BAB I

PENDAHULUAN

1.1 Latar Belakang

Pendapatan Asli Daerah (PAD), merupakan sumber pendapatan yang berasal dari sektor lokal seperti pajak daerah dan retribusi, termasuk parkir, yang menjadi salah satu indikator penting dalam pembangunan dan kesejahteraan kota. Namun, pengelolaan sistem parkir yang tidak efektif dapat menyebabkan masalah kebocoran terhadap PAD, yang secara langsung mengurangi kontribusi sektor parkir terhadap pendapatan daerah. Berdasarkan laporan Anggaran Pendapatan dan Belanja Daerah (APBD) Kota Makassar tahun 2021 realisasi PAD hanya 67.62% sebanyak Rp. 1.140,33 Miliar dari target Rp. 1.686,39 Miliar. Kemudian pada tahun 2022, realisasi PAD mencapai Rp.1.410,81 Miliar dari target Rp.2.014,71 Miliar, yang berarti hanya sekitar 70,03% (Laporan APBD Kota Makassar, 2021 & 2022). Kontribusi dari sektor retribusi parkir terhadap PAD Kota Makassar pada tahun 2021 bahkan tercatat hanya mencapai 0,96%, yaitu sekitar Rp.10.92 Miliar dari target Rp. 80.00 Miliar. Sedangkan pada tahun 2022 retribusi parkir terhadap PAD tercatat 1,06%, yaitu sekitar Rp.14,98 miliar dari target yang seharusnya mencapai Rp.100 Miliar (Laporan Keuangan Pemerintah Kota Makassar, 2021 & 2022). Ini menunjukkan bahwa meskipun sektor parkir memiliki potensi yang signifikan, pemanfaatannya untuk meningkatkan PAD Kota Makassar masih belum optimal.

Untuk menghadapi tantangan besar ini, inovasi *Smart Parking* yang menggunakan kecerdasan buatan (AI) menjadi solusi modern yang efisien dan berkelanjutan. Sistem ini memanfaatkan teknologi visi komputer yang terintegrasi dengan kamera di berbagai titik parkir untuk mengawasi area parkir secara *real-time*. Sistem ini mampu merekam kendaraan yang masuk dan keluar serta secara otomatis mengidentifikasi nomor pelat dan jenis kendaraan. Dengan penerapan teknologi ini, pengelolaan parkir menjadi lebih transparan, akurat, dan efisien. Meskipun inti utama dari sistem ini adalah visi komputer, pada dasarnya sistem ini terdiri dari beberapa layanan yang saling terintegrasi dan bergantung satu sama lain. Maka dari itu agar sistem yang dibangun mudah dikelola dan lebih efisien, sistem ini perlu dirancang menggunakan arsitektur *microservices* yang membagi aplikasi menjadi beberapa layanan tersendiri (Hassan, 2024). *Microservices* menawarkan banyak manfaat seperti skalabilitas, agilitas, dan modularitas.

Walaupun memiliki banyak manfaat, arsitektur *microservices* masih memiliki beberapa tantangan salah satunya adalah komunikasi antar layanannya. Pada arsitektur *microservices* beberapa layanan bekerja secara bersamaan dari lokasi yang berbeda. Oleh karena itu, pemilihan protokol komunikasi sangat berpengaruh pada kinerja sistem baik pada pengiriman data, memproses kumpulan *dataset* besar, atau pengelolaan transaksi antar layanan. Pemilihan protokol komunikasi pada sistem akan berdampak secara langsung pada kecepatan latensi, penggunaan *bandwidth*, keamanan, dan skalabilitasnya (Thumburu, 2022). Dalam konteks sistem *Smart Parking*, pemilihan protokol komunikasi yang tepat adalah aspek yang sangat vital. Layanan-layanan seperti *AI service* untuk analisis data kendaraan, *Service Main* untuk pengelolaan transaksi dan

database, dan *dashboard* untuk pemantauan pendapatan parkir, semuanya bergantung pada komunikasi yang efisien antar layanan. Kesalahan dalam menentukan protokol yang optimal tidak hanya dapat menyebabkan keterlambatan pencatatan data kendaraan, tetapi juga berpotensi mengganggu keseluruhan alur kerja sistem.

Protokol komunikasi yang paling banyak dipakai dalam sistem berbasis *microservices* adalah REST API dan gRPC karena keunggulannya secara efisiensi dan fleksibilitasnya. REST API adalah metode komunikasi layanan yang paling umum digunakan. REST API berbasis HTTP/1.1 dan menggunakan format JSON yang sederhana dan umum, REST API lebih fleksibel dalam interoperabilitas dan *debugging* (Hamo & Saberian, 2023). Sebaliknya gRPC adalah metode komunikasi layanan berbasis HTTP/2 dan menggunakan *Protocol Buffers* untuk serialisasi data terstruktur. *gRPC* menawarkan performa yang lebih efisien dalam hal latensi dan *throughput* (Hassan, 2024). Urgensi perbandingan REST API dan gRPC semakin penting pada arsitektur *microservices* karena setiap layanan perlu saling berkomunikasi secara intensif. Dalam kondisi seperti ini, perbedaan kecil pada *overhead* komunikasi dapat terakumulasi menjadi dampak yang besar terhadap respons sistem, terutama ketika jumlah permintaan meningkat. Maka dari itu dipilihlah REST API dan gRPC sebagai layanan komunikasi populer yang akan dibandingkan.

Dalam sistem *Smart Parking* berbasis *microservices*, setiap permintaan dari layanan *service* akan terjadi berulang kali untuk setiap kendaraan yang masuk dan keluar area parkir. Perbedaan kecil pada *overhead* komunikasi antara REST API dan gRPC dapat terakumulasi menjadi perbedaan signifikan pada mertrik yang diuji. Kondisi ini sangat relevan bagi pemerintah daerah karena keterlambatan atau ketidaktepatan pencatatan transaksi parkir berpotensi menimbulkan kehilangan data pendapatan dan kembali membuka celah kebocoran PAD, sehingga diperlukan analisis terukur untuk menentukan protokol komunikasi yang paling sesuai.

Penelitian ini bertujuan untuk menganalisis dan membandingkan performa kedua protokol komunikasi yaitu REST API dan gRPC dalam sistem *Smart Parking*. Aspek yang akan diuji adalah *response time*, *throughput*, *resource utilization*, *network data rate*, dan *error rate*. Diharapkan hasil dari penelitian ini akan memberikan rekomendasi berbasis data, tentang protokol komunikasi yang paling sesuai dalam sistem *Smart Parking* untuk meningkatkan transparansi, efisiensi, dan kontribusi dari sektor parkir terhadap PAD Kota Makassar.

1.2 Landasan Teori

1.2.1 Sistem *Smart Parking*

Sistem Smart Parking merupakan komponen krusial dalam ekosistem Smart City yang dirancang sebagai solusi inovatif untuk mengatasi inefisiensi parkir di wilayah perkotaan (Fahim et al., 2021; Channamallu et al., 2023). Meningkatnya volume kendaraan di kota-kota besar telah menjadikan pencarian tempat parkir sebagai tantangan harian yang signifikan, yang tidak hanya menyebabkan pemborosan waktu dan bahan bakar, tetapi juga berkontribusi pada peningkatan kemacetan lalu lintas dan polusi udara (Biyik et al., 2021; Fahim et al., 2021). SPS hadir untuk mengatasi masalah ini dengan menyediakan informasi ketersediaan parkir secara *real-time*, mengoptimalkan penggunaan ruang

parkir, serta menawarkan metode pembayaran yang lebih praktis dan efisien (Channamallu et al., 2023).

Tujuan utama dari implementasi SPS adalah untuk menciptakan sistem manajemen parkir yang lebih cepat, mudah, dan efisien (Biyik et al., 2021). Manfaat yang ditawarkan sangat beragam, mulai dari pengurangan kemacetan lalu lintas dan emisi karbon hingga peningkatan pengalaman pengguna (Channamallu et al., 2023). Dengan adanya informasi yang akurat mengenai lokasi parkir yang tersedia, pengemudi dapat langsung menuju *slot* kosong tanpa harus berputar-putar, sehingga secara efektif mengurangi volume kendaraan di jalan dan konsumsi bahan bakar yang tidak perlu (Biyik et al., 2021). Selain itu, bagi pengelola parkir dan pemerintah kota, SPS memungkinkan pemanfaatan ruang yang optimal, yang berpotensi meningkatkan pendapatan serta menyediakan data analitik untuk pengelolaan parkir yang lebih baik di masa depan (Channamallu et al., 2023; Biyik et al., 2021).

Keberhasilan SPS sangat bergantung pada integrasi berbagai teknologi. Pendekatan yang paling dominan adalah pemanfaatan Internet of Things (IoT), di mana berbagai sensor dan perangkat saling terhubung untuk berbagi data secara *real-time* (Fahim et al., 2021). Dalam kerangka IoT, teknologi seperti Wireless Sensor Network (WSN) menjadi sangat populer karena instalasinya yang fleksibel dan hemat biaya, menggunakan sensor seperti ultrasonik, inframerah, atau magnetometer untuk mendeteksi keberadaan kendaraan di setiap *slot* parkir (Channamallu et al., 2023). Selain WSN, sistem berbasis *Computer Vision* atau pemrosesan citra juga banyak digunakan. Teknologi ini memanfaatkan kamera untuk mendeteksi kendaraan serta melakukan pengenalan pelat nomor (*license plate recognition*) yang berguna untuk pembayaran otomatis dan peningkatan keamanan (Channamallu et al., 2023; Fahim et al., 2021).

1.2.2 Visi Komputer

Visi komputer adalah cabang dari kecerdasan buatan yang memungkinkan komputer untuk memahami dan menginterpretasi informasi visual dari dunia nyata (Matsuzaka & Yashiro, 2023). Teknologi ini meniru cara kerja sistem penglihatan manusia dengan menggunakan algoritma dan model matematis untuk mengekstrak informasi bermakna dari data visual. Secara umum, *pipeline computer vision* meliputi tahapan akuisisi citra, *preprocessing* untuk meningkatkan kualitas gambar, ekstraksi fitur, dan tahap analisis/interpretasi untuk menghasilkan *output* yang diinginkan (Chinthaginjala et al., 2025). Pada tahap *preprocessing*, operasi umum yang dilakukan antara lain *resizing* (mengubah ukuran dimensi gambar), *normalization* (menstandarkan nilai piksel), dan *noise reduction* (mengurangi *noise*/citra buram) guna menstandarkan data dan meningkatkan kualitas citra sebelum analisis lebih lanjut.

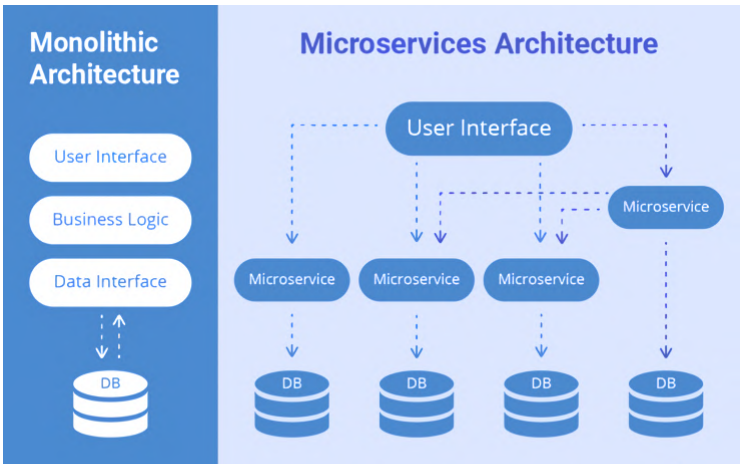
> ***Object detection.*** *Object detection* adalah tugas *computer vision* yang bertujuan untuk mengidentifikasi apa objek yang ada sekaligus di mana letaknya dalam sebuah gambar atau video, biasanya dengan menghasilkan *bounding box* untuk setiap objek. Berbeda dengan *image classification* yang hanya mengklasifikasikan satu label untuk seluruh gambar, *object detection* dapat mendeteksi *multiple* objek beserta lokasinya secara simultan. Algoritma *object*

detection berbasis *deep learning* telah berkembang pesat dalam satu dekade terakhir. Pendekatan awal seperti R-CNN (*Region-Based CNN*) melakukan deteksi objek melalui dua tahap (*region proposals* lalu klasifikasi) sehingga cenderung lambat. Peningkatan efisiensi dicapai oleh *Fast R-CNN* dan *Faster R-CNN*, yang memperkenalkan *training end-to-end* dengan *multi-task loss* serta *Region Proposal Network* untuk mempercepat tahap *proposal region*. Selanjutnya, muncul detektor satu tahap seperti YOLO (*You Only Look Once*) yang langsung memprediksi *bounding box* dan kelas objek dalam satu kali proses inferensi, sehingga mampu melakukan deteksi *real-time* dengan akurasi tinggi (Li et al., 2022; Kang et al., 2025).

Dalam konteks sistem Smart Parking, algoritma *object detection* digunakan untuk mendeteksi keberadaan kendaraan serta mengidentifikasi area pelat nomor kendaraan dengan akurasi tinggi, yang esensial untuk pemrosesan *video stream* CCTV secara *real-time* (Elfaki et al., 2023). Deteksi lokasi pelat nomor ini menjadi langkah awal sebelum tahap pengenalan karakter dilakukan.

***Optical Character Recognition* (OCR).** OCR adalah teknologi yang mengonversi teks pada citra menjadi teks digital yang dapat diedit atau dicari (Su et al., 2024). Dalam sistem Smart Parking, OCR digunakan untuk membaca karakter pada pelat nomor kendaraan secara otomatis setelah *region* pelat nomor tersebut berhasil dideteksi oleh sistem *vision* (Elfaki et al., 2023). Proses OCR umumnya melibatkan beberapa tahap: pertama, *preprocessing* khusus seperti *binarization* (mengubah citra ke bentuk biner hitam-putih) untuk menonjolkan teks, *skew correction* untuk meluruskan kemiringan citra pelat yang miring, serta *noise removal* untuk menghilangkan gangguan pada citra sebelum pengenalan (Akella, 2025).

1.2.3 *Arsitektur Microservices*



Gambar 1. Perbedaan arsitektur monolitik dan *microservices*.

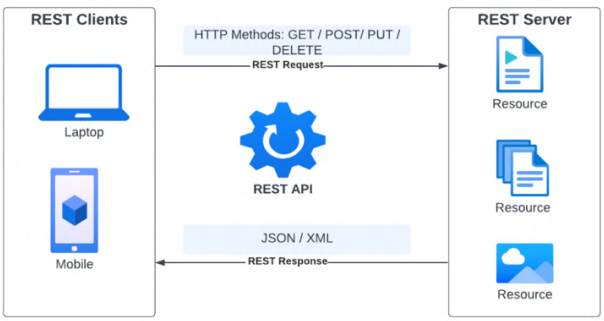
Microservices adalah pendekatan arsitektur perangkat lunak yang membangun aplikasi sebagai kumpulan layanan kecil, independen, dan dapat di-*deploy* secara terpisah (Cerny & Taibi, 2023). Setiap *service* berfokus pada satu kemampuan bisnis dan berkomunikasi melalui antarmuka pemrograman aplikasi (API) yang sudah ditentukan. Berbeda dengan arsitektur monolitik yang menyatukan seluruh komponen dalam satu sistem besar, pendekatan *microservices* membuat setiap bagian aplikasi lebih mandiri dan fleksibel.

Karakteristik utama *microservices* mencakup *decentralized governance* yang memungkinkan setiap tim memilih teknologi yang paling sesuai, *single responsibility principle* yang memastikan setiap *service* memiliki fokus yang jelas, *independent deployment* yang memungkinkan *update* tanpa memengaruhi *service* lain (Susnjara & Smalley, 2025), dan *technology diversity* yang memberikan fleksibilitas dalam pemilihan *stack* teknologi (Auer et al., 2021).

Dibandingkan dengan sistem monolitik, *microservices* menawarkan banyak keuntungan. Keunggulan *microservices* meliputi *scalability* yang lebih baik karena setiap layanan bisa ditingkatkan kapasitasnya secara terpisah, *fault tolerance* yang lebih tinggi karena kegagalan satu *service* tidak langsung memengaruhi seluruh sistem. Selain itu, arsitektur ini memungkinkan penggunaan teknologi yang paling sesuai untuk tiap fungsi sistem serta mendukung kerja tim secara paralel karena setiap tim bisa fokus pada layanan masing-masing (Auer et al., 2021).

Namun, *microservices* juga memiliki tantangan tersendiri. Komunikasi anta rlayanan yang berlangsung melalui jaringan dapat menambah kompleksitas sistem. Pengelolaan data yang tersebar di banyak layanan juga memerlukan strategi khusus agar tetap konsisten. Selain itu, dibutuhkan mekanisme untuk menemukan dan memantau status setiap layanan serta pengaturan komunikasi yang efisien agar tidak menimbulkan *overhead* pada komunikasi yang bisa memengaruhi kinerja sistem secara keseluruhan (Microsoft, 2023).

1.2.4 REST API



Gambar 2. Arsitektur REST API.

REST (*Representational State Transfer*) adalah *software architectural style* untuk sistem terdistribusi yang pertama kali diperkenalkan oleh Roy Fielding pada tahun

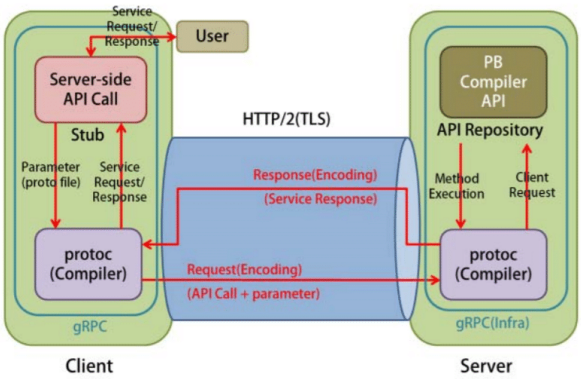
2000. REST mendefinisikan sekumpulan *constraint* yang harus dipenuhi untuk menciptakan *web service* yang *scalable*, *reliable*, dan *maintainable* (Coblenz et al., 2023; Ala-Laurinaho et al., 2022).

Prinsip-prinsip REST meliputi: *stateless*, di mana setiap *request* harus mengandung semua informasi yang diperlukan untuk memproses *request* tersebut; arsitektur klien-server yang memisahkan antarmuka pengguna dari penyimpanan data; *cacheable*, di mana *response* harus dapat di-*cache* untuk meningkatkan performa; *uniform interface* yang menyediakan antarmuka standar antara klien dan server; dan *layered system* yang memungkinkan arsitektur berlapis (Ala-Laurinaho et al., 2022).

HTTP *methods* dalam REST memiliki semantik yang jelas: GET untuk mengambil data, POST untuk membuat data baru, PUT untuk memperbarui data yang ada, dan DELETE untuk menghapus data (Ala-Laurinaho et al., 2022). Setiap *method* memiliki karakteristik *idempotency* yang berbeda, di mana GET, PUT, dan DELETE bersifat *idempotent*, sedangkan POST tidak (Microsoft, 2025).

REST API design yang baik mengikuti konvensi penamaan *resource* yang intuitif (menggunakan *noun* untuk *resource* dan *verb* untuk *action*), menerapkan HTTP *status codes* yang semestinya (misalnya: 200 untuk *success*, 404 untuk *not found*, 500 untuk *server error*) (Ala-Laurinaho et al., 2022), serta menggunakan JSON sebagai format pertukaran data yang ringan dan dapat dibaca manusia (Coblenz et al., 2023). Keunggulan REST antara lain meliputi kesederhanaan dalam implementasi dan *debugging*, interoperabilitas yang tinggi karena berbasis standar HTTP, dan dukungan *caching* yang sangat baik melalui HTTP *headers* (Khan et al., 2025), serta dukungan ekosistem luas di berbagai bahasa pemrograman (Coblenz et al., 2023). Namun, REST juga memiliki keterbatasan seperti *over-fetching* atau *under-fetching* data, *multiple round trips* untuk *query* yang kompleks (Khan et al., 2025), dan limitasi HTTP/1.1 dalam hal performa untuk skenario *throughput* yang tinggi (Niswar et al., 2024).

1.2.5 gRPC



Gambar 3. Arsitektur gRPC.

gRPC (*Google Remote Procedure Call*) adalah *high-performance*, *open-source* RPC *framework* yang dikembangkan oleh Google (Obuse et al., 2020). *gRPC* menggunakan

Protocol Buffers sebagai *Interface Definition Language* (IDL) dan HTTP/2 sebagai *underlying transport protocol*, memberikan keunggulan signifikan dalam hal performa dan efisiensi dibandingkan API berbasis HTTP tradisional (Ain et al., 2025; Johansson, 2023).

Komponen utama *gRPC* meliputi *Protocol Buffers* yang menyediakan format untuk mendeskripsikan dan mengirim data yang bisa dipakai di banyak bahasa pemrograman dan lebih efisien daripada JSON (Shatnawi et al., 2025), *service definition files* (.proto) yang berisi definisi fungsi-fungsi layanan dan bentuk data yang dipakai, serta *code generation* yang secara otomatis menghasilkan kode *client* dan *server* dalam berbagai bahasa pemrograman berdasarkan file .proto tersebut (Obuse et al., 2020).

Keunggulan *gRPC* mencakup performa yang sangat baik dengan *throughput* hingga 10× lebih tinggi dibandingkan REST dalam beberapa skenario, tipe data yang kuat dan jelas karena memakai *Protocol Buffers* sehingga mengurangi kesalahan saat program berjalan; dukungan komunikasi dua arah dan *streaming* untuk kebutuhan *real-time*; pembuatan kode otomatis yang mengurangi penulisan kode berulang; dan fitur bawaan untuk autentikasi dan *load balancing* (Golang gRPC, 2024).

Protocol Buffers sebagai mekanisme pengiriman data di *gRPC* memakai format biner, sehingga ukuran data yang dikirim bisa 3–10 kali lebih kecil dibandingkan JSON (Ain et al., 2025). Dukungan terhadap perubahan skema membuat sistem tetap bisa bekerja meskipun terjadi perubahan struktur data, baik ke belakang (*backward*) maupun ke depan (*forward*) (Obuse et al., 2020). Selain itu, sifatnya yang tidak bergantung pada satu bahasa pemrograman tertentu membuatnya mudah digunakan pada berbagai teknologi yang berbeda (Shatnawi et al., 2025).

Dari sisi HTTP/2, *gRPC* memanfaatkan fitur seperti *multiplexing* yang memungkinkan banyak permintaan berjalan bersamaan dalam satu koneksi, kompresi *header* yang mengurangi ukuran data tambahan, kemampuan server *push* (Meireles et al., 2022), dan penggunaan format biner yang lebih efisien dibandingkan HTTP/1.1 yang berbasis teks (Golang gRPC, 2024). Namun, *gRPC* juga memiliki beberapa keterbatasan, seperti dukungan *browser* yang masih terbatas karena ketergantungan pada HTTP/2 dan format biner, proses *debugging* yang lebih sulit karena data tidak mudah dibaca manusia, kurva belajar yang lebih curam dibandingkan REST, dan ekosistem yang meskipun berkembang pesat, masih belum seluas ekosistem REST (Johansson, 2023).

1.2.6 *Technology Stack*

Python. *Python* adalah bahasa pemrograman *high-level* yang dijalankan secara langsung tanpa perlu dikompilasi terlebih dahulu (*interpreted*). *Python* dikenal karena sintaksnya yang bersih dan mudah dibaca, sehingga mirip seperti menulis bahasa sehari-hari. Sejak dirilis oleh Guido van Rossum pada 1991, *Python* terus berkembang menjadi salah satu bahasa paling populer, terutama dalam *data science* dan *artificial intelligence*. Karakteristik *Python* yang membuatnya ideal untuk pengembangan antara lain penentuan tipe data yang fleksibel tanpa harus didefinisikan di awal (*dynamic typing*, yang memudahkan proses percobaan dan *prototyping*), pustaka standar yang luas dengan banyak

module bawaan yang siap digunakan, serta dukungan komunitas yang kuat dengan ekosistem paket yang sangat kaya. Untuk keperluan AI dan pembelajaran mesin, *Python* menyediakan pustaka seperti *TensorFlow*, *PyTorch*, *OpenCV*, dan *scikit-learn* yang banyak dipakai dalam pengembangan aplikasi pengenalan gambar dan video (*computer vision*) (Raschka et al., 2020).

Django Framework. *Django* adalah *high-level Python web framework* yang mengikuti prinsip "*batteries included*", yaitu sudah menyediakan banyak fitur penting yang dibutuhkan untuk pengembangan aplikasi web dengan cepat tanpa harus membangun semuanya dari nol. *Django* mengimplementasikan arsitektur *Model–View–Template* (MVT), yang membagi aplikasi menjadi tiga bagian, yaitu pengelolaan data (*model*), logika bisnis atau aturan proses (*view*), dan tampilan antarmuka pengguna (*template*) (Chen et al., 2020).

Komponen kunci *Django* meliputi *Object-Relational Mapping* (ORM), yaitu mekanisme yang memungkinkan pengembang mengelola dan mengakses database menggunakan kode *Python* tanpa perlu menulis perintah SQL secara langsung, antarmuka admin yang dibuat otomatis untuk pengelolaan konten dan data, serta sistem pengaturan alamat URL (*URL routing*) yang fleksibel untuk mengarahkan permintaan pengguna ke fungsi yang sesuai (Chen et al., 2020).

PostgreSQL. *PostgreSQL* adalah sistem manajemen basis data relasional (*Relational Database Management System*/RDBMS) *open-source* yang dikenal karena *reliability* (keandalan), kekayaan fitur, dan performanya. *PostgreSQL* mengelola data dalam bentuk tabel-tabel yang saling berelasi dan mendukung tipe data lanjut seperti JSON/JSONB, *array*, dan tipe data khusus (*custom types*). Selain itu, *PostgreSQL* menyediakan berbagai cara pengindeksan (seperti *B-Tree*, *Hash*, *GiST*, *GIN*, dan lain-lain) untuk mempercepat proses pencarian data. Fitur-fitur ini termasuk kepatuhan penuh terhadap sifat-sifat ACID (*Atomicity, Consistency, Isolation, Durability*) untuk menjamin transaksi data tersimpan dengan benar dan konsisten, optimisasi *query* tingkat lanjut untuk meningkatkan kinerja, kemampuan untuk diskalakan baik secara horizontal maupun vertikal, serta mekanisme keamanan yang kuat. Integrasi dengan *Django* melalui *adapter psycopg2* memungkinkan *Django* menggunakan fitur-fitur khusus *PostgreSQL* secara langsung dan lebih optimal (Gupta, 2025).

React. *React* adalah pustaka *JavaScript* untuk membangun antarmuka pengguna (UI) yang dikembangkan oleh *Facebook*. *React* menggunakan arsitektur berbasis komponen yang memungkinkan pengembangan UI modular dengan komponen yang dapat digunakan ulang. Konsep *Virtual DOM* pada *React* memungkinkan *rendering* yang efisien dengan meminimalkan manipulasi DOM nyata, sehingga meningkatkan performa aplikasi (Komperla et al., 2022). *React Hooks* seperti *useState*, *useEffect*, dan *useContext* memungkinkan komponen fungsional mengelola *state* dan *lifecycle* aplikasi secara lebih mudah. Pengelolaan *state* yang kompleks juga dapat dilakukan menggunakan *hook*

bawaan *React* atau pustaka eksternal seperti *Redux* atau *Zustand* (Patel & Bhattacharjee, 2025).

TypeScript. *TypeScript* adalah perluasan dari *JavaScript* yang menambahkan fitur pemeriksaan tipe secara statis (Emmanni, 2021). Kode *TypeScript* dikompilasi menjadi *JavaScript* biasa sehingga dapat berjalan di berbagai lingkungan yang mendukung *JavaScript*. Dengan pemeriksaan tipe yang ketat, *TypeScript* membantu meningkatkan kualitas kode dan produktivitas developer dengan mendeteksi *error* pada tahap *compile time* sebelum program dijalankan. Ketika digunakan bersama *React*, *TypeScript* memberikan pengalaman pengembangan yang lebih baik melalui saran otomatis dari IDE dan pesan *error* secara *real-time*. *Interface* dan tipe data *TypeScript* memungkinkan developer membuat kontrak yang jelas tentang bentuk data antar komponen, sehingga komunikasi antar komponen *React* menjadi lebih aman dan dokumentasi kode lebih mudah dipahami.

1.2.7 Apache JMeter

Apache JMeter adalah aplikasi Java *open-source* yang digunakan untuk menguji beban (*load testing*) dan mengukur kinerja suatu sistem (Czuper, 2022). *JMeter* dapat mensimulasikan banyak pengguna atau permintaan secara bersamaan (*heavy load*) pada *server*, jaringan, atau layanan tertentu untuk mengukur kinerja sistem dan menganalisis apakah sistem tetap berjalan dengan baik di bawah berbagai tingkat beban (Khlamov et al., 2022).

Arsitektur *JMeter* terdiri dari beberapa komponen utama, yaitu *test plan* yang berisi pengaturan dan skenario pengujian, *thread group* yang berfungsi mensimulasikan pengguna virtual, *sampler* yang mengirimkan permintaan (*request*) ke *server* tujuan, dan *listener* yang mengumpulkan serta menampilkan hasil pengujian. Untuk pengujian API, *HTTP Request Sampler* dapat diatur untuk menguji *endpoint* REST, sedangkan pengujian gRPC membutuhkan penambahan *plugin* tertentu agar *JMeter* dapat mendukungnya (Khlamov et al., 2022).

Konfigurasi skenario pengujian beban di *JMeter* meliputi pengaturan parameter seperti jumlah pengguna yang berjalan secara bersamaan (pengguna simultan), lama waktu peningkatan jumlah pengguna (*ramp-up period*), dan durasi total pengujian. *JMeter* menyediakan laporan bawaan (*built-in reporting*) dengan metrik seperti waktu respons (*response time*), jumlah permintaan yang dapat diproses per satuan waktu (*throughput*), dan persentase kegagalan (*error rate*) yang dapat diekspor untuk dianalisis lebih lanjut (Khlamov et al., 2022).

1.2.8 Metrik Evaluasi Kinerja

Evaluasi kinerja sistem digunakan untuk menganalisis dan membandingkan performa dari kedua arsitektur komunikasi, yaitu REST API dan gRPC. Metrik yang digunakan dalam penelitian ini adalah *response time, throughput, resource utilization, network data rate*, dan *error rate*.

***Response Time* (Waktu Respon).** *Response time* adalah total waktu yang dibutuhkan oleh sistem untuk memberikan balasan (*respons*) atas sebuah permintaan (*request*) yang dikirimkan oleh klien. Waktu ini dihitung sejak klien mengirimkan permintaan hingga klien menerima balasan secara penuh. Dalam pengujian, metrik yang digunakan adalah *average response time* (waktu respons rata-rata). Semakin kecil nilai *response time*, semakin cepat sistem merespons sehingga kinerja sistem dinilai semakin baik dan responsif (Ali, 2022).

$$Average\ Response\ Time = \frac{\sum_{i=1}^{n}(T_{selesai} - T_{mulai})_i}{n} \quad (1)$$

Keterangan:
n = jumlah total permintaan
i = permintaan ke-i
$T_{selesai}$ = waktu *respons* selesai diterima
T_{mulai} = waktu *request* dikirim

***Throughput*.** *Throughput* adalah jumlah permintaan yang berhasil diproses oleh sistem dibagi dengan total waktu pengujian dalamsatuan detik. Metrik ini diukur dalam *transactions per second* (TPS) atau *requests per second* (RPS) digunakan untuk menggambarkan kapasitas sistem dalam menangani beban. *Semakin tinggi nilai throughput, semakin besar kemampuan sistem melayani permintaan, sehingga performa sistem dinilai semakin baik* (Ali, 2022).

$$Throughput = \frac{Jumlah\ Request}{Total\ Waktu\ Pengujian} \quad (2)$$

***Resource Utilization* (Utilisasi Sumber Daya).** *Resource utilization* mengukur seberapa banyak sumber daya komputasi (dalam hal ini CPU dan memori) yang dikonsumsi oleh *service* saat memproses permintaan. Metrik ini digunakan untuk mengevaluasi efisiensi sistem. Semakin kecil persentase utilisasi yang dibutuhkan untuk menangani beban kerja yang sama, maka sistem tersebut dianggap semakin efisien dan ringan. *CPU utilization* menunjukkan persentase waktu prosesor (CPU) yang digunakan oleh *service* untuk menjalankan instruksi, sedangkan *memory utilization* menunjukkan jumlah memori (RAM) yang dialokasikan dan digunakan oleh proses *service* selama pengujian (Ali, 2022).

$$Resource\ Utilization\ (\%) = \frac{Sumber\ Daya\ Terpakai}{Total\ Sumber\ Daya\ Tersedia} \times 100\% \quad (3)$$

***Network Data Rate* (Laju Transfer Data).** Network data rate diamati dari Sent KB per sec dan Received KB per sec pada JMeter, yang menunjukkan laju data per detik selama pengujian. Nilainya dipengaruhi oleh throughput dan ukuran data per request, sehingga dapat digunakan untuk melihat efisiensi pertukaran data pada REST dan gRPC dalam kondisi beban yang sama (Ain et al., 2025).

***Error Rate* (Tingkat Kesalahan)**. *Error rate* adalah persentase dari total permintaan yang gagal selama pengujian. Permintaan gagal dapat berupa *timeout*, respons server (kode status 5xx), atau kesalahan koneksi. Metrik ini digunakan untuk mengukur keandalan (*reliability*) sistem di bawah tekanan. Nilai *error rate* yang ideal adalah 0%. Nilai yang tinggi mengindikasikan bahwa sistem tidak stabil atau tidak mampu menangani beban kerja yang diberikan (Ali, 2022).

$$Error\ Rate\ (\%) = \frac{Jumlah\ Request\ Error}{Jumlah\ Request\ Total} \times 100\% \quad (4)$$

1.3 Tujuan dan Manfaat Penelitian

1.3.1 Tujuan

Tujuan yang ingin dicapai dari penelitian ini yaitu:

1. Mengimplementasikan REST API dan gRPC dalam sistem *Smart Parking* berbasis *microservices*.
2. Membandingkan dan mengevaluasi performa REST API dan gRPC dalam komunikasi antar layanan menggunakan metrik *response time*, *throughput*, *resource utilization*, *network data rate*, dan *error rate*.

1.3.2 Manfaat

Adapun manfaat dari penelitian ini adalah:

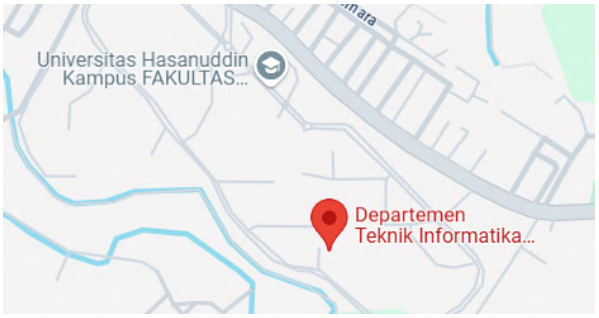
1. Meningkatkan efisiensi sistem *Smart Parking* dengan mengidentifikasi protokol komunikasi yang optimal melalui analisis komprehensif antara REST API dan gRPC.
2. Hasil analisis ini dapat digunakan sebagai acuan dalam pengembangan sistem berbasis arsitektur *microservices*.
3. Menyajikan analisis yang jelas terkait perbedaan kinerja antara REST API dan gRPC dalam konteks sistem *Smart Parking*.

BAB II

METODE PENELITIAN

2.1 Tempat dan Waktu Penelitian

Penelitian ini dilaksanakan sejak disetujuinya judul penelitian pada bulan Maret hingga November 2025. Lokasi pengambilan data dilakukan di Parkiran Departemen Teknik Informatika, Fakultas Teknik, Universitas Hasanuddin. Adapun proses penelitian dilakukan di Laboratorium *Artificial Intelligence* (AI), Departemen Teknik Informatika, Fakultas Teknik, Universitas Hasanuddin.



Gambar 4. Lokasi penelitian dan pengambilan data

2.2 Instrumen Penelitian

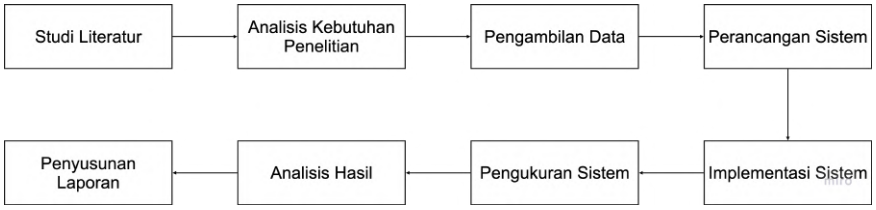
Berikut instrumen atau alat yang digunakan dalam penelitian ini:

1. Perangkat lunak:
   a. Visual Studio Code
   b. Apache JMeter v5.6.3
   c. Google Spreadsheets
   d. Microsoft Word
   e. Microsoft PowerPoint
   f. Microsoft Edge
   g. macOS Sequoia v15.0.1
   h. Terminal bawaan macOS
2. Perangkat keras:
   a. CCTV Vivotek IP9165-LPC
   b. GoPro Hero 11 Black
   c. MacBook Air (Apple M3, 8-core CPU, 16 GB RAM).
3. Bahasa pemrograman:
   a. Python v3.12 dan v3.13.5
   b. JavaScript (TypeScript)
   c. Protocol Buffers (.proto)
4. *Framework* dan Teknologi:
   a. Django REST Framework (DRF)
   b. React

c. gRPC
d. Envoy Proxy

5. *Library*:
   a. Django 5.2, digunakan sebagai *framework backend* utama untuk membangun layanan REST API dan gRPC pada sistem Smart Parking.
   b. Django REST Framework (DRF) 3.16.0, digunakan untuk mengelola endpoint API serta memfasilitasi komunikasi data antar layanan.
   c. grpcio 1.73.1, digunakan sebagai *core library* untuk mengimplementasikan komunikasi gRPC antar layanan microservices.
   d. grpcio-tools 1.73.1, digunakan untuk meng-*generate* kode Python dari file .proto sehingga setiap layanan dapat saling berkomunikasi melalui protokol gRPC.
   e. protobuf 6.31.1, digunakan untuk mendefinisikan format data (*serialization*) yang dikirimkan antar layanan gRPC dengan cara yang lebih efisien dibanding JSON.
   f. NumPy 2.2.5, digunakan untuk berbagai operasi numerik.
   g. OpenCV 4.11.0.86, digunakan untuk pemrosesan citra dan video, seperti pembacaan frame CCTV serta ekstraksi area pelat kendaraan.
   h. PaddleOCR 3.3.1, digunakan untuk mendeteksi serta mengenali teks pada pelat nomor kendaraan hasil deteksi citra.
   i. ONNX Runtime 1.22.0, mempercepat proses inferensi model deteksi berbasis ONNX (YOLO).
   j. psutil 7.0.0, digunakan untuk memantau penggunaan sumber daya sistem, seperti CPU dan memori, selama proses pengujian performa.
   k. PostgreSQL (psycopg2-binary 2.9.10), digunakan sebagai basis data utama untuk menyimpan informasi kendaraan, transaksi parkir, serta hasil deteksi.
   l. React 19.1.0, digunakan untuk membangun antarmuka *frontend* interaktif pada *Dashboard*.
   m. Chart.js 4.5.0, digunakan untuk menampilkan visualisasi data seperti statistik kendaraan dan pendapatan parkir.
   n. TailwindCSS 3.4.17, digunakan untuk mendesain tampilan *dashboard* agar lebih responsif dan modern.

2.3 Tahapan Penelitian



Gambar 5. Tahapan penelitian

Pelaksanaan penelitian ini mengikuti alur kerja yang terstruktur sebagaimana yang ditampilkan pada Gambar 5. Tahapan penelitian. Proses diawali dengan studi literatur, di mana literatur relevan dikumpulkan untuk membangun fondasi teoretis. Selanjutnya, dilakukan Identifikasi analisis kebutuhan penelitian untuk menetapkan ruang lingkup masalah, tujuan, serta metode dan data yang akan digunakan. Tahapan berikutnya adalah pengambilan data, di mana data yang telah dikumpulkan menjadi dasar utama dalam proses pengembangan sistem. Data ini kemudian menjadi acuan utama dalam proses perancangan sistem, di mana arsitektur, alur kerja, dan spesifikasi komponen sistem dirancang secara detail. Sistem yang telah dirancang kemudian diimplementasikan pada tahap implementasi sistem, agar dapat diuji dan dievaluasi kinerjanya. Sistem yang telah jadi selanjutnya memasuki tahap pengukuran sistem. Pada fase ini, kinerja sistem dievaluasi secara kuantitatif menggunakan metrik-metrik yang telah ditentukan sebelumnya untuk mengetahui efisiensinya. Tahap selanjutnya adalah analisis hasil, yang bertujuan untuk menilai efektivitas serta akurasi sistem berdasarkan hasil pengukuran yang telah dilakukan. Akhirnya, seluruh rangkaian proses, temuan, dan analisis didokumentasikan secara sistematis dalam penyusunan laporan penelitian.

2.4 Teknik Pengambilan Data

2.4.1 Data Primer

Seluruh data yang digunakan dalam penelitian ini merupakan video mentah (raw video) hasil rekaman CCTV yang diperoleh secara langsung di area portal parkir kampus. Kamera CCTV dipasang pada struktur besi palang parkir dengan ketinggian sekitar ±2 meter dari permukaan jalan dan sudut kemiringan kurang lebih 30°. Penempatan ini dipilih agar kamera dapat menangkap kendaraan yang melintas pada jalur masuk dan keluar secara jelas, sehingga objek kendaraan dan pelat nomor berada pada jarak serta sudut pandang yang memadai untuk diproses oleh sistem.

Pengambilan data dilakukan di lingkungan kampus dalam kondisi nyata, dengan pencahayaan alami dan aktivitas kendaraan normal tanpa rekayasa khusus, sehingga data merepresentasikan kondisi lapangan secara realistis. Objek pengamatan adalah mobil dan sepeda motor milik mahasiswa yang digunakan bergantian untuk mensimulasikan aktivitas parkir, sekaligus memberikan variasi data dari sisi jenis kendaraan dan karakteristik pelat nomor.

Untuk kebutuhan pengujian, penelitian ini menggunakan empat video mentah sebagai *input* dengan variasi resolusi dan FPS, sebagaimana ditampilkan pada Tabel 1. Spesifikasi Data Video Mentah. Variasi ini digunakan untuk mengevaluasi apakah perbedaan kualitas *input* (misalnya *Full* HD vs 4K dan 30 FPS vs 60 FPS) dapat memengaruhi beban pemrosesan dan komunikasi antar layanan, yang kemudian berdampak pada metrik pengujian. Video mentah tersebut dapat diakses melalui tautan pada Lampiran 3. Video *Input Raw*.

Tabel 1. Spesifikasi Data Video Mentah

Nama File	Resolusi	FPS	Durasi	Size
1080|30-FPS.mp4	1920 × 1080	30	01:23	98,6 MB
1080|60-FPS.mp4	1920×1080	60	01.57	283,8 MB
4K|30-FPS.mp4	3840×2160	30	01.30	510,5 MB
4K|60-FPS.mp4	3840×2160	60	01.26	643,7 MB

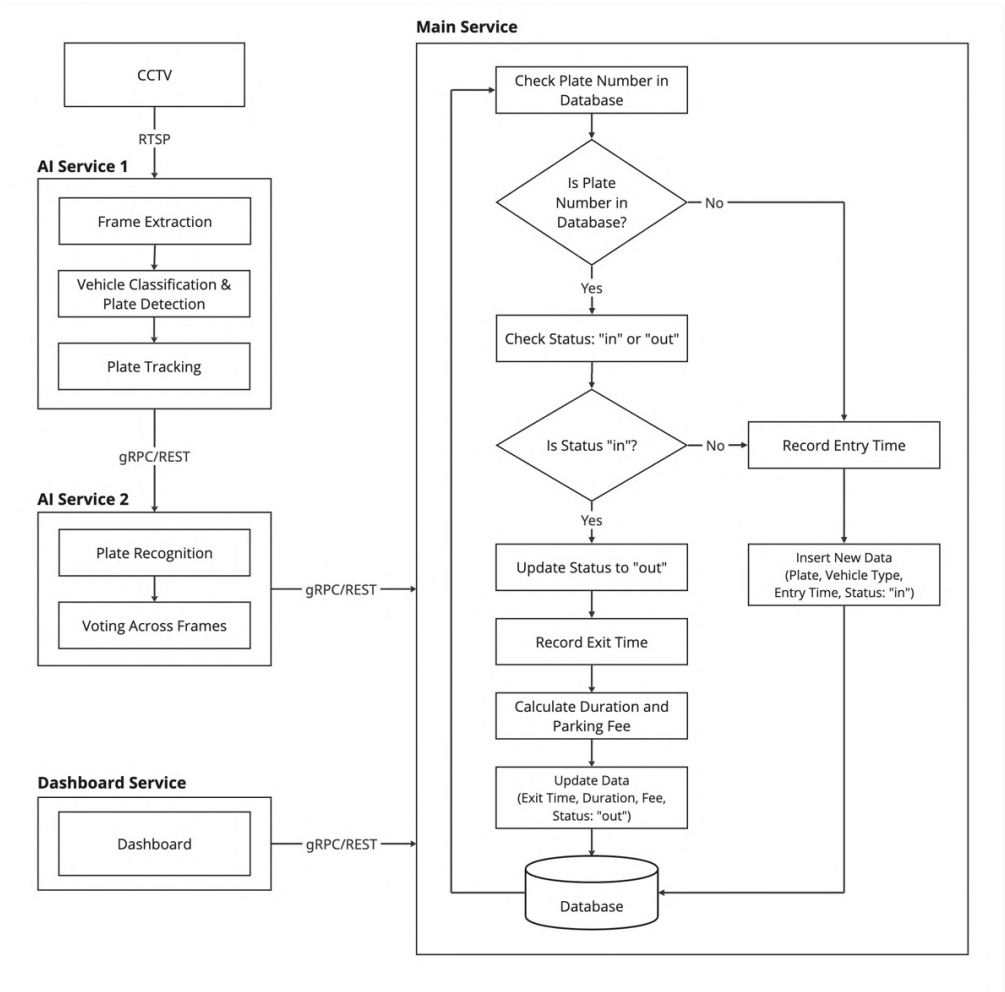
Video mentah kemudian diolah dengan mengekstraksi video menjadi rangkaian frame. Rangkaian frame ini menjadi input utama sistem pada tahap pemrosesan visual, khususnya untuk mendeteksi kendaraan dan pelat nomor sebelum dilakukan pengenalan teks pelat pada tahapan berikutnya. Untuk memperjelas teknik dan posisi pengambilan data, penempatan kamera dan area rekaman ditampilkan pada Gambar 6. Contoh pengambilan data

(a)

(b)

Gambar 6. Contoh pengambilan data: (a) Penempatan kamera; (b) Hasil ekstraksi *frame*

2.5 Perancangan dan Implementasi Sistem



Gambar 7. Perancangan alur sistem

Sistem *Smart Parking* ini dibangun menggunakan arsitektur *microservices* sebagaimana ditampilkan pada Gambar 5, di mana setiap fungsi utama dipisahkan menjadi layanan yang berdiri sendiri dan saling terhubung melalui REST API atau gRPC. Dalam sistem ini terdapat empat layanan utama, yaitu *Service* AI 1, *Service* AI 2, *Service Main*, dan *Service Dashboard*. Pemilihan arsitektur *microservices* dilakukan karena sistem *Smart Parking* membutuhkan fleksibilitas, skalabilitas, dan keandalan tinggi. Melalui pendekatan ini, setiap layanan dapat dikembangkan, diuji, dan diperbarui secara terpisah sesuai kebutuhannya.

2.5.1 Perancangan Sistem

Sistem *Smart Parking* dirancang menggunakan arsitektur *microservices* dengan pembagian fungsi ke beberapa layanan dan komunikasi antar layanan menggunakan REST API dan gRPC. *Source code* dapat diakses melalui tautan pada Lampiran 1. *Source Code*.

***Service* AI 1.** *Service* AI 1 merupakan komponen awal dalam sistem *Smart Parking* yang berfungsi melakukan pendeteksian dan pelacakan kendaraan beserta pelat nomornya secara otomatis dari video CCTV. Tahapan awal dimulai dengan proses ekstraksi *frame* video, di mana setiap video inputan dipecah menjadi sejumlah gambar terpisah menggunakan modul *frame extractor* agar dapat dianalisis secara independen. Setelah itu, setiap *frame* diproses menggunakan model deteksi *YOLOv11 (You Only Look Once)* dalam format ONNX di dalam pembungkus ByteTrack untuk mengenali tiga kelas utama objek, yaitu mobil (car), motor (motorcycle), dan pelat nomor (license-plate), sekaligus memberi ID pelacak (tracker ID) untuk menjaga konsistensi identitas antarframe. Hasil deteksi mencakup koordinat bounding box, label kelas, confidence score, serta ID untuk pelat.

Setelah proses deteksi selesai, sistem melanjutkan ke tahap pelacakan objek menggunakan algoritma *ByteTrack*, yang bertugas menjaga konsistensi identitas setiap objek di antara *frame* video. Melalui tahap ini, sistem mampu mengenali pelat nomor yang sama meskipun objek berpindah posisi, tertutup sebagian, atau muncul kembali setelah beberapa *frame*. Setiap pelat nomor diberikan ID pelacak unik (*tracker ID*) untuk memastikan konsistensi data antar*frame*. Setelah itu, dilakukan proses asosiasi antara pelat nomor dengan kendaraan yang sesuai menggunakan perhitungan *Intersection over Union (IoU)* serta jarak pusat (*centroid distance*). Pendekatan ini memungkinkan sistem menentukan hubungan spasial antara pelat dan kendaraan di sekitarnya, sehingga setiap pelat dapat dikategorikan secara akurat ke dalam tipe kendaraan tertentu, baik mobil maupun motor.

Selanjutnya, setiap hasil deteksi dan pelacakan dilengkapi dengan penanda waktu (*timestamp*) guna mencatat kapan objek terdeteksi dalam *frame*. Pencatatan waktu dilakukan secara otomatis dengan menggunakan zona waktu WITA (UTC+08:00) dan disimpan dalam format ISO 8601 agar selaras dengan standar waktu di seluruh sistem. Penambahan *timestamp* ini penting untuk memastikan setiap kejadian dapat diurutkan secara kronologis sekaligus menjadi dasar bagi tahap analisis berikutnya untuk menentukan waktu masuk atau keluar kendaraan.

```
[
  {
    "Class": "car",
    "Confidence": 0.9413,
    "BBox": [
      37,
      1,
      1329,
      869
    ]
  },
  {
    "Class": "license-plate",
    "Confidence": 0.8727,
    "BBox": [
      897,
      547,
      1157,
      687
    ],
    "ID": 45,
    "VehicleType": "car",
    "FrameTS": "2025-11-06T12:01:25.265+08:00"
  }
]
```


(a)

(b) (c)

Gambar 8. Contoh output *Service* AI 1: (a) hasil JSON; (b) hasil ekstraksi *frame* ; (c) hasil *frame* beranotasi;

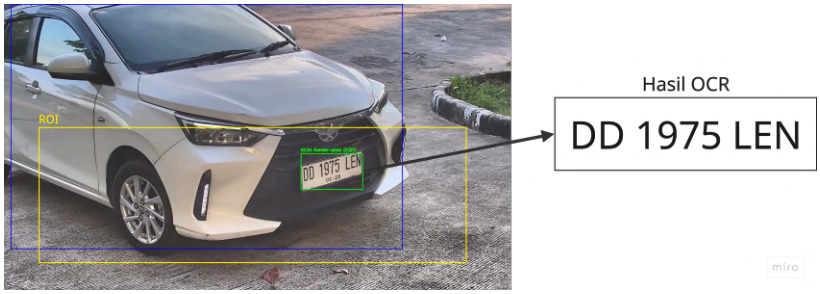
Setelah seluruh *frame* selesai diproses, sistem menghasilkan dua jenis *output*. Sesuai pada Gambar 8 output pertama, berkas *JSON* yang berisi data lengkap hasil deteksi dan pelacakan seperti kelas objek, koordinat *bounding box*, *confidence score*, *tracker* ID, tipe kendaraan hasil asosiasi, serta *timestamp* deteksi. Kedua, frame yang dikirim ke Service AI 2 untuk OCR tetap frame asli tanpa anotasi, sementara gambar beranotasi (overlay bbox) sebagai bukti visual yang disimpan lokal di folder output.

Tabel 2. Fungsi *Service* AI 1

Fungsi	*Request*	*Response*	Deskripsi
Detect Video	• *Path* video	Ringkasan proses dan jumlah frame	Melakukan ekstraksi *frame* dari video, mendeteksi objek kendaraan dan pelat nomor.

***Service* AI 2.** *Service* AI 2 bertugas melakukan pengenalan teks pada pelat nomor kendaraan (*Optical Character Recognition*/OCR) serta mengelola hasil deteksi yang dikirim oleh *Service* AI 1. Layanan ini menerima data berupa hasil deteksi pelat nomor dan kendaraan dalam format JSON beserta *frame*. Proses diawali dengan membaca data deteksi tersebut, kemudian mengekstraksi area pelat nomor dari setiap *frame* menggunakan koordinat *bounding box* yang dikirimkan sebelumnya. Setiap citra pelat yang terpotong ini diproses menggunakan metode *PaddleOCR*, yang dikonfigurasi untuk alfabet Latin/Inggris; penyesuaian terhadap pola pelat nomor Indonesia dilakukan melalui tahapan pemformatan hasil (*formatter*) di sisi aplikasi. Hasil pembacaan teks dari setiap pelat disimpan sebagai dua bentuk data, yaitu teks mentah (*raw text*) dan versi yang telah diformat agar sesuai dengan pola pelat nomor Indonesia.

Selama proses akumulasi data, sistem menerapkan mekanisme pelacakan lintas *frame* yang menggabungkan hasil OCR dari beberapa *frame* berbeda yang memiliki identitas pelat (ID) yang sama. Selain mengandalkan ID pelacak dari *Service* AI 1, sistem juga memetakan ID mentah tersebut ke *canonical ID* berdasarkan kedekatan spasial (kombinasi *Intersection over Union*/IoU dan jarak pusat/*centroid distance*) untuk menjaga konsistensi identitas ketika terjadi perubahan ID antar*frame*. Dengan cara ini, *Service* AI 2 tidak hanya membaca satu kali hasil dari sebuah *frame*, melainkan mengumpulkan beberapa pembacaan dari berbagai sudut atau kondisi pencahayaan, untuk kemudian dilakukan proses *voting* guna menentukan hasil teks yang paling konsisten. Tahapan ini meningkatkan akurasi sistem karena mampu mengurangi kesalahan pembacaan akibat *noise*, refleksi, atau rotasi pelat. Selain itu, *Service* AI 2 juga melakukan proses validasi zona deteksi (*Region of Interest*/ROI) untuk memastikan bahwa pelat benar-benar berada dalam area deteksi yang diinginkan sebelum diproses lebih lanjut, sehingga sistem hanya menganggap pelat yang relevan dengan area yang telah ditentukan.



Gambar 9. Contoh OCR dan *finalize* kendaraan

Setelah data teks hasil OCR terakumulasi dan divalidasi, *Service* AI 2 melakukan proses finalisasi (*finalize*) seperti pada Gambar 9. Contoh OCR dan finalize kendaraan, untuk menentukan pelat nomor yang telah memiliki bukti deteksi yang cukup. Sistem menghitung jumlah kemunculan dan tingkat konsistensi hasil pembacaan dari *frame-frame* sebelumnya; proses *finalize* dipicu otomatis saat pelat keluar dari ROI atau tidak lagi terdeteksi. Jika hasil pembacaan telah mencapai ambang batas minimum yang ditetapkan, misalnya jumlah *votes* tertentu, maka sistem menganggap data pelat tersebut valid. Selain itu, *Service* AI 2 juga menerapkan mekanisme pencegahan duplikasi (*duplicate filtering*) dengan cara menyimpan catatan waktu terakhir setiap pelat yang telah dikirim ke sistem utama. Jika pelat yang sama terdeteksi kembali dalam rentang waktu singkat di bawah batas yang telah ditentukan (misalnya lima menit), sistem akan melewatkannya agar tidak terjadi pengiriman ganda yang dapat memengaruhi pencatatan waktu parkir.

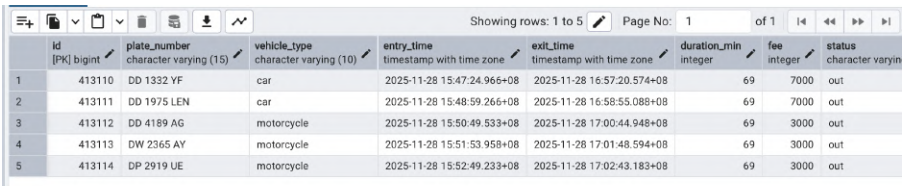
Hasil akhir dari *Service* AI 2 berupa informasi pelat nomor yang telah tervalidasi, jenis kendaraan, waktu deteksi, serta lokasi parkir, yang semuanya disusun dalam format terstruktur dan siap dikirim ke layanan utama (*Service Main*) melalui HTTP atau gRPC sesuai konfigurasi. Data ini kemudian digunakan untuk mencatat aktivitas kendaraan masuk dan keluar pada sistem parkir. Dengan kombinasi antara pembacaan OCR, validasi spasial ROI, *voting* hasil multi-*frame*, dan mekanisme penghindaran duplikasi, *Service* AI 2 memastikan bahwa setiap pelat nomor yang dikirim ke layanan utama memiliki keakuratan tinggi dan bebas dari redundansi data.

Tabel 3. Fungsi *Service* AI 2

Fungsi	Request	Response	Deskripsi
Recognize From Frame	• *Frame* (image/jpeg) • File JSON deteksi • Token *internal*	Pesan status *recognize*	Melakukan pembacaan teks pelat nomor dari setiap frame yang diterima.

Finalize Detections	• *Array* tracker_id • Token *internal*	Pesan status *finalize*	Melakukan voting lintas-frame dan mengirim data hasil final ke *Service Main*.

Service Main. *Service Main* berfungsi sebagai komponen pusat pengendali dan pengelola data kendaraan dalam sistem *Smart Parking*, yang menerima hasil identifikasi dari *Service* AI 2 untuk diolah menjadi informasi parkir yang lengkap. Setiap data yang diterima berisi informasi pelat nomor, jenis kendaraan, waktu deteksi, dan lokasi parkir. Berdasarkan data tersebut, sistem akan menentukan apakah kendaraan baru saja masuk atau hendak keluar dari area parkir. Jika pelat nomor belum tercatat sebelumnya, maka kendaraan dianggap baru masuk dan waktu tersebut disimpan sebagai waktu masuk (*entry time*). Sebaliknya, jika kendaraan dengan pelat yang sama sudah memiliki catatan aktif, maka sistem memperbarui statusnya menjadi keluar (*exit*) dan menghitung lama parkir berdasarkan selisih waktu antara masuk dan keluar. Proses ini memastikan bahwa setiap kendaraan memiliki catatan waktu yang konsisten dan tidak tumpang tindih. Selain itu, sistem juga melakukan validasi format pelat nomor untuk memastikan kesesuaiannya dengan pola standar pelat Indonesia, sehingga data yang tersimpan tetap akurat dan dapat dipercaya.



Showing rows: 1 to 5 Page No: 1 of 1

	id [PK] bigint	plate_number character varying (15)	vehicle_type character varying (10)	entry_time timestamp with time zone	exit_time timestamp with time zone	duration_min integer	fee integer	status character varyin
1	413110	DD 1332 YF	car	2025-11-28 15:47:24.966+08	2025-11-28 16:57:20.574+08	69	7000	out
2	413111	DD 1975 LEN	car	2025-11-28 15:48:59.266+08	2025-11-28 16:58:55.088+08	69	7000	out
3	413112	DD 4189 AG	motorcycle	2025-11-28 15:50:49.533+08	2025-11-28 17:00:44.948+08	69	3000	out
4	413113	DW 2365 AY	motorcycle	2025-11-28 15:51:53.958+08	2025-11-28 17:01:48.594+08	69	3000	out
5	413114	DP 2919 UE	motorcycle	2025-11-28 15:52:49.233+08	2025-11-28 17:02:43.183+08	69	3000	out

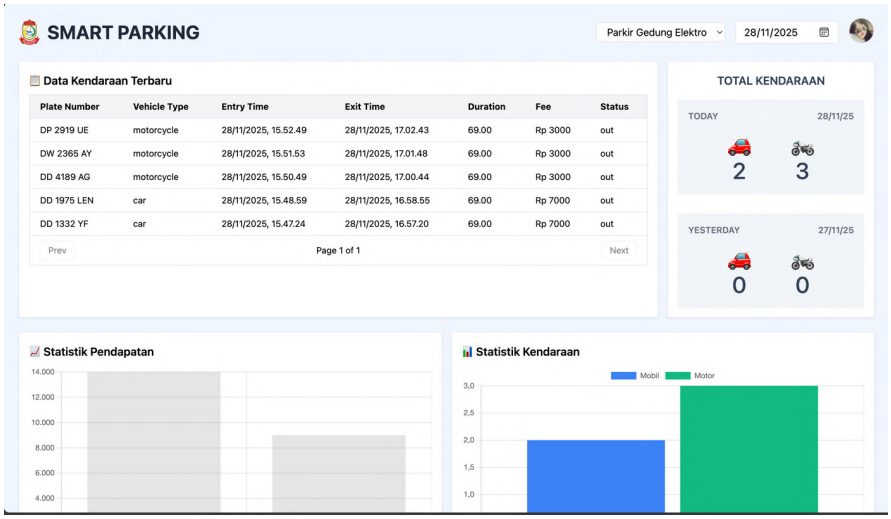
Gambar 10. Contoh *Database*

Setelah proses pembaruan data dilakukan sesuai pada Gambar 10. Contoh *Database*, *Service Main* akan menghitung biaya parkir secara otomatis berdasarkan jenis kendaraan dan durasi parkir yang telah diperoleh. Tarif dasar dan tambahan per jam ditentukan berbeda untuk kendaraan roda dua dan roda empat, menyesuaikan dengan ketentuan area parkir. Semua hasil pengolahan ini digunakan untuk mencatat aktivitas kendaraan, baik yang baru masuk maupun keluar, sekaligus menghasilkan rekapitulasi jumlah kendaraan yang sedang berada di area parkir. Data hasil pengolahan dari *Service Main* kemudian dapat digunakan oleh sistem lain, seperti *Service Dashboard*, untuk menampilkan statistik jumlah kendaraan, durasi rata-rata parkir, serta estimasi pendapatan harian. Dengan demikian, *Service Main* menjadi pusat logika bisnis sistem *Smart Parking* yang mengintegrasikan hasil deteksi dari modul kecerdasan buatan menjadi informasi operasional yang siap digunakan untuk pemantauan dan pengambilan keputusan manajemen parkir secara *real-time*.

Tabel 4. Fungsi *Service Main*

Fungsi	Request	Response	Deskripsi
Process Plate	• *Plate number* • *Vehicle type* • *Timestamp* • *Location ID* • *Token internal*	• Pesan status (masuk/keluar • *Fee*	Menyimpan atau memperbarui data kendaraan yang masuk atau keluar area parkir.

Service Dashboard. *Service Dashboard* berfungsi sebagai antarmuka utama yang menampilkan hasil pengolahan data dari sistem *Smart Parking* secara terpusat dan *real-time*. Halaman *dashboard* terdiri atas beberapa komponen inti, yaitu *VehicleTable*, *SummaryCard*, *IncomeChart*, dan *VehicleChart*, yang masing-masing menampilkan data kendaraan terbaru, rekapitulasi total kendaraan hari ini dan kemarin, statistik pendapatan per jam, serta jumlah kendaraan berdasarkan waktu dan jenis. Di bagian atas halaman terdapat kontrol untuk memilih lokasi parkir dan tanggal pengamatan, disertai menu akun pengguna. Seluruh data diambil secara paralel dari empat *endpoint backend* menggunakan parameter *date* dan *location_id*, kemudian dirender dalam dua bagian besar, yaitu tabel data kendaraan dengan ringkasan di sisi kanan serta dua grafik visual di bagian bawah. Proses pemanggilan data didukung oleh sistem autentikasi token JWT dengan mekanisme penyegaran otomatis jika sesi berakhir, sehingga pengguna tetap dapat mengakses data tanpa gangguan. Seluruh waktu pada tampilan dikonversi ke zona waktu WITA (Asia/Makassar) agar konsisten dengan lokasi operasional sistem.



Gambar 11. Contoh *Dashboard*

Dari sisi visual, *dashboard* menggunakan kombinasi *React*, *TypeScript*, *TailwindCSS*, dan *Chart.js* untuk menghadirkan tampilan yang responsif dan

mudah dibaca seperti pada Gambar 11. Contoh *Dashboard*. Komponen *VehicleTable* menampilkan daftar kendaraan beserta detail waktu masuk dan keluar, durasi parkir, biaya, serta status kendaraan dengan sistem paginasi untuk menjaga keteraturan data. Sementara itu, *SummaryCard* memberikan informasi cepat mengenai total kendaraan yang masuk dan keluar setiap hari, sedangkan grafik pendapatan dan grafik jumlah kendaraan membantu pengguna memantau tren parkir secara langsung. Seluruh data diperbarui secara dinamis tanpa perlu memuat ulang halaman, menghasilkan pengalaman pengguna yang lancar. Dengan rancangan ini, *Service Dashboard* tidak hanya menampilkan informasi, tetapi juga berfungsi sebagai alat pemantauan dan analisis visual yang membantu pengelola dalam mengambil keputusan berbasis data terkait kondisi dan performa area parkir.

Tabel 5. Fungsi *Service Dashboard*

Fungsi	Request	Response	Deskripsi
Login	• *Username* • *Password*	Token akses & refresh	Melakukan autentikasi pengguna agar dapat mengakses *dashboard*.
Get Summary	• *Date*	Total kendaraan hari ini & kemarin	Menampilkan ringkasan jumlah kendaraan pada dua hari terakhir.
Get Income Chart	• *Date*	Data pendapatan per jam	Menampilkan grafik pendapatan parkir berdasarkan jam.
Get Vehicle Chart	• *Date*	Data jumlah kendaraan per jenis per jam	Menampilkan grafik jumlah kendaraan mobil dan motor setiap jam.
Get Vehicle Table	• *Date*	Daftar kendaraan masuk & keluar	Menampilkan tabel kendaraan dengan waktu masuk, keluar, dan tarif.

2.5.2 Struktur Database

Struktur basis data pada sistem *smart parking* menggunakan *PostgreSQL* sebagai sistem manajemen data utama yang terintegrasi langsung dengan *framework* Django. Basis data ini berfungsi untuk menyimpan seluruh informasi kendaraan yang terdeteksi oleh sistem, baik saat kendaraan masuk maupun keluar area parkir. Tabel utama yang digunakan adalah tabel *Vehicle*, yang berisi beberapa kolom penting seperti *plate_number*, *vehicle_type*, *entry_time*, *exit_time*, *duration_min*, *fee*, *status*, dan *location_id*. Setiap kolom memiliki perannya masing-masing, misalnya untuk mencatat nomor pelat kendaraan, jenis kendaraan (mobil atau motor), waktu masuk dan keluar, lama waktu parkir, biaya parkir yang dikenakan, serta status kendaraan apakah masih berada di area parkir atau sudah keluar. Selain itu, kolom *location_id* digunakan untuk membedakan lokasi parkir yang berbeda dalam sistem. Desain tabel ini dibuat agar data dapat diperbarui secara otomatis tanpa menimbulkan duplikasi, sehingga sistem mampu melacak pergerakan kendaraan dengan efisien. Tabel *Vehicle* menjadi pusat penyimpanan utama yang menghubungkan seluruh layanan dalam sistem *smart parking*, mulai dari proses deteksi hingga penampilan data di *dashboard*.

Tabel 6. Skema Entitas Vehicle

Kolom	Tipe Data (PostgreSQL)	Deskripsi
id (PK)	bigserial	Primary key otomatis.
plate_number	*varchar(15)*	Nomor pelat hasil normalisasi (mis. DD 1374 SYA).
vehicle_type	*varchar(10)*	*Jenis kendaraan: car atau motorcycle.*
entry_time	*timestamptz*	Waktu masuk (tersimpan UTC, ditampilkan WITA di FE).
exit_time	*timestamptz*	Waktu keluar (diisi saat kendaraan keluar).
duration_min	*integer*	Durasi parkir (menit), dihitung saat keluar.
fee	*integer*	Biaya parkir (Rupiah), dihitung saat keluar.
status	*varchar(10)*	Status transaksi: *in* atau *out*.
location_id	*varchar(50)*	Kode lokasi parkir (mis. PARKIR-GEDEL-001).

2.5.3 Implementasi Komunikasi REST API

Komunikasi berbasis REST (*Representational State Transfer*) diimplementasikan sebagai mekanisme pertukaran data standar dalam sistem ini. Setiap layanan dirancang untuk mengekspos *endpoint* HTTP yang memungkinkan interaksi terstruktur menggunakan format data JSON. Berikut adalah rincian implementasi untuk masing-masing layanan:

***Service* AI 1.** Implementasi komunikasi pada *service* ini menggunakan pendekatan berbasis HTTP dengan format pertukaran data *multipart/form-data* dan *application/json*. *Service* AI 1 berperan sebagai pengirim (*client*) yang melakukan komunikasi ke *Service* AI 2 setelah proses deteksi dan pelacakan objek selesai dilakukan.

Endpoint utama yang disediakan oleh *service* ini adalah */api/detect/* dengan metode POST. *Endpoint* ini digunakan untuk memulai proses pendeteksian video. Permintaan yang dikirimkan berisi *payload* berupa *path* video yang akan diproses. Setelah permintaan diterima, sistem akan mengekstraksi *frame* dari video tersebut dan kemudian mengirimkan setiap hasil deteksi ke *Service* AI 2 melalui *endpoint* REST API yang telah dikonfigurasi, yaitu AI2_OCR_URL untuk proses pengenalan teks pelat nomor.

Dalam pengiriman data ke *Service* AI 2, *Service* AI 1 menyertakan dua berkas utama dalam format *multipart/form-data*, yaitu berkas gambar (*frame*) hasil ekstraksi dan berkas JSON yang berisi hasil deteksi objek dalam *frame* tersebut. Selain itu, setiap permintaan yang dikirim ke *Service* AI 2 dilengkapi dengan *header* Authorization: Bearer <token>, yang berfungsi sebagai autentikasi antarservice agar komunikasi hanya dapat dilakukan oleh *service* yang terotorisasi. Proses komunikasi ini juga diatur dengan parameter *REQUEST_TIMEOUT*, yang berfungsi membatasi waktu tunggu agar sistem tetap responsif dan dapat menangani banyak permintaan secara paralel.

Tabel 7. REST API – *Service* AI 1

Endpoint	Metode	Path	Deskripsi	Request Body	Response
Deteksi Video	POST	/api/detect/	Memulai proses deteksi video dan mengirim hasil ke AI 2.	{"*video*_path ": "*path/video. mp4*"}	{ "*message*": "Deteksi selesai untuk X *frame*" }

***Service* AI 2.** *Service* AI 2 memiliki dua *endpoint* utama dalam komunikasi REST, yaitu */api/ocr/* dan */api/ocr/finalize/*. *Endpoint* ini menangani proses penerimaan data hasil deteksi dari *Service* AI 1. Setiap permintaan yang masuk membawa dua komponen utama, citra (*frame*) dan berkas JSON hasil deteksi, yang dikirim melalui format *multipart/form-data*. Melalui endpoint */api/ocr/* sistem memproses OCR pada pelat nomor dengan memotong bbox dari frame asli,

Sementara itu, *endpoint /api/ocr/finalize/* berfungsi menerima daftar *tracker IDs* dari kendaraan yang telah selesai dipantau untuk dilakukan proses *voting* yang terjadi otomatis saat pelat keluar ROI. Jika hasil akumulasi telah memenuhi jumlah minimal pembacaan (misalnya 10 kali), *Service* AI 2 akan mengirimkan data final ke *Service Main* untuk dicatat dalam basis data sistem. Seluruh proses pengiriman antarservice disertai autentikasi token menggunakan *header* Authorization: Bearer <token> dan Content-Type: application/json, yang memastikan hanya *service* terotorisasi yang dapat berkomunikasi.

Tabel 8. REST API – *Service* AI 2

Endpoint	Metode	Path	Deskripsi	Request Body	Response
OCR Frame Detection	POST	/api/ocr/	Menerima data frame dan hasil deteksi dari AI 1 untuk dilakukan proses OCR dan akumulasi teks pelat.	multipart/form-*data* (*frame* dan *detections* (JSON))	{"*status*":"*ok*", "*added*":0}
Finalisasi Deteksi	POST	/api/ocr/finalize/	Melakukan proses voting hasil OCR lintas frame dan mengirimkan hasil akhir ke *Service Main*.	{"*tracker_ids*": [1, 2, 3]}	{ "*message*": "*Finalize done*", "*results*": [...]}

Main *Service*. *Service* ini dikembangkan menggunakan *framework Django REST Framework* (DRF) dan menggunakan *PostgreSQL* sebagai basis data utama. Tujuan utamanya adalah untuk mencatat, memperbarui, serta menyediakan data kendaraan dan transaksi parkir yang akan digunakan oleh komponen lain seperti *Service Dashboard*.

Implementasi REST API pada *Service Main* memiliki *endpoint* utama */api/process-plate/*, yang menerima data dari *Service* AI 2 untuk mencatat status kendaraan. Setiap permintaan ke *endpoint* ini berisi parameter seperti *plate_number*, *vehicle_type*, *timestamp*, dan *location_id*, yang digunakan untuk menentukan apakah kendaraan sedang masuk atau keluar dari area parkir. Permintaan ke *endpoint* ini menggunakan format *application/json* dan wajib menyertakan autentikasi antarservice melalui *header* Authorization: Bearer <token> (token internal dari konfigurasi INTERNAL_API_KEY). Jika kendaraan dengan pelat nomor yang sama belum tercatat sebagai *in*, sistem akan mencatatnya sebagai kendaraan masuk beserta waktu kedatangannya. Sebaliknya, jika kendaraan sudah tercatat, sistem akan menandainya sebagai keluar serta menghitung durasi dan biaya parkir berdasarkan tipe kendaraan.

Tabel 9. REST API – *Service Main*

Endpoint	Metode	Path	Deskripsi	Request Body	Response
Proses Data Kendaraan	POST	/api/process-plate/	Menerima data dari AI 2 untuk mencatat status kendaraan masuk atau keluar.	{ "*plate_number*": "DD 1234 XX", "*vehicle_type*": "*car*", "*timestamp*": " 2025-10-30T01:00:00Z", "*location*_id": "PARKIR-GEDEL-001" }	{ "*message*": "Kendaraan masuk/keluar",”*fee*":...,"duration_minutes":...,"*vehicle_type*":...

***Service* Dashboard.** *Service Dashboard* berperan sebagai antarmuka visual utama bagi pengguna sistem *Smart Parking*. Komponen ini dikembangkan menggunakan *React.js* dengan bahasa *TypeScript* dan berfungsi untuk menampilkan seluruh data yang dikumpulkan dan diolah oleh *Service Main* melalui REST API. Seluruh komunikasi antara *Service Dashboard* dan *Service Main* dilakukan secara *stateless* dengan format pertukaran data berbasis JSON, serta dilindungi menggunakan mekanisme autentikasi berbasis JWT (*JSON Web Token*). Setiap permintaan ke *endpoint dashboard* wajib menyertakan *header* Authorization: Bearer <access_token> yang diperoleh dari proses *login*. Dengan pendekatan ini, *dashboard* dapat mengakses data seperti daftar kendaraan, statistik pendapatan, dan rekapitulasi jumlah kendaraan dengan aman dan efisien.

Pada implementasinya, *dashboard* mengonsumsi beberapa *endpoint* REST yang disediakan oleh *Service Main*. Saat pengguna berhasil *login* melalui *endpoint /api/auth/token/*, sistem akan menghasilkan token akses dan token *refresh* yang disimpan secara lokal di *browser*. Token ini digunakan untuk mengotorisasi setiap permintaan selanjutnya ke *endpoint* seperti */api/dashboard/summary/*, */api/dashboard/income/*, */api/dashboard/vehicles/*, dan */api/dashboard/table/*. Seluruh *endpoint dashboard* menggunakan metode GET dan mengembalikan data dalam format JSON (*timestamp* ISO 8601). Apabila *server* mengembalikan 401 Unauthorized, klien akan memanggil *endpoint /api/auth/token/refresh/* dengan *body* { "refresh": "<token>" } untuk memperoleh *access token* baru, lalu mengulangi *request* sebelumnya dengan *header* Authorization yang sudah diperbarui.

Antarmuka *dashboard* dirancang agar mudah digunakan oleh operator parkir atau pengelola area parkir. Pengguna dapat memilih tanggal melalui elemen *input date* sehingga data yang ditampilkan dapat disesuaikan secara dinamis. Selain itu, *dashboard* dilengkapi dengan fitur *logout* yang secara otomatis menghapus token dari penyimpanan lokal untuk menjaga keamanan akses sistem. Semua komunikasi REST API dilakukan secara *asynchronous* melalui fungsi *fetch()* (atau klien HTTP setara) dengan *header* Authorization: Bearer <access_token>; mekanisme *refresh token* dijalankan otomatis ketika token lama sudah tidak valid.

Tabel 10. REST API – *Service Dashboard*

Endpoint	Metode	Path	Deskripsi	Request Body	Response
Login	POST	/api/auth/token/	Melakukan autentikasi pengguna dan menghasilkan token akses serta *refresh*.	*{ "username": "admin", "password": "12345" }*	*{ "access": "<token>", "refresh": "<token>" }*
Refresh Token	POST	/api/auth/token/refresh/	Memperbarui token akses yang telah	*{ "refresh": "<token>" }*	*{ "access": "<new_token>" }*

Endpoint	Metode	Path	Deskripsi	Request Body	Response
			kedaluwarsa agar sesi tetap aktif.		
Ringkasan Kendaraan	GET	/api/dashboard/summary/?date=YYYY-MM-DD	Mengambil jumlah total kendaraan masuk per hari berdasarkan jenis kendaraan.	–	*{ "today": {"car": "...", "motorcycle": "..."}, "yesterday": {"car": "...", "motorcycle": "..."}}*
Statistik Pendapatan	GET	/api/dashboard/income/?date=YYYY-MM-DD	Menampilkan pendapatan parkir per jam berdasarkan data kendaraan keluar.	–	*[{"hour": "...", "total_fee": "..."}, {"hour": "...", "total_fee": "..."}]*
Statistik Kendaraan	GET	/api/dashboard/vehicles/?date=YYYY-MM-DD	Mengambil jumlah kendaraan per jam berdasarkan tipe kendaraan.	–	*[{"hour": "...", "vehicle_type": "...", "total": "..."}]*
Tabel Kendaraan	GET	/api/dashboard/table/	Menampilkan daftar kendaraan lengkap dengan waktu masuk, keluar, durasi, dan biaya parkir.	–	*[{plate_number, vehicle_type, entry_time, exit_time, duration_min, fee, status}]*

2.5.4 Implementasi Komunikasi gRPC

***Service* AI 1.** Dalam implementasi berbasis gRPC, komunikasi antara *Service* AI 1 dan *Service* AI 2 dilakukan melalui pemanggilan *remote procedure call* (RPC) yang terdefinisi pada berkas *Protocol Buffers*. Melalui pendekatan ini, pertukaran data dapat dilakukan lebih cepat dan efisien dibandingkan dengan format berbasis teks seperti JSON, karena gRPC menggunakan serialisasi biner dan koneksi *persistent* berbasis HTTP/2.

Service ini mendefinisikan satu *method* utama, yaitu *DetectVideo*, yang diimplementasikan pada berkas ai_detector/grpc_server.py. *Method* ini menerima gRPC request *DetectionRequest* berisi *video*_path, lalu mengekstrak frame, mendeteksi objek, dan mengirim tiap *frame* serta JSON deteksi ke Service AI 2 untuk proses OCR. Komunikasi ke AI 2 tetap menyertakan metadata autentikasi authorization: Bearer <token> agar hanya service terverifikasi yang dapat mengakses.

Tabel 11. gRPC – *Service* AI 1

Method	Request Message	Response Message	Deskripsi
DetectVideo	*DetectionRequest (video_path: string)*	*DetectionReply (message: spring)*	Menjalankan deteksi dari video mengekstrak *frame*, dan memicu pengiriman data *frame* + JSON ke AI 2 untuk OCR.

***Service* AI 2.** Dalam arsitektur berbasis gRPC, *Service* AI 2 berperan sebagai *server* yang menerima permintaan (*request*) dari *Service* AI 1 dan memberikan balasan (*response*) setelah melakukan pemrosesan data. Komunikasi ini didefinisikan secara eksplisit di dalam berkas *Protocol Buffers*.

Service ini memiliki dua *method* utama, yaitu *RecognizeFromFrame* dan *FinalizeDetections*. *Method RecognizeFromFrame* berfungsi menerima data citra serta berkas JSON hasil deteksi objek yang dikirim oleh *Service* AI 1 dalam bentuk *message OCRRequest*. Setelah data diterima, *Service* AI 2 akan melakukan pembacaan teks pelat menggunakan pustaka *PaddleOCR* dan menyimpan hasilnya sementara untuk dilakukan proses akumulasi lintas *frame*. Seluruh proses dilakukan secara terlindungi dengan sistem autentikasi berbasis *Bearer Token*, di mana setiap *request* dari *Service* AI 1 harus menyertakan *metadata* authorization: Bearer <token> (metadata gRPC) agar komunikasi hanya terjadi antarservice internal. *Payload* yang dikirim adalah *bytes* murni: frame_data (JPEG) dan json_data (string JSON), sehingga efisien di atas HTTP/2.

Setelah *RecognizeFromFrame* mengakumulasi cukup sampel, hasil dibawa ke tahap *finalize*, kombinasi voting lintas *frame* dan validasi ROI untuk memastikan pelat yang sama diberi *canonical* ID yang konsisten. Proses *finalize* dipicu otomatis saat pelat keluar ROI/hilang melalui *method FinalizeDetections*. Hanya data yang melewati ambang *vote* yang dikirim ke Service Main, menandai akhir satu siklus pemrosesan OCR. Dengan mekanisme ini, Service AI 2 bertindak sebagai penghubung cerdas antara proses deteksi objek dan sistem utama yang menyimpan serta menghitung data parkir, sementara jalur komunikasinya tetap efisien karena memakai serialisasi biner dan koneksi HTTP/2.

Tabel 12. gRPC – *Service* AI 2

Method	Request Message	Response Message	Deskripsi
RecognizeFromFrame	*OCRRequest* (*frame*_data, *json*_data)	*OCRReply* (plate_number: string, vehicle_type: string, timestamp: string)	Menerima data dari AI 1 dan melakukan proses OCR untuk membaca teks pelat kendaraan.
FinalizeDetections	*FinalizeRequest* (*tracker*_ids)	*FinalizeReply* (message: string)	Memfinalisasi deteksi berdasarkan tracker_id setelah voting intas frame. Dan mengirimkan data ke Service Main.

Service Main. *Service Main* merupakan pusat koordinasi yang menerima hasil akhir dari proses OCR kendaraan melalui komunikasi gRPC. *Service* ini bertugas melakukan validasi, penyimpanan data kendaraan ke *database*, serta perhitungan waktu parkir dan biaya yang harus dibayarkan. Dalam arsitektur gRPC, *Service Main* berperan sebagai *server* yang melayani permintaan dari *Service* AI 2, sementara AI 2 bertindak sebagai *client* yang mengirimkan data hasil pembacaan pelat nomor. Selain berfungsi sebagai penerima data, *service* ini juga menyediakan *method* gRPC yang digunakan untuk menampilkan data ringkasan, statistik, serta laporan pendapatan. Dengan cara ini, *Service Main* menjadi penghubung langsung antara sistem AI dan modul konsumsi data melalui jalur gRPC.

Method utama yang diimplementasikan pada *Service Main* adalah *ProcessPlate*, yang menerima *message PlateRequest* dari *Service* AI 2. Setiap permintaan yang masuk akan berisi informasi kendaraan yang telah diidentifikasi, termasuk hasil pembacaan OCR dan jenis kendaraan (*car* atau *motorcycle*). Setelah menerima data tersebut, *Service Main* akan melakukan validasi terhadap *authorization token* yang dikirim dalam *metadata* authorization: *Bearer* <*token*> (metadata gRPC). Jika *token* valid, sistem akan memproses data tersebut untuk menentukan apakah kendaraan baru saja masuk atau keluar area parkir dengan membandingkan data pelat terhadap catatan sebelumnya dalam basis data.

Apabila kendaraan baru terdeteksi masuk, *Service Main* akan menyimpan informasi baru ke tabel *Vehicle* dengan mencatat waktu masuk (*entry_time*), jenis kendaraan, dan lokasi parkir. Sebaliknya, jika kendaraan dengan pelat yang sama sudah tercatat dan belum memiliki waktu keluar, sistem akan memperbarui data tersebut dengan *exit_time*, menghitung lama waktu parkir secara otomatis (*duration_min*), serta menentukan besaran biaya parkir (*fee*) berdasarkan durasi dan jenis kendaraan. Semua hasil pemrosesan ini kemudian dikembalikan dalam bentuk *message PlateResponse* kepada *Service* AI 2 sebagai tanda bahwa transaksi berhasil dilakukan.

Tabel 13. gRPC – Service Main

Method	Request Message	Response Message	Deskripsi
ProcessPlate	*PlateRequest* (*plate_number, vehicle_type, timestamp, location*_id)	*PlateResponse* (*message*, *fee*)	Menerima hasil pembacaan pelat dari AI 2, memproses logika masuk/keluar kendaraan, menghitung durasi parkir.

***Service Dashboard*.** Pada komunikasi berbasis gRPC, *Service Dashboard* berperan sebagai klien yang terhubung langsung dengan *Service Main* untuk menampilkan data hasil pengolahan sistem parkir secara *real-time*. *Dashboard* tidak lagi melakukan proses perhitungan atau penyimpanan data sendiri, melainkan hanya menampilkan hasil yang telah diolah dan disediakan oleh *Service Main* melalui beberapa *method* gRPC. Dengan pendekatan ini, beban pemrosesan data terpusat di sisi *server*, sedangkan antarmuka pengguna hanya berfungsi sebagai lapisan visual yang ringan, cepat, dan mudah diakses.

Ketika pengguna memilih tanggal melalui antarmuka, *dashboard* akan melakukan serangkaian panggilan ke beberapa *method* gRPC yang ada di *Service Main*, yaitu *GetSummary*, *GetIncomeStats*, *GetVehicleStats*, dan *GetVehicleTable*. Masing-masing *method* memiliki fungsi spesifik yang digunakan untuk menampilkan komponen-komponen berbeda di halaman *dashboard*. *Method GetSummary* digunakan untuk mengambil jumlah total kendaraan (mobil dan motor) yang masuk pada hari ini dan kemarin, yang *kemudian* ditampilkan pada kartu ringkasan di bagian atas antarmuka. Selanjutnya, *method GetIncomeStats* digunakan untuk memperoleh data total pendapatan parkir per jam, yang divisualisasikan dalam bentuk grafik batang. *Method GetVehicleStats* berfungsi untuk menampilkan jumlah kendaraan per jam berdasarkan jenisnya, sehingga pengguna dapat melihat pola kepadatan kendaraan dalam satu hari. Terakhir, *method GetVehicleTable* digunakan untuk menampilkan daftar lengkap data kendaraan yang masuk dan keluar, termasuk waktu, durasi parkir, biaya, dan status kendaraan. Parameter *request* yang digunakan oleh keempat *method* tersebut adalah tanggal (*SummaryDateRequest.date*, format YYYY-MM-DD).

Setiap permintaan (*request*) dari *dashboard* dikirimkan dalam format *message* terstruktur yang didefinisikan dalam berkas *.proto*, seperti *SummaryDateRequest* atau *VehicleTableResponse*. Struktur ini memastikan bahwa data yang diterima oleh antarmuka telah terformat dengan baik dan dapat langsung digunakan tanpa perlu pemrosesan tambahan di sisi klien. Selain itu, *dashboard* juga dapat melakukan autentikasi pengguna menggunakan *method Login* dari *AuthService*. *Method Login* pada *AuthService* digunakan untuk autentikasi pengguna dashboard melalui gRPC. Klien mengirim *LoginRequest* berisi *username* dan *password*, lalu server memverifikasi kredensial. Jika valid, server mengembalikan *LoginReply* yang memuat sepasang token JWT (*access* dan *refresh*) beserta *message* status. Token access dipakai sebagai header Authorization: Bearer <token> pada panggilan gRPC-web berikutnya, sedangkan token *refresh* disiapkan untuk memperpanjang sesi tanpa perlu *login* ulang.

Tabel 14. gRPC – *Service Dashboard*

Method	**Request Message**	**Response Message**	**Deskripsi**
Login	*LoginRequest (username: string, password: string)*	*LoginReply (access: string, refresh: string, message: string)*	Mengautentikasi *user dashboard* dan mengembalikan pasangan token JWT (*access* + *refresh*).
GetSummary	*SummaryDateRequest (date: YYYY-MM-DD)*	*SummaryResponse (car_today, motorcycle_today, car_yesterday, motorcycle_yesterday)*	Mengambil data total kendaraan (mobil dan motor) pada hari ini dan kemarin untuk ditampilkan dalam *SummaryCard*.
GetIncomeStats	*SummaryDateRequest (date: YYYY-MM-DD)*	*IncomeStatsResponse (repeated HourlyIncome{hour, total_fee})*	Mengambil data pendapatan parkir per jam untuk divisualisasikan pada grafik *income*.
GetVehicleStats	*SummaryDateRequest (date: YYYY-MM-DD)*	*VehicleStatsResponse (repeated HourlyVehicle{hour, vehicle_type, total})*	Mengambil data jumlah kendaraan berdasarkan jenis dan waktu masuk, yang ditampilkan dalam grafik jumlah kendaraan.
GetVehicleTable	*SummaryDateRequest (date: YYYY-MM-DD)*	*VehicleTableResponse (repeated VehicleData { plate_number, vehicle_type, entry_time, exit_time, duration_min, fee, status })*	Mengambil daftar lengkap kendaraan, termasuk waktu masuk, keluar, durasi, biaya, dan status, untuk ditampilkan dalam tabel *dashboard*.

2.6 Pengukuran Kinerja Sistem

Skenario pengujian disusun untuk merepresentasikan alur komunikasi nyata yang terjadi pada sistem *Smart Parking*, mulai dari proses deteksi kendaraan hingga penyajian data ke *dashboard*. Setiap pengujian dilakukan selama 180 detik dengan variasi jumlah pengguna sebanyak 5, 10, 15, 20, dan 25 pengguna, serta *ramp-up period* antara 5–20 detik, menyesuaikan tingkat kompleksitas jalur komunikasi yang diuji. Seluruh pengujian, baik pada implementasi REST API maupun gRPC, menerapkan *Deadline Timeout* sebesar 15.000 milidetik dan *Constant Timer* sebesar 1.000 milidetik antarpermintaan agar beban *server* lebih realistis dan stabil selama pengujian berlangsung.

2.6.1 Parameter Pengujian

Durasi Pengujian 180 Detik. Durasi pengujian 180 detik (3 menit) dipilih berdasarkan standar penelitian akademis untuk *inter-process communication testing* pada sistem *microservices*. Shafabakhsh et al. (2020) dalam evaluasi *inter-process communication* pada arsitektur *microservices* menggunakan durasi pengujian konstan 180 detik untuk memastikan pengumpulan data yang stabil sambil mengobservasi perilaku berbagai teknik IPC (REST API, gRPC, dan RabbitMQ) di bawah variasi beban pengguna. Sebagai standar industri, NGINX juga mengadopsi periode 180 detik untuk pengukuran *throughput* HTTP (NGINX, 2017). Durasi ini memungkinkan sistem mencapai *steady-state performance* sebelum pengukuran metrik dilakukan, sehingga hasil pengujian mencerminkan perilaku sistem dalam skenario operasional normal pada sistem *Smart Parking*.

Variasi Jumlah Pengguna: 5, 10, 15, 20, 25 Pengguna. 5, 10, 15, 20, 25. Rentang ini diambil dari observasi lapangan: jumlah *gate* parkir di Makassar bervariasi dari fasilitas kecil (2 *gate*: Hotel *Four Points*, Apartemen Sudirman) hingga besar (23 *gate*: TSM *Mall*), dengan contoh menengah 4 gate (RS Siloam, Wisma Kalla) dan 15 *gate* (*Mall* Nipah). Karena rentang nyata 2–23 *gate*, dipilih lima level beban: minimal (5) untuk fasilitas kecil, moderat (10–15) untuk fasilitas menengah, dan tinggi (20–25) untuk fasilitas besar, agar skenario uji tetap relevan pada ekosistem parkir yang heterogen. Berikut keterkaitan dengan tiap jalur komunikasi sistem:

Pada jalur *Service AI 1* ke *Service AI 2*, *Service AI 1* beroperasi di sisi *edge* pada tiap *gate* parkir untuk melakukan deteksi secara langsung, lalu mengirim hasilnya ke *Service AI 2*. Dengan demikian, rentang beban 5–25 pengguna merepresentasikan kondisi 5–25 kamera atau gate aktif yang mengirim data ke *Service AI 2* secara simultan.

Pada jalur *Service AI 2* ke *Service Main* yang bersifat terpusat, setiap penyedia parkir menjalankan *Service AI 2* di lokasi masing-masing dan dapat mengirim hasil ke pusat pada waktu yang bersamaan. Mengingat jumlah penyedia di Kota Makassar yang cukup banyak, beban 5–25 pengguna tetap realistis untuk menggambarkan kiriman serentak dari berbagai *site* menuju *Service Main*.

Pada jalur *Service Dashboard* ke *Service Main*, dashboard diakses oleh admin pusat maupun admin dari masing-masing penyedia. Karena banyak penyedia berpotensi membuka dashboard secara bersamaan, penggunaan beban 5–25 pengguna tetap sesuai dengan variasi jumlah akses yang mungkin terjadi di kondisi operasional sebenarnya. Dengan demikian, kelima tingkat beban tersebut diterapkan secara seragam pada setiap jalur pengujian agar perbandingan performa antarprotokol (*REST* dan *gRPC*) dilakukan dalam kondisi yang setara, sekaligus tetap merefleksikan situasi nyata yang ditemukan pada lingkungan operasional.

***Ramp-up Period* 5-25 Detik.** Pemilihan nilai *ramp-up period* ditentukan dengan formula proporsional: jumlah pengguna target = durasi *ramp-up* dalam detik, menghasilkan *user arrival rate* 1 pengguna/detik yang konsisten. Ini membuat pengujian lebih realistis dan menghindari *overload* mendadak pada sistem (Rak et al., 2025).

***Deadline Timeout* 15.000 Milidetik.** Pemilihan nilai *deadline timeout* sebesar 15.000 milidetik ini disesuaikan dengan nilai yang banyak digunakan pada sistem *microservices* modern, seperti *Envoy Proxy* dan AWS *App Mesh*, di mana batas waktu 15 detik dianggap sebagai nilai optimal untuk menangani latensi jaringan dan proses *backend* tanpa mengorbankan *responsiveness* dan *availability* sistem (Envoy Proxy, 2023; AWS App Mesh, 2023).

***Constant Timer* 1.000 Milidetik Antar *Request*.** Pemilihan nilai *constant timer* 1.000 milidetik bertujuan agar *request* tidak dikirim sekaligus, melainkan dengan jeda 1 detik antar*request*. Ini membuat beban pada sistem menjadi lebih teratur dan realistis, sehingga *server* sempat mengembalikan sumber daya sebelum *request* berikutnya datang (Grafana, 2023). Jeda 1 detik ini diterapkan di semua jalur komunikasi (*Service* AI 1 ke *Service* AI 2, *Service* AI 2 ke *Service Main*, *Service Dashboard* ke *Service Main*) untuk memastikan kondisi pengujian yang sama.

2.6.2 Skenario Pengujian

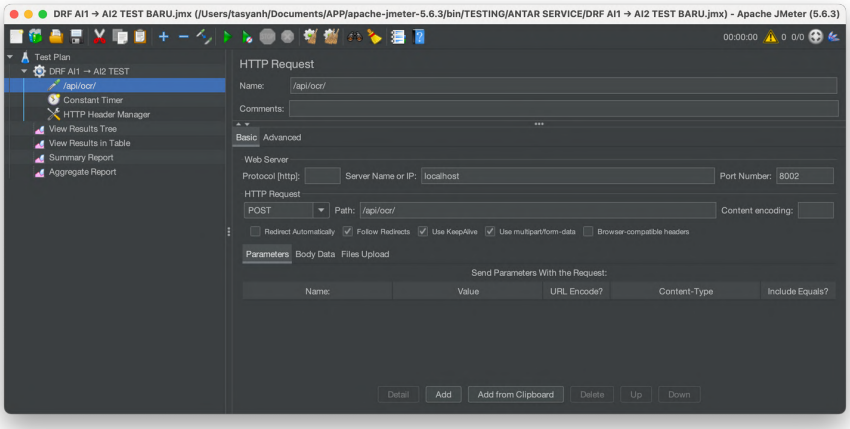
***Service* AI 1 dan *Service* AI 2.** Pengujian ini menggunakan empat variasi video mentah sebagai input, yaitu 1080p 30 FPS, 1080p 60 FPS, 4K 30 FPS, dan 4K 60 FPS. Pengujian komunikasi antara *Service* AI 1 dan *Service* AI 2 dilakukan dengan mengirimkan hasil deteksi kendaraan dari masing-masing video untuk dilakukan proses *Optical Character Recognition* (*OCR*) terhadap pelat nomor kendaraan. Pada implementasi REST API, pengujian dilakukan dengan mengirim permintaan ke endpoint */api/ocr/* menggunakan metode POST. Data yang dikirim terdiri atas berkas citra kendaraan (*frame.jpg*) serta berkas JSON yang berisi hasil deteksi objek. Seluruh data dikirim dalam format *multipart/form-data* melalui *HTTP Request Sampler* pada *Apache JMeter*. Sementara itu, pada implementasi gRPC, pengujian dilakukan terhadap *method ai2.OCRService/RecognizeFromFrame* yang menerima dua parameter utama,

yaitu *frame_data* dan *json_data*, keduanya dikirim dalam bentuk *Base64 encoded string*. Pengujian ini bertujuan untuk menilai kecepatan dan efisiensi proses pengiriman data berukuran besar antarservice, serta mengamati bagaimana performa gRPC dalam menangani *payload* biner dibandingkan REST API yang berbasis JSON. Dengan skenario ini, dapat diketahui seberapa cepat sistem mampu memproses data citra dan mentransfernya antarservice dalam kondisi beban pengguna yang berbeda.

Service* AI 2 dan *Service Main. Pengujian antara *Service* AI 2 dan *Service Main* dilakukan untuk mengukur kinerja sistem dalam tahap perekaman data hasil OCR ke dalam basis data. Pada implementasi REST API, pengujian difokuskan pada endpoint */api/process-plate/* dengan metode POST, yang digunakan untuk mengirim data hasil identifikasi kendaraan, meliputi *plate_number*, *vehicle_type*, *timestamp*, dan *location_id*. Nilai *timestamp* dihasilkan secara dinamis menggunakan *Groovy Script* di *Apache JMeter* dengan format ISO 8601 dan zona waktu *Asia/Makassar* agar sinkron dengan zona operasional sistem. Pada implementasi gRPC, pengujian dilakukan terhadap *method plate.PlateService/ProcessPlate* yang memiliki parameter serupa dan dilengkapi sistem otorisasi antarservice melalui *metadata Bearer token*. Jalur ini berperan penting untuk mengamati performa sistem dalam menangani proses tulis data (*write-heavy operation*), termasuk pencatatan kendaraan masuk dan keluar, perhitungan durasi parkir, serta estimasi biaya parkir berdasarkan waktu penggunaan. Melalui pengujian ini, dapat dinilai kemampuan REST API dan gRPC dalam menangani transaksi yang melibatkan pemrosesan data waktu nyata serta penyimpanan data yang konsisten ke basis data.

Service* Dashboard dan *Service Main. Pengujian terakhir dilakukan antara *Service Dashboard* dan *Service Main*, yang merepresentasikan interaksi antara antarmuka pengguna (*dashboard frontend*) dengan sistem *backend*. Tujuan pengujian ini adalah untuk mengevaluasi performa komunikasi dalam proses pengambilan data (*read-heavy operation*), seperti statistik pendapatan, jumlah kendaraan, dan daftar transaksi. Pada implementasi REST API, pengujian mencakup serangkaian permintaan terhadap beberapa endpoint utama, di antaranya POST */api/auth/token/* untuk autentikasi pengguna, serta GET */api/dashboard/summary/*, */api/dashboard/income/*, */api/dashboard/vehicles/*, dan */api/dashboard/table/* untuk mengambil berbagai data yang ditampilkan di *dashboard*. Sementara itu, pada implementasi gRPC, pengujian dilakukan terhadap *method* dengan fungsi serupa, yaitu *auth.AuthService/Login* untuk login pengguna, serta *plate.PlateService/GetSummary*, *GetIncomeStats*, *GetVehicleStats*, dan *GetVehicleTable* untuk menampilkan data statistik kendaraan dan pendapatan harian.

2.6.3 Prosedur Pengujian



Gambar 12. Alat *Apache Jmeter*

Prosedur pengujian pada penelitian ini dilakukan secara sistematis untuk memastikan hasil pengukuran kinerja sistem dapat diperoleh secara objektif, terukur, dan dapat direplikasi. Dapat dilihat pada Gambar 12. Alat *Apache Jmeter,* seluruh proses pengujian dilakukan menggunakan *Apache JMeter* versi 5.6.3, yang berfungsi sebagai alat utama untuk melakukan simulasi permintaan (*request*) dari sejumlah pengguna secara bersamaan terhadap layanan REST API dan gRPC. Tujuan utama dari prosedur ini adalah untuk mengukur seberapa baik performa masing-masing protokol dalam menangani beban kerja yang sama pada berbagai jalur komunikasi dalam sistem *Smart Parking*.

Langkah pertama adalah persiapan lingkungan pengujian. Semua layanan yang terlibat dalam sistem, yaitu *Service* AI 1, *Service* AI 2, *Service Main*, dan *Service Dashboard*, dijalankan secara lokal pada perangkat *MacBook Air* (*Apple M3*, 8-core CPU, 16 GB RAM) menggunakan *virtual environment* yang terpisah untuk setiap layanan. Seluruh layanan dikonfigurasi agar saling terhubung melalui port tertentu sesuai dengan protokol komunikasi masing-masing. Selain itu, basis data PostgreSQL diinisialisasi ulang sebelum setiap pengujian dimulai untuk menghindari pengaruh sisa data dari uji sebelumnya, sehingga setiap pengujian berjalan dalam kondisi awal yang sama (*clean state*).

Tahap kedua adalah konfigurasi *Apache JMeter*. Pada pengujian REST API, digunakan *HTTP Request Sampler* untuk melakukan permintaan ke endpoint yang telah ditentukan, sementara *HTTP Header Manager* digunakan untuk menambahkan atribut seperti Authorization: Bearer <token> dan Content-Type: application/json. Untuk menjaga konsistensi waktu pada setiap pengujian, *JSR223 PreProcessor* diaktifkan untuk menghasilkan *timestamp* secara dinamis menggunakan *Groovy Script* dengan zona waktu *Asia/Makassar* (WITA). Selain itu, *Constant Timer* disetel sebesar 1.000 milidetik untuk memberi jeda antarpermintaan, dan *Deadline Timeout* sebesar 15.000 milidetik diterapkan

agar setiap permintaan memiliki batas waktu pemrosesan yang seragam. Pada pengujian gRPC, digunakan *gRPC Request Sampler* yang terhubung dengan berkas definisi *.proto* milik setiap layanan. Komponen *JSR223 PreProcessor* digunakan untuk membaca berkas gambar dan JSON dari direktori lokal, kemudian mengubahnya ke format *Base64* sebelum dikirim. Sementara itu, *PostProcessor* digunakan untuk membantu mengekstraksi/merekam nilai metrik *Sent KB/sec* dan *Received KB/sec* dari hasil pengujian untuk diolah di tahap analisis.

Langkah ketiga adalah pelaksanaan pengujian beban (*load testing*). Pada tahap ini, *Apache JMeter* menjalankan simulasi pengguna berdasarkan parameter yang telah ditetapkan sebelumnya, yaitu jumlah pengguna sebanyak 5, 10, 15, 20, dan 25, dengan durasi pengujian selama 180 detik dan *ramp-up period* bervariasi antara 5 hingga 25 detik. *JMeter* secara otomatis mencatat metrik performa seperti *average response time*, *throughput*, *sent KB/sec*, *received KB/sec*, dan *error rate*. Selama proses ini berlangsung, *resource utilization* seperti penggunaan CPU dan memori juga diamati untuk memberikan gambaran menyeluruh mengenai beban sistem.

Tahap keempat adalah perekaman dan pengolahan data hasil pengujian. Seluruh hasil uji terekam secara otomatis pada *Summary Report* dan *Aggregate Report* di *Apache JMeter*. Data mentah tersebut kemudian diekspor ke *Google Spreadsheet* untuk dilakukan pengolahan lanjutan dan perhitungan statistik. Dari setiap pengujian, diambil nilai rata-rata dari tiga kali pengulangan untuk masing-masing metrik utama. Hal ini dilakukan agar hasil yang diperoleh tidak dipengaruhi oleh anomali sementara, seperti fluktuasi performa perangkat atau kondisi jaringan lokal. Setelah itu, data yang telah diolah diubah ke dalam bentuk tabel dan grafik untuk memudahkan analisis perbandingan antara REST API dan gRPC.

Langkah terakhir adalah analisis dan evaluasi hasil pengujian. Analisis dilakukan dengan membandingkan performa kedua protokol komunikasi berdasarkan lima metrik utama, yaitu *response time*, *throughput*, *resource utilization*, *network data rate*, dan *error rate*. Metrik *response time* digunakan untuk mengukur seberapa cepat sistem dalam merespons setiap permintaan. *Throughput* menunjukkan banyaknya permintaan yang dapat diproses per detik, sementara *resource utilization* memperlihatkan efisiensi pemakaian CPU dan memori saat pengujian berlangsung. Metrik *network data rate* merepresentasikan laju transfer data pada jaringan selama pengujian (misalnya Sent KB/sec dan Received KB/sec), sehingga menunjukkan beban/konsumsi bandwidth akibat komunikasi antarservice. Sementara itu, *error rate* digunakan untuk mengukur tingkat kestabilan sistem terhadap beban tinggi. Hasil evaluasi dari seluruh metrik tersebut kemudian dibahas secara mendalam pada Bab III Hasil dan Pembahasan, guna menentukan protokol komunikasi yang paling efisien dan sesuai untuk mendukung arsitektur *microservices* pada sistem *Smart Parking* ini.