Peningkatan Performa Restful API Menggunakan Teknik Caching Dan Query Optimization

Agus Saputra[1], M. Asep Subandri[2], Depandi enda[3]

Program Studi Rekayasa Perangkat Lunak - Politeknik Negeri Bengkalis

Jl. Bathin Alam – Bengkalis Riau - Indonesia

agusptr44@gmail.com, subandri@polbeng.ac.id, dependienda@polbeng.ac.id

***Abstrak*— RESTful API memegang peranan vital dalam pertukaran data arsitektur aplikasi modern. Namun, sistem monitoring lalu lintas di Politeknik Negeri Bengkalis menghadapi kendala performa akibat penggunaan teknik *polling* berkala, yang menyebabkan tingginya beban trafik, latensi, dan kegagalan respons. Penelitian ini bertujuan menganalisis efektivitas penerapan teknik *caching* berbasis Redis dan *query optimization* (meliputi *indexing* dan penyederhanaan *query*) untuk meningkatkan performa API. Metode penelitian menggunakan pendekatan eksperimental dengan *load testing* menggunakan Apache JMeter pada skenario beban 50 hingga 250 pengguna virtual. Hasil pengujian menunjukkan peningkatan performa yang signifikan setelah optimasi. Pada skenario beban normal (50 pengguna), rata-rata waktu respons (*average response time*) berhasil diturunkan dari 36.307 ms menjadi 192 ms. Kapasitas pemrosesan (*throughput*) meningkat drastis dari 1,1 menjadi 11.89 *request* per detik. Selain itu, optimasi ini berhasil menurunkan tingkat kesalahan (*error rate*) dari kisaran 18-29% pada kondisi awal menjadi 0% pada beban menengah, serta menjaga stabilitas sistem pada beban puncak. Penelitian ini menyimpulkan bahwa kombinasi *caching* dan optimasi *query* sangat efektif dalam menangani beban trafik tinggi pada sistem berbasis *polling*.**

***Kata Kunci*— RESTful API, *Caching*, Redis, *Query Optimization*, Apache JMeter, Performa API.**

Abstract* — *RESTful APIs play a pivotal role in data exchange within modern application architectures. However, the traffic monitoring system at Bengkalis State Polytechnic faces performance issues due to periodic polling techniques, resulting in high traffic loads, latency, and response failures. This research aims to analyze the effectiveness of Redis-based caching and query optimization (including indexing and query simplification) in improving API performance. The research method employs an experimental approach with load testing using Apache JMeter under load scenarios ranging from 50 to 250 virtual users. The results demonstrate significant performance improvements post-optimization. In the normal load scenario (50 users), the average response time was successfully reduced from 36.307 ms to 192 ms. Throughput increased drastically from 1.1 to 11,89 requests per second. Furthermore, the optimization successfully reduced the error rate from the 18-29% range in the baseline condition to 0% at medium loads, while maintaining system stability at peak loads. This study concludes that the combination of caching and query optimization is highly effective in handling high traffic loads in polling-based systems.

Keywords* — *RESTful API, Caching, Redis, Query Optimization, Apache JMeter, API Performance.

I. PENDAHULUAN

Application Programming Interface (API) adalah antarmuka perangkat lunak yang memungkinkan komunikasi dan pertukaran data antara berbagai sistem atau aplikasi, seperti aplikasi web, *mobile*, atau *database*. Menurut Faza Alameka et al. API berperan penting dalam memfasilitasi akses data yang efisien [1], dengan menyediakan mekanisme standar untuk bertukar informasi dalam format seperti JSON atau XML. Kemampuan API mendukung interaksi lintas platform menjadikannya elemen penting dalam pengembangan aplikasi modern, yang menuntut skalabilitas dan efisiensi untuk memenuhi harapan pengguna.

API dengan performa tinggi sangat penting untuk menjaga kelancaran alur kerja aplikasi dan meningkatkan pengalaman pengguna. Seperti dijelaskan oleh Deepak, API yang kurang optimal dapat menyebabkan keterlambatan *response*, gangguan operasional, dan penurunan kepuasan pengguna, yang berdampak signifikan pada daya saing aplikasi [2]. Penelitian Fadida Zanetti Junaedy dan Untung Surapati menunjukkan bahwa optimalisasi performa API, seperti melalui penyeimbangan beban dan *caching*, dapat meningkatkan waktu *response* dan stabilitas sistem hingga 5 kali lipat, yang menunjukkan bahwa optimalisasi dapat lebih efisien dan responsif terhadap permintaan klien [3].

Di antara berbagai arsitektur API, RESTful API merupakan arsitektur yang umum digunakan karena kesederhanaan, fleksibilitas, dan kompatibilitasnya dengan berbagai platform. RESTful API memanfaatkan protokol dan metode HTTP seperti *GET*, *POST*, *PUT*, dan *DELETE*, dengan format data yang ringan seperti JSON, sehingga mendukung komunikasi efisien antara *frontend* dan *backend*. Keunggulan RESTful API sebagaimana dikemukakan oleh Deny Eko Septian dan Hutabri, terletak pada kemampuannya untuk mempercepat siklus pengembangan aplikasi dan beradaptasi dengan perubahan

kebutuhan pengguna, sehingga menjadi pilihan yang baik untuk pengembangan API [4].

Untuk memastikan RESTful API dapat permintaan pengguna secara cepat dan stabil, pengujian performa menjadi tahap penting. Pengujian ini bertujuan untuk mengidentifikasi kelemahan sistem, seperti waktu respons yang lambat, tingkat kesalahan (*error rate*) yang tinggi, atau kapasitas sistem yang terbatas (*throughput* rendah). Fadida Zanetti Junaedy dan Untung Surapati menunjukkan bahwa pengujian beban berhasil mengidentifikasi *bottleneck* dalam menangani *request* [3].

Sistem monitoring lalu lintas di Politeknik Negeri Bengkalis yang menjadi fokus penelitian ini umumnya hanya diakses oleh admin atau petugas tertentu. Namun, sistem monitoring tetap menghasilkan beban trafik yang tinggi karena menggunakan teknik *polling* API secara berkala (setiap beberapa detik) untuk memperoleh data *real-time*, seperti jumlah kendaraan dan rata-rata kecepatan. Pola *polling* yang berulang ini membuat jumlah permintaan API terus-menerus tinggi selama sistem beroperasi, meskipun jumlah pengguna terbatas. Kondisi tersebut menuntut API memiliki performa yang andal dan efisien agar tetap mampu merespons secara cepat dan stabil.

Metode *load testing* dinilai efektif dalam menguji kemampuan API pada kondisi beban tinggi. Setiawan, G. H., Adnyana, I. M. B., & Budiarta, K.menyebutkan bahwa *load testing* mensimulasikan beban pengguna dalam jumlah besar untuk mengevaluasi performa API, dengan fokus pada metrik seperti waktu respons (*response time*), jumlah permintaan yang berhasil diproses (*throughput*), dan tingkat kesalahan (*error rate*) [5].

Dalam penelitian ini, Apache JMeter dipilih sebagai alat pengujian karena kemampuannya mensimulasikan beban pengguna secara realistis dan menghasilkan laporan metrik lengkap. JMeter dapat mengatur jumlah pengguna virtual dan jenis permintaan API, serta mendukung analisis hasil pengujian dalam format yang memudahkan evaluasi.

Setelah batas performa API teridentifikasi, langkah optimasi menjadi sangat penting untuk meningkatkan kecepatan, stabilitas, dan efisiensi layanan. Dua teknik utama yang akan diterapkan adalah *caching* dan *query optimization*.

Caching menyimpan data yang sering diakses pada memori sementara seperti Redis untuk mengurangi beban database dan mempercepat waktu respons. Fadida Zanetti Junaedy dan Untung Surapati menunjukkan bahwa penerapan *caching* dapat meningkatkan kinerja API hingga 5 kali lipat dalam hal waktu *response* [3]. Sementara itu, *query optimization* bertujuan menyederhanakan pengambilan data dari *database*, misalnya dengan meminimalkan operasi *join*, memilih kolom tertentu, atau menambahkan indeks untuk mempercepat eksekusi *query*. Penerapan teknik caching pada level aplikasi terbukti mampu memangkas latensi secara signifikan saat sistem menghadapi lonjakan trafik yang tinggi [6]. Selain itu, optimasi pada indeks database juga sangat krusial untuk mempercepat waktu pemuatan data yang besar [7].

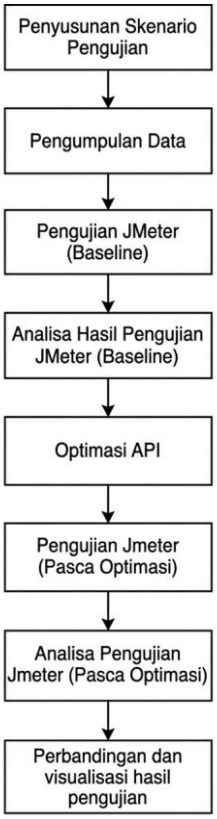
Penelitian ini menekankan evaluasi performa RESTful API dengan menggunakan *load testing* sebelum dan sesudah optimasi diterapkan. Pengujian beban menggunakan Apache JMeter akan mensimulasikan permintaan dalam skala besar dan mengukur metrik utama seperti *response time*, *throughput*, dan *error rate*. Dengan demikian, penelitian ini bertujuan memberikan gambaran nyata tentang dampak teknik optimasi *caching* dan *query optimization* terhadap efisiensi dan keandalan RESTful API, khususnya dalam konteks aplikasi monitoring berbasis data real-time.

II. METODE PENELITIAN

Bagian ini memuat metode yang digunakan pada proses dalam pembuatan penelitian ini.

A. Prosedur Penelitian

Penelitian ini dilakukan melalui beberapa tahapan untuk mencapai tujuan penelitian yang efektif dan efisien. Tahapan-tahap ini dirancang untuk memastikan bahwa proses pengumpulan dan analisis data berjalan dengan baik dan menghasilkan hasil yang diinginkan. Untuk langkah-langkah penelitian yang dilakukan dapat dilihat pada gambar 1:



Gbr. 1 Prosedur Penelitian

B. Penyusunan Skenario Pengujian

Penyusunan skenario pengujian merupakan langkah awal yang kritis dalam penelitian ini untuk mengevaluasi performa RESTful API sebelum dan sesudah penerapan teknik optimasi, yaitu caching (menggunakan Redis) dan query optimization. Skenario ini dirancang penulis untuk mensimulasikan kondisi nyata dengan beban pengguna tinggi, menggunakan Apache JMeter sebagai alat pengujian performa. Skenario ini untuk

mendukung pengalaman API yang optimal, penulis menyusun dokumen perencanaan testing sebagai berikut:

1. Pendahuluan
   Tujuan penelitian ini adalah mengevaluasi performa API sebelum dan sesudah optimasi menggunakan *Caching* dan *query optimization*.
2. Item yang Diuji
   a. Sistem: RESTful API.
3. Fitur yang Diuji
   a. *Endpoint GET* pada API.
   b. Metrik: *Response time, throughput, error rate*.
4. Pendekatan Pengujian
   a. Alat: Apache JMeter 5.6.3.
   b. Pengujian dilakukan dengan menggunakan Apache JMeter untuk men simulasikan 50, 100, 150, 200, 250 pengguna virtual.
   c. Pengujian dilakukan dalam dua fase, yaitu fase *baseline* (tanpa optimasi) dan *pasca-optimasi* (menggunakan *caching* redis dan *query optimization*). Metrik yang diamati meliputi *response time, throughput,* dan *error rate*.

TABEL I
SKENARIO PENGUJIAN 50 USERS VIRTUAL

Fase	Baseline	Pasca-Optimasi
Alat Pengujian	Apache JMeter	Apache JMeter
Jumlah Pengguna	50 pengguna virtual	50 pengguna virtual
Ramp-Up	50 detik	50 detik
Optimasi	Tanpa *Caching*, *query optimization*	Dengan *Caching*, *query optimization*
Tujuan Pengujian	Mengukur performa sistem awal: (*response time*, throughput, *error rate*)	Mengukur performa sistem pasca – optimasi: (*response time*, throughput, *error rate*)
Metrik yang Diukur	- *Response time* - Throughput (req/s) - *Error rate*	- *Response time* - Throughput req/s) - *Error rate*
Skenario Pengujian	Simulasi request oleh 50 pengguna aktif terhadap *endpoint* GET	Simulasi oleh 50 pengguna aktif terhadap *endpoint* GET
Ekspektasi	*Response time* lebih lama dan *throughput* lebih lama.	*Response time* lebih cepat, throughput lebih cepat.

5. Kriteria Keberhasilan
   a. *Response time* rata-rata lebih rendah setelah optimasi
   b. *Throughput* lebih meingkat.
   c. *Error rate* lebih kecil.
6. Sumber Daya
   a. SDM: Peneliti.
   b. Perangkat: JMeter, Redis, MySQL, VPS.
7. *Deliverables*
   a. Laporan perbandingan performa.
   b. Grafik dan tabel hasil pengujian.

C. Pengumpulan Data

Pada tahap ini, dilakukan proses pengumpulan data awal yang akan digunakan sebagai dasar dalam pengujian baseline. Data yang dikumpulkan mencakup daftar endpoint API yang akan diuji, load 2.000.000 row data dengan menyesuaikan field tabel untuk simulasi beban database, beserta detail permintaan (request) dan tanggapan (response) dari masing-masing endpoint. Informasi ini menjadi landasan penting dalam proses pengujian, karena akan menentukan parameter dan skenario yang digunakan untuk mengukur performa awal sistem.

D. Pengujian JMeter (Baselin)

Tahapan ini bertujuan untuk mengukur performa API dalam kondisi awal sebelum dilakukan optimasi. Pengujian dilakukan dengan mengakses *endpoint* yang telah ditentukan dan mencatat metrik-metrik utama, seperti *response time*, throughput, dan error rate. Pengukuran parameter awal ini sangat penting karena konfigurasi sistem yang tidak optimal sering kali memicu tingginya tingkat kesalahan (error rate) meskipun spesifikasi hardware server yang digunakan sudah tinggi [8]. Data yang diperoleh dari pengujian *baseline* ini akan menjadi acuan untuk membandingkan peningkatan performa setelah dilakukan optimasi.

E. Analisa Hasil Pengujian JMeter (Baseline)

Menganalisa hasil pengujian awal (*baseline*) dengan menghitung response time, throughput, dan error rate menggunakan perhitungan berikut:

1. *Response time*

$$Response\ Time = \frac{Jumlah\ Seluruh\ Nilai\ Sample\ Time}{Jumlah\ Sample}$$

2. *Throughput*

$$Througput = \frac{Jumlah\ Request}{Waktu\ Rata - Rata\ Request}$$

3. *Error rate*

$$Error\ Rate\ = (\frac{Jumlah\ Sample\ Gagal}{Total\ Sample})\ x\ 100\%$$

F. Optimasi API

Melakukan modifikasi *code* API untuk mengimplementasikan *caching (Redis)* dan juga *query optimization* terhadap masing-masing *endpoint* yang telah di ujikan sebelum dilakukan pengujian kembali menggunakan skenario yang sama dengan pengujian pada tahapan sebelumnya (*baseline*). Pemilihan Redis sebagai penyedia cache didasarkan pada efektivitasnya untuk menurunkan waktu respons pada aktivitas data yang sering diakses [9], sedangkan optimasi query dilakukan untuk mencegah terjadinya bottleneck komputasi pada database relasional [10].

G. Pengujian JMater (Pasca Optimasi)

Setelah proses optimasi dilakukan, tahap selanjutnya adalah melakukan pengujian ulang terhadap *endpoint* API. Tujuannya adalah untuk mengetahui dampak dari perubahan yang telah

diterapkan. Pengujian ini menggunakan metode dan skenario yang sama dengan baseline untuk menjamin validitas perbandingan. Metrik yang kembali diukur meliputi *response time*, *throughput*, dan *error rate*.

H. Analisa Hasil Pengujian JMeter (Pasca Optimasi)

Menganalisa hasil pengujian pasca optimasi dengan menghitung kembali response time, throughput, dan error rate menggunakan perhitungan yang sama dengan analisa hasil pengujian sebelumnya.

I. Perbandingan dan Visualisasi Hasil Pengujian

Membuat dokumentasi hasil pengujian serta memvisualisasikan hasil dari analisa pengujian dan juga menunjukkan hasil dari perbandingan sebelum dan sesudah di lakukannya optimasi dalam bentuk diagram.

J. Analisa Kebutuhan

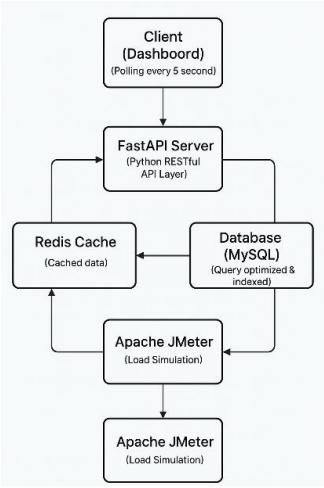
Dalam penelitian ini, sistem yang diuji merupakan RESTful API dari sistem monitoring lalu lintas yang digunakan oleh pihak internal (admin/operator) untuk memantau data pelanggaran, klasifikasi kendaraan, dan riwayat lalu lintas secara real-time, Agar sistem dapat berjalan optimal dalam kondisi penggunaan normal, dilakukan analisa terhadap kebutuhan sistem sebagai berikut:

1. Kebutuhan Fungsional
   Sistem menyediakan endpoint RESTful API untuk:
   a. Menampilkan klasifikasi kendaraan (GET /api/traffic-classification)
   b. Menampilkan dan mengakses riwayat lalu lintas (GET /api/traffic-history)
   c. Menampilkan data history detail (byId) (GET /api/traffic-history/{history_id}
   d. Menampilkan data rata rata kecepatan kendaraan (GET /api/average-speed)
2. Kebutuhan Non-Fungsional
   a. Response time rendah
   b. Permintaan data yang sering (polling) tidak membebani secara berlebihan.

K. Arsitektur Sistem

Arsitektur terdiri dari lima komponen utama terdapat pada gambar 2, yaitu *client* (*Dashboard Monitoring)*, FastAPI Server sebagai penyedia layanan RESTful API, Redis cache, database MySQL, dan Apache JMeter sebagai alat pengujian beban, *Client* berupa *dashboard* admin melakukan *polling* permintaan setiap 5 detik ke *endpoint* FastAPI. Setiap permintaan akan diproses oleh FastAPI Server. Jika data yang diminta tersedia di Redis, maka akan langsung dikembalikan (cache hit). Namun jika tidak (*cache miss*), FastAPI akan mengambil data dari database MySQL yang telah dioptimasi menggunakan indeks dan penyederhanaan query. Data yang diambil dari database kemudian disimpan kembali ke redis untuk permintaan berikutnya.

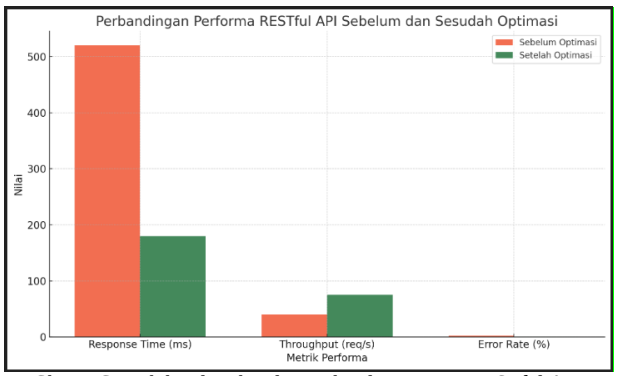
Untuk menguji performa sistem bisa dilihat pada gambar 3 dan 4, Apache JMeter digunakan guna mensimulasikan incremental pengguna virtual yang secara bersamaan mengakses endpoint secara periodik. Pengujian dilakukan dalam dua skenario: baseline (tanpa optimasi) dan pasca-optimasi (dengan redis dan query optimization). Metrik performa yang dianalisis meliputi *response time*, *throughput*, dan *error rate*.



Gbr. 2 Arsitektur Sistem

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throug...	Received ...	Sent KB/sec	Avg. Bytes
Traffic Classi...	600	8621	124	19508	3866.87	0.00%	1.1/sec	772.20	0.15	714916.0
Traffic History	600	8573	1	20060	4067.86	1.50%	1.1/sec	1240.59	0.14	1157031.2
Traffic Histor...	600	8493	1	19607	4012.35	2.00%	1.1/sec	1234.17	0.16	1152184.6
Average Spe...	600	105	0	3679	375.31	1.50%	1.1/sec	0.20	0.14	188.8
TOTAL	2400	6448	0	20060	5034.66	1.25%	4.4/sec	3237.22	0.60	756080.2

Gbr. 3 Hasil pengujian pada JMeter



Gbr. 4 Contoh hasil perbandingan hasil pengujian RESTful API

III. HASIL DAN PEMBAHASAN

A. Arsitektur Sistem dan Persiapan Lingkungan Uji

Implementasi sistem ditempatkan pada lingkungan *Virtual Private Server* (VPS) untuk mensimulasikan lingkungan produksi yang realistis. Penggunaan VPS dipilih untuk menghindari variabilitas performa yang sering terjadi pada lingkungan lokal (*localhost*) akibat proses latar belakang sistem operasi.

Spesifikasi lingkungan pengujian yang digunakan adalah sebagai berikut:

1. *Processor:* 1 vCPU Core
2. *Memory* (RAM): 4 GB
3. *Storage:* 50 GB NVMe
4. *Bandwith:* 4TB

5. *Operating System:* Linux (*Virtualisasi* KVM)
6. *Containerization:* Docker & Docker Compose

Seluruh komponen sistem (Aplikasi FastAPI, Database MySQL, dan Redis) dijalankan dalam container terisolasi di dalam VPS tersebut untuk menjamin konsistensi konfigurasi.

B. Implementasi Basis Data dan Persiapan Data Uji

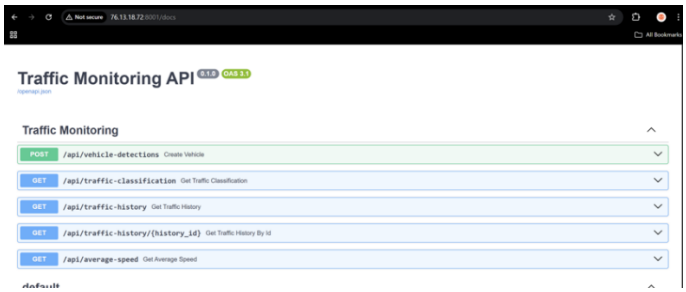
Sistem menggunakan satu tabel utama bernama *vehicle_detections* untuk menyimpan data hasil deteksi kendaraan. Struktur tabel ditunjukkan pada Listing berikut:

```
CREATE TABLE vehicle_detections (
    id INT AUTO_INCREMENT PRIMARY KEY,
    vehicle_type VARCHAR(50) NOT NULL,
    speed_kmph FLOAT,
    confidence FLOAT,
    track_id INT,
    location VARCHAR(100),
    detected_at DATETIME
);
```

Eksperimen dilakukan menggunakan *Virtual Private Server* (VPS) dengan spesifikasi 1 vCPU Core, RAM 4 GB, dan penyimpanan 50 GB NVMe. Seluruh layanan (FastAPI, MySQL, dan Redis) dijalankan di dalam *container* Docker. Data uji dikonfigurasi secara masif dengan melakukan seeding sebanyak 2.000.000 baris data pada tabel *vehicle_detections* untuk mensimulasikan beban riil pada database.

C. API pada Kondisi Awal (Baseline)

Pada fase *baseline*, RESTful API dijalankan tanpa menggunakan mekaniSme *caching* maupun indeks basis data. Pengujian beban dilakukan menggunakan Apache JMeter dengan menaikkan beban secara bertahap dari 50 hingga 250 *virtual users*, bisa dilihat pada gambar 5.



Gbr. 5 API berjalan di VPS

D. Pengujian API Menggunakan Apache JMeter (Baseline)

Pengujian kondisi baseline dilakukan untuk mengukur performa API sebelum diterapkannya mekanisme optimasi, pengujian difokuskan pada empat endpoint utama API, yaitu:

1. GET – Traffic Classification
2. GET – Traffic History
3. GET – Traffic History by ID
4. GET – Average Speed

Untuk memastikan validitas perbandingan performa antara kondisi *baseline* dan pasca-optimasi, seluruh parameter pengujian dikonfigurasi secara identik pada kedua fase tersebut. Konfigurasi ini bertujuan untuk mengukur metrik *response time*, *throughput*, dan *error rate* ketika sistem menerima beban, berikut detail skenario pengujian yang diterapkan:

TABEL II
SKENARIO PENGUJIAN USERS VIRTUAL

Fase	Baseline	Pasca-Optimasi
Alat Pengujian	Apache JMeter	Apache JMeter
Jumlah Pengguna	200 pengguna virtual	200 pengguna virtual
Ramp-Up	50 detik	50 detik
Optimasi	Tanpa *Caching*, *query optimization*	Dengan *Caching*, *query optimization*
Tujuan Pengujian	Mengukur performa sistem awal (*response time*, throughput, *error rate*)	Mengukur performa sistem pasca-optimasi(*response time*, throughput, *error rate*)
Metrik yang Diukur	- *Response time* - Throughput (req/s) - *Error rate*	- *Response time* - Throughput (req/s) - *Error rate*
Skenario Pengujian	Simulasi *polling* oleh 200 pengguna aktif terhadap *endpoint* GET	Simulasi *polling* oleh 200 pengguna GET
Ekspektasi	*Response time* lebih lama dan *throughput* lebih lama.	*Response time* lebih cepat, throughput lebih cepat.

TABEL III
SKENARIO PENGUJIAN 200 USERS

Label	Users	Average	Median	90% Line
GET - Traffic Classification	200	58264	67119	115549
GET - Traffic History by ID	200	79367	87410	114293
GET - Average Speed	200	107466	108168	140347
GET - Traffic History	200	195487	203231	260268
TOTAL	800	110146	103451	207790

Min	Max	Error %	Throughput
549	123680	1.50%	1.1532
92	135810	22.00%	0.67268
3191	213905	21.00%	0.40955
48338	307397	52.00%	0.29163
92	307397	24.13%	1.15903

Pada skenario pengujian beban puncak dengan 250 *virtual users*, tercatat adanya kegagalan permintaan (*failed samples*) sebanyak 57 *request* dengan total *error rate* mencapai 23.40%. Kegagalan paling signifikan terjadi pada endpoint GET - Traffic History yang mengalami tingkat kesalahan hingga 68.97%, bisa dilihat pada gambar 6.

```
root@srv1265336:~# free -h
               total        used        free      shared  buff/cache   available
Mem:           3.8Gi       3.8Gi       107Mi       1.4Mi        45Mi       7.4Mi
Swap:             0B          0B          0B
```

Gbr. 6 Ram usage full

Berdasarkan pemantauan sumber daya server seperti terlihat pada Gambar 4.3, teridentifikasi bahwa kegagalan ini disebabkan oleh fenomena *resource exhaustion*, di mana kapasitas memori utama (RAM) terpakai sepenuhny, Inefisiensi Query Database, tanpa mekanisme *caching* dan *indexing*, database dipaksa melakukan pemindaian data (*full table scan*) secara berulang untuk melayani 250 permintaan konkuren. Ini menyebabkan lonjakan penggunaan memori untuk proses *sorting* dan *buffer* data.

Saat penggunaan RAM mencapai titik jenuh (100%), sistem operasi mengaktifkan mekanisme *Out of Memory (OOM) Killer* yang secara paksa menghentikan proses layanan untuk mencegah *crash* total pada sistem operasi. Inilah yang menyebabkan putusnya koneksi yang terdeteksi sebagai *error* pada Apache JMeter, kondisi ini membuktikan bahwa spesifikasi infrastruktur saat ini tidak mampu menangani beban tinggi tanpa adanya penerapan strategi optimasi perangkat lunak.

E. Metode Analisis dan Validasi Hasil

1) Analisa Hasil Pengujian Baseline

Berdasarkan serangkaian pengujian beban (load testing) yang telah dilakukan menggunakan Apache JMeter pada kondisi baseline (tanpa mekanisme caching dan optimasi query), dilakukan analisis mendalam mengenai kinerja API. Pengujian difokuskan pada metrik Average Response Time, 90th Percentile (P90), Throughput, dan Error Rate dengan variasi beban mulai dari 50 hingga 250 virtual users.

Data hasil pengujian menunjukkan degradasi performa yang signifikan dan ketidakstabilan latensi yang tinggi pada sisi server.

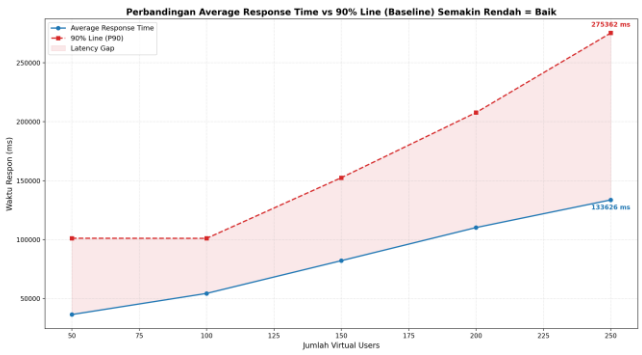
Kesimpulan Analisis Baseline Hasil pengujian *baseline* menyimpulkan bahwa performa API saat ini tidak layak untuk lingkungan produksi (*not production-ready*). Tingginya nilai P90 yang mencapai ratusan detik menunjukkan bahwa pengalaman pengguna (*user experience*) sangat buruk. Masalah ini disebabkan oleh ketidakmampuan server menangani *query* basis data secara efisien, yang memicu saturasi RAM dan CPU. Oleh karena itu, penerapan mekanisme *Redis Caching* dan *Query Optimization* diperlukan.

Untuk memberikan gambaran mengenai degradasi performa sistem, berikut disajikan ringkasan metrik utama (*Average Response Time*, *P90 Response Time*, *Throughput*, dan *Error Rate*) dari seluruh skenario pengujian.

TABEL IV
REKAP PERFORMA SISTEM KONDISI BASELINE

Jumlah Users (Virtual)	Avg. Response Time (ms)	90% Line (ms)	Throughput (req/sec)	Error Rate (%)
50	36.307	101.082	0,63	18,00%
100	54.252	101	1,11	23,25%
150	82.041	152.496	1,18	29,00%
200	110.146	207.79	1,15	24,13%
250	133.626	275.362	1,07	23,40%

Berdasarkan Tabel 3, terlihat jelas bahwa nilai 90% Line selalu jauh lebih tinggi daripada rata-rata, yang menandakan variabilitas performa yang buruk. Gambar 7 Berikut disajikan visualisasi perbandingan keduanya.:



Gbr. 7 Perbandingan average response time vs 90% line (baseline)

Grafik pada Gambar 7 menunjukkan kesenjangan (*gap*) yang signifikan antara waktu respons rata-rata dan *90th Percentile* (P90). Area luas yang diarsir merah muda mengindikasikan ketidakstabilan sistem, di mana mayoritas pengguna mengalami latensi yang jauh lebih buruk dibandingkan rata-rata statistic.

2) Optimasi API

Setelah dilakukan pengujian baseline, dilakukan penerapan optimasi performa terhadap API yang telah tersedia. Optimasi dilakukan tanpa mengubah logika bisnis utama API, melainkan difokuskan pada efisiensi proses pengambilan data, tujuannya adalah meminimalkan waktu eksekusi query, mengurangi beban komputasi CPU, serta membatasi ukuran data (payload) yang ditransfer melalui jaringan.

Berikut adalah rincian teknis dari strategi optimasi yang diterapkan:

1. Penerapan Database Indexing

Salah satu penyebab utama lambatnya respon pada kondisi *baseline* adalah mekanisme pemindaian tabel secara

menyeluruh (*full table scan*). Untuk mengatasi hal ini, diterapkan strategi *indexing* yang komprehensif mencakup *Single Column Index* sesuai dengan pola akses data pada aplikasi.

Berdasarkan struktur tabel vehicle_detections, berikut adalah indeks yang diterapkan untuk mempercepat pencarian data:

a. idx_vehicle_type: Digunakan untuk mempercepat proses agregasi pengelompokan (GROUP BY) jenis kendaraan pada fitur klasifikasi.
b. idx_detected_at: Digunakan untuk mempercepat penyaringan rentang waktu (*time-window filtering*).

Implementasi indeks tersebut direpresentasikan dalam perintah DDL (*Data Definition Language*) sebagai berikut:

```
-- 1. Index untuk mempercepat grouping berdasarkan tipe kendaraan
CREATE INDEX idx_vehicle_type ON vehicle_detections (vehicle_type);

-- 2. Index untuk mempercepat filter rentang waktu
CREATE INDEX idx_detected_at ON vehicle_detections (detected_at);
```

2. Optimasi *Query* dan Logika Aplikasi

Selain struktur indeks yang lengkap, logika pengambilan data pada kode program dimodifikasi untuk memanfaatkan indeks tersebut secara maksimal.

Implementasi kode Python menggunakan FastAPI dan SQLAlchemy setelah optimasi adalah sebagai berikut:

```
Python
from fastapi_cache.decorator import cache
from app_optimation.models.vehicle_model import VehicleDetection
from datetime import datetime, timedelta
from sqlalchemy import func, desc
from sqlalchemy.orm import Session
import pytz

wib = pytz.timezone("Asia/Jakarta")

@cache(expire=60)
def get_all_classification(db: Session):
    """
    Mengambil data agregat jumlah kendaraan per jenis.
    Optimasi: Query ini secara otomatis memanfaatkan 'idx_vehicle_type'
    untuk menghindari full table scan saat melakukan GROUP BY.
    """
    result = (
        db.query(
            VehicleDetection.vehicle_type,
            func.count(VehicleDetection.id).label("count")
        )
        .group_by(VehicleDetection.vehicle_type)
        .all()
    )
    return [{"vehicle_type": r.vehicle_type, "count": r.count} for r in result]

@cache(expire=10)
def get_history(db: Session, limit: int = 100):
    """
    Optimasi:
    1. Limit Query: Membatasi hanya 100 baris.
    2. Indexing: Menggunakan index waktu ('idx_detected_at') untuk sorting.
    """
    results = (
        db.query(VehicleDetection)
        .order_by(desc(VehicleDetection.detected_at))
        .limit(limit)
        .all()
    )
    return [
        {
            "id": r.id,
            "vehicle_type": r.vehicle_type,
            "speed_kmph": r.speed_kmph,
            "confidence": r.confidence,
            "location": r.location,
            "detected_at": r.detected_at.isoformat() if r.detected_at else None
        }
        for r in results
    ]

@cache(expire=60)
def get_average_speed(db: Session):
    """
    Menghitung rata-rata kecepatan 24 jam terakhir.
    Optimasi: Time-Window Filtering.
    Menggunakan 'idx_detected_at' untuk melompati jutaan data lama
    dan hanya menghitung data hari ini.
    """
    last_24h = datetime.now(wib) - timedelta(hours=24)

    result = db.query(func.avg(VehicleDetection.speed_kmph))\
        .filter(VehicleDetection.detected_at >= last_24h)\
        .scalar()

    return {
        "average_speed": round(result, 2) if result else 0.0,
        "unit": "km/h",
        "period": "Last 24 Hours"
    }
```

3. Analisis Optimasi

Berdasarkan implementasi di atas, terdapat beberapa poin kunci optimasi:

1. *Sorting* Tanpa *Filesort*

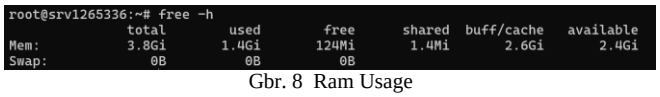
Dengan adanya idx_detected_at dan varian urutannya (_asc), operasi ORDER BY detected_at DESC pada fungsi get_history tidak perlu melakukan pengurutan manual di memori (*filesort*). Basis data cukup membaca indeks dari "belakang ke depan" untuk mendapatkan data terbaru secara instan.

2. Efisiensi *Where Clause*:

Pada fungsi get_average_speed, filter .filter(detected_at >= last_24h) dieksekusi menggunakan *Range Scan* pada indeks. Ini berarti basis data langsung melompat ke posisi data 24 jam terakhir tanpa menyentuh data historis yang berjumlah besar, mengurangi *I/O Disk* secara drastis

c. Pengujian API Menggunakan Apache JMeter (Pasca Optimasi)

Setelah optimasi diterapkan dan API, dilakukan kembali pengujian performa API menggunakan Apache JMeter dengan skenario yang sama seperti pada pengujian *baseline*. Hal ini bertujuan untuk memperoleh perbandingan performa yang objektif, berikut detail skenario pengujian yang diterapkan



Gbr. 8 Ram Usage

Gambar 8 memperlihatkan status penggunaan memori (*RAM usage*) pada server saat menangani beban puncak 250 *virtual users* pada fase pasca-optimasi. Berbeda dengan kondisi *baseline* yang mengalami *resource exhaustion* hingga menyebabkan kegagalan sistem, gambar ini menunjukkan bahwa konsumsi memori kini jauh lebih terkendali dan stabil . Minimnya sample error dan absennya kegagalan pada skenario 250 users membuktikan bahwa arsitektur sistem kini jauh lebih efisien dibandingkan kondisi baseline. Sebagai langkah verifikasi akhir, beban pengujian akan dinaikkan menjadi 500 users.

TABEL V
SKENARIO PENGUJIAN 500 USERS VIRTUAL

Fase	Baseline	Pasca-Optimasi
Alat Pengujian	Apache JMeter	Apache JMeter
Jumlah Pengguna	500 pengguna virtual	500 pengguna virtual
Ramp-Up	50 detik	50 detik
Optimasi	Tanpa *Caching*, *query optimization*	Dengan *Caching*, *query optimization*
Tujuan Pengujian	Mengukur performa sistem awal (*response time*, throughput, *error rate*)	Mengukur performa sistem pasca-optimasi(*response time*, throughput, *error rate*)
Metrik yang Diukur	- *Response time* - Throughput (req/s) - *Error rate*	- *Response time* - Throughput (req/s) - *Error rate*
Skenario Pengujian	Simulasi polling oleh 500 pengguna aktif terhadap *endpoint* GET	Simulasi polling oleh 500 pengguna aktif terhadap *endpoint* GET
Ekspektasi	Performa standar, *response time* lebih lama dan *throughput* lebih lama.	*Response time* lebih cepat, throughput lebih cepat.

TABEL VI
HASIL PENGUJIAN USERS VIRTUAL

Min	Max	Error %	Throughput
227	202897	0.20%	1.68064
84	175810	0.40%	1.71893
66	173326	32.40%	1.77332
91	165200	18.60%	2.05199
66	202897	12.90%	6.45245

3) *Analisa Hasil Pengujian Pasca Optimasi*

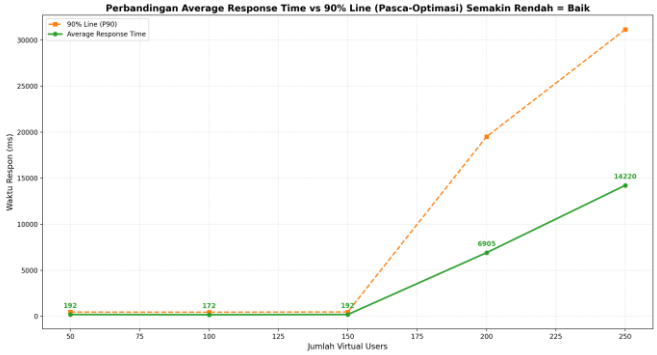
Pada tahapan ini, dilakukan analisis mendalam terhadap peningkatan kinerja dan stabilitas RESTful API setelah diterapkannya integrasi Redis *Caching* dan *Query Optimization*. Pengujian ulang dilakukan menggunakan skenario beban konkurensi yang identik dengan fase baseline (50 hingga 250 *users*) untuk mengukur efektivitas perbaikan perangkat lunak secara objektif. Evaluasi difokuskan pada pemantauan stabilitas latensi (*Average Response Time* dan P90) pada beban rendah hingga menengah guna memastikan hilangnya gejala kelambatan (*lag*) pada sisi klien. Selain itu, analisis juga dilakukan terhadap lonjakan nilai *throughput* untuk memastikan terlepasnya sumbatan pemrosesan permintaan (*unblocking bottleneck*) pada layer aplikasi. Pengujian ini juga mengukur kemampuan arsitektur baru dalam mengeliminasi tingkat kesalahan (*error rate* hingga 0%) akibat masalah *timeout*, serta menguji ketahanan (*resilience*) operasional server dan efisiensi konsumsi memori (RAM) saat menghadapi lonjakan trafik ekstrem pada kondisi beban puncak agar tidak terjadi kelumpuhan sistem (*system crash*).

Untuk memberikan gambaran mengenai peningkatan performa sistem setelah optimasi, berikut disajikan ringkasan metrik utama (*Average Response Time*, *P90 Response Time*, *Throughput*, dan *Error Rate*) dari kelima skenario pengujian.

TABEL VII
REKAP HASIL PENGUJIAN PASCA-OPTIMATION

Jumlah Users (Virtual)	Avg. Response Time (ms)	90% Line (ms)	Throughput (req/sec)	Error Rate (%)
50	192	461	4,04	0,00%
100	172	445	7,95	0,00%
150	192	472	11,89	0,00%
200	6.905	19.507	11,15	0,00%
250	14.22	31.15	9,75	0,70%

Berdasarkan tabel di atas terlihat pola performa yang sangat bagus di mana sistem mampu melayani permintaan dengan kecepatan tinggi hingga titik jenuh di 150 pengguna. Di atas angka tersebut, sistem melakukan *graceful degradation* (melambat tapi tidak mati), gambar 9 adalah visualisasi tren performa pasca-optimasi:

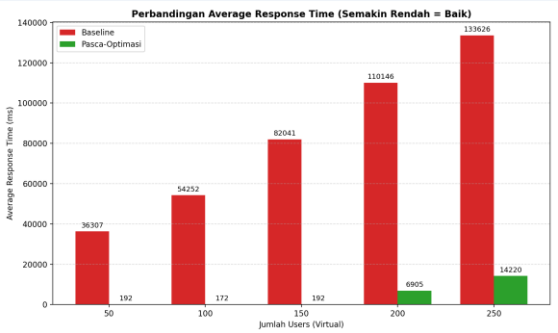


Gbr. 9 *Perbandingan average response time vs p90 (Pasca Optimasi)*

4) *Perbandingan Dan Visualisasi Hasil Pengujian*

Evaluasi efektivitas optimasi dilakukan dengan membandingkan metrik kinerja antara kondisi baseline (sebelum optimasi) dan kondisi pasca-optimasi. Pengujian dilakukan dengan variasi beban yang identik, mulai dari 50 hingga 250 pengguna virtual (virtual users), untuk memastikan validitas perbandingan. Evaluasi difokuskan pada empat indikator utama kualitas layanan (Quality of Service), yaitu Average Response Time, 90th Percentile Response Time, Throughput, dan Error Rate.

1. Perbandingan Average Response Time (Kecepatan Rata-rata)
   a. Kondisi Baseline
      Sistem mengalami latensi tinggi sejak beban terendah. Pada 50 users, rata-rata waktu respon tercatat 36.307 ms (36 detik) dan meningkat secara linear hingga mencapai 133.626 ms (2 menit 13 detik) pada beban 250 users. Hal ini menunjukkan ketidakmampuan basis data menangani antrean query tanpa indeks.
   b. Pasca-Optimasi
      Waktu respon menurun drastis menjadi kisaran 172 ms hingga 192 ms pada beban normal (50–150 users). Pada beban puncak 250 users, meskipun terjadi antrean, rata-rata waktu respon tercatat 14.220 ms, bisa dilihat pada gambar 10.

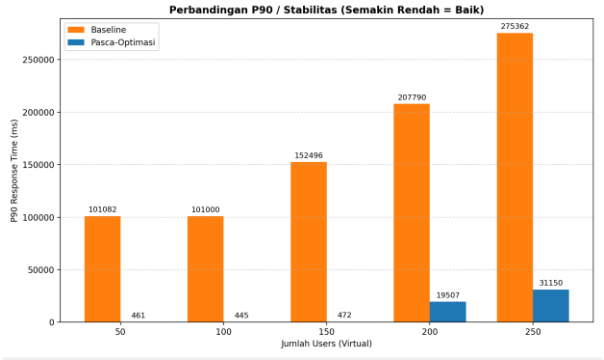


Gbr. 10 Perbandingan average response time

2. Perbandingan *90th Percentile* (Stabilitas Sistem)
   Metrik *90th Percentile* (P90) menunjukkan batas waktu maksimum yang dialami oleh 90% pengguna.
   Analisis data P90 menunjukkan peningkatan stabilitas yang fundamental:
   a. Ketidakstabilan *Baseline*:

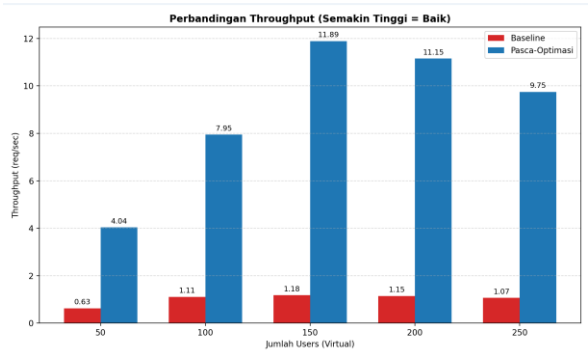
      Nilai P90 pada kondisi awal sangat fluktuatif dan ekstrem, mencapai 101.082 ms pada 50 *users* dan melonjak hingga 275.362 ms pada 250 *users*. Ini mengindikasikan bahwa mayoritas pengguna mengalami *lag* yang parah.
   b. Stabilitas Pasca-Optimasi:

      Penerapan *Redis Caching* berhasil memangkas latensi ekor. Pada beban 50 *users*, P90 turun menjadi 461 ms (sebelumnya 101 detik). Bahkan pada beban tersulit (250 *users*), P90 tercatat 31.150 ms, jauh lebih stabil dibandingkan *baseline*, bisa dilihat pada gambar 11.



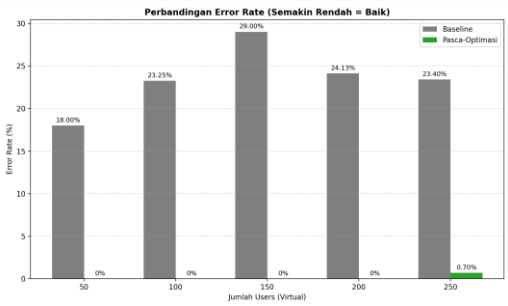
Gbr. 11 Perbandingan p90

3. Perbandingan *Throughput* (Kapasitas Transaksi)
   Throughput mengukur jumlah permintaan yang berhasil diproses server per detik (*requests per second*). Peningkatan *throughput* menandakan peningkatan kapasitas (*scalability*) sistem.
   a. *Baseline*:
      Pada kondisi awal, *throughput* mengalami stagnasi di angka ~1,1 req/sec mulai dari beban 100 *users*. Penambahan beban pengguna tidak meningkatkan jumlah transaksi yang diproses, melainkan hanya memperpanjang antrean.
   b. Peningkatan Kapasitas
      Pasca-optimasi, *throughput* meningkat secara linear seiring penambahan pengguna, mencapai puncaknya di angka 11,89 req/sec pada 150 *users*. Hal ini membuktikan bahwa kapasitas layanan API meningkat hingga 10 kali lipat, bisa dilihat pada gambar 12.



Gbr. 12 Perbandingan throughput

4. Perbandingan *Error Rate* (Keandalan)
   Metrik ini menunjukkan persentase permintaan yang gagal diproses (misalnya akibat *timeout* atau *server crash*).
   a. *Baseline*
      Sistem dinilai tidak layak produksi (*not production-ready*) karena menghasilkan tingkat kesalahan antara 18% hingga 29% di seluruh skenario pengujian. Pada beban 250 *users*, terjadi kegagalan sistem parsial akibat *resource exhaustion* (RAM penuh).
   b. Pasca-Optimasi
      Optimasi berhasil mengeliminasi kesalahan sepenuhnya (0.00% Error) hingga beban 200 *users*. Pada beban ekstrem 250 *users*, sistem tetap berjalan stabil dengan tingkat kesalahan minor sebesar 0.70%, membuktikan ketahanan (*resilience*) sistem yang jauh lebih baik, bisa dilihat pada gambar 13.



Gbr. 13 Perbandingan error rate

IV. KESIMPULAN DAN SARAN

Berdasarkan hasil penelitian, implementasi, dan pengujian yang telah dilakukan, dapat disimpulkan bahwa penerapan kombinasi caching menggunakan Redis dan query optimization (meliputi indexing dan query rewriting) telah terbukti sangat efektif dalam mengatasi masalah bottleneck serta mereduksi beban komputasi pada sistem monitoring lalu lintas akibat metode polling. Sistem yang dikembangkan mampu menyajikan data langsung dari memori tanpa membebani database utama, yang dibuktikan dengan peningkatan performa drastis pada tiga metrik utama saat load testing. Melalui optimasi ini, rata-rata response time pada skenario normal berhasil dipangkas dari 36.307 ms menjadi hanya 192 ms (di atas 180 kali lipat lebih cepat), kapasitas layanan (throughput) meningkat pesat hingga 10 kali lipat dari 1,1 menjadi 11,89 request per detik, serta mampu mengeliminasi kegagalan sistem secara signifikan dengan mempertahankan error rate sebesar 0% hingga 200 pengguna dan hanya 0,70% pada beban puncak 250 pengguna. Dengan demikian, arsitektur API yang dioptimalkan ini dinilai memiliki ketahanan (resilience) yang sangat baik dan layak digunakan untuk menangani beban trafik tinggi secara efisien.

Meskipun demikian, pengembangan lebih lanjut masih diperlukan untuk meningkatkan skala dan keamanan sistem agar lebih siap diimplementasikan dalam lingkungan produksi nyata yang lebih ekstrem. Fokus penelitian selanjutnya dapat diarahkan pada peningkatan skalabilitas melalui penerapan Load Balancing, migrasi ke layanan cloud dengan fitur auto-scaling guna menangani fluktuasi beban, serta penggunaan message broker untuk mengoptimalkan efisiensi operasi tulis (write) secara asinkron. Selain itu, aspek keamanan RESTful API juga perlu diperkuat dan ditingkatkan melalui implementasi Rate Limiting dan mekanisme autentikasi menggunakan JWT guna mencegah potensi serangan siber serta menjaga integritas data pengguna secara menyeluruh.

V. DAFTAR PUSTAKA

[1] F. Alameka *et al.*, "Implemantasi Rest API untuk E-Herbarium," *Inform. Mulawarman J. Ilm. Ilmu Komput.*, vol. 18, no. 2, pp. 93–98, 2023.

[2] Deepak, "The Impact of API Performance Optimization on User Experience," Deepak.

[3] F. Z. Junaedy and U. Surapati, "Optimisasi Web Service REST API Menggunakan Load Balancer dan Cache dengan Algoritma Round Robin (Studi Kasus: Madani Infosphere)," *J. Indones. Manaj. Inform. dan Komun.*, vol. 5, no. 3, pp. 3158–3169, 2024.

[4] D. E. Septian and E. Hutabri, "Optimasi Sistem Akuntansi Berbasis Web Mengunakan Metode Agile Scrum Studi Kasus PT Segara Catur Perkasa," *J. Inf. Dan Teknol.*, pp. 70–79, 2024.

[5] G. H. Setiawan, I. M. B. Adnyana, and K. Budiarta, "Pengujian Performa API (Application Programming Interface) dengan Metode Load Testing," in *Seminar Nasional CORIS 2022*, 2022.

[6] A. I. Suryawana and A. Muliantaraa, "Database Performance Optimization using Lazy Loading with Redis on Online Marketplace Website," *J. Elektron. Ilmu Komput. Udayana p-ISSN*, vol. 12, no. 3, pp. 627-632, 2024.

[7] S. I. Nurhafida, "Optimasi Query Database Web Scraping Pada Jurnal SINTA," 2022, *Nusa Putra*. [Online]. Available: https://repository.nusaputra.ac.id/id/eprint/445/1/selva indah nurhafida Si22.pdf

[8] A. Ismail, A. Y. Ananta, S. N. Arief, and E. N. Hamdana, "Performance testing sistem ujian online menggunakan jmeter pada lingkungan virtual," J. Inform. Polinema, vol. 9, no. 2, pp. 159–164, 2023.

[9] A. H. T. Harianto, "Analisis Kinerja Mekanisme Caching Redis pada Moodle," *J. Pengemb. Teknol. Inf. dan Ilmu Komput.*, vol. 7, no. 6, pp. 2889–2894, 2023.

[10] U. Usman, W. Widhiarso, and M. Rachmadi, "Implementasi SOLID dan Optimalisasi Query pada Aplikasi Pencatatan Kemiskinan," in MDP Student Conference, 2025, pp. 284–291.