IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Diterima 30 November 2021, disetujui 16 Februari 2022, tanggal publikasi 18 Februari 2022, tanggal versi terkini 28 Februari 2022.

Pengidentifikasi Objek Digital 10.1109/ACCESS.2022.3152803

Arsitektur Monolitik vs. Arsitektur Layanan Mikro: Evaluasi Kinerja dan Skalabilitas

GRZEGORZ BLINOWSKI[1](Anggota IEEE), ANNA OJDOWSKA2DAN ADAM PRZYBYŸEK2
[1]1Institut Ilmu Komputer, Universitas Teknologi Warsawa, 00-665 Warsawa, Polandia
2Fakultas Elektronika, Telekomunikasi dan Informatika, Universitas Teknologi Gdaÿsk, 80-233 Gdaÿsk, Polandia

Penulis korespondensi: Grzegorz Blinowski (grzegorz.blinowski@pw.edu.pl)

ABSTRAK **Konteks.** Sejak diproklamasikan pada tahun 2012, arsitektur berbasis microservices telah mendapatkan popularitas luas karena keunggulannya, seperti peningkatan ketersediaan, toleransi kesalahan, dan skalabilitas horizontal, serta peningkatan ketangkasan pengembangan perangkat lunak. **Motivasi.** Namun, melakukan refactoring monolit menjadi microservices oleh bisnis kecil dan mengharapkan migrasi tersebut akan membawa manfaat serupa dengan yang dilaporkan oleh perusahaan global terkemuka, seperti Netflix, Amazon, eBay, dan Uber, mungkin merupakan ilusi. Memang, untuk sistem yang tidak memiliki ribuan pengguna bersamaan dan dapat diskalakan secara vertikal, manfaat dari migrasi tersebut belum cukup diteliti, sementara bukti yang ada tidak konsisten.
Tujuan. Tujuan makalah ini adalah untuk membandingkan kinerja dan skalabilitas arsitektur monolitik dan mikroservis pada aplikasi web referensi. **Metode.** Aplikasi diimplementasikan dalam empat versi berbeda, yang mencakup tidak hanya dua gaya arsitektur yang berbeda (monolitik vs. mikroservis) tetapi juga dua teknologi implementasi yang berbeda (Java vs. C# .NET). Selanjutnya, kami melakukan serangkaian eksperimen terkontrol di tiga lingkungan penyebaran yang berbeda (lokal, Azure Spring Cloud, dan Azure App Service). **Temuan.** Pelajaran utama yang dipetik adalah sebagai berikut: (1) pada satu mesin, monolitik berkinerja lebih baik daripada rekanan berbasis mikroservisnya; (2) platform Java memanfaatkan mesin yang lebih kuat dalam kasus layanan yang intensif komputasi dibandingkan dengan .NET; efek platform teknologi berbalik ketika layanan yang tidak intensif komputasi dijalankan pada mesin dengan kapasitas komputasi rendah; (3) penskalaan vertikal lebih hemat biaya daripada penskalaan horizontal di cloud Azure; (4) penskalaan di luar jumlah instance tertentu menurunkan kinerja aplikasi; (5) teknologi implementasi (baik Java atau C# .NET) tidak memiliki dampak yang nyata terhadap kinerja skalabilitas.

INDEKS ISTILAH Arsitektur perangkat lunak, layanan mikro, monolit, pengukuran perangkat lunak, benchmarking, kinerja, skalabilitas, komputasi awan, Azure.

I. PENDAHULUAN

Evolusi arsitektur perangkat lunak didorong oleh kebutuhan untuk mencapai pemisahan tanggung jawab yang lebih baik. Istilah pemisahan tanggung jawab mengacu pada kemampuan untuk menguraikan dan mengatur sistem menjadi modul-modul yang kohesif secara logis dan terhubung secara longgar yang menyembunyikan implementasinya satu sama lain dan menyajikan layanan melalui antarmuka yang terdefinisi dengan baik [1], [2].

Saat ini, dua paradigma rekayasa perangkat lunak mendominasi pengembangan aplikasi perusahaan modern: arsitektur monolitik dan berbasis microservice [3]. Yang pertama adalah pendekatan tradisional di mana sebuah aplikasi dibangun dengan basis kode tunggal yang mencakup banyak layanan. Layanan-layanan ini tidak dapat dieksekusi secara independen [4]. Mereka berkomunikasi dengan pengguna akhir dan sistem eksternal melalui antarmuka yang berbeda, termasuk HTTP(S)/HTML, layanan Web, dan REST API [5].

Arsitektur microservice menguraikan domain bisnis menjadi konteks-konteks kecil yang konsisten dan terdefinisi dengan baik yang diimplementasikan oleh layanan-layanan otonom, mandiri, terhubung secara longgar, dan dapat diimplementasikan secara independen [6]–[8]. Salah satu pelopor microservice adalah Netflix yang mulai beralih dari arsitektur monolitiknya pada tahun 2009 ketika istilah *microservice* bahkan belum ada. Istilah ini diciptakan oleh sekelompok arsitek perangkat lunak pada tahun 2011 dan secara resmi diumumkan setahun kemudian pada Konferensi Degree ke-33 di Namun, hal ini baru mulai populer pada tahun 2014, ketika Lewis dan Fowler menerbitkan blog mereka tentang topik tersebut [6], sementara Netflix membagikan keahlian mereka dari transisi yang sukses [9] yang membuka jalan bagi perusahaan lain. Sejak saat itu

Editor rekanan yang mengoordinasikan peninjauan manuskrip ini dan Muhammad Ali Babar menyetujui publikasinya.

Seiring berjalannya waktu, microservices telah mendapat perhatian yang signifikan baik dari kalangan akademisi maupun industri [4], [10]–[18], sementara penyebaran teknologi kontainer, seperti Kubernetes dan Docker [19]–[22] telah membantu gaya arsitektur baru ini untuk mendapatkan momentum yang lebih besar, terutama di lingkungan berbasis cloud [23]–[25]. Memang, microservices telah berhasil diadopsi oleh perusahaan global, seperti Amazon, eBay, Zalando, Spotify, Uber, Airbnb, LinkedIn, Twitter, Groupon, dan Coca-Cola.

A. MOTIVASI DAN PERNYATAAN MASALAH

Terinspirasi oleh kisah sukses perusahaan teknologi raksasa, banyak perusahaan kecil atau startup mempertimbangkan untuk bergabung dengan tren ini dan mengadopsi microservices sebagai pengubah permainan. Mereka berharap hal ini akan membantu mereka meningkatkan skalabilitas, ketersediaan, pemeliharaan, dan toleransi kesalahan aplikasi yang diterapkan [26], [27] (yang dilaporkan sulit dicapai dalam sistem TI [28], [29]). Namun, aplikasi berbasis microservices memiliki tantangan tersendiri, termasuk:

- mengidentifikasi batas-batas microservice yang optimal [30], [31]; • orkestrasi layanan yang kompleks (kompleksitas aplikasi microservice didorong dari komponen ke tingkat integrasi [15], [23]); • menjaga konsistensi data dan manajemen transaksi di seluruh microservice [15], [19], [32], [33]; • kesulitan dalam memahami sistem secara holistik [7], [27]; • peningkatan konsumsi sumber daya komputasi [7], [11], [17].

Untuk sistem skala kecil, tantangan ini mungkin lebih besar daripada manfaatnya [27].

Sebaliknya, perusahaan global yang disebutkan di atas beralih ke microservices sebagai respons terhadap tekanan pertumbuhan yang mereka hadapi, bukan karena mengikuti tren terbaru [32]. Volume aktivitas yang dilakukan sistem mereka melebihi kapasitas pilihan teknologi awal, sementara ukuran sistem yang sangat besar memperlambat pengembangan yang dilakukan oleh banyak tim [30]. Oleh karena itu, microservices telah menjadi solusi yang menarik bagi mereka meskipun menambah kompleksitas dalam membangun dan menjalankan aplikasi terdistribusi yang terperinci [32].

Selain itu, ada ribuan bisnis sukses di seluruh dunia yang dibangun di atas aplikasi monolitik.
Jadi, kapan sebuah perusahaan membutuhkan microservices? Sayangnya, pengetahuan tentang topik ini masih terbatas karena kurangnya bukti empiris. Setelah melihat kesenjangan ini, kami berupaya membandingkan kinerja dan skalabilitas arsitektur monolitik dan microservice dalam konteks sistem yang tidak memiliki ribuan pengguna bersamaan dan dapat diskalakan secara vertikal. Pertanyaan penelitian berikut diajukan untuk memandu studi ini:

- **(RQ1)** Apa perbedaan kinerja antara aplikasi monolitik dan aplikasi microservice?
- **(RQ2)** Dari kedua arsitektur dan pendekatan penskalaan tersebut, manakah yang sebaiknya dipilih agar aplikasi mendapatkan manfaat terbaik dari penskalaan?
- **(RQ3)** Dalam keadaan apa teknologi implementasi (Java vs. C# .NET) memiliki keunggulan atau kekurangan kinerja?

B. GARIS BESAR MAKALAH

Bagian selanjutnya dari makalah ini disusun sebagai berikut. Bagian selanjutnya menjelaskan gaya arsitektur monolitik dan mikroservis, serta membandingkan dan membedakan kelebihan dan kekurangannya. Bagian 3 membahas karya terkait. Metode penelitian dan desain eksperimental dijelaskan di Bagian 4. Ini diikuti oleh Bagian 5, yang berisi hasil eksperimen. Di Bagian 6, kami membahas temuan kami.

Pada Bagian 7, kami menguraikan ancaman terhadap validitas yang relevan dengan penelitian kami dan bagaimana kami mengatasinya. Terakhir, Bagian 8 menyimpulkan penelitian ini beserta saran untuk penelitian selanjutnya.

II. LATAR BELAKANG

Pada bagian selanjutnya, kami akan menyajikan detail lebih lanjut mengenai persamaan dan perbedaan, serta kelebihan dan kekurangan dari kedua gaya arsitektur tersebut, dan juga dua pendekatan utama untuk penskalaan aplikasi.

A. ARSITEKTUR MONOLITIK

Aplikasi perusahaan sering dibangun secara internal menurut model tiga tingkat klasik dan karenanya terdiri dari: (1) kode antarmuka pengguna (biasanya halaman HTML dan JavaScript yang berjalan di browser pada mesin pengguna); (2) logika bisnis sisi server yang menangani permintaan HTTP, mengeksekusi logika domain, mengambil dan memperbarui data dari basis data, dan memilih serta mengisi tampilan HTML untuk dikirim ke browser; dan (3) backend basis data [34]. Aplikasi sisi server adalah monolit - sebuah eksekusi logis tunggal [6].

Dari sudut pandang sistem operasi, aplikasi monolitik berjalan sebagai satu proses tunggal di lingkungan server aplikasi. Ketika versi baru aplikasi diimplementasikan, versi tersebut menggantikan versi sebelumnya dalam satu langkah (misalnya, untuk mengimplementasikan aplikasi di bawah JEE Application Server, satu file EAR/WAR yang berisi file eksekusi aplikasi harus disalin ke folder yang ditentukan).

Keunggulan paling signifikan dari arsitektur monolitik adalah kesederhanaannya – dibandingkan dengan aplikasi terdistribusi dari berbagai genre, aplikasi monolitik jauh lebih mudah untuk diuji, diterapkan, di-debug, dan dipantau. Semua data disimpan dalam satu basis data tanpa perlu sinkronisasi; semua komunikasi internal dilakukan melalui mekanisme intra-proses.
Oleh karena itu, pendekatan ini cepat dan tidak mengalami masalah yang lazim terjadi pada komunikasi antar proses (IPC). Pendekatan monolitik adalah pendekatan alami dan pilihan utama untuk membangun aplikasi – semua logika untuk menangani permintaan berjalan dalam satu proses. Fitur dasar dari tim pengembang adalah...

Bahasa pilihan dapat digunakan untuk menyusun aplikasi ke dalam kelas, fungsi, dan namespace.

Namun, seiring bertambahnya ukuran dan kompleksitas aplikasi, masalah mulai muncul – memodifikasi kode sumber aplikasi menjadi lebih sulit karena kode yang semakin kompleks mulai berperilaku dengan cara yang tidak terduga. Perubahan pada satu modul dapat menyebabkan perilaku yang tidak terduga pada modul lain dan serangkaian kesalahan. Ukuran monolit itu sendiri mengakibatkan waktu mulai yang lebih lama, yang pada gilirannya memperlambat pengembangan dan menjadi hambatan bagi penerapan berkelanjutan. Seiring waktu, semakin sulit bagi tim pengembang untuk menjaga agar perubahan yang terkait dengan modul tertentu hanya memengaruhi modul tersebut, dan pada akhirnya, untuk mempertahankan struktur modular aplikasi. Selain itu, seiring bertambahnya ukuran aplikasi, jumlah pengembang meningkat, yang seringkali menyebabkan pemanfaatan tenaga kerja yang tidak merata dan, akibatnya, kerugian produktivitas [34].

B. ARSITEKTUR LAYANAN MIKRO

Salah satu upaya pertama untuk mendeskripsikan gaya arsitektur microservice dilakukan oleh Lewis dan Fowler [6]. Dalam postingan blog mereka yang terkenal, mereka mendefinisikan istilah arsitektur baru ini sebagai "pendekatan untuk mengembangkan aplikasi tunggal sebagai serangkaian layanan kecil, masing-masing berjalan dalam prosesnya sendiri dan berkomunikasi dengan mekanisme ringan, seringkali API sumber daya HTTP". Setiap microservice berisi fungsi penanganan pengguna, logika bisnis, dan fungsi backend-nya sendiri.
Microservice juga dapat menyertakan layanan basis datanya sendiri (tetapi dimungkinkan juga untuk berbagi satu backend di antara beberapa microservice).

Prinsip utama arsitektur ini adalah: • Tanggung jawab tunggal per layanan – menurut prinsip SOLID, satu unit hanya boleh memiliki satu tanggung jawab dan tidak boleh ada dua unit yang berbagi satu tanggung jawab atau satu unit memiliki lebih dari satu tanggung jawab. • Microservices bersifat otonom – mereka adalah layanan mandiri dan dapat diimplementasikan secara independen yang sepenuhnya bertanggung jawab untuk menjalankan bisnis tertentu [33], [35]–[37]. Karena otonominya, mereka mengandung semua dependensi seperti: pustaka, lingkungan eksekusi – server web dan kontainer atau mesin virtual. Dengan demikian, microservices meningkatkan kemungkinan monetisasi bagian sistem, karena akses ke API microservice yang relevan dapat dikenakan biaya [38]. • Layanan adalah warga negara kelas satu – mereka mengekspos titik akhir layanan sebagai API dan mengabstraksi semua detail implementasinya. Struktur internal: logika implementasi, arsitektur, dan teknologi (termasuk bahasa pemrograman, basis data, dll.) sepenuhnya tersembunyi di balik API.

Perlu disebutkan bahwa paradigma komunikasi microservice berbeda secara signifikan dari pendekatan berbasis Arsitektur Berorientasi Layanan (SOA) [39] seperti Enterprise Service Bus (ESB), yang mencakup hal-hal yang canggih dan Fasilitas "berat" untuk perutean, penyaringan, dan transformasi pesan. Pendekatan microservice lebih menyukai: "endpoint cerdas dan saluran bodoh" [3]. Tidak ada standar untuk mekanisme komunikasi atau transportasi untuk microservice. Microservice berkomunikasi satu sama lain menggunakan protokol internet ringan yang terstandarisasi dengan baik, seperti HTTP dan REST [5], atau protokol perpesanan, seperti JMS atau AMQP.

Fitur paling menarik dari arsitektur microservice adalah dekomposisi aplikasi kompleks menjadi komponen yang lebih kecil yang lebih mudah dikembangkan, dikelola, dan dipelihara daripada aplikasi monolit tunggal [40]. Selama API publik tidak berubah, modifikasi internal suatu layanan lebih mudah, sederhana, dan murah dibandingkan dengan perubahan serupa pada model tradisional.

Mikroservis bersifat otonom dan berkomunikasi melalui protokol terbuka, oleh karena itu mikroservis dapat dikembangkan secara cukup independen dan bahkan dengan teknologi yang berbeda [5], [41]–[43].

Aplikasi berbasis microservice dapat diskalakan secara horizontal dengan baik, tidak hanya dalam arti teknis, tetapi juga menyangkut struktur organisasi tim pengembang, yang dapat dijaga agar lebih kecil dan lebih lincah [5], [22], [44], [45]. Upaya untuk mengintegrasikan opsi tersebut secara mulus ke dalam manajemen proses bisnis adaptif telah diterapkan dalam praktik bisnis [46]. Lebih jauh lagi, membagi aplikasi besar menjadi microservice individual memberikan tingkat kemandirian yang lebih tinggi kepada tim agile [7], [25], yang mendukung penskalaan metode agile. Setiap tim dapat mengerjakan microservice yang berbeda dan mengembangkan user story yang hanya memengaruhi microservice mereka [44].
Selama tim tidak mengubah kontrak antar layanan, keputusan dapat dibuat dalam tim layanan daripada di antara beberapa tim yang mengerjakan aplikasi monolitik besar [7]. Dengan demikian, adopsi arsitektur microservice menyiratkan pengurangan kebutuhan koordinasi antar tim, yang merupakan tantangan serius dalam pengembangan perangkat lunak skala besar [47]–[52].

Manfaat lain dari aplikasi microservice adalah arsitektur yang terhubung secara longgar membuatnya lebih tahan terhadap kesalahan [37] – kegagalan satu komponen tidak selalu mengakibatkan tidak tersedianya seluruh sistem, karena layanan yang berfungsi masih dapat memenuhi permintaan pengguna. Selain itu, dimungkinkan untuk mengidentifikasi fungsi bisnis yang penting dan menerapkan microservice yang sesuai dalam lingkungan yang lebih redundan.

Terlepas dari banyak keuntungannya, arsitektur microservice memiliki beberapa kekurangan dan kelemahan, terutama terkait dengan sifat terdistribusinya. Penyebaran, penskalaan, dan pemantauan sistem multi-layanan merupakan tugas yang lebih kompleks dibandingkan dengan aplikasi monolitik. Karena alasan ini, berbagai prosedur otomatisasi dalam pipeline continuous integration / continuous delivery (CI/CD) [53], pemantauan, dan autoscaling berbasis permintaan digunakan dalam pengembangan aplikasi tersebut [6], [34]. Untuk memanfaatkan sepenuhnya siklus pengembangan-ke-operasi yang singkat, pengujian siklus hidup juga harus diotomatisasi, yang merupakan tugas yang lebih menantang di lingkungan terdistribusi [42], [54]. Tantangan lain terletak pada desainnya.

Dalam hal fasilitas manajemen data, prinsip-prinsip arsitektur microservice menyatakan bahwa isolasi layanan maksimum lebih disukai.
Oleh karena itu, beberapa sistem basis data independen diperkenalkan ke dalam aplikasi terdistribusi, sehingga meningkatkan kompleksitas dan mengurangi kemudahan pengelolaan [34].

Dalam karya ini, kami berfokus pada kinerja aplikasi – IPC yang dibutuhkan antar komponen microservice menimbulkan overhead yang substansial jika dibandingkan dengan komunikasi intra-proses (panggilan fungsi dan pemanggilan metode) yang digunakan dalam aplikasi monolitik. IPC diimplementasikan sebagai layanan kernel sistem operasi. Dalam kebanyakan kasus, hal ini membutuhkan penyalinan data antara ruang pengguna dan kernel, dan karenanya mengurangi (dalam banyak kasus secara signifikan) kinerja aplikasi. Aplikasi yang tidak melayani lalu lintas pengguna yang signifikan akan menunjukkan penurunan throughput dan waktu respons ketika dimigrasikan dari arsitektur monolitik ke arsitektur terdistribusi. Hanya ketika permintaan pengguna meningkat, penskalaan yang tepat dari aplikasi berbasis microservice dapat mengimbangi overhead komunikasi – mempelajari fenomena ini, pada kenyataannya, adalah topik utama dari karya kami.

C. SKALA VERTIKAL DAN HORIZONTAL

Skalabilitas adalah sifat suatu sistem untuk menangani jumlah pekerjaan yang semakin besar dengan menambahkan sumber daya ke sistem [55]. Cara penambahan sumber daya menentukan pendekatan penskalaan mana yang diambil [8], [14], [56], [57] - *penskalaan vertikal* atau *penskalaan horizontal.* Yang pertama, juga dikenal sebagai *penskalaan naik,* mengacu pada penambahan lebih banyak sumber daya (CPU, memori, dan penyimpanan) ke mesin yang sudah ada. Ini adalah pendekatan yang lebih mudah, tetapi dibatasi oleh perangkat keras paling canggih yang tersedia di pasaran [56]. Sedangkan untuk Azure App Service, instance VM paling canggih yang tersedia hanya memiliki delapan core dan 32 GB RAM.
Selain itu, di luar konfigurasi sumber daya perangkat keras tertentu, biaya meningkat secara drastis. Perlu disebutkan bahwa di platform cloud, penskalaan vertikal memungkinkan penambahan atau penghapusan sumber daya virtual ke mesin virtual (VM) yang sedang berjalan [58], sehingga tidak menyebabkan downtime.

Sebaliknya, *penskalaan horizontal,* juga dikenal sebagai *penskalaan keluar,* mengacu pada penambahan lebih banyak mesin dan pendistribusian beban kerja. Hal ini lebih kompleks karena memiliki pengaruh pada arsitektur aplikasi, tetapi dapat menawarkan skala yang jauh melebihi skala yang mungkin dilakukan dengan penskalaan vertikal [56]. Penskalaan horizontal lebih umum pada aplikasi microservice [36], meskipun aplikasi monolitik juga dapat diskalakan keluar dengan menjalankan banyak instance di belakang load balancer [6]. Namun demikian, penskalaan keluar aplikasi monolitik mungkin tidak begitu efektif karena umumnya menawarkan banyak layanan - beberapa di antaranya lebih populer daripada yang lain. Untuk meningkatkan ketersediaan aplikasi monolitik, seluruh aplikasi perlu direplikasi. Hal ini menyebabkan penskalaan berlebihan pada layanan yang tidak populer, yang mengkonsumsi sumber daya server bahkan ketika layanan tersebut tidak aktif, dan akibatnya menghasilkan pemanfaatan sumber daya yang suboptimal [59].
Di sisi lain, untuk meningkatkan ketersediaan aplikasi microservice, hanya microservice yang sangat dibutuhkan dan mengonsumsi banyak sumber daya server yang akan mendapatkan lebih banyak instance [18].

III. KARYA TERKAIT

A. APLIKASI PEMBANDINGAN UNTUK MEMBANDINGKAN ARSITEKTUR MONOLITIK VERSUS ARSITEKTUR MIKROLAYANAN

Untuk membandingkan kinerja dan skalabilitas dari dua arsitektur yang diteliti, kita membutuhkan versi monolitik dan mikroservis dari aplikasi yang sama. Ketika melakukan pencarian menyeluruh di web, Aderaldo *dkk.* [23] hanya menemukan dua sistem seperti itu: *Acme Air* dan *MusicStore.* Tiga tahun kemudian, Francesco *dkk.* [15] melakukan studi pemetaan sistematis untuk mengkarakterisasi keadaan terkini dalam arsitektur dengan mikroservis. Setelah menyaring 103 studi primer, mereka hanya mengidentifikasi satu aplikasi benchmarking, yaitu *Acme Air.* Akibatnya, mereka menyerukan pengembangan aplikasi benchmarking open-source yang dapat digunakan untuk membandingkan arsitektur monolitik versus mikroservis. Selain itu, kami mengidentifikasi satu aplikasi benchmarking lagi, yaitu *JHipster.* Ketiga aplikasi tersebut akan dibahas secara singkat di bawah ini.

MusicStore awalnya dikembangkan oleh Microsoft untuk mendemonstrasikan komponen ASP.NET. Kemudian, MusicStore dipecah menjadi beberapa layanan independen oleh tim Steeltoe[1] untuk mengilustrasikan cara menggunakan pustaka sumber terbuka mereka yang ditujukan untuk mengembangkan aplikasi layanan mikro .NET berbasis cloud. Sayangnya, implementasi monolitik aslinya belum diperbarui sejak tahun 2018, sementara repositori GitHub-nya telah diarsipkan. Karena implementasi monolitik tersebut menggunakan teknologi usang, sedangkan implementasi berbasis microservice menggunakan pustaka pihak ketiga, *MusicStore* bukanlah aplikasi benchmark yang tepat.

Acme Air2 mensimulasikan situs web untuk perusahaan penerbangan fiktif. Ia tersedia tidak hanya dalam dua gaya arsitektur tetapi juga dalam dua bahasa yang berbeda (yaitu, Java EE dan Node.js). Selain itu, ia telah sering digunakan dan dibahas dalam literatur penelitian microservices [59], [60].
Sayangnya, *Acme Air* juga belum diperbarui sejak Agustus 2015, sementara Java telah berkembang pesat dalam kurun waktu tersebut. Oleh karena itu, aplikasi ini juga tidak dapat digunakan sebagai tolok ukur.

JHipster [41] adalah platform pengembangan yang digunakan untuk menghasilkan aplikasi web. Platform ini diimplementasikan dengan kerangka kerja Java Spring Boot dan Angular JS. Kode sumbernya tersedia secara publik di GitHub dalam versi monolitik dan microservice.3 Meskipun demikian, ketika versi Java dari aplikasi benchmark kami dikembangkan [61], kami tidak mengetahui *JHipster,* karena makalah yang mempopulerkannya diterbitkan kemudian.

B. EVALUASI KINERJA ARSITEKTUR MONOLITIK VERSUS ARSITEKTUR MIKROLAYANAN

Topik perbandingan kinerja dan biaya aplikasi mikroservis dan monolitik telah dibahas dalam literatur. Dalam [41] Al-Debagy dan Martinek membandingkan kinerja aplikasi yang dibangun baik secara monolitik

1https://github.com/SteeltoeOSS/Samples/tree/main/MusicStore
2https://github.com/acmeair/acmeair
3https://github.com/eugenp/tutorials/tree/master/jhipster

TABEL 1. Variabel independen dan dependen menurut percobaan.

	Independent variables					Dependent variables	
Deployment environment	**Architecture** {monolith, microservices}	**Service** {City, Route}	**Technology** {Java, .NET}	**VM type**	**#instances**	**Performance**	**Cost**
Local	✓	✓	✓			✓	
Azure Spring Cloud	✓	✓	[a]	✓[b]	✓[c]	✓	✓
Azure App Service	✓	✓	✓	✓[d]	✓[e]	✓	✓

[a] since Azure Spring Cloud did not support .NET, we tested only Java implementations; hence *technology* was not an independent variable in this experiment

[b] {1 vCPU 1GB RAM, 2 vCPU 2GB RAM, 3 vCPU 3GB RAM, 4 vCPU 6GB RAM} [c] {1, 2, 3, 6}

[d] {B1, S1, S2, S3, P3v2} [e] {1, 3, 6, 10, 15, 20}

dan bergaya microservice. Aplikasi ini dikembangkan dengan Spring Boot dan AngularJS dengan Apache JMeter yang digunakan sebagai platform pengujian. Pengujian dilakukan di lingkungan lokal.

Waktu respons dan throughput digunakan sebagai metrik kinerja. Dalam pengujian konkurensi, versi monolitik aplikasi menunjukkan kinerja yang lebih baik sebesar 6% dalam throughput dibandingkan dengan varian berbasis microservice, sedangkan dalam skenario pengujian beban, tidak ada perbedaan signifikan antara kedua pendekatan tersebut. Perlu dicatat bahwa dalam penelitian yang dikutip di atas, aplikasi berjalan di lingkungan lokal (yaitu, bukan cloud), di mana efek penskalaan vertikal maupun horizontal tidak dievaluasi.

Dalam [62] Garces *et al.* telah melakukan perbandingan biaya menjalankan aplikasi web dalam tiga varian arsitektur: menggunakan monolitik dan microservice yang dioperasikan klien di bawah cloud AWS EC2 serta microservice yang dioperasikan penyedia di bawah lingkungan cloud AWS Lambda. Para penulis menggunakan teknologi Java dengan framework Play dan Jax-RX serta Node.js.
Arsitektur monolitik digunakan sebagai dasar, yang menjadi acuan pengujian microservice selanjutnya. Ukuran yang digunakan adalah jumlah maksimum permintaan per menit yang didukung oleh arsitektur tertentu. Hasil pengujian menunjukkan bahwa microservice yang dioperasikan klien memang mengurangi biaya infrastruktur sebesar 13% dibandingkan dengan arsitektur monolitik standar, dan dalam kasus layanan yang dirancang khusus untuk penskalaan optimal di lingkungan cloud yang dioperasikan penyedia, biaya infrastruktur berkurang sebesar 77%.

Dalam [59] Ueda *et al.* menganalisis perilaku aplikasi yang diimplementasikan sebagai varian microservice dan monolitik untuk dua runtime bahasa populer – Node.js dan Java Enterprise Edition (EE) menggunakan rangkaian benchmark Acme Air dan Apache JMeter untuk pengumpulan data kinerja. Selain itu, pengujian dilakukan baik untuk proses asli maupun penyebaran kontainer Docker. Throughput dan siklus per instruksi (CPI) digunakan sebagai metrik kinerja; lingkungan pengujian setara dengan penyebaran cloud pribadi. Para penulis mengamati overhead yang signifikan dalam arsitektur microservice – pada konfigurasi perangkat keras yang sama, kinerja model microservice 79% lebih rendah dibandingkan dengan model monolitik. Model microservice menghabiskan waktu yang lebih signifikan di pustaka runtime untuk memproses satu permintaan klien daripada model monolitik, yaitu – 4,22x pada server aplikasi Node.js dan pada versi 2.69x pada server aplikasi Java EE.

Seperti yang ditunjukkan oleh ulasan singkat ini, hasil uji kinerja yang dilakukan dengan asumsi berbeda, dan di lingkungan yang sulit untuk dikaitkan satu sama lain memberikan hasil yang bertentangan. Beberapa menunjukkan peningkatan kinerja yang signifikan untuk arsitektur microservice, sementara yang lain untuk arsitektur monolitik.
Dalam karya ini, kami menyajikan lingkungan benchmark yang lebih komprehensif yang berfokus hanya pada penerapan cloud, dan mencakup berbagai varian penerapan dengan skala yang berbeda – informasi lebih lanjut akan dijelaskan di bagian selanjutnya.

IV. METODE

A. DESAIN PENELITIAN

Untuk menjawab pertanyaan penelitian, kami melakukan tiga eksperimen terkontrol. Setiap eksperimen dilakukan dalam lingkungan penerapan yang berbeda (lihat Tabel 1) dan menyelidiki pengaruh beberapa faktor (yaitu variabel independen), yang bervariasi di antara lingkungan penerapan, terhadap *kinerja* dan *biaya infrastruktur. Kinerja* dihitung sebagai jumlah permintaan yang diproses per detik.

Selanjutnya, *biaya infrastruktur* ditentukan oleh jumlah dan jenis instance mesin virtual yang digunakan untuk menyebarkan aplikasi.

Dalam hal lingkungan lokal kami, kami mempertimbangkan pengaruh *arsitektur* (monolit vs. layanan mikro), *layanan* (Kota vs. Rute), dan *teknologi* (Java vs. C# .NET).
Oleh karena itu, kami menggunakan desain faktorial 2 × 2 × 2. Dalam desain faktorial, setiap level dari satu faktor dikombinasikan dengan setiap level dari faktor lainnya untuk menghasilkan semua kemungkinan kombinasi. Setiap kombinasi kemudian menjadi suatu kondisi dalam eksperimen [63]. Perlu dicatat bahwa biaya infrastruktur sama untuk semua percobaan di lingkungan lokal karena konfigurasi perangkat keras yang sama digunakan, sehingga kami tidak menghitungnya.

Jika berbicara tentang kedua eksperimen cloud tersebut, ada dua faktor lagi yang terlibat:

- *Tipe VM* - seperti pada Azure Spring Cloud, ini menentukan jumlah vCPU dan jumlah memori untuk satu instance VM, sedangkan dalam kasus Azure App Service, ini mengacu pada deskripsi umum dari sebuah VM;
- *#instances* - menunjukkan jumlah instance VM jenis.

Karena semua kombinasi level untuk semua faktor akan

Karena membutuhkan sejumlah besar percobaan dan sumber daya yang signifikan, desain faktorial penuh tidak praktis dalam kasus tersebut. Selain itu, tidak semua kemungkinan kombinasi level di semua faktor tersedia atau menarik.

Sebagai contoh, untuk mesin B1 di Azure App Service, jumlah instance maksimum adalah tiga, sedangkan Azure Spring Cloud hanya mendukung aplikasi Java Spring Boot.

Dengan demikian, kami menggunakan desain faktorial fraksional yang terdiri dari subset percobaan yang dipilih secara memadai dari desain faktorial penuh. Semua skenario, serta variabel independen dan dependen, dibahas secara rinci dalam subbagian berikut.

B. OBJEK EKSPERIMENTAL

Karena tidak ada aplikasi benchmarking yang ada yang cocok untuk membandingkan kinerja arsitektur yang diteliti secara adil, kami mengembangkan aplikasi baru. Aplikasi benchmarking kami diimplementasikan dalam empat versi yang secara fungsional identik, mencakup tidak hanya dua gaya arsitektur yang berbeda (monolit dan layanan mikro) tetapi juga dua teknologi terkemuka yang ditujukan untuk pengembangan perangkat lunak sisi server:

- Java – diimplementasikan dalam Java 84 dengan framework Spring Boot 2.3.0,
- .NET – diimplementasikan dalam C# versi 8 dengan ASP.NET Kerangka kerja inti 3.1.

Alasan di balik pemilihan di atas adalah sebagai berikut: Spring Boot adalah kerangka kerja Java yang paling dominan saat ini [65], dan jelas merupakan yang terdokumentasi dengan baik dan paling umum dalam penerapan cloud [8]; ASP.NET Core adalah versi baru dari lingkungan pemrograman ASP.NET yang populer dan matang, basis kodenya dilisensikan sebagai open-source dan tersedia di GitHub. Mirip dengan Spring, ia terdokumentasi dengan baik dan memiliki komunitas pengembang yang besar yang mendukungnya. Ini juga merupakan pilihan pertama dan alami untuk penerapan aplikasi microservice di cloud Azure.

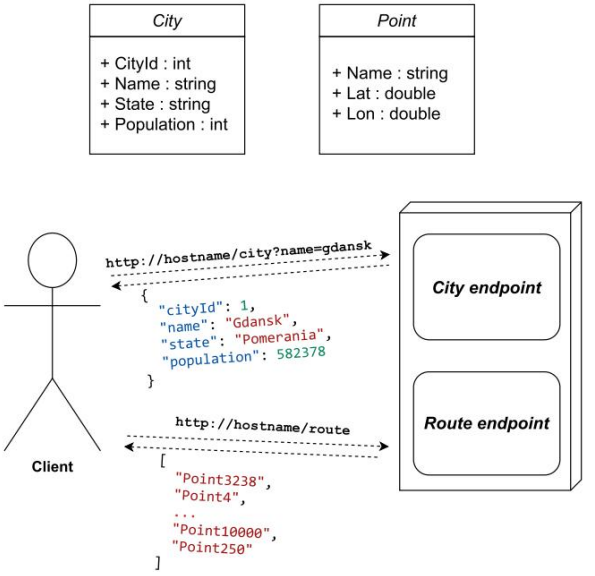
Semua versi aplikasi kami mengekspos dua endpoint REST yang mengembalikan objek JSON yang telah diserialisasi seperti yang ditunjukkan pada Gambar 1. Endpoint REST tersebut sesuai dengan dua layanan:

- Layanan *kota* – mensimulasikan kueri objek tunggal sederhana, input berisi string nama kota, sedangkan respons berisi data kota (id, nama, negara bagian, dan populasi) • Layanan *rute* –

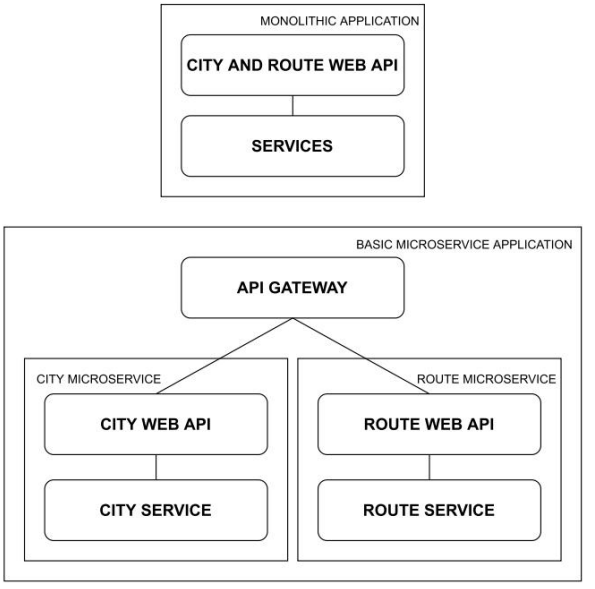
mensimulasikan kueri yang membutuhkan banyak komputasi, output berisi jalur (serangkaian titik yang terurut) yang merupakan rute terpendek antara 10.000 titik yang dipilih secara acak, setiap kali rute dihitung oleh algoritma heuristik dari kelas pedagang keliling.

Skema dasar versi monolitik dan mikroservis ditunjukkan pada Gambar 2 – dalam kasus monolitik, API REST mengarahkan kueri klien langsung ke logika bisnis aplikasi. Dalam kasus versi mikroservis, sebuah API

4. Survei tahunan mengenai keadaan ekosistem JVM yang dilakukan pada tahun 2020 [64], menunjukkan bahwa 64% pengembang masih menggunakan Java 8 di lingkungan produksi.



GAMBAR 1. Layanan kota dan rute - data respons dan skema komunikasi dasar.



GAMBAR 2. Arsitektur logika monolitik dan mikroservis.

Gateway digunakan untuk mengarahkan permintaan ke layanan yang bertanggung jawab, yang merupakan solusi umum untuk jenis arsitektur ini.

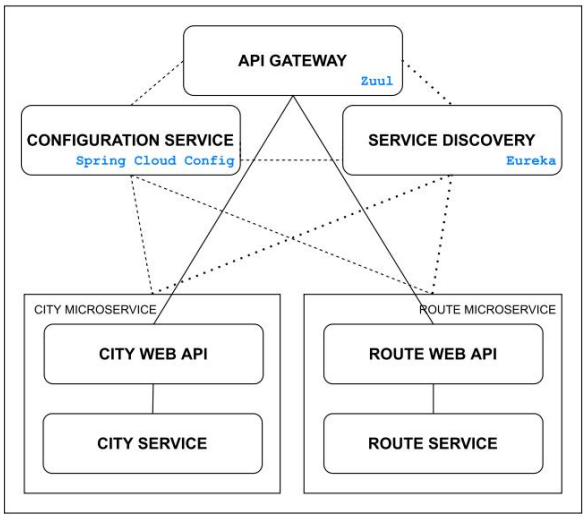
Aplikasi ini diuji di lingkungan lokal (yaitu, bukan di cloud) sesuai dengan standar yang telah ditetapkan dan di cloud Azure.

Dalam kasus aplikasi versi Java, kami telah menguji dua varian penyebaran: Azure Spring Cloud dan Azure App Service. Dalam kasus versi monolitik, tidak ada perbedaan antara versi lokal dan versi cloud.

Dalam kasus varian microservice, komponen tambahan ditambahkan untuk memungkinkan penskalaan horizontal, yaitu – aplikasi diperluas untuk menggunakan kerangka kerja Sprint Cloud, yang

Termasuk: load balancer Zuul, Spring Cloud Config, dan Eureka5 – sebuah registry yang menyediakan penemuan layanan.

Dalam kasus Java, perbedaan antara penerapan Azure Spring Cloud dan Azure App Service adalah bahwa yang terakhir menawarkan komponen gateway aplikasi "bawaan" yang merupakan bagian dari kerangka kerja. Namun, dalam kasus Spring Cloud, gateway harus diterapkan secara manual sebagai layanan yang disediakan oleh pengguna. Arsitektur versi microservice dari aplikasi uji kami di bawah Azure Spring Cloud ditunjukkan pada Gambar 3.



GAMBAR 3. Aplikasi microservices yang diimplementasikan ke Azure Spring Cloud.

C. PERLENGKAPAN PERCOBAAN

1) LINGKUNGAN LOKAL

Pengujian lokal dilakukan pada mesin PC yang menjalankan Microsoft Windows versi 10 Enterprise dengan parameter perangkat keras berikut: Intel(R) Core(TM) i7-9850H CPU 2.60GHz, enam inti fisik, 12 inti logis, dan RAM 32 GB (Dell Precision 7540). Selama pengujian lokal, hanya satu instance dari setiap layanan tanpa penskalaan yang digunakan.

2) LINGKUNGAN AZURE SPRING CLOUD

Konfigurasi lingkungan pengujian untuk Azure Spring Cloud ditunjukkan pada Tabel 2. Sayangnya, Azure Spring Cloud tidak menawarkan penskalaan sumber daya yang terperinci. Pada Tingkat Standar, jika ukuran sumber daya yang dikonsumsi tidak melebihi 16 core dan 32 GB RAM, Azure Spring Cloud mengenakan biaya tetap per jam. Artinya, menurut daftar harga yang tersedia untuk Wilayah US East pada September 2020, biaya infrastruktur untuk menjalankan aplikasi kami di setiap skenario penyebaran ditetapkan sebesar 1,65 USD per jam (39,6 USD per 24 jam). Dengan demikian, untuk membandingkan efektivitas biaya dari setiap skenario penyebaran, kami menghitung "Biaya Efektif 24 Jam" dengan asumsi harga sumber daya yang terperinci sebagai berikut: 0,08 USD per core per jam dan 0,01 USD per 1 GB RAM per jam. Ini

5https://github.com/Netflix/eureka

TABEL 2. Skenario penerapan untuk Azure Spring Cloud.

Architecture	# cores	RAM [GB]	# instances	24H Effective Cost [USD]
monolithic	1	1	1	2.16
monolithic	3	3	1	6.48
monolithic	4	6	1	9.12
microservice	1	1	1	4.32
microservice	1	1	3	8.64
microservice	2	2	2	10.8
microservice	3	3	1	8.64
microservice	1	1	6	15.12

Perhitungan ini memperhitungkan harga kelebihan memori penyedia (0,00825 USD per GB-jam) dan harga kelebihan vCPU (0,0783 USD per vCPU-jam). Perlu dicatat bahwa "Biaya Efektif 24 Jam" mengacu pada total biaya, termasuk biaya layanan API Gateway yang dikonfigurasi dengan 1 inti CPU dan 1 GB RAM.

3) AZURE APP SERVICE

Dalam kasus Azure App Service, biaya didasarkan pada ukuran dan jumlah instance sesuai dengan *tingkatan harga* yang ditunjukkan pada Tabel 3. (Harap dicatat bahwa nilai dari kolom "Identifier" akan digunakan lebih lanjut dalam pekerjaan ini untuk menentukan konfigurasi sumber daya perangkat keras).

TABEL 3. Tingkat layanan aplikasi Azure.

Name	Identifier	# cores	RAM [GB]	Price [USD/h]	Max # instances
B1	B1:1.75	1	1.75	0.08	3
S1	S1:1.75	1	1.75	0.10	10
S2	S2:3.5	2	3.5	0.20	10
S3	S4:7	4	7	0.40	10
P3v2	P4:14	4	14	0.80	30

Dalam hal total biaya varian microservice, kita juga harus memperhitungkan sumber daya yang dibutuhkan oleh application gateway, yaitu 0,025 USD per jam kerja (application gateway hanya membutuhkan satu core dan 1 GB RAM).

Berbagai varian penerapan terkait arsitektur, teknologi, dan sumber daya yang kami uji tercantum dalam Tabel 4. "Biaya 24 Jam" adalah biaya pengoperasian aplikasi secara terus menerus selama 24 jam dengan semua komponen yang dibutuhkan.

D. PROSEDUR

Kami telah menggunakan Apache JMeter6 sebagai alat pengujian. JMeter adalah alat perangkat lunak sumber terbuka populer yang dapat mensimulasikan beban pada server, sekelompok server, jaringan, atau objek. Kami memilihnya karena fleksibilitasnya – memungkinkan kami untuk mengkonfigurasi konfigurasi beban secara tepat [66], yaitu: • jumlah total permintaan, • jumlah thread yang akan digunakan untuk menjalankan pengujian, yang mensimulasikan jumlah pengguna yang mengakses aplikasi secara bersamaan, • metode verifikasi hasil.

6https://jmeter.apache.org

TABEL 4. Skenario penerapan untuk layanan aplikasi Azure.

Architecture	Technology	Pricing tier	# instances	24H Cost
monolithic	.NET	B1	1	1.8
monolithic	.NET	S2	1	4.8
monolithic	.NET	S3	1	9.6
monolithic	.NET	P3v2	1	19.2
monolithic	Java	B1	1	1.8
monolithic	Java	S2	1	4.8
monolithic	Java	S3	1	9.6
monolithic	Java	P3v2	1	19.2
microservice	.NET	B1	1	2.4
microservice	.NET	B1	3	6
microservice	.NET	S1	6	15
microservice	.NET	S1	10	24.6
microservice	.NET	S2	1	5.4
microservice	.NET	S2	3	15
microservice	.NET	S2	6	29.4
microservice	.NET	S2	10	48.6
microservice	.NET	S3	1	10.2
microservice	.NET	S3	3	29.4
microservice	.NET	S3	6	58.2
microservice	.NET	S3	10	96.6
microservice	.NET	P3v2	1	19.8
microservice	.NET	P3v2	3	58.2
microservice	.NET	P3v2	6	115.8
microservice	.NET	P3v2	10	192.6
microservice	.NET	P3v2	15	288.6
microservice	.NET	P3v2	20	384.6
microservice	Java	B1	1	2.4
microservice	Java	B1	3	6
microservice	Java	S1	6	15
microservice	Java	S1	10	24.6
microservice	Java	S2	1	5.4
microservice	Java	S2	3	15
microservice	Java	S2	6	29.4
microservice	Java	S2	10	48.6
microservice	Java	S3	1	10.2
microservice	Java	S3	3	29.4
microservice	Java	S3	6	58.2
microservice	Java	S3	10	96.6
microservice	Java	P3v2	1	19.8
microservice	Java	P3v2	3	58.2
microservice	Java	P3v2	6	115.8
microservice	Java	P3v2	10	192.6
microservice	Java	P3v2	15	288.6
microservice	Java	P3v2	20	384.6

Sebagai ukuran *kinerja* , kami menggunakan throughput, yang dihitung oleh JMeter sebagai jumlah permintaan yang diproses oleh server, dibagi dengan total waktu dalam detik untuk memproses permintaan tersebut. Waktu diukur dari awal permintaan pertama hingga akhir permintaan terakhir. Ini termasuk interval apa pun di antara permintaan, karena dimaksudkan untuk mewakili beban pada server.

Throughput adalah ukuran yang paling umum digunakan dalam karya serupa – lihat misalnya [42], [59]. Ukuran lain termasuk waktu respons dan siklus per instruksi (CPI), tetapi throughput paling cocok jika kita juga ingin membandingkan biaya keseluruhan menjalankan aplikasi dalam teknologi dan konfigurasi tertentu.

Untuk mengumpulkan data kinerja, kami menggunakan prosedur pengujian berikut: layanan *Kota* dipanggil 1000 kali, layanan *Rute* dipanggil 100 kali, jumlah thread diatur menjadi 10 untuk kedua skenario. Setiap pengujian diulang.

20 kali. Karena lingkungan cloud tidak selalu menjamin kinerja yang stabil karena banyak faktor yang berkaitan dengan stabilitas jaringan, ketersediaan server virtual, dll. – untuk mengkompensasi berbagai variasi yang tidak terduga, kami mengulangi setiap rangkaian pengujian sebanyak lima kali, menghitung median dari hasilnya, dan akhirnya, kami memilih pengujian dengan median yang merupakan median dari median yang diperoleh dari semua pengujian. Validitas pendekatan tersebut dibahas dalam karya "Software Microbenchmarking in the Cloud" oleh Laaber *et al.* [67].

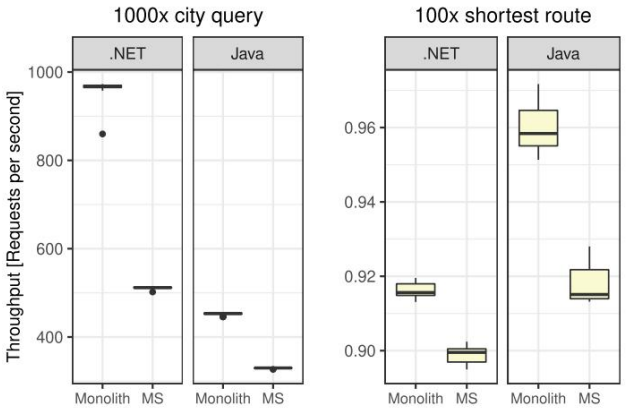
Perlu dicatat bahwa ketika menguji kinerja aplikasi berbasis microservice, hanya satu layanan yang berjalan. Ini adalah pendekatan umum dalam hal penskalaan horizontal microservice – setiap microservice populer mendapatkan mesin virtualnya sendiri sesuai dengan beban yang terkait [18], [68].
Namun demikian, pendekatan ini sedikit menghambat aplikasi monolitik, di mana kedua layanan berjalan bersamaan meskipun salah satunya sedang tidak aktif.

V. HASIL

Pada bab ini, kami menyajikan dan menganalisis hasil pengujian kinerja dan skalabilitas aplikasi.

A. LINGKUNGAN LOKAL

Pengujian yang dilakukan di lingkungan lokal (non-cloud) memungkinkan kami untuk menetapkan tolok ukur kinerja. Pada Gambar 4, kami menyajikan throughput untuk aplikasi monolitik dan microservice yang ditulis dalam .NET dan Java, baik untuk layanan *Kota* maupun *Rute* .



GAMBAR 4. Throughput di lingkungan lokal, layanan kota (kiri), layanan rute (kanan). Selanjutnya, MS merujuk pada layanan mikro.

Dalam aplikasi microservice, permintaan diteruskan antara API Gateway dan layanan backend, yang menimbulkan overhead komunikasi yang menurunkan kinerja. Sedangkan untuk layanan *Route* yang membutuhkan banyak komputasi , overhead ini relatif kecil dibandingkan dengan waktu eksekusi layanan. Oleh karena itu, throughput hanya sedikit lebih tinggi untuk aplikasi monolitik. Sebaliknya, dalam kasus layanan *City* yang tidak membutuhkan banyak komputasi , waktu CPU yang dibutuhkan untuk mengeksekusi layanan mungkin memiliki orde besaran yang sama dengan waktu CPU untuk mengirim dan menerima data dari layanan backend. Dengan demikian, sistem monolitik menangani

rata-rata terdapat lebih dari 2 kali lipat permintaan pada versi .NET, dan 1,37 kali lipat permintaan pada versi Java.

Dengan membandingkan .NET dan Java, kita dapat menyimpulkan bahwa dalam kasus layanan *Kota* yang tidak membutuhkan komputasi intensif , .NET lebih efisien dalam menangani permintaan komunikasi daripada Java. Hal ini dapat dikaitkan dengan tingkat optimasi kinerja yang tinggi dalam pustaka .NET Core, terutama terkait penanganan permintaan jaringan7 dan serialisasi data JSON.8,9 Di sisi lain, dengan layanan *Rute ,* implementasi Java menunjukkan throughput yang lebih baik daripada .NET baik untuk aplikasi monolitik maupun microservice – masing-masing sebesar 5% dan 1,5%.

Beberapa wawasan tentang pemanfaatan CPU memberikan jawaban mengenai throughput Java yang lebih baik dalam kasus ini – penggunaan CPU pada varian .NET secara nyata lebih rendah; oleh karena itu kita dapat menyimpulkan bahwa Java, dalam kasus aplikasi yang membutuhkan komputasi intensif, lebih agresif dalam mengalokasikan sumber daya ini ke aplikasi (pemanfaatan RAM serupa untuk kedua kasus).

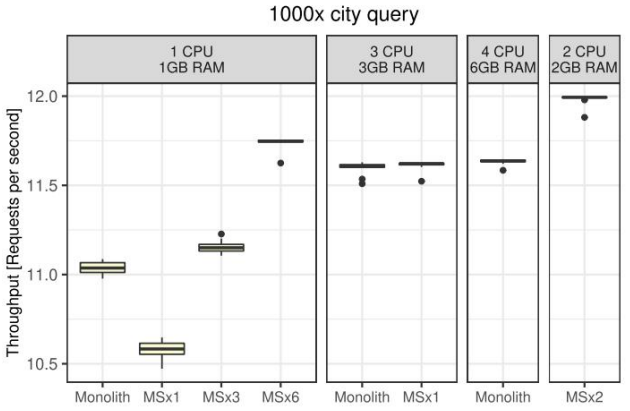
B. LINGKUNGAN AWAN MUSIM SEMI AZURE

Karena Spring adalah framework khusus Java, hanya versi Java dari aplikasi yang diuji di lingkungan Azure Spring Cloud. Kami menguji setiap layanan di bawah empat varian alokasi sumber daya CPU/RAM yang berbeda.

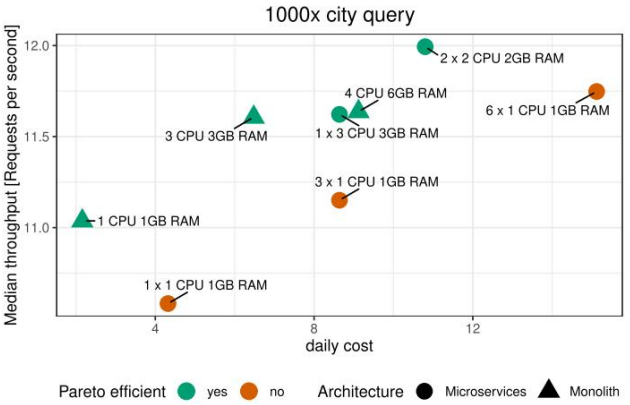
Throughput tertinggi pada layanan City 5 dicapai oleh microservice yang dikonfigurasi pada dua instance mesin dengan 2 CPU & 2 GB RAM; selanjutnya adalah konfigurasi enam instance dengan 1 CPU & 1 GB RAM, sedangkan performa terburuk adalah satu instance microservice yang dikonfigurasi dengan 1 CPU & 1 GB RAM. Kita dapat melihat bahwa dalam kasus aplikasi ringan ini, kita dapat mencapai peningkatan throughput ketika jumlah instance yang lebih tinggi (enam dalam kasus ini) digunakan – throughput MSx3 sebanding dengan throughput varian monolitik.

Kita juga dapat menyatakan bahwa penambahan CPU dan RAM pada varian monolitik hanya meningkatkan kinerja hingga batas tertentu – bandingkan 4 CPU & 6 GB RAM dengan 4 CPU & 6 GB RAM. Terakhir, throughput terbaik dicapai dalam konfigurasi yang diskalakan secara vertikal dan horizontal – MSx2 2 CPU & 2 GB RAM.

Biaya harian aplikasi vs. kinerja ditunjukkan pada Gambar 6, di mana harga dihitung berdasarkan data yang disajikan pada Tabel 2. Di sini, dan dalam kasus serupa selanjutnya, kami selalu menggunakan median throughput yang diperoleh untuk perbandingan biaya. Pada gambar ini dan gambar selanjutnya, konfigurasi yang didominasi Pareto yang ditandai dengan warna merah berkaitan dengan kasus di mana kinerja yang sama atau lebih baik dapat dicapai dengan konfigurasi yang lebih murah – efisien Pareto. Dengan membandingkan biaya, kita dapat menyimpulkan bahwa di lingkungan Spring Cloud, penskalaan horizontal mesin berkinerja rendah (1 CPU & 1 GB RAM) selalu mengarah pada konfigurasi yang didominasi Pareto.



GAMBAR 5. Throughput di lingkungan Azure Spring Cloud—aplikasi kota.



GAMBAR 6. Throughput dan biaya di lingkungan Azure Spring Cloud—aplikasi kota.

Pengujian rute memberikan hasil yang berbeda – lihat Gambar 7 – konfigurasi dengan kinerja terbaik adalah enam instance mesin dengan 1 CPU & 1 GB RAM, hasil throughput yang baik juga diperoleh oleh dua instance dengan 2 CPU & 2 GB RAM dan versi monolitik. Membandingkan hasil ini dengan hasil sebelumnya, kita dapat menyimpulkan bahwa dalam kasus aplikasi yang intensif komputasi, hasil positif dari penskalaan horizontal lebih terlihat – throughput meningkat hampir secara linier dengan jumlah instance. Selain itu, tidak ada perbedaan kinerja yang substansial antara aplikasi monolitik dan microservice dengan sumber daya CPU yang sama.

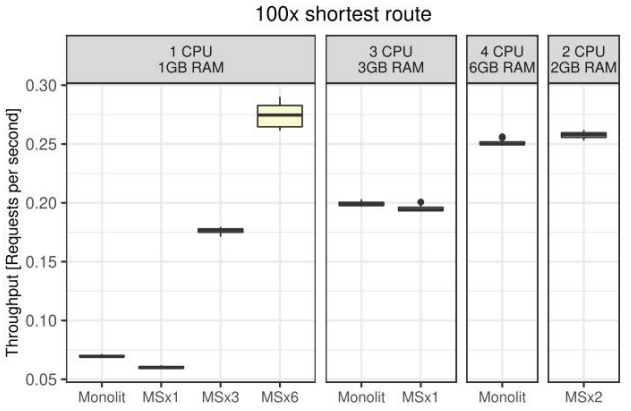
Rasio biaya harian terhadap kinerja layanan *rute* ditunjukkan pada Gambar 8. Dalam kasus ini, tiga konfigurasi microservice didominasi Pareto, sedangkan versi monolitik yang sesuai dengan sumber dayanya efisien Pareto. Perlu dicatat bahwa MSx6 dengan 1 CPU & 1 GB RAM ternyata merupakan konfigurasi yang paling kuat, tetapi MSx2 dengan 2 CPU & 2 GB RAM dan versi monolitik dengan 4 CPU & 6 GB RAM hanya memberikan kinerja yang sedikit lebih rendah dengan biaya yang jauh lebih rendah.

C. LINGKUNGAN LAYANAN APLIKASI AZURE

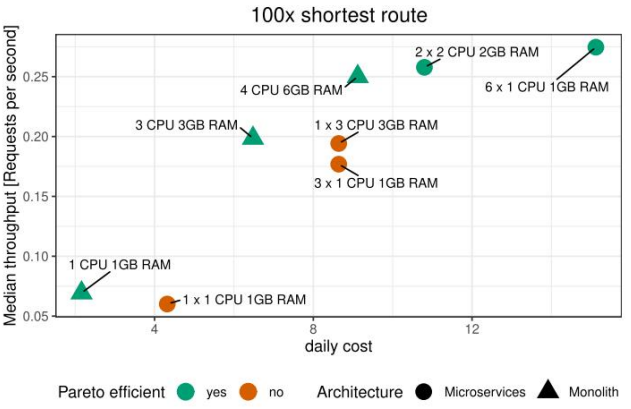
Gambar 9 mengilustrasikan hasil throughput untuk layanan *City* di lingkungan Azure App Service. Seperti pada sebelumnya

7https://devblogs.microsoft.com/dotnet/performance-improvements-in-inti jaringan

8https://devblogs.microsoft.com/dotnet/try-the-new-system-text-json-apis 9https://www.techempower.com/blog/2019/07/09/framework-tolok ukur putaran ke-18



GAMBAR 7. Throughput di lingkungan Azure Spring Cloud—layanan rute.



GAMBAR 8. Throughput dan biaya di lingkungan Azure Spring Cloud —layanan rute.

Bagian – monolit dan MSx1, MSx2, . . . , MSx20 berkaitan dengan jumlah instance microservice mulai dari 1 hingga 20.
Hasilnya dikelompokkan berdasarkan konfigurasi mesin virtual (yaitu, jumlah CPU dan ukuran RAM) yang ditunjukkan sebagai label – lihat tabel 4. Bentuk penyajian ini memungkinkan kita untuk menganalisis efek penskalaan horizontal.

Informasi yang sama disajikan pada Gambar 10 untuk menganalisis efek penskalaan vertikal, sebagai fungsi dari peningkatan daya komputasi mesin virtual dan dikelompokkan berdasarkan jumlah instance microservice. Kumpulan data lengkap dan gambar box plot dari throughput yang diukur di bawah Azure App Service tersedia di repositori GitHub kami.10 Konfigurasi berkinerja terbaik dalam uji *Kota*

adalah 6 × S4:7 yang ditulis dalam .NET. Throughput yang sangat mirip diperoleh oleh 3 × S4:7 juga di bawah .NET. Konfigurasi terbaik untuk aplikasi Java adalah 6 × S4:7 – peringkat kelima. Aplikasi Java yang berkinerja terburuk berjalan pada mesin B1:1.75 dan S1:1.75. Dalam sebagian besar konfigurasi, aplikasi .NET berkinerja lebih baik daripada aplikasi Java. Hal ini konsisten dengan hasil pengujian lokal – di sini, juga Java

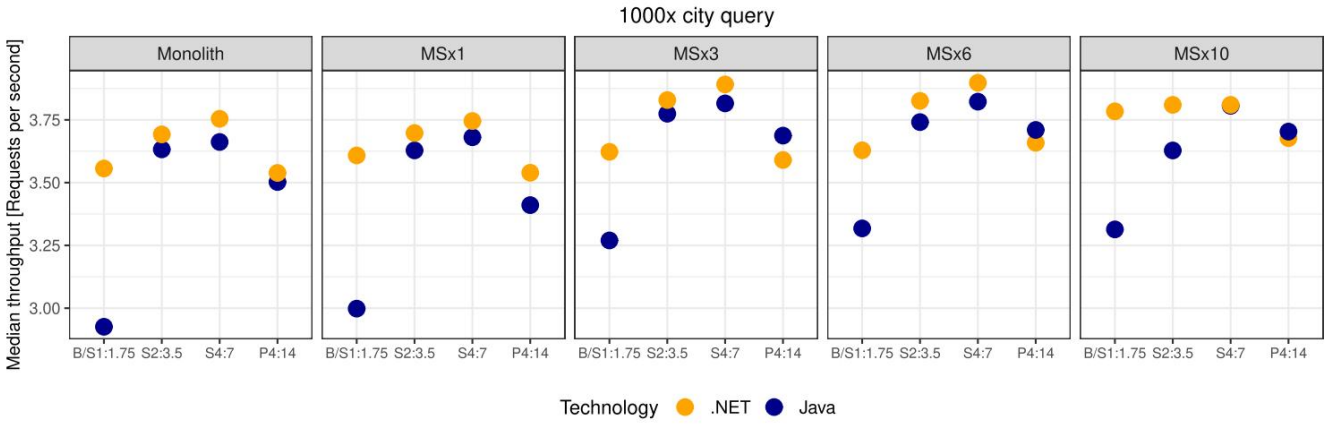
10https://github.com/przybylek/Monolith-vs-Microservices

Aplikasi berkinerja paling buruk dalam konfigurasi daya rendah, dan serupa atau lebih baik daripada .NET di lingkungan yang diskalakan secara vertikal seperti P4:14. Hasil ini konsisten dengan pengamatan sebelumnya bahwa aplikasi Java memiliki overhead sumber daya yang signifikan dan membutuhkan daya komputasi yang lebih besar.

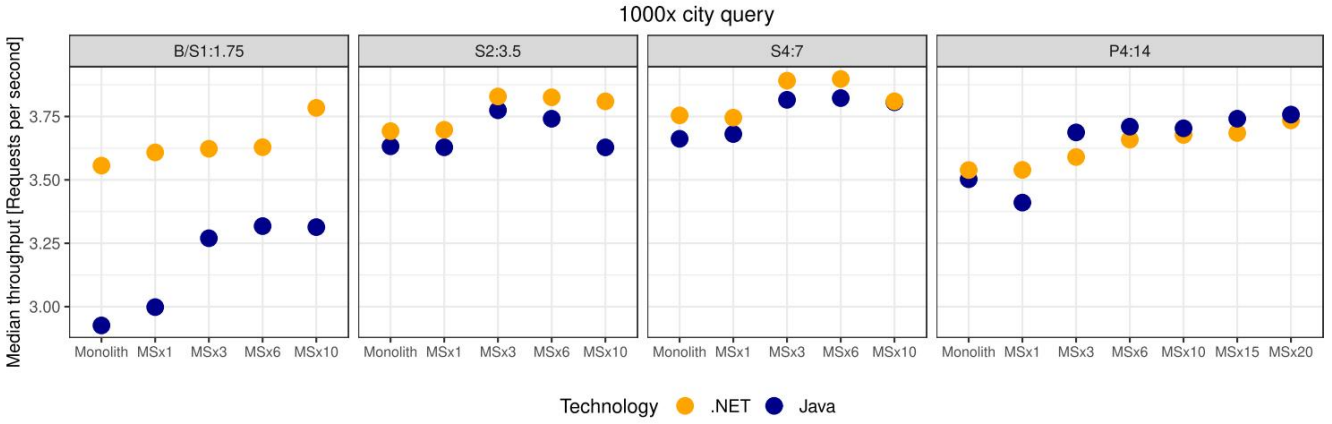
Hasil yang mengejutkan adalah bahwa pada konfigurasi daya tertinggi – P4:14, baik aplikasi Java maupun .NET menunjukkan penurunan kinerja dibandingkan dengan S4:7 meskipun P4:14 berasal dari Paket Layanan Premium v2, yang menyediakan prosesor lebih cepat dan penyimpanan SSD dibandingkan dengan VM yang ditawarkan dalam Paket Layanan Standar. Hasil kami menunjukkan bahwa jenis CPU yang digunakan dalam konfigurasi P4:14 kurang cocok untuk melayani sejumlah besar permintaan singkat dan lalu lintas jaringan yang padat.

Untuk mengilustrasikan efek penskalaan, kami telah menyiapkan dua plot tambahan – Gambar 11 menunjukkan bagaimana median throughput berubah dalam % relatif terhadap konfigurasi sebelumnya – ketika jumlah instance meningkat – dari 1 menjadi 3, dari 3 menjadi 6, . . . , dan akhirnya dari 15 menjadi 20; secara berturut-turut. Gambar 12 menunjukkan bagaimana median throughput berubah (juga dalam %) ketika konfigurasi mesin virtual tunggal ditingkatkan – dari: B/S1:1,75 menjadi S2:3,5, dari S2:3,5 menjadi S4:7, dan dari S4:7 menjadi P4:14. Dari Gambar 11 kita dapat menyimpulkan bahwa penskalaan horizontal aplikasi permintaan sederhana & singkat paling bermanfaat ketika jumlah instance ditingkatkan secara moderat, di sini: dari 1 menjadi 3 dan dari 3 menjadi 6. Peningkatan lebih lanjut jumlah instance tidak terlalu bermanfaat, dan dalam setengah kasus, hal itu mengakibatkan penurunan kinerja (.NET S2:3.5, Java S2:3.5, .NET S4:7, Java S4:7, Java P4:14, Java B/S1:1.75 dengan sepuluh instance) – ini disebabkan oleh overhead komunikasi karena penyeimbangan beban dan kebutuhan penerusan permintaan. Kesimpulan pertama yang jelas dari Gambar 12 yang berkaitan dengan penskalaan vertikal adalah bahwa perubahannya lebih signifikan jika dibandingkan dengan penskalaan horizontal – bandingkan peningkatan throughput sebesar 24% dalam kasus arsitektur Java Monolith dan MSx1 yang ditingkatkan ke konfigurasi S2:3.5 dengan peningkatan maksimum sebesar 9% dalam kasus arsitektur yang sama yang diskalakan secara horizontal dari 1 menjadi 3 instance. Selain itu, kita tidak mengamati peningkatan kinerja yang signifikan dengan peningkatan konfigurasi lebih lanjut. Namun, kita harus berhati-hati dalam menafsirkan hasil ini – peningkatan kinerja yang signifikan, terutama untuk arsitektur Java, disebabkan oleh efek "penskalaan kanan" – konfigurasi dasar B/S1:1.75 tidak sepenuhnya memenuhi persyaratan aplikasi, terutama di bawah lingkungan runtime Java yang lebih berat. Terakhir, kita juga harus menyebutkan dampak negatif dari perpindahan ke konfigurasi P4:14 yang disebabkan oleh perubahan arsitektur CPU yang telah dibahas sebelumnya dalam Paket Layanan Premium v2.

Biaya harian aplikasi vs. kinerja ditunjukkan pada Gambar 13. Seperti pada kasus sebelumnya, penetapan harga dihitung berdasarkan data yang disajikan pada Tabel 2 dan median throughput yang diperoleh digunakan. Dengan membandingkan biaya menjalankan layanan *Kota* dalam konfigurasi yang berbeda, kita dapat menyimpulkan bahwa varian konfigurasi Pareto efisien untuk .NET adalah: monolitik B1:1,75, S2:3,5, S4:7 dan berbasis microservice:



GAMBAR 9. **Throughput vs. penskalaan horizontal di lingkungan layanan aplikasi Azure—layanan kota.**



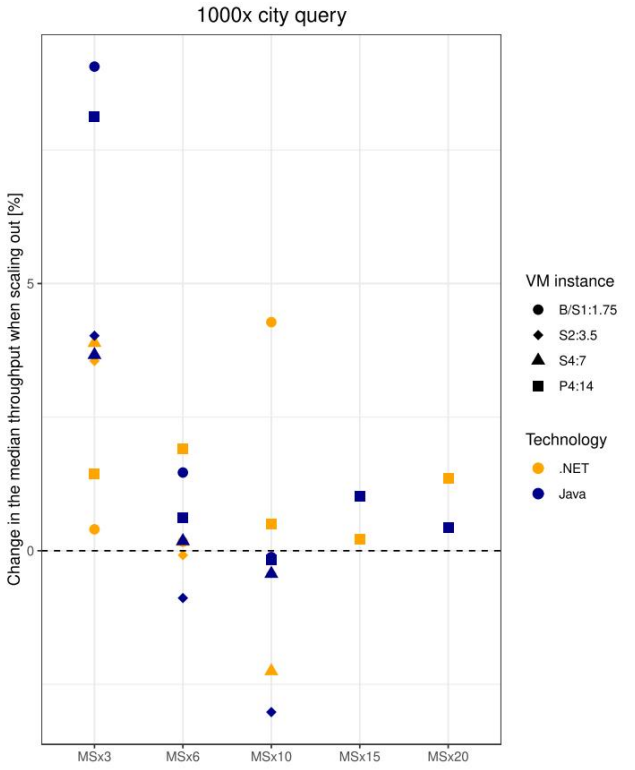
GAMBAR 10. **Throughput vs. penskalaan vertikal di lingkungan layanan aplikasi Azure—layanan kota.**

B1:1.75, 1 x S2:3.5, 3 x S2:3.5, 3 x S4:7 dan 6 x S4:7. Dalam kasus Java, serangkaian konfigurasi yang hampir identik dengan tambahan 1 x S4:7 adalah efisien Pareto.

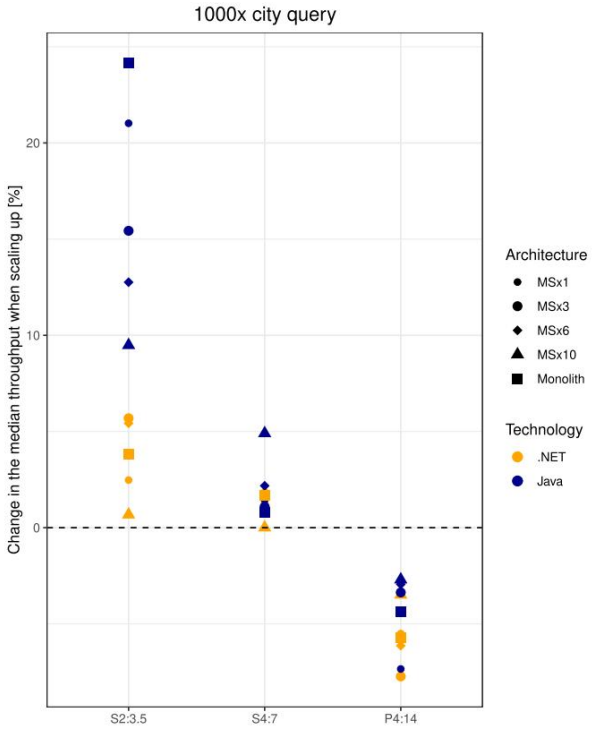
Sekarang kita akan fokus pada pengujian layanan *Rute* – lihat Gambar 14 yang menyajikan median throughput sebagai fungsi dari jumlah instance yang dikelompokkan berdasarkan konfigurasi instance (skala horizontal) dan Gambar 15 yang menyajikan data yang sama tetapi dikelompokkan berdasarkan jumlah instance (skala vertikal).
Java menunjukkan performa yang lebih baik di sini – mencapai hasil yang lebih baik di semua kasus kecuali satu. Efek dari penskalaan horizontal dan vertikal terlihat jelas – semakin kuat konfigurasi dan semakin besar jumlah instance microservice, semakin baik throughput-nya. Performa terbaik tercatat untuk konfigurasi Java 10 x P4:14. Hasil ini konsisten dengan hasil yang diperoleh dalam konfigurasi lokal Route *di* mana Java juga merupakan pilihan yang lebih baik untuk layanan yang membutuhkan komputasi intensif ini. Kita juga dapat mengamati tren yang menarik: untuk konfigurasi daya rendah, throughput serupa untuk varian .NET dan Java, tetapi seiring peningkatan daya mesin virtual, varian Java menunjukkan peningkatan performa yang besar.

.NET – bandingkan hasil untuk S4:7 dan P4:14 pada Gambar 14. Java juga menunjukkan penskalaan vertikal yang lebih baik – lihat hasil untuk P4:14 pada Gambar 15.

Mirip dengan layanan *Kota* dalam kasus layanan *Rute* , kami juga memvisualisasikan efek penskalaan pada dua plot tambahan – Gambar 16 menunjukkan bagaimana median throughput berubah dalam % relatif terhadap nilai sebelumnya ketika jumlah instance meningkat; Gambar 17 menunjukkan bagaimana median throughput berubah (juga dalam %) ketika konfigurasi mesin virtual tunggal ditingkatkan. Nilai pada sumbu x identik, seperti pada layanan *Kota* . Peningkatan throughput terbesar diukur ketika jumlah instance ditingkatkan dari 1 menjadi 3 dan dari 3 menjadi 6 – ini konsisten dengan hasil sebelumnya. Skala peningkatan jauh lebih menonjol pada layanan *Rute* . Dalam lima kasus (baik .NET dan Java), kinerja meningkat lebih dari 100% relatif terhadap konfigurasi sebelumnya. Peningkatan penskalaan paling signifikan yang diamati adalah 148% (dibandingkan dengan peningkatan maksimum 9% pada layanan *Kota).* Sekali lagi, mirip dengan layanan sebelumnya, kami juga mengamati tidak ada atau peningkatan minimal ketika jumlah



GAMBAR 11. **Perubahan median throughput sebagai efek dari penskalaan horizontal di lingkungan layanan aplikasi Azure—layanan kota.**



GAMBAR 12. **Perubahan median throughput sebagai efek dari penskalaan vertikal di lingkungan layanan aplikasi Azure—layanan kota.**

Jumlah instance meningkat menjadi 15, dan terjadi penurunan throughput dengan peningkatan menjadi 20 instance. Efek penskalaan vertikal juga jauh lebih besar dalam kasus layanan *Rute* – kami telah mengamati peningkatan kinerja 200% dengan perubahan dari konfigurasi 1 × B1.1:75 menjadi 1 × S2:3.5 dalam kasus platform Java, dan peningkatan signifikan tidak kurang dari 48% dalam kasus konfigurasi horizontal lainnya. Perlu dicatat bahwa dalam kasus pengujian *Kota* , peningkatan skala vertikal menghasilkan penurunan kinerja aktual dalam sebelas kasus. Di sini, hanya dalam satu kasus, penurunan kinerja minimal terukur. Secara umum, peningkatan skala vertikal hampir selalu terbukti bermanfaat dalam pengujian ini, dan hasilnya bermakna (setidaknya peningkatan 30%) di semua kasus kecuali empat kasus.

Terakhir, pada bagian ini, kita akan menganalisis biaya berbagai konfigurasi layanan *Rute* – lihat Gambar 18.
Dengan membandingkan biaya untuk layanan *Rute* , kita dapat menyimpulkan bahwa varian konfigurasi Pareto efisien adalah: monolitik B1:1,75, S2:3,5, S4:7 dan P4:14 serta berbasis microservice: 1 × S2:3,5 i 3 × S2:3,5, 3 × P4:14, 6 × P4:14, 10 × P4:14. Dalam kasus Java: serangkaian konfigurasi yang hampir identik dengan tambahan 1 × S4:7 - baik untuk .NET maupun Java. Selain itu, 15 × P4:14 dapat dipilih untuk microservice .NET dan 1 × P4:14 untuk microservice-

berbeda dengan Java. Hampir semua konfigurasi daya tinggi (kecuali 20 × P4:14) efisien Pareto – ini berbeda dengan uji *City* di mana konfigurasi daya tinggi didominasi Pareto.

VI. DISKUSI

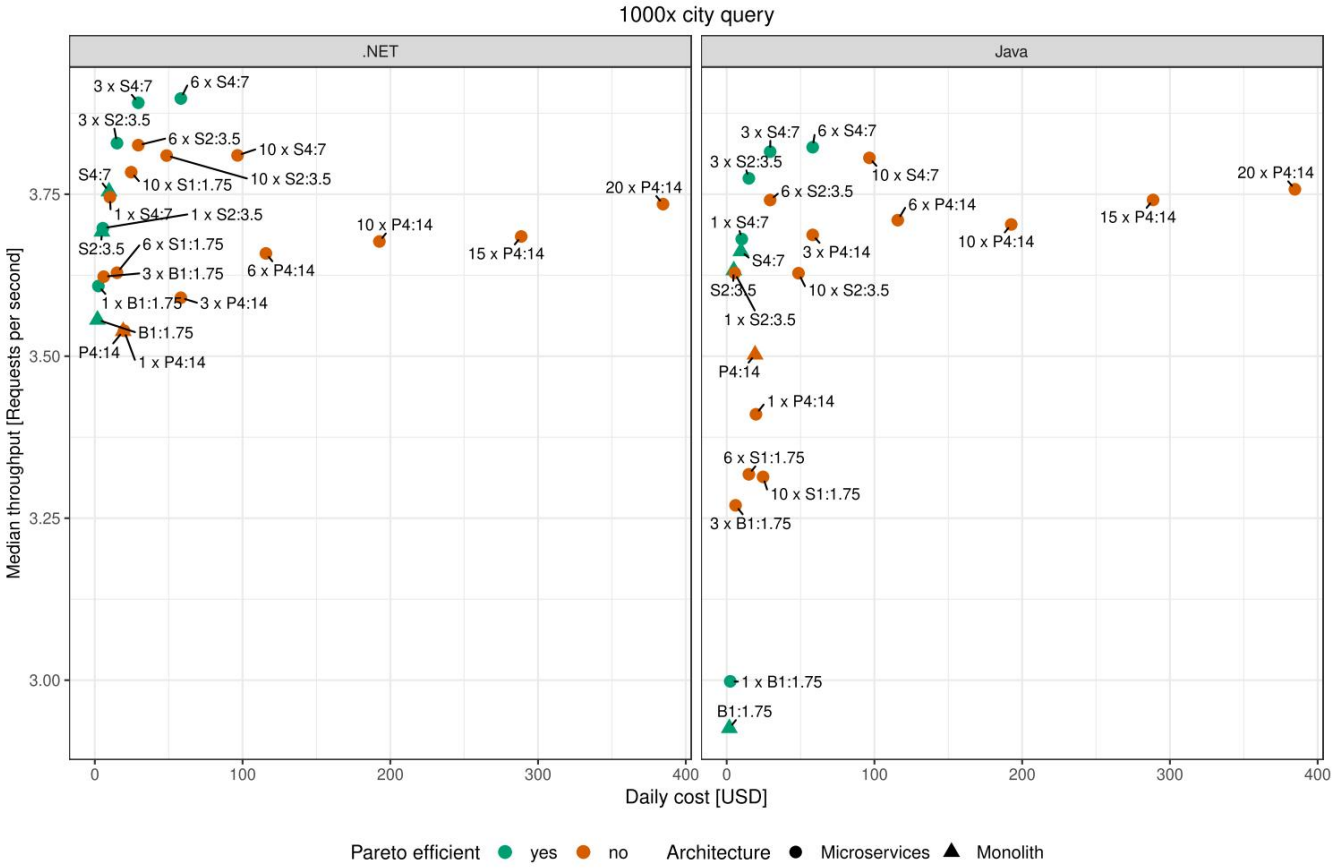
Bagian ini membahas hasil studi kami – kami menjawab pertanyaan penelitian yang dinyatakan dalam pendahuluan mengenai perbedaan kinerja antara aplikasi monolitik dan mikroservis, pendekatan penskalaan, dan teknologi implementasi. Di sini, kami menggeneralisasi penamaan layanan benchmark kami: *City* disebut sebagai layanan *sederhana* dan *Route* sebagai layanan *kompleks* .

A. (RQ1) APAKAH PERBEDAAN KINERJA ANTARA APLIKASI MONOLITIK DENGAN APLIKASI MIKROLAYANAN?

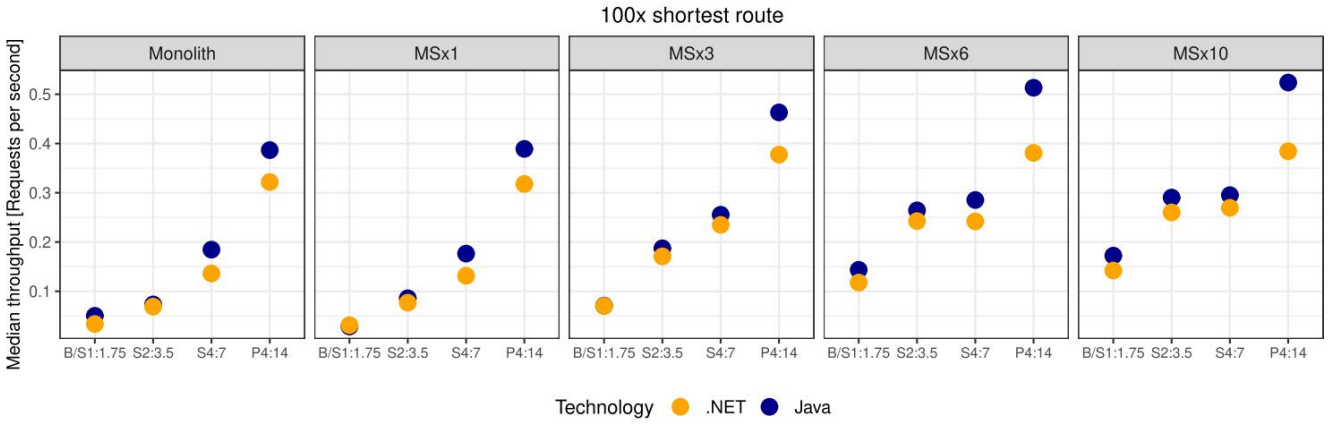
Pada satu mesin, arsitektur monolit berkinerja lebih baik daripada arsitektur microservice (lihat Gambar 4) karena adanya overhead tambahan dalam pengiriman permintaan antar komponen microservice.
Perlu dicatat bahwa perbedaan ini tidak terlihat dalam benchmark cloud kami karena application gateway di-deploy pada mesin virtual terpisah, yang mengurangi beban mesin utama yang menampung microservice.

Dengan demikian, untuk kedua eksperimen cloud tersebut, alih-alih membandingkan kinerja kedua arsitektur yang berjalan pada tipe VM yang sama, kita harus membandingkan kinerja arsitektur yang dihosting pada konfigurasi yang memiliki biaya infrastruktur serupa.

Ketika perbandingan semacam itu dilakukan, arsitektur monolit mengungguli arsitektur microservices (lihat Gambar 6, 8, 13, dan 18), meskipun microservices masih memiliki hak akses istimewa, karena saat pengujian kinerja, hanya satu layanan yang berjalan (untuk detailnya lihat Bagian IV-D).



GAMBAR 13. **Throughput dan biaya di lingkungan layanan aplikasi Azure—layanan kota.**

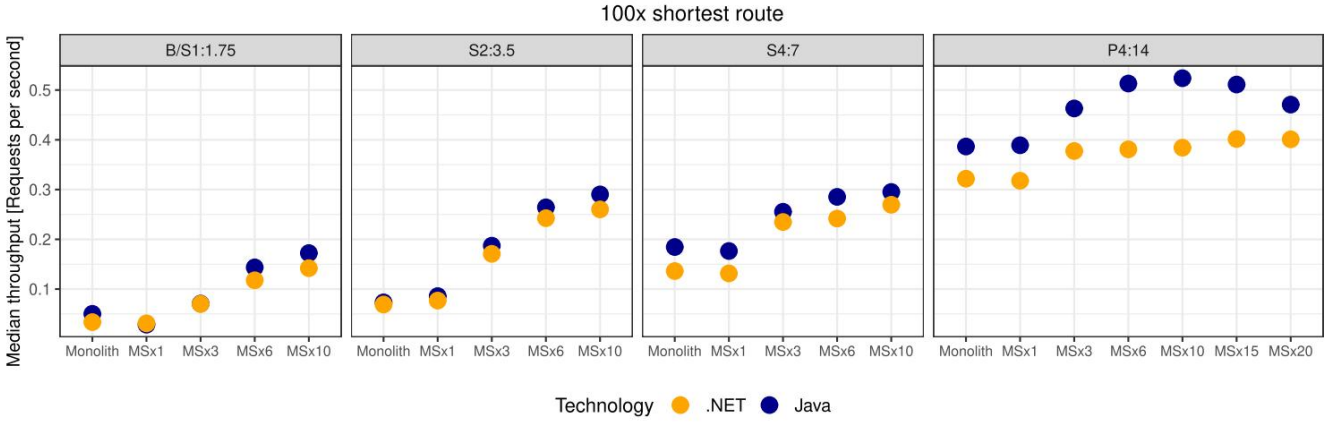


GAMBAR 14. **Throughput vs. penskalaan horizontal di lingkungan layanan aplikasi Azure—layanan rute.**

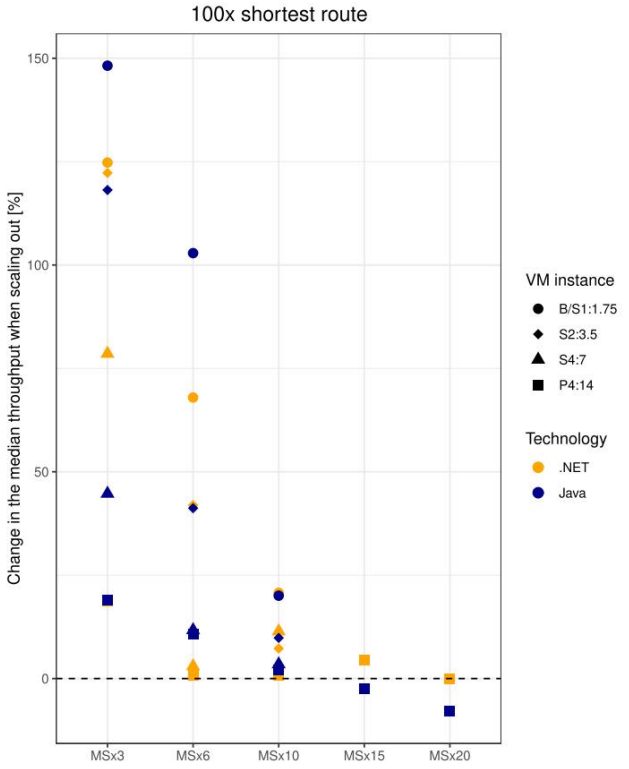
B. (RQ2) MANAKAH DARI DUA ARSITEKTUR DAN PENDEKATAN SKALA YANG HARUS DIPILIH UNTUK MENDAPATKAN MANFAAT TERBAIK BAGI APLIKASI DARI SKALA?

Monolit yang diskalakan secara vertikal efisien Pareto dalam melayani permintaan sederhana dan kompleks pada semua konfigurasi perangkat keras kecuali P4:14 terlepas dari implementasinya. teknologi. Perlu dicatat juga, bahwa semua percobaan yang dijalankan pada P4:14 ternyata didominasi Pareto.

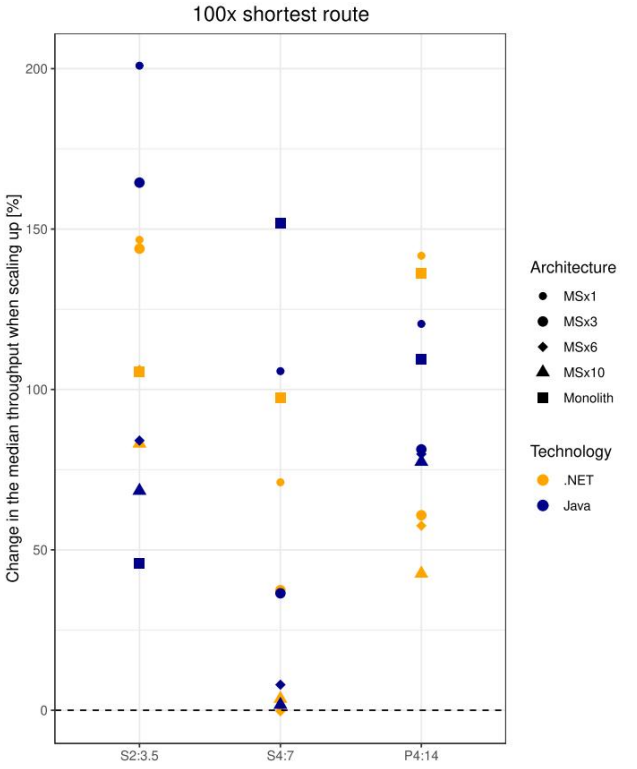
Demikian pula, peningkatan skala aplikasi berbasis microservice berkinerja baik dan lebih hemat biaya daripada peningkatan skala horizontal. Namun demikian, peningkatan skala vertikal dibatasi oleh VM paling canggih yang tersedia; oleh karena itu, kinerja terbaik



GAMBAR 15. **Throughput vs. penskalaan vertikal di lingkungan layanan aplikasi Azure—layanan rute.**



GAMBAR 16. **Perubahan median throughput sebagai efek dari penskalaan horizontal di lingkungan layanan aplikasi Azure—layanan rute.**

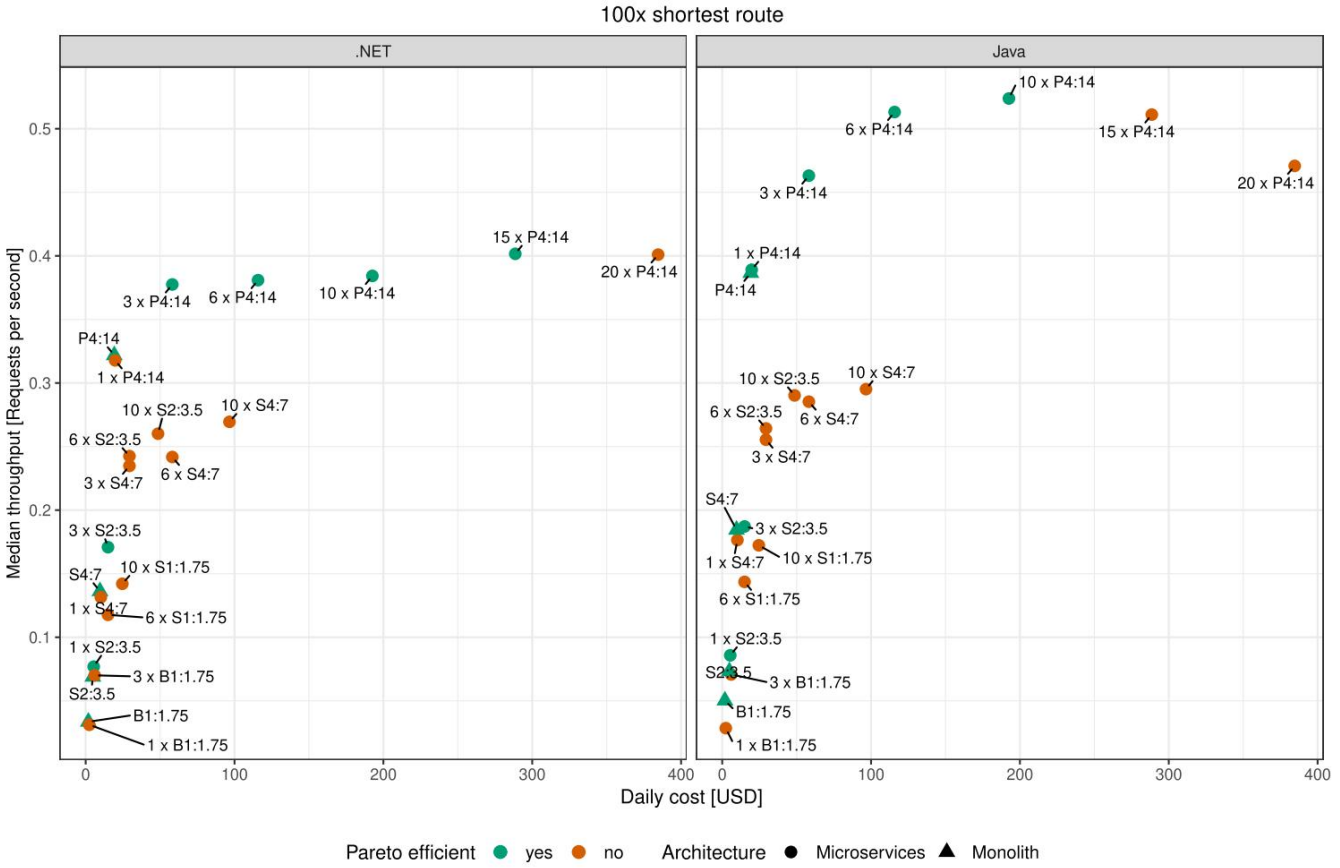


GAMBAR 17. **Perubahan median throughput sebagai efek dari penskalaan vertikal di lingkungan layanan aplikasi Azure—layanan rute.**

Hal ini dicapai melalui arsitektur microservice yang diskalakan secara vertikal dan horizontal.

Eksperimen kami juga menunjukkan bahwa rasio biaya-kinerja terbaik dicapai dengan tiga instance dari platform yang paling canggih. Selain itu, penskalaan horizontal dan vertikal menunjukkan peningkatan throughput yang signifikan dalam kasus layanan yang kompleks dibandingkan dengan layanan yang sederhana.

Selain itu, dengan arsitektur microservice, efek penskalaan horizontal berbeda secara signifikan dalam kasus arsitektur sederhana dan layanan kompleks. Mengenai jumlah instance, dalam kasus layanan sederhana, kinerja puncak dicapai dengan jumlah mesin virtual yang lebih sedikit daripada pada layanan kompleks. Terdapat batasan yang terlihat pada penskalaan horizontal untuk kedua jenis layanan, di mana peningkatan lebih lanjut jumlah instance tidak meningkatkan dan bahkan dapat menurunkan kinerja. Efek penskalaan berlebih ini muncul ketika beban CPU yang dihasilkan dari distribusi beban melebihi manfaat dari peningkatan daya pemrosesan total.



GAMBAR 18. Throughput dan biaya di lingkungan layanan aplikasi Azure—layanan rute.

Terakhir, teknologi implementasi (Java vs. C# .NET) tidak memberikan dampak yang signifikan pada kinerja skalabilitas.

C. (RQ3) DALAM KEADAAN APAKAH TEKNOLOGI IMPLEMENTASI (Java VS. C# .NET) APAKAH ADA KEUNTUNGAN ATAU KERUGIAN DALAM HAL KINERJA?

Dalam pemilihan platform teknologi, karakteristik beban jaringan versus beban CPU harus diperhitungkan.

– Untuk layanan jaringan intensif dengan beban CPU rendah, .NET adalah pilihan yang lebih baik daripada Java; di sisi lain, Java secara konsisten terbukti memanfaatkan CPU lebih baik dalam layanan yang intensif komputasi. .NET juga lebih baik memanfaatkan perangkat keras murah dengan kapasitas komputasi yang lebih rendah selama komputasi tidak intensif (lihat grafik pertama pada Gambar 10). Di sisi lain, platform Java memanfaatkan mesin yang lebih bertenaga dalam kasus layanan yang intensif komputasi (lihat grafik terakhir pada Gambar 15).

D. TEMUAN LAINNYA

Implementasi Java yang diimplementasikan di Azure Spring Cloud berkinerja lebih baik (terutama dalam kasus layanan sederhana) dibandingkan dengan implementasi serupa yang diimplementasikan pada platform lain. mesin di bawah lingkungan Azure App Service. Selain itu, biaya menjalankan aplikasi di bawah konfigurasi perangkat keras tertentu di Azure Spring Cloud lebih rendah daripada di Azure App Service.

VII. ANCAMAN TERHADAP VALIDITAS

Pada bagian ini, kami melaporkan ancaman terhadap validitas penelitian kami. Kami membedakan antara tiga jenis ancaman terhadap validitas [2], [69]: •

validitas konstruk – sejauh mana variabel independen dan dependen secara akurat mengukur konsep yang seharusnya diukur;

• Validitas internal – sejauh mana efek yang diamati hanya disebabkan oleh kondisi perlakuan eksperimental; • Validitas eksternal – sejauh mana temuan

penelitian dapat digeneralisasikan di luar lingkungan eksperimental.

A. VALIDITAS KONSTRUK

Kami mempersempit penilaian kinerja pada pengukuran throughput. Meskipun throughput adalah metrik kinerja yang dominan, metrik lain, seperti waktu respons (latensi) dan pemanfaatan CPU, juga telah digunakan dalam studi sebelumnya.

[4], [14], [17], [18], [42], [70]. Motivasi kami adalah bahwa throughput juga umum digunakan sebagai ukuran skalabilitas [18], sementara dalam penelitian ini, kami juga tertarik untuk membandingkan skalabilitas kedua gaya arsitektur. Meskipun demikian, kami menyadari bahwa throughput tidak mencakup semua aspek kinerja, dan penelitian selanjutnya harus menyelidiki metrik kinerja lainnya.

B. VALIDITAS INTERNAL

Pembaca perlu mengingat bahwa eksperimen di cloud publik tidak dapat sepenuhnya dikendalikan [67]. Memang, dalam studi percontohan, kami mengamati bahwa hasil kinerja sangat bervariasi antara berbagai percobaan. Misalnya, beberapa percobaan menunjukkan throughput terukur yang lebih buruk untuk mesin virtual kelas atas dibandingkan dengan mesin kelas bawah. Alasan fenomena ini tidak diketahui. Kita mungkin menduga adanya variabilitas antar instance VM atau gangguan jaringan acak.
Di sisi lain, variabilitas dalam satu rangkaian pengujian sangat stabil. Untuk menghilangkan pengaruh bias acak tersebut, kami telah menggunakan pengulangan pengujian dan metode pemilihan median dari median yang dijelaskan pada Bagian IV-D.

Faktor lain yang mungkin memengaruhi hasil eksperimen adalah pemilihan titik rute secara acak dalam benchmark *Rute* . Kami telah memverifikasi di lingkungan lokal bahwa pengulangan serangkaian kueri identik menghasilkan hasil throughput yang lebih baik. Namun, kebetulan seperti itu sangat tidak mungkin, mengingat 2000 kueri dikirim dalam satu rangkaian pengujian.

C. VALIDITAS EKSTERNAL

Ancaman terpenting terhadap validitas eksternal studi kami berkaitan dengan keterwakilan objek eksperimen. Karena hanya terdiri dari dua layanan independen, aplikasi kami jauh dari tolok ukur yang realistis. Dalam aplikasi nyata, puluhan (jika bukan ratusan) layanan mikro perlu terus berkomunikasi melalui protokol seperti HTTP. Komunikasi ini dapat mengakibatkan overhead karena komunikasi antar layanan jika dibandingkan dengan panggilan metode yang dilakukan secara lokal di monolit [13], [71], [72]. Namun, tujuan kami adalah untuk mengisolasi skalabilitas dan menghilangkan faktor-faktor yang dapat memengaruhi variabel dependen. Oleh karena itu, kami mengorbankan sebagian validitas eksternal untuk mendapatkan lebih banyak validitas internal.

Analisis kami mengenai dampak penskalaan vertikal dan horizontal terhadap kinerja aplikasi sebagian besar didasarkan pada hasil pengujian yang dilakukan di bawah layanan cloud Azure App Service.
Oleh karena itu, kesimpulan kami mengenai penskalaan dapat diterapkan pada aplikasi yang diimplementasikan di lingkungan serupa dan pada mesin virtual dengan karakteristik yang sebanding. Namun, analisis biaya kami sangat terkait dengan model penetapan harga platform Azure. Dengan demikian, sebelum menggeneralisasikan hasil kami ke cloud publik lainnya, pembaca harus membandingkan model penetapan harga mereka dengan cermat. Di sisi lain, kesimpulan kami mengenai dampak teknologi yang dipilih (Java vs. C# .NET) pada kinerja aplikasi dapat digeneralisasikan ke lingkungan cloud dan jenis aplikasi lainnya.

VIII. KESIMPULAN DAN PEKERJAAN MASA

DEPAN Meskipun microservices semakin mendapatkan momentum di industri TI, studi empiris yang mengevaluasi kinerja dan skalabilitasnya masih jarang, sementara pengalaman langsung hanya berasal dari raksasa TI global dengan jutaan pengguna bersamaan. Oleh karena itu, peralihan ke microservices oleh perusahaan kecil seringkali didasarkan pada intuisi daripada informasi yang solid. Untuk mendukung arsitek perangkat lunak dalam pengambilan keputusan rasional tentang migrasi sistem ke microservices, atau mengembangkan seluruh aplikasi dari awal dengan gaya arsitektur ini, kami melakukan serangkaian eksperimen terkontrol di tiga lingkungan penerapan yang berbeda (lokal, Azure Spring Cloud, dan Azure App Service).

Kami secara ekstensif menyelidiki pengaruh gaya arsitektur (monolitik vs. layanan mikro) dan teknologi implementasi (Java vs. C# .NET) terhadap kinerja dan skalabilitas aplikasi. Pelajaran penting yang kami peroleh adalah sebagai berikut:

- Pada satu mesin, arsitektur monolit berkinerja lebih baik daripada arsitektur lainnya. rekanan berbasis microservice;
- Platform Java memanfaatkan mesin yang lebih bertenaga dengan lebih baik untuk layanan yang membutuhkan komputasi intensif jika dibandingkan dengan .NET; efek platform tersebut berbalik jika layanan yang tidak membutuhkan komputasi intensif dijalankan pada perangkat keras dengan kapasitas komputasi rendah;
- Penskalaan vertikal lebih hemat biaya daripada penskalaan horizontal di cloud Azure; • Penskalaan melampaui jumlah instance tertentu menurunkan kinerja aplikasi;
- Teknologi implementasi tidak memiliki dampak yang signifikan terhadap kinerja skalabilitas.

Kesimpulannya, arsitektur microservice bukanlah yang paling cocok untuk setiap konteks. Arsitektur monolitik tampaknya merupakan pilihan yang lebih baik untuk sistem sederhana dan berukuran kecil yang tidak perlu mendukung sejumlah besar pengguna bersamaan. Kami berharap temuan kami akan membantu perusahaan menghindari terburu-buru menggunakan microservice hanya karena sedang tren, terutama jika hasil yang lebih baik dapat diperoleh dengan meningkatkan skala sistem monolitik mereka.

Kami juga percaya bahwa baik peneliti maupun praktisi dapat memperoleh manfaat dari aplikasi benchmarking referensi kami,11 dan menggunakannya sebagai titik awal untuk eksperimen lebih lanjut.

Dalam pekerjaan selanjutnya, kami bermaksud untuk membuat aplikasi benchmarking kami lebih kompleks sehingga akan lebih menyerupai sistem nyata yang digunakan oleh perusahaan. Perluasan pertama adalah mengembangkan lebih banyak microservice, yang beberapa di antaranya akan saling berkomunikasi. Realitas aplikasi juga akan mendapat manfaat dari keberadaan basis data. Arah masa depan menarik lainnya adalah menyediakan alternatif implementasi baru untuk bahasa pemrograman lain yang direkomendasikan untuk pengembangan microservice, misalnya, Golang, Python, dan Node.js. Selain itu, kami berencana untuk menerapkan dan melakukan benchmarking aplikasi kami di platform cloud lain, termasuk Amazon Web Services, Google Cloud Platform, dan Alibaba Cloud. Akan menarik juga untuk mengadopsi lebih banyak metrik penilaian kinerja, termasuk waktu respons (latensi) dan pemanfaatan CPU. Terakhir,

11https://github.com/annaojdowska/monolith-vs-microservices

Untuk mengatasi masalah fluktuasi kinerja akibat lingkungan bersama, dalam studi lanjutan, kami mempertimbangkan untuk menjalankan VM di lingkungan pribadi (misalnya Azure App Service Environment) yang dikhususkan secara eksklusif untuk satu pelanggan.

PENGAKUAN

Para penulis ingin mengucapkan terima kasih kepada Szymon Okrój, yang mengembangkan implementasi Java dari aplikasi benchmark mereka sebagai bagian dari proyek tesis masternya [61].

REFERENSI

[1] A. Przybyÿek, "Di mana letak kebenaran: AOP dan dampaknya terhadap modularitas perangkat lunak," dalam *Pendekatan Fundamental untuk Rekayasa Perangkat Lunak,* D. Giannakopoulou dan F. Orejas, Eds. Berlin, Jerman: Springer, 2011, hlm. 447–461.

[2] A. Przybyÿek, "Studi empiris tentang dampak AspectJ terhadap evolusi perangkat lunak," *Empirical Softw. Eng.,* vol. 23, no. 4, hlm. 2018–2050, 2018.

[3] M. Fowler. *Microservice Premium.* Diakses: 30 Mei 2020. [Online]. Tersedia: https://Martinfowler.com/bliki/MicroservicePremium.html [4] N. Bjørndal, A. Bucchiarone, M. Mazzara, N. Dragoni, S. Dustdar, FB Kessler, dan T. Wien, "Migrasi dari monolit ke layanan mikro: Tolok ukur studi kasus," Laporan Teknis, 2020. [Online]. Tersedia: https://www.researchgate.net/profile/Manuel-Mazzara/publication/339749917_Migration_from_Monolith_to_Microservices_Benchmarking_a_Case_Study/links/5e6359034585153fb3c8515f/Migration-from-Monolith-to-Microservices-Benchmarking-a-Case-Study.pdf [5] B. Terziÿ, V. Dimitrieski, S. Kordiÿ, G. Milosavljeviÿ, dan I. Lukoviÿ, "Pengembangan dan evaluasi microbuilder: Alat berbasis model untuk spesifikasi arsitektur perangkat lunak microservice REST," *Enterprise Inf. Syst.,* vol. 12, no. 8–9, hlm. 1034–1057, 2018.

[6] J. Lewis dan M. Fowler. (Mar. 2014). *Microservices: Definisi Istilah Arsitektur Baru Ini.* [Online]. Tersedia: https://www.Martinfowler.com/articles/microservices.html [7] C. Posta, *Microservices for Java Developers: A Hands-on Introduction to Frameworks Containers.* Newton, MA, USA: O'Reilly Media, 2016.

[8] R. Rajesh, *Spring Microservices.* London, Inggris: Packt, 2016.

[9] A. Cockroft. (Agustus 2004). *Migrasi ke Layanan Mikro.* [Online]. Tersedia: https://youtu.be/1wiMLkXz26M [10] PZYN Doro ski, A. Brzeski, J. Cychnerski, dan T. Dziubich, "Menuju komputasi awan perawatan kesehatan," dalam *Prosiding Konferensi Internasional ke-36 tentang Arsitektur Sistem Informasi. Technol.,* J. ÿwiÿtek, L. Borzemski, A. Grzech, dan Z. Wilimowska, Eds. Cham, Swiss: Springer, 2016, hlm. 87–97.

[11] J. Soldani, DA Tamburri, dan W.-J. Van Den Heuvel, "Kesulitan dan keuntungan microservices: Tinjauan literatur abu-abu sistematis," *J. Syst. Softw.,* vol. 146, hlm. 215–232, Des. 2018.

[12] H. Vural, M. Koyuncu, dan S. Misra, "Studi kasus tentang pengukuran ukuran microservices," dalam *Aplikasi Ilmu Komputer,* O. Gervasi, B. Murgante, S. Misra, E. Stankova, CM Torre, AMA Rocha, D. Taniar, BO Apduhan, E. Tarantino, dan Y. Ryu, Eds. Cham, Swiss: Springer, 2018, hlm. 454–463.

[13] L. Carvalho, A. Garcia, WKG Assunç ao, R. de Mello, dan MJ de Lima, "Analisis kriteria yang diadopsi di industri untuk mengekstrak layanan mikro," dalam *Prosiding Lokakarya Internasional Bersama ke-7 yang Melakukan Studi Empiris Industri,* 2019, hlm. 22–29.

[14] A. Kwan, J. Wong, H.-A. Jacobsen, dan V. Muthusamy, "Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres," dalam *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS),* Des. 2019, hlm. 80–90.

[15] P. Di Francesco, P. Lago, dan I. Malavolta, "Membangun arsitektur dengan layanan mikro: Sebuah studi pemetaan sistematis," *J. Syst. Softw.,* vol. 150, hlm. 77–97, April 2019.

[16] J. Jaworski, W. Karwowski, dan M. Rusek, "Aplikasi cloud berbasis microservice yang diporting ke unikernel: Perbandingan kinerja berbagai teknologi," dalam *Proc. 40th Anniversary Int. Conf. Inf. Syst. Archit. Technol.,* L. Borzemski, J. ÿwiÿtek, dan Z. Wilimowska, Eds. Cham, Swiss: Springer, 2019, hlm.255–264.

[17] M. Jayasinghe, J. Chathurangani, G. Kuruppu, P. Tennage, dan S. Perera, "Analisis perilaku throughput dan latensi di bawah dekomposisi microservice," dalam *Web Engineering,* M. Bielikova, T. Mikkonen, dan C. Pautasso, Eds. Cham, Swiss: Springer, 2020, hlm. 53–69.

[18] F. Auer, V. Lenarduzzi, M. Felderer, dan D. Taibi, "Dari sistem monolitik ke layanan mikro: Kerangka penilaian," *Inf. Softw. Technol.,* vol. 137, Des. 2021, Art. no. 106600.

[19] M. Viggiato, R. Terra, H. Rocha, M. Tulio Valente, dan E. Figueiredo, "Microservices in practice: A survey study," 2018, *arXiv:1808.04836.*

[20] M. Jagieÿÿo, M. Rusek, dan W. Karwowski, "Kinerja dan ketahanan terhadap kegagalan aplikasi berbasis cloud: Perbandingan arsitektur monolitik dan berbasis microservices," dalam *Sistem Informasi Komputer dan Manajemen Industri,* K. Saeed, R. Chaki, dan V. Janev, Eds. Cham, Swiss: Springer, 2019, hlm. 445–456.

[21] J. Fritzsch, J. Bogner, S. Wagner, dan A. Zimmermann, "Migrasi layanan mikro di industri: Niat, strategi, dan tantangan," dalam *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME),* Des. 2019, hlm. 481–490.

[22] A. Poniszewska-Maraÿda dan E. Czechowska, "Klaster Kubernetes untuk otomatisasi lingkungan produksi perangkat lunak," *Sensors,* vol. 21, no. 5, hal. 1901, 2021.

[23] CM Aderaldo, NC Mendonca, C. Pahl, dan P. Jamshidi, "Persyaratan tolok ukur untuk penelitian arsitektur layanan mikro," dalam *Prosiding Lokakarya Internasional IEEE/ACM ke-1 tentang Pembentukan Infrastruktur Arsitektur Berbasis Rekayasa Perangkat Lunak Komunitas (ECASE),* Mei 2017, hlm. 8–13.

[24] A. Poth, H. Urban, dan A. Riel, *Membuat Persyaratan Layanan Produk Dapat Dikirim—Dari Visi Layanan Cloud ke Aliran Nilai Berkelanjutan yang Memenuhi Kebutuhan Pengguna Saat Ini dan Masa Depan.* Cham, Swiss: Springer, 2021.

[25] Y. Wang, H. Kadiyala, dan J. Rubin, "Janji dan tantangan layanan mikro: Sebuah studi eksplorasi," *Empirical Softw. Eng.,* vol. 26, no. 4, hlm. 1–44, Juli 2021.

[26] D. Taibi, V. Lenarduzzi, dan C. Pahl, "Proses, motivasi, dan masalah migrasi ke arsitektur microservices: Investigasi empiris," *IEEE Cloud Comput.,* vol. 4, no. 5, hlm. 22–32, Sep. 2017.

[27] J. Ghofrani dan D. Lübke, "Tantangan arsitektur microservices: Survei tentang keadaan praktik," dalam *Proc. ZEUS,* 2018, hlm. 1–8.

[28] S. Butt, S. Abbas, dan M. Ahsan, "Siklus hidup pengembangan perangkat lunak & jenis pengukuran kualitas perangkat lunak," *Asian J. Math. Comput. Res.,* vol. 11, no. 2, hlm. 112–122, 2016.

[29] A. Jarzÿbowicz dan P. Marciniak, "Sebuah survei tentang mengidentifikasi dan mengatasi masalah analisis bisnis," *Found. Comput. Decis. Sci.,* vol. 42, hlm. 315–337, Des. 2017.

[30] S. Li, H. Zhang, Z. Jia, Z. Li, C. Zhang, J. Li, Q. Gao, J. Ge, dan Z. Shan, "Pendekatan berbasis aliran data untuk mengidentifikasi layanan mikro dari aplikasi monolitik," *J. Syst. Softw.,* vol. 157, Nov. 2019, Art. no. 110380.

[31] H. Stranner, S. Strobl, M. Bernhart, dan T. Grechenig, "Dekomposisi microservice: Studi kasus migrasi perangkat lunak industri besar di industri otomotif," dalam *Proc. 15th Int. Conf. Eval. Novel Approaches Softw. Eng.,* 2020, hlm. 498–505.

[32] M. Bruce dan PA Pereira, *Microservices in Action.* New York, NY, USA: Simon and Schuster, 2018.

[33] A. Banijamali, P. Kuvaja, M. Oivo, dan P. Jamshidi, "Kuksa: Mikroservis adaptif mandiri dalam sistem otomotif," dalam *Peningkatan Proses Perangkat Lunak Berfokus Produk,* M. Morisio, M. Torchiano, dan A. Jedlitschka, Eds. Cham, Swiss: Springer, 2020, hlm. 367–384.

[34] M. Kalske, N. Mäkitalo, dan T. Mikkonen, "Tantangan ketika beralih dari arsitektur monolit ke arsitektur layanan mikro," dalam *Tren Terkini dalam Rekayasa Web* (Catatan Kuliah dalam Ilmu Komputer), vol. 10544, I. Garrigós dan M. Wimmer, Eds. Cham, Swiss: Springer, 2018, doi: 10.1007/978- 3-319-74433-9_3.

[35] W. Karwowski, M. Rusek, G. Dwornicki, dan A. Orÿowski, "Sistem berbasis swarm untuk manajemen layanan mikro terkontainerisasi di cloud yang terdiri dari server heterogen," dalam *Proc. 38th Int. Conf. Inf. Syst. Arsitek. Technol.,* L. Borzemski, J. ÿwiÿtek, dan Z. Wilimowska, Eds. Cham, Swiss: Springer, 2018, hlm. 262–271.

[36] B. Terzic dan V. Dimitrieski, "Pendekatan berbasis model untuk pembentukan arsitektur perangkat lunak microservice," dalam *Proc. Ann. Comput. Sci. Inf. Syst.,* Sep. 2018, hlm. 73.

[37] M. Štefanko, O. Chaloupka, dan B. Rossi, "Pola saga dalam lingkungan microservice reaktif," dalam *Prosiding Konferensi Teknologi Perangkat Lunak Internasional ke-14,* 2019, hlm. 483–490.

[38] C. Rajasekharaiah, *Studi Kasus: Energi.* Berkeley, CA, AS: Apress, 2021, hlm.1–12.

[39] A. Poniszewska-Maraÿda, P. Vesely, O. Urikova, dan I. Ivanochko, "Membangun arsitektur layanan mikro untuk perbankan cerdas," dalam *Advances in Intelligent Networking and Collaborative Systems,* L. Barolli, H. Nishino, dan H. Miwa, Eds. Cham, Swiss: Springer, 2020, hlm. 534–543.

[40] J. Ghofrani dan A. Bozorgmehr, "Migrasi ke microservices: Hambatan dan solusi," dalam *Applied Informatics,* H. Florez, M. Leon, JM Diaz-Nafria, dan S. Belli, Eds. Cham, Swiss: Springer, 2019, hlm. 269–281.

[41] O. Al-Debagy dan P. Martinek, "Tinjauan perbandingan arsitektur microservices dan monolitik," dalam *Proc. IEEE 18th Int. Symp. Comput. Intel. informasi. (CINTI),* November 2018, hlm.149–154.

[42] A. de Camargo, I. Salvadori, RDS Mello, dan F. Siqueira, "Arsitektur untuk mengotomatisasi pengujian kinerja pada layanan mikro," dalam *Prosiding Konferensi Internasional ke-18. Konferensi Integrasi Informasi Layanan Aplikasi Berbasis Web,* November 2016, hlm. 422–429.

[43] V. Lenarduzzi, F. Lomio, N. Saarimäki, dan D. Taibi, "Apakah migrasi sistem monolitik ke layanan mikro mengurangi hutang teknis?" *J. Syst. Perangkat Lunak,* jilid. 169, November 2020, Pasal. TIDAK. 110710.

[44] V. Lenarduzzi dan O. Sievi-Korte, "Tentang dampak negatif independensi tim dalam pengembangan perangkat lunak microservices," dalam *Prosiding Konferensi Internasional ke-19. Konferensi Pengembangan Perangkat Lunak Agile, Companion,* New York, NY, AS, Mei 2018, hlm. 1–4.

[45] F. Ramin, C. Matthies, dan R. Teusner, "Lebih dari sekadar kode: Kontribusi dalam tim rekayasa perangkat lunak Scrum," dalam *Prosiding Konferensi Internasional IEEE/ACM ke-42. perangkat lunak. bahasa Inggris Lokakarya,* New York, NY, AS, Juni 2020, hlm.137–140.

[46] B. Marcinkowski dan B. Gawin, "Studi tentang pendekatan adaptif terhadap peningkatan proses bisnis multi-skenario yang didorong oleh teknologi," *Inf. Technol. People,* vol. 32, no. 1, hlm. 118–146, Feb. 2019.

[47] M. Kalenda, P. Hyna, dan B. Rossi, "Meningkatkan ketangkasan di organisasi besar: Praktik, tantangan, dan faktor keberhasilan," *J. Softw., Evol. Process,* vol. 30, no. 10, hal. e1954, Okt. 2018.

[48] P. Diebold, A. Schmitt, dan S. Theobald, "Menskalakan agile: Bagaimana memilih kerangka kerja yang paling tepat," dalam *Prosiding Konferensi Perangkat Lunak Agile Internasional ke-19. Develop., Companion,* New York, NY, USA, Mei 2018, hlm. 1–4.

[49] S. Theobald, A. Schmitt, dan P. Diebold, "Membandingkan kerangka kerja agile yang berskala berdasarkan praktik yang mendasarinya," dalam *Proses Agile dalam Rekayasa Perangkat Lunak dan Pemrograman Ekstrem,* R. Hoda, Ed. Cham, Swiss: Springer, 2019, hlm. 88–96.

[50] A. Poth, M. Kottke, dan A. Riel, "Menskalakan Agile—Pandangan perusahaan besar tentang penyampaian dan memastikan transisi berkelanjutan," dalam *Proses Agile dalam Rekayasa Perangkat Lunak dan Pemrograman Ekstrem,* A. Przybyÿek dan ME Morales-Trujillo, Eds. Cham, Swiss: Springer, 2020, hlm. 1–18.

[51] A. Khalid, SA Butt, T. Jamal, dan S. Gochhait, "Masalah Agile Scrum pada proyek terdistribusi skala besar: Pengembangan proyek Scrum secara luas," *Int. J. Lunakw. Inovasi. (IJSI),* jilid. 8, tidak. 2, hlm.85–94, 2020.

[52] M. Kowalczyk, B. Marcinkowski, dan A. Przybyÿek, "Kerangka kerja agile berskala: Menangani tantangan terkait proses perangkat lunak dari sebuah grup keuangan dengan pendekatan penelitian tindakan," *J. Softw., Evol. Process,* 2022.

[53] M. Kösling dan A. Poth, "Pengembangan Agile menawarkan kesempatan untuk menetapkan prosedur kualitas otomatis," dalam *Peningkatan Proses Sistem, Perangkat Lunak dan Layanan,* J. Stolfa, S. Stolfa, RV O'Connor, dan R. Messnarz, Eds. Cham, Swiss: Springer, 2017, hlm. 495–503.

[54] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, MK Reiter, dan V. Sekar, "Gremlin: Pengujian ketahanan sistematis layanan mikro," dalam *Proc. IEEE 36th Int. Conf. Distrib. Comput. Syst. (ICDCS),* Jun. 2016, hlm. 57–66.

[55] AB Bondi, "Karakteristik skalabilitas dan dampaknya terhadap kinerja," dalam *Prosiding Lokakarya Internasional ke-2 tentang Kinerja Perangkat Lunak,* New York, NY, AS, 2000, hlm. 195–203.

[56] B. Wilder, *Pola Arsitektur Cloud: Menggunakan Microsoft Azure.* Newton, MA, AS: O'Reilly Media, 2012.

[57] L. Lu, X. Zhu, R. Griffith, P. Padala, A. Parikh, P. Shah, dan E. Smirni, "Penskalaan vertikal dinamis yang digerakkan oleh aplikasi dari mesin virtual dalam kumpulan sumber daya," dalam *Proc. IEEE Netw. Oper. Manage. Symp. (NOMS),* Mei 2014, hlm. 1–9.

[58] S. Spinner, N. Herbst, S. Kounev, X. Zhu, L. Lu, M. Uysal, dan R. Griffith, "Penskalasian memori proaktif aplikasi virtualisasi," dalam *Proc. IEEE 8th Int. Conf. Cloud Comput.,* Jun. 2015, hlm. 277–284.

[59] T. Ueda, T. Nakaike, dan M. Ohara, "Karakterisasi beban kerja untuk layanan mikro," dalam *Proc. IEEE Int. Symp. Workload Characterization (IISWC),* Sep. 2016, hlm. 1–10.

[60] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, dan A. Di Salle, "Menuju pemulihan arsitektur perangkat lunak sistem berbasis layanan mikro," di *Proc. IEEE Int. Konf. perangkat lunak. Arsitek. Lokakarya (ICSAW),* April 2017, hlm.46–53.

[61] S. Okrój, "Analisis perbandingan kinerja arsitektur monolitik dan mikroservis," Tesis MS, Universitas Teknologi Gdansk, Gdaÿsk, Polandia, 2018.

[62] M. Villamizar, O. Garces, dan L. Ochoa, "Perbandingan biaya infrastruktur menjalankan aplikasi web di cloud menggunakan aws lambda dan arsitektur monolitik dan microservice," dalam *Proc. 16th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGrid),* 2016, hlm. 179–182.

[63] J. Shaughnessy, E. Zechmeister, dan J. Zechmeister, *Metode Penelitian Psikologi.* Surrey, BC, Kanada: Kwantlen Polytechnic Univ., 2019.

[64] B. Vermeer. (2020). *Laporan Ekosistem JVM 2020.* Diakses: 22 Oktober 2021. [Online]. Tersedia di: https://snyk.io/wp-content/uploads/jvm_2020.pdf

[65] B. Vermeer. (2021) *Laporan Ekosistem JVM 2021.* Diakses: 22 Oktober 2021. [Online]. Tersedia: https://res.cloudinary.com/snyk/image/upload/v162 3860216/reports/jvm-ecosystem-report-2021.pdf

[66] A. Derezinska dan K. Kwaánik, "Refactoring berbasis kinerja aplikasi web: Sebuah kasus transportasi umum," dalam *Prosiding Konferensi Internasional ke-15 tentang Evaluasi Pendekatan Baru untuk Rekayasa Perangkat Lunak,* 2020, hlm. 611–618.

[67] C. Laaber, J. Scheuner, dan P. Leitner, "Mikrobenchmarking perangkat lunak di cloud. Seberapa burukkah sebenarnya?" *Empirical Softw. Eng.,* vol. 24, no. 4, hlm. 2469–2508, Agustus 2019.

[68] MR López dan J. Spillner, "Menuju batas terukur untuk penskalaan horizontal elastis layanan mikro," dalam *Prosiding Konferensi Internasional ke-10 tentang Komputasi Awan Utilitas,* New York, NY, AS, 2017, hlm. 35–40.

[69] YY Ng dan A. Przybyÿek, "Kehadiran instruktur dalam kuliah video: Temuan awal dari eksperimen online," *IEEE Access,* vol. 9, hlm. 36485–36499, 2021.

[70] T. Salah, MJ Zemerly, CY Yeun, M. Al-Qutayri, dan Y. Al-Hammadi, "Perbandingan kinerja antara layanan berbasis kontainer dan berbasis VM," dalam *Prosiding Konferensi ke-20 tentang Inovasi Komputasi Awan, Jaringan Internet (ICIN),* 2017, hlm. 185–190.

[71] D. Namiot dan M. Sneps-Sneppe, "Tentang arsitektur layanan mikro," *Int. Jurnal Teknologi Informasi Terbuka,* vol. 2, hlm. 24–27, Oktober 2014.

[72] H. Knoche, "Mempertahankan kinerja runtime sambil memodernisasi perangkat lunak monolitik transaksional secara bertahap menuju layanan mikro," dalam *Proc. 7th ACM/SPEC Int. Conf. Perform. Eng.,* New York, NY, USA, Mar. 2016, hlm. 121–124.

GRZEGORZ BLINOWSKI (Anggota, IEEE) lahir di Warsawa, Polandia. Ia menerima gelar M.Sc. dan Ph.D. di bidang ilmu komputer dari Fakultas Elektronika dan Teknologi Informasi, Institut Ilmu Komputer, Universitas Teknologi Warsawa, Polandia, masing-masing pada tahun 1993 dan 2001. Ia memegang Sertifikat Certified Information Systems Security Professional (CISSP) sejak tahun 2014. Sejak tahun 2001, ia menjabat sebagai Asisten Profesor di Kelompok Riset Komputasi Paralel dan Terdistribusi, Institut Ilmu Komputer. Ia adalah penulis dua buku. Minat penelitian dan ilmiahnya meliputi: sistem memori terdistribusi, rekayasa perangkat lunak, teknologi internet, dan keamanan jaringan dan sistem, terutama dalam konteks WSN dan IoT dan baru-baru ini sistem VLC. Ia telah menerima Penghargaan Rektor Universitas Teknologi Warsawa untuk Prestasi Akademik sebanyak dua kali.

ANNA OJDOWSKA menerima gelar Magister Sains (M.Sc.) di bidang ilmu komputer dari Universitas Teknologi Gdaÿsk pada tahun 2020. Ia telah bekerja sebagai Insinyur Perangkat Lunak di IHS Markit sejak Juni 2018. Ia berspesialisasi dalam teknologi .NET. Minat profesional utamanya meliputi arsitektur perangkat lunak, pemrograman fungsional dan aplikasi praktisnya, serta dampaknya pada rekayasa perangkat lunak.

ADAM PRZYBYÿEK menerima gelar master di bidang sistem informasi manajemen dan gelar Ph.D. di bidang rekayasa perangkat lunak, masing-masing pada tahun 2002 dan 2011. Antara tahun 2002 dan 2011, ia bekerja sebagai Konsultan Jaringan dan Instruktur di Cisco Networking Academy. Ia telah bekerja sebagai Asisten Profesor di Universitas Teknologi Gdaÿsk, Polandia, sejak Oktober 2012. Minat penelitian utamanya meliputi rekayasa perangkat lunak empiris dengan fokus pada modularitas perangkat lunak, paradigma pasca-berorientasi objek, dan metode agile. Ia adalah Pendiri Konferensi Internasional tentang Pengembangan Perangkat Lunak Lean dan Agile (https://lasd.pl). Ia juga telah menjabat di komite program ENASE dan ACM SAC, sejak 2015, dan MADEISD@ADBIS sejak awal berdirinya, pada tahun 2019.