BAB II

TINJAUAN PUSTAKA

2.1 Landasan Teori

2.1.1 Microservice

Microservice atau layanan mikro dapat diartikan sebagai kumpulan layanan atau proses independen yang berukuran kecil yang biasa berkomunikasi untuk membentuk aplikasi kompleks (Putra 2018). Pengembangan yang dilakukan oleh tim pengembang pada suatu layanan *microservice* tidak akan berpengaruh pada layanan yang lain karena sifatnya yang independen. Berbeda dengan arsitektur *monolithic, microservice* mendorong penerapan independen dan dapat dikembangkan menggunakan tumpukan teknologi yang berbeda (Baresi, Garriga, and De Renzis 2017). Setiap *microservice* dibangun di sekitar kemampuan bisnis, berjalan dalam prosesnya sendiri, dan berkomunikasi dengan *microservice* yang lain dalam aplikasi melalui mekanisme ringan. Gaya arsitektur *microservice* dapat dilihat sebagai kepanjangan alami dari *service oriented architecture* (SOA), yang menekankan pada manajemen mandiri atau layanan mandiri, dan bersifat ringan (Soldani, Tamburri, and Van Den Heuvel 2018).

2.1.2 Container

Container dapat dinyatakan sebagai sistem operasi ringan yang dapat bekerja secara langsung di dalam *host* sistem operasi (Fihri, Negara, and Sanjoyo 2019). *Containerization* membungkus kode aplikasi bersama dengan *file*

konfigurasi terkait seperti *library* dan semua *dependency* yang dibutuhkan untuk menjalankan aplikasi. *Container* diabstraksikan dari *host* sistem operasi, dengan demikian, menjadi *portable* dan mandiri yang dapat berjalan di berbagai *platform* (Dewi et al. 2019). Banyak penyedia layanan yang telah mengadopsi dengan sejumlah alasan di antaranya: (1) untuk mengurangi kerumitan saat menggunakan *microservice*; (2) untuk menskalakan, menghapus, dan *deploy* bagian dari sistem atau aplikasi dengan mudah; (5) untuk meningkatkan fleksibilitas dengan menggunakan *framework* dan *tool* yang berbeda; (4) untuk meningkatkan skalabilitas keseluruhan; dan (5) meningkatkan ketahanan sistem (Khazaei et al. 2016).

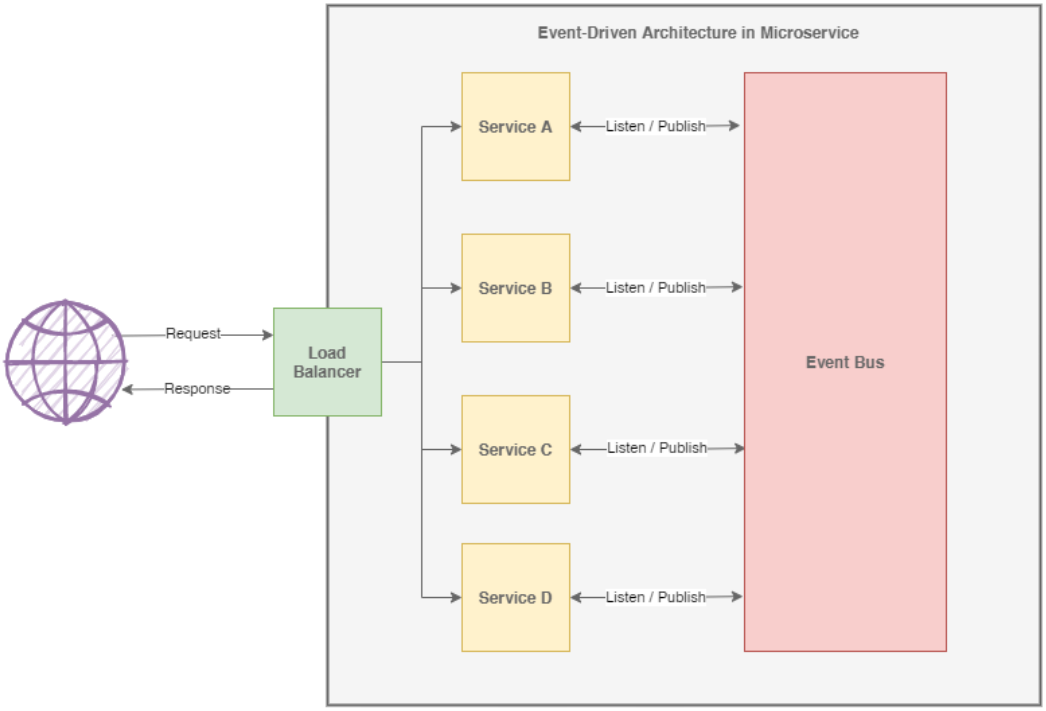
Salah satu pendekatan *application-oriented container* yang populer di *container* adalah Docker. Docker bergantung pada fitur *kernel* Linux, seperti *namespace* dan *control groups*. *Container* Docker mengenkapsulasi aplikasi beserta depedensi perangkat lunaknya, dan aplikasi yang telah dienkapsulasi tersebut dapat dijalankan pada mesin Linux yang berbeda dengan mesin Docker (Plauth, Feinbube, and Polze 2017). Dalam pembuatan *container* diperlukan sebuah *Docker Image* yang didalamnya data-data *library, command,* dan kebutuhan aplikasi lainnya. *Docker Image* sendiri dapat dibuat melalui *file* konfigurasi yang disebut dengan *Dockerfile*.

Pendekatan lain dapat dilakukan secara efektif menggunakan Kubernetes yang merupakan *platform open-source* untuk mengelola aplikasi dalam *container* termasuk mengelola beban kerja dan layanan. Kubernetes dirancang untuk mengotomatisasi penerapan, penskalaan, dan pengoperasian aplikasi dalam

container (Dewi et al. 2019) dan memiliki kemampuan untuk portabilitas dan ektensibilitas (Menouer 2021). Dalam implementasinya Kubernetes dapat dibantu oleh Skaffold yang merupakan *command-line tools* untuk menangani *workflow, building, pushing,* dan *deploying* aplikasi atau *service*. Skaffold dapat diterapkan pada *local* atau *remote* Kubernetes *cluster* saat pengembangan aplikasi (K. P. Singh 2020).

2.1.3 *Event-Driven Architecture* (EDA)

Event-Driven Architecture (EDA) mengacu pada sistem *microservice* yang digabungkan secara longgar dan bertukar informasi atau data satu sama lain melalui *event publish* dan *event listen*. EDA memungkinkan informasi diserap ke dalam ekosistem *event-driven* kemudian di-*broadcast* ke layanan yang me-*listen* atau layanan yang akan menerima *event* terkait (Jansen and Saladas 2020). EDA berkomunikasi menggunakan *message event* dan bekerja secara *asynchronous,* berbeda dengan *API-Driven Architecture* yang berkomunikasi menggunakan *API call* dan bekerja secara *synchronous* (Kannedy 2018). Dalam beberapa tahun terakhir, EDA telah banyak digunakan di beberapa domain seperti *network instruction detection, sensor networks,* pasar saham, *fast trading, realtime system control, healthcare monitoring, mobile and wearable computing*. Alasan utamanya adalah EDA memberikan solusi untuk mengembangkan sistem terdistribusi yang memfasilitasi fleksibilitas dan konkurensi yang tinggi (Tragatschnig, Stevanetic, and Zdun 2018).



Gambar 2.1 *Event-Driven Architecture Communication Process*

Seperti yang telah dijelaskan sebelumnya EDA berkomunikasi menggunakan *message event*, pada Gambar 2.1 digambarkan bahwa ketika *client* dari luar ekosistem *microservice* melakukan *request* maka akan diteruskan oleh *Load Balancer* dan diarahkan ke *service* tujuan, dimana *service* terkait akan memproses *request* dan mem-*publish event* ke *event-bus* dimana *service* lain yang me-*listen event* tersebut akan menerima *event notification* untuk ikut memproses *request* atau mensinkronisasi data. Setelah *request* selesai diproses maka akan mengembalikan *response* kepada *client*. Pembuatan *event-bus* dapat dibantu menggunakan Apache Kafka (Jansen and Saladas 2020), RabbitMQ (Hong, Sik Yang, and Kim 2018), atau NATS Streaming.

2.2 Penelitian Terkait (*State-Of-The-Art*)

Berikut merupakan penelitian terkait bidang *microservice*, baik itu pengembangan model, metode, algoritma maupun solusi yang ditawarkan atas permasalahan penelitian yang diteliti. Penelitian terkait sudah menjawab sedikitnya pertanyaan penelitian yang dilakukan terkait permasalahan kinerja pada *microservice* yaitu penggunaan *container* sebagai teknologi pendukung dalam penerapan arsitektur *microservice*. Selain bidang *microservice*, di antaranya penelitian mengenai *containerization*, dan *Event-Driven Architecture* disajikan pada Tabel 2.1.

Tabel 2.1 Matriks Penelitian Terkait *(State-Of-The-Art)*

No	Penulis dan Tahun Penelitian	Model/Metode/ Algoritma/Solusi	Hasil Penelitian
1	(Cavallari, Tornieri, and De Marco 2017)	*eService Techniques*	Temuan konseptual menyatakan bahwa *eService* sebagai bagian dari DevOps dan arsitektur *Microservice* dapat membantu industri untuk merangkul perubahan dalam bisnis dan pengembangan perangkat lunak. Temuan penelitian menunjukan bahwa *eService* merangkum semua konsep dan strategi bentuk organisasi baru untuk metodologi pengembangan perangkat lunak sebagai indikasi hasil literatur ilmiah.
2	(Kang, Le, and Tao 2016)	Penerapan *Containerization* dan *Microservice-Style Design* pada *OpenStack*	Menggunakan *OpenStack* sebagai studi kasus, dan meng-*containerization* infrastruktur *cloud* dan menggabungkannya dengan gaya arsitektur *microservice* sangat meningkatkan efisiensi operasional. Saat layanan infrastruktur ditangani menunjukan bahwa menjalankan layanan dalam *container* tidak cukup untuk mendapatkan manfaat penuh dari *containerization* dan *microservice-style*.
3	(Djogic, Ribic, and Donko 2018)	*Redesign Platform*	- Pemeliharaan jauh lebih baik karena perubahan kode dapat dilakukan hanya pada *microservice.* - *Deployment* ditingkatkan karena hanya menerapkan layanan yang diubah secara independen.

No	Penulis dan Tahun Penelitian	Model/Metode/ Algoritma/Solusi	Hasil Penelitian
3	(Djogic et al., 2018)	*Redesign Platform*	- Skalabilitas ditingkatkan karena dapat memiliki jumlah *instance* yang berbeda untuk *microservice* yang berbeda tanpa perubahan kode. - Manajemen sumber daya jauh lebih baik karena dapat meningkatkan jumlah *microservice*. - *Production deployment* ditingkatkan karena tidak perlu mematikan seluruh platform. Pesan akan disimpan dalam antrian dan setelah *deployment platform* akan melanjutkan pemrosesan. - Pengembangan di masa depan ditingkatkan karena tim yang berbeda dapat mengerjakan layanan yang berbeda dan menggunakan teknologi yang berbeda. - Platform hosting ditingkatkan karena sekarang dapat dilakukan juga di cloud.
4	(Monteiro et al. 2018)	*Beethoven*	Menyajikan *platform* untuk orkestrasi *microservice* yang memudahkan pembuatan aliran data *microservice* yang kompleks.
5	(Filip et al. 2018)	*Microservice Scheduling Model*	- *First Came First Served* (FCFS) paling cocok dengan persyaratan sistem *realtime* yang homogen sementara lingkungan yang lebih heterogen dapat mengambil keuntungan besar dari algoritma *greedy* (dan juga *time-space polynomial*) untuk penjadwalan tugas. - Meningkatkan pencocokan *subset* tugas ke *subset* mesin yang sesuai dapat mengoptimalkan penyeimbangan beban dari sistem yang dibatasi waktu / tenggat *realtime.* - Algoritma MM (*min-min*) memberikan waktu tunggu antrian rata-rata yang lebih baik yang berarti penundaan *job* yang lebih kecil. Peneliti memperhatikan bahwa, untuk mengirim tugas, metrik itu 7 kali lebih besar, tetapi dalam mengevaluasi fakta itu, peneliti juga mempertimbangkan bahwa mengirim *task* 5 kali lebih jarang dan lebih berat.
6	(Mazlami, Cito, and Leitner 2017)	Model ekstraksi *monolithic* ke *microservice*	- Evaluasi kinerja menunjukan bahwa, sebagian besar pendekatan berskala sehubungan dengan ukuran riwayat revisi (*logic coupling* and *contributor*). - Evaluasi kualitas menunjukan bahwa pendekatan peneliti dapat mengurangi ukuran tim *microservice* menjadi seperempat dari ukuran tim *monolithic* atau bahkan lebih rendah. Evaluasi kualitas dapat

No	Penulis dan Tahun Penelitian	Model/Metode/ Algoritma/Solusi	Hasil Penelitian
			memandu arsitek perangkat lunak untuk menggunakan pendekatan peneliti sesuai dengan kebutuhan mereka (yaitu, mengurangi ukuran tim dan menurunkan redudansi domain dari layanan yang diekstraksi).
7	(Otterstad and Yarygina 2017)	*The Security Monitor Service*	*The security monitor service* menjadi bagian dari infrastrukur yang mirip layanan *logging*, pemantauan, dan penemuan yang diperlukan agar sistem *microservice* berukuran wajar (kecil) dapat berfungsi dengan baik. Berbeda dengan layanan dasar, *security monitor* mencoba mengurangi serangan secara otonom, membuat keseluruhan sistem lebih tangguh terhadap eksploitasi tingkat rendah.
8	(Suryotrisongko 2017)	Model *proof of concept* berbasis *microservice* dan Docker *container*	Aspek resiliensi dapat diamati pada ketangguhan sistem ketika salah satu *instance microservice* mengalami gangguan. Dari sudut pandang *end-user*, gangguan ini tidak akan dapat dirasakan, karena backend telah ditangani oleh kombinasi antara *Eureka* dan *Zuul proxy* yang akan berfungsi sebagai *load-balancer* untuk membagi beban sekaligus untuk menyembunyikan gangguan yang dialami oleh sistem dibelakang *layer*. Dari hasil pengujian, dapat disimpulkan bahwa aspek kualitas resiliensi dapat dicapai pada arsitektur sistem yang diusulkan di penelitian ini.
9	(Balalaie, Heydarnoori, and Jamshidi Dermani 2016)	*Enables DevOps*	Pengalaman dan pembelajaran yang dipetik: - *Deployment di lingkungan pengembangan*: memastikan aplikasi yang dikembangkan dan yang didistribusikan sama. - *Service contract sangat penting*: perubahan kecil dalam kontrak dapat merusak sebagian dari sistem atau keseluruhan sistem. Maka solusi yang ditawarkan adalah *service versioning*, meskipun dapat membuat prosedur setiap layanan menjadi lebih kompleks. Dengan demikian, teknik seperti *Tolerant Reader* lebih direkomendasikan untuk menghidari pembuatan versi layanan. - *Pengembangan sistem terdistribusi membutuhkan pengembang yang terampil*: *Microservice* adalah gaya arsitektur terdistribusi. Untuk memaksimalkannya diperlukan beberapa layanan pendukung seperti *service discovery*, *load balancer*, dan sebagainya. - *Membuat service development template itu penting*: langkah awal *refactoring* arsitektural dimulai dengan membuat *service development template.*

No	Penulis dan Tahun Penelitian	Model/Metode/ Algoritma/Solusi	Hasil Penelitian
			- *Microservice bukanlah peluru perak: microservice* sangat bermanfaat untuk fleksibilitas sistem, dan tools *Spring Cloud* dan *Netflix OSS* dapat memudahkan produksi migrasi dan pengembangan. Namun, dengan mengadopsi *microservice* maka kompleksitas menjadi tantangan yang baru dan perlu upaya untuk menyelesaikannya.
10	(Guerrero, Lera, and Juiz 2017)	*Non-dominated Sorting Genetic Algorithm* II (NSGA-II)	Hasilnya menunjukan bahwa pendekatan memberikan solusi yang sesuai untuk pengalamatan masalah ini, dan menemukan solusi yang dioptimalkan di dalamnya jumlah generasi yang wajar (100) dan dengan ukuran populasi yang wajar (200). Hasilnya telah dibandingkan, diperoleh dengan implementasi Kubernetes *allocation policies*. Pendekatan menunjukan keunggulan perilaku, dengan mendapatkan nilai yang lebih kecil untuk keempat tujuan. Pendekatan memperoleh nilai hingga 58.1% untuk *Network Distance*, 44.1% untuk *Balanced Cluster*. Selebihnya, 44.1% untuk *System Failure*, dan 453.9% untuk *Threshold Distance*.
11	(Kratzke and Quint 2017)	*Ppbench*	- Bahasa pemrograman (atau pustaka HTTP dan TCP standar bahasa pemrograman terkait) mungkin berdampak besar pada kinerja REST. Selain itu, tiga dari empat bahasa (Go, Ruby, Java, Dart) yang dianalisis menunjukan perilaku jaringan yang tidak berkelanjutan, sehingga pesan yang berukuran beberapa *byte* lebih besar atau lebih kecil dapat menunjukan latensi atau kecepatan transfer yang sama sekali berbeda. Diidentifikasi efek tersebut 1/10 TCP_{window} (Ruby, Java) dan 3 x TCP_{window} (Dart). Menurut para peneliti tidak ada bahasa pemrograman yang menampilkan hasil terbaik untuk semua ukuran pesan yang dikirim. - *Container* dinyatakan ringan. Dampak kinerja *container* mungkin tidak terlalu berdampak namun tetap perlu diperhatikan. Kinerja pesan mungkin kecil lebih rentan terhadap dampak kinerja *container* daripada kinerja pesan besar. - Dampak SDN (*Software Defined Network*) bisa sangat sangat berdampak. Terutama untuk jenis mesin *small core.* Namun, karena efek *attenuation*, SDN bahkan dapat menunjukan efek positif jika terjadi perilaku jaringan yang tidak berkelanjutan. Jenis mesin dengan *more cores* dapat mengurangi dampak kinerja SDN karena efek *contention* CPU berkurang. Dampak SDN dapat dikurangi dengan jenis mesin virtual 8-core ke tingkat yang sebanding dengan *container*-nya.

No	Penulis dan Tahun Penelitian	Model/Metode/ Algoritma/Solusi	Hasil Penelitian
12	(V. Singh and Peddoju 2017)	*Container-based microservice architecture*	- *Virtual Machine* (VM) *vs Container*. *Container* mudah dikelola dengan ringan, mengungguli VM dalam banyak aspek. - *Monolithic vs Microservice* : Telah diamati bahwa aplikasi *microservice* mengungguli aplikasi *monolithic* dengan waktu respons yang leibh sedikit untuk sejumlah permintaan. Dalam kasus *microservice*, permintaan didistribusikan antar *microservice* independent yang berbeda. Di sisi lain, dalam aplikasi monolitik, semua perminataan ditangani oleh satu aplikasi yang berisi semua layanan dalam satu basis kode. Pendekatan *microservice* yang lebih lebih tinggi dengan lebih banyak jumlah permintaan yang ditangani per detik. - *Deployment and Scaling* : Setelah *deployment* awal aplikasi perlu ditingkatkan atau diturunkan skalanya sesuai dengan kebutuhan layanan tertentu. Karena setiap *microservice* diterapkan secara terpisah, mereka membutuhkan waktu yang relatif lebih sedikit daripada aplikasi monolitik. Selain itu pendekatan *microservice* memungkinkan aplikasi diskalakan pada tingkat *granular* dan karenanya dapat menskalakan *microservice* yang berbeda secara independen. - *Rolling Updates*: Pembaharuan berkelanjutan memungkinkan aplikasi *microservice* terus terintegrasi setiap kali ada beberapa pembaharuan untuk *microservice*. Karena monolitik memerlukan seluruh aplikasi untuk dikompilasi dan diterapkan ulang, butuh 400 detik untuk mengirimkan pembaharuan. Sedangkan untuk *microservice* butuh 160 detik untu mengirim pembaharuan. Yang menunjukan bagaiman aplikasi *microservice* yang diterapkan menggunakan sistem otonom dapat mengurangi *downtime* aplikasi dan terus memberikan pembaharuan dengan upaya minimal.
13	(Zhelev and Rozeva 2019)	*Microservice* dan EDA	*Microservice* dan EDA sangat cocok untuk membangun platform yang mengumpulkan dan memproses *Big Data stream.*
14	(Putra 2018)	Analisa implementasi arsitektur *microservice* berbasis *container* pada *Opendaylight project*	Setiap nilai dari variabel-variabel independen memiliki masing-masing peranan nilai yang berbeda dengan dijabarkan antara lain: - Faktor *source code commit*, tidak memiliki pengaruh terhadap kesuksesan implementasi arsitektur *microservice* yang cukup berarti.

No	Penulis dan Tahun Penelitian	Model/Metode/ Algoritma/Solusi	Hasil Penelitian
		(Analaisis kuantitatif)	- Faktor *merge source code* atau *repository* memiliki pengaruh daripada faktor keberhasilan implementasi arsitektur *microservice.*
15	(Khazaei et al. 2016)	Model untuk mendukung kasus penggunaan manajemen *microservice*	Berkat skalabilitas linier dari model analitis dan parameter realistis dari eksperimen, peneliti dapat mempelajari kinerja penyediaan platform *microservice* dalam skala besar. Hasilnya, dengan memanfaatkan model dan eksperimen yang diusulkan, analisis *what-if* dan perencanaan kapasitas untuk platform *microservice* dapat dilakukan secara sistematis dengan jumlah waktu dan biaya yang minimum.
16	(Yahia et al. 2016)	*Platform Medley for Service Composition*	Komposisi yang dihasilkan mengonsumsi jumlah yang cukup rendah sumber daya dan platform berskala baik pada server utama dan perangkat tertanam seperti Raspberry Pi. Dibandingkan dengan pendekatan tradisional berdasarkan BPEL atau ESB, Mendley memungkinkan adaptasi yang mulus pada saat menjalakan komposisi layanan berdasarkan ketersediaanya.
17	(Hong, Sik Yang, and Kim 2018)	*Advance Message Queuing Protocol (AMQP)*	Hasil percobaan yang diperoleh menunjukan bahwa ketika sejumlah besar pengguna mengirim permintaan ke aplikasi web pada saat yang sama, lebih stabil untuk menggunakan RabbitMQ sebagai middleware berorientasi pesan daripada metode komuniksai REST API.
18	(Akbulut and Perros 2019)	*API Gateway Design Pattern, Chain of Responsibility Design Pattern,* dan *Asynchronous Messaging Design Pattern*	Secara umum, tidak ada pola *microservice* yang lebih baik dari yang lain. Sebaliknya, setiap pola desain bekerja lebih baik dalam skenario yang berbeda. Arsitektur kompleks hadir dengan siklus pengembangan jangka Panjang dan biaya lisensi tambahan untuk aplikasi pihak ketiga. Cara termudah untuk mengevaluasi keberhasilan *microservice* adalah dengan memastikan bahwa layanan tersebut memenuhi atau melampaui kinerja pra-migrasi monolitik.
19	(Dewi et al. 2019)	Penerapan Kubernetes	Implementasi skalabilitas ke *container* (pada beberapa server) mengurangi penggunaan CPU karena distribusi beban ke *container* yang tersebar di banyak *worker*/*service.* Waktu respons di *multi-server* membutuhkan lebih lama daripada *single-server* karena penundaan *overhead* penskalaan *container*.

2.3 Kebaruan Penelitian

Beberapa jurnal terkait berhubungan dengan penggunaan arsitektur komunikasi, teknologi, tujuan dan objek penelitian dengan penelitian yang sedang dilakukan. Tabel 2.2 menggambarkan perbedaan penelelitian yang diusulkan dengan penelitian-penelitian terkait.

Tabel 2.2 Matriks Kebaruan Penelitian

No	Penulis	Ruang Lingkup										
		Architecture Communication		Technology		Purpose					Object	
		API-Driven	Event-Driven	Virtual Machine	Container	Implementation	Design	Analysis	Scalability	Performance	Application/Other	Microservice
1	(Hong, Sik Yang, and Kim 2018)	√	√					√		√		√
2	(Kurniawan et al. 2019)	√				√					√	
3	(Chandra et al. 2020)		√		√	√			√		√	
4	(Mulyono, Dwi, and Kurniawan 2019)	√				√	√				√	
5	(Fihri, Negara, and Sanjoyo 2019)	√			√	√					√	
6	**(Fuji, 2021)**		√		√	√		√		√		√