PERBANDINGAN EFISIENSI REST API DAN GRAPHQL PADA SISTEM PENJADWALAN PERKULIAHAN BERBASIS CLOUD-NATIVE DENGAN DOCKER, KUBERNETES, DAN LOAD BALANCING

PROPOSAL PENELITIAN AWAL

Tugas Mata Kuliah Metodologi Penelitian (IF25-41029)
Program Studi Teknik Informatika
Institut Teknologi Sumatera

Oleh:
Reyhan Capri Moraga
123140022

PROGRAM STUDI TEKNIK INFORMATIKA
INSTITUT TEKNOLOGI SUMATERA
2025

DAFTAR ISI

BAB I
PENDAHULUAN

1.1 Latar Belakang

Transformasi digital di sektor pendidikan tinggi telah mendorong peningkatan ekspektasi terhadap ketersediaan dan performa layanan akademik. Sistem informasi krusial, seperti sistem penjadalan perkuliahan, dituntut untuk tidak hanya fungsional tetapi juga harus skalabel dan responsif, terutama saat menghadapi lonjakan lalu lintas data (traffic) secara tiba-tiba, seperti pada masa pengisian Rencana Studi Mahasiswa (RSM) [1]. Untuk menjawab tantangan ini, arsitektur aplikasi modern telah beralih dari sistem monolitik menuju arsitektur cloud-native yang memanfaatkan teknologi seperti kontainerisasi (Docker) dan orkestrasi (Kubernetes) untuk mencapai skalabilitas elastis dan ketahanan sistem (resilience) [2]. Dalam arsitektur ini, mekanisme load balancing menjadi krusial untuk mendistribusikan lalu lintas data secara merata ke berbagai copy layanan (kontainer), memastikan tidak ada satu layanan pun yang kelebihan beban dan meningkatkan ketersediaan layanan [3].

Inti dari sistem terdistribusi ini adalah Application Programming Interface (API), yang berfungsi sebagai jembatan komunikasi antar layanan. Selama bertahun-tahun, REST (Representational State Transfer) telah menjadi standar de-facto dalam perancangan API karena kesederhanaan dan adopsinya yang luas [4]. Namun, seiring dengan meningkatnya kompleksitas relasi data pada aplikasi modern—seperti sistem penjadalan yang melibatkan entitas mahasiswa, dosen, mata kuliah, dan ruangan—kelemahan REST mulai terlihat, terutama masalah over-fetching (mengambil data berlebih) dan under-fetching (memerlukan banyak request untuk mendapatkan data lengkap) [5].

Sebagai alternatif, GraphQL muncul sebagai query language untuk API yang menawarkan fleksibilitas tinggi, memungkinkan klien untuk meminta data yang spesifik hanya dengan satu request [6]. Sejumlah penelitian telah berupaya membandingkan efisiensi kedua arsitektur ini. Banyak studi awal, seperti yang dilakukan oleh Hartanto [4] dan Cerny & Donahoo [5], berfokus pada perbandingan performa dalam lingkungan aplikasi monolitik atau lingkungan pengujian lokal. Studi-Sistem-studi ini umumnya menemukan bahwa GraphQL unggul dalam mengurangi ukuran payload dan waktu respons untuk query yang kompleks.

Meskipun demikian, terdapat celah penelitian (research gap) yang signifikan. Sangat sedikit penelitian yang mengevaluasi kedua arsitektur API ini dalam konteks cloud-native yang sesungguhnya [2, 6]. Kinerja di lingkungan laboratorium (monolitik) mengabaikan variabel dinamis produksi seperti overhead latensi keamanan

(misal, validasi token) [7], latensi jaringan antar-layanan, service discovery, dan dampak langsung dari algoritma load balancing Kubernetes terhadap distribusi beban. Perbandingan yang adil (fair comparison) di lingkungan yang lebih realistis ini masih sangat dibutuhkan.

Oleh karena itu, penelitian ini akan melakukan analisis perbandingan efisiensi REST API dan GraphQL pada studi kasus sistem penjadalan perkuliahan, yang diimplementasikan sebagai testbed yang adil di atas platform cloud-native (Docker, Kubernetes) dan diuji menggunakan skenario load balancing untuk mensimulasikan beban kerja dunia nyata.

1.2 Rumusan Masalah

1. Bagaimana merancang sebuah testbed perbandingan yang adil (fair comparison) untuk prototipe backend REST dan GraphQL, yang mencakup simulasi overhead latensi keamanan (mock middleware), di atas platform Docker dan Kubernetes?
2. Seberapa signifikankah perbedaan kinerja (ditinjau dari throughput, latency, dan payload size) antara REST dan GraphQL saat diuji dengan skenario query yang bervariasi: (a) query sederhana (data tunggal), (b) query kompleks (relasi data majemuk), dan (c) simulasi under-fetching REST?
3. Manakah arsitektur API yang menunjukkan skalabilitas dan ketahanan (resilience) lebih baik ketika dihadapkan pada skenario beban tinggi (high load) yang didistribusikan melalui load balancer Kubernetes?

1.3 Tujuan Penelitian

1. Merancang dan mengimplementasikan prototipe backend sistem penjadwal (REST dan GraphQL) dengan jaminan fairness—termasuk implementasi mock middleware latensi—dan menjalankannya sebagai testbed di atas platform Docker dan Kubernetes.
2. Menganalisis secara kuantitatif perbedaan kinerja kedua API pada tiga skenario query spesifik (sederhana, kompleks, dan simulasi under-fetching) untuk metrik throughput, latency, dan payload size.
3. Mengevaluasi dan menentukan arsitektur API yang memiliki skalabilitas dan ketahanan (resilience) superior di bawah skenario pengujian beban tinggi (stress testing) dengan load balancing.

1.4 Batasan Masalah

1. Penelitian berfokus pada perbandingan performa API di sisi backend. Aspek frontend (UI/UX) dari aplikasi penjadwal tidak akan dikembangkan atau dievaluasi.
2. Logika bisnis atau algoritma untuk generating jadwal perkuliahan (misalnya, penyelesaian konflik jadwal) berada di luar cakupan penelitian. Sistem dianggap sudah memiliki data jadwal yang siap diakses melalui API.
3. Pengujian dilakukan dalam lingkungan cluster Kubernetes yang terkontrol. Variabel spesifik dari penyedia layanan cloud publik (seperti AWS, GCP, atau Azure) tidak akan dibahas secara mendalam.
4. Penelitian ini tidak mengimplementasikan sistem keamanan (otentikasi/otorisasi) secara fungsional. Namun, sebuah mock middleware yang mensimulasikan overhead latensi konstan (misal: 1-2ms untuk validasi token) akan ditambahkan pada kedua API untuk menjamin fairness perbandingan yang lebih mendekati kondisi nyata.
5. Penelitian tidak berfokus pada optimasi query di database layer. Kinerja basis data diasumsikan setara dan konstan untuk kedua API, yang akan menggunakan skema dan query data mentah yang serupa.
6. Upaya fairness dilakukan pada logika bisnis, namun perbedaan performa minor yang berasal dari library atau framework-specific (misal: ORM, GraphQL resolver) merupakan batasan yang diketahui.

1.5 Kontribusi Penelitian

1. Menyajikan data empiris dan analisis baru mengenai perbandingan performa REST API vs. GraphQL dalam konteks arsitektur cloud-native melalui sebuah testbed perbandingan yang adil (fair comparison) dan lebih mendekati skenario nyata (memasukkan simulasi overhead latensi dan load balancing).
2. Memberikan rekomendasi berbasis data bagi para arsitek sistem dalam memilih arsitektur API yang paling efisien untuk aplikasi berskala besar, khususnya yang memiliki karakteristik relasi data kompleks dan pola lalu lintas spiky (padat pada waktu-waktu tertentu).
3. Menghasilkan blueprint dan prototipe sistem sebagai testbed yang dapat direplikasi untuk penelitian lebih lanjut mengenai optimasi performa API di lingkungan cloud-native.

BAB 2. TINJAUAN PUSTAKA

2.1 Arsitektur Cloud-Native dan Microservices

Transformasi arsitektur perangkat lunak telah bergeser secara signifikan dari model monolitik menuju microservices. Dalam konteks cloud-native, aplikasi dibangun sebagai sekumpulan layanan kecil yang independen dan terisolasi dalam kontainer (seperti Docker). Wang dan Li (2022) menjelaskan bahwa arsitektur ini menawarkan skalabilitas elastis dan fault isolation yang lebih baik dibandingkan monolitik. Namun, pergeseran ini memperkenalkan kompleksitas baru dalam komunikasi antarlayanan, di mana efisiensi pertukaran data melalui Application Programming Interface (API) menjadi faktor kritis yang menentukan performa sistem secara keseluruhan. Orkestrasi menggunakan Kubernetes menjadi standar industri untuk mengelola lifecycle kontainer ini, termasuk fitur auto-scaling dan self-healing.

2.2 Perbandingan Arsitektur API: REST vs GraphQL

2.2.1 REST (Representational State Transfer)

REST telah menjadi standar de-facto untuk desain API selama bertahun-tahun. Karakteristik utamanya adalah statelessness dan penggunaan metode HTTP standar (GET, POST, PUT, DELETE) untuk memanipulasi sumber daya. Kelebihan REST terletak pada kesederhanaannya dan kemampuan caching yang baik pada level HTTP. Namun, Cerny dan Donahoo (2021) menyoroti kelemahan fundamental REST dalam aplikasi data-sentris yang kompleks, yaitu masalah over-fetching (klien menerima data lebih banyak dari yang dibutuhkan) dan under-fetching (klien memerlukan beberapa request ke endpoint berbeda untuk mendapatkan data relasional).

2.2.2 GraphQL

Sebagai respon terhadap keterbatasan REST, Facebook memperkenalkan GraphQL, sebuah query language untuk API. Berbeda dengan REST yang memiliki banyak endpoint, GraphQL mengekspos satu endpoint tunggal yang memungkinkan klien mendefinisikan struktur data persis yang dibutuhkan. Suryadi (2022) dalam studinya menemukan bahwa GraphQL secara signifikan mengurangi ukuran payload data pada aplikasi mobile, yang berimplikasi langsung pada efisiensi bandwidth. Fitur schema-first design pada GraphQL juga memudahkan evolusi API tanpa merusak

kompatibilitas dengan klien lama.

2.3 Mekanisme Load Balancing pada Kubernetes

Dalam lingkungan Kubernetes, load balancing berfungsi mendistribusikan lalu lintas jaringan ke berbagai pod (replika layanan) untuk menjamin ketersediaan dan responsivitas. Sharma dan Gupta (2021) menganalisis berbagai algoritma load balancing (seperti Round Robin dan Least Connection) dan menyimpulkan bahwa efisiensi distribusi beban sangat dipengaruhi oleh karakteristik request. Pada konteks perbandingan API, cara load balancer (seperti Nginx Ingress atau Service tipe LoadBalancer) menangani request HTTP (REST) versus payload POST tunggal (GraphQL) dapat mempengaruhi latensi total sistem, sebuah aspek yang jarang dibahas dalam studi komparasi API tradisional.

2.4 State of the Art dan Research Gap

Terdapat sejumlah penelitian yang membandingkan performa REST dan GraphQL. Hartanto (2021) melakukan perbandingan pada aplikasi e-commerce monolitik dan menemukan GraphQL lebih unggul dalam fleksibilitas data. Studi lain oleh Wang dan Li (2022) mulai menyentuh aspek microservices, namun pengujian dilakukan pada lingkungan lokal tanpa orkestrasi yang kompleks.

Kelemahan utama dari literatur yang ada adalah mayoritas pengujian dilakukan dalam lingkungan yang terisolasi atau monolitik, mengabaikan variabel dinamis yang ada pada lingkungan produksi cloud-native modern. Variabel seperti overhead jaringan antar-pod dalam klaster Kubernetes, latensi yang diperkenalkan oleh ingress controller atau load balancer, serta dampak keamanan (validasi token pada middleware) seringkali tidak diperhitungkan.

Research Gap: Belum ada studi komprehensif yang membandingkan efisiensi REST API dan GraphQL secara head-to-head dalam skenario "dunia nyata" yang melibatkan arsitektur cloud-native penuh dengan Kubernetes, load balancing, dan simulasi latensi keamanan. Penelitian ini hadir untuk mengisi celah tersebut dengan merancang testbed yang adil (fair comparison) untuk mengukur dampak arsitektur API terhadap performa sistem berskala besar.

BAB 3. METODOLOGI PENELITIAN

3.1 Kerangka Penelitian

Penelitian ini akan dilaksanakan mengikuti alur kerja sistematis untuk menjamin validitas hasil eksperimen. Kerangka penelitian digambarkan dalam tahapan berikut:

1. **Studi Literatur:** Memperdalam pemahaman tentang REST, GraphQL, Docker, Kubernetes, dan metrik evaluasi kinerja.
2. **Perancangan Sistem (Design):** Merancang arsitektur microservices untuk sistem penjadwalan kuliah, skema basis data, dan spesifikasi API (OpenAPI untuk REST dan Schema GraphQL).
3. **Implementasi Testbed:** Membangun lingkungan pengujian yang terdiri dari klaster Kubernetes, konfigurasi load balancer, dan pengembangan layanan backend (dua versi: REST dan GraphQL) beserta mock middleware keamanan.
4. **Pengujian & Pengumpulan Data:** Menjalankan skenario uji beban (load testing) menggunakan alat bantu otomatisasi untuk mengukur metrik kinerja.
5. **Analisis Hasil:** Membandingkan data yang diperoleh secara statistik untuk menjawab rumusan masalah.
6. **Penarikan Kesimpulan:** Menyimpulkan arsitektur mana yang lebih efisien berdasarkan bukti empiris.

3.2 Metode Perancangan dan Pengembangan

Sistem yang dikembangkan bukanlah aplikasi produksi penuh, melainkan sebuah Prototipe Testbed yang dirancang khusus untuk memfasilitasi perbandingan yang adil (fair comparison).

Arsitektur Sistem: Menggunakan pola microservices sederhana yang dijalankan di atas Docker container dan diorkestrasi oleh Kubernetes (Minikube atau K3s).

Komponen Backend:

- **Layanan REST:** Dibangun menggunakan Node.js (Express) atau Go (Gin), menyediakan endpoint terpisah untuk entitas Mahasiswa, Dosen, Mata Kuliah, dan Jadwal.

- **Layanan GraphQL:** Dibangun menggunakan teknologi setara (misal Apollo Server), menyediakan satu endpoint /graphql dengan resolver yang memetakan ke struktur data yang sama.
- **Database:** Menggunakan PostgreSQL atau MongoDB yang sama persis untuk kedua layanan guna mengeliminasi bias pada lapisan data.
- **Fairness Mechanism:** Kedua layanan akan dilengkapi dengan mock middleware yang menyuntikkan latensi sintetis (misal: 10ms) pada setiap request untuk mensimulasikan proses otentikasi/otorisasi (JWT validation) tanpa melakukan komputasi kriptografi berat yang tidak relevan dengan tujuan penelitian.

3.3 Metode Pengumpulan Data

Data penelitian ini termasuk dalam kategori Data Primer Kuantitatif yang diperoleh melalui eksperimen terkontrol (benchmarking).

Instrumen Pengujian: Menggunakan tools uji beban modern seperti k6 atau JMeter. Alat ini akan bertindak sebagai klien yang mengirimkan ribuan request ke sistem.

Skenario Pengujian:

1. **Skenario A (Single Resource):** Mengambil data profil satu mahasiswa. (Menguji overhead dasar protokol).
2. **Skenario B (Nested Resource/Complex Query):** Mengambil data jadwal kuliah seorang mahasiswa beserta detail mata kuliah dan nama dosen pengampunya. (Menguji kemampuan fetching data relasional).
3. **Skenario C (High Concurrency/Stress Test):** Mengirimkan beban tinggi (misal: 1000 virtual users bersamaan) untuk melihat titik jenuh (breaking point) dan perilaku load balancer.

Metrik yang Diukur:

- **Response Time / Latency (ms):** Rata-rata, P95, dan P99.
- **Throughput (Requests per Second / RPS).**
- **Payload Size (Kilobytes):** Ukuran data yang dikirim melalui jaringan.
- **Resource Utilization:** Penggunaan CPU dan Memori pada kontainer (dipantau menggunakan metrics-server Kubernetes atau Prometheus).

3.4 Metode Pengujian dan Evaluasi (Verifikasi & Validasi)

Untuk memastikan hasil penelitian valid dan reliabel, rencana V&V berikut akan diterapkan:

Verifikasi Fungsional: Sebelum uji beban, dilakukan Unit Testing dan Integration Testing sederhana untuk memastikan kedua API (REST dan GraphQL) mengembalikan data yang benar dan identik secara semantik untuk input yang sama.

Validasi Lingkungan: Memastikan kondisi cluster Kubernetes "bersih" dan stabil sebelum setiap pengambilan data. Menggunakan teknik warm-up runs sebelum pencatatan data aktual untuk menghindari bias akibat cold start pada aplikasi (khususnya yang berbasis JIT seperti Node.js atau Java).

Analisis Statistik: Data hasil eksperimen tidak hanya dilihat rata-ratanya, tetapi juga dianalisis standar deviasinya untuk melihat konsistensi performa. Uji hipotesis (seperti T-Test) akan digunakan jika diperlukan untuk menentukan apakah perbedaan performa yang diamati signifikan secara statistik.

DAFTAR PUSTAKA

[1] Y. Pratama and A. B. Santoso, "Analisis kinerja sistem informasi akademik berbasis cloud computing saat terjadi lonjakan trafik," *Jurnal Teknologi Informasi dan Ilmu Komputer (JTIK)*, vol. 8, no. 3, pp. 501–510, 2021.

[2] L. Wang and J. Li, "Performance comparison of graphql and rest apis in microservices architecture," in *2022 IEEE International Conference on Web Services (ICWS)*, Shanghai, China, 2022, pp. 310–318.

[3] R. Sharma and P. K. Gupta, "Performance analysis of load balancing algorithms in kubernetes clusters," in *2021 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*, 2021.

[4] D. Hartanto, "Analisis performa graphql dan restful api pada aplikasi e-commerce monolitik," *Jurnal Teknika: Jurnal Sains dan Teknologi*, vol. 17, no. 1, pp. 45–54, 2021.

[5] T. Cerny and M. Donahoo, "Graphql vs. rest: A performance comparison," *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 10, no. 1, pp. 1–15, 2021.

[6] A. Suryadi, "Payload size optimization using graphql in mobile applications," *International Journal of Computer Science and Network Security*, vol. 22, no. 4, pp. 112–119, 2022.

[7] A. F. A. Wibowo and I. D. P. H. S., "Analisis overhead latensi autentikasi jwt pada rest api gateway," *Jurnal Nasional Teknik Elektro dan Teknologi Informasi (JNTETI)*, vol. 10, no. 1, 2021.