# IMPLEMENTASI gRPC SEBAGAI PROTOKOL KOMUNIKASI APLIKASI MOBILE BERBASIS ANDROID UNTUK DETEKSI DINI PENYAKIT KULIT

**Tugas Akhir**

**Disusun Oleh**
**Muammar Zaki**
**NIM. 220705045**

**Mahasiswa Fakultas Sains dan Teknologi**
**Program Studi Teknologi Informasi**


**FAKULTAS SAINS DAN TEKNOLOGI**
**UNIVERSITAS ISLAM NEGERI AR–RANIRY**
**BANDA ACEH**
**2026/1447**

**LEMBAR PERSETUJUAN**

# IMPLEMENTASI gRPC SEBAGAI PROTOKOL KOMUNIKASI APLIKASI MOBILE BERBASIS ANDROID UNTUK DETEKSI DINI PENYAKIT KULIT

## TUGAS AKHIR

Diajukan kepada Fakultas Sains dan Teknologi
Universitas Islam Negeri (UIN) Ar-Raniry Banda Aceh
Sebagai salah satu Beban Studi Memperoleh Gelar Sarjana (S1)
Dalam Ilmu Teknologi Informasi


Oleh :

**Muammar Zaki**
**NIM. 220705045**

**Mahasiswa Fakultas Sains dan Teknologi**
**Progam Studi Teknologi Informasi**


**Disetujui untuk Dimunaqasyahkan oleh :**

| Pembimbing I, | Pembimbing II, |
|---|---|
| **Mulkan Fadhli, S.T., M.T.** | **Ghufran Ibnu Yasa, M.T.** |
| **NIP. 198811282020121006** | **NIP. 198409262014031005** |

Mengetahui,
**Ketua Program Studi Teknologi informasi,**

**Malahayati, M.T.**
**NIP. 198301272015032003**

**LEMBAR PENGESAHAN**

# IMPLEMENTASI gRPC SEBAGAI PROTOKOL KOMUNIKASI APLIKASI MOBILE BERBASIS ANDROID UNTUK DETEKSI DINI PENYAKIT KULIT

## TUGAS AKHIR

Telah Diuji oleh Panitia Ujian Munaqasyah Tugas Akhir
Fakultas Sains dan Teknologi UIN Ar-Raniry Banda Aceh dan Dinyatakan Lulus
Serta Diterima Sebagai Salah Satu Beban Studi Program Sarjana (S1)
Dalam Program Studi Teknologi Informasi

Pada Hari/Tanggal: Kamis, 07 Mei 2026 M
20 Dzulqaidah 1447 H
Di Darussalam, Banda Aceh

Panitia Ujian Munaqasyah Tugas Akhir:

| Ketua | Sekretaris, |
|---|---|
| **Mulkan Fadhli, S.T., M.T.** | **Ghufran Ibnu Yasa, M.T.** |
| **NIP. 198811282020121006** | **NIP. 198409262014031005** |
| Penguji I, | Penguji II, |
| **Malahayati, M.T. NIP. 198301272015032003** | **Nizam Albar, S.T., M.T.** |
| | **NUPTK. 2849779680130032** |

Mengetahui:
Dekan Fakultas Sains dan Teknologi
UIN Ar-Raniry Banda Aceh

**Prof. Dr. Ir. Muhammad Dirhamsyah, M.T., IPU**
**NIP. 196210021988111001**

## LEMBAR PERNYATAAN KEASLIAN TUGAS AKHIR

Yang bertanda tangan di bawah ini:

| | | |
|---|---|---|
| Nama | : | Muammar Zaki |
| NIM | : | 220705045 |
| Program Studi | : | Teknologi Informasi |
| Fakultas | : | Sains dan Teknologi |
| Judul | : | Implementasi gRPC Sebagai Protokol Komunikasi Aplikasi Mobile Berbasis Android Untuk Deteksi Dini Penyakit Kulit |

Dengan ini menyatakan bahwa dalam penulisan tugas akhir ini, saya:

1. Tidak menggunakan ide orang lain tanpa mampu mengembangkan dan mempertanggungjawabkan;
2. Tidak melakukan plagiasi terhadap naskah karya orang lain;
3. Tidak menggunakan karya orang lain tanpa menyebutkan sumber asli atau tanpa izin pemilik karya;
4. Tidak memanipulasi dan memalsukan data;
5. Mengerjakan sendiri karya ini dan mampu bertanggungjawab atas karya ini.

Bila dikemudian hari ada tuntutan dari pihak lain atas karya saya, dan telah melalui pembuktian yang dapat dipertanggungjawabkan dan ternyata memang ditemukan bukti bahwa saya telah melanggar pernyataan ini, maka saya siap dikenai sanksi berdasarkan aturan yang berlaku di Fakultas Sains dan Teknologi UIN Ar-Raniry Banda Aceh. Demikian pernyataan ini saya buat dengan sesungguhnya dan tanpa paksaan dari pihak manapun.

Banda Aceh, 11 Mei 2026

Yang Menyatakan,


Muammar Zaki

# ABSTRAK

Pesatnya perkembangan mobile health (mHealth) menuntut efisiensi transmisi data, terutama untuk citra medis resolusi tinggi pada diagnosis berbasis *deep learning*. Penelitian ini mengevaluasi implementasi protokol gRPC dengan mekanisme *client-side streaming* untuk pengiriman citra penyakit kulit pada platform Android, dibandingkan dengan pendekatan tradisional REST API (multipart/form-data). Kinerja diukur melalui berbagai simulasi kondisi jaringan, mencakup 4G, 3G, serta lingkungan terdegradasi (*packet loss* dan latensi tinggi). Hasil penelitian menunjukkan bahwa gRPC mampu mencapai *throughput* puncak sebesar 5,1 *requests/second* (RPS) pada kondisi jaringan normal, lebih tinggi dibandingkan REST yang hanya mencapai 3,9 RPS. Selain itu, gRPC secara signifikan mengurangi konsumsi sumber daya, dengan efisiensi memori puncak server sebesar ~29,9% dan beban kerja CPU klien 31% lebih rendah. Namun, ditemukan adanya *trade-off* pada aspek reliabilitas; tingkat keberhasilan (*success rate*) gRPC menurun hingga 82,4% pada jaringan 3G dalam beban berkelanjutan (*soak test*) akibat sensitivitas protokol HTTP/2 terhadap *packet loss* yang terus-menerus. Penelitian ini menyimpulkan bahwa meskipun gRPC menawarkan efisiensi dan *skalabilitas* yang unggul untuk sistem mHealth dengan konkurensi tinggi, mekanisme penanganan kesalahan yang kuat seperti *adaptive chunking* dan auto-retry sangat diperlukan untuk menjaga keandalan pada lingkungan jaringan seluler yang tidak stabil.

**Kata Kunci:** *gRPC, REST API, Android, Mobile Health, Client-Side Streaming, Rekayasa Kinerja.*

## ABSTRACT

The rapid development of mobile health (mHealth) demands highly efficient data transmission, particularly for high-resolution medical imagery utilized in deep learning-based diagnostics. This study evaluates the implementation of the gRPC protocol featuring a client-side streaming mechanism for transmitting skin disease images on the Android platform, comparing it against the traditional REST API (multipart/form-data) approach. Performance was measured across various simulated network conditions, including 4G, 3G, and degraded environments characterized by packet loss and high latency. The experimental results demonstrate that gRPC achieves a peak throughput of 5.1 requests per second (RPS) under normal network conditions, outperforming REST, which only reached 3.9 RPS. Furthermore, gRPC significantly minimizes resource consumption, yielding a peak server memory efficiency of approximately 29.9% and a 31% lower client CPU workload. However, a distinct trade-off in reliability was observed; the gRPC success rate declined to 82.4% during a continuous soak test over a 3G network due to the sensitivity of the underlying HTTP/2 protocol to persistent packet loss. This study concludes that while gRPC offers superior efficiency and scalability for high-concurrency mHealth systems, robust error-handling mechanisms such as adaptive chunking and auto-retry are imperative to maintain reliability in unstable mobile network environments



**Keywords:** gRPC, REST API, Android, Mobile Health, Client-Side Streaming, Performance Engineering.

# KATA PENGANTAR

Bismillahirrahmanirrahim.

Segala puji bagi Allah, Tuhan Yang Maha Esa dan Penguasa semesta alam. Semoga Nabi Muhammad, keluarganya, dan para sahabatnya mendapatkan kedamaian dan berkah. Tugas akhir penulis, "Implementasi gRPC Sebagai Protokol Komunikasi Aplikasi Mobile Berbasis Android Untuk Deteksi Dini Penyakit Kulit" dapat terwujud berkat karunia dan rahmat Allah, Yang Maha Pengasih dan Penyayang. Kriteria untuk memperoleh gelar Sarjana dalam Program Studi Teknologi Informasi di Fakultas Sains dan Teknologi Universitas Islam Negeri Ar-Raniry Banda Aceh meliputi penyelesaian karya ilmiah ini.

Penulis mengucapkan terima kasih yang tulus kepada semua pihak yang telah membantu dalam proses belajar, transfer pengetahuan, dukungan moral, dan bantuan dan dukungan lain yang sangat penting untuk menyelesaikan tugas akhir ini, baik secara langsung maupun tidak langsung. Penulis ingin mengucapkan terima kasih kepada:

1) Kedua orang tua penulis, Bapak M. Jabir, S.E. dan Ibu Susanna, S.Pd., yang senantiasa memberikan kasih sayang, dukungan, motivasi, serta doa yang tiada henti. Cinta dan pengorbanan mereka tidak akan pernah mampu terbalaskan sepenuhnya oleh penulis. Semoga Allah SWT senantiasa melimpahkan rahmat, kesehatan, dan keberkahan kepada mereka, serta memberikan kesempatan untuk menyaksikan keberhasilan dan kemakmuran penulis di masa yang akan datang.
2) Bapak Mulkan Fadhli, S.T, M.T., yang telah berkenan menjadi pembimbing akademik sekaligus pembimbing tugas akhir penulis, serta senantiasa memberikan arahan, masukan, dan dukungan sejak awal perkuliahan hingga penyusunan tugas akhir ini.
3) Bapak Ghufran Ibnu Yasa, M.T., yang bertindak sebagai pembimbing akademik penulis dan secara terus-menerus memberikan arahan dan nasihat sejak awal perkuliahan hingga saat ini.

4) Para dosen Program Studi Teknologi Informasi yang telah membantu penulis dalam menyusun dan menyelesaikan tugas akhir ini dengan memberikan bimbingan dan pengetahuan.

5) Selama proses penyusunan tugas akhir, Ibu Cut Ida Rahmadiana, S.Si., seorang staf Program Studi Teknologi Informasi, telah memberikan bantuan dan dukungan dalam berbagai mata kuliah akademik.

6) Kementerian Komunikasi dan Informatika Republik Indonesia, Bangkit, dan Dicoding atas ilmu, wawasan, dan pengalaman belajar yang sangat berarti sehingga penulis dapat menyelesaikan tugas akhir ini. Penulis juga mengucapkan terima kasih kepada Faqihza Mukhlish melalui Kelas Terbuka serta Sandhika Galih melalui Web Programming UNPAS yang telah menjadi bagian penting dalam perjalanan awal penulis mempelajari pemrograman dan ilmu komputer.

7) Teman-teman dari Program Studi Program Studi Teknologi Informasi angkatan 2022 serta teman-teman dekat yang telah memberikan semangat, doa, dukungan, bantuan, dan kerja sama kepada penulis selama masa studi hingga penyusunan tugas ini.

Penulis dengan jujur mengakui bahwa masih banyak kekurangan dalam tugas akhir ini, baik dari segi penulisan teknis maupun konten. Oleh karena itu, penulis dengan rendah hati menerima kritik dan saran yang bermanfaat untuk pengembangan di masa depan. Terakhir, semoga tugas akhir ini menjadi amal baik yang tercatat sebagai amal baik di mata Allah SWT dan bermanfaat bagi penulis dan pembaca. Rabbal 'Alamin ya Aamiin.

Banda Aceh, 5 Mei 2026

Penulis,

Muammar Zaki

# DAFTAR ISI

# DAFTAR GAMBAR

# DAFTAR TABEL

# DAFTAR LAMPIRAN

# BAB I
# PENDAHULUAN

## 1.1 Latar Belakang

Pesatnya perkembangan teknologi mobile health (mHealth) telah mendorong integrasi kecerdasan buatan/*Artificial Intelligence* (AI) ke dalam perangkat seluler untuk mendukung layanan kesehatan yang lebih *aksesibel*. Dalam konteks dermatologi, deteksi dini penyakit kulit berbasis citra menjadi salah satu domain penelitian yang berkembang signifikan (Hikmah et al., 2025). Mengingat keterbatasan sumber daya pada perangkat seluler (seperti kapasitas memori, daya pemrosesan CPU/GPU, dan baterai), arsitektur aplikasi umumnya menerapkan paradigma *computational offloading*. Dalam paradigma ini, proses inferensi model *Deep Learning* yang berat dipindahkan dari perangkat *client* ke server yang memiliki kapasitas komputasi lebih tinggi (Kumar & Pal, 2025).

Meskipun *computational offloading* menawarkan solusi atas keterbatasan perangkat keras, pendekatan ini menempatkan komunikasi data sebagai komponen paling kritis. Pada aplikasi deteksi penyakit kulit, data yang ditransmisikan adalah citra medis resolusi tinggi yang ukurannya relatif besar (umumnya di atas 1 MB) guna menjaga detail visual lesi kulit (Katz et al., 2025). Tantangan muncul ketika aplikasi digunakan pada kondisi infrastruktur jaringan seluler yang tidak ideal. Fluktuasi *bandwidth*, latensi yang tinggi, serta *packet loss* yang sering terjadi pada jaringan seluler dapat menyebabkan kegagalan pengiriman data (*request timeout*) (Taha et al., 2024). Kegagalan ini berdampak langsung pada *User Experience* (UX), sehingga pengguna harus mengulang proses pengambilan dan pengunggahan gambar dari awal.

Saat ini, *Representational State Transfer (*REST*)* API berbasis protokol HTTP/1.1 masih menjadi standar *de facto* dalam pengembangan aplikasi Android karena *interoperabilitasnya* yang luas (Hari & Sharma, 2024). Namun, REST memiliki keterbatasan mendasar dalam penanganan data biner berukuran besar di jaringan yang tidak stabil (Grigorik, 2013). Meskipun REST API mampu mengirimkan data biner secara efisien melalui mekanisme *multipart/form-data*,

pendekatan ini memiliki keterbatasan arsitektural yang signifikan dalam skenario jaringan seluler yang tidak stabil. Pengiriman data pada REST API umumnya bersifat monolitik, sehingga keseluruhan citra medis berukuran besar harus dikirim dalam satu siklus *request–response* HTTP/1.1. Pada kondisi jaringan dengan latensi tinggi dan *packet loss*, kegagalan pada sebagian kecil transmisi dapat menyebabkan pemutusan koneksi TCP dan memicu *request timeout*, sehingga proses pengunggahan harus diulang dari awal (Kurose & Ross, 2024). Selain itu, karakteristik komunikasi HTTP/1.1 yang sinkron juga berpotensi mengalami *Head-of-Line* (HOL) *Blocking*, yang semakin memperburuk keandalan transmisi data berukuran besar pada jaringan seluler.

Penelitian terdahulu yang dilakukan oleh Niswar et al. (2024) menunjukkan bahwa gRPC unggul dalam performa *throughput* dibandingkan REST pada arsitektur *microservices* di lingkungan *backend*. Namun, pemrosesan *computational offloading* pada lingkungan *mobile health* (mHealth) memiliki karakteristik berbeda, dengan tantangan utama yang tidak hanya berkaitan dengan kecepatan, tetapi juga limitasi *bandwidth* dan tingginya *packet loss* pada jaringan seluler. Kebanyakan literatur saat ini belum mengevaluasi secara komparatif pengaruh mekanisme pemecahan data (*chunking*) pada *client-side streaming* gRPC terhadap keandalan pengiriman (*delivery rate*) citra medis beresolusi tinggi dibandingkan transmisi monolitik pada REST API. Oleh karena itu, penelitian ini mengisi celah tersebut dengan mengevaluasi *trade-off* antara latensi waktu penyelesaian dan tingkat keberhasilan pengiriman citra penyakit kulit pada perangkat Android di bawah berbagai simulasi degradasi jaringan seluler (Fielding, 2000; Hari & Sharma, 2024).

Penelitian ini tidak hanya bertujuan membandingkan performa, tetapi menekankan pada aspek kontrol transmisi data. Dengan mekanisme *streaming*, aplikasi diharapkan mampu mempertahankan koneksi dan integritas data meskipun terjadi degradasi kualitas jaringan. Penelitian ini akan mengimplementasikan dan mengevaluasi kinerja gRPC *client streaming* dibandingkan dengan REST API pada aplikasi deteksi dini penyakit kulit berbasis Android, dengan fokus pengujian pada kondisi jaringan yang disimulasikan mengalami limitasi *bandwidth* dan gangguan *packet loss*.

## 1.2 Rumusan Masalah

Berdasarkan latar belakang yang telah dijelaskan, terdapat beberapa titik fokus kendala yang ingin diselesaikan dalam penelitian ini:

1) Bagaimana pengaruh penerapan mekanisme *client-side streaming* pada gRPC terhadap tingkat keberhasilan pengiriman (*success rate*) dan waktu penyelesaian (*completion time*) transmisi citra medis berukuran besar dibandingkan dengan pendekatan monolitik REST API pada kondisi jaringan seluler yang terdegradasi?
2) Bagaimana perbandingan efisiensi transmisi (ukuran *payload* jaringan) antara mekanisme pemecahan data gRPC dengan *multipart/form-data REST API*?
3) Bagaimana dampak penggunaan kedua protokol tersebut terhadap utilitas komputasi perangkat *client* (alokasi memori RAM dan waktu CPU) selama proses pengiriman berlangsung?
4) Bagaimana kualitas layanan/*Quality of Service* (QoS) backend gRPC dibandingkan dengan REST API dalam menangani lonjakan konkurensi lalu lintas data berdasarkan metrik *latency* dan *throughput*, serta stabilitas memori server saat menerima transmisi citra dalam skala besar?

## 1.3 Tujuan Penelitian

Dari hasil perumusan masalah yang telah dipaparkan, penelitian ini bertujuan untuk mencapai sasaran sebagai berikut:

1) Menganalisis pengaruh penerapan mekanisme *client-side streaming* pada gRPC terhadap tingkat keberhasilan pengiriman (*success rate*) dan waktu penyelesaian (*completion time*) transmisi citra medis berukuran besar dibandingkan dengan pendekatan monolitik REST API pada kondisi jaringan seluler yang terdegradasi.
2) Menganalisis perbandingan efisiensi transmisi data berdasarkan ukuran *payload* jaringan antara mekanisme pemecahan data (*chunking*) pada gRPC *client-side streaming* dan pendekatan *multipart/form-data* pada REST API dalam pengiriman citra medis berukuran besar.

3) Menganalisis dampak penggunaan *gRPC client-side streaming* dan REST API monolitik terhadap utilitas sumber daya komputasi perangkat client, khususnya penggunaan memori RAM dan waktu pemrosesan CPU, selama proses pengiriman citra medis berlangsung.

4) Menganalisis kualitas layanan/*Quality of Service* (QoS) arsitektur backend gRPC dibandingkan dengan REST API dalam menangani lonjakan konkurensi lalu lintas data, berdasarkan metrik *throughput* dan *latency*, serta stabilitas penggunaan memori server selama transmisi citra medis berukuran besar secara simultan.

## 1.4 Manfaat Penelitian

Sejalan dengan tujuan yang telah ditetapkan, penelitian ini diharapkan memberikan manfaat sebagai berikut:

1) Memberikan bukti empiris mengenai kinerja protokol gRPC pada ekosistem perangkat mobile, khususnya dalam skenario transmisi data citra medis berukuran besar pada kondisi jaringan seluler yang tidak stabil, sehingga dapat memperkaya literatur rekayasa perangkat lunak yang selama ini lebih banyak berfokus pada implementasi gRPC di lingkungan *backend microservices*.

2) Menjadi rujukan akademik terkait karakteristik keandalan dan integritas data pada mekanisme komunikasi gRPC *client-side streaming* dibandingkan dengan pendekatan REST API dalam pengiriman data citra medis berukuran besar pada jaringan berlatensi tinggi.

## 1.5 Batasan Penelitian

Untuk menjaga fokus dan ketercapaian tujuan penelitian, maka penelitian ini memiliki beberapa batasan sebagai berikut:

1) Implementasi sisi *client* terbatas pada platform Android.

2) Penelitian difokuskan pada mekanisme transmisi citra menggunakan gRPC dan inferensi model.

3) Menggunakan model pra-latih/*pre-trained model* format ONNX yang sudah tersedia. Penelitian tidak mencakup proses pelatihan (*training*) ulang atau modifikasi arsitektur model AI.

4) Pengujian terbatas pada aspek kinerja teknis (stabilitas koneksi, *delivery rate*, integritas data, dan latensi), tidak mencakup uji validitas medis ataupun tingkat kepuasan pengguna /*User Acceptance Testing*.

# BAB II
# TINJAUAN PUSTAKA

## 2.1 Penelitian Terdahulu

Bagian ini meninjau penelitian terdahulu terkait arsitektur komunikasi data pada perangkat mobile, dengan fokus khusus pada keterbatasan protokol REST dalam menangani transmisi citra medis. Pembahasan diarahkan untuk menyoroti urgensi teknis implementasi gRPC sebagai solusi guna menjamin integritas data dan stabilitas koneksi yang krusial bagi keandalan sistem.

Tabel 2. 1 Penelitian Terkait

| Judul & Peneliti | Metode | Kesimpulan |
|---|---|---|
| *Imtidad: A Reference Architecture and a Case Study on Developing Distributed AI Services for Skin Disease Diagnosis over Cloud, Fog and Edge;* Janbi et al. (2022) | Studi kasus implementasi arsitektur referensi (*Imtidad*) dan evaluasi performa distribusi beban kerja. | Mengidentifikasi bahwa latensi jaringan adalah kendala utama pada diagnosis berbasis Cloud terpusat. Distribusi beban ke *Edge/Fog* terbukti meningkatkan responsivitas. |
| *Secure Computational Offloading with gRPC: A Performance Evaluation in a Mobile Cloud Computing Environment*; de Matos et al. (2021) | Eksperimen pengukuran performa dan konsumsi energi menggunakan protokol gRPC pada Android. | membuktikan bahwa penerapan *computational offloading* menggunakan gRPC mampu menurunkan konsumsi energi baterai hingga 88% dibandingkan jika proses tersebut dilakukan secara lokal (*local processing*) pada perangkat mobile. |
| *Performance evaluation of microservices communication with REST, GraphQL, and gRPC;* | *Benchmarking* kuantitatif mengukur *throughput*, penggunaan CPU/Memori, dan | gRPC menunjukkan *throughput* tertinggi dan penggunaan sumber daya paling efisien |

| | | |
|---|---|---|
| Niswar et al. (2024) | latensi antara REST, GraphQL, dan gRPC. | dibandingkan REST dan GraphQL. |
| *Dermatologist-level classification of skin cancer with deep neural networks;*<br>Esteva et al. (2017) | Pelatihan model *Convolutional Neural Network* (CNN) menggunakan dataset masif (129.000+ citra klinis). | Membuktikan algoritma CNN mampu mencapai tingkat akurasi diagnostik yang setara dengan dokter spesialis kulit. |
| *Analisis Perbandingan Efektivitas Arsitektur Restful dan Arsitektur GRPC pada Implementasi Web Service*;<br>Yanuardi et al. (2024) | Analisis komparatif parameter *Quality of Service* (QoS) antara arsitektur RESTful dan gRPC. | Menunjukkan bahwa dalam skenario Web Service dengan beban data bervariasi, gRPC menawarkan stabilitas koneksi yang lebih baik daripada RESTful API. |
| *OsiriXgRPC: Rapid development and deployment of state-of-the-art artificial intelligence for clinical practice;*<br>Mun et al. (2022) | Pengembangan plugin berbasis gRPC untuk mentransfer data citra medis (MRI/CT) antara penampil gambar dan server AI. | Membuktikan bahwa gRPC andal dan aman untuk menangani transfer data citra medis (MRI/CT) yang besar, serta memvalidasi efisiensi plugin berbasis gRPC dalam lingkungan klinis nyata. |

Berdasarkan sintesis pada Tabel 2.1, berbagai studi komparatif telah mengonfirmasi keunggulan gRPC dibandingkan REST atau GraphQL dari berbagai parameter.

1) Kecepatan & *Throughput*, Niswar et al. (2024) secara konsisten menemukan bahwa gRPC menawarkan *throughput* tertinggi dan latensi terendah dibandingkan REST, terutama karena penggunaan HTTP/2 dan kompresi *header*.

2) Efisiensi Sumber Daya, penelitian de Matos et al. (2021) menyoroti efisiensi energi yang signifikan pada perangkat Android, menjadikannya protokol yang ideal untuk skenario *mobile computing*.

3) Stabilitas Koneksi, Yanuardi et al. (2024) menekankan bahwa gRPC memiliki stabilitas koneksi/*Quality of Service* yang lebih baik saat menangani beban data bervariasi.

Kebaruan penelitian ini terletak pada evaluasi mekanisme *client-side streaming* pada gRPC sebagai strategi kontrol transmisi untuk meningkatkan keandalan pengiriman citra medis berukuran besar pada kondisi jaringan seluler yang terdegradasi. Berbeda dengan penelitian sebelumnya yang berfokus pada pengukuran performa di lingkungan backend atau jaringan stabil, penelitian ini secara spesifik menganalisis dampak mekanisme pemecahan data (*chunking*) terhadap metrik keberhasilan pengiriman (*success rate*) dan waktu penyelesaian (*completion time*) pada perangkat Android. Dengan demikian, penelitian ini tidak hanya menguji efisiensi komunikasi gRPC, tetapi juga memberikan bukti empiris mengenai trade-off antara latensi dan keandalan transmisi data dalam skenario jaringan yang tidak ideal.

Oleh karena itu, penelitian ini mengisi kesenjangan tersebut dengan membangun dan mengevaluasi sistem berbasis gRPC untuk transmisi citra medis pada aplikasi mobile, dengan fokus pada peningkatan keandalan komunikasi data melalui pendekatan *streaming*.

## 2.2 Landasan Teori

Pada bagian ini dijelaskan teori-teori yang menjadi dasar dan acuan dalam pelaksanaan penelitian.

### 2.2.1 Platform Android

Platform Android saat ini telah bertransformasi menjadi ekosistem pengembangan modern dengan diadopsinya Kotlin sebagai bahasa pemrograman utama yang mendukung efisiensi komputasi mobile. Dalam konteks penelitian ini, penggunaan Android API menjadi aspek fundamental karena dukungan fitur manajemen memori dan *concurrency* melalui Kotlin Coroutines. Kemampuan ini sangat krusial dalam implementasi arsitektur *client-side streaming* gRPC, aplikasi

dituntut untuk memecah data citra menjadi segmen-segmen kecil (*chunking*) dan mengirimkannya secara asinkron tanpa memblokir *Main Thread*, sehingga menjamin aplikasi tetap responsif terhadap interaksi pengguna selama proses transmisi data berlangsung (Android Developers, 2025).

Selain kapabilitas manajemen proses, Android menyediakan antarmuka perangkat keras yang aman untuk akuisisi citra medis. Melalui integrasi komponen sistem seperti Photo Picker dan Content Resolver, platform memungkinkan aplikasi untuk mengakses data biner citra resolusi tinggi secara langsung dari penyimpanan lokal dengan izin yang terisolasi (Android, 2025). Fleksibilitas akses ini mendukung penerapan mekanisme pra-pemrosesan/*preprocessing* di sisi *client* seperti konversi format citra menjadi aliran *byte array* sebelum data dikirimkan ke *server*, memastikan integritas struktur data yang sesuai dengan definisi kontrak gRPC dan kebutuhan input model *Deep Learning* (Android Developers, 2025).

### 2.2.2 Komunikasi Data

Terdapat berbagai jenis metode komunikasi data yang digunakan dalam pengembangan aplikasi modern, di antaranya adalah *Representational State Transfer* (REST) API, WebSocket, GraphQL, dan *Remote Procedure Call* (RPC) (Steen & Tanenbaum, 2017). Setiap metode memiliki karakteristik dan tujuan yang berbeda.

#### 2.2.2.1 REST API

Arsitektur *Representational State Transfer* (REST) merupakan gaya arsitektur komunikasi *client-server* yang pertama kali didefinisikan oleh (Fielding, 2000) dalam disertasinya. Gaya arsitektur ini telah diadopsi secara luas dalam pengembangan aplikasi web dan menjadi salah satu pilar dalam implementasi layanan mikro/*microservices* (Newman, 2015). REST memanfaatkan metode-metode standar HTTP (seperti GET, POST, PUT, DELETE) untuk melakukan operasi pada sumber daya (*resources*), yang diidentifikasi melalui URI (*Uniform Resource Identifiers*).

Prinsip inti dari REST adalah *statelessness* (tanpa status), yang mengharuskan setiap permintaan dari *client* ke server berisi semua informasi yang diperlukan untuk memproses permintaan tersebut, tanpa server perlu menyimpan konteks sesi

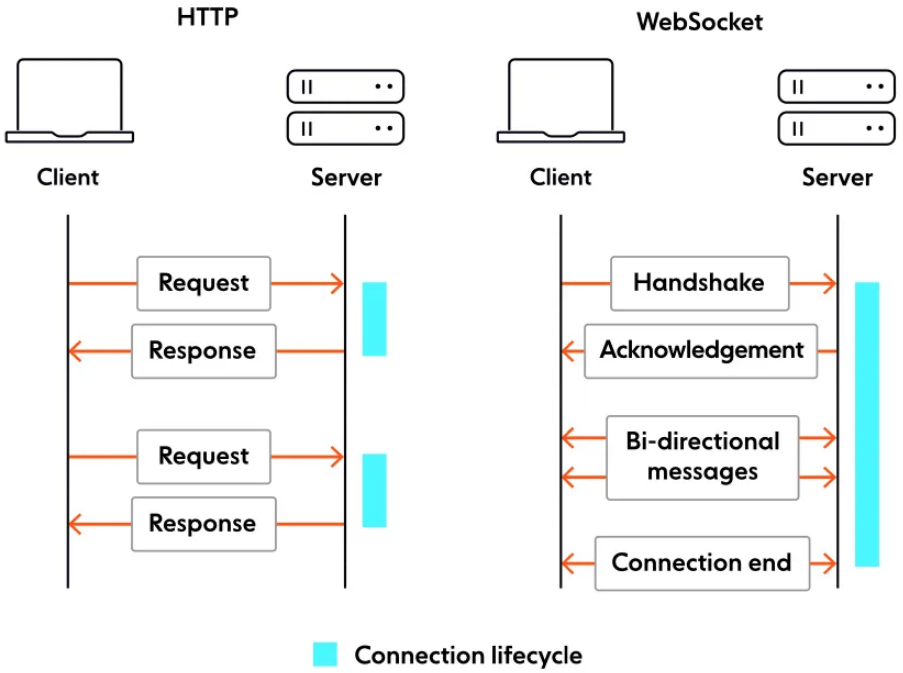
*client* (Fielding, 2000). Salah satu keuntungan utama REST API adalah kesederhanaan, *interoperabilitas*, dan dukungannya yang luas di berbagai platform, menjadikannya pilihan populer untuk layanan berbasis CRUD (Create, Read, Update, Delete) (Richardson & Ruby, 2007).

Meskipun demikian, penggunaan REST API juga memiliki kelemahan, terutama dalam skenario pengambilan data yang kompleks. REST dapat mengalami masalah *over-fetching* (mengambil data lebih banyak dari yang dibutuhkan) atau *under-fetching* (memerlukan beberapa panggilan API untuk mengumpulkan semua data yang diperlukan), yang dapat memengaruhi efisiensi dan performa (Quiña-Mera et al., 2023). Selain itu, arsitektur REST seringkali menuntut pengelolaan banyak *endpoint* untuk setiap sumber daya. Karena itu, dalam pengembangan aplikasi, pemilihan REST API sebagai mekanisme komunikasi harus mempertimbangkan secara cermat beban data, frekuensi permintaan, serta skenario caching dan skalabilitas sistem.

#### 2.2.2.2 WebSocket

WebSocket adalah protokol komunikasi yang didefinisikan oleh *Internet Engineering Task Force* (IETF) untuk menyediakan kanal komunikasi dua arah *(full-duplex)* yang bersifat *persisten* antara *client* dan server melalui satu koneksi TCP tunggal (Melnikov & Fette, 2011). Berbeda fundamental dari model *request-response* HTTP, WebSocket memungkinkan server untuk secara aktif mengirimkan *(push)* data ke *client* kapan saja tanpa *client* perlu melakukan permintaan baru (Grigorik, 2013).

Perbedaan fundamental mekanisme koneksi antara HTTP tradisional dan WebSocke*t* dapat dilihat pada Gambar 2.1. Ilustrasi tersebut menunjukkan bahwa WebSocket mempertahankan koneksi dua arah (*bi-directional*) yang *persisten* setelah *handshake* awal, berbeda dengan HTTP yang harus membuka koneksi baru untuk setiap permintaan.




Gambar 2. 1 Perbedaan Cara Kerja Http Dan WebSocket

sumber: *(O'Riordan, 2024)*

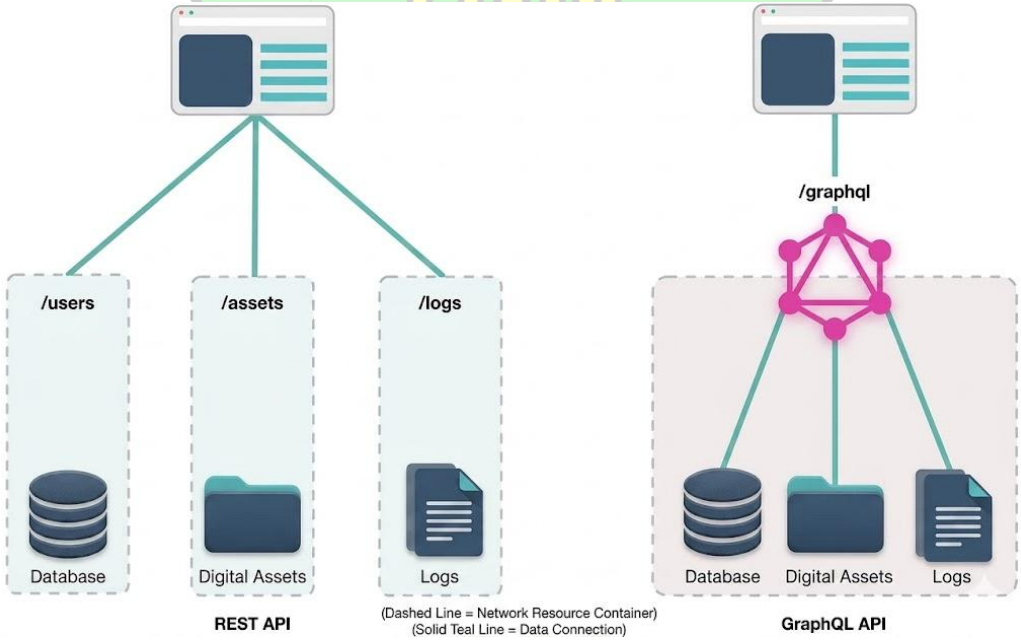
Meskipun demikian, implementasi WebSocket membawa pertimbangan arsitektural yang berbeda dari HTTP. Sifatnya yang stateful (koneksi harus tetap terbuka) menuntut manajemen sumber daya yang lebih kompleks di sisi server, terutama terkait penanganan batas koneksi, timeout, dan strategi skalabilitas horizontal (Grigorik, 2013). Oleh karena itu, dalam konteks aplikasi yang melibatkan event aktif dan notifikasi mendekati waktu nyata, WebSocket dapat menjadi pilihan yang tepat apabila jaminan komunikasi real-time merupakan kebutuhan kritis.

Namun, dalam konteks pengiriman citra medis berukuran besar, WebSocket kurang efisien karena *overhead framing* yang tinggi untuk data biner dan ketiadaan fitur kontrol aliran (*flow control*) bawaan yang sekuat HTTP/2 pada gRPC. Oleh karena itu, WebSocket tidak dipilih dalam penelitian ini yang lebih memprioritaskan integritas pengiriman data biner daripada latensi pesan teks real-time.

#### 2.2.2.3 GraphQL

GraphQL adalah sebuah bahasa kueri (*query language*) untuk API beserta runtime sisi server untuk mengeksekusi kueri tersebut (Quiña-Mera et al., 2023). Fitur utamanya adalah memungkinkan *client* untuk mendefinisikan secara eksplisit struktur data yang dibutuhkan dalam sebuah kueri. Server kemudian merespons dengan mengembalikan data yang hanya sesuai dengan struktur yang diminta tersebut. Pendekatan ini secara efektif meminimalkan masalah *over-fetching* (menerima data lebih banyak dari yang dibutuhkan) dan *under-fetching* (memerlukan beberapa panggilan API untuk mendapatkan data yang lengkap), dua masalah yang sering terjadi pada implementasi REST tradisional (Quiña-Mera et al., 2023).

Efisiensi pengambilan data pada GraphQL divisualisasikan pada Gambar 2.2, *client* dapat mendefinisikan struktur data spesifik dalam satu request tunggal, menghindari pola *multiple round-trip* yang sering terjadi pada REST API.




Gambar 2. 2 Proses *Request* Dari Graphql

sumber: *(Lagdhir, 2025)*

Berbeda dengan arsitektur REST yang umumnya mengekspos banyak *endpoint* untuk sumber daya yang berbeda, GraphQL secara tipikal hanya mengekspos satu *endpoint* (Fielding, 2000; Quiña-Mera et al., 2023). Inti dari

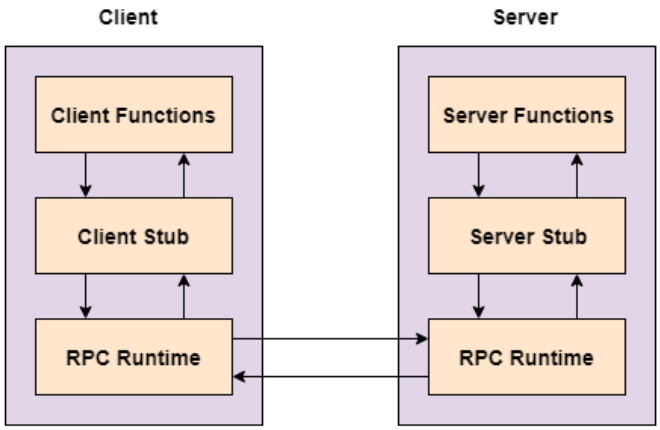
arsitektur GraphQL adalah skema yang kuat (*strongly typed schema*). Skema ini berfungsi sebagai kontrak terperinci antara *client* dan server, mendefinisikan semua operasi dan tipe data yang tersedia, sehingga membuat API menjadi lebih fleksibel dan adaptif terhadap perubahan kebutuhan di sisi front-end.

Selain itu, mekanisme caching dapat menjadi lebih kompleks dibandingkan dengan REST. Karena sebagian besar kueri GraphQL dikirim melalui permintaan *HTTP POST* ke satu *endpoint* statis, *caching* level HTTP bawaan peramban *(yang bergantung pada URL endpoint GET)* menjadi tidak dapat diandalkan, sehingga memerlukan strategi *caching* di level aplikasi (Grigorik, 2013; Quiña-Mera et al., 2023).

Meskipun GraphQL unggul dalam menghindari *over-fetching* data tekstual, protokol ini tidak dirancang secara *native* untuk menangani streaming data biner (*file upload*) secara efisien tanpa ekstensi tambahan yang kompleks. Karena fokus penelitian ini adalah transmisi citra dalam bentuk *blob*, gRPC dinilai lebih relevan dibandingkan GraphQL.

#### 2.2.2.4 RPC

*Remote Procedure Call* (RPC) adalah sebuah model komunikasi yang memungkinkan sebuah program (*client*) meminta layanan dari sebuah program (server) yang mungkin berada di komputer atau jaringan yang berbeda, tanpa *client* tersebut harus memahami detail jaringan secara eksplisit (Birrell & Nelson, 1984). RPC bekerja dengan model *client-server* yang memungkinkan *client* memanggil sebuah prosedur jarak jauh. Panggilan ini kemudian diterima oleh stub *client*, yang bertugas mengemas parameter (*marshalling*), dan mengirimkannya melalui jaringan ke *stub* server (Steen & Tanenbaum, 2017). Mekanisme pengemasan parameter (marshalling) dan pengiriman pesan antar-*stub client* dan server diilustrasikan pada Gambar 2.3.




Gambar 2. 3 Alur Komunikasi RPC

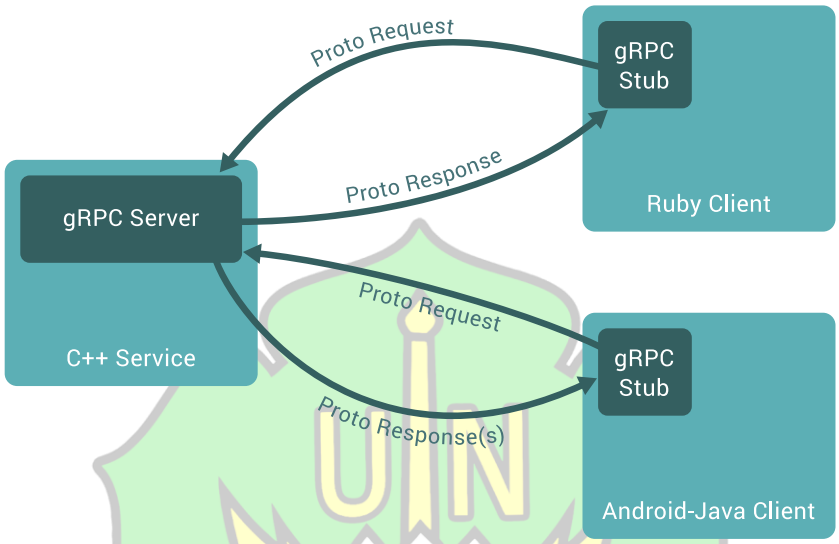
sumber: *(Onsman, 2023)*

Di sisi server, *stub* server menerima pesan, membongkar parameter *(unmarshalling)*, dan memanggil prosedur lokal yang sebenarnya. Setelah eksekusi selesai, hasilnya dikembalikan ke *client* melalui proses yang sama secara terbalik (Coulouris et al., 2011). Tujuan utama dari desain RPC adalah untuk mencapai transparansi lokasi; idealnya, pemanggilan prosedur jarak jauh harus terlihat identik dengan pemanggilan prosedur lokal dari sudut pandang pemrogram (Birrell & Nelson, 1984). Dalam konteks sistem terdistribusi, RPC memberikan kemudahan pengembangan karena memungkinkan pengembang untuk fokus pada logika aplikasi, bukan pada kompleksitas komunikasi jaringan.

### 2.2.3 gRPC

gRPC adalah kerangka kerja komunikasi yang bersifat open source dan dikembangkan pertama kali oleh Google (Indrasiri & Kuruppu, 2020). Secara teknis, gRPC memungkinkan aplikasi untuk berkomunikasi antar proses atau antar perangkat melalui jaringan dengan menggunakan konsep *Remote Procedure Call* (RPC), yaitu mekanisme pemanggilan fungsi yang berjalan di server seolah-olah dijalankan secara lokal di *client*. gRPC menggunakan *Protocol Buffers* (Protobuf) sebagai format serialisasi data, yang lebih efisien dibandingkan format berbasis teks seperti JSON atau XML. Efisiensi ini membuat gRPC sangat cocok untuk aplikasi real-time, berskala besar, serta memerlukan latensi rendah (Makadia, 2025).

Arsitektur komunikasi gRPC antara *client* dan server ditunjukkan pada Gambar 2.4. Pada sisi server, layanan diimplementasikan dalam sebuah gRPC Server yang berjalan di atas *C++* Service dan bertanggung jawab memproses permintaan dari *client*. Sementara itu, pada sisi *client* terdapat aplikasi dengan bahasa pemrograman yang berbeda, yaitu Ruby *Client* dan Android-Java *Client*, yang berinteraksi dengan server melalui gRPC *Stub*.




Gambar 2. 4 Arsitektur Komunikasi gRPC Multibahasa

Sumber: (gRPC.io, 2025)

Beberapa keunggulan gRPC dibandingkan arsitektur komunikasi berbasis REST adalah:

1) Efisiensi Data, protobuf menghasilkan *payload* data yang lebih kecil dan lebih cepat diproses.
2) Multi-bahasa, gRPC mendukung berbagai bahasa pemrograman seperti Java, Kotlin, Python, Go, C++, dan lainnya.
3) Dukungan *Streaming*, gRPC mendukung komunikasi *server streaming, client streaming,* serta *bidirectional streaming*.
4) Keamanan, gRPC memiliki integrasi native dengan TLS (*Transport Layer Security*) sehingga lebih aman untuk komunikasi *client*-server.
5) Kontrak Ketat (*Strict Contract*), penggunaan *file .proto* menjamin struktur data yang dikirim dan diterima selalu konsisten (*strongly typed*), meminimalkan risiko kesalahan format data yang sering terjadi pada JSON.

Penelitian terdahulu menunjukkan potensi besar gRPC dalam meningkatkan performa aplikasi. Niswar et al. (2024) membandingkan kinerja REST API dengan gRPC dan menemukan bahwa gRPC secara konsisten memberikan *throughput* tertinggi dan latensi terendah pada arsitektur layanan mikro. Selain itu, penelitian oleh Mun et al. (2022) menyimpulkan bahwa gRPC lebih unggul dalam menangani komunikasi intensif data seperti pada aplikasi kesehatan yang membutuhkan pemrosesan gambar medis.

### 2.2.4 Penyakit Kulit

Beberapa penyakit kulit yang umum diantaranya adalah jerawat (*acne*), eksim (*eczema*), psoriasis, vitiligo, dermatitis, hingga kanker kulit (*melanoma*). Penyakit-penyakit ini biasanya menimbulkan perubahan pada tekstur, warna, bentuk, maupun pola permukaan kulit, yang dapat diamati secara visual. Karakteristik visual tersebut menjadikan penyakit kulit relatif lebih mudah untuk dianalisis menggunakan teknologi pengolahan citra digital.

Perubahan visual pada kulit dapat dianalisis melalui citra digital berkualitas tinggi, baik yang diambil langsung maupun yang tersimpan dalam perangkat, kemudian diolah lebih lanjut untuk mendeteksi pola yang mengindikasikan suatu penyakit. Dengan demikian, penyakit kulit termasuk salah satu objek medis yang sangat potensial untuk dianalisis menggunakan pendekatan berbasis *Computer Vision* (Debelee, 2023). Dalam konteks machine learning, khususnya deep learning, algoritma *Convolutional Neural Network* (CNN) terbukti unggul dalam pengenalan pola citra. CNN dirancang untuk mengekstraksi fitur spasial dari gambar melalui operasi konvolusi, *pooling*, dan aktivasi non-linear (Malik et al., 2024). Hal ini sangat sesuai untuk mengidentifikasi detail visual seperti warna, tekstur, batas tepi (*edges*), dan bentuk lesi kulit.

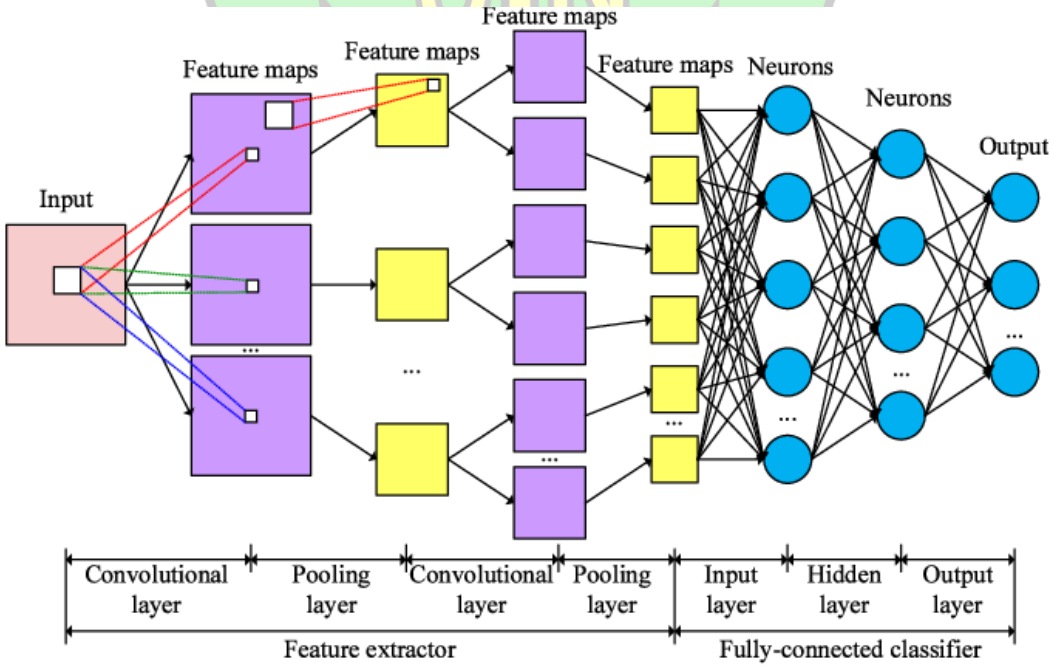
### 2.2.5 Model Machine Learning

*Machine Learning* (ML) adalah cabang dari *Artificial Intelligence* (AI) yang memungkinkan sistem komputer untuk belajar dari data tanpa diprogram secara eksplisit (Kelly & Xiu, 2023). Prinsip utama ML adalah membuat model yang dapat mengenali pola berdasarkan data latih (*training data*) dan kemudian mampu melakukan prediksi pada data baru (*test data*). Model ML sangat luas aplikasinya,

mulai dari pengenalan suara, pemrosesan bahasa alami, hingga pengenalan citra medis.

Dalam konteks analisis citra medis, salah satu pendekatan yang paling banyak digunakan adalah Deep Learning, khususnya *Convolutional Neural Network* (CNN). CNN dirancang untuk mengekstraksi fitur spasial dari citra melalui lapisan konvolusi (*convolutional layers*), *pooling*, dan *fully connected layers* (Malik et al., 2024). Kemampuan CNN dalam mengidentifikasi pola visual seperti warna, tekstur, tepi (*edges*), dan bentuk menjadikannya sangat efektif untuk klasifikasi penyakit kulit.

Arsitektur umum CNN untuk klasifikasi citra disajikan pada Gambar 2.5, yang memperlihatkan tahapan ekstraksi fitur melalui lapisan konvolusi dan pooling sebelum masuk ke lapisan klasifikasi (*fully-connected layer*) untuk menghasilkan prediksi.




Gambar 2. 5 Arsitektur Umum Dari CNN

sumber: https://pemrogramanmatlab.com/

Selain CNN, model *machine learning* lain seperti Support Vector Machine (SVM) dan Random Forest juga digunakan dalam klasifikasi penyakit kulit.

Namun, hasil penelitian secara konsisten menunjukkan bahwa CNN memiliki performa paling tinggi karena kemampuannya mengekstraksi fitur secara otomatis tanpa memerlukan rekayasa fitur manual (Hadi et al., 2021; Kushartanto et al., 2024). Berdasarkan studi literatur tersebut, CNN dipilih karena terbukti memiliki performa ekstraksi fitur spasial yang unggul untuk klasifikasi citra medis. CNN tidak hanya memberikan akurasi tinggi, tetapi juga memungkinkan penerapan nyata melalui integrasi dengan aplikasi Android berbasis gRPC.

### 2.2.6 Mekanisme *Chunking*

Transmisi data pada aplikasi mobile health (mHealth) sering kali dihadapkan pada tantangan pengiriman berkas berukuran besar, seperti citra medis resolusi tinggi yang umumnya membutuhkan kapasitas di atas 1 MB guna mempertahankan detail visual lesi kulit (Katz et al., 2025). Mengirimkan data berukuran masif secara utuh atau monolitik dalam satu siklus permintaan menuntut alokasi memori yang sangat besar secara instan pada perangkat klien. Hal ini menjadi masalah krusial mengingat perangkat seluler memiliki keterbatasan pada kapasitas memori akses acak (RAM) dan daya pemrosesan CPU (de Matos et al., 2021). Pendekatan transmisi monolitik, seperti yang lazim diterapkan pada protokol REST API berbasis HTTP/1.1, sangat rentan membebani memori klien secara instan dan berisiko memicu kegagalan transmisi apabila terjadi degradasi kualitas jaringan seluler.

Sebagai solusi teknis atas keterbatasan sumber daya dan stabilitas jaringan tersebut, mekanisme pemecahan data atau chunking diimplementasikan pada lapisan komunikasi. Chunking merupakan teknik arsitektural yang memecah aliran data biner yang besar menjadi segmen-segmen berukuran kecil (chunks) sebelum ditransmisikan secara berurutan melalui jaringan. Pada kondisi infrastruktur jaringan nirkabel, fluktuasi bandwidth, latensi tinggi, serta hilangnya paket data (packet loss) sering terjadi dan dapat menyebabkan pemutusan koneksi Transmission Control Protocol (TCP) secara prematur yang berujung pada request timeout (Kurose & Ross, 2024; Taha et al., 2024). Dengan memecah aliran data menggunakan mekanisme client-side streaming pada gRPC, sistem dapat mengirimkan data secara bertahap. Pendekatan ini memungkinkan aplikasi

memiliki kontrol granular terhadap proses pengiriman, sehingga kegagalan transmisi dapat diisolasi pada chunk tertentu tanpa harus menggagalkan keseluruhan proses dan memaksa pengguna mengulang pengunggahan dari awal.

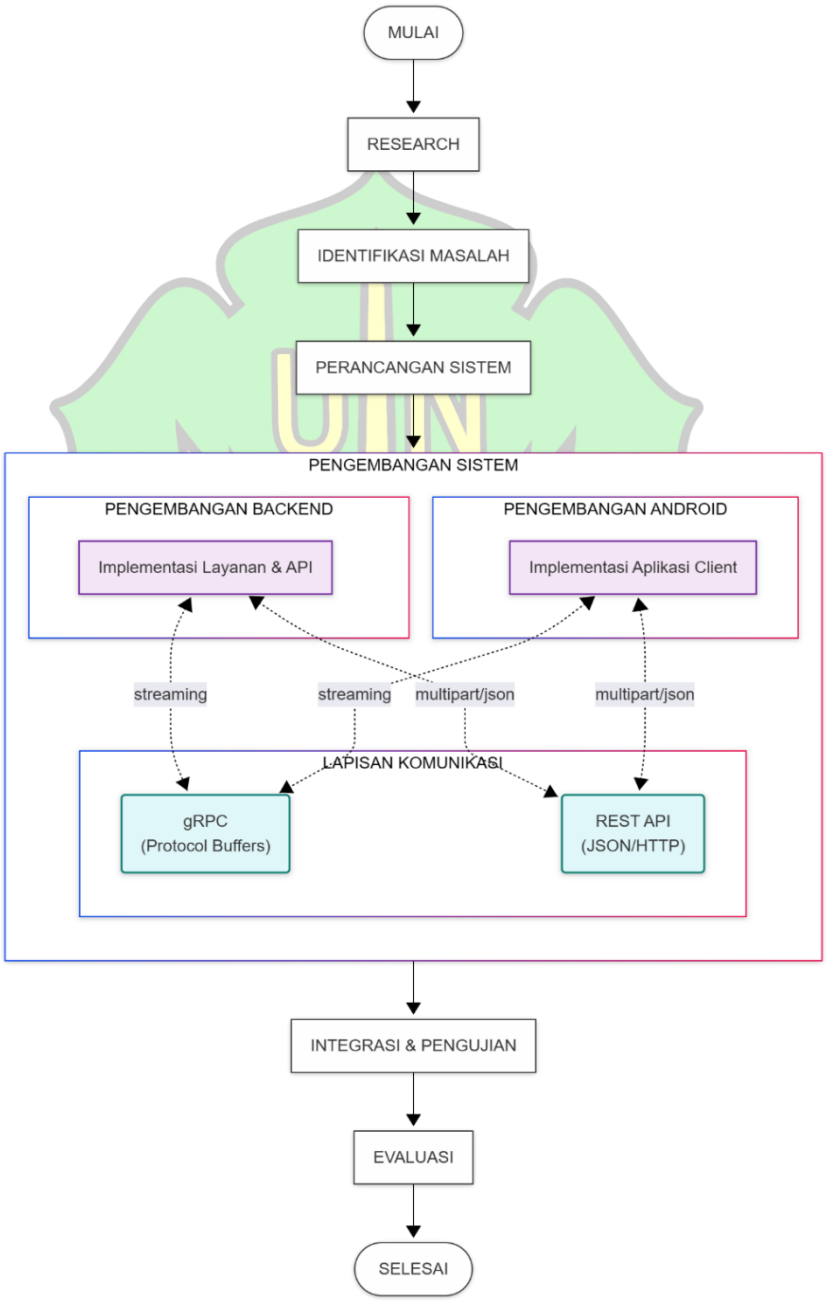
Lebih lanjut, efektivitas mekanisme chunking sangat bergantung pada penentuan ukuran segmen data yang digunakan selama proses transmisi. Ukuran chunk perlu mempertimbangkan keseimbangan antara efisiensi pemanfaatan bandwidth jaringan seluler dan meminimalkan overhead komunikasi yang dihasilkan. Secara konseptual, penetapan ukuran transmisi ini berkaitan erat dengan metrik Bandwidth-Delay Product (BDP), yang merepresentasikan batas kapasitas logis data yang dapat ditransmisikan ke dalam jaringan pada satu waktu tertentu tanpa memicu bottleneck atau antrean paket (Grigorik, 2013). Dengan menetapkan ukuran chunk yang diukur secara cermat seperti 256 KB aplikasi klien dapat terus mengirimkan aliran byte array secara stabil, mencegah penumpukan memori lokal, serta mempertahankan integritas data biner pada lingkungan jaringan dengan reliabilitas yang rendah.

Secara teknis, mekanisme chunking pada protokol modern seperti gRPC client-side *streaming* berinteraksi secara sinergis dengan lapisan transport untuk mengoptimalkan throughput. Dengan mengirimkan data secara bertahap, sistem dapat menghindari pemicuan algoritma congestion control yang agresif pada protokol TCP, seperti fase slow start yang sering terganggu oleh transmisi data masif yang tiba-tiba (Kurose & Ross, 2024). Pendekatan ini mengadopsi prinsip yang serupa dengan *Dynamic Adaptive Streaming* over HTTP (DASH) pada pengiriman media, dengan transmisi yang dipecah untuk menjaga stabilitas koneksi jaringan seluler yang fluktuatif (Stockhammer, 2011). Melalui integrasi prinsip flow control pada HTTP/2, mekanisme chunking menjamin integritas data biner tetap terjaga meskipun beroperasi di bawah kondisi reliabilitas jaringan yang rendah.

# BAB III
# METODOLOGI PENELITIAN

## 3.1 Tahapan Penelitian

Untuk mencapai tujuan penelitian secara sistematis, pelaksanaan penelitian disusun mengikuti alur kerja yang tertera pada Gambar 3.1. Tahapan dimulai dari studi literatur hingga evaluasi akhir kinerja sistem.




Gambar 3. 1 Alur Tahapan Penelitian

### 3.1.1 Studi Literatur

Pada tahap ini, dilakukan eksplorasi teknis terhadap pustaka dan dokumentasi resmi untuk memastikan kelayakan implementasi sistem. Fokus studi diarahkan pada tiga pilar teknis utama, yaitu:

1) Mekanisme *Client Streaming* gRPC, Menganalisis dokumentasi protokol gRPC, khususnya metode *client streaming* untuk menangani pengunggahan berkas biner (citra) berukuran besar. Fokus studi mencakup teknik pemecahan data (*chunking*) menjadi paket-paket kecil guna menghindari *timeout* dan memori *overflow* pada perangkat dengan sumber daya terbatas, serta implementasi *Protocol Buffers* untuk definisi kontrak data yang ketat.
2) Arsitektur *Backend* dan *Interoperabilitas* Model AI, mempelajari integrasi ONNX *Runtime* ke dalam layanan *backend* berbasis Go. Kajian difokuskan pada mekanisme memuat model *Deep Learning* (CNN) pra-latih tanpa ketergantungan pada *framework* berat seperti TensorFlow, serta teknik standarisasi input/output *tensor* agar kompatibel dengan data citra yang dikirimkan oleh client.
3) Pengembangan Android Modern, mengkaji penggunaan toolkit UI deklaratif Jetpack Compose dan Android API untuk mengelola dan menggunakan gambar yang tersedia dalam sistem android. Studi juga mencakup manajemen status asinkron menggunakan Kotlin Coroutines dan Flow untuk menangani proses pengiriman data jaringan (gRPC *channel*) tanpa memblokir antarmuka pengguna (UI Thread).

### 3.1.2 Identifikasi Masalah

Berdasarkan kajian literatur, teridentifikasi tiga kendala teknis utama pada sistem diagnosis jarak jauh konvensional yang menjadi fokus penyelesaian dalam penelitian ini:

1) Kegagalan Transmisi pada Jaringan Fluktuatif, pengiriman citra dermatologi resolusi tinggi (umumnya >3 MB) menggunakan protokol HTTP/1.1 standar sering mengalami kegagalan (*timeout*) ketika *bandwidth* jaringan menurun drastis. Metode pengiriman monolitik

(sekali kirim) tidak memiliki mekanisme resume atau *chunking* bawaan yang efisien, memaksa pengguna mengulang proses dari awal.

2) Overhead pada Transmisi Monolitik, Meskipun menggunakan format biner *multipart/form-data* yang efisien secara ukuran *payload*, REST API mengemas seluruh data citra (misalnya 3 MB) ke dalam satu siklus *request* tunggal. Pada jaringan nirkabel yang fluktuatif, pendekatan ini membebani memori client secara instan dan sangat berisiko; jika terjadi *packet loss* di tengah pengiriman, koneksi TCP memutus seluruh proses, memaksa sistem mengulang transmisi dari awal.

3) Risiko Integritas Data, komunikasi berbasis REST API sering kali bersifat *loosely-typed*, sehingga perubahan kecil pada struktur data di sisi server tidak terdeteksi secara otomatis di sisi client (Android). Dalam konteks medis, inkonsisten ini berisiko menyebabkan kegagalan interpretasi hasil diagnosis.

Berdasarkan masalah di atas, sistem dirancang untuk memenuhi kebutuhan teknis berikut:

1) Fungsional, sistem harus mampu memecah citra menjadi segmen kecil (*chunks*) dan mengirimkannya secara bertahap (*streaming*) untuk mencegah *timeout*.

2) Non-Fungsional, sistem wajib menjamin integritas struktur data melalui skema yang ketat (*Strict Contract*) dan memiliki mekanisme pemulihan koneksi (*connection resilience*) otomatis.

### 3.1.3 Perancangan Sistem

Tahap perancangan menerjemahkan kebutuhan sistem menjadi spesifikasi arsitektur teknis yang siap diimplementasikan. Perancangan dibagi menjadi lima komponen utama:

#### 3.1.3.1 Arsitektur Topologi Sistem

Sistem dibangun menggunakan arsitektur Client-Server terpisah (*decoupled*) yang berkomunikasi melalui protokol gRPC. Komponen utama sistem terdiri dari:

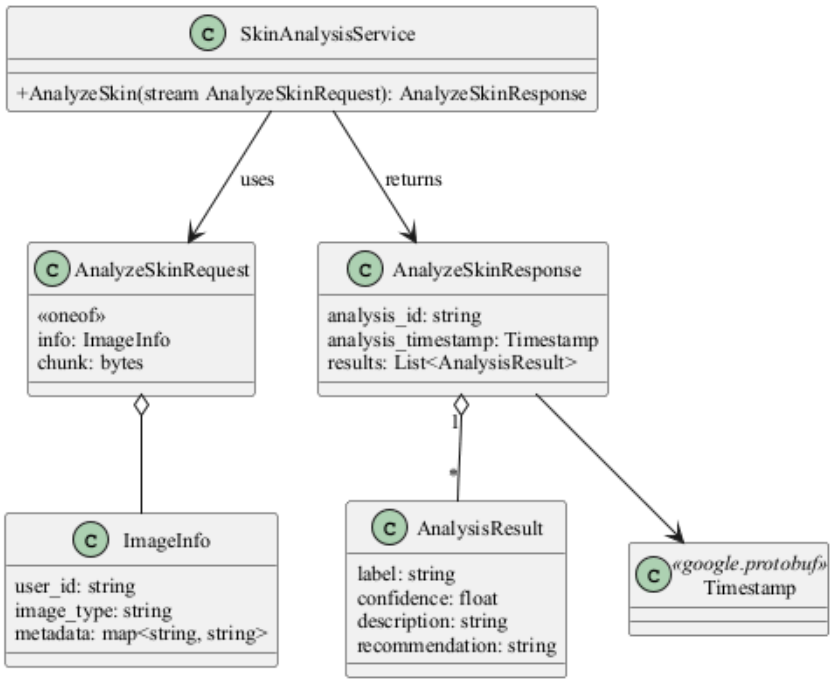
1) Android Client, bertanggung jawab sebagai lapisan presentasi, menangani android API untuk akuisisi citra, dan mengelola kanal gRPC.

2) gRPC Server, bertindak sebagai gerbang komunikasi (*gateway*) yang menangani *request streaming*, validasi data, dan orkestrasi proses inferensi.

3) *Inference Engine* (ONNX *Runtime*), modul terintegrasi di dalam server Golang yang menjalankan model CNN yang telah dikonversi tanpa ketergantungan pada Python/TensorFlow.

### 3.1.3.2 Perancangan Skema API

Inti komunikasi sistem didefinisikan dalam berkas *.proto*, yang berfungsi sebagai kontrak skema antara client dan server. API dirancang menggunakan metode *Remote Procedure Call* (RPC) dengan mekanisme *client streaming* melalui fungsi AnalyzeSkin. Struktur pesan dan definisi layanan secara rinci digambarkan dalam Diagram Kelas pada Gambar 3.2. Diagram ini memperlihatkan penggunaan tipe data *stream* pada AnalyzeSkinRequest yang memungkinkan pengiriman data citra secara bertahap (*chunking*) dari client ke server.




Gambar 3. 2 Class dari Code Contract gRPC

Fungsi AnalyzeSkin digunakan untuk pengiriman citra dari client ke server. Client memecah berkas citra menjadi beberapa segmen (*chunks*) dan mengirimkannya secara berurutan. Server akan menunggu hingga seluruh data citra

diterima sebelum melakukan proses analisis dan mengembalikan satu respons berupa hasil diagnosis.

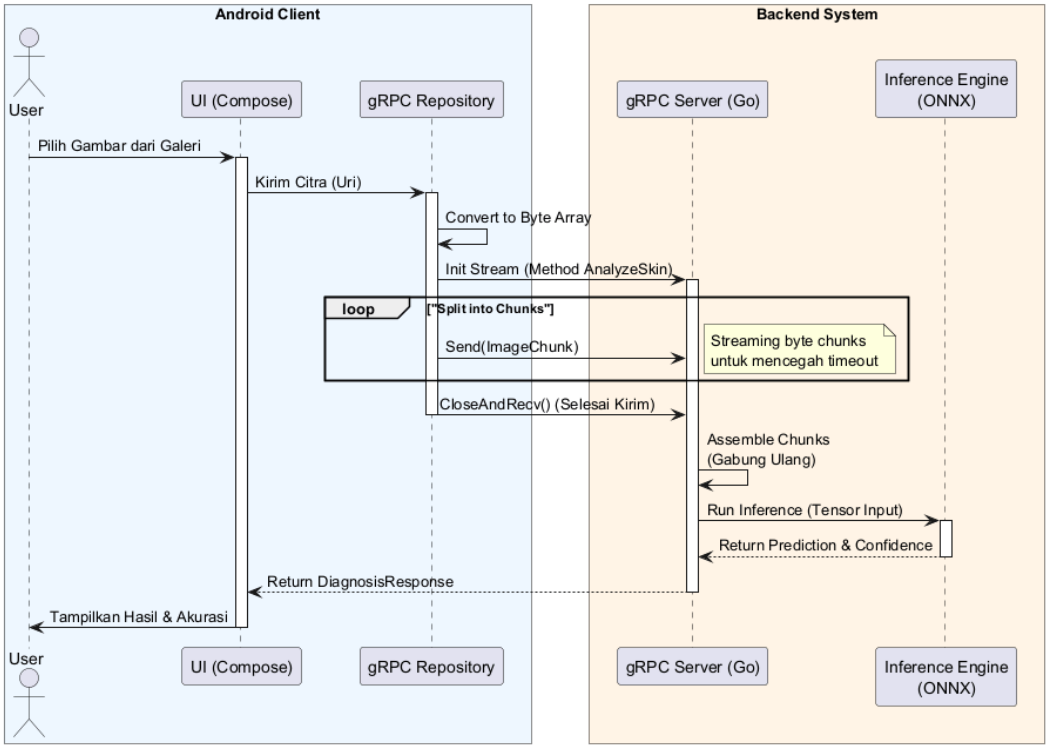
Pendekatan ini memungkinkan aplikasi untuk memiliki kontrol granular terhadap proses pengiriman. Berbeda dengan metode REST monolithic yang menganggap kegagalan sebagian sebagai kegagalan total, mekanisme streaming memungkinkan server memproses data yang masuk secara bertahap. Hal ini mencegah terjadinya request timeout pada tataran HTTP/1.1 yang sering terjadi ketika mengirim satu berkas besar dalam satu waktu di jaringan lambat.

#### 3.1.3.3 Perancangan Alur Data dan Antarmuka

Alih-alih berfokus pada estetika visual, perancangan antarmuka difokuskan pada efisiensi alur data (*Data Flow*):

1) Akuisisi Citra, menggunakan galeri foto android untuk mengambil gambar yang akan didiagnosis.
2) *State Management*, menggunakan ViewModel untuk memantau status pengiriman (Loading, Success, Error) dan memberikan umpan balik visual *real-time* kepada pengguna.
3) Visualisasi Hasil, menampilkan label prediksi (nama penyakit) dan tingkat kepercayaan (probabilitas) yang dikembalikan oleh server.

Interaksi antar komponen sistem selama proses diagnosis diilustrasikan pada Gambar 3.3. Diagram *sequential* tersebut mempertegas mekanisme looping pengiriman *byte array*, yang dirancang untuk mencegah *timeout* pada kondisi jaringan yang tidak stabil.




Gambar 3. 3 *Sequence* Diagram Alur Pengiriman Citra

Meskipun gRPC *stream* bersifat sekuensial, implementasi ini dirancang untuk memberikan umpan balik (*feedback*) kegagalan *per-chunk*. Jika terjadi interupsi jaringan, aplikasi client dapat mendeteksi kegagalan pada *chunk* spesifik tanpa perlu menunggu *timeout* HTTP global, memungkinkan implementasi mekanisme pengiriman ulang (*retry*) yang lebih efisien di masa depan.

#### 3.1.3.4 *Pipeline* Konversi Model AI

Untuk memastikan efisiensi di sisi *backend*, model *Deep Learning* (CNN) tidak dijalankan menggunakan interpreter Python. Perancangan melibatkan *pipeline* konversi model:

1) Sumber, model pra-latih (Format .h5 atau *Saved Model* TensorFlow).
2) Konversi, menggunakan *library* tf2onnx untuk mengubah model menjadi format ONNX (Open Neural Network Exchange).
3) Optimasi, melakukan *graph optimization* (penghapusan node yang tidak perlu) untuk mempercepat waktu inferensi pada CPU server.

Penggunaan ONNX *Runtime* di lingkungan Go secara signifikan mengurangi *footprint* memori server dibandingkan menjalankan container Python/TensorFlow penuh, yang penting untuk efisiensi biaya pada *deployment* cloud.

#### 3.1.3.5 Perancangan Penyimpanan dan Deployment

1) Sisi Client (*Local Caching*), menggunakan Room Database (SQLite) untuk menyimpan riwayat diagnosis secara lokal, memungkinkan pengguna melihat hasil pemeriksaan lampau tanpa koneksi internet.
2) *Containerization* (Docker), seluruh lingkungan server (Golang *Runtime* dan ONNX *Libraries*) dikemas dalam satu Docker Image berbasis Debian. Hal ini menjamin konsistensi lingkungan dan mempermudah proses *deployment* di berbagai infrastruktur cloud.

### 3.1.4 Pengembangan Sistem

Tahap ini merupakan fase realisasi, rancangan arsitektur dan desain sistem diterjemahkan ke dalam kode program yang dapat dieksekusi. Pengembangan dilakukan secara terpisah antara sisi server (*backend*) dan sisi client (mobile) namun tetap mengacu pada skema komunikasi (*interface definition*) yang sama.

#### 3.1.4.1 Pengembangan Sisi Server

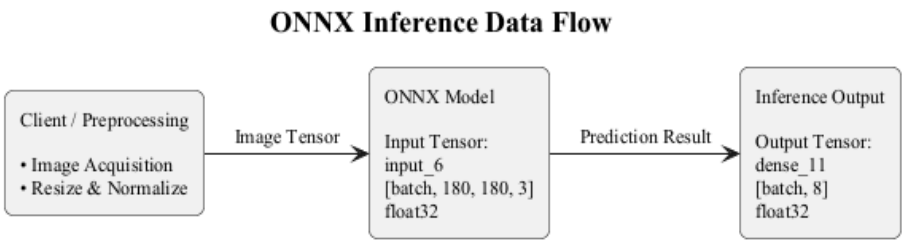
Pengembangan sisi server difokuskan pada pembangunan layanan *backend* yang berkinerja tinggi menggunakan bahasa pemrograman Go. Proses pengembangan server meliputi langkah-langkah berikut:

1) Implementasi Kontrak gRPC, langkah awal adalah mendefinisikan struktur pesan (*message*) dan layanan (*service*) pada berkas *.proto*. Definisi ini mencakup format data citra (dalam bentuk *byte stream*) dan format respons hasil diagnosis.
2) Generasi Kode (*Code Generation*), menggunakan kompilator protoc, berkas *.proto* dikompilasi untuk menghasilkan kode *stub* dan *skeleton* dalam bahasa Go. Kode ini menyediakan antarmuka (*interface*) dasar yang harus diimplementasikan oleh logika server.
3) Implementasi Logika Bisnis dan Layanan:

a. Server *Initialization*, membangun gRPC Server menggunakan *library* gRPC-go yang berjalan pada *port* yang ditentukan.

b. *Request Handler*, mengembangkan fungsi untuk menangani aliran data (*stream*) dari client. Server akan mengumpulkan potongan-potongan data (*chunks*) yang masuk hingga membentuk satu berkas citra utuh.

c. Integrasi Model AI, mengimplementasikan *runtime* ONNX untuk memuat model *Deep Learning* yang telah dilatih. Server melakukan pra-pemrosesan (*resizing* dan normalisasi) pada citra yang diterima sebelum melakukan inferensi. Transformasi dimensi data dari sisi client hingga masuk ke model inferensi ditunjukkan pada Gambar 3.4. Citra yang diterima dikonversi menjadi tensor input dengan dimensi [1, 180, 180, 3] agar sesuai dengan input layer pada model CNN yang digunakan.




Gambar 3. 4 *Input* Dan *Output* Dari Model Yang Digunakan

d. Selain layanan gRPC, dikembangkan pula antarmuka *REST* standar pada *port* terpisah (misalnya *port* 8088) yang memiliki logika bisnis identik. Layanan REST ini berfungsi sebagai tolok ukur (*baseline)* dalam pengujian komparatif untuk memvalidasi peningkatan performa yang ditawarkan gRPC.

4) *Containerization*, seluruh aplikasi server beserta dependensinya (model ONNX dan *library* sistem) dikemas ke dalam Docker Container. Ini memastikan server dapat dijalankan secara konsisten di berbagai lingkungan infrastruktur tanpa kendala kompatibilitas.

### 3.1.4.2 Pengembangan Sisi Client

Pengembangan sisi client dilakukan pada platform Android menggunakan bahasa pemrograman Kotlin dan toolkit UI modern Jetpack Compose. Fokus pengembangan meliputi antarmuka pengguna dan logika komunikasi jaringan:

1. Konfigurasi Project, menambahkan plugin protobuf-gradle-plugin dan dependensi gRPC ke dalam konfigurasi Gradle serta menambahkan library Retrofit dan OkHttp untuk menangani komunikasi berbasis REST API sebagai pembanding. Hal ini memungkinkan aplikasi Android untuk memproses berkas *.proto* yang sama dengan server dan menghasilkan *Client Stub* secara otomatis.
2. Implementasi Antarmuka Pengguna, membangun tampilan aplikasi menggunakan Jetpack Compose yang bersifat deklaratif, terdiri dari:
    a. Halaman Pemilihan Citra, mengimplementasikan integrasi dengan Android Photo Picker (Galeri) untuk memungkinkan pengguna memilih citra kulit yang tersimpan di perangkat.
    b. Halaman Pratinjau (Preview): menampilkan citra yang dipilih sebelum dikirim ke server.
    c. Halaman Hasil Diagnosis, menampilkan nama penyakit terdeteksi dan tingkat akurasi (probabilitas).
3. Logika Komunikasi dan Penyimpanan:
    a. *File Handling*, membaca berkas citra dari URI galeri dan mengonversinya menjadi aliran *byte* (*byte array*).
    b. gRPC *Streaming*, mengimplementasikan logika *chunking* (memecah *byte array* menjadi segmen kecil) dan mengirimkannya melalui *ManagedChannel* menggunakan metode *client streaming*.
    c. Manajemen Data Lokal, mengimplementasikan Room Database untuk menyimpan riwayat diagnosis (jalur berkas citra dan hasil

prediksi) secara lokal, sehingga dapat diakses kembali tanpa koneksi internet.

### 3.1.5 Pengintegrasian dan Testing

Tahap ini memvalidasi interaksi antara sub-sistem Android dan *backend* serta mengukur batas kemampuan sistem. Pengujian dibagi menjadi tiga skenario spesifik:

#### 3.1.5.1 Pengujian Fungsional

Pengujian ini memverifikasi logika bisnis dan alur data menggunakan metode *Black Box Testing*. Fokus pengujian meliputi:

1) Validasi Kontrak, memastikan data yang dikirim oleh client sesuai dengan skema *.proto* dan dapat di-*decode* dengan benar oleh server.
2) Skenario *Edge Case*, menguji respons sistem terhadap input non-standar, seperti citra format yang tidak didukung, ukuran berkas melebihi batas, atau gangguan koneksi tiba-tiba (*network interrupt*).

#### 3.1.5.2 Pengujian Kinerja dan Pemantauan

Pengujian kinerja dilakukan menggunakan Locust untuk mensimulasikan beban pengguna serentak (*concurrency*) pada server.

1) Alat Pengujian, *script* pengujian ditulis menggunakan Locust Python untuk melakukan request gRPC dan *REST* secara paralel.
2) Kondisi Jaringan, pengujian kinerja menyimulasikan lima profil jaringan (Normal, 4G, 3G, Poor, dan Worst) menggunakan utilitas Traffic Control (tc) pada ekosistem Linux NetEm untuk merepresentasikan kondisi operasional seluler yang riil. Parameter bandwidth dan latensi pada profil 3G dan 4G diadopsi secara spesifik dari standar rata-rata arsitektur telekomunikasi seluler (3rd Generation Partnership Project, 1999; OpenSignal, 2020). Sementara itu, profil Poor dan Worst dirancang sebagai skenario degradasi menggunakan batas maksimal latensi 300 ms sesuai rekomendasi (International Telecommunication Union, 2003) serta tingkat packet loss 1-3% guna menguji ketahanan protokol terhadap reaksi pengendalian kemacetan protokol TCP (Kurose & Ross, 2024).

3) Skenario Uji:

   a. *Stress Test,* meningkatkan jumlah *Virtual Users* (VUs) secara bertahap hingga titik jenuh server untuk mengetahui batas maksimum *Throughput* (RPS).

   b. *Load Test,* memberikan beban trafik stabil untuk mengukur kinerja sistem pada kondisi penggunaan normal.

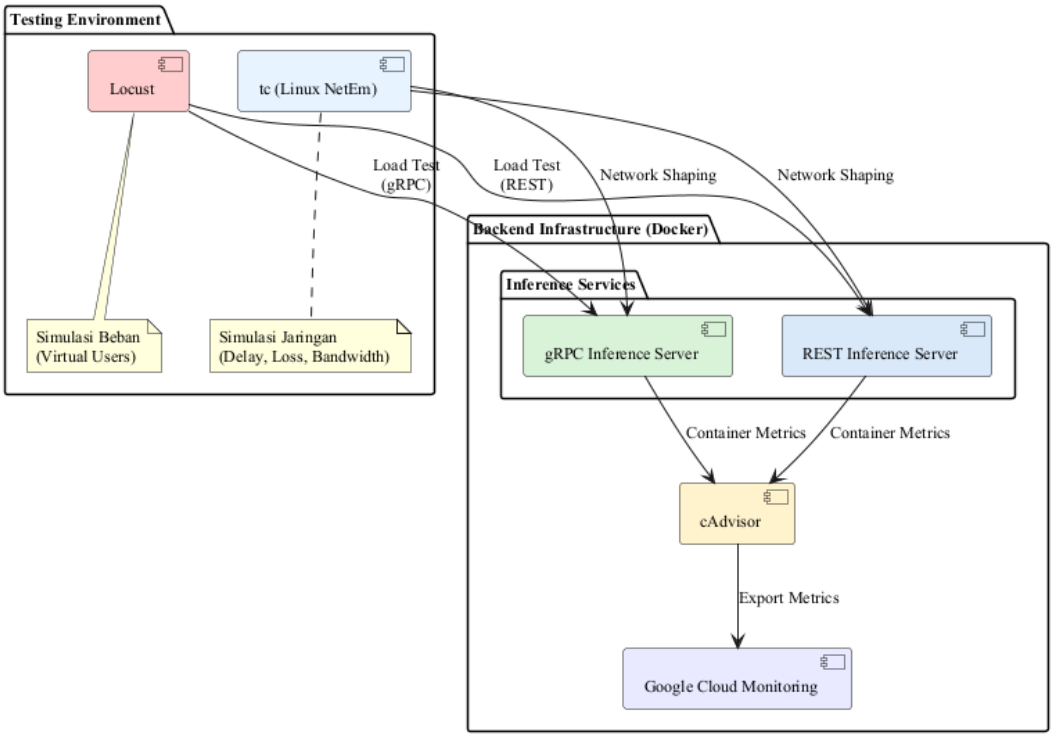
   c. *Spike Test*, memberikan lonjakan trafik tiba-tiba untuk menguji kecepatan pemulihan sistem (*recovery time*).

   d. *Soak Test,* memberikan beban berkelanjutan dalam jangka waktu panjang untuk menguji stabilitas sistem.

4) Proses Uji, dilakukan dengan memanipulasi kondisi jaringan pada setiap skenario pengujian menggunakan tc, serta dilaksanakan sebanyak (jumlah kondisi jaringan dikalikan jumlah skenario pengujian). Kemudian hasilnya akan disimpan sebagai csv untuk dianalisis selanjutnya.

5) Pemantauan Metrik, selama pelaksanaan pengujian menggunakan Locust, metrik kinerja server seperti penggunaan CPU dan memori secara real-time menggunakan cAdvisor. Data metrik tersebut kemudian divisualisasikan melalui layanan pemantauan Google Cloud Platform.

Konfigurasi lingkungan pengujian kinerja secara keseluruhan ditunjukkan pada Gambar 3.5, yang memperlihatkan integrasi Locust sebagai generator beban, simulasi kondisi jaringan menggunakan utilitas tc pada sistem Linux, serta cAdvisor dan Google Cloud Monitoring sebagai sistem pemantauan kinerja.




Gambar 3. 5 Arsitektur Dari Performa Testing

Selain pemantauan di sisi server, pengujian kinerja juga dilakukan di sisi perangkat client. Pengukuran ini menggunakan instrumen Android Studio Profiler dan Perfetto yang dihubungkan ke perangkat uji melalui Android Debug Bridge (ADB). Metrik yang direkam mencakup total waktu komputasi CPU untuk menyelesaikan satu siklus transmisi, serta puncak alokasi Heap Memory (RAM) selama fungsi *streaming* data dieksekusi.

### 3.1.5.3 Perbandingan gRPC dan REST

Evaluasi akhir dilakukan dengan membandingkan protokol gRPC (solusi yang diusulkan) terhadap REST API (pendekatan tradisional) dalam kondisi terkontrol:

1) Metrik Utama (Kinerja & Reliabilitas)

   a. *Success Rate (Tingkat Keberhasilan)*, persentase request yang berhasil terkirim sempurna (HTTP 200 / gRPC OK) pada berbagai profil kondisi jaringan seluler yang disimulasikan, yaitu kondisi Normal, 4G, 3G, Poor, dan Worst. Ini adalah indikator utama stabilitas sistem.

b. Kapasitas *Throughput*, Mengukur tingkat efisiensi pemrosesan data oleh server backend dalam menangani konkurensi (beban Virtual Users yang disimulasikan menggunakan Locust). Throughput diukur dalam satuan Requests Per Second (RPS) dengan persamaan (1):

$$\boldsymbol{Th = \frac{\sum N_{sukses}}{T_{uji}}} \qquad \textbf{(1)}$$

Keterangan:

$\boldsymbol{Th}$ : Kapasitas *throughput* sistem (RPS)

$\boldsymbol{\sum N_{sukses}}$ : Total jumlah transmisi citra medis yang berhasil diproses dan diklasifikasikan oleh server.

$\boldsymbol{T_{uji}}$ : Total durasi waktu pengujian pada suatu skenario spesifik (detik).

c. *Completion Time* (Waktu Penyelesaian / Latensi), Total waktu yang dibutuhkan dari saat aplikasi Android menginisiasi pengiriman hingga menerima balikan hasil diagnosis secara utuh dari mesin inferensi ONNX. Formulasi latensi (2) pada sistem ini didefinisikan sebagai:

$$\boldsymbol{L = t_{response} - t_{inisiasi}} \qquad \textbf{(2)}$$

Keterangan:

$\boldsymbol{L}$ : Waktu penyelesaian atau latensi transmisi (ms)

$\boldsymbol{t_{response}}$ : Waktu (timestamp) saat client menerima objek AnalyzeSkinResponse secara utuh.

$\boldsymbol{t_{inisiasi}}$ : Waktu (timestamp) saat client mulai mengirimkan byte array pertama (REST) atau memulai *stream* metadata ImageInfo (gRPC).

2) Metrik Sekunder (Efisiensi Sumber Daya)

a. *Payload Size*, mengukur efisiensi ukuran data biner (Protobuf) dibandingkan biner menggunakan metode *multipart/form*.

b. *Resource Utilization*, mengukur penggunaan memori (RAM) pada perangkat Android selama proses transmisi untuk memvalidasi efisiensi *streaming* dibandingkan memuat seluruh citra ke memori.

### 3.1.6 Variabel Penelitian dan Justifikasi Parameter

Penelitian ini menggunakan desain eksperimen terkontrol dengan penentuan variabel sebagai berikut:

1) Variabel Bebas, Protokol komunikasi data (gRPC *Client-side Streaming* vs REST API *Multipart*) dan Profil Jaringan Seluler (Normal, 4G, 3G, Poor, Worst) yang disimulasikan melalui Linux NetEm.

2) Variabel Terikat, Metrik pada sisi Client meliputi *Success Rate* (%), *Completion Time* (ms), Utilisasi Memori (MB), dan Waktu Komputasi CPU (ms). Metrik pada sisi Server meliputi Kapasitas *Throughput* (RPS), Waktu Pemulihan / *Recovery Time* (detik), dan Stabilitas Alokasi Memori Server (MB).

*3)* Justifikasi Parameter *Chunk Size,* Pengiriman *streaming* gRPC pada sisi client diimplementasikan menggunakan ukuran *chunk* tetap (*fixed chunk size*) sebesar 256 KB. Pemilihan ukuran *chunk* sebesar 256 KB digunakan sebagai nilai *baseline* tetap untuk menjaga konsistensi dan kontrol variabel selama pengujian, sehingga perbandingan kinerja antar skenario tidak dipengaruhi oleh variasi parameter transmisi. Nilai ini tidak ditentukan melalui proses optimasi, melainkan berdasarkan pertimbangan konseptual *Bandwidth-Delay Product* (BDP) untuk menyeimbangkan efisiensi penggunaan bandwidth dan overhead komunikasi. Dengan demikian, eksplorasi variasi ukuran *chunk* berada di luar cakupan penelitian ini dan diusulkan sebagai arah penelitian lanjutan.

### 3.1.7 Ancaman Validitas

A. Validitas Internal

Penggunaan *Traffic Control* (tc) untuk menyimulasikan gangguan jaringan dilakukan secara deterministik, sehingga memberikan kontrol eksperimen yang baik dan hasil yang reprodusibel. Namun, pendekatan ini belum sepenuhnya merepresentasikan kondisi jaringan nyata yang bersifat dinamis, seperti interferensi sinyal, perpindahan antar BTS, dan variasi propagasi sinyal.

Pengujian beban dilakukan menggunakan Locust dengan *library* gRPCio native Python, yang memungkinkan transmisi biner protobuf secara langsung serta mendukung *shared channel* HTTP/2 untuk *concurrent stream*. Meskipun mendekati arsitektur gRPC sebenarnya, terdapat potensi *instrumentation bias* karena Locust menggunakan *cooperative threading* (gevent), yang berbeda dengan mekanisme *Kotlin Coroutines* pada Android. Oleh karena itu, metrik yang dihasilkan lebih merepresentasikan beban sintetis terkontrol dibandingkan performa absolut pada lingkungan nyata.

B. Validitas External

*Profiling* memori dan CPU dalam penelitian ini dilakukan pada satu perangkat Android fisik dengan konfigurasi spesifik, sehingga pengukuran bersifat konsisten namun sangat dipengaruhi oleh karakteristik perangkat tersebut. Variasi versi Android, kernel, serta arsitektur dan generasi chipset berpotensi menghasilkan perbedaan metrik, sehingga hasil yang diperoleh perlu dipahami sebagai representasi pada perangkat uji dan tidak dapat digeneralisasi secara langsung ke seluruh ekosistem Android.

## 3.2 Waktu Penelitian

Penelitian ini dilaksanakan dalam kurun waktu kurang lebih empat bulan, dimulai pada tanggal 8 Januari hingga 30 April. Pelaksanaan penelitian mencakup

tahap penetapan tema, pengumpulan data, pengolahan dan analisis data, hingga penyusunan laporan akhir.

### 3.3 Alat dan Bahan

Bagian ini merinci perangkat keras, perangkat lunak, dan bahan lain yang akan digunakan dalam pelaksanaan penelitian ini. Pemilihan alat dan bahan didasarkan pada kebutuhan pengembangan aplikasi, pengujian kinerja gRPC, dan analisis data.

#### 3.3.1 Perangkat Keras

Spesifikasi perangkat keras yang digunakan untuk pengembangan dan pengujian server disajikan pada Tabel 3.1.

Tabel 3. 1 Perangkat Keras

| **Device** | **Spesifikasi** |
|---|---|
| Compute Engine Server | vCPU: 5c; RAM: 5G;<br>OS: Alpine Linux; TYPE: Container |
| Compute Engine Tester | vCPU: 2c; RAM: 4G<br>OS: Ubuntu Minimal; TYPE: VM |
| Redmi Note 14 Pro 5G | CPU: 8c; RAM: 8G; OS: Android (16); TYPE: Mobile Device; WIFI: 50Mbps. |

#### 3.3.2 Perangkat Lunak

Pada penelitian ini, perangkat lunak digunakan untuk mendukung proses perancangan, pengembangan, pengujian, dan analisis hasil penelitian. Rincian perangkat lunak beserta versinya dirangkum dalam Tabel 3.2.

Tabel 3. 2 Perangkat Lunak

| **Perangkat Lunak** | **Kategori** | **Versi** |
|---|---|---|
| Android Studio | IDE | Otter 2 |
| Goland | IDE | 2025 |

| | | |
|---|---|---|
| Kotlin | Programming Language | 2.2.21 |
| Go-lang | Programming Language | 1.25 |
| Protobuf | Data Serialization Format Tool | 25.7 |
| Postman | API Development Tool | 11.68.5 |
| Google Cloud Monitor | Analytics And Visualization Tool | - |
| cAdvisor | Monitoring And Alerting Tool | 0.55.1 (gcr.io) |
| Locust | Performance Testing Tool | 2.24.0 |
| Perfetto | Trace Visualization Tool | 0.16.0 |
| LeakCanary | Memory Leak Detection Library | 2.14 |

# BAB IV
# HASIL DAN PEMBAHASAN

## 4.1 Hasil Implementasi Sistem

Implementasi sistem menghasilkan aplikasi Android dan layanan backend yang dapat berkomunikasi menggunakan mekanisme *client-side streaming* gRPC. Sistem ini mampu mengirim citra medis dalam bentuk potongan data (*chunk*) dari perangkat Android ke server serta menerima hasil inferensi model.

Hasil implementasi menunjukkan bahwa mekanisme *client streaming* berjalan sesuai dengan skema yang didefinisikan dan dapat digunakan sebagai dasar untuk pengujian kinerja serta perbandingan dengan *REST API* pada tahap selanjutnya.

### 4.1.1 Hasil Rancangan Skema gRPC

Skema gRPC pada sistem ini direalisasikan dalam berkas *skin_analysis.proto* yang digunakan oleh sisi server dan client. Kontrak tersebut mendefinisikan satu layanan utama dengan metode *client-side streaming* untuk pengiriman data citra serta satu respons berupa hasil analisis. Gambar 4.1 menampilkan skema *Service* yang akan digunakan oleh protokol gRPC yang diimplementasikan pada sistem.

```
syntax = "proto3";
package skin_analyzer;
option go_package = "skin-analyzer-service/gen;citra";
import "google/protobuf/timestamp.proto";

service SkinAnalysisService {
rpc AnalyzeSkin (stream AnalyzeSkinRequest) returns (AnalyzeSkinResponse);
}
message ImageInfo {
  string user_id = 1;
  string image_type = 2;
  bytes client_sha256 = 3;
  map<string, string> metadata = 4;
}
message AnalyzeSkinRequest {
```

```
  oneof request_payload {
    ImageInfo info = 1;
    bytes chunk = 2;
  }
}
message AnalysisResult {
  string label = 1;
  float confidence = 2;
  string description = 3;
  string recommendation = 4;
}
message AnalyzeSkinResponse {
  string analysis_id = 1;
  google.protobuf.Timestamp analysis_timestamp = 2;
  bytes server_sha256 = 4;
  repeated AnalysisResult results = 5;
}
```

Gambar 4. 1 Skema dari Protobuf

Berdasarkan Kode Program 4.1, skema *Service* gRPC pada sistem ini terdiri atas layanan *client-side streaming* untuk pengiriman data citra, struktur pesan permintaan yang memuat *metadata* dan potongan data citra, serta struktur respons yang mengembalikan hasil analisis dan informasi validasi data. Berkas *.proto* tersebut dikompilasi menggunakan protoc untuk menghasilkan kode *stub* yang digunakan pada implementasi server dan client.

### 4.1.2 Implementasi Sisi Server

Implementasi sisi server dilakukan pada fungsi *AnalyzeSkin* yang menangani permintaan *client-side streaming* gRPC. Metode ini menerima aliran data dari client, menggabungkan potongan data citra yang diterima, serta memproses citra menggunakan model inferensi sebelum mengirimkan respons ke client. Kode Program 4.2 menunjukkan implementasi fungsi *AnalyzeSkin* pada sisi server.

```
func (s *SkinAnalysisServer) AnalyzeSkin(
        stream citra.SkinAnalysisService_AnalyzeSkinServer,
) error {
        var buffer bytes.Buffer
        var info *citra.ImageInfo
```

```
        for {
                req, err := stream.Recv()
                if err == io.EOF {
                        break
                }
                switch payload := req.RequestPayload.(type) {
                case *citra.AnalyzeSkinRequest_Info:
                        info = payload.Info
                case *citra.AnalyzeSkinRequest_Chunk:
                        buffer.Write(payload.Chunk)
                }
        }
        imageBytes := buffer.Bytes()
        serverChecksum := sha256.Sum256(imageBytes)
        if !bytes.Equal(serverChecksum[:], info.ClientSha256) {
                return status.Error(codes.DataLoss, "checksum mismatch")
        }
        input, _ := service.Preprocessing(&imageBytes)
        predictions, _ :=
                s.inferenceService.GetTopKPredictions(input, 1)
        var results []*citra.AnalysisResult
        for _, p := range predictions {
                results = append(results, &citra.AnalysisResult{
                        Label:      p.ClassName,
                        Confidence: p.Confidence,
                })
        }
        response := &citra.AnalyzeSkinResponse{
                AnalysisId:        uuid.New().String(),
                AnalysisTimestamp: timestamppb.New(time.Now()),
                ServerSha256:      serverChecksum[:],
                Results:           results,
        }
        return stream.SendAndClose(response)
}
```

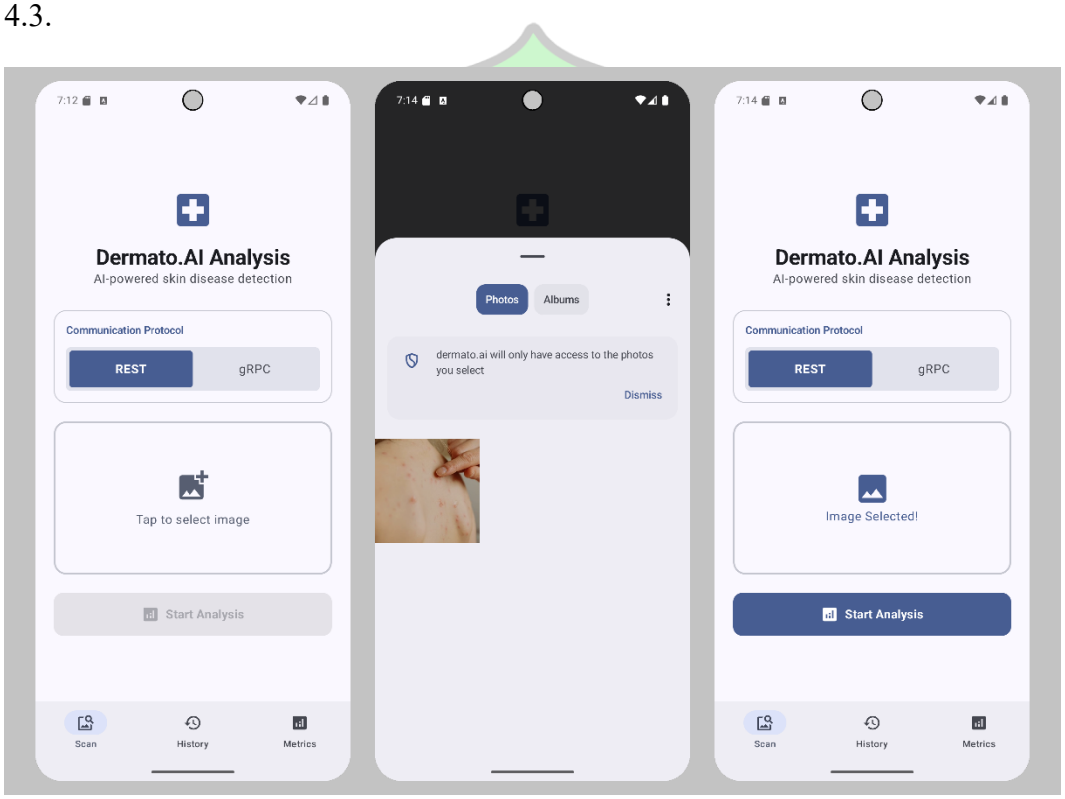
Gambar 4. 2 Implementasi Logika Klasifikasi Di Sisi Server

Pada implementasi ini, server menerima paket *metadata* dan data citra secara berurutan, merekonstruksi data citra menjadi satu kesatuan, melakukan validasi integritas menggunakan *checksum* SHA-256, serta menghasilkan respons yang

memuat hasil analisis dan informasi pendukung lainnya. Respons kemudian dikirimkan ke client setelah seluruh data diterima.

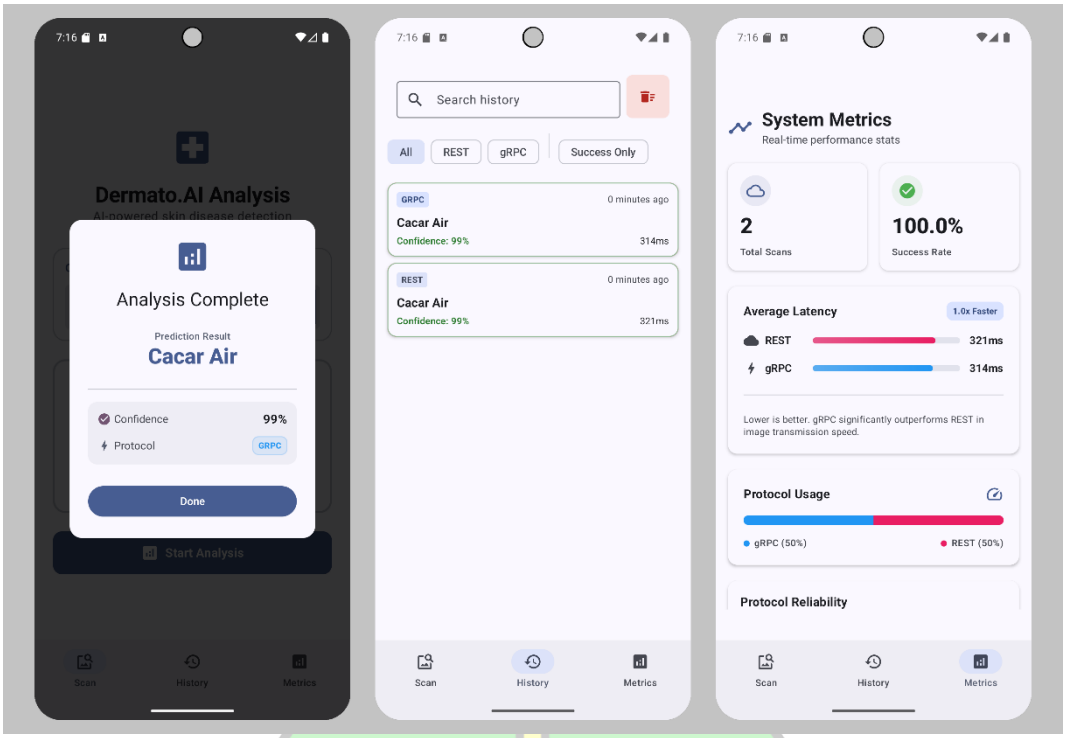
### 4.1.3 Implementasi Sisi *Client*

Sisi client dikembangkan menggunakan bahasa pemrograman Kotlin dengan kerangka kerja Jetpack Compose. Implementasi difokuskan pada penyediaan antarmuka pengguna serta integrasi mekanisme komunikasi dengan server melalui REST dan gRPC. Antarmuka aplikasi menyediakan fitur pemilihan citra dari penyimpanan lokal, pemilihan protokol komunikasi, serta penampilan hasil analisis. Implementasi halaman utama dan halaman hasil ditunjukkan pada Gambar 4.3.




Gambar 4. 3 Tampak Halaman Utama

Pada halaman utama Gambar 4.3, pengguna dapat memilih citra menggunakan Android Photo Picker API dan menentukan protokol komunikasi yang digunakan. Setelah proses analisis dijalankan, aplikasi menampilkan hasil diagnosis beserta tingkat kepercayaan dan informasi protokol yang digunakan, sebagaimana ditunjukkan pada Gambar 4.4.




Gambar 4. 4 Halaman Hasil dari Klasifikasi

### 4.1.4 Implementasi Mekanisme *Streaming*

Pengiriman data citra pada sisi client diimplementasikan menggunakan mekanisme *client-side streaming* gRPC. Logika ini berada pada lapisan *SkinAnalysisRepository*, sebagaimana ditunjukkan pada gambar 4.5.

```
private suspend fun fetchGRPCRaw(imageBytes: ByteArray): AnalyzeSkinResponse {
        val imageChecksum = imageBytes.sha256()
        val requestFlow = flow {
            emit(
                AnalyzeSkinRequest.newBuilder()
                    .setInfo(
                        ImageInfo.newBuilder()
                            .setImageType("jpeg")
                            .setUserId("android-user")
                            .setClientSha256(
                                ByteString
                                    .copyFrom(imageChecksum)
                            )
                            .build()
                    )
```

```
                    .build()
            )
            val chunkSize = 256 * 1024
            var offset = 0
            while (offset < imageBytes.size) {
                val length = min(chunkSize, imageBytes.size - offset)
                emit(
                    AnalyzeSkinRequest.newBuilder()
                        .setChunk(
                            ByteString
                                .copyFrom(imageBytes, offset, length)
                        )
                        .build()
                )
                offset += length
            }
        }
        return analyseStub
            .withDeadlineAfter(30, TimeUnit.SECONDS)
            .analyzeSkin(requestFlow)
    }
```

Gambar 4. 5 Fungsi Pengiriman Data Citra gRPC pada Sisi Client

Ukuran setiap potongan data citra ditetapkan sebesar 256 KB. Proses pengiriman dilakukan hingga seluruh data citra terkirim ke server. Setelah aliran data selesai, client menunggu respons hasil analisis dari server melalui fungsi analyzeSkin.

Untuk membuktikan bahwa mekanisme *client-side streaming* berjalan sesuai dengan rancangan kontrak layanan, dilakukan pemantauan terhadap *payload* data yang ditransmisikan antara client dan server. Pemantauan ini memverifikasi bahwa citra tidak dikirim secara monolitik, melainkan dipecah menjadi *stream* data. Dapat dilihat pada gambar 4.6.

```
---------------------------- PROCESS STARTED (6617) for package
com.github.dermatoai ----------------------------
2026-03-04 08:24:52.459  6617-6712  NetworkAna...Repository com.github.dermatoai
D  Sending request
2026-03-04 08:24:52.571  6617-6712  NetworkAna...Repository com.github.dermatoai
D  Sent metadata
```

```
2026-03-04 08:24:52.577  6617-6712  NetworkAna...Repository com.github.dermatoai
D  Sent chunk 0

2026-03-04 08:24:52.642  6617-6712  NetworkAna...Repository com.github.dermatoai
D  Sent chunk 65536

2026-03-04 08:24:53.169  6617-6714  NetworkAna...Repository com.github.dermatoai
D  Sent chunk 1114112

2026-03-04 08:24:53.593  6617-6714  NetworkAna...Repository com.github.dermatoai
D  Sent chunk 2162688

2026-03-04 08:24:53.820  6617-6714  NetworkAna...Repository com.github.dermatoai
D  Sent chunk 2228224

2026-03-04 08:24:53.828  6617-6714  NetworkAna...Repository com.github.dermatoai
D  Sent chunk 2293760

2026-03-04 08:24:53.829  6617-6714  NetworkAna...Repository com.github.dermatoai
D  Sent all chunks

2026-03-04 08:24:55.754  6617-6714  NetworkAna...Repository com.github.dermatoai
D  Received response

...
```

Gambar 4. 6 *Log* dari Proses Analisis Citra

Gambar 4.6 memperlihatkan log transmisi request yang diinisiasi oleh client Android. Terlihat bahwa paket data pertama yang dikirimkan adalah *metadata ImageInfo* yang memuat *image_type* dan client_sha256 untuk keperluan validasi integritas data. Setelah metadata diterima oleh Server, *Client* secara *iteratif* mengirimkan *payload chunk* berupa *byte array* berukuran 256 KB secara berurutan hingga seluruh ukuran citra habis tertransmisikan. Mekanisme ini terlihat pada log yang mencetak status *send chunk* secara *sekuensial* pada satu koneksi aktif yang sama.

### 4.1.5 Indikator Keberhasilan dan Kegagalan Sistem

Untuk menghitung tingkat keberhasilan pengiriman (*Success Rate*) pada pengujian kinerja di sub-bab selanjutnya, sistem harus memiliki parameter yang jelas mengenai keadaan sebuah transmisi dinyatakan Sukses dan keadaaan dinyatakan Gagal (Error). Indikator ini dilacak secara real-time melalui log penanganan eksepsi pada lapisan *SkinAnalysisRepository* di sisi *clien*.

A. Indikator Sistem Sukses

Sebuah transmisi gRPC dinyatakan sukses apabila client berhasil memecah dan mengirimkan seluruh potongan *byte array* (*chunk*) ke server, lalu menerima balikan (*Unary Response*) berupa

objek *AnalyzeSkinResponse* tanpa adanya interupsi jaringan. Bukti log dari skenario sukses ini dapat dilihat pada Gambar 4.7.

```
Response: # com.github.dermatoai.AnalyzeSkinResponse@788aa6bb
analysis_id: "58321673-43f6-419c-99ee-826e17c1fab1"
analysis_timestamp {
  nanos: 544656268
  seconds: 1772587498
}
results {
  confidence: 0.93327117
  description: "Cacar air adalah infeksi virus varicella-zoster yang
menyebabkan ruam lepuh berisi cairan disertai demam dan rasa gatal."
  label: "Cacar Air"
  recommendation: "Istirahat cukup, hindari menggaruk ruam, gunakan
obat pereda gatal, dan konsultasikan ke dokter bila gejala berat."
}
server_sha256:
"\306{\001\311iOc\230\315\3636\r\256l\257$\365\345\2570-
\342\226f\njj\355/c\307\233"
```

Gambar 4. 7 Respons *Success*

B. Indikator Sistem Gagal

Skenario kegagalan terjadi apabila proses *streaming* terputus di tengah jalan akibat hilangnya sinyal seluler (*packet loss* 100%) atau waktu tunggu yang habis melebihi batas (melebihi deadline/timeout 30 detik) sampel dapat dilihat pada Gambar 4.8, 4.9. Pada kondisi ini, *library* gRPC akan melempar *exception* yang ditangkap oleh blok *catch* pada Android.

```
DEADLINE_EXCEEDED: CallOptions deadline exceeded after 29.681778300s.
Name resolution delay 0.041011200 seconds. [closed=[],
committed=[buffered_nanos=157514300, remote_addr=/10.0.2.2:8008]]
```

Gambar 4. 8 Respons *Timeout*

```
HTTP 0:
```

Gambar 4. 9 Tidak Bisa Terhubung Server REST

### 4.2 Hasil Pengujian Fungsional

Pengujian fungsional dilakukan menggunakan metode *Black Box Testing* untuk memverifikasi kesesuaian antara input yang diberikan dengan output yang dihasilkan oleh sistem. Pengujian ini berfokus pada evaluasi alur kerja utama aplikasi, validasi kontrak layanan data (*.proto*), serta ketahanan sistem dalam menangani skenario batas (*Edge Cases*) seperti interupsi jaringan atau format berkas yang tidak valid. Hasil keseluruhan dari skenario pengujian fungsional dirangkum pada Tabel 4.1.

Tabel 4. 1 Hasil Pengujian Fungsional Sistem Berbasis *Black Box Testing*

| No. | Skenario Pengujian | Deskripsi Singkat | Hasil Yang Diharapkan | Hasil Pengujian | Status |
|---|---|---|---|---|---|
| 1 | Validasi Kontrak Normal | Mengunggah citra kulit dengan format standar (JPG/PNG) berukuran di bawah 1MB. | Client berhasil memecah citra, server merakit ulang (decode) dan mengembalikan respons hasil diagnosis dengan benar. | Sesuai Harapan | selesai |
| 2 | Pengujian *Chunking* (Berkas Besar) | Mengunggah citra kulit beresolusi tinggi dengan ukuran di atas 5 MB. | Sistem memecah citra menjadi banyak chunks dan mengirimkannya secara *streaming* tanpa timeout hingga selesai. | Sesuai Harapan | selesai |

| | | | | | |
|---|---|---|---|---|---|
| 3 | Skenario *Edge Case*: Format Tidak Valid | Pengguna mencoba mengunggah berkas selain citra (misal: PDF atau teks) dari perangkat Android. | Aplikasi client memvalidasi input atau server menolak paket metadata pertama, menampilkan pesan error "Format tidak didukung" | Sesuai Harapan | selesai |
| 4 | Skenario *Edge Case*: Interupsi Jaringan | Memutuskan koneksi internet (mengaktifkan Airplane Mode) di tengah proses upload citra yang sedang berlangsung. | Proses *streaming* gRPC terputus, aplikasi tidak mengalami crash, dan memunculkan notifikasi "Koneksi terputus/Gagal mengirim". | Sesuai Harapan | selesai |

Berdasarkan Tabel 4.1, seluruh fungsionalitas utama sistem telah berjalan sesuai dengan spesifikasi kebutuhan. Pada skenario pengujian normal maupun transmisi berkas besar, mekanisme *client-side streaming* terbukti mampu mengelola aliran data *byte array* secara berurutan.

Selain itu, pada pengujian *edge case* (gangguan jaringan tiba-tiba), sistem menunjukkan penanganan kesalahan (*error handling*) yang baik. Aplikasi Android client tidak mengalami *Force Close* (*crash*) saat *stream* gRPC terputus di tengah jalan, melainkan berhasil menangkap e*xception* dan meneruskannya menjadi umpan balik visual (UI *State* Error*)* kepada pengguna. Hal ini memastikan stabilitas

aplikasi tetap terjaga meskipun beroperasi pada kondisi infrastruktur yang tidak ideal.

### 4.3 Hasil Pengujian Kinerja

Pengujian kinerja bertujuan untuk mengevaluasi kemampuan, stabilitas, dan konsumsi sumber daya sistem di bawah beban kerja yang disimulasikan menggunakan Locust. Untuk memperoleh representasi kinerja yang komprehensif pada aplikasi *mobile health*, pengujian dirancang dalam bentuk matriks skenario yang mengombinasikan profil beban trafik dengan berbagai kondisi jaringan seluler.

#### 4.3.1 Matriks Skenario Pengujian

Pengujian kinerja sistem dilakukan dengan mengombinasikan lima profil simulasi jaringan dan lima jenis pengujian beban untuk merepresentasikan berbagai kondisi penggunaan. Profil simulasi jaringan yang digunakan beserta parameternya ditampilkan pada Tabel 4.2, sedangkan skenario pengujian beban yang diterapkan dirangkum pada Tabel 4.3.

Tabel 4. 2 Profile Simulasi Jaringan

| ***Mode*** | ***Delay*** | ***Packet Loss*** | ***Bandwidth*** |
|---|---|---|---|
| ***Normal*** | *0 ms* | 0% | Tidak Terbatas |
| ***Poor*** | *100 ms* | 1% | Tidak Terbatas |
| ***Worst*** | *300 ms* | 3% | Tidak Terbatas |
| ***3G*** | *200 ms* | 2% | 384 *Kbps* |
| ***4G*** | *80 ms* | 0.5% | 10 *Mbps* |

Tabel 4. 3 Skenario Beban Test

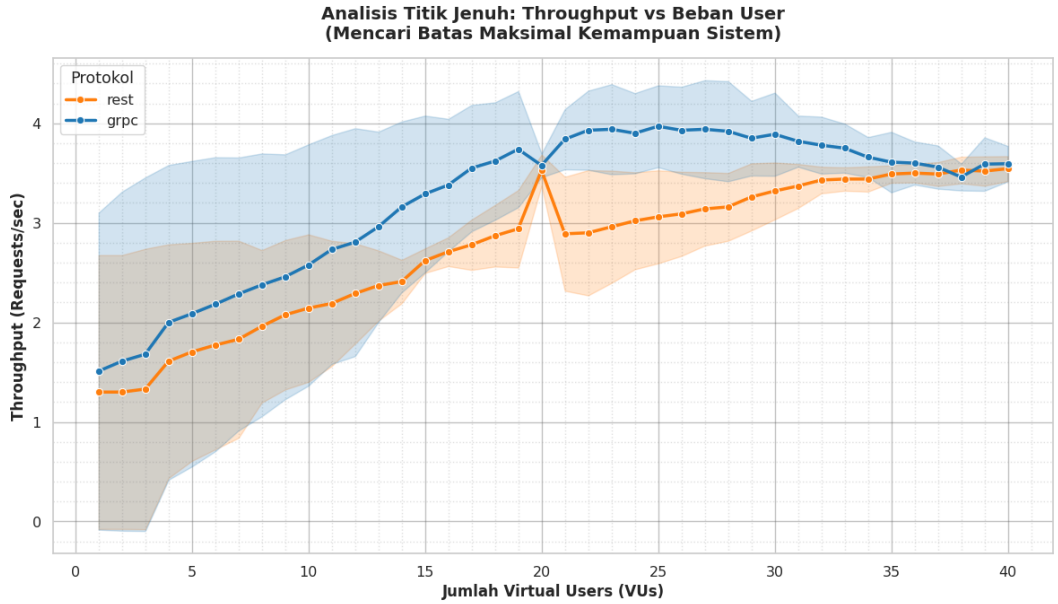
| **Jenis Pengujian** | **Tujuan Utama** | **Durasi/*VUs*** | **Parameter** |
|---|---|---|---|

| | | | |
|---|---|---|---|
| ***Smoke Test*** | Verifikasi awal (*Sanity Check*) sistem di lingkungan *production* | Beban konstan 1 VUs selama 1 menit. , diikuti fase *cooldown* 10 detik. | Normal, 4G, 3G, Poor, Worst |
| ***Load Test*** | Mengukur kinerja pada beban trafik normal/ekspektasi harian | Ramping hingga 10 VUs (2 menit), dipertahankan selama 5 menit, kemudian diturunkan selama 2 menit. | Normal, 4G, 3G, Poor, Worst |
| ***Stress Test*** | Menemukan titik jenuh (*bottleneck*) dan kapasitas maksimal (RPS) | Bertahap bertingkat (*stepped*) mencapai 20 VUs, lalu ditekan hingga puncak 40 VUs dengan total durasi 16 menit. | Normal, 4G, 3G, Poor, Worst |
| ***Spike Test*** | Mengevaluasi latensi dan *recovery time* saat lonjakan mendadak | Beban meningkat dari 10 VUs ke 50 VUs dalam 30 detik, dipertahankan selama 3 menit, kemudian diturunkan untuk observasi *recovery*. | Normal, 4G, 3G, Poor, Worst |
| ***Soak Test*** | Menguji stabilitas memori (indikasi memory leak) jangka panjang | Beban konstan 15 VUs selama 30 menit (dibatasi oleh sumber daya pengujian). | Normal, 4G, 3G, Poor, Worst |

Selain itu, pada penelitian ini dilakukan pengujian kinerja sebanyak lima kali untuk memperoleh data yang lebih akurat dan konsisten, sehingga hasil yang diperoleh dapat digunakan sebagai dasar analisis yang lebih andal. Adapun hasil agregat dari kelima pengujian tersebut disajikan pada Lampiran 1.

### 4.3.2 Kapasitas *Throughput*

Pengujian stress pada jaringan normal dilakukan untuk menganalisis kemampuan server dalam memproses permintaan hingga mencapai batas kapasitasnya. Hasil pengujian yang ditunjukkan pada Gambar 4.10 memperlihatkan bahwa protokol gRPC mampu mencapai throughput maksimum sekitar 4 requests/detik pada rentang 20 hingga 30 Virtual Users (VUs), sebelum mengalami stabilisasi atau penurunan performa. Sebaliknya, protokol REST menunjukkan peningkatan throughput yang lebih landai, dengan puncak performa berada di sekitar 3.5 requests/detik, serta mengalami fluktuasi yang cukup signifikan pada titik 20 VUs. Secara keseluruhan, kurva throughput gRPC secara konsisten berada di atas REST, yang mengindikasikan kapasitas pemrosesan yang lebih tinggi.




Gambar 4. 10 Kapasitas *Throughput* Di Network Normal.

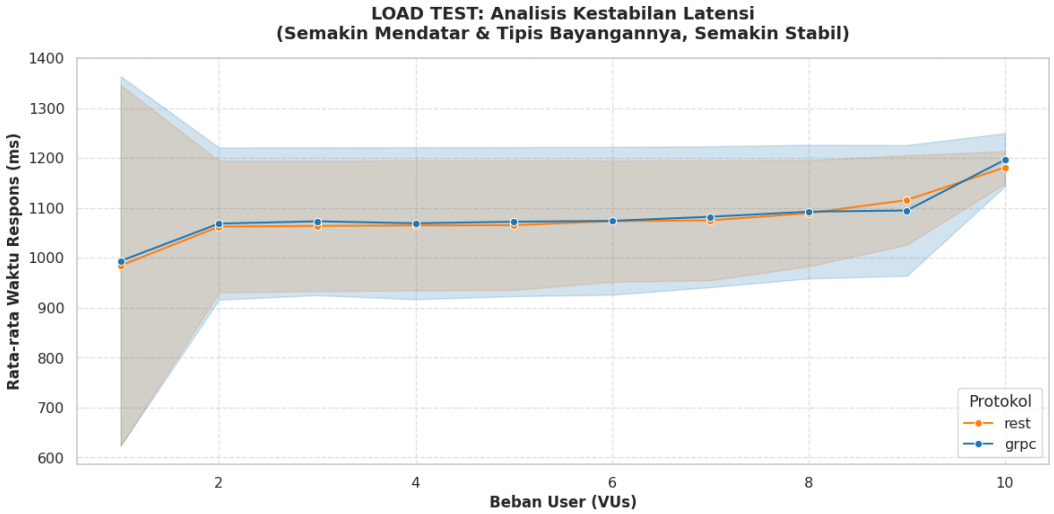
Perbedaan performa ini disebabkan oleh karakteristik protokol yang digunakan. gRPC memanfaatkan HTTP/2 yang mendukung multiplexing, sehingga memungkinkan banyak permintaan diproses secara paralel dalam satu koneksi

tanpa mengalami Head-of-Line Blocking. Selain itu, penggunaan format data biner Protocol Buffers mengurangi overhead ukuran payload dan mempercepat proses serialisasi. Sebaliknya, REST yang berjalan di atas HTTP/1.1 cenderung menggunakan model koneksi yang lebih terbatas dan rentan terhadap blocking, sehingga efisiensi pemrosesan menurun ketika jumlah permintaan meningkat secara simultan.

Implikasi dari temuan ini menunjukkan bahwa gRPC lebih mampu mempertahankan kapasitas throughput yang tinggi pada kondisi beban intensif, sehingga lebih sesuai digunakan pada sistem yang membutuhkan skalabilitas dan penanganan concurrent request dalam jumlah besar. Meskipun demikian, variasi (standar deviasi) yang lebih tinggi pada gRPC mengindikasikan adanya fluktuasi performa yang perlu diperhatikan, terutama pada kondisi mendekati titik jenuh sistem. Oleh karena itu, penggunaan gRPC memberikan keuntungan dari sisi kapasitas maksimum, namun tetap memerlukan pengelolaan sumber daya yang baik untuk menjaga stabilitas performa pada beban tinggi.

### 4.3.3 Kinerja Sistem Pada Beban Standar

Kinerja sistem dianalisis melalui load test pada jaringan normal dengan beban standar untuk mengevaluasi kestabilan latensi, sebagaimana ditunjukkan pada Gambar 4.11. Hasil pengujian menunjukkan bahwa kedua protokol, gRPC dan REST, memiliki tren latensi yang relatif serupa dan stabil pada rentang 2 hingga 9 Virtual Users (VUs), dengan waktu respons yang berada pada kisaran 1000 ms hingga 1100 ms. Pada awal pengujian (1 VU), terlihat area bayang-bayang yang lebar yang mengindikasikan variansi latensi yang tinggi. Namun, seiring peningkatan beban hingga 10 VUs, terjadi sedikit kenaikan waktu respons yang diikuti dengan penyempitan area bayang-bayang, yang menunjukkan peningkatan konsistensi sistem dalam menangani permintaan.




Gambar 4. 11 Variasi Latensi Pada *Load Test* di Jaringan Normal

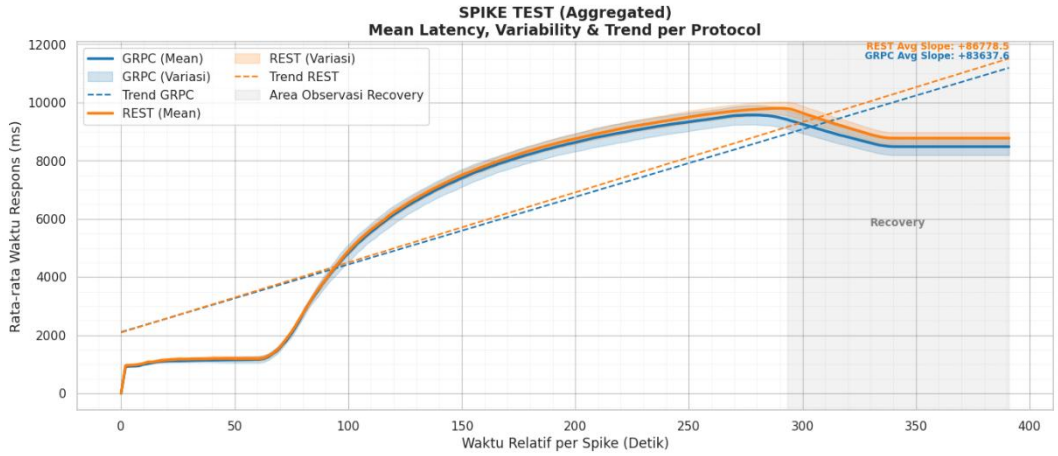
Fenomena ini dapat dijelaskan oleh kondisi sistem yang masih berada dalam kapasitas optimal pada beban rendah hingga menengah. Pada fase awal, variansi latensi cenderung tinggi akibat proses inisialisasi koneksi, alokasi resource, serta pemanasan (*warm-up*) pada komponen server. Setelah sistem mencapai kondisi steady state, baik gRPC maupun REST mampu mempertahankan kestabilan latensi karena jumlah permintaan masih berada dalam batas kapasitas pemrosesan server. Selain itu, pada skenario beban yang relatif rendah, keunggulan arsitektural gRPC seperti multiplexing HTTP/2 belum memberikan dampak signifikan dibandingkan REST.

Implikasi dari hasil ini menunjukkan bahwa pada kondisi beban standar, pemilihan antara gRPC dan REST tidak memberikan perbedaan signifikan dari sisi latensi, sehingga keduanya dapat dianggap memiliki performa yang setara dalam skenario penggunaan ringan hingga menengah. Dengan demikian, keunggulan gRPC dalam penelitian ini lebih relevan pada kondisi beban tinggi atau jaringan yang terdegradasi, bukan pada kondisi operasional normal dengan jumlah pengguna terbatas.

### 4.3.4 Kinerja Sistem terhadap Lonjakan Trafik Mendadak

Kinerja sistem terhadap lonjakan trafik mendadak dianalisis melalui spike test pada jaringan normal untuk mengevaluasi respons sistem terhadap peningkatan beban yang signifikan, sebagaimana ditunjukkan pada Gambar 4.12. Hasil

pengujian menunjukkan bahwa kedua protokol, gRPC dan REST, mengalami peningkatan latensi yang cukup tajam seiring waktu pengujian. Namun, gRPC secara konsisten mempertahankan nilai latensi yang lebih rendah dibandingkan REST. Hal ini juga tercermin pada nilai kemiringan rata-rata (Avg Slope), gRPC memiliki nilai +83637.6 yang lebih rendah dibandingkan REST sebesar +86778.5, yang menunjukkan bahwa laju peningkatan latensi pada gRPC lebih terkendali. Selain itu, pada fase akhir pengujian (setelah detik ke-350), area variasi latensi pada gRPC tampak lebih sempit dibandingkan REST, yang mengindikasikan stabilisasi performa yang lebih baik setelah sistem mencapai kondisi jenuh.




Gambar 4. 12 Peningkatan Latensi Pada *Spike Test*

Perbedaan ini disebabkan oleh karakteristik protokol gRPC yang memanfaatkan HTTP/2 dengan dukungan multiplexing dan flow control yang lebih efisien dalam menangani banyak permintaan secara simultan. Mekanisme ini memungkinkan distribusi beban yang lebih merata pada satu koneksi, sehingga mengurangi penumpukan antrian permintaan yang dapat menyebabkan lonjakan latensi. Sebaliknya, REST yang berbasis HTTP/1.1 cenderung lebih rentan terhadap peningkatan latensi secara drastis akibat keterbatasan pengelolaan koneksi dan potensi terjadinya blocking ketika terjadi lonjakan trafik yang tiba-tiba.

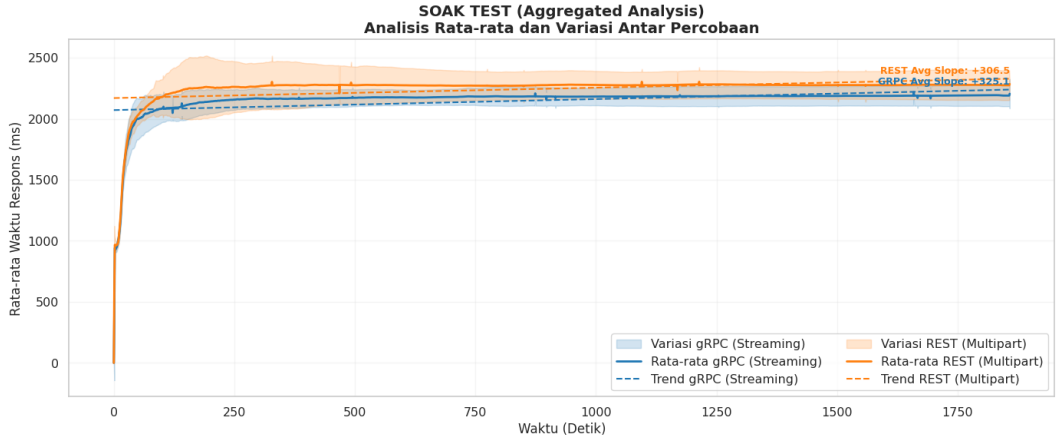
Implikasi dari hasil ini menunjukkan bahwa gRPC memiliki ketahanan yang lebih baik dalam menghadapi skenario lonjakan beban mendadak, sehingga lebih sesuai untuk digunakan pada sistem yang berpotensi mengalami trafik tidak terduga atau burst traffic. Kemampuan gRPC dalam mengendalikan laju kenaikan latensi

dan mencapai stabilisasi lebih cepat menjadi keunggulan penting dalam menjaga kualitas layanan, khususnya pada aplikasi real-time atau layanan yang sensitif terhadap keterlambatan respons.

Selain analisis tren latensi, pengukuran terhadap Recovery Time juga dilakukan untuk mengamati kecepatan sistem dalam kembali ke kondisi stabil setelah fase lonjakan beban (*spike*) berakhir. Berdasarkan hasil pengujian *spike test* yang divisualisasikan pada Gambar 4.12, protokol gRPC menunjukkan kemampuan pemulihan yang lebih responsif dibandingkan REST API. Hal ini didukung oleh data nilai *Avg Slope* gRPC yang lebih rendah, yaitu sebesar +83637,6, jika dibandingkan dengan REST yang mencapai +86778,5. Nilai kemiringan yang lebih rendah ini mengindikasikan bahwa laju peningkatan latensi pada gRPC lebih terkendali, sehingga sistem mampu mereduksi penumpukan antrean data dan kembali ke titik stabil (*baseline*) dalam waktu yang lebih singkat segera setelah beban trafik diturunkan. Efisiensi pemulihan ini merupakan dampak langsung dari penggunaan protokol HTTP/2 yang mendukung mekanisme *multiplexing* dan *flow control* yang lebih adaptif, memungkinkan *backend* gRPC mengelola sisa *requests* secara lebih efektif selama fase pemulihan dibandingkan model koneksi HTTP/1.1 pada REST yang lebih rentan terhadap fenomena *blocking*.

### 4.3.5 Stabilitas Sistem pada Beban Berkelanjutan

Stabilitas sistem pada beban berkelanjutan dianalisis melalui soak test dengan beban konstan 15 VUs selama 30 menit pada jaringan normal, sebagaimana ditunjukkan pada Gambar 4.13. Hasil pengujian menunjukkan bahwa kedua protokol mengalami lonjakan latensi di awal sebelum mencapai kondisi stabil pada kisaran 2100–2300 ms. Secara konsisten, gRPC memiliki rata-rata latensi yang lebih rendah serta variasi yang lebih kecil dibandingkan REST, yang terlihat dari area bayang-bayang yang lebih sempit.



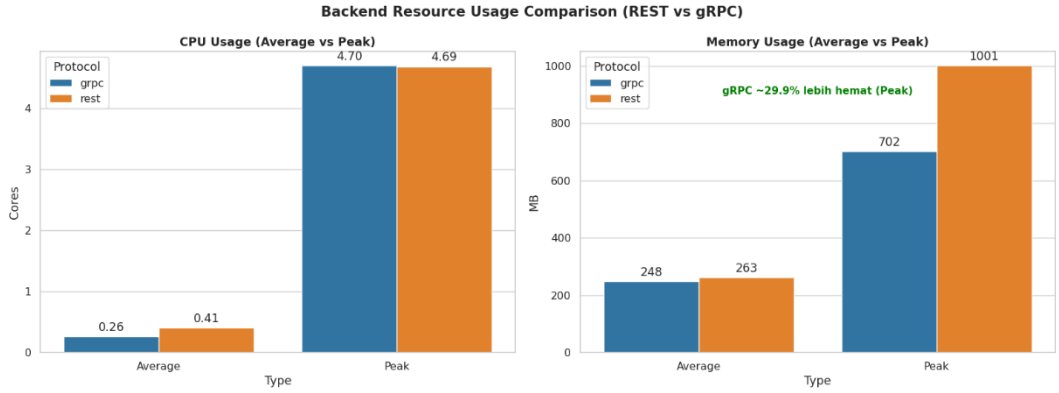

Gambar 4. 13 Latensi pada Beban Menengah dalam 30 Menit

Kondisi ini disebabkan oleh efisiensi mekanisme *streaming* gRPC berbasis HTTP/2 yang mampu mengelola aliran data dan koneksi secara lebih stabil dalam jangka panjang. Sebaliknya, pendekatan multipart pada REST cenderung menghasilkan variasi latensi yang lebih besar akibat overhead transmisi dan pengelolaan request yang kurang efisien pada beban berkelanjutan. Hasilnya, gRPC lebih mampu menjaga konsistensi performa dalam penggunaan jangka panjang, sehingga lebih cocok untuk sistem yang membutuhkan stabilitas layanan secara terus-menerus.

Analisis terhadap stabilitas sistem dalam jangka panjang juga mencakup observasi mendalam untuk mengidentifikasi potensi terjadinya kebocoran memori (memory leak) selama operasional berlangsung. Berdasarkan grafik deret waktu penggunaan sumber daya yang disajikan pada Lampiran 4, terlihat bahwa alokasi memori pada sisi *backend* untuk kedua protokol mencapai titik jenuh (*plateau*) yang stabil setelah fase inisialisasi beban puncak. Grafik tersebut menunjukkan tren penggunaan memori yang mendatar pada fase akhir pengujian tanpa adanya kenaikan akumulatif yang progresif. Selain itu, fluktuasi penurunan penggunaan memori yang terjadi secara periodik mengonfirmasi bahwa mekanisme pembebasan memori (*garbage collection*) pada lingkungan *runtime* berjalan dengan efektif untuk mengalokasikan kembali sumber daya yang sudah tidak digunakan. Pola konsumsi sumber daya yang konsisten ini memberikan bukti empiris bahwa arsitektur sistem memiliki manajemen memori yang stabil dan bebas dari indikasi *memory leak* selama durasi pengujian *soak test* berlangsung.

### 4.3.6 Penggunaan CPU dan Memori Server

Penggunaan CPU dan memori dianalisis selama lima kali pengujian untuk mengamati pola pemanfaatan sumber daya sistem serta mengidentifikasi potensi bottleneck yang dapat memengaruhi kinerja backend, sebagaimana ditunjukkan pada Gambar 4.14. Hasil pengujian menunjukkan bahwa kedua protokol, gRPC dan REST, memiliki puncak penggunaan CPU yang hampir identik, yaitu 4.70 Cores untuk gRPC dan 4.69 Cores untuk REST, yang mengindikasikan bahwa keduanya mampu memanfaatkan kapasitas maksimum prosesor secara penuh pada kondisi beban tinggi. Namun, jika dilihat dari rata-rata penggunaan CPU, gRPC menunjukkan efisiensi yang lebih baik dengan nilai sebesar 0.26 Cores dibandingkan REST yang mencapai 0.41 Cores. Perbedaan yang lebih signifikan terlihat pada penggunaan memori, REST mencatatkan puncak alokasi hingga 1000.97 MB, sedangkan gRPC hanya mencapai 701.69 MB. Hal ini menunjukkan bahwa gRPC mampu menghemat penggunaan memori sekitar 299 MB dibandingkan REST selama periode pengujian. Untuk detail grafik penggunaan sumber daya secara lengkap, dapat dilihat pada Lampiran 4.




Gambar 4. 14 Perbandingan Sumber Daya Yang Terpakai Tiap Protokol

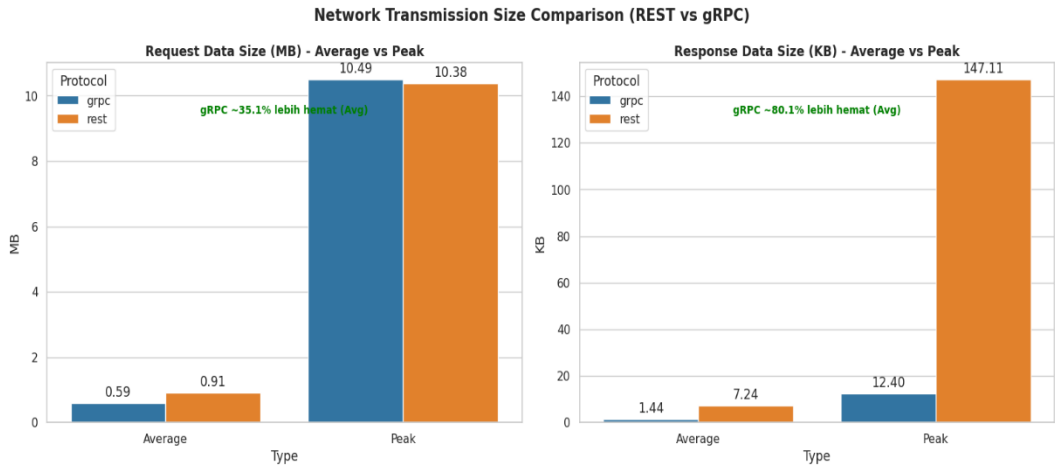
Perbedaan efisiensi ini disebabkan oleh karakteristik arsitektur komunikasi masing-masing protokol. gRPC memanfaatkan protokol HTTP/2 yang mendukung koneksi persisten serta mekanisme *multiplexing*, sehingga memungkinkan banyak permintaan diproses secara paralel dalam satu koneksi tanpa perlu membuat *thread* atau koneksi baru secara berulang. Pendekatan ini secara signifikan mengurangi *overhead* manajemen koneksi serta alokasi memori. Sebaliknya, REST yang menggunakan pendekatan *multipart* berbasis HTTP/1.1 cenderung bersifat

*stateless*, sehingga setiap permintaan diproses secara independen dan sering kali memerlukan pembentukan koneksi baru atau alokasi resource tambahan. Hal ini menyebabkan peningkatan konsumsi memori yang cukup signifikan, terutama pada skenario pengiriman data citra berukuran besar yang dilakukan secara berulang.

Implikasi dari temuan ini menunjukkan bahwa gRPC lebih efisien dalam pemanfaatan sumber daya komputasi, khususnya dalam penggunaan memori dan rata-rata CPU, sehingga lebih mendukung skalabilitas sistem *backend* dalam jangka panjang. Efisiensi ini menjadi faktor penting pada sistem yang menangani beban kerja intensif dan kontinu, karena dapat mengurangi risiko *bottleneck* serta meningkatkan kemampuan server dalam menangani lebih banyak permintaan secara simultan tanpa mengalami degradasi performa yang signifikan.

### 4.3.7 Ukuran Data Transmisi

Ukuran data transmisi dianalisis untuk mengevaluasi efisiensi komunikasi antara client dan server, sebagaimana ditunjukkan pada Gambar 4.15. Hasil pengujian menunjukkan bahwa ukuran data transmisi masuk (request) relatif serupa karena bergantung pada ukuran file citra, dengan rata-rata gRPC sebesar 0.59 MB dan REST sebesar 0.91 MB. Namun, perbedaan signifikan terlihat pada data keluaran (response), gRPC hanya membutuhkan rata-rata 1.44 KB (puncak 12.40 KB), sedangkan REST mencapai rata-rata 7.24 KB dengan lonjakan hingga 147.11 KB. Hal ini menunjukkan bahwa gRPC menghasilkan *payload response* yang jauh lebih kecil dibandingkan REST. Untuk detail grafik, dapat dilihat pada Lampiran 5.




Gambar 4. 15 Perbandingan Ukuran Data Transmisi Tiap Protokol

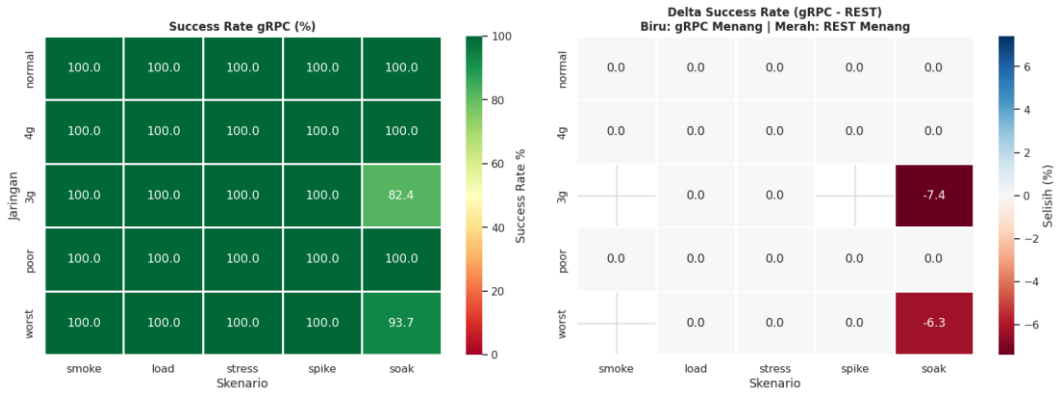
Perbedaan ini disebabkan oleh penggunaan format biner *Protocol Buffers* (Protobuf) pada gRPC yang lebih ringkas dibandingkan format teks JSON pada REST. Selain itu, dukungan kompresi *header* melalui mekanisme HPACK pada HTTP/2 turut mengurangi overhead transmisi, sementara penggunaan multipart/form-data pada REST menambah ukuran payload secara signifikan.

gRPC lebih efisien dalam penggunaan bandwidth jaringan, terutama pada pengiriman response, sehingga lebih cocok untuk sistem yang membutuhkan komunikasi data ringan, cepat, dan skalabel, khususnya pada lingkungan dengan keterbatasan jaringan.

## 4.4 Temuan Penelitian

### 4.4.1 Tingkat Keberhasilan

Pada pengujian dengan kondisi jaringan fluktuatif, sebagaimana ditunjukkan pada Gambar 4.16, terlihat bahwa gRPC mengalami penurunan tingkat keberhasilan (success rate) pada skenario tertentu. Secara umum, gRPC mampu mencapai tingkat keberhasilan 100% pada sebagian besar skenario pengujian. Namun, pada skenario soak test dengan kondisi jaringan yang lebih buruk, terjadi penurunan signifikan, yaitu sebesar 82.4% pada jaringan 3G dan 93.7% pada kondisi worst. Berdasarkan grafik delta success rate, gRPC menunjukkan performa yang lebih rendah dibandingkan REST dengan selisih masing-masing sebesar -7.4% dan -6.3% pada kedua kondisi tersebut, yang mengindikasikan adanya degradasi reliabilitas pada jaringan yang tidak stabil.



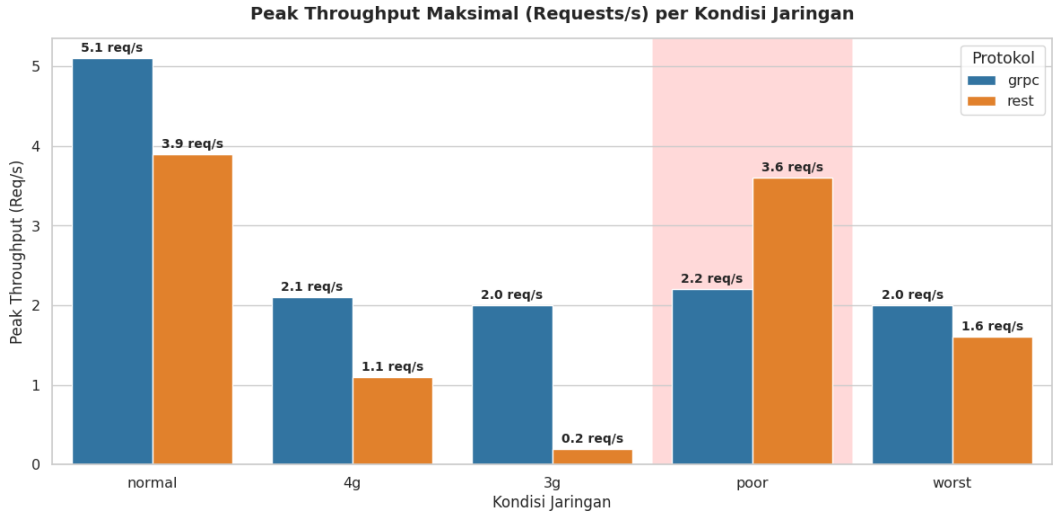

Gambar 4. 16 Tingkat Keberhasilan Sistem gRPC

Penurunan ini disebabkan oleh karakteristik gRPC yang menggunakan koneksi persisten (*long-lived connection*) berbasis HTTP/2. Pada kondisi jaringan dengan latensi tinggi dan packet loss yang berulang, koneksi yang bersifat kontinu ini menjadi lebih rentan terhadap gangguan, seperti terputusnya *stream* (*broken stream*) selama proses komunikasi berlangsung. Dalam skenario pengujian jangka panjang (soak test), akumulasi gangguan jaringan memperbesar kemungkinan kegagalan koneksi. Sebaliknya, REST yang menggunakan pendekatan stateless berbasis HTTP/1.1 cenderung lebih toleran terhadap gangguan jaringan, karena setiap permintaan dilakukan secara independen sehingga memungkinkan proses retry atau handshake ulang terjadi tanpa memengaruhi permintaan lainnya.

Implikasi dari temuan ini menunjukkan bahwa meskipun gRPC unggul dari sisi efisiensi dan performa pada kondisi normal dan beban tinggi, protokol ini memiliki kelemahan dalam hal reliabilitas pada jaringan yang tidak stabil dalam jangka waktu lama. Oleh karena itu, penggunaan gRPC pada lingkungan dengan kualitas jaringan rendah perlu disertai mekanisme tambahan seperti retry, timeout handling, atau fallback strategy untuk menjaga tingkat keberhasilan sistem.

### 4.4.2 Anomali *Throughput* Pada Jaringan Tertentu

Berdasarkan hasil pengujian pada berbagai kondisi jaringan, sebagaimana ditunjukkan pada Gambar 4.17, ditemukan adanya anomali pada nilai throughput, khususnya pada kondisi jaringan *poor*. Pada kondisi ini, REST justru mencatat throughput lebih tinggi sebesar 3.6 req/s dibandingkan gRPC yang hanya mencapai 2.2 req/s, yang bertolak belakang dengan tren umum pada kondisi jaringan lainnya. Di luar kondisi tersebut, gRPC tetap menunjukkan performa yang lebih tinggi dan stabil, seperti pada jaringan normal dengan throughput mencapai 5.1 req/s dibandingkan REST sebesar 3.9 req/s.



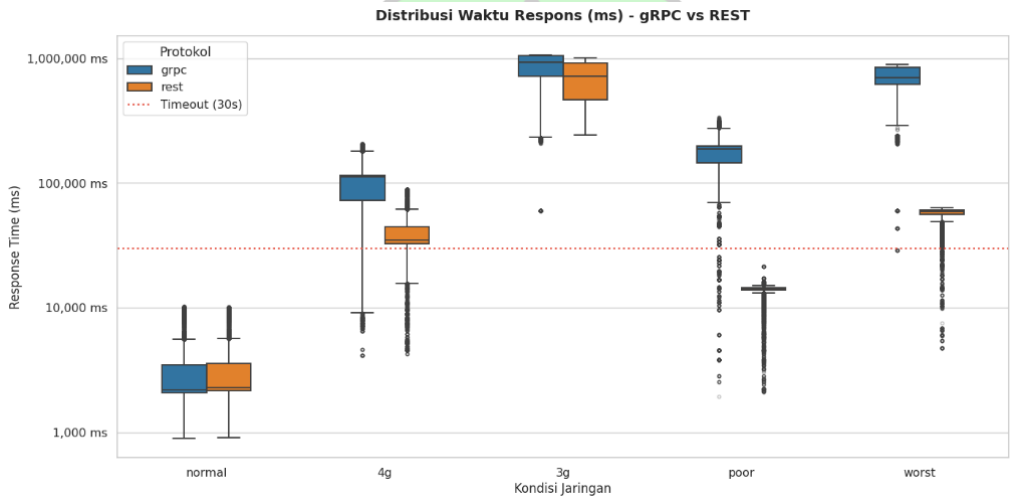

Gambar 4. 17 *Throughput* Maksimal Pada Tiap Jaringan

Fenomena ini disebabkan oleh perbedaan mekanisme kontrol transmisi pada kedua protokol. gRPC yang berjalan di atas HTTP/2 menerapkan flow control yang adaptif dan cenderung konservatif, sehingga secara otomatis menurunkan laju pengiriman data saat terjadi gangguan jaringan seperti latensi tinggi atau *packet loss*. Pendekatan ini menjaga stabilitas sistem, namun berdampak pada penurunan throughput. Sebaliknya, REST berbasis HTTP/1.1 tidak memiliki mekanisme flow control yang ketat, sehingga tetap mengirimkan permintaan secara agresif meskipun kondisi jaringan memburuk. Hal ini membuat throughput terlihat lebih tinggi, namun dengan konsekuensi peningkatan penggunaan sumber daya, sebagaimana didukung oleh hasil pada Gambar 4.14 yang menunjukkan konsumsi memori REST yang lebih besar.

*Throughput* yang lebih tinggi pada REST dalam kondisi jaringan *poor* tidak selalu mencerminkan efisiensi yang sebenarnya, melainkan menunjukkan pendekatan agresif yang berpotensi menimbulkan bottleneck pada sistem. Sebaliknya, gRPC menunjukkan karakteristik yang lebih stabil dan terkontrol dalam menghadapi variasi kualitas jaringan, sehingga lebih andal untuk penggunaan jangka panjang meskipun tidak selalu menghasilkan throughput maksimum pada kondisi tertentu.

### 4.4.3 Latensi Buruk Pada Jaringan Tidak Normal

Sejalan dengan penelitian terdahulu oleh Niswar et al. yang menyatakan bahwa gRPC unggul dalam throughput dan latensi, hasil penelitian ini menunjukkan temuan yang berbeda pada kondisi jaringan tertentu, sebagaimana ditunjukkan pada Gambar 4.18. Pada kondisi jaringan yang menurun dari 4G hingga *poor* dan *worst*, gRPC justru menunjukkan distribusi waktu respons yang lebih tinggi dan lebih menyebar dibandingkan REST. Hal ini terlihat dari sebaran data yang lebih lebar serta median latensi gRPC yang pada beberapa skenario melampaui ambang batas *timeout* 30 detik. Sebaliknya, REST meskipun mengalami peningkatan latensi, tetap menunjukkan distribusi yang lebih terkendali dan cenderung berada di sekitar batas *timeout*.



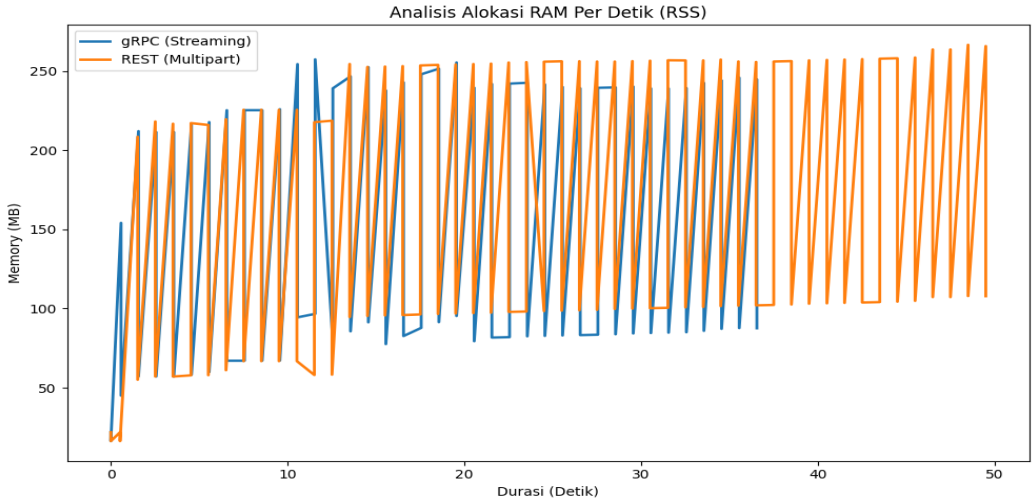

Gambar 4. 18 Distribusi Waktu *Response*

Perbedaan ini disebabkan oleh sensitivitas gRPC terhadap kondisi jaringan yang tidak stabil. gRPC yang berbasis HTTP/2 menggunakan mekanisme kontrol seperti flow control dan manajemen *stream* yang adaptif, sehingga ketika terjadi packet loss atau latensi tinggi, sistem akan menyesuaikan laju transmisi secara konservatif. Dampaknya, waktu respons dapat meningkat secara signifikan dan menjadi lebih variatif (tinggi jitter). Sebaliknya, REST berbasis HTTP/1.1 dengan pendekatan stateless cenderung tidak terlalu terpengaruh oleh kondisi koneksi sebelumnya, sehingga setiap permintaan dapat diproses secara independen tanpa akumulasi dampak dari gangguan jaringan.

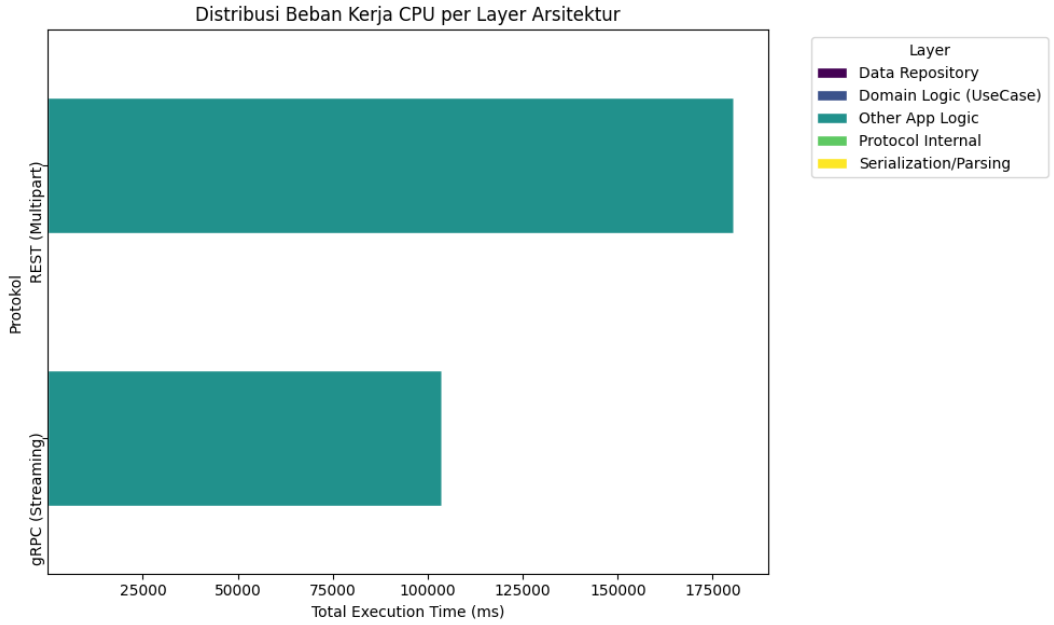
Meskipun gRPC unggul pada kondisi jaringan normal dan stabil, protokol ini menunjukkan kelemahan dalam hal kestabilan latensi pada jaringan yang buruk. Dalam skenario dengan kualitas jaringan rendah, REST justru memberikan performa yang lebih konsisten dari sisi waktu respons, sehingga lebih tahan terhadap fluktuasi jaringan. Temuan ini menegaskan bahwa pemilihan protokol komunikasi tidak hanya bergantung pada efisiensi teoritis, tetapi juga harus mempertimbangkan karakteristik jaringan yang digunakan.

### 4.4.4 Beban Kerja CPU dan Memori Aplikasi Client

Beban kerja CPU dan memori pada aplikasi client dianalisis menggunakan Android Profiler melalui pengujian masing-masing protokol, sebagaimana ditunjukkan pada Gambar 4.19 dan 4.20. Hasil pengujian menunjukkan bahwa gRPC memiliki total beban CPU yang lebih rendah, yaitu sebesar 25.842,84 ms, dibandingkan REST yang mencapai 37.463,39 ms (lihat Lampiran 2). Selain itu, pada aspek memori, gRPC juga menunjukkan penggunaan yang lebih efisien dengan puncak alokasi sebesar 257,53 MB, lebih rendah dibandingkan REST sebesar 266,73 MB. Perbedaan ini juga didukung oleh waktu proses *serialization/parsing* yang lebih cepat pada gRPC, yaitu 1,08 ms dibandingkan REST sebesar 2,15 ms (lihat Lampiran 3).




Gambar 4. 19 Alokasi RAM Selama Satu Kali Percobaan




Gambar 4. 20 Distribusi Beban Kerja CPU Per-Protokol

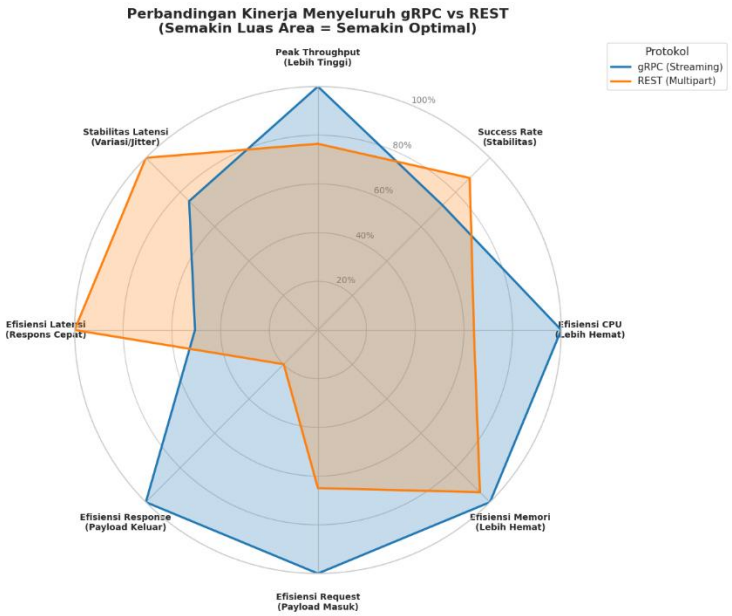
Perbedaan efisiensi ini disebabkan oleh tingginya overhead pada REST, khususnya pada lapisan *other app logic* yang mencapai 180.467,91 ms, jauh lebih besar dibandingkan gRPC yang sebesar 103.710,68 ms. Hal ini berkaitan dengan penggunaan format teks JSON dan mekanisme multipart yang memerlukan proses parsing serta manipulasi data yang lebih kompleks. Sebaliknya, gRPC menggunakan format biner Protocol Buffers yang lebih ringan dan efisien dalam proses serialisasi. Meskipun gRPC memiliki tambahan beban pada lapisan protokol internal, dampaknya relatif kecil dan tidak signifikan terhadap total beban kerja sistem.

Efeknya gRPC lebih optimal dalam mengelola sumber daya CPU dan memori pada sisi client, sehingga lebih sesuai untuk digunakan pada perangkat dengan keterbatasan resource seperti aplikasi mobile. Efisiensi ini tidak hanya berdampak pada performa aplikasi, tetapi juga berpotensi meningkatkan daya tahan baterai serta responsivitas sistem secara keseluruhan.

### 4.5 Sintesis Hasil Pengujian Kinerja gRPC dan REST

Bagian ini menyajikan sintesis kinerja gRPC dan REST dalam bentuk diagram radar pada Gambar 4.21, luas area merepresentasikan kekuatan relatif masing-masing protokol pada berbagai metrik pengujian. Hasil visualisasi

menunjukkan bahwa gRPC memiliki area yang lebih dominan pada sebagian besar dimensi, terutama pada throughput dan efisiensi penggunaan sumber daya. Di sisi lain, REST menunjukkan keunggulan pada aspek latensi, stabilitas latensi (lihat Subbab 4.4.4), serta tingkat keberhasilan (*success rate*) yang lebih tinggi pada kondisi tertentu (lihat Subbab 4.4.1).




Gambar 4. 21 Perbandingan Kinerja Secara Menyeluruh

Perbedaan ini dipengaruhi oleh desain protokol, gRPC (HTTP/2) lebih efisien dalam pengolahan data dan manajemen koneksi, sementara REST (HTTP/1.1) lebih toleran terhadap gangguan jaringan. Dengan demikian, gRPC lebih sesuai untuk kebutuhan performa tinggi dan efisiensi sistem, sedangkan REST lebih unggul dalam menjaga kestabilan dan reliabilitas pada jaringan yang tidak stabil.

# BAB V
# KESIMPULAN DAN SARAN

Berdasarkan hasil pengujian, protokol gRPC dengan mekanisme *client-side streaming* menunjukkan kinerja yang lebih unggul dibandingkan REST API dalam hal efisiensi transmisi data, pemanfaatan sumber daya, dan *throughput*. Namun, pada kondisi jaringan yang tidak stabil, reliabilitas gRPC menurun karena lebih sensitif terhadap packet loss dan latensi tinggi. Oleh karena itu, pemilihan protokol komunikasi perlu disesuaikan dengan kebutuhan sistem dan kondisi jaringan yang dihadapi.

## 5.1 Kesimpulan

1. Efektivitas *client-side streaming* pada Jaringan terdegradasi, Penerapan mekanisme client-side streaming pada gRPC berhasil menjaga stabilitas pengiriman citra medis berukuran besar, namun menunjukkan trade-off pada kondisi ekstrem. Meskipun gRPC memiliki waktu penyelesaian (completion time) yang sangat baik di jaringan normal, pada kondisi jaringan 3G dan worst, gRPC mengalami penurunan tingkat keberhasilan (success rate) menjadi 82.4% dan 93.7% akibat sensitivitas protokol HTTP/2 terhadap packet loss yang berkepanjangan dibandingkan REST API yang bersifat stateless.

2. Efisiensi transmisi dan ukuran payload, gRPC menunjukkan efisiensi transmisi data yang lebih baik dibandingkan REST API. Dengan penggunaan Protocol Buffers dan kompresi header pada HTTP/2, ukuran *response* rata-rata gRPC hanya sebesar 1.44 KB, jauh lebih kecil dibandingkan REST API yang mencapai 7.24 KB. Hal ini mengindikasikan bahwa gRPC lebih hemat dalam penggunaan bandwidth, khususnya pada pengiriman data hasil prediksi.

3. Dampak terhadap Utilitas Sumber Daya Sisi Client, gRPC terbukti lebih efisien dalam penggunaan CPU dan memori pada perangkat *client* (Android). Hasil *profiling* menunjukkan total beban kerja CPU gRPC hanya 25.842,84 ms, jauh lebih rendah dibandingkan REST yang mencapai 37.463,39 ms. Efisiensi ini dipengaruhi oleh waktu serialisasi/

*parsing* format biner Protobuf yang dua kali lebih cepat (**1,08 ms**) dibandingkan format teks JSON pada REST (2,15 ms). Selain itu, alokasi puncak memori (RAM) pada gRPC juga lebih rendah, yaitu 257,53 MB berbanding 266,73 MB pada REST. Efisiensi kuantitatif ini sangat mendukung kelancaran penggunaan aplikasi *mobile* yang memiliki keterbatasan sumber daya baterai dan komputasi.

4. Ketahanan Infrastruktur Backend dan Konkurensi, Pada sisi *backend*, gRPC mampu mencapai *throughput* puncak yang lebih tinggi, yaitu mencapai 5,1 requests/detik dibandingkan REST yang hanya 3,9 requests/detik pada kondisi jaringan normal. Selain itu, gRPC menunjukkan efisiensi penggunaan sumber daya server yang signifikan, dengan rata-rata utilitas CPU hanya 0,26 Cores (berbanding 0,41 Cores pada REST) dan penghematan alokasi puncak memori hingga ~299 MB (gRPC 701,69 MB vs REST 1.000,97 MB). Stabilitas metrik ini menegaskan bahwa gRPC lebih andal dalam menangani lonjakan beban tinggi dan permintaan secara simultan, sehingga lebih sesuai untuk arsitektur sistem dengan kebutuhan skalabilitas yang tinggi.

## 5.2 Saran

1. **Pengembangan Mekanisme *Auto-Retry* Berbasis *Chunk***, Mengingat adanya penurunan *success rate* pada jaringan buruk (3G/*Worst*), disarankan untuk menambahkan fitur pengiriman ulang otomatis pada level *chunk* tertentu yang gagal, sehingga sistem tidak perlu mengulang seluruh proses *streaming* dari awal saat terjadi gangguan koneksi.

2. **Implementasi *Adaptive Chunk Size***, Penentuan ukuran *chunk* sebesar 256 KB pada penelitian ini bersifat tetap. Saran untuk penelitian selanjutnya adalah mengimplementasikan algoritma yang dapat menyesuaikan ukuran *chunk* secara dinamis berdasarkan kualitas sinyal (*Bandwidth-Delay Product*) untuk mengoptimalkan utilisasi jaringan seluler.

3. **Optimasi Kendali Aliran (Flow Control) HTTP/2**, Diperlukan penelitian lebih lanjut mengenai konfigurasi parameter flow control pada

level protokol gRPC untuk meningkatkan ketangguhan koneksi long-lived terhadap packet loss tinggi, guna meminimalkan insiden broken *stream* yang ditemukan pada skenario soak test.

4. **Perluasan Pengujian pada Variasi Perangkat**, Disarankan untuk melakukan profiling kinerja pada berbagai kelas perangkat Android (low-end hingga high-end) dengan versi OS yang berbeda, guna memvalidasi konsistensi utilisasi CPU dan RAM client di seluruh ekosistem Android.
5. **Peningkatan Akurasi dan Validitas Medis Model**, Mengingat fokus penelitian saat ini masih terbatas pada performa transmisi data, disarankan untuk melakukan pelatihan ulang (retraining) atau modifikasi arsitektur model CNN menggunakan dataset yang lebih variatif guna meningkatkan akurasi klasifikasi. Hal ini diperlukan untuk memastikan bahwa sistem tidak hanya unggul secara teknis dalam pengiriman data, tetapi juga memberikan hasil diagnosis yang valid dan dapat dipertanggungjawabkan secara klinis.

# DAFTAR PUSTAKA


3rd Generation Partnership Project. (1999). *Universal Mobile Telecommunications System (UMTS); General Description; Release 1999 (TS 21.101)*.

Android. (2025). *Android Open Source Project*. https://source.android.com/?hl=id

Android Developers. (2025). *Alat Developer Aplikasi Seluler Android – Developer Android | Android Developers*. https://developer.android.com/?hl=id

Birrell, A. D., & Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, *2*(1), 39–59. https://doi.org/10.1145/2080.357392

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2011). *Distributed Systems: Concepts and Design* (5th ed.). Addison-Wesley Publishing Company.

de Matos, F. F. S. B., Rego, P. A. L., & Trinta, F. A. M. (2021). Secure Computational Offloading with gRPC: A Performance Evaluation in a Mobile Cloud Computing Environment. *Proceedings of the 11th ACM Symposium on Design and Analysis of Intelligent Vehicular Networks and Applications, DIVANet '21*, 45–52. https://doi.org/10.1145/3479243.3487295

Debelee, T. G. (2023). Skin Lesion Classification and Detection Using Machine Learning Techniques: A Systematic Review. *Diagnostics*, *13*(19). https://doi.org/10.3390/diagnostics13193147

Esteva, A., Kuprel, B., Novoa, R. A., Ko, J., Swetter, S. M., Blau, H. M., & Thrun, S. (2017). Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, *542*(7639), 115–118. https://doi.org/10.1038/nature21056

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*.

Grigorik, I. (2013). *High-Performance Browser Networking*.

gRPC.io. (2025, November 12). *Introduction to gRPC | gRPC*. https://grpc.io/docs/what-is-grpc/introduction/

Hadi, Z. G., Ajel, A. R., & Hussain, A. (2021). Comparison Between Convolutional Neural Network CNN and SVM in Skin Cancer Images Recognition. In

*Journal of Techniques* (Vol. 3, Number 4). http://creativecommons.org/licenses/by/4.0/

Hari, K. S. M., & Sharma, R. (2024). COMPARATIVE STUDY OF ORCHESTRATION USING GRPC API AND REST API IN SERVER CREATION TIME: AN OPENSTACK CASE. *International Journal of Computer Networks and Communications*, *16*(1), 87–104. https://doi.org/10.5121/ijcnc.2024.16106

Hikmah, N., Izzuddin, A., & Velinda, S. (2025). Deep Learning-Based Implementation of Convolutional Neural Networks for Skin Disease Detection Through Image Classification on Mobile Platforms. *ENERGY: JURNAL ILMIAH ILMU-ILMU TEKNIK*, *15*(2), 324–334. https://doi.org/10.51747/energy.v15i2.15216

Indrasiri, K., & Kuruppu, D. (2020). *GRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes*. O'Reilly Media. https://books.google.co.id/books?id=Az7gxwEACAAJ

International Telecommunication Union. (2003). *Recommendation G.114: One-way transmission time*.

Janbi, N., Mehmood, R., Katib, I., Albeshri, A., Corchado, J. M., & Yigitcanlar, T. (2022). Imtidad: A Reference Architecture and a Case Study on Developing Distributed AI Services for Skin Disease Diagnosis over Cloud, Fog and Edge. *Sensors*, *22*(5). https://doi.org/10.3390/s22051854

Katz, C., Ruiz, J. M., Saigí-Rubió, F., & Novillo-Ortiz, D. (2025). The State of the Art of Telemedicine Implementation Architecture: Rapid Umbrella Review of Systematic Reviews. *J Med Internet Res*, *27*, e70276. https://doi.org/10.2196/70276

Kelly, B. T., & Xiu, D. (2023). *Financial Machine Learning*.

Kumar, A., & Pal, A. (2025). A Survey on Computation Offloading and Current Trends. *IEEE Access*, *PP*, 1. https://doi.org/10.1109/ACCESS.2025.3625187

Kurose, J. F., & Ross, K. W. (2024). Computer Networking. *TOP DOWN APP*.

Kushartanto, A. I., Fauziah, F., & Aldisa, R. T. (2024). Comparison of CNN and SVM Methods on Web-based Skin Disease Classification Process. *Sinkron*, *8*(2), 778–788. https://doi.org/10.33395/sinkron.v8i2.13349

Lagdhir, G. (2025, July 21). *GraphQL in CMS architecture - Data Fetching Made Efficient*. https://gtcsys.com/transform-your-cms-with-graphql-data-fetching-made-efficient/

Makadia, M. (2025). *gRPC vs REST – Speed, Performance & Which Is Faster?* https://marutitech.com/rest-vs-grpc/

Malik, S. G., Jamil, S. S., Aziz, A., Ullah, S., Ullah, I., & Abohashrh, M. (2024). High-Precision Skin Disease Diagnosis through Deep Learning on Dermoscopic Images. *Bioengineering*, *11*(9). https://doi.org/10.3390/bioengineering11090867

Melnikov, A., & Fette, L. (2011, December). *RFC 6455 - The WebSocket Protocol*. https://datatracker.ietf.org/doc/html/rfc6455

Mun, T. S. H., Holbrey, R., Chua, S., Huang, P., Messiou, C., Doran, S. J., & Blackledge, M. D. (2022). *OsiriXgrpc: Rapid development and deployment of state-of-the-art artificial intelligence for clinical practice.*

Newman, S. (2015). *Building Microservices* (1st ed.). O'Reilly Media, Inc.

Niswar, M., Arisandy Safruddin, R., Bustamin, A., & Aswad, I. (2024). Performance evaluation of microservices communication with REST, GraphQL, and gRPC. *International Journal of Electronics and Telecommunications*, *70*(No 2), 429–436. https://doi.org/10.24425/ijet.2024.149562

Onsman, A. (2023, November 1). *Remote Procedure Call (RPC)*. https://www.tutorialspoint.com/remote-procedure-call-rpc

OpenSignal. (2020). *The State of Mobile Network Experience*.

O'Riordan, M. (2024, September 2). *The Road to WebSockets | WebSocket.org*. https://websocket.org/guides/road-to-websockets/

Quiña-Mera, A., Fernandez, P., Garc\'{\i}a, J. M., & Ruiz-Cortés, A. (2023). GraphQL: A Systematic Mapping Study. *ACM Comput. Surv.*, *55*(10). https://doi.org/10.1145/3561818

Richardson, L., & Ruby, S. (2007). *Restful web services* (First). O'Reilly.

Steen, M. van., & Tanenbaum, A. S. . (2017). *Distributed systems*. Maarten van Steen.

Stockhammer, T. (2011). Dynamic adaptive streaming over HTTP –: standards and design principles. *Proceedings of the Second Annual ACM Conference on Multimedia Systems, MMSys '11*, 133–144. https://doi.org/10.1145/1943552.1943572

Taha, M., Abdulrahman, N., Mohammed, A., & Hama, D. (2024). Impact of Wireless Network Packet Loss on Real-Time Video Streaming Application: A Comparative Study of H.265 and H.266 Codecs. *Kurdistan Journal of Applied Research*, *9*, 23–41. https://doi.org/10.24017/science.2024.2.3

Yanuardi, M., Aminudin, A., & Faiqurahman, M. (2024). Analisis Perbandingan Efektivitas Arsitektur Restful dan Arsitektur Grpc pada Implementasi Web Service. *Jurnal Edukasi Dan Penelitian Informatika (JEPIN)*, *10*, 333. https://doi.org/10.26418/jp.v10i2.80845

# DAFTAR ISTILAH

Berikut adalah daftar istilah teknis beserta penjelasannya yang digunakan dalam penelitian ini.

**Protocol Buffer Definition File (.proto)**

Berkas definisi yang digunakan untuk mendefinisikan skema layanan dan struktur pesan dalam gRPC menggunakan bahasa Interface Definition Language (IDL) Protobuf. Berkas ini dikompilasi untuk menghasilkan kode stub.

**Application Programming Interface (API)**

Antarmuka yang memungkinkan dua aplikasi atau sistem untuk saling berkomunikasi dan bertukar data. API mendefinisikan aturan dan protokol tentang bagaimana komponen perangkat lunak berinteraksi.

**Bandwidth**

Kapasitas maksimum suatu saluran komunikasi untuk mentransfer data dalam satuan waktu tertentu, biasanya diukur dalam Kbps (kilobit per detik) atau Mbps (megabit per detik).

**Bandwidth-Delay Product (BDP)**

Ukuran yang menyatakan jumlah data yang dapat ada dalam jaringan pada satu waktu tertentu, dihitung dengan mengalikan bandwidth dengan latensi (delay). Digunakan sebagai dasar penentuan ukuran chunk dalam penelitian ini.

**Black Box Testing**

Metode pengujian perangkat lunak yang mengevaluasi fungsionalitas sistem tanpa melihat struktur internal atau kode sumber. Pengujian berfokus pada input dan output yang dihasilkan sistem.

**Bottleneck**

Titik dalam sistem yang membatasi kinerja atau kapasitas keseluruhan sistem. Bottleneck terjadi ketika satu komponen memiliki kapasitas lebih rendah dibandingkan komponen lainnya.

**Broken Stream**

Kondisi di mana aliran data (*stream*) dalam komunikasi gRPC terputus sebelum selesai, biasanya akibat gangguan jaringan seperti packet loss atau timeout.

**Container Advisor (cAdvisor)**

Alat pemantauan sumber daya yang menganalisis penggunaan CPU dan memori container secara real-time. Digunakan untuk memantau kinerja backend server dalam penelitian ini.

**Chunk**

Potongan data berukuran tertentu yang digunakan dalam mekanisme *streaming*. Dalam penelitian ini, citra medis dipecah menjadi potongan-potongan berukuran 256 KB sebelum dikirim ke server.

**Chunk Size**

Ukuran setiap potongan data yang dikirim dalam proses *streaming*. Dalam penelitian ini, chunk size ditetapkan sebesar 256 KB sebagai nilai baseline yang konsisten.

**Client-Side Streaming**

Pola komunikasi dalam gRPC di mana client mengirimkan serangkaian data (*stream*) ke server, kemudian server membalas dengan satu respons setelah seluruh data diterima.

**Convolutional Neural Network (CNN)**

Jenis arsitektur jaringan saraf tiruan yang dirancang khusus untuk memproses data berbentuk grid seperti citra digital. CNN digunakan dalam penelitian ini untuk mengklasifikasikan penyakit kulit.

**Completion Time**

Waktu yang dibutuhkan untuk menyelesaikan satu siklus pengiriman data dari awal hingga penerimaan respons. Digunakan sebagai salah satu metrik pengujian kinerja.

**Compute Engine**

Layanan mesin virtual (VM) berbasis cloud yang menyediakan infrastruktur komputasi skalabel. Digunakan sebagai server backend dan mesin pengujian dalam penelitian ini.

**Concurrent Stream**

Beberapa aliran data yang diproses secara bersamaan dalam satu koneksi. Kemampuan ini didukung oleh protokol HTTP/2 yang digunakan oleh gRPC.

**Deadline/Timeout**

Batas waktu maksimum yang diberikan untuk menyelesaikan suatu operasi. Dalam penelitian ini, batas waktu ditetapkan sebesar 30 detik untuk setiap transmisi gRPC.

**Deployment**

Proses penempatan atau pengiriman aplikasi/layanan ke lingkungan produksi agar dapat diakses oleh pengguna akhir.

**Edge Case**

Skenario ekstrem atau kondisi batas yang jarang terjadi tetapi perlu diuji untuk memastikan ketahanan sistem. Contohnya: pengunggahan berkas dengan format tidak valid atau interupsi jaringan mendadak.

**Error Handling**

Mekanisme dalam pemrograman untuk mendeteksi, merespons, dan memulihkan dari kondisi error atau pengecualian (*exception*) yang terjadi selama eksekusi program.

**Exception**

Kondisi error yang terjadi saat runtime yang dapat ditangkap dan ditangani oleh blok catch dalam kode program, sehingga aplikasi tidak crash.

**Flow Control**

Mekanisme dalam protokol komunikasi untuk mengatur laju pengiriman data antara pengirim dan penerima, mencegah pengirim mengoverwhelm penerima dengan data yang terlalu banyak.

**Go / Golang**

Bahasa pemrograman yang dikembangkan oleh Google, dikenal karena performa tinggi dan efisiensi dalam pemrograman konkuren. Digunakan untuk mengembangkan sisi server dalam penelitian ini.

**GraphQL**

Bahasa query untuk API yang memungkinkan client mendefinisikan secara tepat data apa yang dibutuhkan dari server, mengurangi over-fetching dan under-fetching data.

**gRPC**

Framework RPC modern yang dikembangkan oleh Google, menggunakan HTTP/2 sebagai protokol transport dan Protocol Buffers sebagai mekanisme serialisasi data. gRPC mendukung berbagai pola komunikasi termasuk *streaming*.

**Head-of-Line Blocking**

Fenomena dalam HTTP/1.1 di mana request berikutnya harus menunggu request sebelumnya selesai diproses, menyebabkan penundaan pada jalur komunikasi.

**HPACK**

Algoritma kompresi header yang digunakan dalam protokol HTTP/2 untuk mengurangi ukuran header yang dikirim dalam setiap request, meningkatkan efisiensi transmisi data.

**Hypertext Transfer Protocol 1.1 (HTTP/1.1)**

Versi protokol HTTP yang banyak digunakan, dengan model koneksi satu request per koneksi yang rentan terhadap Head-of-Line Blocking. Digunakan oleh REST API dalam penelitian ini.

**Hypertext Transfer Protocol 2 (HTTP/2)**

Versi terbaru dari protokol HTTP yang mendukung multiplexing, kompresi header (HPACK), dan *streaming* bidirectional. Digunakan sebagai protokol transport oleh gRPC.

**Inferensi (*Inference*)**

Proses menggunakan model machine learning yang telah dilatih untuk membuat prediksi atau klasifikasi berdasarkan data input baru.

**Jetpack Compose**

Framework toolkit UI modern untuk pengembangan antarmuka pengguna Android yang menggunakan pendekatan deklaratif. Digunakan untuk membangun antarmuka aplikasi client dalam penelitian ini.

**JavaScript Object Notation (JSON)**

Format pertukaran data berbasis teks yang ringan dan mudah dibaca manusia, umum digunakan dalam REST API. Dibandingkan dengan Protobuf, JSON memiliki ukuran payload yang lebih besar.

**Kotlin**

Bahasa pemrograman modern yang berjalan di JVM (Java Virtual Machine). Digunakan sebagai bahasa utama pengembangan aplikasi Android client dalam penelitian ini.

**Kotlin Coroutines**

Fitur dalam bahasa Kotlin untuk menangani operasi asinkron secara lebih efisien menggunakan konsep coroutine yang ringan (lightweight thread). Digunakan untuk mengelola komunikasi jaringan di sisi client.

**Latensi (*Latency*)**

Waktu yang dibutuhkan data untuk berpindah dari pengirim ke penerima, atau waktu respons suatu sistem. Diukur dalam milidetik (ms) dan merupakan salah satu metrik kinerja utama dalam penelitian ini.

**LeakCanary**

Library deteksi memory leak untuk aplikasi Android yang secara otomatis mendeteksi dan melaporkan kebocoran memori selama pengujian. Digunakan untuk memantau penggunaan memori client dalam penelitian ini.

**Linux NetEm**

Modul kernel Linux yang memungkinkan emulasi kondisi jaringan seperti delay, packet loss, dan pembatasan bandwidth. Digunakan untuk mensimulasikan berbagai kondisi jaringan seluler dalam penelitian ini.

**Load Test**

Jenis pengujian kinerja yang mensimulasikan beban trafik normal atau ekspektasi harian untuk mengukur performa sistem dalam kondisi operasional biasa.

**Locust**

Alat pengujian kinerja berbasis Python yang memungkinkan simulasi banyak pengguna virtual (VU) secara bersamaan. Digunakan untuk melakukan pengujian beban pada server dalam penelitian ini.

**Long-Lived Connection**

Koneksi jaringan yang dipertahankan dalam jangka waktu lama selama sesi komunikasi. gRPC menggunakan koneksi persisten ini untuk mendukung *streaming*, namun rentan terhadap gangguan jaringan jangka panjang.

**Machine Learning**

Cabang kecerdasan buatan yang memungkinkan sistem belajar dari data dan membuat prediksi atau keputusan tanpa diprogram secara eksplisit. Digunakan untuk membangun model deteksi penyakit kulit dalam penelitian ini.

**Memory Leak**

Kondisi di mana program gagal melepaskan memori yang sudah tidak digunakan, menyebabkan konsumsi memori terus meningkat seiring waktu dan berpotensi menyebabkan crash.

**Metadata**

Data yang mendeskripsikan atau memberikan informasi tentang data lain. Dalam konteks penelitian ini, metadata berupa informasi seperti tipe citra dan checksum yang dikirim sebelum data citra utama.

**Multipart/Form-Data**

Format encoding yang digunakan dalam HTTP untuk mengirim data berupa file atau data biner melalui REST API. Memiliki overhead yang lebih besar dibandingkan format biner Protobuf.

**Multiplexing**

Kemampuan protokol HTTP/2 untuk mengirim dan menerima beberapa request dan response secara bersamaan dalam satu koneksi TCP tunggal, mencegah terjadinya Head-of-Line Blocking.

**Packet Loss**

Persentase paket data yang hilang atau gagal sampai ke tujuan selama transmisi jaringan. Packet loss yang tinggi dapat menyebabkan penurunan kinerja, terutama pada protokol yang menggunakan koneksi persisten seperti gRPC.

**Payload**

Data utama yang dibawa oleh suatu paket atau pesan dalam komunikasi jaringan, tidak termasuk header dan informasi kontrol lainnya.

**Perfetto**

Alat trace visualization untuk platform Android yang memungkinkan analisis mendalam terhadap kinerja CPU, memori, dan I/O aplikasi. Digunakan untuk profiling kinerja aplikasi client dalam penelitian ini.

**Pipeline**

Rangkaian proses atau tahapan yang diorganisir secara berurutan, di mana output dari satu tahap menjadi input bagi tahap berikutnya. Dalam penelitian ini merujuk pada proses konversi model AI.

**Protocol Buffers (Protobuf)**

Format serialisasi data biner yang dikembangkan oleh Google, lebih kompak dan lebih cepat diproses dibandingkan format teks seperti JSON. Digunakan oleh gRPC untuk mengemas dan mengirim data.

**Quality of Service (QoS)**

Kemampuan jaringan untuk menyediakan layanan dengan tingkat kualitas yang terjamin, mencakup parameter seperti bandwidth, latensi, dan packet loss.

**Recovery Time**

Waktu yang dibutuhkan sistem untuk pulih dan kembali ke kondisi normal setelah mengalami lonjakan beban atau gangguan. Digunakan sebagai metrik pada pengujian spike test.

**Representational State Transfer (REST API)**

Gaya arsitektur untuk membangun layanan web yang menggunakan HTTP sebagai protokol komunikasi dan JSON sebagai format data. REST bersifat stateless, di mana setiap request berdiri sendiri.

**Remote Procedure Call (RPC)**

Protokol komunikasi yang memungkinkan program memanggil prosedur atau fungsi yang berjalan di komputer lain melalui jaringan seolah-olah memanggil fungsi lokal.

**Serialisasi (*Serialization*)**

Proses mengubah objek atau struktur data dalam memori menjadi format yang dapat disimpan atau ditransmisikan, seperti byte array. Protobuf digunakan sebagai mekanisme serialisasi dalam gRPC.

**SHA-256 (*Checksum*)**

Algoritma hash kriptografi yang menghasilkan nilai hash sepanjang 256-bit dari suatu data. Digunakan dalam penelitian ini untuk memvalidasi integritas data citra yang ditransmisikan.

**Smoke Test**

Pengujian awal (sanity check) ringan untuk memverifikasi bahwa fungsi dasar sistem bekerja dengan benar di lingkungan produksi sebelum pengujian yang lebih komprehensif dilakukan.

**Soak Test**

Jenis pengujian kinerja yang menjalankan sistem dengan beban normal dalam jangka waktu panjang untuk mendeteksi masalah seperti memory leak, degradasi performa, atau kebocoran sumber daya.

**Spike Test**

Jenis pengujian kinerja yang mensimulasikan lonjakan beban yang tiba-tiba dan signifikan untuk mengevaluasi kemampuan sistem dalam merespons dan memulihkan diri dari peningkatan trafik mendadak.

**Stateless**

Sifat protokol atau sistem di mana setiap request diproses secara independen tanpa menyimpan informasi dari interaksi sebelumnya. REST API bersifat stateless, membuatnya lebih toleran terhadap gangguan jaringan.

**Steady State**

Kondisi sistem yang stabil setelah melewati fase inisialisasi dan pemanasan (warm-up), di mana sistem beroperasi pada kondisi keseimbangan yang konsisten.

**Streaming**

Teknik pengiriman data secara berkelanjutan dan bertahap, bukan sekaligus dalam satu paket besar. gRPC mendukung *streaming* di sisi client, server, maupun bidirectional.

**Stress Test**

Jenis pengujian kinerja yang mendorong sistem melampaui batas kapasitas normalnya untuk menemukan titik jenuh (bottleneck) dan kapasitas maksimal sistem.

**Stub**

Kode klien yang dihasilkan secara otomatis oleh compiler gRPC (protoc) berdasarkan berkas *.proto*, yang menyediakan antarmuka untuk memanggil layanan RPC seolah-olah memanggil fungsi lokal.

**Success Rate**

Persentase request yang berhasil diselesaikan tanpa error dari total request yang dikirim. Merupakan metrik reliabilitas utama dalam pengujian kinerja penelitian ini.

**Traffic Control (TC)**

Utilitas Linux yang digunakan untuk mensimulasikan kondisi jaringan tertentu seperti delay, packet loss, dan pembatasan bandwidth pada antarmuka jaringan.

**Throughput**

Jumlah data atau permintaan yang berhasil diproses oleh sistem dalam satuan waktu tertentu. Diukur dalam RPS (Requests Per Second) dan merupakan indikator kapasitas pemrosesan server.

**Unary Response**

Pola respons dalam gRPC di mana server mengirim satu respons tunggal setelah menerima seluruh request. Dalam penelitian ini, server mengirim satu respons hasil analisis setelah menerima seluruh *stream* data citra.

**Universally Unique Identifier (UUID)**

Identifikasi unik berukuran 128-bit yang digunakan untuk mengidentifikasi objek secara unik dalam sistem terdistribusi. Digunakan untuk mengidentifikasi setiap sesi analisis dalam penelitian ini.

**Virtual Users (VUs)**

Simulasi pengguna virtual yang digunakan dalam pengujian beban untuk mensimulasikan banyak pengguna mengakses sistem secara bersamaan. Digunakan oleh Locust dalam pengujian kinerja penelitian ini.

**Warm-Up**

Fase awal di mana sistem melakukan inisialisasi koneksi, alokasi resource, dan pemanasan komponen sebelum mencapai kondisi steady state. Fase ini biasanya ditandai dengan variasi latensi yang tinggi.

Lampiran 1

Data Agregat Hasil dari Pengujian Kinerja

| NO | Protokol | Kondisi Jaringan | Skenario Pengujian | Rata-rata Peak Throughput (RPS) | Rata-rata Waktu Respon (ms) | Rata-rata P99 (ms) | Total Permintaan | Tingkat Keberhasilan (%) |
|---|---|---|---|---|---|---|---|---|
| 1 | grpc | 3G | Load | 74 | 40486 | 42585 | 52 | 100.00% |
| 2 | grpc | 3G | Smoke | 0 | 6005 | 600 | 5 | 100.00% |
| 3 | grpc | 3G | Soak | 11 | 96691 | 111342 | 165 | 82.42% |
| 4 | grpc | 3G | Spike | 20 | 23258 | 29361 | 250 | 100.00% |
| 5 | grpc | 3G | Stress | 10 | 65709 | 86016 | 200 | 100.00% |
| 6 | grpc | 4G | Load | 84 | 6316 | 10231 | 303 | 100.00% |
| 7 | grpc | 4G | Smoke | 1 | 812 | 1068 | 32 | 100.00% |
| 8 | grpc | 4G | Soak | 114 | 10721 | 16727 | 1221 | 100.00% |
| 9 | grpc | 4G | Spike | 208 | 9515 | 16501 | 338 | 100.00% |
| 10 | grpc | 4G | Stress | 148 | 13966 | 25634 | 678 | 100.00% |
| 11 | grpc | NORMAL | Load | 348 | 12 | 192 | 6593 | 100.00% |
| 12 | grpc | NORMAL | Smoke | 44 | 93 | 103 | 104 | 100.00% |
| 13 | grpc | NORMAL | Soak | 382 | 216 | 301 | 32830 | 100.00% |
| 14 | grpc | NORMAL | Spike | 492 | 668 | 1358 | 5747 | 100.00% |
| 15 | grpc | NORMAL | Stress | 452 | 443 | 864 | 15665 | 100.00% |
| 16 | grpc | POOR | Load | 76 | 10733 | 17305 | 188 | 100.00% |
| 17 | grpc | POOR | Smoke | 4 | 1586 | 1888 | 18 | 100.00% |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 18 | grpc | POOR | Soak | 102 | 17991 | 27982 | 730 | 100.00% |
| 19 | grpc | POOR | Spike | 208 | 14528 | 18878 | 283 | 100.00% |
| 20 | grpc | POOR | Stress | 146 | 2402 | 39649 | 435 | 100.00% |
| 21 | grpc | WORST | Load | 7 | 36073 | 40631 | 59 | 100.00% |
| 22 | grpc | WORST | Smoke | 0 | 5565 | 566 | 6 | 100.00% |
| 23 | grpc | WORST | Soak | 112 | 76735 | 103556 | 190 | 93.71% |
| 24 | grpc | WORST | Spike | 20 | 23145 | 3018 | 250 | 100.00% |
| 25 | grpc | WORST | Stress | 102 | 65209 | 86086 | 201 | 100.00% |
| 26 | rest | 3G | Load | 0 | 34096 | 34931 | 6 | 100.00% |
| 27 | rest | 3G | Soak | 12 | 75677 | 10344 | 97 | 89.79% |
| 28 | rest | 3G | Stress | 0 | 61225 | 64526 | 8 | 100.00% |
| 29 | rest | 4G | Load | 78 | 2154 | 4632 | 836 | 100.00% |
| 30 | rest | 4G | Smoke | 1 | 758 | 1056 | 28 | 100.00% |
| 31 | rest | 4G | Soak | 86 | 3399 | 7001 | 3665 | 100.00% |
| 32 | rest | 4G | Spike | 78 | 5506 | 12426 | 526 | 100.00% |
| 33 | rest | 4G | Stress | 94 | 5093 | 1170 | 1646 | 100.00% |
| 34 | rest | NORMAL | Load | 35 | 118 | 185 | 6639 | 100.00% |
| 35 | rest | NORMAL | Smoke | 48 | 93 | 102 | 106 | 100.00% |
| 36 | rest | NORMAL | Soak | 382 | 225 | 32 | 32134 | 100.00% |
| 37 | rest | NORMAL | Spike | 374 | 683 | 112 | 5465 | 100.00% |
| 38 | rest | NORMAL | Stress | 38 | 451 | 74 | 15264 | 100.00% |
| 39 | rest | POOR | Load | 88 | 1329 | 2455 | 1280 | 100.00% |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 40 | rest | POOR | Smoke | 2 | 1188 | 1564 | 17 | 100.00% |
| 41 | rest | POOR | Soak | 134 | 1411 | 2492 | 8316 | 100.00% |
| 42 | rest | POOR | Spike | 35 | 1286 | 2442 | 3519 | 100.00% |
| 43 | rest | POOR | Stress | 296 | 1383 | 2524 | 7694 | 100.00% |
| 44 | rest | WORST | Load | 42 | 5211 | 9138 | 323 | 100.00% |
| 45 | rest | WORST | Soak | 56 | 5885 | 10493 | 2116 | 100.00% |
| 46 | rest | WORST | Spike | 134 | 4841 | 9072 | 877 | 100.00% |
| 47 | rest | WORST | Stress | 116 | 5596 | 10273 | 1942 | 100.00% |

Lampiran 2

Perbandingan Kinerja Utama gRPC dan REST pada Aplikasi Android

| **Parameter Kinerja** | **gRPC** | **REST** | **Satuan** |
|---|---|---|---|
| Total CPU Workload (sched) | 25842.84 | 37463.39 | ms |
| Peak Memory Usage | 257.53 | 266.73 | MB |
| RAM Standard Deviation | 82.46 | 84.33 | MB |
| CPU Peak Slice | 214.79 | 283.18 | ms |
| CPU Average Slice | 0.282 | 0.374 | ms |

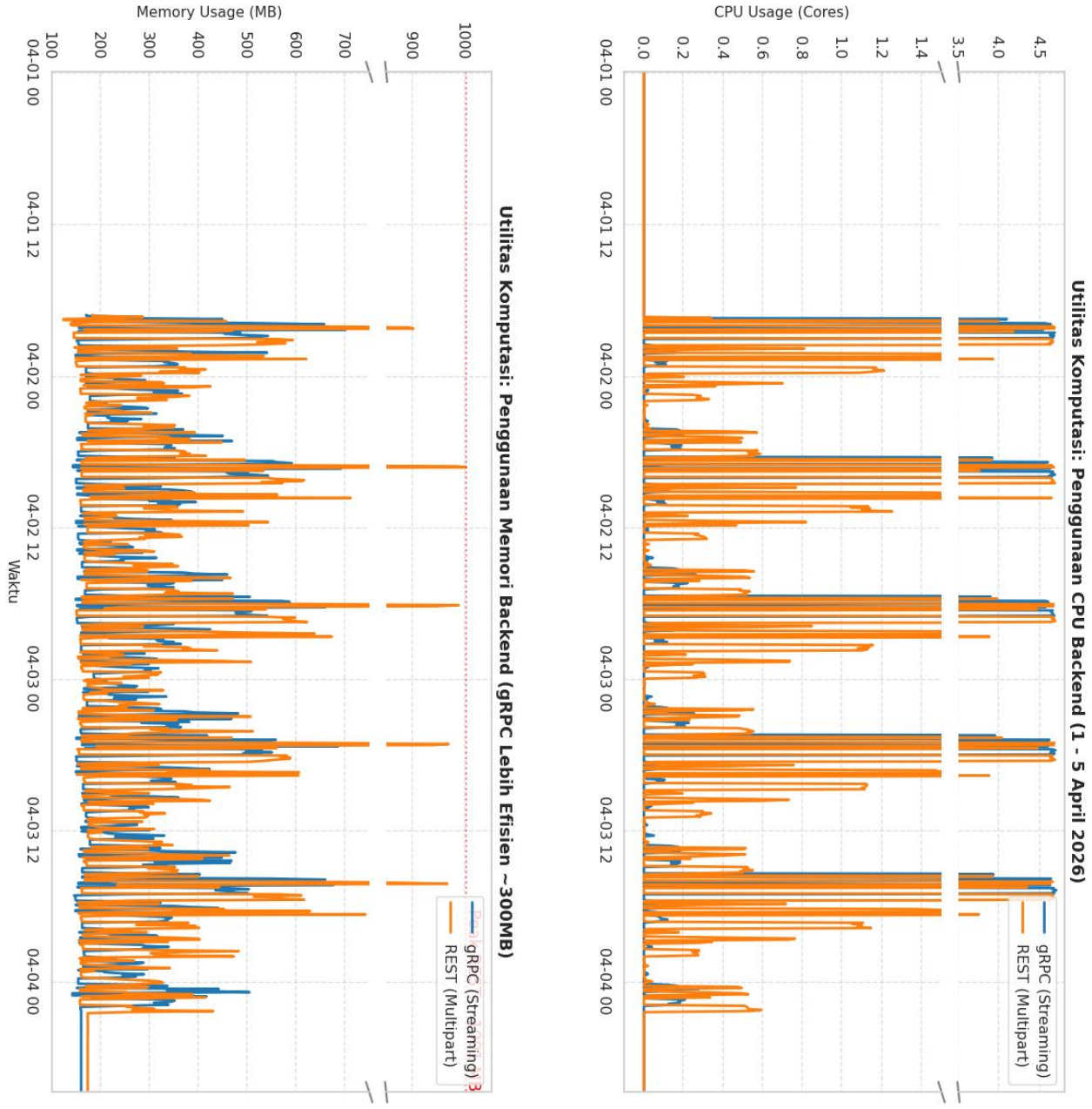
Lampiran 3

Rincian Distribusi Waktu Eksekusi Aplikasi Android

| **Komponen / Lapisan** | **gRPC (ms)** | **REST (ms)** |
|---|---|---|
| Other App Logic | 103710.68 | 180467.91 |
| Protocol Internal | 33.82 | 5.26 |
| Serialization / Parsing | 1.08 | 2.15 |
| Data Repository | 0.15 | 0.17 |
| Domain Logic (UseCase) | 0.06 | 0.08 |

Lampiran 4

Grafik perbandingan deret waktu penggunaan *resource* (CPU dan memori) antara REST dan gRPC pada *backend*



Lampiran 5

Grafik perbandingan efisiensi jaringan berdasarkan ukuran data transmisi jaringan pada request dan response antara REST dan gRPC