# Implementasi Container Kubernetes untuk Mendukung Scalability

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

## ABSTRAK

Perkembangan teknologi yang semakin maju menyebabkan banyaknya tuntutan pada *client- server* untuk menjalaskan tugasnya. Kemampuan *server* yang ada akan semakin menurun dengan banyaknya akses yang datang dari *client*. Kemampuan *server* yang melebihi batas akan mengalami *overload* sehingga banyak dampak buruk yang akan ditimbulkan antara lain penurunan kecepatan waktu mengakses bahkan menimbulkan *server down*.

Penelitian ini mencoba mengusulkan solusi dengan memberikan tambahan jalur akses pada *server* yaitu menggunakan teknologi *containerization* pada Kubernetes. Dimana *container* merupakan jalur akses bagi *client* ke *server*. Teknologi *containerazitation* ini diimplementasikan dengan memanfaatkan *scalability. Scalability* pada *container* akan menyesuaikan dengan kebutuhan *client* maupun *server*.

Dengan teknologi *container*, akses terhadap *server* dapat dilakukan dengan penambahan *container* untuk jalur akses jika dirasa penuh dan akan mengurangi *container* jika jumlah *client* yang mengakses berkurang. Hal ini akan meminimalisir *server* yang *overload* sehingga *client* dapat melakukan akses terhadap *server* tanpa kesulitan akibat sedikitnya jalur akses yang ada.

**Kata Kunci:** Kubernetes, Container, Scalability

## *ABSTRACT*

*Increasingly advanced technological developments have caused many demands on the client-server to carry out their duties. Existing server capabilities of access that comes from the client. The ability of the server to exceed the limit will experience overload so that many of the adverse effects that will be caused include a decrease in the speed of access even causing a server down.*

*This study tries to propose a solution by providing an additional access point on the server that is using containerization technology on Kubernetes. Where the container is the access point for the client to the server. This containerazitation technology is implemented by utilizing scalability. Scalability in the container will adjust to the needs of the client and server.*

*With container technology, access to the server can be done by adding a container for the access point if it is felt full and will resuce the container if the number of accessing clinets decreases. This will minimize the server overload so that the client can access the server without difficulty due to the lack of access points.*

***Keywords:*** *Kubernetes, container, scalability*

## 1. PENDAHULUAN

Perkembangan teknologi memberikan pengaruh bagi kehidupan salah satunya adalah layanan komputasi. Model komputasi modern yang sering dipakai sekarang ini adalah *client server*. *Client server* saling berinteraksi dan mengakses satu sama lain. Masing-masing *server* mempunyai kemampuan penyimpanan yang berbeda-beda tergantung dari kapasitas yang ada. Pada kenyataannya kemampuan *server* juga mempunyai batasan tertentu atau limitasi. *Server* yang melebihi batas penyimpanan atau pengaksesan akan mengalami *overload*. Dimana dampak yang ditimbulkan yaitu *memory* daya tampung yang penuh, penurunan kecepatan pada waktu mengakses atau bahkan menimbulkan *server down* ketika sedang diakses oleh banyak user atau *client* dalam waktu yang bersamaan..

Ketika *server* sudah mengalami *overload* banyak akibat yang terjadi antara lain penurunan kecepatan saat user mengakses, *user* dapat ter-*reject* dan ditolak untuk mengakses *server*, atau bahkan *server* menjadi *crash* atau *down*. Dimana untuk mengatasi hal tersebut terdapat alternative untuk mengatasi hal tersebut yaitu dengan membatasi jumlah *client* yang mengakses, memperbesar kapasitas *server* dengan menambahkan memory, atau bahkan membuka *server* baru. Namun solusi yang paling dapat diimplementasikan sesuai kebutuhan diantara alternatif tersebut adalah proses *scalability*. Kemampuan *server* untuk mengotomatisasi dalam penambahan atau pengurangan *server* sangat diperlukan. Jika jumlah *user* yang mengakses semakin banyak maka *server* akan otomatis bertambah, sedangkan jika jumlah *user* sudah mulai berkurang maka *server* akan otomatis berkurang.

Salah satu teknologi yang sedang *trend* saat ini adalah Kubernetes. Kubernetes merupakan *system opensource* yang berfungsi untuk mengotomatisasi penyebaran maupun penskalaan *container* [8]. Dengan adanya teknologi tersebut dapat membantu mengatasi adanya masalah *distributed server*. Dimana semua *server*, *aplikasi*, *database* akan ditampung dalam sebuah *container* yang terdapat dalam Kubernetes. Kubernetes akan memanage semua *container* yang ada sesuai dengan kebutuhan.

Dengan menggabungkan proses *scalability* pada teknologi Kubernetes, maka proses otomatisasi *container* dapat diimplementasikan sesuai dengan jumlah user yang mengakses. *Container* akan melakukan penambahan secara otomatis jika user melebihi batas limitasi tertentu. Proses *scalability container* ini dapat diterapkan pada Kubernetes dengan beberapa parameter yang mendukung. Diharapkan dengan penerapan *scalability* akan

meningkatkan performa kinerja serta waktu *response server* terhadap user tanpa mengurangi kemampuan *utility server*.

Penelitian ini berfokus melakukan penerapan *scalability* pada Kubernetes, dimana akan menjawab pengaruh implementasi *scalability* terhadap performance atau kinerja *container* Kubernetes, perbandingan *response time* serta *concurrent user* dari implementasi *scalability*.

Dimana fokus pada *scalability* terletak pada *container*. Serta aplikasi untuk pengujian yang dibuat yaitu implementasi PRS online yang akan dipecah menjadi beberapa *microservice* sesuai dengan fungsinya masing-masing.

# 2. DASAR TEORI

## 2.1 Virtualisasi

Virtualisasi merupakan representasi logis dari computer dalam perangkat lunak. Dengan memisahkan *hardware* dari sistem operasi, virtualisasi menyediakan fleksibilitas operasional yang lebih meningkatkan tingkat pemanfaatan perangkat keras fisik yang mendasarinya [7]. Virtualisasi terbagi menjadi virtualisasi perangkat keras dan virtualisasi data. Dengan adanya virtualisasi akan memudahkan pengguna untuk dapat menjalankan sistem operasi yang berbeda dari semua mesin virtual pada satu mesin fisik. Ketika terjadi kesalahan maupun kekeliruan dalam penginstalan program, tidak akan mempengaruhi mesin virtual yang lainnya. Pengguna dapat dengan bebas melakukan *eksperimen* tanpa perlu khawatir dengan hilangnya maupun rusaknya program serta data bawaan yang ada.

## 2.2 *Container*

*Container* adalah adalah unit software yang mengemas semua *code* dengan dependensinya sehingga aplikasi dapat berjalan dengan cepat dari satu lingkungan komputasi ke lingkungan komputasi yang lain [4]. Dengan kata lain *container* dapat diasumsikan sebagai sebuah wadah yang dapat menampung banyak data maupun aplikasi dengan tujuan menjadi lebih ringkas untuk dijalankan pada sistem..

Perbandingan *infrastructure* VM dan *container* adalah dimana *container* hanya mengisolasi *library* serta aplikasi yang digunakan tanpa mengisolasi keseluruhan perangkat keras, kernel, dan OS. Berbeda dengan VM yang mengisolasi keseluruhan system. Secara umum *container* terbagi menjadi dua jenis yaitu :

1. *Container* berbasis sistem operasi, yaitu *container* yang mengisolasi level sistem operasi. Teknologi ini menawarkan fitur yang mirip dengan virtualisasi namun dengan peningkatan *performa* yang cukup signifikan. Sifatnya yang memiliki lingkungan terisolasi antara satu dengan yang lainnya serta *high performance* memberikan keunggulan terhadap *container*.
2. *Container* berbasis aplikasi, yaitu *container* yang mengisolasi level aplikasi. Dimana memudahkan para pengguna untuk membuat dan memaksimalkan suatu aplikasi yang dikelola. Selain memanfaatkan *hardware* sang induk, juga dapat memanfaatkan *service-service* lain dari *container-container* yang saling berhubungan dan berjalan pada sistem [3].

## 2.3 *Scalability Container*

*Scalability container* adalah dimana aplikasi *container* dapat menangani peningkatan beban jumlah user [9]. *Scalability* dapat dilakukan dengan konfigurasi arsitektur yang ada, maupun menggunakan tambahan *container*, sehingga dapat membuat jalan akses yang baru dari user terhadap aplikasi tanpa memakan waktu yang lama. *Scalability* mebuat *container* menjadi elastis, dalam arti menyesuaikan beban jumlah user yang ada. Saat user bertambah, *container* akan mengikuti perubahan jumlah user dengan membuat jalan akses yang baru, dan pada saat user berkurang, *container* mengurangi jalan akses yang ada. Selain memudahkan user yang mengakses, *scalability* juga menghemat waktu bahkan biaya dalam prosesnya. Tidak ada *resource* yang terbuang karena penyesuaian (*scalability*) yang *real time*.

## 2.4 *Microservice*

*Microservice* yaitu membagi *service monolithic* yang ada ke bagian yang lebih kecil dimana *service-service* tersebut saling independen dan mempunyai fungsi yang berbeda-beda [1].

Beberapa faktor penting dari *mircoservice architecture* adalah :

1. *Language agnostic API* (*API* tidak bergantung pada pemilihan bahasa program)
2. *Small building blocks* (terdiri dari blok-blok yang kecil)
3. *Highly decoupled* (terpisah dan sangat independen)
4. *Focused on doing small task* (berfokus untuk menyelesaikan tugas-tugas yang ringan)
5. *Modular approach* (cara kerja yang stadar)
6. *Continously deployed systems* (cocok untuk sistem yang dinamis dan berkembang)

## 2.5 Kubernetes

Kubernetes adalah sistem *open source* untuk mengotomatisasi penyebaran, penskalaan, dan pengelolaan *container*. Pada awalnya dirancang oleh *Google*, kemudian sekarang dikelola oleh *Cloud Native Computing Foundation*. Kubernetes sendiri berfungsi sebagai pengelola *container-container* serta menyediakan *Platform* untuk mendukung fungsinya. Design Kubernetes terdiri dari Pods, Labels and Selectors, Controllers, Services [8]. Objek dasar yang dimiliki oleh Kubernetes antara lain :

1. *Pod* : unit terkecil dan paling sederhana dalam model objek Kubernetes. Pod mewakili unit penyebaran dimana satu *instance* aplikasi di Kubernetes dapat terdiri dari satu container atau sejumlah container yang berbagi sumber daya.
2. *Service* : Abstraksi yang mendefinisikan serangkaian Pods sejenis, yang menjadi jalan masuk bagi *request* oleh pod-pod tersebut. *Service* mengawasi dan mencatat adanya perubahan *state* dari *pod-pod* yang merupakan bagiannya
3. *Volume* : Bagian dari penyimpanan di *cluster* yang telah disediakan. Data yang disimpan pada *Volume* tidak akan hilang ketika dilakukan *destroy* terhadap *pod.*
4. Namespace : Cara untuk membagi sumber daya *cluster* diantara beberapa pengguna (berdasar kuota sumber daya).
5. *Node* : Mesin pekerja di Kubernetes yang dapat berupa mesin virtual atau fisik, tergantung pada cluster. Node dapat memiliki banyak *pod* dan master

## 2.6 Kubernetes Architecture

Kubernetes terbagi menjadi *master node* dan *worker node* yang memiliki fungsi yang berbeda. *Master node* pada Kubernetes berfungsi sebgai *master* yang mengontrol keseluruhan *unit* dan *cluster*, megatur *workload* serta komunikasi antar sistem. Sedangkan *worker node* atau yang dikenal sebagai pekerja atau minion, merupakan mesin tempat *container* (beban kerja) digunakan.

Master node pada Kubernetes berfungsi sebagai *master* yang mengontrol keseluruhan *unit* dari *cluster*, mengatur *workload* serta komunikasi antar sistem. Berikut adalah komponen pada *master node* :

1. *Etcd* : Digunakan untuk menyimpan data *cluster*, serta memberikan notifikasi kepada semua *cluster* ketika mengalami perubahan konfigurasi. Etcd hanya dapat diakses lewat API *server* demi keamanan.
2. API *server* : Komponen utama dan melayani kubernetes API menggunakan *JSON* melalui *HTTP* yang menyediakan *interface internal* maupun *eksternal* pada kubernetes.
3. *Scheduler* : Komponen yang mengatur penjadwalan dari *node-node* dan pod yang tidak terjadwal. *Scheduler* melacak berdasar sumber daya pada setiap *node* untuk memastikan bahwa beban kerja tidak melebihi sumber daya yang tersedia. Pada dasarnya *scheduler* melakukan pengecekkan antara sumber daya yang tersedia dengan permintaan.
4. *Controller manager* : *Loop* yang mendorong status *cluster* actual menuju status *cluster* yang diinginkan.

*Worker node* atau yang dikenal sebagai pekerja atau minion, merupakan mesin tempat *container* (beban kerja) digunakan. Beberapa komponen pada *worker node* antara lain :

1. *Kubelet* : Berfungsi untuk melaporkan kondisi *node* dan *pod* kepada *master node* jika mengalami perubahan. Kubelet mengambil konfigurasi dari API *server* untuk memastikan *pod* berjalan yang tersimpan pada *master node*.
2. *Kube – proxy* : Merupakan implementasi dari *network proxy* dan *load balancer*, dan mendukung abstraksi layanan bersama dengan operasi jaringan lainnya. Bertugas untuk mengarahkan jalan *container* yang sesuai berdasar *IP address* dan *port number* dari *request* yang masuk.
3. *Container runtime* : Tingkat terendah dari layanan mikro yang menampung aplikasi yang sedang berjalan, *libraries*, dan dependensinya. Salah satu contoh yang didukung oleh Kubernetes adalah Docker.

*Addons* ialah komponen pendukung selain komponen utama yang disediakan untuk mendukung Kubernetes *cluster*. Beberapa *Addons* yang umum digunakan ialah :

1. *Kubectl :* Untuk mengirimkan perintah ke *master node* Kubernetes.
2. *Kubeadm :* Untuk mempermudah proses pembuatan Kubernetes *cluster*.
3. *Container Resource Monitoring :* Menyediakan *runtime* aplikasi yang andal dan dapat meningkatkan atau menurunkan beban kerja, dengan arti mampu mengefektifkan dan memantau *performance workload.*
4. *Metric-server :* Berfungsi untuk mengumpulkan data pemakaian sumber daya dari tiap *node*.
5. *Cluster – level logging :* Kemampuan untuk memiliki penyimpanan dan *life-cycle* yang terpisah dari *node*, *pod*, atau *container* dengan tujuan untuk menghindari hilangnya data akibat kegagalan *node* atau *pod*.

## 2.7 *Horizontal Pod Autoscaler*

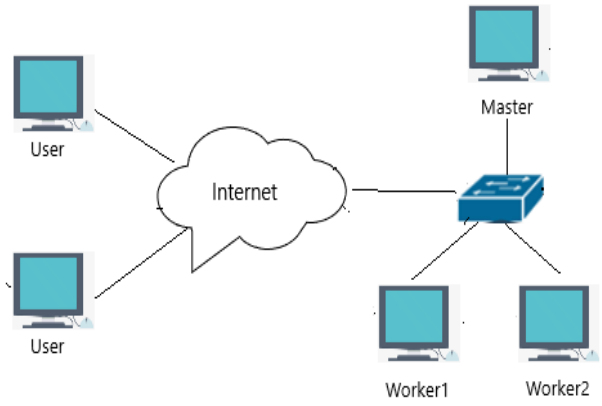
*Horizontal pod autoscaler (HPA)* akan menskala *container* dengan sumber daya yang tersedia. Penskalaan jumlah *pod* berdasar pemanfaatan *CPU* atau memori yang diamati [8]. Dalam kata lain *horizontal pod autoscaler* akan menambahkan wadah atau yang dikenal dengan sebutan *container* jika sudah melebihi batas maksimum. Untuk melakukan konfigurasi pada *HPA* dapat membuat skala *pods* berdasar *varied, external, and custom metrics. Horizontal pod autoscaler* lebih cocok digunakan ketika dibutuhkan kapasitas untuk *scalability* yang cukup banyak. Dengan mereplika *container-container* yang ada.

# 3. DESAIN SISTEM

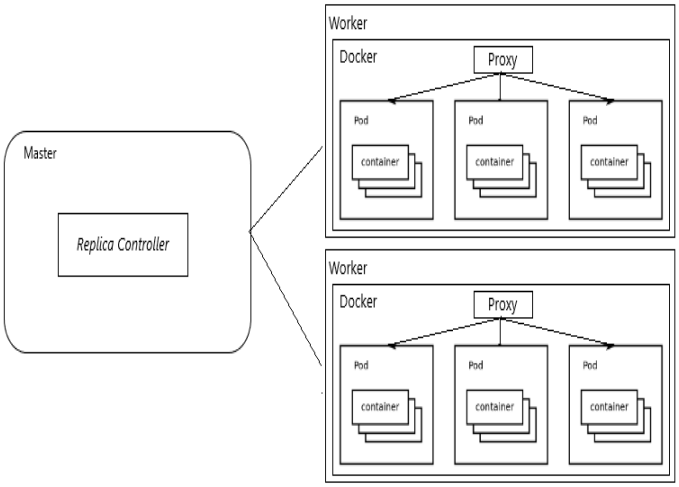
## 3.1 Desain Arsitektur Sistem

Dari permasalahan yang terjadi, maka diberikan solusi untuk mengatasi masalah banyaknya *user* yang mengakses secara bersamaan. Dengan mengimplementasikan aplikasi kedalam Kubernetes *container*, akan memberikan jalan alternatif kepada *user* yang mengakses. Untuk Kubernetes yang akan digunakan terdiri dari 3 komputer, dimana 1 buah komputer digunakan sebagai *master node*, 2 komputer digunakan sebagai *worker node*.

Sedangkan untuk simulasi *multiple user* yang akan dilakukan menggunakan aplikasi untuk menggenerate *multiple user* dan akan mengakses *url* dari *microservice* itu sendiri. Pada *master* dilakukan pengaturan *microservice* yang terdapat di *worker*, dimana pada setiap *worker* nantinya akan terdapat beberapa *container*. Berikut pada Gambar 1 dijelaskan mengenai desain arsitektur sistem yang akan dibuat. Sedangkan mengenai desain *container* pada *worker* Kubernetes akan dijelaskan pada Gambar 2.




**Gambar 1. Desain arsitektur sistem**



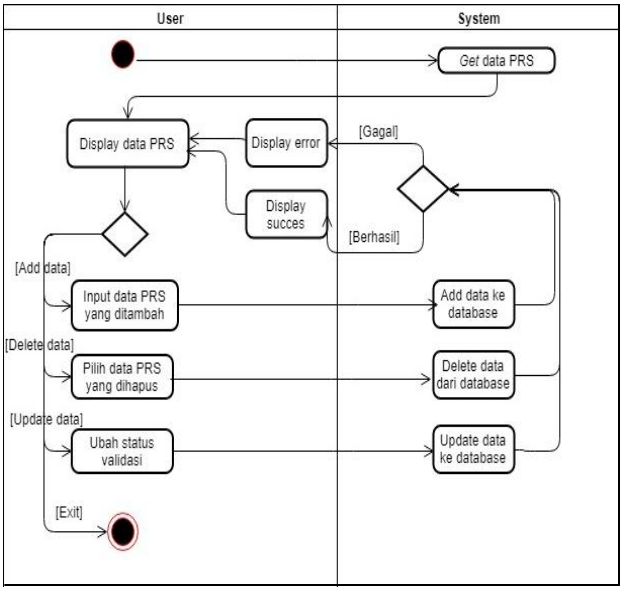

**Gambar 2. Desain arsitektur sistem**

## 3.2 Desain Alur Aplikasi

Langkah pertama yang dilakukan yaitu mengatur *setting* pada Kubernetes termasuk melakukan instalasi pada *master* dan *worker* dimana pada *worker* tersebut akan dibuat *container-container* yang akan dipakai sesuai dengan kebutuhan. Langkah kedua yang dilakukan yaitu dengan melakukan *create and deployment* sistem *microservice* yang sudah dibuat. Kemudian langkah selanjutnya yaitu melakukan *setting up scalabilitiy* untuk pengaturan *container* yang akan diuji, dengan memperhatikan replikasi *container* yang akan bertambah maupun berkurang sesuai dengan kebutuhan.

Pada proses *setting up* Kubernetes akan dilakukan instalasi *ansible*, *master* dan *worker* untuk Kubernetes. Dimana *ansible* berfungsi untuk melakukan pengelolaan *server* yang lebih mudah dan terstruktur, serta dapat menghemat waktu yang ada selagi melakukan instalasi untuk *worker-worker* yang ada. *Master* pada Kubernetes berfungsi untuk mengatur semua *container* yang ada. Dimana *worker* berfungsi sebagai pekerja dan tempat dimana *container* bekerja.

Untuk proses *create and deployment microservice* dimana proses pembuatan *microservice* akan dilakukan terlebih dahulu sebelum proses implementasi. Berikut pada Gambar 3 berisi gambar mengenai rancangan proses keseluruhan dari *webservice* yang akan dibuat, dimana nantinya akan dibagi menjadi beberapa *microservice* berdasarkan fungsi dari masing-masing *service*.




**Gambar 3. *Activity Diagram Microservies* secara keseluruhan**

# 4. PENGUJIAN SISTEM

Pengujian dilakukan dengan menguji aplikasi PRS online yang sudah diimplementasikan pada Kubernetes. Akan dicatat perbandingan *concurrent user* dan *response time* dari *multiple server (container)* terhadap *single server.*

## 4.1 *Front-end* Pengujian *Microservice*

Pada bagian ini akan dijelaskan *microservice* yang telah dibuat. Terdapat 5 *microservice* dari pedoman PRS online yaitu 2 *microservice* untuk *get*, satu untuk mendapatkan jadwal kelas, yang satu untuk mendapatkan jadwal uts dan uas, kemudian 1 *microservice* untuk *post* yaitu untuk melakukan input data pilihan mata kuliah pada database, 1 *microservice* untuk *put* yaitu untuk melakukan *update* data yang telah tervalidasi, dimana status validasi akan berubah ketika dilakukan proses *update*, 1 *microservice* untuk *delete* yaitu untuk menghapus data mata kuliah yang telah dipilih oleh mahasiswa jika merasa terjadi kesalahan dalam memilih, proses *delete* tidak berlaku jika mahasiswa telah melakukan validasi.

## 4.2 Proses Pengujian

Proses pengujian menggunakan *tools apache jmeter* untuk melakukan simulasi dari *user* yang akan melakukan PRS *online*. Dimana kita dapat menggenerate *multiple user* untuk melakukan akses terhadap *microservice* yang sudah ditentukan.

## 4.3 Hasil Pengujian

**Pengujian *microservice* yang dilakukan dibagi menjadi pengujian tanpa *script* dan pengujian dengan menggunakan *script*. Untuk pengujian tanpa menggunakan *script* akan dilakukan akses terhadap *microservice* Berikut pada Tabel 1 merupakan hasil *cpu usage pod* dari pengujian dengan *single server*. Sedangkan pada**

Tabel 2 merupakan hasil dengan *multiple server* dimana sudah diterapkan fungsi *scalability*.

**Tabel 1. Hasil pengujian dengan *single server* pada *worker***

| ***Scenario*** | ***Microservice*** | **Cpu *usage pod* (*milicore*)** |
|---|---|---|
| Rata-rata | *Get* Kelas | **625.33** |
| | *Get* Uts dan Uas | **595.33** |
| | *Post* | **567.66** |
| | *Delete* | **0** |
| | *Put* | **0** |

**Tabel 2. Hasil pengujian dengan *multiple server***

| ***Scenario*** | ***Microservice*** | **Cpu *usage pod* (*milicore*)** |
|---|---|---|
| Rata-rata | *Get* Kelas | **236.66** |
| | *Get* Uts dan Uas | **233.66** |
| | *Post* | **264** |
| | *Delete* | **0** |
| | *Put* | **0** |

Berdasarkan perbedaan tabel diatas terlihat perbedaan pada hasil *cpu* tiap *container* yang digunakan, dimana *resource cpu* tiap *container* telah terbagi berdasarkan *scale* yang telah dibuat. Yaitu tersebar di antara *worker* 1 dan *worker* 2 dimana tiap *worker* memiliki masing-masing *container* nya sendiri.

Kemudian berikutnya untuk pengujian dengan *script* hasil dapat dilihat pada Tabel 3 dan Tabel 4.

**Tabel 3. Hasil pengujian *script* dengan *single server***

| ***Scenario*** | ***Microservice*** | **Cpu *usage pod* (*milicore*)** |
|---|---|---|
| 1 | *Script* | 576 |
| 2 | *Script* | 526 |
| 3 | *Script* | 498 |
| Rata - rata | | **533.33** |

**Tabel 4. Hasil pengujian *script* dengan *multiple server***

| *Scenario* | *Microservice* | Cpu *usage pod* (*milicore*) |
|---|---|---|
| 1 | *Script* | 209 |
| 2 | *Script* | 369 |
| 3 | *Script* | 333 |
| Rata - rata | | **303.66** |

Sedangkan untuk perbandingan *concurrent user* pada *single server* yaitu sebesar 1.998 pada satu *container*. Jika *maximal* pod yang ditentukan sebesar 10 *pod*, maka untuk *maximal concurrent user* saat terjadi *scalability* yaitu 10 kali lipat dibandingkan dengan *single server*. Untuk perbandingan *response time* lebih memakan waktu untuk pengujian dengan *multiple server* dikarenakan untuk proses pembuatan *container* memakan waktu yang dapat menyebabkan pengaruh terhadap *response time*.

# 5. KESIMPULAN DAN SARAN

## 5.1 Kesimpulan

Berdasarkan hasil pengujian yang dilakukan pada sistem, maka dapat disimpulkan bahwa :

- Dengan implementasi *scalability* terhadap *container*, adanya penghematan *cpu usage pod* pada saat *scalability* diterapkan. Dikarenakan adanya pembagian *container* yang tersebar di tiap *worker* nya.
- Perbandingan *concurent user* antara *single server* dan *multiple server* dengan pengujian 2000 *user* yaitu 1.998 : 1.998 dimana jumlah maksimal *concurent user* berlaku 1:10 untuk jumlah *maximum container* 10.
- *Perbandingan response time* antara *single server* dan *multiple server* memakan waktu yang lebih lama untuk *multiple server* dikarenakan adanya *delay* waktu dari pembuatan *container* yang di *scalability*.

## 5.2 Saran

Berdasarkan hasil pengujian yang dilakukan pada sistem, maka dapat diberikan saran yaitu:

- Penerapan *scalability* bisa diterapkan juga pada database yang ada
- Pengujian dapat lebih dioptimalkan dengan *cpu* dan *memory* dari *client* yang sesuai dengan spesifikasi *server* yang ada sehingga tidak terjadi *delay* ketika proses pengujian berlangsung.

# 6. DAFTAR PUSTAKA


[1] Baskarada, S & Koronios, A. 2018. Architecting Microservices: Practical Opportunities and Challenges. *Journal of Computer Information Systems*. 1-9. 10.1080/08874417.2018.1520056.

[2] Casalicchiom, E & Perciballi, V. 2017. Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics. Paper presented at 2017 IEEE 2nd International Workshops on Foundations and Applications of Self Systems (FASW).

[3] Cloudmatika. 2019. Archives: container. URI= https://www.cloudmatika.com/

[4] Docker. 2018. What is Container. URI=https://www.docker.com/resources/what-container.

[5] George Sammons. 2017. Basics of Kubernetes. URI=https://www.kubernetes.io/

[6] Hoenisch, P., Weber, I., Schulte, S., Zhu, L., & Fekete, A. 2015. Four-Fold Auto-Scaling on a Contemporary Deployment Platform Using Docker Containers. *Service-Oriented Computing Lecture Notes in Computer Science*,316-323. doi:10.1007/978-3-662-48616-0_20

[7] IBM. 2007. Virtualization in Education. The Greaves Group, 4, 3-4.

[8] Kubernetes. 2018. URI= https://kubernetes.io/

[9] Rice, L. & Hausenblas, M. 2018. Kubernetes Security. How to Built and Operate Applications Securely on Kubernetes.