Jurnal Teknik Informatika (JUTIF)
Vol. 5, No. 6, Desember 2024, hlm. 1861-1872

DOI: https://doi.org/10.52436/1.jutif.2024.5.6.2617
p-ISSN: 2723-3863
e-ISSN: 2723-3871

ANALISIS DAMPAK REFAKTORING DARI APLIKASI MONOLITIK UNTUK LAYANAN MIKRO PADA WAKTU RESPON MENGGUNAKAN MDA DAN SCA PENDEKATAN

Shidqi Fadhlurrahman Yusri*1 , Dawam Dwi Jatmiko Suwawi2 , Monterico Adrian3

1,2,3Informatika, Fakultas Komputasi, Universitas Telkom, Indonesia
E-mail: 1shidqify@student.telkomuniversity.ac.id, 2dawamdjs@telkomuniversity.ac.id, 3monterico@telkomuniversity.ac.id

(Artikel diterima: 6 Agustus 2024; Revisi: 21 Agustus 2024; diterbitkan: 29 Desember 2024)

Abstrak

Studi ini menyelidiki dampak refactoring dari arsitektur monolitik ke arsitektur microservices terhadap waktu respons aplikasi. Arsitektur monolitik, yang awalnya dipilih karena kemudahan pengembangan, menghadapi tantangan skalabilitas seiring pertumbuhan aplikasi. Microservices menawarkan solusi dengan memungkinkan penyebaran layanan independen dan peningkatan skalabilitas. Penelitian ini menggunakan metodologi Meta-Data Aided (MDA) dan Static Code Analysis (SCA) untuk memfasilitasi proses refactoring, menerapkannya pada proyek aplikasi inventaris dari platform pengembangan perangkat lunak kolaboratif (GitHub). Refactoring melibatkan dekomposisi aplikasi monolitik, pengemasannya dengan Docker, dan evaluasi kinerja menggunakan JMeter. Hasil menunjukkan bahwa microservices secara signifikan mengurangi waktu respons, terutama pada tugas interaksi API. Meskipun microservices meningkatkan skalabilitas dan fleksibilitas, mereka membutuhkan pengelolaan komunikasi layanan yang cermat. Penelitian ini meningkatkan pemahaman tentang manfaat microservices dalam hal waktu respons dan menawarkan panduan praktis bagi pengembang yang mempertimbangkan refactoring.

Kata kunci: *Layanan mikro, Bantuan metadata, Monolitik, Refaktor, Waktu respons, Analisis kode statis.*

1. PENDAHULUAN

Arsitektur monolitik adalah pilihan pertama bagi pengembang perangkat lunak saat membangun aplikasi mereka. Pilihan ini bukan tanpa alasan, karena menawarkan kemudahan dalam pengembangan, pengujian, dan penyebaran, yang dapat dikelola melalui penyebaran tunggal [1], [2]. Namun, kemudahan ini tidak dapat dipertahankan seiring dengan perkembangan aplikasi, sehingga menimbulkan tantangan yang semakin besar dalam pemeliharaannya. Masalah seperti persyaratan skalabilitas, struktur kode yang kompleks, dan ukuran aplikasi yang semakin besar menyebabkan waktu penyebaran yang lebih lama, yang menjadi hambatan signifikan dalam pemeliharaan [1], [3], [4].

Mengingat alasan-alasan di atas, microservices menghadirkan solusi untuk masalah-masalah ini. Microservices menawarkan keuntungan seperti skalabilitas yang lebih baik, layanan yang berfokus pada fungsi spesifik, dan penyebaran layanan yang independen [5], [6], [7]. Namun, terlepas dari manfaat-manfaat ini, tantangan tetap ada, khususnya dalam mentransformasikan aplikasi dari struktur monolitik ke arsitektur microservices. Refactoring adalah salah satu pendekatan yang dapat digunakan untuk mengatasi tantangan ini.

Refactoring adalah praktik umum di kalangan pengembang perangkat lunak untuk meningkatkan kualitas perangkat lunak mereka. Fokusnya adalah meningkatkan kualitas dan pemeliharaan perangkat lunak tanpa mengubah fungsinya [8], [9]. Pendekatan ini diadopsi karena perangkat lunak terus berevolusi untuk memenuhi persyaratan baru, meningkatkan fitur yang ada, dan mengatasi kekurangan yang ada [8]. Meskipun refactoring merupakan metode praktis untuk perubahan struktural dan peningkatan kualitas dalam aplikasi, proses ini kompleks bagi pengembang dan perusahaan [5]. Kompleksitas ini muncul karena perlunya mempertimbangkan berbagai aspek, khususnya mengenai kinerja perangkat lunak, seperti waktu respons.

Dalam standar ISO/IEC 25010:2011 [10], yang menguraikan kualitas perangkat lunak, salah satu aspek pengukurannya adalah Perilaku Waktu, dengan waktu respons sebagai komponen dari aspek ini.
Waktu respons mewakili waktu tunggu untuk hasil yang diproses oleh aplikasi. Waktu tunggu ini adalah durasi yang dibutuhkan data dari saat klien mengirimkan permintaan hingga klien menerima data kembali dari server [11], [12]. Semakin lama waktu yang dibutuhkan, semakin buruk pengalaman pengguna. Selanjutnya, Khan R. [11] menjelaskan bahwa waktu respons adalah parameter yang digunakan untuk mengevaluasi kinerja dan efisiensi suatu aplikasi.

Untuk memastikan bahwa kinerja perangkat lunak tetap optimal setelah proses refactoring, khususnya terkait waktu respons, pengembang perlu memilih pendekatan yang tepat untuk refactoring. Beberapa pendekatan refactoring tersedia, termasuk Meta-data Aided (MDA) dan Static Code Analysis (SCA).
[1], [4]. MDA adalah pendekatan yang menganalisis aplikasi berdasarkan sumber data yang tersedia seperti diagram, deskripsi aplikasi, spesifikasi sistem, dan dokumen lainnya [13]. Sebaliknya,

SCA adalah pendekatan yang menggunakan data kode sumber dari aplikasi sebagai masukan untuk analisis [1], [14].

Penelitian mengenai refactoring aplikasi monolitik menjadi microservices telah dibahas di beberapa jurnal [5], [6], [8], [15].

Goncalves N. et al. [15]
Menyoroti tantangan refactoring dan dampaknya terhadap kinerja, menunjukkan variasi hasil latensi antara arsitektur monolitik dan mikroservis.

Traini L. et al. [8] meneliti dampak refactoring terhadap waktu eksekusi, dan menemukan bahwa sekitar 55% commit mempengaruhi kinerja, dengan 75% menunjukkan tidak ada perubahan signifikan. Ren Q. et al. [5] membahas tahapan dan tantangan refactoring serta manfaat arsitektur microservices dalam hal ketahanan dan manajemen. Penelitian ini menekankan pentingnya memilih metode refactoring yang tepat untuk meningkatkan kinerja aplikasi.

Zaragoza [6], dalam studinya, mengeksplorasi proses migrasi dari sistem perangkat lunak monolitik ke arsitektur microservices (MSA). Dia menekankan manfaat MSA, seperti peningkatan pemeliharaan, skalabilitas yang lebih baik, dan lebih cepat.

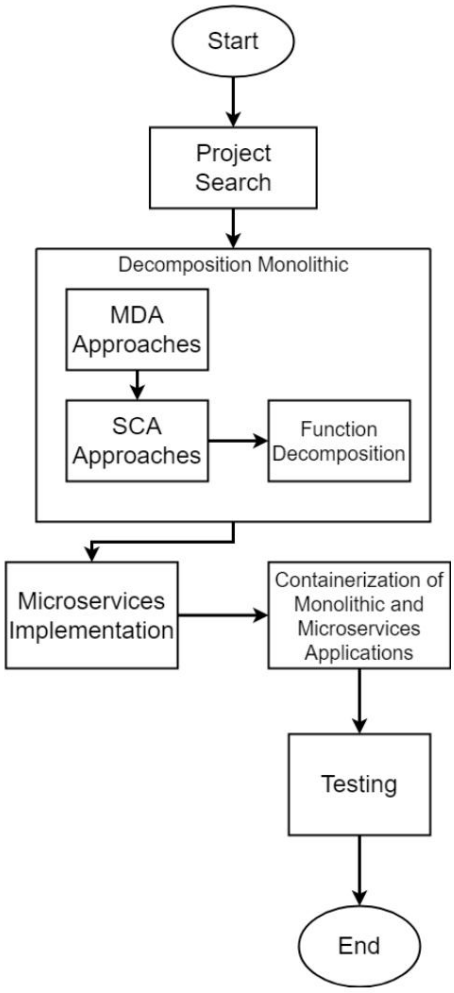
Penyebaran. Migrasi ini terdiri dari dua fase utama: pertama, membangun arsitektur microservices dari kode sumber monolitik yang ada, dan kedua, mengubah kode menjadi microservices yang mematuhi prinsip-prinsip MSA.

Studi ini mengusulkan metode baru menggunakan pola transformasi untuk mendukung transisi ini tanpa mengorbankan logika bisnis dan kinerja aplikasi. Zaragoza [6] juga membahas kompleksitas yang muncul, khususnya mengenai ketergantungan kelas, yang sering mengganggu enkapsulasi microservice yang efektif. Dengan memanfaatkan alat otomatis yang disebut MonoToMicro, pendekatan ini diuji dalam konteks aplikasi sistem Java monolitik. Metode ini bertujuan untuk menyederhanakan identifikasi dan pengelompokan kelas ke dalam microservice potensial, sehingga mengatasi tantangan dalam transformasi kode dan memastikan bahwa karakteristik operasional MSA dapat terpenuhi.

Oleh karena itu, penelitian ini akan berfokus pada eksplorasi menyeluruh tentang efek waktu respons yang dihasilkan dari refactoring aplikasi monolitik menjadi aplikasi microservices menggunakan pendekatan MDA dan SCA. Pengujian akan dilakukan pada kedua arsitektur dengan kasus uji dan lingkungan yang sama, untuk memastikan perbandingan yang adil di antara keduanya. Studi ini bertujuan untuk memberikan wawasan yang lebih besar kepada pengembang dalam memilih arsitektur yang paling tepat untuk pengembangan aplikasi mereka dan pendekatan yang akan digunakan.

2. METODE PENELITIAN

Metodologi yang digunakan dalam penelitian ini secara umum mencakup pencarian proyek, dekomposisi atau pemisahan aplikasi monolitik, pembangunan aplikasi microservices, dan pengujian seperti yang dijelaskan pada Gambar 1.



Gambar 1. Metode Penelitian

Penelitian dimulai dengan pencarian proyek pada platform pengembangan perangkat lunak kolaboratif seperti GitHub. Setelah proyek yang sesuai diidentifikasi berdasarkan kriteria penelitian, dekomposisi monolitik akan dilakukan untuk memisahkan modul yang tidak memiliki ketergantungan [15]. Proses dekomposisi monolitik menggunakan dua pendekatan: Meta-data Aided (MDA) dan Static Code Analysis (SCA). Setelah menerapkan dan menganalisis pendekatan MDA dan SCA, perlu untuk memisahkan fungsi terkait berdasarkan hasil dari MDA dan SCA.

Setelah proses dekomposisi selesai, langkah selanjutnya adalah implementasi microservices sesuai dengan hasil dekomposisi. Selama proses implementasi microservices, penyesuaian pada kode akan dilakukan untuk memfasilitasi komunikasi antara fungsi-fungsi di dalam aplikasi dan fungsi-fungsi di API lain.

Setelah mengimplementasikan microservices, aplikasi akan dikontainerisasi untuk memungkinkan komunikasi API. Kontainerisasi juga akan diterapkan pada aplikasi monolitik, memastikan bahwa lingkungan pengembangan untuk aplikasi monolitik konsisten dengan lingkungan pengembangan aplikasi microservices. Proses terakhir yang akan dilakukan adalah pengujian, yang akan membandingkan hasil kinerja waktu respons antara aplikasi monolitik dan microservices.

2.1. Pencarian Proyek

Proyek yang digunakan dalam penelitian ini akan dipilih melalui pencarian di GitHub. GitHub adalah platform yang menampung banyak proyek open-source di berbagai domain [16]. Proyek yang dipilih akan menggunakan bahasa pemrograman JavaScript, karena microservices umumnya dikaitkan dengan bahasa pemrograman ringan seperti JavaScript dan telah terbukti efektif dalam praktiknya [17]. Selain itu, proyek yang dipilih harus memiliki dokumen artefak terkait di dalam repositori GitHub-nya. Dokumen-dokumen ini digunakan untuk mengimplementasikan pendekatan MDA, yang membutuhkan artefak seperti diagram dan spesifikasi aplikasi.

Dari pencarian peneliti terhadap aplikasi berdasarkan kriteria yang disebutkan, ditemukan satu repositori yang sesuai: aplikasi inventaris yang dikembangkan oleh Mohamed Eleshmawy [18]. Aplikasi ini dirancang untuk melacak status pesanan sejak pelanggan melakukan pemesanan hingga pesanan ditandai sebagai dikirim oleh penjual dan kemudian sebagai diterima oleh penerima. Proyek ini menggunakan JavaScript sebagai bahasa pemrogramannya dan mencakup dokumen ERD (Entity Relationship Diagram) dan Persyaratan Fungsional untuk menjelaskan spesifikasi aplikasi.

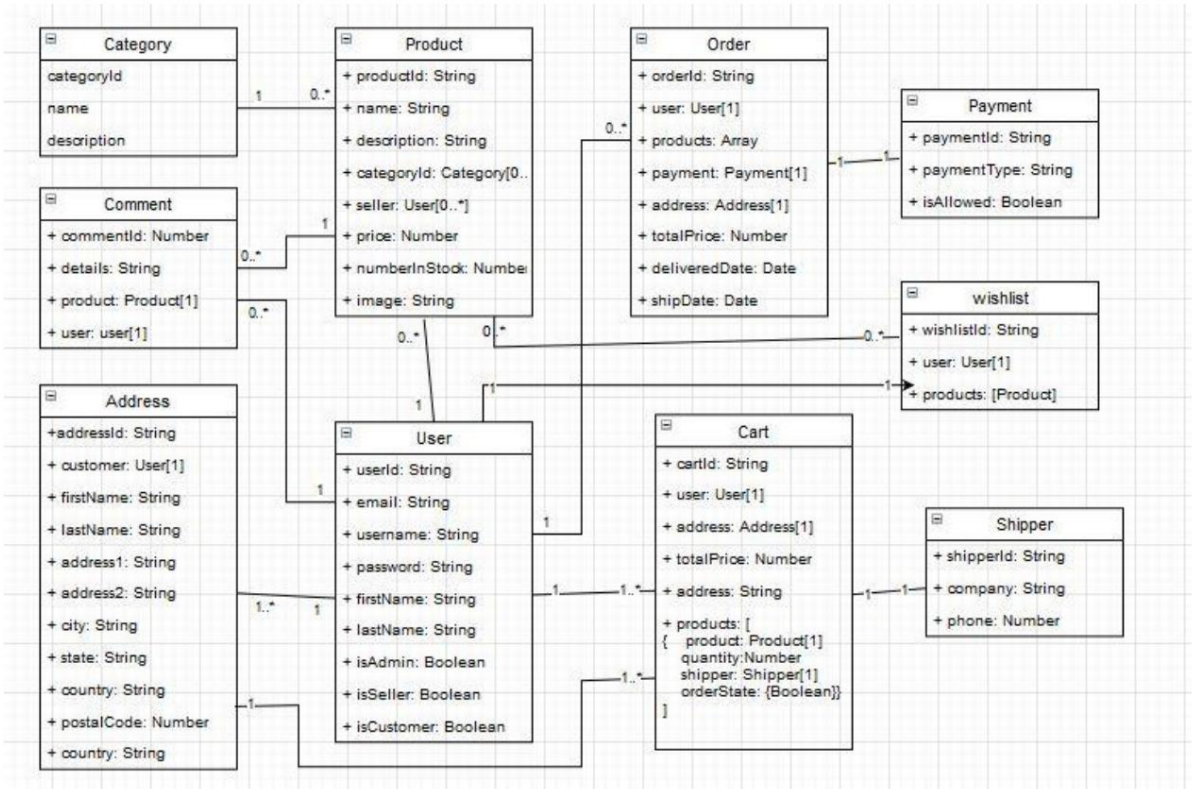
2.2. Dekomposisi Monolitik

Dekomposisi monolitik adalah tahap pemisahan modul yang tidak memiliki ketergantungan [15].
Pada fase ini, peneliti akan menggunakan dua pendekatan: Meta-data Aided (MDA) dan Static Code Analysis. (SCA). Untuk pendekatan MDA, peneliti akan menggunakan dokumen artefak yang tersedia di repositori GitHub dari proyek yang dipilih, termasuk Diagram Hubungan Entitas (ERD) dan spesifikasi aplikasi. Untuk pendekatan SCA, peneliti akan mengidentifikasi fungsi-fungsi dalam kode sumber dan kemudian memindai kode sumber menggunakan SonarQube untuk mengidentifikasi duplikasi kode.

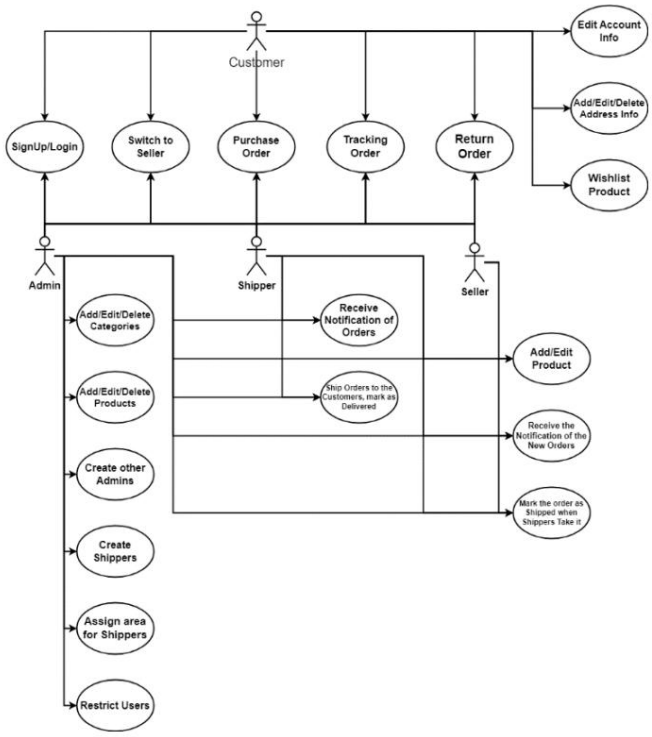
2.2.1. Pendekatan MDA

Meta-Data Aided (MDA) adalah pendekatan untuk melakukan refactoring aplikasi yang memanfaatkan sumber umum seperti diagram, spesifikasi, dan deskripsi aplikasi [3]. Dokumen seperti ERD dan persyaratan fungsional tersedia untuk digunakan dalam aplikasi yang dipilih. Selain dokumen-dokumen ini, peneliti juga menyertakan diagram Use Case sesuai dengan persyaratan yang ditemukan di repositori GitHub. ERD dan Use Case untuk aplikasi ini dijelaskan pada Gambar 2 *Diagram Entitas Relasi* dan Gambar 3 Diagram *Use Case* .

Berdasarkan diagram di atas dapat disimpulkan bahwa aplikasi dapat dibagi berdasarkan peran pengguna. Pembagian fungsi dan layanan didasarkan pada kasus penggunaan yang dilakukan oleh setiap jenis pengguna. Misalnya, layanan penjual hanya dapat menjalankan fungsi yang spesifik untuk penjual, dan layanan pengirim hanya dapat menjalankan fungsi yang relevan dengan pengirim. Meskipun pemisahan fungsi dan layanan telah dirumuskan, masih ada satu pendekatan lagi yang perlu dilakukan sebelum menyelesaikan pembagian layanan.



Gambar 2 *Diagram Relasi Entitas*



Gambar 3 Diagram *Kasus Penggunaan*

2.2.2. Pendekatan SCA

Pendekatan Analisis Kode Statis (SCA) melibatkan pengambilan data kode sumber dari aplikasi sebagai input untuk analisis [1], [14]. Oleh karena itu, peneliti menggunakan kode sumber dari aplikasi inventaris untuk analisis. Proses analisis melibatkan pemindaian kode sumber dengan SonarQube.

SonarQube adalah aplikasi yang membantu pengembang menganalisis kode sumber suatu aplikasi [19].

Berdasarkan hasil pemindaian, kategori yang perlu diperhatikan adalah bug dan duplikasi. Sebagian besar bug yang terdeteksi berasal dari kode CSS yang digunakan untuk penataan gaya, yang tidak berdampak signifikan pada kinerja aplikasi. Mengenai duplikasi, sekitar 56,4% ditemukan di folder Validasi, yang berisi file untuk validasi data. Hal ini dapat diabaikan karena kode yang diduplikasi digunakan berulang kali sesuai desain. Selain itu, terdapat 14% duplikasi kode di folder controller, dengan 21,9% berasal dari cartController.js dan 37% dari userController.js. Kode yang diduplikasi di cartController.js melibatkan blok yang bertanggung jawab untuk memperbarui total harga di keranjang belanja. Hal ini dapat diatasi dengan membuat fungsi updateTotalCartPrice(), yang dapat dipanggil oleh fungsi lain yang perlu memperbarui total harga.

A

Demikian pula, di userController.js, kode yang diduplikasi berkaitan dengan pembuatan kata sandi yang di-hash. Hal ini dapat diatasi dengan membungkus blok kode tersebut ke dalam fungsi baru bernama genHashedPassword().

Tabel 1 Daftar Pengontrol dari Aplikasi Monolitik

Modul	Metode
Alamat	Posting: TambahkanAlamat
	Masukkan : EditAddress
	Hapus : HapusAlamat
	Dapatkan : GetAddress
Keranjang	Masukkan : tambahkan ke keranjang
	Dapatkan: userCartInfo
	Dapatkan: removeFromCart
	Masukkan: ubahJumlahDariKeranjang
	Masukkan : chooseOrderAddress
Kategori	Dapatkan: categoryIndex
	Posting: buatKategori
	Dapatkan: detail kategori
	Hapus : hapusKategori
	Masukkan :updateCategory
Memesan	Dapatkan: orderSuccess
	Dapatkan: userOrderHistory
	Dapatkan: ordersToShip
	Dapatkan: pesanan dikirim
	Dapatkan : markAsShipped
	Dapatkan: ordersToDeliver
	Dapatkan : markAsDelivered
Izin	Dapatkan: getAllUsers
	Dapatkan: getAllShipers
	Masukkan: addShipper
	Masukkan: addShiperInfo
	Masukkan: addAdmin
	Masukkan: batasiPengguna
Produk	Dapatkan: semuaProduk
	Dapatkan : userProducts
	Postingan: buatProduk
	Dapatkan: detail produk
	Hapus: deleteProduct
	Postingan: updateProduk
Pengguna	Postingan: buatPengguna
	Postingan: login
	Dapatkan: getUser
	Masukkan: editUser
Daftar Keinginan	Dapatkan: addToWishlist
	Dapatkan: userWishlist
	Dapatkan: hapusDariDaftarKeinginan

Selain pemindaian dengan SonarQube, proses analisis juga akan meninjau penggunaan setiap fungsi dalam aplikasi. Hal ini akan membantu memisahkan modul-modul yang saling bergantung.

2.2.3. Dekomposisi Fungsi

Setelah proses analisis dilakukan menggunakan dua pendekatan sebelumnya, pemisahan fungsi dapat dilakukan berdasarkan hasil analisis yang diperoleh dari langkah-langkah sebelumnya. Pemisahan fungsi dan layanan didasarkan pada hasil analisis menggunakan pendekatan MDA dan SCA, yang telah membantu mengidentifikasi modul yang dapat dipisahkan dan dioptimalkan. Dengan pemisahan ini, diharapkan setiap layanan dapat beroperasi secara independen dan efisien, mendukung pengembangan aplikasi lebih lanjut menuju arsitektur microservices. Proses ini sangat penting untuk memastikan setiap fungsi beroperasi dengan beban minimal dan mengurangi ketergantungan antar layanan, yang pada akhirnya meningkatkan kinerja aplikasi secara keseluruhan.

2.3. Implementasi Microservices

Implementasi microservices merupakan tahap kunci dalam refactoring aplikasi monolitik. Pada tahap ini, pola desain yang digunakan akan sama dengan yang digunakan pada aplikasi asli, memastikan bahwa tidak ada faktor pola desain yang akan mengubah kinerja aplikasi. Berikut adalah pola desain yang akan digunakan dalam aplikasi microservices.

Layanan

```
ÿÿÿ pengontrol/
ÿ ÿÿÿ ···
ÿÿÿ middleware/
ÿ ÿÿÿ validasi/
ÿ ÿÿÿ ···
ÿÿÿ model/
ÿ ÿÿÿ ···
ÿÿÿ rute/
ÿ ÿÿÿ ···
ÿÿÿ index.js
ÿÿÿ paket*.json
```

Pemandangan

```
ÿÿÿ publik/
ÿ ÿÿÿ ···
ÿÿÿ src/
ÿ ÿÿÿ aset/
ÿ ÿ ÿÿÿ ÿ ···
ÿÿÿ Komponen/
ÿ ÿ ÿÿÿ pengaturan-akun/
ÿ ÿ ÿ ÿÿÿ ÿ ÿ ÿÿÿ ···
keranjang/
ÿ ÿ ÿ ÿÿÿ ÿ ÿ ···
ÿÿÿ dasbor/
ÿ ÿ ÿ ÿÿÿ ÿ ÿ ÿÿÿ ···
halaman beranda/
ÿ ÿ ÿ ÿÿÿ ÿ ÿ ···
ÿÿÿ login&signup/
ÿ ÿ ÿ ÿÿÿ ÿ ÿ ···
ÿÿÿ ÿ ÿÿÿ ···
redux/
ÿ ÿ ÿÿÿ tindakan/
ÿ ÿ ÿ ÿÿÿ ÿ ÿ ···
ÿÿÿ pereduksi/
ÿ ÿ ÿ ÿÿÿ ÿ ÿ ···
ÿÿÿ ÿ ÿÿÿ ···
gaya/
ÿ ÿ ÿÿÿ ÿ ···
ÿÿÿ app.js
ÿ ÿÿÿ index.js
ÿÿÿ paket*.json
```

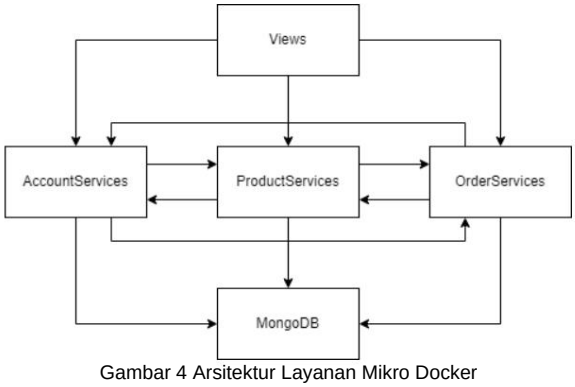
Proses refactoring dimulai dengan peneliti menduplikasi kode dari aplikasi monolitik ke setiap repositori sesuai dengan hasil dekomposisi sebelumnya. Peneliti juga menyiapkan basis data dan mengatur layanan penyimpanan gambar di Cloudinary untuk mendukung penyimpanan file terpisah.

Selanjutnya, peneliti melakukan penyesuaian pada beberapa fungsi yang diidentifikasi memiliki kode duplikat berdasarkan hasil pemindaian SonarQube. Fungsi yang memiliki ketergantungan pada fungsi lain di luar domain API-nya dimodifikasi untuk memanggil fungsi dari API lain, sesuai dengan prinsip kopling longgar [4]. Setelah penyesuaian ini selesai, peneliti melakukan pengujian ringan untuk memastikan bahwa semua layanan berfungsi dengan baik dan sesuai harapan. Pengujian ini mencakup pemeriksaan setiap endpoint API untuk memverifikasi integritas dan fungsionalitas keseluruhan aplikasi.

2.4. Kontainerisasi Aplikasi Monolitik dan Mikroservis

Setelah proses refactoring selesai, layanan-layanan tersebut perlu ditempatkan ke dalam kontainer Docker untuk memfasilitasi komunikasi antar layanan.

Kontainerisasi dimulai dengan pembuatan image untuk aplikasi dan basis data yang digunakan. Proses pembuatan image juga mempertimbangkan port yang digunakan oleh setiap API untuk menghindari konflik saat aplikasi dijalankan. Kontainerisasi tidak hanya diterapkan pada aplikasi microservices tetapi juga pada aplikasi monolitik untuk menempatkan kedua aplikasi tersebut dalam lingkungan yang sama.



Gambar 4 Arsitektur Layanan Mikro Docker

```
DARI node:16
WORKDIR /usr/src/app
Salin file package*.json ./
JALANKAN npm install
SALIN . .
UNGKAP 8081
CMD ["npm", "run", "start"]
```

Gambar 5 Konfigurasi Dockerfile untuk Layanan Mikro API

```
DARI node:16 sebagai build
WORKDIR /usr/src/app
Salin file package*.json ./
JALANKAN npm install
SALIN . .
JALANKAN npm run build
DARI nginx:alpine
SALIN --dari=build /usr/src/app/build /usr/share/nginx/html
Salin file nginx.conf ke /etc/nginx/nginx.conf
UNGKAPKAN 3000
CMD ["nginx", "-g", "daemon off;"]
```

Gambar 6 Konfigurasi Dockerfile untuk Layanan Mikro Views

Merujuk pada Gambar 4, , Dockerfile diperlukan untuk buat image untuk setiap API dan View untuk Frontend. Semua API akan berjalan di lingkungan Node versi 16, dan kode akan disalin ke direktori yang ditentukan di WORKDIR. Proses duplikasi dimulai dengan file package*.json, karena file-file ini akan digunakan pertama kali untuk menginstal dependensi library yang dibutuhkan oleh API, diikuti dengan perintah RUN npm install. Setelah selesai, kode dapat disalin, dengan port diekspos menggunakan perintah EXPOSE, dan image akan dijalankan menggunakan perintah CMD ["npm", "run", "start"].

Namun, terdapat perbedaan antara Dockerfile untuk API dan Views. Untuk Views, aplikasi akan dibangun terlebih dahulu, kemudian output pembangunan akan disalin ke lingkungan baru yang berjalan di nginx:alpine. Peneliti juga membuat konfigurasi Nginx untuk pengaturan server dan untuk mengarahkan permintaan klien ke API backend.

Setelah semua Dockerfile dibuat, langkah selanjutnya adalah membangun Dockerfile tersebut menjadi kontainer. Pembuatan image dapat dilakukan satu per satu, namun proses ini bisa memakan waktu cukup lama [5]. Oleh karena itu, diperlukan file docker-compose untuk mengkonsolidasikan semua Dockerfile ke dalam satu konfigurasi dan membangunnya secara bersamaan. Berikut adalah contoh file docker-compose yang digunakan.

Dalam docker-compose, setiap image dikonfigurasi sesuai dengan persyaratan setiap API, seperti image sumber yang akan dibangun, nama kontainer, port, dan variabel lingkungan. File docker-compose juga menyertakan image MongoDB untuk mengintegrasikannya ke dalam kontainer. Dengan cara ini, peneliti dapat menyebarkan semua image yang telah dibuat ke dalam kontainer masing-masing dengan satu perintah.

```
versi: '3.8'

Layanan:
  mongodb:
    gambar: mongo:terbaru
    nama_kontainer: mongodb
    pelabuhan:
      - "27017:27017"
    volume:
      - mongo-data:/data/db

  layanan akun:
    membangun:
      konteks: ./AccountService
    nama_kontainer: layanan-akun
    pelabuhan:
      - "8080:8080"
    lingkungan:
      -

  layanan pemesanan:
    membangun:
      konteks: ./OrderService
    nama_kontainer: layanan-pesanan
    pelabuhan:
      - "8082:8082"
    lingkungan:
      -

  produk-layanan:
    membangun:
      konteks: ./ProductService
    nama_kontainer: layanan-produk
    pelabuhan:
      - "8081:8081"
    lingkungan:
      -

  antarmuka pengguna (frontend):
    membangun:
      konteks: ./views
    nama_kontainer: frontend
    pelabuhan:
      - "3000:3000"
    lingkungan:
      -
```

Gambar 7 Konfigurasi Docker-Compose untuk Layanan Mikro

```
DARI node:14 SEBAGAI build
WORKDIR /app
Salin file package*.json ./
JALANKAN npm install
SALIN . .
SALIN .env ./views
JALANKAN npm install --prefix views
JALANKAN npm run build --prefix views
DARI nginx:alpine
SALIN --dari=build /app/views/build /usr/share/nginx/html
SALIN --dari=build /app /app
Salin file nginx.conf ke /etc/nginx/conf.d/default.conf
WORKDIR /app
JALANKAN apk add --no-cache nodejs npm
SALIN start.sh /start.sh
JALANKAN chmod +x /start.sh
TERUNGKAP 2024
UNGKAPKAN 5000
LINGKUNGAN PORT=5000
Perintah ["/start.sh"]
```

Gambar 8 Konfigurasi Dockerfile untuk Aplikasi Monolitik

Berbeda dengan aplikasi microservices, aplikasi monolitik hanya menggunakan satu Dockerfile. Aplikasi monolitik dijalankan dalam lingkungan yang menggunakan

Node.js versi 14. Hal ini dilakukan agar sesuai dengan versi dependensi pustaka yang ada di aplikasi.
Konfigurasi Dockerfile untuk aplikasi monolitik identik dengan Dockerfile yang digunakan untuk aplikasi microservices. Namun, dalam Dockerfile untuk aplikasi monolitik, perlu menambahkan Node.js dan npm ke lingkungan nginx:alpine untuk menjalankan program backend dengan perintah RUN apk add --no-cache nodejs npm. Image kemudian dijalankan dengan perintah CMD ["/start.sh"].

```
npm run start &
nginx -g 'daemon off;'
```

Gambar 9 Konfigurasi file start.sh

```
versi: '3.8'
Layanan:
  aplikasi:
    membangun: .
    pelabuhan:
      - "2024:2024"
      - "5000:5000"
    lingkungan:
      -
    volume:
      - .:/app
    bergantung pada:
      - db

  db:
    gambar: mongo:terbaru
    pelabuhan:
      - "27019:27017"
    volume:
      - mongo-data:/data/db
```

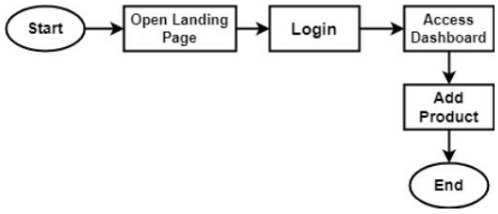
Gambar 10 Konfigurasi Docker-Compose untuk Monolitik

Setelah membuat Dockerfile, peneliti kemudian membuat file docker-compose untuk menyebarkan image ke dalam kontainer. Image yang dikonfigurasi dalam file docker-compose hanya mencakup aplikasi monolitik dan MongoDB.

2.5. Pengujian

Pengujian diperlukan untuk menilai dampak refactoring guna memahami perbedaan kinerja sebelum dan sesudah refactoring. Pengujian akan dilakukan melalui pengujian beban menggunakan JMeter untuk mengamati waktu respons aplikasi. Pengujian ini akan melibatkan dua tahap: pertama adalah pembuatan fungsi pengujian, dan kedua adalah pengujian aplikasi itu sendiri.

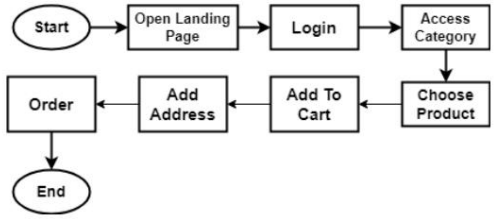
Dalam pembuatan rencana pengujian, peneliti melakukan pengujian dalam dua skenario: Permintaan Tunggal dan Permintaan Grup. Permintaan Tunggal hanya melibatkan pengujian satu permintaan per pengujian, menggunakan fungsi Login dan Tambah ke Keranjang. Sementara itu, Permintaan Grup dilakukan dengan merekam aktivitas pengguna di dalam aplikasi, sehingga urutan permintaan yang dilakukan selama aktivitas pengguna akan direkam. Ada dua jenis pengguna yang diuji dalam Permintaan Grup: pelanggan dan penjual. Alur aktivitas yang dilakukan dapat dilihat pada Gambar 11 dan



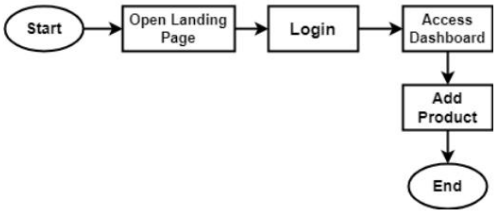
Gambar 12 ·

Tabel 2 Kasus Pengujian

Pengujian Kasus	Fungsi
Permintaan Tunggal	Login
	Tambahkan ke Keranjang
Permintaan Grup	Pelanggan
	Penjual



Gambar 11 Alur Pengujian Skenario Pelanggan



Gambar 12 Alur Pengujian Skenario Penjual

Konfigurasi yang digunakan di JMeter melibatkan 300 pengguna thread, periode ramp-up 6 detik, dan jumlah loop 1. Tes ini bertujuan untuk mensimulasikan beban 300 pengguna yang mengakses aplikasi secara bersamaan selama periode ramp-up singkat untuk mengevaluasi kemampuan aplikasi dalam menangani beban tinggi. Hasil tes ini memberikan wawasan tentang perbedaan beban yang dialami oleh arsitektur monolitik dan microservices, yang ditunjukkan oleh nilai waktu respons. Data waktu respons yang dikumpulkan hanya akan menggunakan waktu respons rata-rata untuk setiap 50 pengguna dan keseluruhan tes untuk setiap kasus uji.

3. HASIL

Setelah melakukan proses riset yang meliputi pencarian proyek, dekomposisi monolitik, implementasi microservices, containerisasi aplikasi, dan pengujian, diperoleh hasil sebagai berikut:

3.1. Hasil Pencarian Proyek

Dari pencarian proyek GitHub, para peneliti berhasil mengidentifikasi sebuah repositori yang memenuhi kriteria penelitian, yaitu aplikasi inventaris yang dikembangkan oleh Mohamed Eleshmawy. Proyek ini menggunakan JavaScript sebagai bahasa pemrogramannya dan mencakup artefak dokumentasi seperti Diagram Hubungan Entitas (ERD) dan Analisis Fungsional.

Persyaratan. Dokumen-dokumen ini berfungsi sebagai dasar bagi proses dekomposisi monolitik yang akan dilakukan.

3.2. Hasil Dekomposisi Monolitik

3.2.1. Hasil Pendekatan MDA

Pendekatan Meta-Data Aided (MDA) dilakukan dengan menganalisis dokumen artefak yang tersedia. Analisis Diagram Hubungan Entitas (ERD) dan Diagram Kasus Penggunaan mengidentifikasi pemisahan fungsi berdasarkan peran pengguna, seperti yang ditunjukkan pada

Tabel 3 .

Tabel 3 Pemisahan fungsi dan layanan berdasarkan MDA

Peran	Kasus Penggunaan
Admin	Login Akun, Pendaftaran, Edit Akun, Tambah/Edit/Hapus Alamat, Tukar Peran
	Tambah/Edit/Hapus Kategori, Tambah/Edit/Hapus Produk, Buat Admin, Buat Pengirim, Tetapkan Area untuk Pengirim, Batasi Pengguna
Penjual	Pencarian Produk Pelanggan, Daftar Keinginan Produk, Pembelian Pesanan, Pelacakan Pesanan, Pengembalian Pesanan, Ulasan
	Tambah/Edit Produk, Terima Notifikasi dari Pelanggan, Ubah Status menjadi Terkirim
Pengirim	Terima Notifikasi dari Penjual, Ubah status untuk dikirim

Pemisahan fungsi dan layanan mempermudah pengembangan layanan dalam arsitektur microservices.

3.2.2. Hasil Pendekatan MDA

Pendekatan Analisis Kode Statis (SCA) dilakukan menggunakan SonarQube, yang menghasilkan deteksi masalah seperti yang ditunjukkan pada

Tabel 4 .

Tabel 4 Hasil pemindaian SonarQube

Hasil Kategori	
Serangga	4
Kerentanan	0
Hotspot Keamanan	7
Code Smells	142
Duplikasi	12,2%

Analisis ini menunjukkan bahwa duplikasi kode paling banyak terjadi di folder Validasi dan controller. Duplikasi kode yang signifikan diidentifikasi dalam file cartController.js dan userController.js. Untuk mengatasi masalah ini, fungsi-fungsi dengan kode yang berduplikasi, seperti updateTotalCartPrice() dan genHashedPassword(), dikembangkan untuk mengurangi redundansi dan meningkatkan efisiensi.

3.2.3. Dekomposisi Fungsi

Berdasarkan hasil MDA dan SCA, pemisahan fungsi dilakukan dengan pemisahan seperti yang ditunjukkan pada **Error! Reference source not found..**

Tabel 5 Pemisahan Fungsi dan Layanan

Nama Layanan	Metode	Keterangan	Pengontrol	Meja
Layanan Akun	AMBIL/TUANG/POSTING	Akun Pengelolaan	Pengguna, Alamat, Daftar Keinginan, Izin	Pengguna, Alamat, Daftar Keinginan
Layanan Pemesanan	AMBIL/LETAKKAN	Manajemen Pesanan	Pesan, Keranjang	Pesanan, Keranjang, Pembayaran, Pengirim
ProdukLayanan	GET/PUT/POST/DELETE	Produk Pengelolaan	Produk, Kategori	Produk, Kategori, Komentar
Pemandangan		Tampilan FrontEnd		

Pemisahan ini memisahkan setiap fungsi berdasarkan domain dan tanggung jawabnya, mendukung prinsip kopling longgar dan memfasilitasi pengembangan serta pemeliharaan aplikasi.

3.3. Hasil Implementasi Microservices

Implementasi microservices melibatkan duplikasi kode dari aplikasi monolitik ke dalam repositori layanan individual. Penyesuaian kode dilakukan untuk mengatasi masalah duplikasi yang terdeteksi oleh SonarQube. Fungsi-fungsi yang saling terkait dimodifikasi untuk menggunakan API lain sesuai dengan prinsip kopling longgar (loose coupling). Pengujian awal memastikan bahwa semua layanan berfungsi dengan benar dan memenuhi harapan.

3.4. Hasil dari Kontainerisasi

Penggunaan kontainer diterapkan untuk kedua arsitektur aplikasi, yaitu microservices dan monolitik.

.

3.4.1. Kontainerisasi Aplikasi Layanan Mikro

Proses kontainerisasi untuk aplikasi microservices melibatkan pembuatan image Docker untuk setiap API dan komponen frontend. Dockerfile untuk API mengkonfigurasi port dan dependensi aplikasi, sementara Dockerfile untuk frontend membangun aplikasi dan menyiapkan Nginx. Docker Compose digunakan untuk mengelola dan membangun semua image secara bersamaan, termasuk MongoDB. Pengaturan ini memastikan bahwa semua komponen dikontainerisasi dan dapat berinteraksi dengan lancar dalam lingkungan yang terpadu.

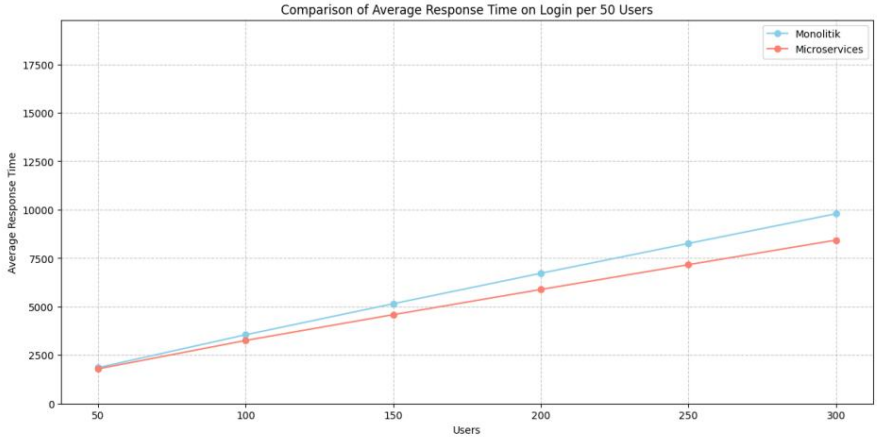
3.4.2. Kontainerisasi Aplikasi Monolitik

Untuk aplikasi monolitik, Dockerfile disesuaikan agar sesuai dengan versi Node.js yang digunakan pada aplikasi sebelumnya. Docker Compose mengelola kontainer untuk aplikasi monolitik dan

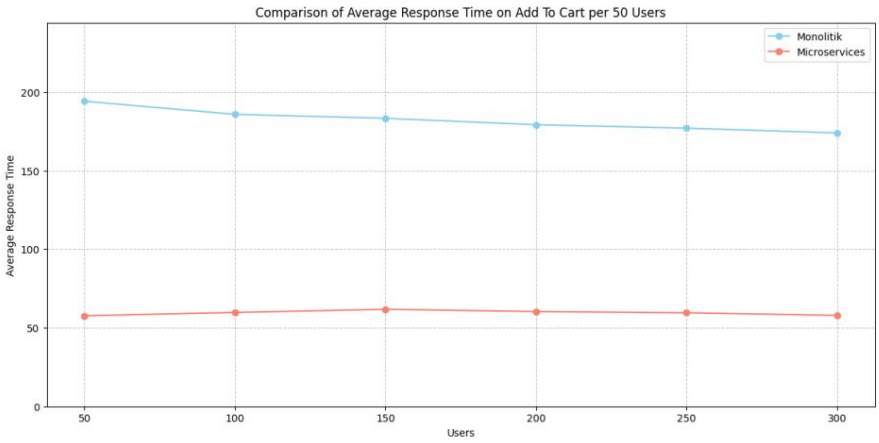
MongoDB, memastikan lingkungan pengembangan yang konsisten dan selaras dengan pengaturan microservices.

3.5. Hasil Pengujian Permintaan Tunggal

Pengujian permintaan tunggal dilakukan untuk mengevaluasi apakah proses refactoring secara signifikan memengaruhi proses individual. Permintaan yang diuji adalah untuk fungsi Login dan Tambah ke Keranjang. Fungsi login sering digunakan oleh pengguna saat mengakses aplikasi tetapi tidak berinteraksi dengan API lain selama prosesnya. Sebaliknya, fungsi Tambah ke Keranjang memanggil API lain, yaitu ProductServices, untuk memeriksa ketersediaan item dalam basis data. Pengaturan ini diharapkan akan menghasilkan hasil yang berbeda ketika pengujian dijalankan. Hasil pengujian permintaan tunggal dirinci dalam gambar dan tabel berikut.



Gambar 13 Perbandingan Waktu Respons Rata-rata untuk Fungsi Login



Gambar 14 Perbandingan Waktu Respons Rata-rata untuk Fungsi Tambah ke Keranjang

Merujuk pada Gambar 13 dan Gambar 14, terlihat jelas bahwa aplikasi arsitektur microservices memiliki waktu respons rata-rata yang lebih rendah dibandingkan dengan aplikasi monolitik. Namun, perbedaan waktu respons rata-rata antara skenario login dan tambah ke keranjang bervariasi. Seperti yang telah dijelaskan sebelumnya, fungsi tambah ke keranjang dalam arsitektur microservices melibatkan pemanggilan API lain, yang mengakibatkan beban yang lebih kecil pada API yang menangani fungsi tambah ke keranjang dibandingkan dengan aplikasi monolitik. Sebaliknya, fungsi login tidak melibatkan panggilan ke API lain. Hal ini mengakibatkan perbedaan waktu respons rata-rata yang lebih signifikan untuk skenario tambah ke keranjang dibandingkan dengan skenario login.

Tabel 6 Data Rata-rata Keseluruhan untuk Fungsi Login

Permintaan Login	**Mikroservis**	**Monolitik**
Waktu Respons Rata-rata (ms)	9791,89	8432.576
Waktu Respon Minimum (ms)	81	78
Waktu Respons Maksimum (ms)	18870	15843

Tabel 7 Data Rata-rata Keseluruhan untuk Fungsi Tambah ke Keranjang

Permintaan Tambah ke	**Mikroservis**	**Monolitik**
Keranjang Waktu Respons Rata-rata (ms)	174.013	57.883
Waktu Respon Minimum (ms)	24	20
Waktu Respons Maksimum (ms)	358	108

Tabel 8 Perbandingan Waktu Respons Rata-Rata untuk Permintaan Tunggal

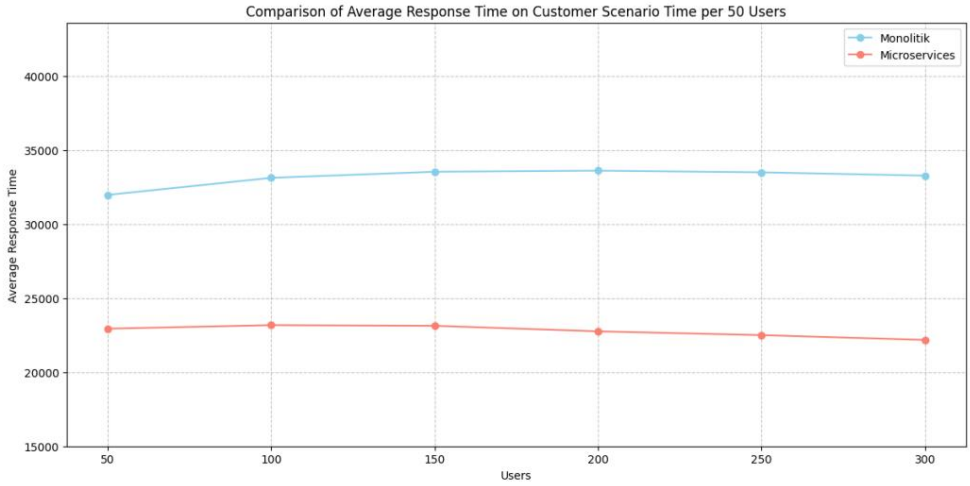
Fungsi	**Mikroservis**	**Monolitik**	**%**
Login	9791.89	8432.576	16,12%
Tambahkan ke Keranjang	174.013	57.883	200,63%

Data dari Tabel 6 dan Tabel 7 juga menunjukkan perbedaan signifikan antara aplikasi monolitik dan aplikasi microservices, dan Tabel 8 memperkuat hal tersebut.

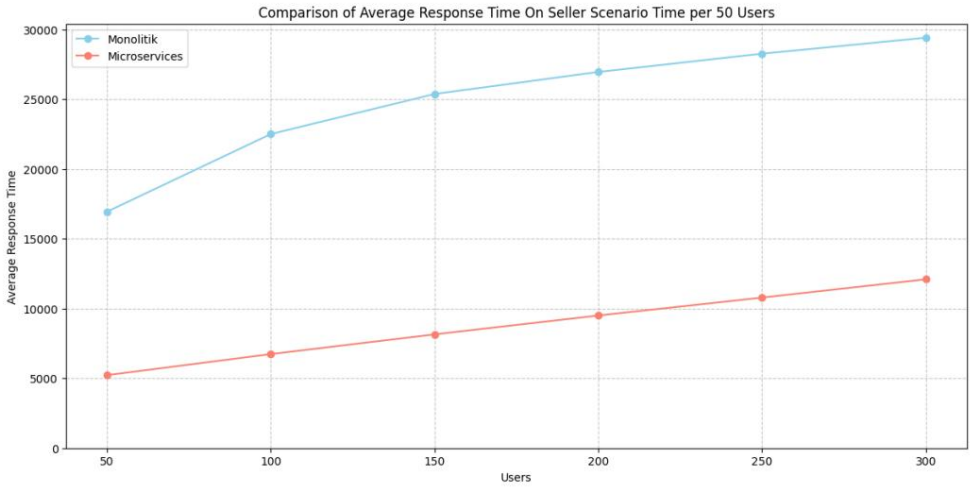
Komunikasi antar-API sangat memengaruhi waktu respons.

3.6. Hasil Pengujian Permintaan Kelompok

Pengujian permintaan kelompok bertujuan untuk mengevaluasi dampak perbedaan arsitektur aplikasi terhadap pengguna. pengalaman selama penggunaan aplikasi. Oleh karena itu, pengujian ini memeriksa beberapa permintaan secara bersamaan, yang mewakili aktivitas pengguna di dalam aplikasi.
Hasil pengujian permintaan kelompok dirinci dalam gambar dan tabel berikut.



Gambar 15 Perbandingan Waktu Respons Rata-rata untuk Skenario Pelanggan



Gambar 16 Perbandingan Waktu Respons Rata-rata untuk Skenario Penjual

Seperti yang terlihat pada Gambar 15 dan Gambar 16, arsitektur microservices menunjukkan perbedaan signifikan dalam waktu respons. Dalam skenario pelanggan, waktu respons rata-rata tampak relatif stabil untuk kedua arsitektur. Namun, dalam skenario penjual, waktu respons rata-rata cenderung meningkat seiring bertambahnya jumlah pengguna.

Tabel 9 Data Rata-rata Keseluruhan untuk Skenario Pelanggan

Permintaan Kasus	**Mikroservis**	**Monolitik**
Pengguna Waktu Respons Rata-rata (ms)	33285.577	22182.86
Waktu Respon Minimum (ms)	20257	6362
Waktu Respons Maksimum (ms)	34492	24034

Tabel 10 Data Rata-rata Keseluruhan untuk Skenario Penjual

Permintaan Tambah ke	**Mikroservis**	**Monolitik**
Keranjang Waktu Respons Rata-rata (ms)	29395,44	12091.293
Waktu Respon Minimum (ms)	10631	2290
Waktu Respons Maksimum (ms)	46081	32150

Tabel 11 Perbandingan Waktu Respons Rata-rata untuk Permintaan Grup

Skenario	**Mikroservis**	**Monolitik**	**%**
Pelanggan	33285.577	22182.86	50,05%
Penjual	29395.44	12091.293	143,11%

Data pada Tabel 9 dan Tabel 10 menunjukkan bahwa waktu respons rata-rata untuk aplikasi dengan arsitektur microservices masih memiliki keunggulan signifikan saat menjalankan skenario yang umum bagi berbagai jenis pengguna. Selain itu, Tabel 11
Hasil penelitian menunjukkan bahwa perbedaan waktu respons untuk skenario penjual lebih tinggi, sekitar 143,11%, antara arsitektur monolitik dan mikroservis, sedangkan pada skenario pelanggan, perbedaannya sekitar 50,05%.

4. DISKUSI

Pengujian aplikasi dengan arsitektur monolitik dan mikroservis menunjukkan bahwa arsitektur mikroservis secara konsisten memberikan waktu respons yang lebih rendah dibandingkan dengan arsitektur monolitik, baik dalam skenario permintaan tunggal maupun permintaan kelompok. Dalam pengujian permintaan tunggal, fungsi login dan tambah ke keranjang menunjukkan dampak nyata dari panggilan antar-API pada kinerja sistem. Fungsi tambah ke keranjang, yang melibatkan interaksi antara API OrderServices dan ProductServices, menunjukkan peningkatan efisiensi dalam arsitektur mikroservis dibandingkan dengan arsitektur monolitik. Hal ini disebabkan oleh kemampuan mikroservis untuk mendistribusikan beban kerja di berbagai layanan yang terpisah.

Dalam pengujian permintaan grup, hasilnya menunjukkan bahwa arsitektur microservices lebih efektif dalam menangani beban yang kompleks dan berskala besar, terutama selama lonjakan permintaan. Waktu respons rata-rata yang lebih stabil menunjukkan kemampuan arsitektur untuk mengelola beragam permintaan pengguna dengan lebih efisien.
Penelitian sebelumnya menunjukkan bahwa arsitektur microservices dapat mengoptimalkan alur kerja dengan membagi tugas menjadi layanan yang lebih kecil dan saling terhubung, sehingga menghasilkan peningkatan kinerja [6]. Selain itu, analisis aktivitas penjual mengungkapkan bahwa microservices lebih efisien dalam memproses operasi yang membutuhkan komputasi intensif. Metodologi yang digunakan dalam penelitian ini menggunakan pendekatan MDA dan SCA, yang memungkinkan identifikasi modul yang dapat diuraikan dan dioptimalkan. Namun, keterbatasan metode ini adalah tidak memungkinkan pengembang untuk melakukan refactoring dengan logika program yang mereka rancang sendiri, sehingga mencegah optimasi kompleksitas waktu eksekusi melalui perubahan logika program [20].

Implementasi microservices melalui containerisasi semakin meningkatkan isolasi dan manajemen layanan.

Secara keseluruhan, hasil penelitian ini menunjukkan bahwa transisi dari arsitektur monolitik ke arsitektur microservices memiliki dampak positif yang signifikan terhadap kinerja aplikasi. Studi ini berkontribusi pada pemahaman yang lebih mendalam tentang bagaimana arsitektur microservices dapat diterapkan untuk meningkatkan efisiensi dan responsivitas sistem, serta memberikan panduan praktis bagi pengembang yang mempertimbangkan untuk melakukan refactoring aplikasi mereka. Namun, penting untuk dicatat bahwa implementasi microservices juga membutuhkan pertimbangan yang cermat terhadap pengelolaan kompleksitas antar-layanan dan pemeliharaan infrastruktur yang lebih terdistribusi.

5. KESIMPULAN

Studi ini berhasil menunjukkan bahwa transisi dari arsitektur monolitik ke arsitektur microservices memiliki dampak positif yang signifikan terhadap kinerja aplikasi, khususnya dalam hal waktu respons. Hasil pengujian menunjukkan bahwa Arsitektur microservices secara konsisten memberikan waktu respons yang lebih rendah dibandingkan dengan arsitektur monolitik, baik dalam skenario permintaan tunggal maupun permintaan kelompok.

Keunggulan microservices dalam mendistribusikan beban kerja dan mengelola komunikasi antar-layanan memungkinkannya untuk mengatasi hambatan yang sering ditemui pada sistem monolitik. Temuan ini sejalan dengan literatur yang ada, yang menunjukkan bahwa microservices dapat meningkatkan skalabilitas dan fleksibilitas sistem, serta memberikan manfaat dalam hal manajemen dan pemeliharaan perangkat lunak dalam jangka panjang [5], [15].

Implementasi refactoring menggunakan pendekatan Meta-Data Aided (MDA) dan Static Code Analysis (SCA) memungkinkan identifikasi dan pemisahan modul yang tepat, yang pada praktiknya meningkatkan efisiensi operasional dan kinerja aplikasi. Namun, metode ini tidak membahas optimasi kompleksitas waktu eksekusi dalam pendekatannya. Penggunaan Docker untuk mengemas layanan lebih lanjut mendukung isolasi dan manajemen layanan yang lebih baik, sejalan dengan prinsip desain kopling longgar yang mendasari arsitektur microservices.

Meskipun arsitektur microservices menawarkan banyak keuntungan, implementasinya memerlukan pertimbangan cermat terkait koordinasi layanan dan pemeliharaan infrastruktur yang lebih kompleks. Oleh karena itu, sangat penting bagi pengembang untuk mempertimbangkan kebutuhan dan konteks spesifik aplikasi mereka sebelum memutuskan untuk beralih ke arsitektur ini.

Secara keseluruhan, penelitian ini memberikan wawasan berharga bagi para pengembang yang mempertimbangkan transisi ke arsitektur microservices dan menawarkan panduan praktis untuk refactoring yang efektif. Hasil ini diharapkan dapat mendorong penelitian lebih lanjut di masa mendatang untuk mengeksplorasi strategi terbaik dalam mengatasi tantangan terkait dengan implementasi microservices dan untuk meningkatkan kinerja serta efisiensi sistem.

REFERENSI

[1] Y. Wei, Y. Yu, M. Pan, dan T. Zhang, “Pendekatan Tabel Fitur untuk menguraikan aplikasi monolitik menjadi layanan mikro,” dalam *Seri Prosiding Konferensi Internasional ACM,* Asosiasi untuk Mesin Komputasi, Nov. 2020, hlm. 21–30. doi: 10.1145/3457913.3457939.

[2] F. Ponce, G. Márquez, dan H. Astudillo, “Migrasi dari arsitektur monolitik ke layanan mikro: Tinjauan Singkat,” *Konferensi Internasional ke-38 Masyarakat Ilmu Komputer Chili (SCCC) 2019,* hlm. 1–7, 2019, doi: 10.1109/SCCC49216.2019.8966423.

[3] BJ Široký, “Dari Monolit ke Layanan Mikro: Pola Refactoring,” 2021.

[4] J. Fritzsch, J. Bogner, A. Zimmermann, dan

S. Wagner, “Dari monolit ke layanan mikro: Klasifikasi pendekatan refactoring,” dalam *Lecture Notes in Computer Science (termasuk subseri Lecture Notes in Artificial Intelligence dan Lecture Notes in Bioinformatics),* Springer Verlag, 2019, hlm. 128–141. doi: 10.1007/978-3-030-06019-

0_10.

[5] Q. Ren dan S. Li, “Metode Refactoring Monolit menjadi Layanan Mikro,” *Jurnal Perangkat Lunak,* vol. 13, no. 12, hlm. 646–653, Des. 2018, doi: 10.17706/jsw.13.12.646-653.

[6] P. Zaragoza, AD Seriai, A. Seriai, HL Bouziane, A. Shatnawi, dan M. Derras, “Refactoring kode sumber berorientasi objek monolitik untuk mewujudkan arsitektur berorientasi layanan mikro,” dalam *Prosiding Konferensi Internasional ke-16 tentang Teknologi Perangkat Lunak, ICSOFT 2021,* SciTePress, 2021, 78–89. doi: hlm. 10.5220/0010557800780089.

[7] M. Kalske, N. Mäkitalo, dan T. Mikkonen, “Tantangan Saat Beralih dari Arsitektur Monolit ke Arsitektur Mikroservis,” dalam *Lecture Notes in Computer Science (termasuk subseri Lecture Notes in Artificial Intelligence dan Lecture Notes in Bioinformatics),* Springer Verlag, 2018, hlm. 32–47. doi: 10.1007/978-3-

319-74433-9_3.

[8] L. Traini *dkk.,* “Bagaimana Refactoring Perangkat Lunak Mempengaruhi Waktu Eksekusi,” *ACM Transactions on Software Engineering and Methodology,* vol. 31, no. 2, Apr. 2022, doi: 10.1145/3485136.

[9] A. Almogahed, M. Omar, dan NH Zakaria, “Dampak Refactoring Perangkat Lunak terhadap Kualitas Perangkat Lunak di Lingkungan Industri: Tinjauan Studi Empiris,” 2018. [On line]. Tersedia di: http://www.kmice.cms.net.my/

[10] ISO/IEC 25010, “ISO/IEC 25010:2011 Rekayasa sistem dan perangkat lunak — Persyaratan dan Evaluasi Kualitas Sistem dan Perangkat Lunak (SQuaRE) — Model kualitas sistem dan perangkat lunak,” 2011

[11] RB Khan, “Studi Perbandingan Alat Pengujian Kinerja: Apache JMeter dan HP LoadRunner,” 2016, [Online].

Tersedia di: www.bth.se

[12] D. Al Fansha, M. Yusril, H. Setyawan, dan MN Fauzan, “Load Test pada Microservice yang menerapkan CQRS dan Event Sourcing,” *Jurnal Buana Informatika,* vol. 12, tidak. 2, hal. 126–134, 2021, doi: https://doi.org/10.24002/jbi.v12i2.4749.

[13] Q. Gu, S. Wagner, dan J. Fritzsch, “Pendekatan Meta untuk Memandu Refactoring Arsitektur dari Aplikasi Monolitik ke Layanan Mikro,” 2020, http://dx.doi.org/ doi: 10.18419/opus-11501.

[14] J. Zhao dan K. Zhao, “Menerapkan Refactoring Microservice pada Sistem Warisan Berorientasi Objek,” dalam *Prosiding - Konferensi Internasional ke-8 tentang Sistem Andal dan Aplikasinya, DSA 2021,* Institute of Electrical and Electronics Engineers Inc., 2021, hlm. 467–473. doi: 10.1109/DSA52907.2021.00070.

[15] N. Goncalves, D. Faustino, AR Silva, dan M. Portela, “Modularisasi Monolit Menuju Layanan Mikro: Refactoring dan Kompromi Kinerja,” dalam *Prosiding - 2021 IEEE 18th International Conference on Software Architecture Companion, ICSA-C 2021,* Institute of Electrical and Electronics Engineers Inc., Mar. 2021, pp. 54–61. doi: 10.1109/ICSA-C52384.2021.00015.

[16] J. Han, S. Deng, X. Xia, D. Wang, dan J. Yin, “Karakterisasi dan prediksi proyek populer di GitHub,” dalam *Prosiding - Konferensi Perangkat Lunak dan Aplikasi Komputer Internasional,* IEEE Computer Society, Juli 2019, hlm. 21–26. doi: 10.1109/COMPSAC.2019.00013.

[17] L. Baresi dan M. Garriga, “Microservices: Evolusi dan kepunahan layanan web?,” dalam *Microservices: Science and Engineering,* Springer International Publishing, 2019, hlm. 3–28. doi: 10.1007/978-3-030-31646-4_1.

[18] Mohamed Eleshmawy, “inventarisasi-aplikasi.” Diakses: 19 Maret 2024. [On line]. Tersedia di: https://github.com/moelashmawy/inventory-application

[19] D. Marcilio, R. Bonifacio, E. Monteiro, E. Canedo, W. Luz, dan G. Pinto, “Apakah pelanggaran analisis statis benar-benar diperbaiki? Tinjauan lebih dekat pada penggunaan realistis Sonarqube,” dalam *Konferensi Internasional IEEE tentang Pemahaman Program,* IEEE Computer Society, Mei 2019, 209–219. doi: hlm. 10.1109/ICPC.2019.00040.

[20] Dr. D. De Silva, P. Samarasekera, dan H. Ridmi, “Analisis Komparatif Teknik Analisis Kode Statis dan Dinamis,” 2023, doi: 10.36227/techrxiv.22810664.v1.