BAB II

TINJAUAN KEPUSTAKAAN

2.1 Arsitektur *Microservice*

2.1.1 Definisi Dan Karakteristik Dari Arsitektur *Microservice*

Arsitektur *microservice* adalah pendekatan yang digunakan dalam pengembangan aplikasi, di mana aplikasi dibagi menjadi beberapa komponen yang lebih kecil dan independen yang disebut "*microservice*" (Newman, 2021). Setiap *microservice* bertanggung jawab untuk menjalankan fungsionalitas spesifik dan berkomunikasi dengan *microservice* lain melalui API yang terdefinisi dengan baik (Lewis & Fowler, 2014). Karakteristik utama dari arsitektur *microservice* meliputi:

a. Modularitas: *microservice* memungkinkan pemisahan fungsionalitas menjadi modul-modul kecil yang independen dan mudah dikelola (Newman, 2015).
b. Skalabilitas: Setiap *microservice* dapat diskalakan secara individual sesuai dengan kebutuhan, memungkinkan aplikasi untuk mengakomodasi beban yang beragam (Lewis & Fowler, 2014).
c. Keberagaman teknologi: *microservice* memungkinkan penggunaan teknologi yang berbeda dalam satu aplikasi, karena setiap *microservice* dapat dikembangkan dengan bahasa pemrograman, kerangka kerja, dan basis data yang berbeda (Newman, 2021).

2.1.2 Kelebihan Dan Kekurangan Arsitektur *Microservice*

Microservice adalah suatu pendekatan dalam pengembangan perangkat lunak di mana aplikasi dibangun sebagai kumpulan dari layanan yang lebih kecil dan dapat beroperasi secara independen. Pendekatan ini memberikan sejumlah keuntungan yang signifikan, namun tentunya tidak luput dari beberapa tantangan dan kelemahan. Seperti dua sisi mata uang, pendekatan *microservice* memiliki kelebihan dan kekurangan yang harus dipertimbangkan sebelum memutuskan untuk menerapkannya.

Kelebihan arsitektur *microservice* meliputi (Lewis & Fowler, 2014; Newman, 2021):

a. Kemudahan pengembangan dan pemeliharaan: *microservice* memudahkan pengembangan dan pemeliharaan aplikasi dengan memisahkan fungsionalitas menjadi modul yang lebih kecil dan independen.
b. Fleksibilitas dalam pemilihan teknologi: *microservice* memungkinkan tim untuk menggunakan teknologi yang paling sesuai untuk kebutuhan spesifik mereka.
c. Resiliensi: Kegagalan dalam satu *microservice* umumnya tidak akan mengakibatkan kegagalan seluruh aplikasi, sehingga meningkatkan keandalan.

Kekurangan arsitektur *microservice* meliputi (Lewis & Fowler, 2014; Newman, 2021):

a. Kompleksitas: Mengelola dan mengkoordinasikan banyak *microservice* dapat meningkatkan kompleksitas sistem secara keseluruhan.
b. Tantangan komunikasi: Komunikasi antara *microservice* dapat menyebabkan *overhead* tambahan dan mempengaruhi kinerja aplikasi.
c. Kesulitan dalam pengujian: Pengujian aplikasi yang dibangun dengan arsitektur *microservice* mungkin lebih sulit karena ketergantungan antar layanan dan kebutuhan untuk menguji komunikasi antara layanan.

2.1.3 Implementasi Arsitektur *microservice* pada Aplikasi

Untuk mengimplementasikan arsitektur *microservice* pada aplikasi, langkah-langkah berikut dapat diikuti (Lewis & Fowler, 2014; Newman, 2021):

a. Identifikasi komponen: Tentukan fungsionalitas utama aplikasi dan pecah menjadi komponen independen yang dapat dikembangkan sebagai *microservice*.
b. Desain API: Buat API yang terdefinisi dengan baik untuk komunikasi antara *microservice.*

c. Pilih teknologi: Tentukan teknologi yang paling sesuai untuk setiap *microservice*, seperti bahasa pemrograman, kerangka kerja, dan basis data.
d. Pengembangan dan integrasi: Kembangkan setiap *microservice* secara individual dan integrasikan dengan aplikasi secara keseluruhan melalui API yang telah ditentukan.
e. Monitor dan kelola: Gunakan alat pemantauan dan manajemen yang sesuai untuk mengawasi kinerja, keandalan, dan skalabilitas *microservice* dalam aplikasi.

Dengan mengimplementasikan arsitektur *microservice*, aplikasi akan menjadi lebih modular, fleksibel, dan andal. Namun, penting untuk mempertimbangkan kompleksitas tambahan dan tantangan yang mungkin dihadapi selama pengembangan dan pengujian. Dalam banyak kasus, keuntungan dari arsitektur *microservice* akan melebihi kekurangannya, terutama dalam aplikasi skala besar yang memerlukan tingkat skalabilitas dan keandalan yang tinggi.

2.2 Optimasi Aplikasi

2.2.1 Definisi Optimasi Aplikasi

Dalam dunia teknologi, optimasi aplikasi dipandang sebagai upaya kritis yang memperbaiki kualitas aplikasi dari berbagai dimensi. Berdasarkan definisi yang diajukan oleh Fowler (2019), optimasi aplikasi adalah proses kompleks yang melibatkan peningkatan kinerja, efisiensi, dan keandalan sebuah aplikasi melalui implementasi berbagai teknik dan metode. Proses ini tidak hanya mencakup peningkatan kualitas kode dan struktur program, tetapi juga menyangkut aspek seperti peningkatan kinerja basis data, penyesuaian infrastruktur, dan peningkatan desain antarmuka pengguna.

Fokus utama dari optimasi aplikasi tidak terbatas pada perbaikan teknis semata, tetapi juga mencakup peningkatan kualitas pengalaman pengguna. Hal ini ditunjukkan oleh Souders (2007), yang menyatakan bahwa tujuan utama dari optimasi aplikasi adalah untuk memperbaiki pengalaman pengguna dan mencapai tujuan bisnis dengan memanfaatkan sumber daya yang ada secara efisien. Ini berarti

bahwa optimasi aplikasi juga melibatkan pemahaman mendalam tentang kebutuhan dan harapan pengguna, serta penyesuaian aplikasi dengan tujuan bisnis perusahaan.

Secara keseluruhan, optimasi aplikasi bukanlah suatu proses yang statis, tetapi merupakan siklus perbaikan berkelanjutan yang mengharuskan tim pengembangan untuk secara konsisten memantau kinerja aplikasi, menganalisis isu-isu yang ada, dan menerapkan perbaikan yang diperlukan. Optimasi aplikasi adalah elemen penting dalam siklus hidup pengembangan perangkat lunak, yang bertujuan untuk memaksimalkan nilai dari produk akhir baik untuk pengguna akhir maupun untuk perusahaan.

2.2.2 Jenis-jenis Optimasi Aplikasi

Kinerja aplikasi dapat dimaksimalkan melalui proses optimasi, yang menjadi langkah penting dalam pengembangan perangkat lunak. Beberapa aspek aplikasi, mulai dari penulisan kode hingga desain antarmuka pengguna dan konfigurasi infrastruktur, dapat ditingkatkan melalui optimasi. Berikut adalah beberapa jenis optimasi aplikasi yang dijelaskan beserta rujukan literatur yang relevan (Allspaw & Robbins, 2010; Bai, 2011; Krug, 2013; McConnell, 2004).

Optimasi aplikasi dapat dibagi menjadi beberapa jenis, di antaranya:

a. Optimasi kode: Melibatkan penulisan kode yang lebih efisien dan ringkas, meminimalkan kompleksitas, dan mengurangi redundansi.
b. Optimasi basis data: Meningkatkan kinerja akses dan manipulasi data dengan mengoptimalkan *query* dan struktur basis data.
c. Optimasi desain UI/UX: Meningkatkan keterlibatan pengguna dan kepuasan dengan memperbaiki desain antarmuka dan alur aplikasi.
d. Optimasi infrastruktur: Menyesuaikan kapasitas dan konfigurasi sistem untuk memperbaiki kinerja dan skalabilitas aplikasi.

2.2.3 Manfaat Optimasi Aplikasi

Di dalam konteks pengembangan perangkat lunak, optimasi aplikasi memberikan berbagai manfaat signifikan yang membawa dampak positif baik untuk pengguna maupun pengembang. Beberapa manfaat penting ini mencakup peningkatan kinerja, penggunaan sumber daya yang efisien, keandalan, stabilitas,

dan peningkatan pengalaman pengguna. Detail dari manfaat ini akan dijelaskan lebih lanjut.

a. Peningkatan kinerja: Aplikasi yang dioptimalkan menawarkan waktu respons yang lebih cepat dan memproses data dengan lebih efisien (Souders, 2007).
b. Penggunaan sumber daya yang efisien: Optimasi mengurangi konsumsi sumber daya seperti memori, CPU, dan *bandwidth*, sehingga mengurangi biaya operasional (Allspaw & Robbins, 2010).
c. Keandalan dan stabilitas: Aplikasi yang dioptimalkan cenderung memiliki risiko kegagalan yang lebih rendah dan lebih mudah dikelola (Fowler, 2019).
d. Pengalaman pengguna yang lebih baik: Optimasi aplikasi menghasilkan antarmuka yang lebih responsif, intuitif, dan mudah digunakan, sehingga meningkatkan kepuasan pengguna (Krug, 2013).

2.2.4 Strategi Optimasi Aplikasi

Optimasi aplikasi adalah suatu proses yang melibatkan serangkaian strategi dan teknik yang secara spesifik dirancang untuk memperbaiki kinerja dan efisiensi aplikasi. Berbagai strategi ini mencakup, namun tidak terbatas pada, *profiling* dan *benchmarking*, analisis dan perbaikan kode, penyesuaian basis data, optimasi desain dan alur aplikasi, *monitoring* dan analisis data, serta kompresi pengiriman data.

a. *Profiling* dan *benchmarking*: Mengukur kinerja aplikasi untuk mengidentifikasi area yang memerlukan perbaikan (Myers dkk., 2011).
b. Analisis dan perbaikan kode: Mengkaji kode sumber dan mengimplementasikan perubahan yang diperlukan untuk meningkatkan kinerja dan efisiensi (McConnell, 2004).
c. Penyesuaian basis data: Mengoptimalkan *query* dan struktur basis data untuk mempercepat akses dan manipulasi data (Bai, 2011).

d. Optimasi desain dan alur aplikasi: Memperbaiki desain antarmuka dan alur aplikasi untuk meningkatkan keterlibatan dan kepuasan pengguna (Krug, 2013).
e. Pemantauan dan analisis data: Mengumpulkan dan menganalisis data pengguna untuk mengidentifikasi area yang memerlukan perbaikan dan menyesuaikan strategi optimasi (Allspaw & Robbins, 2010).
f. Kompresi pengiriman data: Mengurangi besar ukuran data dengan cara mengompresi sebelum mengirim data.

Dengan menerapkan strategi optimasi ini, diharapkan aplikasi menjadi lebih efisien, responsif, dan ramah pengguna, menciptakan pengalaman yang lebih baik bagi pengguna dan membantu organisasi mencapai tujuan bisnisnya.

2.3 HPack

2.3.1 Definisi HPack

HPack adalah algoritma kompresi yang digunakan untuk mengurangi ukuran data dalam komunikasi HTTP/2 menggunakan *Huffman encoding*. Algoritma ini memanfaatkan struktur tabel dinamis yang memungkinkan pengkodean dan dekoding *Header* yang lebih efisien, mengurangi *overhead*, dan meningkatkan kinerja (Peon & Ruellan, 2015). Tujuan utama dari HPack adalah untuk mengurangi latensi dan meningkatkan kecepatan transfer data dalam protokol HTTP/2.

2.3.2 Fungsi dan Kelebihan HPack

Fungsi utama HPack adalah untuk mengompresi *Header* pada protokol HTTP/2, sehingga mengurangi ukuran data yang perlu ditransmisikan. kelebihan HPack meliputi Efisiensi kompresi, HPack menggunakan struktur tabel dinamis dan statis yang mengurangi redundansi dalam data, sehingga menghasilkan tingkat kompresi yang lebih baik (Peon & Ruellan, 2015).

2.3.3 Cara Kerja HPack

HPack bekerja dengan menggunakan dua tabel, yaitu tabel statis dan tabel dinamis, untuk mengasosiasikan *field header* ke indeks. Tabel statis adalah tabel yang telah ditentukan sebelumnya dan berisi *field header* yang umum (sebagian

besar dengan nilai kosong). Sedangkan tabel dinamis bersifat dinamis dan dapat digunakan oleh pengode untuk mengindeks *field header* yang diulangi dalam daftar *header* yang dikodekan (Peon & Ruellan, 2015).

Proses kompresi di HPack tidak secara spesifik menggambarkan algoritma untuk *encoder*. Sebaliknya, HPack mendefinisikan dengan tepat bagaimana *decoder* diharapkan beroperasi, memungkinkan *encoder* menghasilkan *encoding* apa pun yang diperbolehkan oleh definisi tersebut.

Untuk mendekompresi blok *header*, *decoder* hanya perlu mempertahankan tabel dinamis sebagai konteks *decoding*. Tidak ada status dinamis lain yang diperlukan. Saat digunakan untuk komunikasi dua arah, seperti dalam HTTP, tabel dinamis *encoding* dan *decoding* yang dipertahankan oleh titik akhir sepenuhnya independen, yaitu, tabel dinamis permintaan dan respons adalah terpisah (Peon & Ruellan, 2015).

Decoder memproses blok *header* secara berurutan untuk merekonstruksi daftar *header* asli. Blok *header* adalah penggabungan representasi *field header*. Setelah *field header* didekode dan ditambahkan ke daftar *header* yang direkonstruksi, *field header* tidak dapat dihapus.*field header* yang ditambahkan ke daftar *header* dapat dengan aman diteruskan ke aplikasi. Dengan meneruskan *field header* yang dihasilkan ke aplikasi, *decoder* dapat diimplementasikan dengan komitmen memori transitoris minimal selain memori yang diperlukan untuk tabel dinamis (Peon & Ruellan, 2015).

Semua representasi *field header* yang terkandung dalam blok *header* diproses dalam urutan munculnya. Representasi yang di indeks melibatkan penambahan *field header* yang sesuai dengan entri yang dirujuk dalam tabel statis atau dinamis ke daftar *header* yang didekodekan. Representasi literal yang tidak ditambahkan ke tabel dinamis melibatkan penambahan *field header* ke daftar *header* yang didekodekan. Representasi literal yang ditambahkan ke tabel dinamis melibatkan penambahan *field header* ke daftar *header* yang didekodekan dan penambahan *field header* ke awal tabel dinamis. Penambahan ini bisa mengakibatkan pengusiran entri sebelumnya di tabel dinamis (Peon & Ruellan, 2015).

Untuk membatasi kebutuhan memori pada sisi *decoder*, ukuran tabel dinamis dibatasi. Ukuran tabel dinamis adalah jumlah ukuran entri-entrinya. Ukuran entri adalah jumlah panjang namanya dalam oktet, panjang nilainya dalam oktet, dan 32. Ukuran entri dihitung menggunakan panjang namanya dan nilai tanpa *Huffman Encoding* yang diterapkan.

2.4 Gzip

Gzip adalah program kompresi data yang dikembangkan oleh Jean-loup Gailly. Program ini menggunakan algoritma Lempel-Ziv (LZ77) untuk mengurangi ukuran *file*. Gzip menggantikan *file* asli dengan *file* yang memiliki ekstensi '.gz', sambil mempertahankan kepemilikan, mode, dan waktu akses dan modifikasi yang sama. Gzip hanya akan mencoba mengompresi *file* reguler dan akan mengabaikan *symbolic links* .

2.4.1 Cara Kerja Gzip

Gzip menggunakan metode kompresi yang disebut DEFLATE. DEFLATE merupakan kombinasi dari algoritma kompresi LZ77 (Lempel-Ziv 1977) dan *huffman coding*. Algoritma LZ77 mencoba untuk mengidentifikasi dan menghapus redundansi dalam data dengan mencari pola kesamaan antara *string-string* yang berdekatan. Setelah itu, *Huffman coding* digunakan untuk mengurangi ukuran data dengan memberikan representasi bit yang lebih pendek untuk *string-string* yang lebih umum.

langkah-langkah utama dalam proses kompresi dengan Gzip:

a. Membaca *file* Terkompresi: Gzip memulai dengan membaca *file* terkompresi yang akan didekompresi.
b. Mengembalikan *Huffman Coding*: Gzip menggunakan informasi tambahan dalam *file* terkompresi untuk mengembalikan tabel Huffman *coding* yang digunakan dalam proses kompresi.
c. Menggabungkan *String*: Gzip menggabungkan *string-string* yang ditemukan dengan menggunakan kamus kompresi yang disertakan dalam *file* terkompresi.

d. Pengembalian Data Asli: Setelah penggabungan *string*, Gzip mengembalikan data asli dengan mengganti penunjuk kamus dengan *string-string* yang sesuai.
e. Rekonstruksi Data: Terakhir, Gzip merekonstruksi data asli dengan menggabungkan *string-string* yang telah dikembalikan sesuai dengan urutan aslinya.

2.4.2 Keuntungan Menggunakan Gzip

Gzip biasanya dapat mengurangi ukuran teks seperti kode sumber atau bahasa Inggris sebesar 60-70%. Kompresi umumnya jauh lebih baik daripada yang dicapai oleh LZW (seperti yang digunakan dalam kompres), Huffman *coding* (seperti yang digunakan dalam HPack), atau *adaptive Huffman coding* (*compact*) (Gailly, 2022).

2.5 *JSON* (*JavaScript Object Notation*)

JSON (*JavaScript Object Notation*) adalah format pertukaran data yang ringan dan mudah dibaca oleh manusia serta mudah diinterpretasikan oleh komputer. Dalam konteks pengembangan web dan aplikasi, *JSON* digunakan secara luas untuk mentransmisikan dan menyimpan data struktural.

2.5.1 Definisi dan Struktur *JSON*

JSON merupakan representasi struktural data yang terdiri dari pasangan "kunci-nilai" (*key-value pairs*). Data dalam format *JSON* disusun dalam objek (*object*), larik (*array*), dan tipe data primitif seperti *string*, angka, *boolean*, dan nilai *null*. Format *JSON* sering kali mengadopsi sintak mirip dengan objek dan larik dalam *JavaScript*, sehingga mudah dipahami oleh para pengembang (Tiwary dkk., 2021).

2.5.2 Keunggulan *JSON*

Salah satu keunggulan utama *JSON* adalah keterbacaannya (Tiwary dkk., 2021). Dengan menggunakan sintak yang intuitif dan mudah dipahami, *JSON* memungkinkan pengembang dan pengguna manusia untuk membaca dan mengerti data dengan mudah. Selain itu, *JSON* juga mendukung interoperabilitas, yang berarti format ini dapat digunakan secara luas dalam berbagai bahasa pemrograman

dan dapat diimplementasikan di berbagai platform. *JSON* juga sering digunakan dalam API (*Application Programming Interface*) untuk mentransmisikan data antara server dan klien. Format *JSON* juga memungkinkan pengolahan data struktural dengan mudah, termasuk *parsing*, manipulasi, dan serialisasi.

2.5.3 Implementasi *JSON*

Pemrosesan *JSON* dapat dilakukan baik pada sisi server maupun pada sisi klien. Pada sisi server, banyak bahasa pemrograman menyediakan pustaka atau modul untuk membaca dan menulis data *JSON*. Sebagai contoh, PHP menyediakan fungsi *json_encode*() dan *json*_decode(), sedangkan Python menyediakan modul *json*. Pada sisi klien, *JavaScript* memiliki metode *built-in* untuk *parsing* dan mengonversi data *JSON* menjadi objek yang dapat diakses. Dengan menggunakan metode *JSON*.parse(), data *JSON* dapat diubah menjadi objek *JavaScript* yang dapat dimanipulasi dan digunakan dalam pengembangan aplikasi web (Tiwary dkk., 2021).

JSON digunakan dalam berbagai kasus penggunaan di dalam pengembangan aplikasi dan pertukaran data. Salah satu contoh penggunaan umum adalah dalam komunikasi data antara server dan klien melalui API REST menggunakan format *JSON*. Dalam arsitektur ini, server akan menyediakan *endpoint* API yang mengirimkan data dalam format *JSON* kepada klien, yang kemudian dapat mengonsumsi data tersebut. Selain itu, *JSON* juga sering digunakan untuk menyimpan konfigurasi dan pengaturan aplikasi dalam *file JSON* yang dapat dengan mudah diakses dan dikelola. Selain itu, *JSON* juga memungkinkan pertukaran data antara aplikasi yang berbeda dengan menggunakan format yang seragam dan mudah dipahami.

2.6 Penelitian Terdahulu

Subbab ini memberikan gambaran tentang penelitian-penelitian terdahulu yang relevan dengan topik skripsi ini, yaitu implementasi HPACK dan GZIP pada optimasi aplikasi dengan arsitektur *microservice*. Beberapa penelitian sebelumnya telah dipelajari dan dijadikan sebagai referensi dalam penulisan skripsi ini.

Sebuah penelitian yang dilakukan oleh Sergiu Jecan dan Nicolae Goga, berjudul "*The Influence of the Compression Algorithms on the Information Transmitted Through the HTTP Protocol*", telah menunjukkan pengaruh signifikan dari algoritma kompresi terhadap kualitas informasi yang ditransmisikan melalui protokol HTTP. Hasil penelitian mereka menunjukkan bahwa algoritma GZIP memberikan hasil terbaik dalam hal efisiensi kompresi dan kecepatan transmisi data. Penelitian lain yang relevan adalah "*Multi-Objective Optimization of Container-Based Microservice Scheduling in Edge Computing*" oleh Guisheng Fan, Liang Chen, Huiqun Yu, dan Wei Qi. Penelitian ini berfokus pada optimasi *multi*-objektif dari penjadwalan *microservice* berbasis kontainer dalam *edge computing*.

Selanjutnya, penelitian berjudul "*Ant Colony Algorithm for Multi-Objective Optimization of Container-Based Microservice Scheduling in Cloud*" oleh Miao Lin, Jianqing Xi, Weihua Bai, dan Jiayin Wu membahas tentang optimasi *multi*-objektif dari penjadwalan *microservice* berbasis kontainer dalam *cloud computing*. Penelitian ini menawarkan algoritma *Ant Colony* sebagai solusi untuk permasalahan penjadwalan tersebut.

Meskipun fokus dan konteks dari penelitian-penelitian ini berbeda dengan skripsi ini, mereka memberikan wawasan penting tentang cara kerja algoritma kompresi dan arsitektur *microservice*, yang menjadi bagian penting dari skripsi ini. Pengaruh algoritma kompresi dan arsitektur *microservice* pada kualitas dan efisiensi aplikasi menjadi pertimbangan penting dalam penelitian ini.

Tabel 2.1 Penelitian Terdahulu

No.	Judul	Peneliti
1	"*The Influence of the Compression Algorithms on the Information Transmitted Through the HTTP Protocol*"	Sergiu Jecan, Nicolae Goga
	Hasil Penelitian: Dalam penelitian ini, ditunjukkan bahwa kualitas informasi yang ditransmisikan melalui protokol HTTP dapat dipengaruhi oleh algoritma kompresi. Beberapa algoritma kompresi yang berbeda dibandingkan oleh peneliti, dan ditemukan bahwa algoritma GZIP memberikan hasil terbaik dalam hal efisiensi kompresi dan kecepatan transmisi data.	
	Perbedaan: Antara dua penelitian ini terletak pada fokus dan konteks mereka. Pengaruh algoritma kompresi pada kualitas informasi yang ditransmisikan melalui protokol HTTP secara umum menjadi fokus dalam penelitian pertama, sementara penelitian kedua lebih spesifik, dengan berfokus pada implementasi HPACK dan GZIP dalam konteks aplikasi dengan arsitektur *microservice*. Meskipun keduanya membahas tentang algoritma kompresi, konteks dan tujuan mereka berbeda.	
2	"*Multi-Objective Optimization of Container-Based Microservice Scheduling in Edge Computing*"	Guisheng Fan, Liang Chen, Huiqun Yu, dan Wei Qi
	Hasil Penelitian: Dalam penelitian ini, dibahas tentang optimasi *multi*-objektif dari penjadwalan *microservice* berbasis kontainer dalam *edge computing*. Algoritma LRLBAS (*Latency, Reliability, and Load Balancing Aware Scheduling*) yang didasarkan pada *particle swarm optimization* (PSO) diusulkan oleh peneliti. Tujuan dari algoritma ini adalah mengoptimalkan latensi jaringan antar *microservice*, keandalan aplikasi *microservice*, dan keseimbangan beban kluster.	

Tabel 2.1 Tabel Penelitian (Lanjutan)

No.	Judul	Peneliti
	Perbedaan: Perbedaan antara dua penelitian ini terletak pada fokus dan konteks mereka. Optimasi *multi*-objektif dari penjadwalan *microservice* berbasis kontainer dalam *edge computi*ng menjadi fokus dalam penelitian pertama, sementara penelitian kedua lebih spesifik, dengan berfokus pada implementasi HPACK dan GZIP dalam konteks aplikasi dengan arsitektur *microservice*. Meskipun keduanya membahas tentang arsitektur *microservice*, konteks dan tujuan mereka berbeda.	
3	"*Ant Colony Algorithm for Multi-Objective Optimization of Container-Based Microservice Scheduling in Cloud*"	Miao Lin, Jianqing Xi, Weihua Bai, Jiayin Wu
	Hasil Penelitian: Dalam penelitian ini, dibahas tentang optimasi *multi*-objektif dari penjadwalan *microservice* berbasis kontainer dalam cloud computing. Algoritma *Ant Colony* diusulkan oleh peneliti untuk menyelesaikan masalah penjadwalan. Algoritma ini mempertimbangkan bukan hanya pemanfaatan sumber daya komputasi dan penyimpanan dari *node* fisik, tetapi juga jumlah permintaan *microservice* dan tingkat kegagalan dari *node* fisik. Hasil eksperimental menunjukkan bahwa algoritma optimasi yang diusulkan mencapai hasil yang lebih baik dalam optimasi keandalan layanan kluster, keseimbangan beban kluster, dan *overhead* transmisi jaringan.	
	Perbedaan: Diantara dua penelitian ini terletak pada fokus dan konteks mereka. Optimasi *multi*-objektif dari penjadwalan *microservice* berbasis kontainer dalam *cloud computing* menjadi fokus dalam penelitian pertama, sementara penelitian kedua lebih spesifik, dengan berfokus pada implementasi HPACK dan GZIP dalam konteks aplikasi dengan arsitektur *microservice*. Meskipun keduanya membahas tentang arsitektur *microservice*, konteks dan tujuan mereka berbeda.	