi *Cattobuyz* yang berarsitektur *Microservice* dan *Monolitik* dalam waktu 60 detik. Tingkat *CPU Utilization Microservice* pada *Docker Container* berada di angka **30%** (50 dan 100 *request),* **32%** (500 dan 1000 *request).* Tingkat *CPU Utilization Monolitik* pada *Docker Container* berada di angka **90%** (50 *request*), **91%** (100 *request),* **92%** (500 *request),* **93%** (1000 *request).* Pada skenario ini layanan *Microservice* lebih

unggul dalam menggunakan *resource CPU Utilization Docker Container* untuk 1 service, tetapi jika keseluruhan service digabungkan maka layanan *Monolitik* masih unggul (*CPU Utilization AWS*). Pada Gambar pada aplikasi *Microservice* total *resource* akan dibagi sesuai dengan jumlah *service* (4 *service*) dalam aplikasi tersebut. Sedangkan pada *Monolitik* hanya memiliki 1 *service* beserta *databasenya*, maka *Monolitik* bisa memanfaatkan lebih banyak *resource*.

3) Skenario pengujian *load testing* dengan parameter *Memori Usage Docker Container* pada Aplikasi *Cattobuyz* berarsitektur *Microservice* dan *Monolitik* dalam waktu 60 detik.

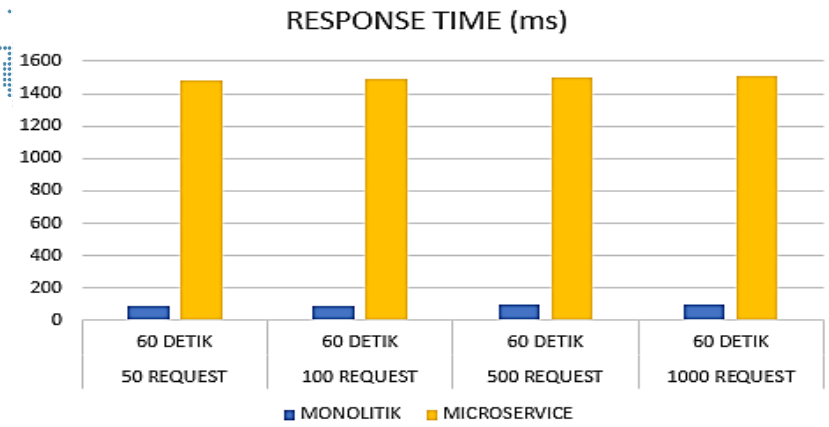


Gambar 6. Diagram Memory Usage Docker Container dalam 60 detik

Pada skenario ini dilakukan pengujian dengan jumlah *request* 50, 100, 500 dan 1000 pada Aplikasi *Cattobuyz* yang berarsitektur *Microservice* dan *Monolitik* dalam waktu 60 detik. Tingkat *Memory Usage Microservice* pada *Docker Container* berada di angka **13%** (50 - 100 *request)*, **14%** (500 - 1000 *request)*. Tingkat *Memory Usage Monolitik* pada *Docker Container* berada di angka **13%** (50 *request)*, **14%** (100 *request)*, **15%** (500 *request)*, **17%** (1000 *request)*. Pada skenario ini layanan *Microservice* lebih rendah dalam penggunaan *resource memory*. *Microservice* memakai *resource memory* lebih tinggi karena *Microservice* melakukan integrasi antar *service* yang rumit. Namun ketika jumlah *request* dinaikan, *Monolitik* yang akan memakan lebih banyak *memory* dikarenakan aplikasi akan melakukan integrasi data dalam satu kesatuan aplikasi (Gambar 6).

4) Skenario pengujian *load testing* dengan parameter *Response Time* pada Aplikasi *Cattobuyz* berarsitektur *Monolitik* dalam waktu 60 detik.

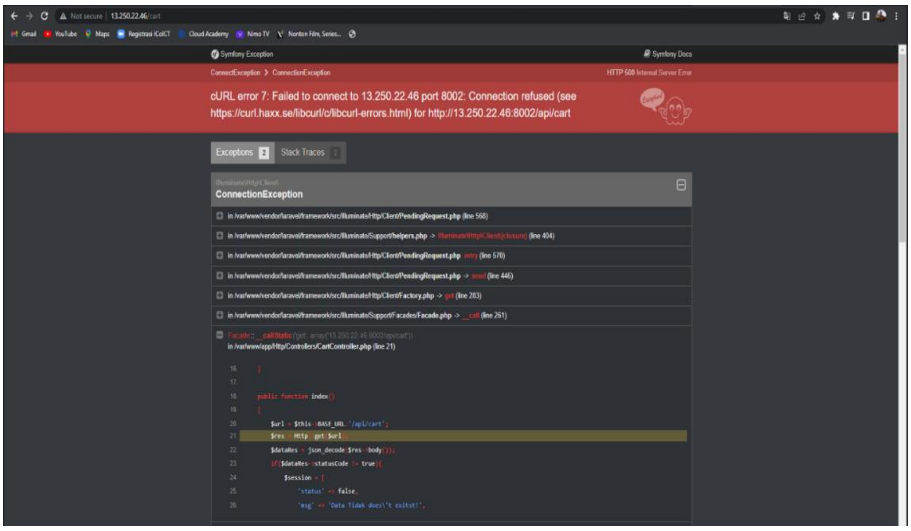
Pada skenario ini dilakukan pengujian dengan jumlah request 50, 100, 500 dan 1000 pada Aplikasi *Cattobuyz* yang berarsitektur *Microservice* dan *Monolitik* dalam waktu 60 detik. Tingkat *Response Time Microservice* berada di angka **1481,7 ms** (50 *request)*, **1490,8 ms** (100 *request)*, **1500,9 ms** (500 *request)*, **1506,7 ms** (1000 *request)*. Tingkat *Response Time Monolitik* berada di angka **89,3 ms** (50 *request)*, **89,4 ms** (100 *request)*, **94,7 ms** (500 *request)*, **96,3 ms** (1000 *request)*. Pada skenario ini layanan *Monolitik* memperoleh hasil yang lebih baik secara *response time* (Gambar 7). Hal ini terjadi karena tingginya *resource* yang dipakai sehingga terbatasnya sisa *resource* yang tersedia pada *Microservice* yang mengakibatkan waktu respon yang lebih lama.



Gambar 7. Diagram Response Time layanan dalam 60 detik

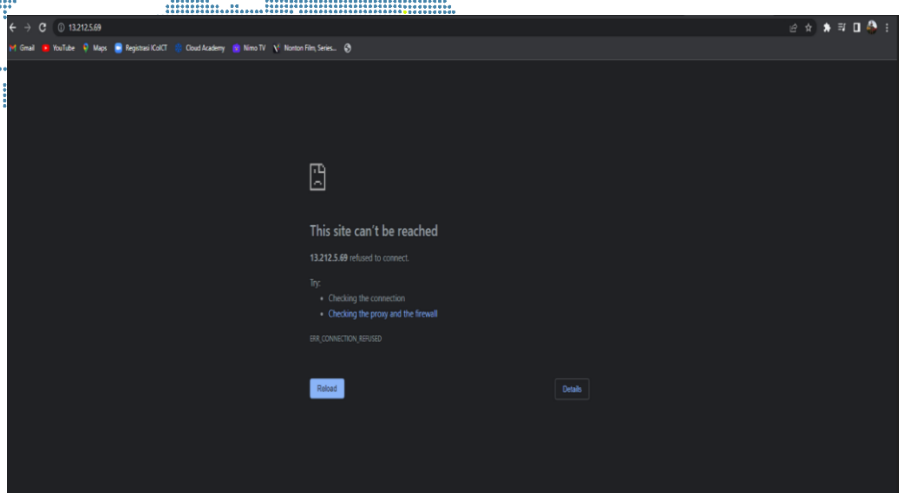
Skenario pengujian kedua pada aspek skalabilitas pada layanan *Microservice* dan *Monolitk*

5) Skenario pengujian *terminate atau stop service* kemudian dijalankan kembali dengan parameter *Container Creating Time* pada Aplikasi *Cattobuyz* berarsitektur *Microservice* dan *Monolitik.*



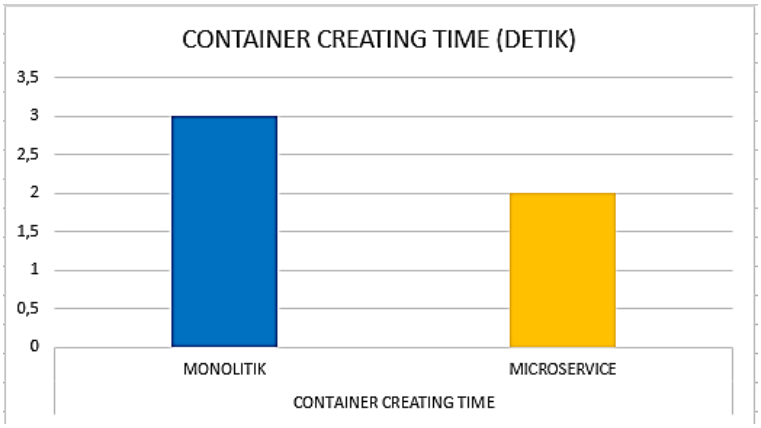
Gambar 8. Tampilan error service cart saat dimatikan

Skenario pengujian yang kedua untuk availabilitas yaitu dengan melakukan *terminate*/*stop* pada *service mono (port 80)* dan *service cart (port 8002)* kemudian coba kembali akses aplikasi. Pada Aplikasi *Cattobuyz* berarsitektur *Microservice service cart (port 8002)* dimatikan dan dicoba akses aplikasi. Secara keseluruhan aplikasi masih bisa diakses hanya saja *service cart* yang tidak bisa diakses (Gambar 8). Tetapi pada saat layanan *Monolitik* dimatikan semua layanan tidak dapat diakses oleh user (Gambar 9).



Gambar 9. Tampilan error service mono saat dimatikan

Setelah itu Aplikasi *Cattobuyz* dinyalakan kembali *servicenya* dan di dapatkan aplikasi dengan arsitektur *Microservice* lebih cepat dengan waktu 3 detik dibandingkan *Monolitik* dengan waktu 2 detik (Gambar).



Gambar 10. Grafik Creating Time

Hal ini dikarenakan pada *Microservice* hanya menjalankan salah satu *servicenya* saja, sedangkan pada *Monolitik* dengan menjalankan 1 kesatuan aplikasi.

4. Kesimpulan

Hasil analisis pada pengujian yang telah dilakukan pada penelitian ini dapat diperoleh kesimpulan sebagai berikut. Berdasarkan uji skalabilitas yang telah dilakukan pada kedua arsitektur, *Monolitik* unggul dalam pengujian ini karena menggunakan lebih sedikit *resource* yang tersedia serta proses integrasi antar *service* yang rumit membuat *Microservice* tertinggal dalam pengujian ini. Sedangkan pada uji availabilitas, *Microservice* memperoleh hasil yang lebih baik saat salah satu *service* dimatikan maka layanan tetap masih bisa diakses, berbeda pada *Monolitik* yang tidak bisa diakses. Waktu yang diperlukan untuk menjalankan *service* pada *Microservice* juga lebih cepat karena hanya perlu menjalankan salah satu *service* saja, berbeda dengan *Monolitik* yang harus menjalankan satu kesatuan aplikasi. Secara keseluruhan dapat disimpulkan bahwa aplikasi dengan arsitektur *Microservice* bukanlah yang paling cocok untuk diterapkan namun harus disesuaikan dengan kebutuhan pengguna, sebagai contoh untuk penggunaan yang sederhana dan sedikit *user* (kurang dari 100) yang mengakses maka lebih baik menggunakan *Monolitik*, karena akan lebih sederhana dan tidak memakan banyak

resource CPU Utilization dan Memory Usage. Namun jika penggunaan untuk kebutuhan besar memang lebih baik menggunakan aplikasi dengan arsitektur *Microservice* karena lebih baik untuk pengembangannya ke depannya dan jika ingin menambah fitur atau *update* pun lebih ringan, karena secara keseluruhan aplikasi tidak begitu berpengatuh besar. Pada saat terjadi kegagalan sistem aplikasi dengan arsitektur *Microservice* juga tidak begitu berpengaruh besar, akan tetapi diperlukan *resource* yang besar juga untuk mengembangkan aplikasi dengan arsitektur *Microservice* ini.

Daftar Pustaka

[1] "Mengenal Cloud Computing: Pengertian, Tipe, dan Fungsinya," *https://indonesiancloud.com/mengenal-cloud-computing/*, Jun. 22, 2022.

[2] V. Singh and S. K Peddoju, "Container-based Microservice Architecture for Cloud Applications," *Indian Institute of Technology Roorkee*, 2017.

[3] N. Dinh Nguyen and T. Kim, "Toward Highly Scalable Load Balancing in Kubernetes Clusters," *Chungbuk National University*, 2020.

[4] J. ' Shah and D. ' Dubaria, "Building modern clouds: Using docker, kubernetes google cloud platform," *IEEE 9th Annu. Comput. Commun. Work. Conf. CCWC 2019*, pp. 184–189, 2019.

[5] L. ' A. Vaygan, M. ' A. Saied, M. ' Toeroe, and F. ' Khendek, "Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes," *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS) 2019*, pp. 176–185.

[6] G. Blinowski, A. Ojdowska, and A. Przybyłek, "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation," *Institute of Computer Science, Warsaw University of Technology*, 2022.

[7] L. Abdollahi Vayghan, M. Aymen Saied, M. Toeroe, and F. Khendek, "Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned," *Engineering and Computer Science Concordia University Montreal, Canada*, 2018.

[8] M. Kalske, N. Makitalo, and T. Mikkonen, "Challenges When Moving from Monolith to Microservice Architecture," *Department of Computer Science, University of Helsinki, Helsinki, Finland*, 2018.

[9] F. Tapia, M. Ángel Mora, W. Fuertes, H. Aules, E. Flores, and T. Toulkeridis, "From Monolithic Systems to Microservices: A Comparative Study of Performance," *Department of Computer Sciences, Universidad de las Fuerzas Armadas ESPE, Av. General Rumiñahui S/N, P.O. Box 17-15-231B, Sangolquí 171103, Ecuador*, 2020.

[10] O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," *Department of Electronics Technology Budapest University of Technology and Economics Budapest, Hungary*, 2018.