RANCANG BANGUN MIDDLEWARE LARAVEL BERBASIS REDIS STREAM DENGAN ARSITEKTUR LOAD-BALANCING

SKRIPSI

YUSUF RAFIF KARBACK

2107421021

PROGRAM STUDI TEKNIK MULTIMEDIA DAN JARINGAN

JURUSAN TEKNIK INFORMATIKA DAN KOMPUTER

POLITEKNIK NEGERI JAKARTA

TAHUN 2025

RANCANG BANGUN MIDDLEWARE LARAVEL BERBASIS REDIS STREAM DENGAN ARSITEKTUR LOAD-BALANCING

SKRIPSI

Dibuat untuk Melengkapi Syarat-Syarat yang Diperlukan untuk Memperoleh Diploma Empat Politeknik

YUSUF RAFIF KARBACK
2107421021

PROGRAM STUDI TEKNIK MULTIMEDIA DAN JARINGAN
JURUSAN TEKNIK INFORMATIKA DAN KOMPUTER
POLITEKNIK NEGERI JAKARTA
2025

SURAT PERNYATAAN BEBAS PLAGIARISME

Saya yang bertanda tangan dibawah ini:

Nama	:	Yusuf Rafif Karback
NIM	:	2107421021
Jurusan/Program Studi	:	Teknik Informatika dan Komputer / Teknik Multimedia dan Jaringan
Judul Skripsi	:	Rancang Bangun *Middleware* Laravel Berbasis Redis *Stream* Dengan Arsitektur *Load-Balancing*

Menyatakan dengan sebenarnya bahwa skripsi ini benar-benar merupakan hasil karya saya sendiri, bebas dari peniruan terhadap karya dari orang lain. Kutipan pendapat dan tulisan orang lain ditunjuk sesuai dengan cara-cara penulisan karya ilmiah yang berlaku.

Apabila di kemudian hari terbukti atau dapat dibuktikan bahwa dalam skripsi ini terkandung ciri-ciri plagiat dan bentuk-bentuk peniruan lain yang dianggap melanggar peraturan, maka saya bersedia menerima sanksi atas perbuatan tersebut.

Depok, 5 Juni 2025

Yang Membuat Pernyataan

Yusuf Rafif Karback

NIM. 2107421021

LEMBAR PENGESAHAN

Skripsi diajukan oleh:

Nama	:	Yusuf Rafif Karback
NIM	:	2107421021
Program Studi	:	Teknik Multimedia dan Jaringan
Judul Skripsi	:	Rancang Bangun *Middleware* Laravel Berbasis Redis *Stream* Dengan Arsitektur *Load-Balancing*

Telah diuji oleh tim penguji dalam Sidang Skripsi pada hari Rabu Tanggal 2 Bulan
Juli Tahun 2025, dan dinyatakan **LULUS**.

Disahkan Oleh

Pembimbing I	Defiana Arnaldy, S.Tp., M.Si.	( )
Penguji I	Dr. Indra Hermawan., M.Kom	( )
Penguji II	Fachroni Arbi Murad, S.Kom., M.Kom.	( )
Penguji III	Iik Muhamad Malik Matin, S.Kom., M.T.	( )

Mengetahui:
Jurusan Teknik Informatika dan
Komputer Ketua

Dr. Anita Hidayati, S.Kom., M.Kom.
NIP. 197908032003122003

SURAT PERNYATAAN PERSETUJUAN PUBLIKASI SKIRPSI UNTUK KEPENTINGAN AKADEMIS

Sebagai sivitas akademik Politeknik Negeri Jakarta, saya bertanda tangan dibawah ini:

Nama	:	Yusuf Rafif Karback
NIM	:	2107421021
Jurusan/Program Studi	:	Teknik Informatika dan Komputer / Teknik Multimedia dan Jaringan

Demi pengembangan ilmu pengetahuan, menyetujui untuk memberikan kepada Politeknik Negeri Jakarta Hak Bebas Royalti Non-Eksklusif atas karya ilmiah saya yang berjudul:

Rancang Bangun Middleware Laravel Berbasis Redis Stream Dengan Arsitektur Load Balancing

Beserta perangkat yang ada (jika diperlukan). Dengan Hak Bebas Royalti NonEksklusif ini Politeknik Negeri Jakarta Berhak menyimpan, mengalihmediakan/formatkan, mengelola dalam bentuk pangkalan data (database), merawat, dan mempublikasikan skripsi saya tanpa meminta izin dari saya selama tetap mencantumkan nama saya sebagai penulis/pencipta dan sebagai pemilik Hak Cipta. Demikian pernyatan ini saya buat dengan sebenarnya.

Depok, 5 Juni 2025
Yang Membuat Pernyataan

Yusuf Rafif Karback
NIM. 2107421021

KATA PENGANTAR

Puji syukur penulis panjatkan ke hadirat Tuhan Yang Maha Esa, karena berkat rahmat dan karunia-Nya penulis dapat menyelesaikan skripsi yang berjudul "Rancang Bangun *Middleware* Berbasis Redis *Stream* Dengan Arsitektur *Master-Slave*" ini. Skripsi ini disusun untuk memenuhi salah satu syarat meraih gelar Sarjana Komputer pada Program Studi Teknik Multimedia dan Jaringan, Jurusan Teknik Informatika dan komputer, Politeknik Negeri Jakarta. Dalam proses penyusunan skripsi ini, penulis banyak mendapatkan bimbingan, masukan, dan dukungan dari berbagai pihak. Oleh karena itu, pada kesempatan ini penulis ingin mengucapkan terima kasih dan penghargaan yang setinggi-tingginya kepada:

1. Tuhan Yang Maha Esa yang selalu memberikan hikmat dan rahmatnya dalam menyelesaikan Tugas Akhir.
2. Kepada Bapak Defiana Arnaldy, S.Tp., M.Si., selaku Pembimbing, yang telah memberikan arahan, saran, dan koreksi yang sangat berharga.
3. Seluruh keluarga tercinta yang selalu memberikan doa, dukungan moral, dan kesabaran selama penulis menempuh studi.
4. Kepada Janet Tricia, Seluruh sahabat konjep, dan teman-teman TMJ yang selalu memberi dukungan dan doa selama menempuh studi.

Penulis menyadari bahwa skripsi ini masih jauh dari sempurna. Oleh karena itu, kritik dan saran yang membangun sangat di harapkan demi kesempurnaan penelitian selanjutnya. Semoga skripsi ini dapat memberikan kontribusi positif bagi pengembangan ilmu dan praktik rekayasa perangkat lunak, khususnya pada bidang sistem terdistribusi dan teknologi streaming data.

Akhir kata, semoga karya ini bermanfaat bagi pembaca sekalian.

Depok, Juni 2025

Yusuf Rafif Karback

RANCANG BANGUN MIDDLEWARE LARAVEL BERBASIS REDIS STREAM DENGAN ARSITEKTUR LOAD-BALANCING

ABSTRAK

Menyebarkan data dari satu aplikasi ke beberapa database merupakan hal yang memakan banyak sumber daya dan rentan terhadap kegagalan. Ketika database target menjadi offline, pendekatan konvensional akan menghentikan atau menunda transaksi tanpa batas waktu, sehingga membahayakan konsistensi. Oleh karena itu, penelitian ini mengusulkan lapisan middleware yang secara asinkron merutekan data ke setiap tujuan sambil menjaga integritas transaksi selama periode tidak tersedianya database. Teknologi yang digunakan untuk penelitian ini adalah Redis, Google Cloud, Laravel, PHP, dan Visual Studio Code. Hasil dari penelitian ini yaitu berhasil mendistribusikan data ke database tujuan serta mengimplementasi load-balancing dengan menyelesaikan 1000 permintaan dalam 29,119 detik, throughput server tercatat sebesar 34,34 permintaan per detik dan konsumsi ram untuk redis server dalam uji skenario normal sebesar 5,9% dan uji partial-failure sebesar 7%.

Kata Kunci: *middleware, Redis, Redis stream, load-balancing, Google Cloud*

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

BAB I
PENDAHULUAN

1.1 LATAR BELAKANG

Dalam perkembangan teknologi informasi, sistem terdistribusi menjadi arsitektur yang semakin banyak digunakan untuk meningkatkan efisiensi, skalabilitas, dan keandalan dalam pengolahan data. Salah satu tantangan utama dalam sistem terdistribusi adalah distribusi data yang efisien dan andal. Dalam banyak skenario, aplikasi yang berjalan dalam sistem terdistribusi harus memastikan bahwa data yang dikirim dapat diterima oleh setiap node dengan latensi rendah, keandalan tinggi, dan toleransi terhadap kegagalan (fault tolerance).

Sistem terdistribusi adalah sekumpulan sistem komputer yang terhubung melalui jaringan dan berkomunikasi menggunakan *middleware* atau *message broker*. Salah satu karakteristik utama sistem ini adalah kemampuannya untuk menangani proses secara bersamaan (konkurensi) dan memastikan kelangsungan operasional meskipun salah satu komponennya mengalami kegagalan. Namun, tantangan seperti sinkronisasi waktu global dan penanganan kegagalan antar sistem menjadi fokus utama dalam pengembangannya (Pratama, 2021).

Salah satu *message broker* yang dapat digunakan untuk berkomunikasi antar sistem adalah Redis. Redis adalah *in-memory database* bertipe *key-value* yang mendukung penyimpanan berbagai tipe data (Zhu et al., 2023). Salah satu tipe data yang dipunyai Redis adalah *stream,* Redis *stream* diperkenalkan pada Redis 5.0, yang dapat *publish* data ke s*tream*, dan *consumer* yang *subscribe* ke *stream* tersebut dapat menerimanya. Keuntungan utama dibandingkan model pub/sub adalah bahwa Redis *stream* mempertahankan pesan. Oleh karena itu, ia dapat menjamin pengiriman pesan yang tepat (Weerasinghe and Perera, 2023).

Untuk kinerja redis sebagai *message broker* berdasarkan penelitian yang berjudul Perbandingan Kinerja Redis, Mosquitto, dan MongoDB sebagai *Message Broker* pada *IoT Middleware*, peneliti melakukan pengujian untuk mengirim data

menggunakan Redis sebagai *message broker* bertujuan untuk mengetahui penggunaan CPU, *memory*, disk i/o, *runtime*, dan skalabilitas masing-masing *message broker* saat menangani banyak data saat proses *publish* dan *subscribe*. Hasil dari penelitian tersebut adalah Redis memiliki penggunaan CPU, memori, dan *runtime* terbaik saat menulis data. Nilai penggunaan CPU saat menulis data adalah 13%–41%, penggunaan memori saat menulis dan membaca data adalah 53MB–64MB, dan 54MB–59MB, dan *runtime* saat menulis dan membaca data adalah 2,37 detik dan 0,05 detik (Febriyani, Pramukantoro, and Bakhtiar, 2019).

Berdasarkan penelitian yang berjudul Implementasi Pengiriman Pesan Broadcast dengan Redis Pub/Sub dan Bahasa Pemrograman Nim, peneliti melakukan pengiriman data menggunakan Redis *Pub/Sub* yang bertujuan untuk untuk mengirimkan pesan sekaligus ke banyak penerima secara *realtime.* Hasil dari penelitian tersebut adalah menunjukkan bahwa proses pengiriman pesan dengan Redis Pub/Sub sangat efektif, dengan 86% responden menyatakan "sangat setuju" (Rachel and Susetyo). Kedua penelitian tersebut memiliki kekurangan yaitu tidak ada penjelasan mengenai status data yang dikirim oleh Redis apabila *subscriber* yang dituju sedang dalam keadaan mati atau *down* dan tidak ada *backup* apabila server Redis tidak mampu menangani *message* dari *publisher* dalam jumlah banyak.

Saat ini pada PT.Bangun Abadi Teknologi Indonesia (BATI) memiliki beberapa aplikasi yang terintegrasi melalui *Application Programming Interface* (API). Permasalahan yang terjadi adalah jika aplikasi WHMCS internal akan mendistribusikan data kepada 5 *database* yang saat ini *existing* dengan cara memanggil *endpoint* API, jika aplikasi internal harus memanggil beberapa *endpoint* maka akan memakan waktu dan sumber daya. Permasalahan yang kedua adalah jika aplikasi internal ingin mendistribusikan data ke *database* tetapi server dari *database* tersebut sedang dalam keadaan mati atau *down* sehingga data tidak diterima oleh *database* tersebut.

Berdasarkan permasalahan pada studi kasus yang telah diuraikan, diusulkan sebuah penelitian yang berfokus pada perancangan aplikasi *middleware* berbasis *website* menggunakan Redis *Stream* dengan penerapan arsitektur *load-balancing*. Pada

penelitian ini, akan dirancang sebuah sistem *middleware* yang memanfaatkan *Redis Stream* untuk menyimpan dan mendistribusikan data ke *server database* tujuan. Sistem ini dirancang agar mampu menyimpan data sementara di dalam *memory* apabila *server* tujuan mengalami gangguan (*down*), dan secara otomatis mengirimkan kembali data tersebut ketika *server* kembali tersedia (*up*). Selain itu, arsitektur sistem akan dibangun dengan *load-balancing* guna mendukung ketersediaan sistem yang tinggi (*high availability*) serta menjamin kontinuitas proses distribusi data.

1.2 RUMUSAN MASALAH

A. Bagaimana hasil penerapan *load-balancing* untuk menunjang *high availability*?.
B. Bagaimana hasil *resource* yang digunakan oleh tipe data *stream* sebagai cara menyimpan data di Redis *server*?
C. Apakah data tetap tersimpan di Redis server apabila *database* yang dituju dalam keadan *down?*
D. Apakah aplikasi *middleware* dapat mendistribusikan data dengan lengkap?

1.3 BATASAN MASALAH

A. Penerapan tipe data *stream* untuk mengirim data ke *server* Redis.
B. Sistem hanya difokuskan pada pengelolaan data yang bersifat *write* (tulis) dari *producer* dan pengiriman ke database tujuan.
C. Menggunakan 2 VM sebagai *server middleware.*
D. Menggunakan layanan *Load Balancing* dari *Google Cloud.*
E. Menggunakan 5 *database* server sebagai *database* tujuan.
F. Data dikirim oleh *producer* menggunakan metode HTTP.
G. *Load balancing* akan diterapkan untuk membagi trafik secara merata antar VM Redis *middleware*.

1.4 TUJUAN DAN MANFAAT

1.4.1 Tujuan

A. Menerapkan tipe data *stream* untuk mengirim data dari server Redis ke *database* tujuan
B. Menerapkan *load balancing* untuk menunjang *high availability* pada server *middleware*
C. Menjaga agar data tetap tersimpan di dalam server Redis apabila database yang akan dituju dalam keadan *down* dan akan mengirim ulang data tersebut apabila *database* tujuan sudah dalam keadan *up.*

1.4.2 Manfaat

A. Penyebaran beban kepada VM *middleware* secara merata.
B. Konsistensi dan integritas data terjaga.
C. Pengurangan kehilangan data (*data loss*).

BAB II
TINJAUAN PUSTAKA

2.1 Penelitian Terdahulu

Berikut adalah tabel penelitian terdahulu dan kebaruan yang akan dilakukan oleh peneliti

Tabel 2.1 Penelitian Terdahulu

No	Penelitian	Hasil	Perbedaan
1	Implementasi Pengiriman Pesan Broadcast dengan Redis Pub/Sub dan Bahasa Pemrograman Nim (Rachel and Susetyo).	Hasil dari penelitian tersebut adalah menunjukkan bahwa proses pengiriman pesan dengan Redis Pub/Sub sangat efektif, dengan 86% responden menyatakan "sangat setuju"	Menggunakan Redis stream untuk mengirim data
2	Perbandingan Kinerja Redis, Mosquitto, dan MongoDB sebagai *Message Broker* pada *IoT Middleware*(Febriyani, Pramukantoro, and Bakhtiar, 2019).	Redis memiliki penggunaan CPU, memori, dan *runtime* terbaik saat menulis data. Nilai penggunaan CPU saat menulis data adalah 13%–41%, penggunaan memori saat menulis dan membaca data adalah 53MB–64MB, dan	Menambahkan *load balancing* untuk menunjang *high availability*

		54MB–59MB, dan *runtime* saat menulis dan membaca data adalah 2,37 detik dan 0,05 m detik	
3	IMPLEMENTASI SISTEM BROADCAST MESSAGE MENGGUNAKAN PYTHON DAN REDIS PUB/SUB (Claudiyap, Ocsa, and Saian, 2022).	Hasil dari penelitian tersebut menunjukkan bahwa proses pengiriman pesan dengan Redis Pub/Sub dapat dilakukan dengan bahasa pemrograman Python dengan hasil UAT dengan indeks rata-rata 79%	Menggunakan bahasa pemrograman PHP dan *framework* Laravel
4	Distributed MQTT Broker: A Load-Balanced Redis-based Architecture (Doshi et al., 2024).	Redis stream digunakan untuk menyalurkan pesan-pesan yang dipublikasikan di antara para broker untuk pengiriman pesan yang konsisten kepada *subscriber* yang dituju. Pendekatan distribusi *packet-level* ini terbukti efisien dalam mengimplementasikan broker MQTT.	Menggunakan protokol HTTP dan API untuk mengirim data dari aplikasi menuju redis server

5	Optimized Strategy for Inter-Service Communication in Microservices (Weerasinghe and Perera, 2023).	Mengurangi *latency* saat berkomunikasi antar *services* menggunakan Redis *stream.* Saat mengirim data akan di serialisasi dan dikirim sebagai *protocol buffer.*	Mencatat log data yang dikirim apakah sukses atau pending
6	A Performance Evaluation of In-Memory Databases Operations in Session Initiation Protocol	Memcached memakai memori 6,61 %—hanya 0,56 % lebih besar saat menulis dibanding membaca—serta menjadi opsi bagus bila dibutuhkan persistensi ringan. sedangkan Redis memakan memori sedikit lebih besar daripada Memcached, tetapi menawarkan struktur data canggih dan fitur lebih kaya sehingga cocok untuk kebutuhan lanjutan.	Menggunakan Apache Benchmark untuk menguji *load-balancing*

2.2 Redis

Redis adalah penyimpan struktur basis data dalam memori yang sudah banyak digunakan dan dapat diakses secara lokal atau di seluruh jaringan serta pilihan yang

sangat baik untuk "*cache data*" (Eddelbuettel, 2022). Database NoSQL Redis menggunakan sistem *in-memory* untuk pengambilan data dan menyimpan data dalam bentuk pasangan *key-value*. Akibatnya, Redis lebih cepat membaca data daripada *database* NoSQL lainnya. Dikenal sebagai database, Redis juga dapat berfungsi sebagai *broker* pesan dan *cache*. Mekanisme kerja Redis sebagai penyedia pesan menggunakan model *publish-subscribe*, yaitu berbasis topik (Febriyani, Pramukantoro, and Bakhtiar, 2019). Redis juga mendukung beberapa tipe data seperti *string, hashes, sets, lists, sorted sets* serta 16 tipe data lainnya (Eddelbuettel, 2022).

2.3 Middleware

Pada bidang sistem terdistribusi, kerangka kerja *middleware* untuk pemrosesan aliran data telah dikembangkan guna memfasilitasi analisis *real-time* terhadap data heterogen di lingkungan *big data*. Dipaparkan bahwa sebuah *distributed stream processing middleware framework* bernama ESTemd diperkenalkan untuk integrasi dan interoperabilitas data sensor warisan dan data big data menggunakan Apache Kafka dan Kafka Streams (Akanbi & Masinde, 2020). *Framework* tersebut diuji dengan studi kasus prediksi kekeringan, yang menunjukkan bahwa arsitektur *middleware* ini mampu mentransformasi dan memproses data secara kontinu dengan latensi rendah dan throughput tinggi melalui pendekatan *publish/subscribe* pada tingkat lapisan middleware (Akanbi & Masinde, 2020). Dengan demikian, ESTemd mendemonstrasikan peran penting *middleware* dalam memfasilitasi pengolahan data waktu-nyata dalam sistem terdistribusi modern.

2.4 Load-Balancing

Dalam sistem terdistribusi, load balancing didefinisikan sebagai proses mendistribusikan permintaan atau beban kerja secara merata ke beberapa node untuk mencegah kemacetan dan memastikan utilisasi sumber daya yang optimal. Tujuannya adalah untuk meningkatkan reliabilitas, efisiensi, dan respon waktu dalam infrastruktur heterogen (Shahakar et al., 2023). Dengan demikian, load balancing memungkinkan sistem untuk mencapai ketersediaan tinggi dan

scalability, karena beban yang tidak merata dapat menyebabkan bottleneck dan degradasi performa .Lebih lanjut, load balancing digambarkan berfungsi sebagai mekanisme adaptif yang secara dinamis memantau kondisi sistem—seperti utilisasi CPU, latensi jaringan, dan throughput—serta mengatur distribusi beban untuk menanggapi perubahan real-time (Liu et al., 2025).

2.5 Laravel

Versi awal *framework* Laravel dirilis oleh Taylor Otwel pada 9 Juni 2011 dengan tujuan menambahkan fitur-fitur untuk membangun *website* dari *framework* yang sudah ada sebelumnya yaitu Codeigniter. Tahun 2019 Laravel merilis versi 6 dan menunjukan fungsi yang sangat berguna yang digabung dengan *design pattern* MVC*(Model View Controller)*. Fungsi-fungsi tersebut termasuk: *routing* yang cepat dan sederhana ,*dependency injection*, *database* ORM(*Object Relational Mappin*g) yang intuitif, pekerjaan pemrosesan *background* yang kuat, *scheduling* server tunggal, dan *real-time event broadcasting*. Laravel adalah *framework* yang sesuai dengan pola desain MVC. Ini menempatkan kedua lapisan View dan Controller di dalam direktori "Resources" dan "Routes". Selain itu, *framework* Laravel menetapkan direktori "Database" yang digunakan untuk operasi data di lapisan model. Untuk menyederhanakan proses pengujian, kerangka kerja Laravel juga menetapkan direktori "Publik" (Hsieh et al., no date).

2.6 Database

Terdapat dua jenis *database* yang umum digunakan yaitu SQL dan NoSQL. SQL dan NoSQL berbeda dalam hal atribut, penerapan, dan metode penyimpanannya. Basis data ini juga memiliki perbedaan dalam hal kinerja dan jenis fitur yang dimiliki masing-masing, sehingga cocok untuk tugas-tugas tertentu. Basis data relasional menyimpan data dalam baris dan tabel, yang berbeda secara signifikan dari NoSQL. NoSQL menggunakan beberapa struktur, dan diantaranya berorientasi pada kolom. Dalam jenis struktur penyimpanan ini, data sering disimpan di dalam sel yang dikelompokkan dalam jumlah kolom yang hampir tak terbatas, bukan baris. Mode penyimpanan lainnya adalah penyimpanan nilai kunci. Sebagai bagian

dari model data, ini adalah asosiasi array, juga disebut disebut juga sebagai *map* atau *dictionary*. Penyimpanan dokumen juga menggunakan dokumen untuk menangani data, seperti data yang dikodekan dalam format standar. Format yang umum termasuk XML dan JSON. Manfaat mengadopsi penyimpanan dokumen adalah mendukung penggunaan tipe data yang berbeda di dalam satu basis data (Muniswamaiah, Agerwala, and Tappert, 2023).

2.7 Apache Benchmark

Apache Benchmark telah dimanfaatkan sebagai alat pengujian beban yang sederhana namun efektif untuk mengevaluasi kinerja server dalam berbagai konfigurasi sistem terdistribusi. Alat ini digunakan untuk mengukur metrik performa seperti *requests per second*, *time per request*, dan *transfer rate*, yang sangat relevan dalam penilaian daya tahan sistem saat menerima trafik tinggi. Apache Benchmark digunakan untuk membandingkan performa server Apache yang dijalankan dalam lingkungan Docker dan native. Hasil pengujian menunjukkan bahwa perbedaan throughput dan latensi dapat diukur secara akurat menggunakan Apache Benchmark, dan alat ini mampu memetakan batas kapasitas server dalam skenario simulasi trafik HTTP (Saputra et al., 2023).

2.8 PDO (*php data object)*

PHP Data Objects (PDO) adalah ekstensi PHP yang menyediakan lapisan abstraksi akses data berorientasi objek, memungkinkan pengembang menggunakan antarmuka yang seragam untuk berbagai sistem manajemen basis data (DBMS) seperti MySQL, PostgreSQL, SQLite, dan lainnya. Dengan PDO, interaksi dengan database termasuk membuat koneksi, menyiapkan *prepared statements*, mengelola transaksi, dan menangani kesalahan dapat dilakukan menggunakan metode yang sama tanpa mengubah sintaks SQL saat berpindah DBMS, sehingga meningkatkan portabilitas dan keamanan kode.(php.net)

BAB III

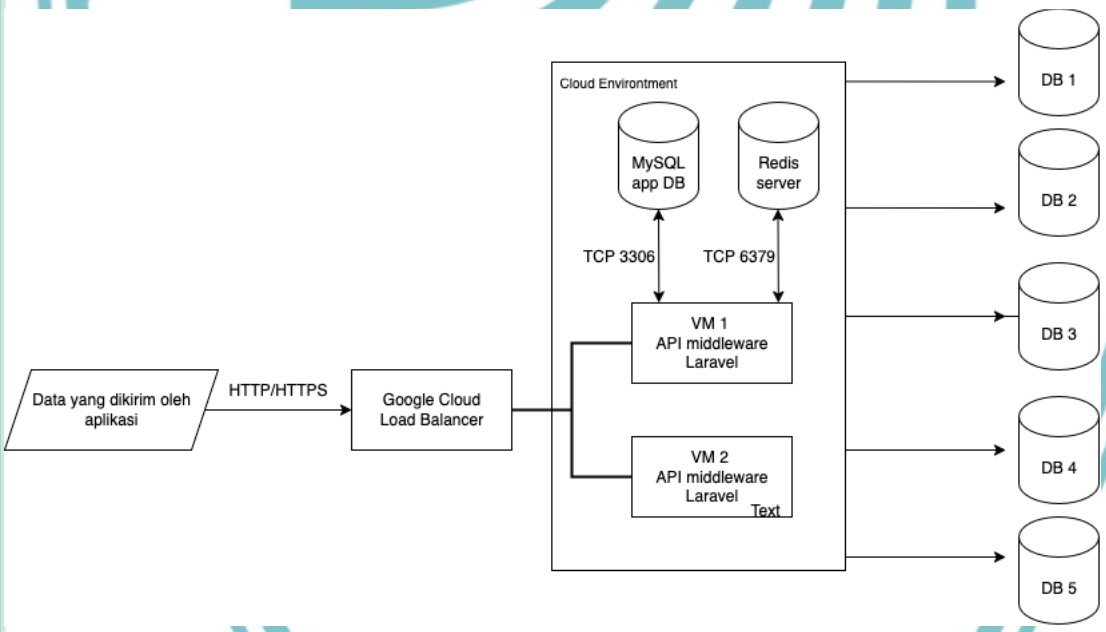
METODE PENELITIAN

3.1 Rancangan Penelitian

Penelitian yang akan dilaksanakan adalah Rancang Bangun *Middleware* Berbasis Redis Stream Dengan Arsitektur *Load-Balancing,* bahasa pemrograman yang akan digunakan untuk membuat *middleware* tersebut adalah PHP dengan bantuan *framework* Laravel.

3.1.1 Rancangan arsitektur

Pada gambar 3.1 adalah rancangan arsitektur yang akan dibuat oleh peneliti

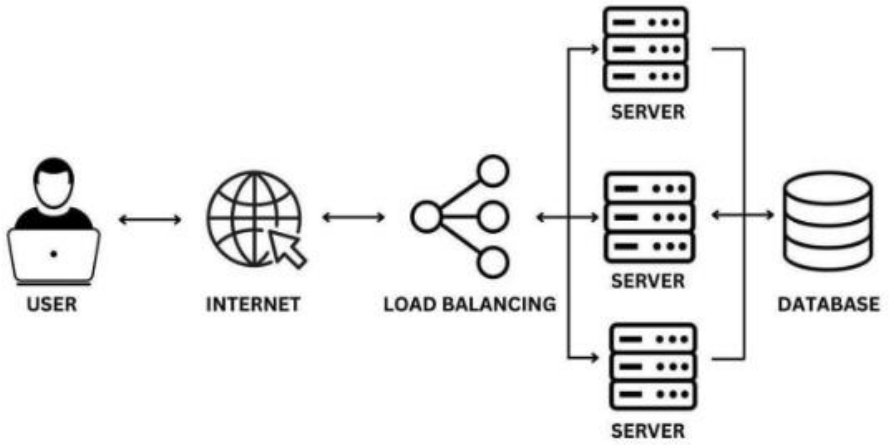


Gambar 3.1 Rancangan Arsitektur

Arsitektur ini menggunakan *load-balancing* dari *Google Cloud* dan Laravel sebagai *framework* dari aplikasi *middleware* untuk menerima data dari pengirim dan meneruskannya ke Redis Server, yang berfungsi sebagai tempat penyimpanan sementara data yang masuk. Redis menyimpan data menggunakan tipe data *stream* dan mendistribusikan data menggunakan PDO(*PHP Data Object)* yang terkoneksi ke 5 database (DB A sampai DB B) sesuai dengan kebutuhan masing-masing

database. *Load-Balancer* berperan sebagai penyeimbang *traffic* yang memastikan data tetap terkirim jika salah satu aplikasi *middleware* mengalami kegagalan. Dengan pendekatan ini, sistem menjadi memiliki ketahanan terhadap kegagalan.

Sebagai acuan arsitektur, berikut adalah penelitian yang dilakukan oleh Narsipuram, M., Rai, A. and Tiwari, A. mengenai *Cloud Load Balancing.*

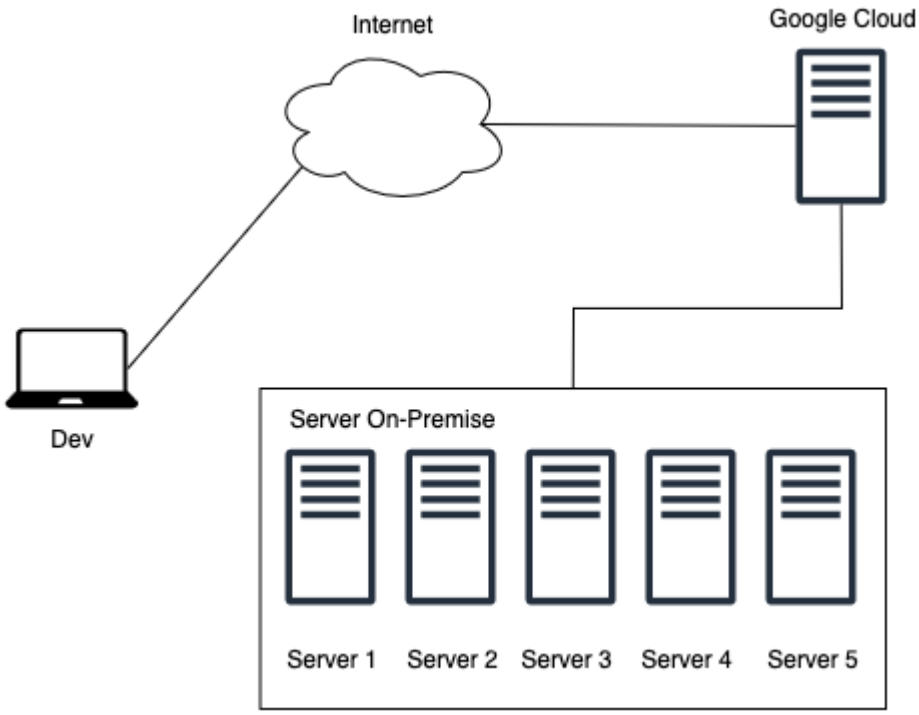


Gambar 3.2 Arsitektur Narsipuram, M., Rai, A. and Tiwari, A.

Pada lingkungan *Cloud Computing*, *load balancer* ditempatkan di antara klien dan *server* untuk mendistribusikan *traffic*. Mekanismenya terletak pada algoritma yang mempertimbangkan variabel seperti tingkat beban server dan jarak geografis. Secara umum, algoritma ini dibagi menjadi dua kelompok: statis dan dinamis, Algoritma statis seperti *Round Robin* mengalirkan permintaan ke setiap server secara berurutan dan *Weighted Round Robin* memberi bobot lebih besar pada server yang memiliki kapasitas lebih tinggi sehingga dapat menampung trafik aplikasi yang lebih besar. Sementara itu, algoritma dinamis menyesuaikan keputusan secara *real-time* contohnya, *Least Connections* mengarahkan sesi baru ke server dengan koneksi aktif paling sedikit. *Least Response Time* memprioritaskan server yang menunjukkan waktu respons rata-rata tercepat sekaligus koneksi tersedikit. Adapun *Least Bandwidth* memilih server yang belakangan ini menggunakan bandwidth paling rendah (Narsipuram, M., Rai, A. and Tiwari, A. 2024)

3.1.2 Topologi Arsitektur

Pada gambar 3.3 adalah rancangan topologi arsitektur yang akan dibuat oleh peneliti

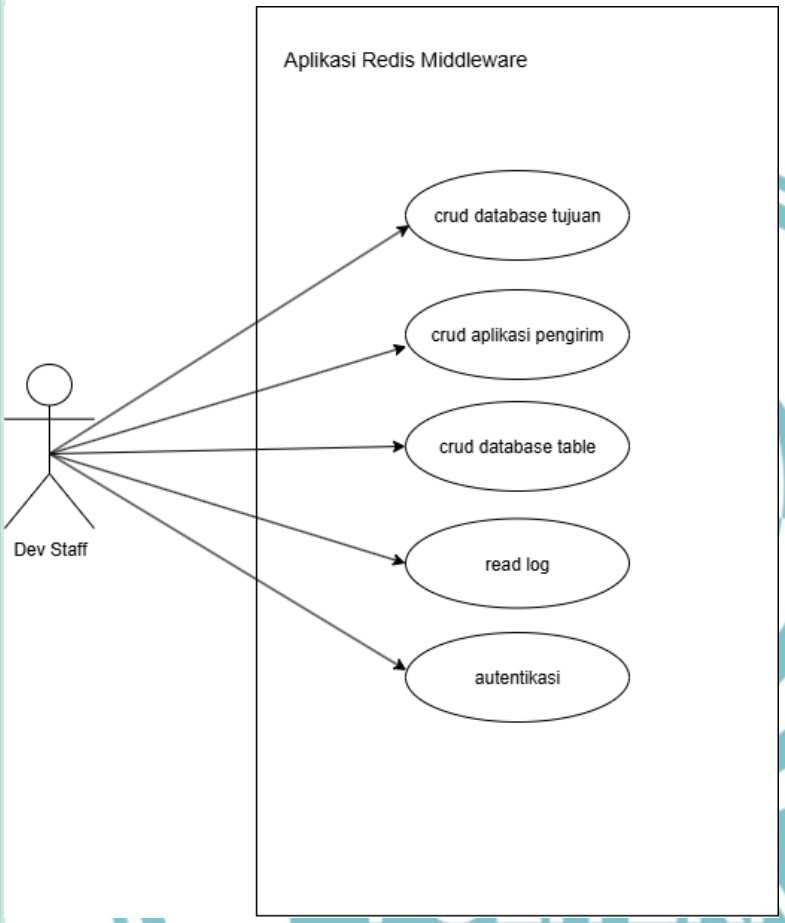


Gambar 3.3 Topologi Arsitektur

Pada topologi tersebut, sebuah lingkungan hybrid-cloud diperlihatkan di mana laptop tim *developer* dihubungkan ke *instance* Google Cloud melalui internet. *instance* cloud itu kemudian dipakai sebagai lingkungan *middleware* yang meneruskan data ke *database* on-premise melalui koneksi TCP.

3.1.3 *Use Case Diagram*

Pada gambar 3.4 adalah *use case* dari sistem yang dibuat.



Gambar 3.4 *use case diagram*

Pada diagram use-case ditampilkan bahwa Aplikasi Redis *Middleware* digambarkan sebagai sistem yang menyediakan lima fungsi utama yaitu: CRUD database tujuan, CRUD aplikasi pengirim, CRUD tabel *database*, pembacaan log, dan autentikasi yang seluruhnya dapat diakses oleh aktor Dev Staff.

3.1.4 *Project Requirement*

A. 4 Anggota Tim Dev dapat mengirim data ke 5 database
B. *Database* tujuan hanya menerima data yang dibutuhkan dari aplikasi pengirim.
C. Menyimpan data di dalam server redis dan kembali mengirim data jika database tujuan sudah *up*
D. Mencatat log distribusi data

3.1.5 Alat dan Bahan

A. Visual Studio Code
B. PHP dan Laravel
C. Server Redis
D. Server Database
E. Server *middleware*

3.2 Tahapan Penelitian

1. Analisis Permasalahan

   Pada bagian latar belakang sudah disebutkan beberapa penelitian terdahulu mengenai penerapan Redis tetapi masih ada kebaruan yang dapat dilakukan oleh peneliti, kebaruan tersebut yaitu: menggunakan Redis *stream* untuk mengirim data, membuat arsitektur *load-balancing* untuk menunjang *high availability,* dan mencatat log pengiriman data pada *middleware.* Peneliti akan mengimplementasi kebaruan tersebut untuk studi kasus di PT. Bangun Abadi Teknologi Indonesia(BATI).

2. Studi Literatur

   Pada tahapan ini peneliti mengumpulkan jurnal referensi yang berkaitan dengan penelitian dan teknologi yang akan digunakan.

3. Analisis Kebutuhan

   Sistem *middleware* yang akan dibuat harus memenuhi kebutuhan perusahaan yaitu: mengirim data ke *database* tujuan, data yang belum terkirim harus tersimpan di redis dan dapat dikirim kembali, dan mencatat log pengiriman. Peneliti melakukan analisis terhadap kebutuhan teknis yang akan digunakan. Peneliti menggunakan *Visual Studio Code* sebagai *code editor*, *google cloud* sebagai server testing Redis dan *load-balancing*, PHP sebagai bahasa pemrograman, dan Redis sebagai tempat penyimpanan sementara.

4. Perancangan Sistem

   Pada tahap perancangan sistem, peneliti akan mengumpulkan dokumentasi dan tutorial dari teknologi yang digunakan kemudian akan membuat *flowchart* dan rancangan arsitektur *database* sebelum melakukan implementasi.

5. Implementasi Sistem

   Pada tahap implementasi sistem, akan membuat kode program aplikasi *middleware* dan arsitektur *load-balancing* menggunakan *Google Cloud,* peneliti juga akan membuat server *database* sebagai tujuan dari data yang dikirimkan.

6. Evaluasi dan Analisis

   Pada tahap evaluasi, peneliti akan melakukan pengujian dan analisis dari sistem yang telah dibuat, pengujian akan meliputi: tes fungsional guna memastikan aplikasi *middleware* berfungsi dengan baik dengan parameter dapat mengirim data ke *database* tujuan sesuai dengan kebutuhan *database* tersebut, dapat menyimpan data apabila *database* tujuan sedang dalam keadaan *down* dan akan mengirim data kembali apabila database tujuan sudah dalam keadaan *up.* (tes performa)

7. *Deploy* dan *Maintenance*

   Pada tahap ini setelah melakukan evaluasi dan analisis, peneliti akan mengimplementasikan sistem menggunakan server yang disediakan oleh perusahaan dan melakukan *maintenance* jika ada kebutuhan baru atau perbaikan di masa yang akan datang.

8. Laporan Akhir

   Pada tahap ini peneliti akan membuat laporan akhir yang disesuaikan dengan pedoman skripsi jurusan teknik informatika dan Komputer serta dokumentasi sistem yang sudah dibuat.

3.3 Objek Penelitian

Fokus penelitian tugas akhir ini adalah implementasi dan analisis Redis *stream* untuk mengirim data ke database tujuan. Pada penerapannya, *stream* berguna untuk menyimpan data apabila database yang dituju sedang dalam keadaan *down* dan akan mengirimkan kembali saat database tujuan sudah dalam keadaan *up* dan arsitektur *load-balancing* berguna untuk menyebarkan *traffic* yang diterima oleh aplikasi *middleware.*Objek pengujian pada penelitian ini berada di PT. Bangun Abadi Teknologi Indonesia yang dilakukan pada tanggal 7 Februari sampai dengan 20 Juni 2025.

BAB IV
HASIL DAN PEMBAHASAN

4.1 Analisis Kebutuhan

Sistem yang akan dirancang adalah sistem distribusi data menggunakan Redis sebagai tempat penyimpanan data sementara yang dikirim oleh aplikasi. Data yang dikirim dari aplikasi ke Redis menggunakan tipe data Redis *stream*, dalam proses perancangannya sistem ini membutuhkan server pendukung untuk diimplementasikan. Kebutuhan yang harus dipenuhi oleh sistem adalah mendistribusikan data ke 5 *database* dan mengirimkan data kembali jika *database tujuan* sudah dalam status *up*.

4.1.1 Analisis Perangkat Lunak

Tabel 4.2 berisi perangkat lunak yang dibutuhkan untuk mengembangkan sistem.

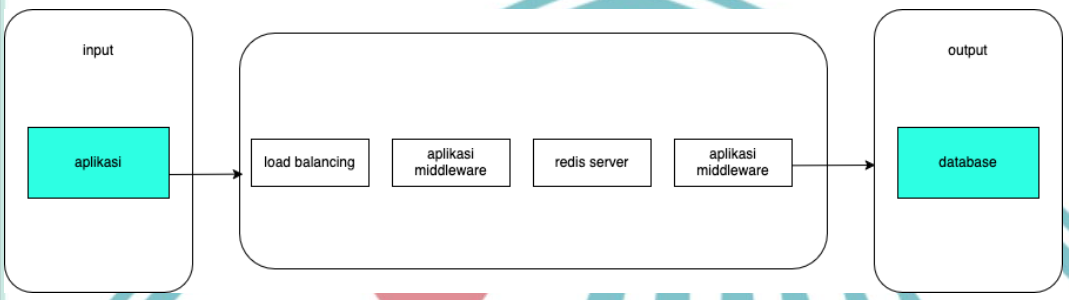
Tabel 4.2 tabel analisis kebutuhan

No	Fungsi	Kebutuhan Perangkat Lunak
1	Pembuatan Perangkat Lunak	• Visual Studio Code: Perangkat lunak yang digunakan untuk menuliskan kode • PHP: Bahasa pemrograman yang digunakan untuk membuat *middleware*
2	Penyimpanan Data	• Redis: Perangkat lunak untuk menyimpan data sementara yang dikirim dari aplikasi • MySql: Perangkat lunak yang digunakan untuk menyimpan konfigurasi

Pembuatan perangkat lunak dipenuhi oleh Visual Studio Code sebagai lingkungan pengembangan untuk menulis kode, serta PHP sebagai bahasa pemrograman utama yang membangun komponen middleware. Fungsi penyimpanan data menggunakan Redis untuk menyimpan data secara sementara sebelum diproses, dan MySQL sebagai *database* untuk menyimpan data konfigurasi.

4.2 Perancangan Sistem

4.2.1 Diagram Blok Sistem

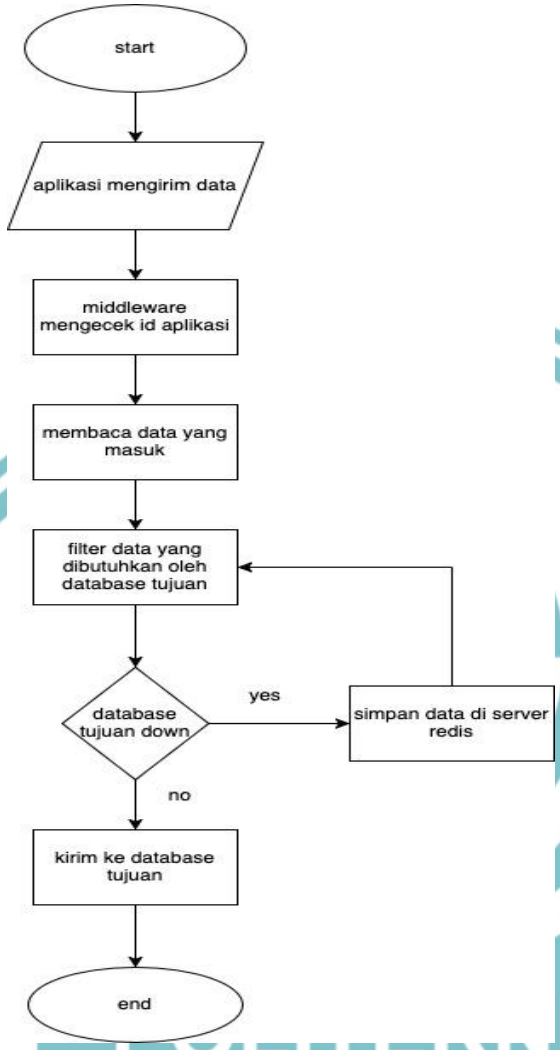


Gambar 4.1 Diagram Blok Sistem

Pada gambar 4.1 merupakan diagram blok yang menggambarkan alur *input,* proses, dan *output.* Pada blok *input,* aplikasi akan mengirimkan data melalui HTTP request ke alamat IP *load-balancing* kemudian akan meneruskannya ke VM aplikasi *middleware* yang akan mengirim data ke server redis, kemudian *listener* di dalam aplikasi *middleware* mendeteksi data yang baru masuk dan memproses data tersebut sebelum dikirim ke *database* tujuan.

4.2.2 Flowchart Sistem

Pada gambar 4.2 adalah alur diagram sistem rancang bangun dan implementasi *middleware* berbasis redis stream dengan arsitektur *load-balancing,*

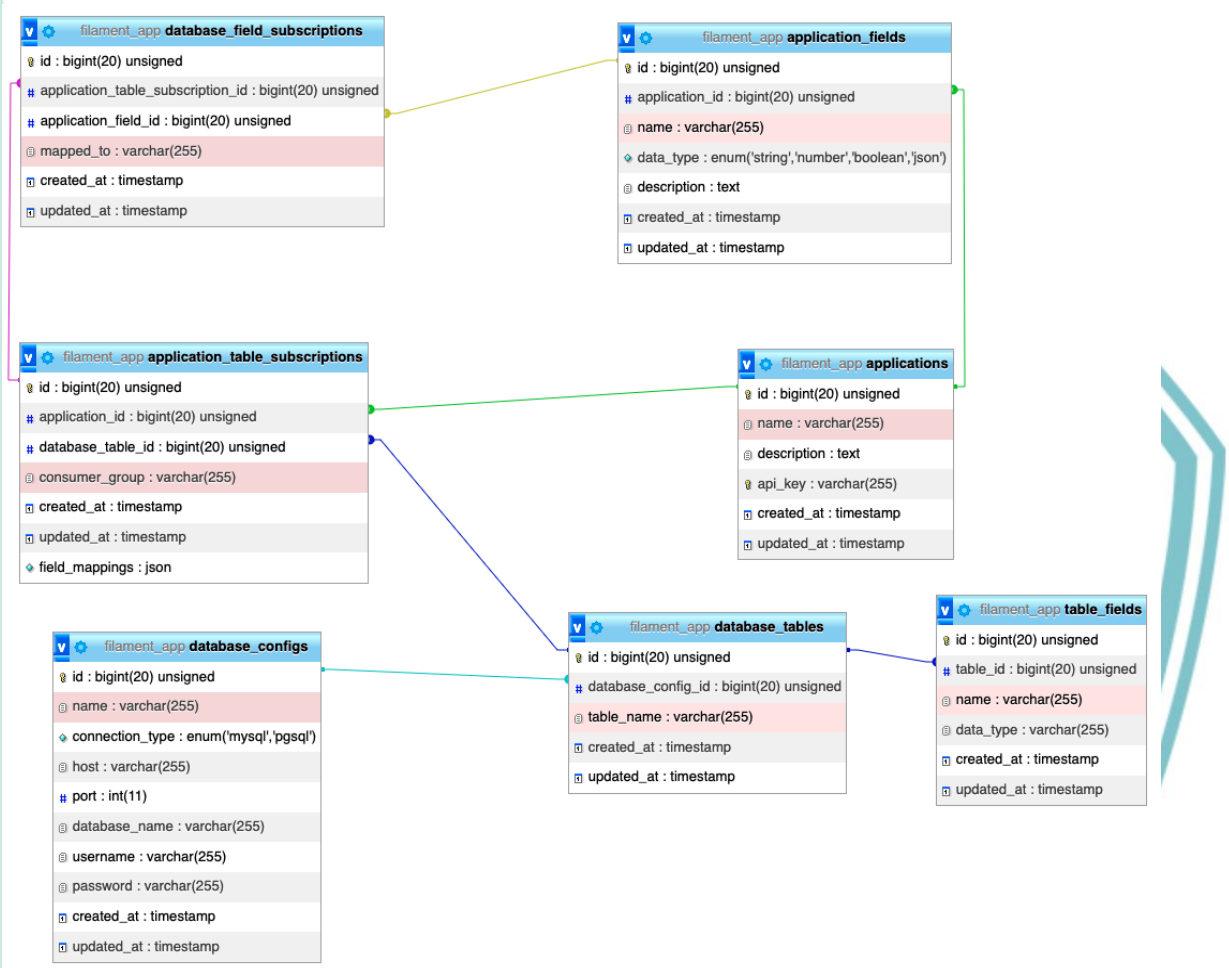


Gambar 4.2 Rancangan Diagram Sistem

Pertama-tama user akan mengirim data menggunakan *endpoint* API yang akan dikembangkan menggunakan PHP Laravel dan data yang akan dikirim berbentuk JSON serta membutuhkan *api-key* pada *header* sebagai id dari aplikasi yang akan mengirim data. Saat data berhasil diterima oleh *middleware* kemudian data akan disimpan kedalam server Redis dengan tipe data *stream.* Di dalam *middleware* terdapat *listener* yang akan mendengarkan apakah ada data *stream* baru yang masuk kedalam Redis server, jika terdapat data *stream* baru yang masuk maka *middleware* akan melakukan pengecekan terhadap *database* yang dituju apakah dalam keadaan *up* atau *down,* jika *database* dalam keadaan *down* maka *middleware* akan menahan data tersebut dan akan mencoba mengirimkan ulang saat *database* sudah dalam status *up* serta *middleware* akan mencatat log pengiriman data didalam *middleware.*

4.2.3 Struktur Database

Agar aplikasi *middleware* dapat menyimpan data konfigurasi aplikasi, data yang dikirim oleh aplikasi, *database* tujuan, dan tabel yang dimiliki oleh *database* tujuan maka dibuatlah struktur *database* seperti pada gambar 4.3.



Gambar 4.3 Diagram Database

Pada Gambar 4.3 ditampilkan struktur *database* yang digunakan untuk mengatur alur distribusi data antara aplikasi dan *database* tujuan. Struktur database tersebut dibuat berdasarkan kebutuhan perusahaan agar dapat: menyimpan aplikasi yang akan mengirim data, data yang akan dikirim oleh aplikasi tersebut, dan ke *database* mana data tersebut akan dikirim serta data apa yang akan diterima oleh aplikasi pengirim. Di dalam struktur tersebut, data setiap aplikasi dicatat terlebih dahulu

pada tabel *applications*, tempat nama aplikasi dan api_key disimpan agar proses autentikasi dapat dilakukan ketika payload dikirim. Data yang dikirim oleh aplikasi disimpan pada tabel *application_fields*. Detail koneksi untuk setiap *database* tujuan, seperti host, port, dan nama *database*, disimpan dalam tabel *database_configs* sehingga koneksi dapat dibuat secara dinamis sesuai kebutuhan. Setiap tabel yang dimiliki oleh *database* tujuan disimpan oleh tabel *database_tables*, sedangkan kolom-kolom yang dimiliki setiap tabel tersebut disimpan pada tabel *table_fields*. Hubungan pemetaan antara atribut aplikasi dan kolom tabel tujuan diatur melalui tabel *database_field_subscriptions*, yang berfungsi sebagai jembatan agar setiap *field* dari aplikasi dapat dikirim secara tepat pada kolom di *database* tujuan. Terakhir, relasi *many-to-many* antara aplikasi dan tabel tujuan dikelola oleh tabel *application_table_subscriptions*, memungkinkan sistem menentukan tabel mana saja yang secara sah dapat diisi oleh sebuah aplikasi tanpa harus mengubah skema pemetaan kolom.

4.3 Implementasi Sistem

Setelah perancangan kebutuhan, alur sistem, dan struktur *database* sudah selesai dibuat, selanjutnya adalah tahap implementasi sistem *middleware* seperti: konfigurasi *environment variable* untuk terhubung ke redis *server*, kode *controller* untuk menerima *request*. kode untuk membaca data baru yang masuk ke redis server, dan kode untuk mendistribusikan data ke *database* tujuan.

4.3.1 Pembuatan Kode Program *middleware*

Pada Gambar 4.4 diperlihatkan konfigurasi *environment variable* untuk terkoneksi dengan Redis server.

```
REDIS_HOST=35.208.189.243
REDIS_USERNAME=yusuf
REDIS_PASSWORD=yusuf64
REDIS_CLIENT=phpredis
REDIS_UNIFIED_STREAM=app:data:stream
QUEUE_CONNECTION=redis-stream
REDIS_QUEUE_CONNECTION=default
REDIS_QUEUE=redis
REDIS_QUEUE_NAME=app:data:stream
REDIS_STREAM_GROUP=stream-workers

LISTENER_CONNECTION=redis-stream
```

Gambar 4.4 Konfigurasi *environment variable* untuk Redis

Pada konfigurasi tersebut terdapat variabel lingkungan untuk membentuk jalur kerja Redis dan antrian Laravel, alamat IP Redis server telah ditentukan pada variabel REDIS_HOST, untuk kredensial autentikasi ke Redis server disediakan oleh REDIS_USERNAME dan REDIS_PASSWORD, ekstensi PHP yang digunakan untuk berkomunikasi dengan Redis disimpan oleh REDIS_CLIENT, dan *key stream* yang menyimpan data *stream* di definisikan pada REDIS_UNIFIED_STREAM. Selanjutnya, *driver* antrian Laravel diarahkan ke koneksi khusus stream melalui QUEUE_CONNECTION, sementara koneksi Redis bawaan untuk job biasa diatur oleh REDIS_QUEUE_CONNECTION dengan nama koneksi REDIS_QUEUE dan nama antrean REDIS_QUEUE_NAME. Pengelompokan konsumen pada operasi XREADGROUP difasilitasi oleh REDIS_STREAM_GROUP, dan driver antrian yang memicu proses pendengar disimpan di LISTENER_CONNECTION. Keseluruhan variabel ini kemudian dibaca oleh berkas konfigurasi, sehingga satu jalur kerja listener yang melakukan pembacaan blokir pada stream dan satu jalur worker yang mengeksekusi job Laravel reguler dapat terintegrasi.

Pada Gambar 4.4 diperlihatkan metode *store* pada DataController yang berfungsi sebagai *endpoint* dari aplikasi pengirim ke *middleware* Redis.

```
3 references | 0 implementations | You, 2 weeks ago | 1 author (You)
class DataController extends Controller
{
    1 reference | 0 overrides
    public function store(Request $request): JsonResponse|mixed
    {

        //validate API key

        $apiKey = $request->header(key: 'X-API-Key');

        $application = Application::where(column: 'api_key', operator: $apiKey)->firstOrFail(columns: ['id', 'api_key']);
        $validFields = $application->applicationFields()->pluck(column: 'name')->toArray();

        $streamKey = env(key: 'REDIS_UNIFIED_STREAM', default: 'app:data:stream');

        $filteredData = $request->only(keys: $validFields);
        $filteredData['application_id'] = $application->id;
        $filteredData['api_key'] = $apiKey;
        $filteredData['enqueued_at'] = Carbon::now()->toIso8601String();
        try {
            $MessageId = Redis::command(
                method: 'xadd',
                parameters: [
                    $streamKey,
                    '*',
                    $filteredData,
                ]
            );

            //dd($MessageId);
            return response()->json(data: [
                'message' => 'Data received and queued',
                'message_id' => $MessageId,
                "data" => $filteredData
```

Gambar 4.4 kode program mengirim data ke dalam server redis

Pertama-tama sistem membaca X-API-Key pada *header request* untuk mengotentikasi aplikasi, kemudian hanya atribut yang terdaftar di tabel *application_fields* yang diizinkan lewat sehingga potensi *payload injection* dapat dicegah. Setelah kolom valid dipilah, sistem menambahkan data *application_id*, *api_key*, dan waktu *enqueued_at* sebelum akhirnya mengeksekusi perintah XADD melalui Redis::command untuk menyimpan data ke stream yang ditentukan oleh *environment variable REDIS_UNIFIED_STREAM*

Pada Gambar 4.5 diperlihatkan implementasi metode pop() dalam kelas RedisStreamQueue, yang berfungsi sebagai *listener* data terbaru pada Redis *Stream* sekaligus penghubung ke sistem antrian Laravel.

```
class RedisStreamQueue extends Queue implements QueueContract
    */
    0 references | 0 overrides
    public function pop($queue = null): null
    {

        $messages = Redis::xreadgroup(
            $this->group,
            $this->consumer,
            [$this->stream => '>'],
            1,
            0
        );      You, 2 weeks ago • change config in database and queue …

        $entries = $messages[$this->stream] ?? [];
        dump(vars: $entries);
        foreach ($entries as $id => $fields) {
            // 1) Hand it off to a normal Laravel queue job,
            //    forcing it onto the "redis" connection:
            ProcessStreamMessage::dispatch(arguments: $id, $fields)
                ->onConnection(connection: 'redis')
                ->onQueue(queue: config(key: 'default')); // or a named queue
            // 2) Acknowledge it in the stream so it won't be re-delivered
            $this->redis->xack($this->stream, $this->group, [$id]);
        }

        // We handled dispatch & ack; nothing else for Laravel to do here
        return null;
    }
```

Gambar 4.5 kode program mengambil data dari server redis

Pertama-tama sistem memanggil XREADGROUP dengan argumen *consumer group*, *stream key*, dan kursor > untuk menarik satu data yang belum pernah diproses kemudian disimpan dalam variabel *entries* sehingga setiap baris berisi pasangan id=>payload. Setiap entri kemudian di kirim sebagai *job* ProcessStreamMessage melalui metode *dispatch*. Setelah *job* berhasil dikirim, sistem segera memanggil XACK untuk mengonfirmasi id pesan sehingga data tersebut tidak lagi berada di *pending list* dan tidak akan dibaca ulang.

Pada Gambar 4.6 diperlihatkan metode handle() dalam *job* ProcessStreamMessage, yang mengeksekusi seluruh logika distribusi data.

```php
public function handle(): void
    {
        try {
            \Log::info('Processing payload', $this->payload);
            $app = Application::where('api_key', $this->payload['api_key'] ?? null)->first();
            if (!$app) {
                return;
            }

            $sentAt = Carbon::parse($this->payload['enqueued_at'] ?? now());

            $rawData = collect($this->payload)
                ->except(['api_key', 'enqueued_at'])
                ->toArray();

            $subscriptions = ApplicationTableSubscription::with([
                'databaseTable.database',
                'fieldMappings.applicationField',
            ])->where('application_id', $app->id)->get();
            //dump($subscriptions->count());
            foreach ($subscriptions as $sub) {
                $dbConfig = $sub->databaseTable->database;
                $tableName = $sub->databaseTable->table_name;
                $consumer = $sub->consumer_group;
                //dump("sub id: {$sub->id}");
                $mapped = [];
                foreach ($sub->fieldMappings as $mapping) {
                    $appFieldName = $mapping->applicationField->name;
                    if (isset($this->payload[$appFieldName])) {
                        $mapped[$mapping->mapped_to] = $this->payload[$appFieldName];
                    }
                }

                if (empty($mapped)) {
                    // nothing to insert for this table
                    continue;
                }
                //dump($mapped);
                if ($this->isDatabaseServerReachable($dbConfig->host, $dbConfig->port)) {
                    $this->insertIntoTable($dbConfig, $tableName, $mapped, $app->name, $rawData,
 $sentAt);
                } else {
                    $this->holdForRetry($consumer, $tableName, $mapped, $app->name, $rawData, $sentAt,
subId: $sub->id);
                }
            }
        } catch (\Throwable $th) {
            //throw $th;
            \Log::error("Error processing message {$this->messageId}: {$th->getMessage()}", [
                'payload' => $this->payload,
                'trace' => $th->getTraceAsString(),
            ]);
        }
    }
```

Gambar 4.6 kode program menjalankan job untuk distribusi data

Pertama-tama sistem memverifikasi *api_key* pada *payload* untuk memperoleh *instance* aplikasi selanjutnya sistem mengambil *raw data* yang berisi data yang dikirim dan daftar *ApplicationTableSubscription* milik aplikasi tersebut lengkap dengan konfigurasi *database* dan kolomnya. Pada setiap *subscription*, *payload* dipetakan ulang agar hanya kolom yang disetujui melalui fieldMappings yang dikirim. Sebelum mengirm data ke *database* tujuan, metode *isDatabaseServerReachable* memastikan bahwa *host* dan *port database* tujuan dalam keadaaan *up*, bila dalam keadaan *up*, data dimasukkan ke *database* tujuan menggunakan fungsi *insertIntoTable*. Jika *database* tujuan sedang dalam keadaan

down, data dialihkan ke antrian ulang menggunaka fungsi *holdForRetry* untuk dicoba kembali pada dalam waktu 5 menit sehingga pesan tidak hilang.

Pada Gambar 4.7 diperlihatkan implementasi metode *insertIntoTable* yang menjalankan proses pengiriman data ke *database* tujuan.

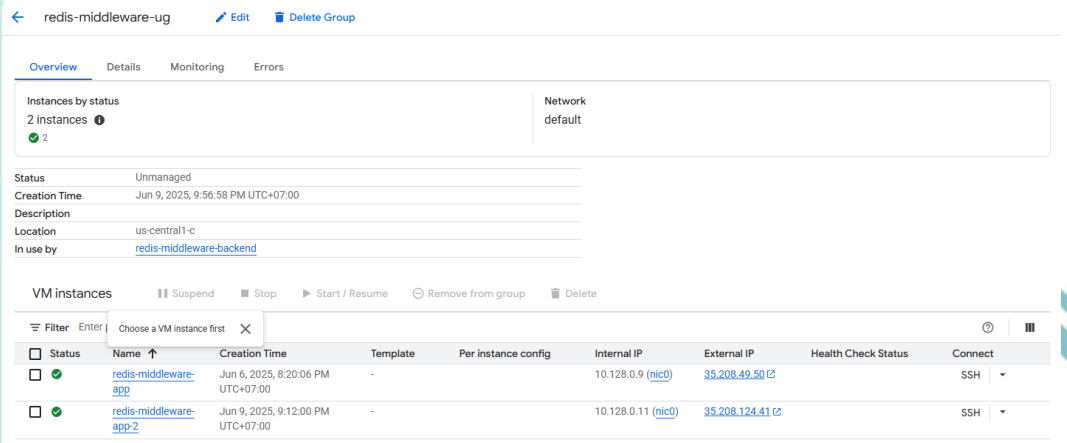
```
class ProcessStreamMessage implements ShouldQueue
    protected function insertIntoTable($db, string $table, array $mapped, string $source, array $rawData, Carbon $sentAt): void
    {
        $pdo = new PDO(
            dsn: "{$db->connection_type}:host={$db->host};dbname={$db->database_name}",
            username: $db->username,
            password: $db->password,
            options: [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION]
        );
        try {
            $cols = implode(separator: ', ', array: array_keys(array: $mapped));
            $ph = implode(separator: ', ', array: array_map(callback: fn($c): string => ":{$c}", array: array_keys(…$mapped)));

            $pdo->beginTransaction();
            $stmt = $pdo->prepare(query: "INSERT INTO {$table} ({$cols}) VALUES ({$ph})");
            foreach ($mapped as $col => $val) {
                $stmt->bindValue(param: ":{$col}", value: $val);
            }
            $stmt->execute();
            $pdo->commit();

            // 7) Log success
            Log::create(attributes: [
                'source' => $source,
                'destination' => $table,
                'data_sent' => json_encode(value: $rawData),
                'data_received' => json_encode(value: $mapped),
                'sent_at' => $sentAt,
                'received_at' => now(),
                'status' => 'OK',
                'message' => 'inserted successfully',
            ]);
        } catch (\Throwable $e) {
            \Log::info(message: "Error: {$e->getMessage()}");      You, 2 weeks ago • add process stream message job in main …
        }
    }
```

Gambar 4.7 kode program untuk mengirim data ke database tujuan

Objek PDO dibentuk dengan parameter *host*, jenis koneksi, nama *database*, serta kredensial autentikasi yang diambil dari konfigurasi. Nama-nama kolom dan *placeholder* nilai disusun menggunakan *implode()* serta *prepared statement*, kemudian setiap nilai diikat (bindValue) sehingga risiko SQL *injection* dapat dicegah. Prosedur *beginTransaction* dan *commit* membungkus eksekusi *insert* jika satu baris gagal, seluruh operasi otomatis dibatalkan. Setelah transaksi berhasil, fungsi mencatat detail lengkap pada tabel log, meliputi sumber data, *database* tujuan, data yang dikirim, waktu pengiriman serta penerimaan, dan status operasi OK.

4.3.2 Implementasi *Load Balancing* Menggunakan *Google Cloud*

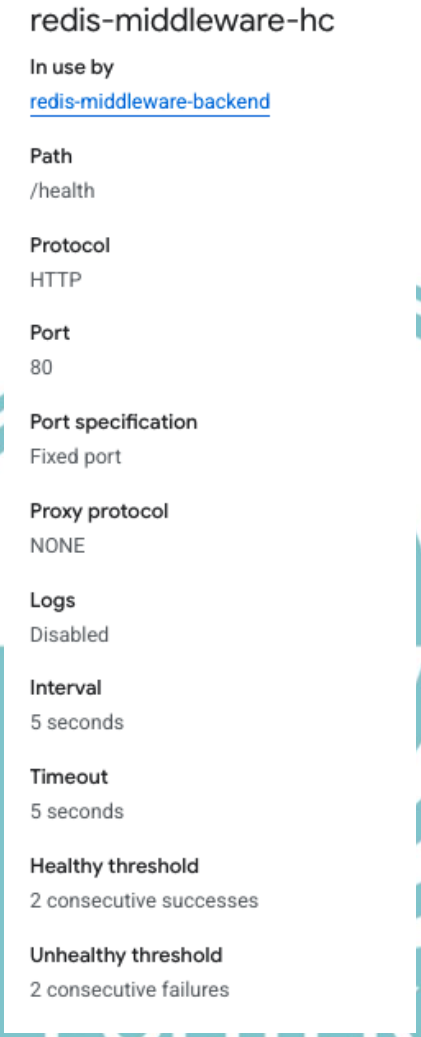
Pada Gambar 4.8 ditampilkan konfigurasi sebuah *unmanaged instance group* di zona us-central1-c yang berisi dua VM aplikasi Redis-middleware



Gambar 4.8 konfigurasi *unmanaged group*

Instance group ini dibuat agar *load-balancer* memiliki satu *backend service* tunggal, sehingga *traffic* klien dapat didistribusikan secara merata ke kedua VM. Tipe *unmanaged* dipilih karena VM aplikasi telah tersedia sebelumnya dan kebutuhan autoscaling tidak menjadi prioritas pada skenario pengujian ini.

Pada Gambar 4.9 ditampilkan konfigurasi *health check* bernama redis-middleware-hc yang dipakai oleh backend service redis-middleware-backend untuk memantau ketersediaan dua VM Redis-middleware.



Gambar 4.9 konfigurasi *health check*

Mekanisme ini diatur menggunakan protokol HTTP pada port 80 dan mengarah ke endpoint /health, setiap 5 detik load balancer mengirim satu permintaan dengan batas waktu respons 5 detik. Sebuah *instance* akan berstatus *healthy* jika menerima dua respons berturut-turut yang sukses, dan akan dianggap *unhealthy* bila gagal merespons dua kali berturut-turut. Konfigurasi ambang minimal tersebut memungkinkan deteksi kegagalan yang cepat tanpa menghasilkan lonjakan trafik pemeriksaan, sekaligus memastikan hanya VM yang benar-benar responsif yang menerima beban produksi. 5 db

redis-middleware-backend

General properties

Load balancer type	Classic Application Load Balancer (EXTERNAL)
Endpoint protocol	HTTP
In use by	redis-middleware-map
Timeout	30 seconds
IP address selection policy	Only IPv4
Health check	redis-middleware-hc View health check details
Backend security policy	None
Session affinity	None
Cloud CDN	Disabled
Connection draining timeout	0 seconds
Custom request headers	Currently there are no custom request headers configured
Custom response headers	Currently there are no custom response headers configured
Backend authentication	Disabled
Tags	–

Backends

Name	Type	IP stack type	Scope	Healthy	Autoscaling	Balancing mode	Selected ports
redis-middleware-ug	Instance group	IPv4	us-central1-c	2 of 2	No configuration	N/A	80

Gambar 4.10 konfigurasi *backend service*

Pada gambar 4.10 menunjukan konfigurasi *backend service* yang menjadi pusat pengaturan aliran *traffic* pada implementasi *load balancer*. Status "2 of 2 healthy" menandakan bahwa kedua VM dalam grup lulus pemeriksaan *health check* redis-middleware-hc yang memanggil *endpoint* /health alhasil hanya VM responsif yang akan menerima beban.

4.4 Pengujian

4.4.1 Deskripsi Pengujian

Dalam pengujian ini, alur sistem dipastikan berfungsi dengan langkah-langkah berikut: pertama, simulasi *producer* dilakukan untuk mengirimkan *payload* ke *stream*, dan verifikasi dilakukan untuk memastikan setiap pesan diterima dan diantarkan ke semua database tujuan yang terdaftar melalui *worker* Redis-Stream dan Redis *Queue*. Selanjutnya, salah satu *database* dimatikan untuk menguji mekanisme *retry*, di mana pesan yang gagal secara otomatis ditahan dan diproses ulang setelah *database* kembali *online*. Uji beban dilakukan pada GCP dengan mengukur efektivitas *load balancing* untuk memastikan distribusi trafik masuk

yang merata. Terakhir, untuk uji penggunaan resource, endpoint *load-balancing* akan dipenuhi oleh request menggunakan Apache Benchmark untuk mengukur *resource* CPU yang digunakan oleh VM aplikasi redis middleware untuk menerima *request* dan resource RAM yang digunakan oleh server redis untuk menerima *read* dan *write* data.

4.4.2 Prosedur Pengujian

4.4.2.1 Uji Skenario Normal

Pada uji skenario normal, semua *database* tujuan memiliki status *up* atau dalam keadaan menyala dan data dikirimkan ke aplikasi middleware melalui permintaan HTTP, dan seluruh data yang dikirimkan diharapkan dapat diterima oleh database tujuan secara utuh. Tujuan dari uji tersebut adalah untuk memastikan *database* menerima data yang dibutuhkan jika status *database* tersebut dalam keadaan menyala.

4.4.2.2 Uji Skenario *Partial-Failure*

Pada uji skenario *partial-failure*, setengah dari jumlah permintaan dikirimkan terlebih dahulu, kemudian VM server-database-2 dimatikan. Setelah itu, sisa permintaan dikirimkan, dan VM server-database-2 diaktifkan kembali. Tujuan dari uji tersebut adalah untuk memastikan bahwa seluruh data yang tertunda dapat diterima oleh VM server-database-2 setelah kembali aktif.

4.4.2.3 Uji *Load-Balancing*

Pada uji *load-balancing,* sistem menerima data dari request HTTP kemudian menyebarkan traffic secara merata kepada kedua VM aplikasi redis middleware. Tujuan dari uji tersebut adalah untuk memastikan *request* tersebar ke 2 VM sehingga menjaga *high-availability* pada sistem.

4.4.2.4 Uji Penggunaan *Resource*

Pada pengujian *resource* untuk VM aplikasi redis *middleware* dan server redis akan menggunakan Apache Benchmark. Apache Benchmark akan di *install* pada server terpisah lalu akan mengirimkan beban kepada *endpoint* dari *load-balancing* dan akan menunjukan status dari beban yang terkirim. Untuk hasil penggunaan *resource* akan dilihat menggunakan grafik dari *Google Cloud*. Tujuan dari pengujian ini adalah untuk mengetahui resource CPU dari grup *load-balancing* dan memori dari server redis.

4.4.3 Hasil Pengujian

4.4.3.1 Hasil Pengujian Skenario Normal

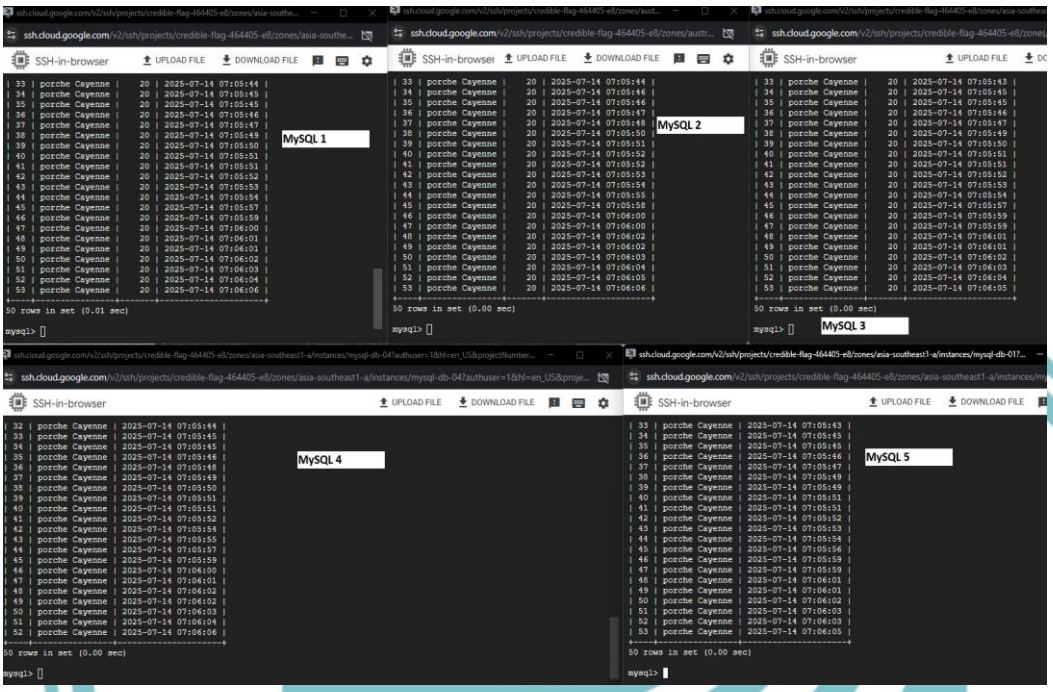
Pada Gambar 4.11 ditunjukkan potongan kode pengujian yang mengirim 50 permintaan HTTP POST berturut-turut ke *endpoint middleware.*

```
custom_post_headers = {
    "X-Api-Key": "jxw1IH2tIXckPL4zbjtVThwVjafZwiC3",
}

# Data yang akan dikirim sebagai JSON body
data_to_send = {
    "car_name": "porche 911",
    "stock": 10,
}

for i in range(50):
    response_object = send_json_post_request(
        post_target_url, headers=custom_post_headers, json_body=data_to_send
    )

    if response_object:
        print("\nPOST Request berhasil dikirim!")
    else:
        print("\nPOST Request gagal.")
```

Gambar 4.11 skrip tes mengirim data

Pada baris awal, *header* X-Api-Key diisi dengan token aplikasi pengirim untuk diverifikasi oleh sistem. *Payload* JSON sederhana berisi pasangan atribut *car_name* dan *stock*, merepresentasikan skenario pengisian data inventori. Dengan perulangan for i in range(50), fungsi send_json_post_request() dipanggil berulang, setiap respons dievaluasi dan ditandai berhasil atau gagal melalui keluaran konsol.

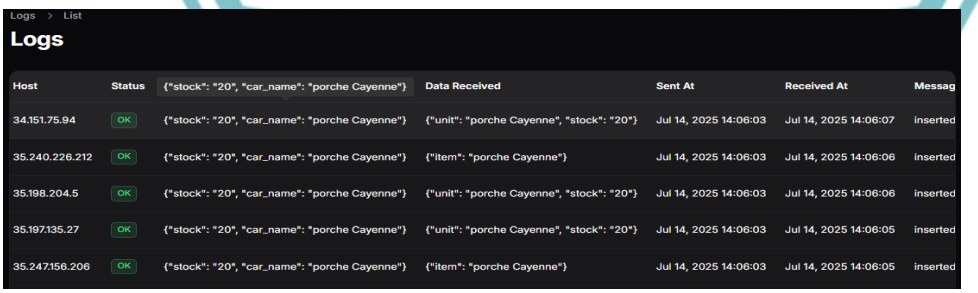
Pada Gambar 4.12 diperlihatkan hasil uji *skenario normal* gambar konsol MySQL dari 5 basis data tujuan:



Gambar 4.12 hasil uji skenario normal 5 database

Masing-masing menampilkan tepat 50 baris berurutan menunjukkan seluruh permintaan POST yang dikirim pada pengujian berhasil diproses tanpa kehilangan data. Konsol MySQL 1, 2, 3 menerima data *car_name* dan *stock* sedangkan konsol MySQL hanya membutuhkan data *car_name* saja.

Pada Gambar 4.13 tersaji tampilan tabel Logs yang mencatat setiap pengiriman data ke *database*.



Logs > List

Logs

Host	Status	{"stock": "20", "car_name": "porche Cayenne"}	Data Received	Sent At	Received At	Messag
34.151.75.94	OK	{"stock": "20", "car_name": "porche Cayenne"}	{"unit": "porche Cayenne", "stock": "20"}	Jul 14, 2025 14:06:03	Jul 14, 2025 14:06:07	inserted
35.240.226.212	OK	{"stock": "20", "car_name": "porche Cayenne"}	{"item": "porche Cayenne"}	Jul 14, 2025 14:06:03	Jul 14, 2025 14:06:06	inserted
35.198.204.5	OK	{"stock": "20", "car_name": "porche Cayenne"}	{"unit": "porche Cayenne", "stock": "20"}	Jul 14, 2025 14:06:03	Jul 14, 2025 14:06:06	inserted
35.197.135.27	OK	{"stock": "20", "car_name": "porche Cayenne"}	{"unit": "porche Cayenne", "stock": "20"}	Jul 14, 2025 14:06:03	Jul 14, 2025 14:06:05	inserted
35.247.156.206	OK	{"stock": "20", "car_name": "porche Cayenne"}	{"item": "porche Cayenne"}	Jul 14, 2025 14:06:03	Jul 14, 2025 14:06:05	inserted

Gambar 4.13 hasil log pengiriman data

Kolom Host menampilkan alamat IP *database* tujuan, pada kasus ini IP *database* 34.151.75.94, 35.198.204.5, 35.197.135.27 menerima data *unit* dan *stock*, sedangkan IP *database* 35.240.226.212 dan 35.247.156.206 hanya menerima data item.

4.4.3.2 Hasil Pengujian Partial Failure

Pada gambar 4.14 adalah data log jika *database* tujuan yang menerima data sedang dalam keadaan *down* dan gambar 4.15 adalah gambar jika data berhasil dikirim kembali saat *database tujuan sudah dalam keadaan* up.

34.151.75.94	OK	{"stock": "16", "car_name": "Nissan GTR"}	{"unit": "Nissan GTR", "stock": "16"}	Jul 15, 2025 01:57:57	Jul 15, 2025 01:58:05	inserted success
35.240.226.212	OK	{"stock": "16", "car_name": "Nissan GTR"}	{"item": "Nissan GTR"}	Jul 15, 2025 01:57:57	Jul 15, 2025 01:58:04	inserted success
35.198.204.5	RETRY	{"stock": "16", "car_name": "Nissan GTR"}	[]	Jul 15, 2025 01:57:57	Jul 15, 2025 01:58:04	held for retry: da
35.197.135.27	RETRY	{"stock": "16", "car_name": "Nissan GTR"}	[]	Jul 15, 2025 01:57:57	Jul 15, 2025 01:58:02	held for retry: da
35.247.156.206	OK	{"stock": "16", "car_name": "Nissan GTR"}	{"item": "Nissan GTR"}	Jul 15, 2025 01:57:57	Jul 15, 2025 01:57:59	inserted success

Gambar 4.14 hasil log pengiriman data

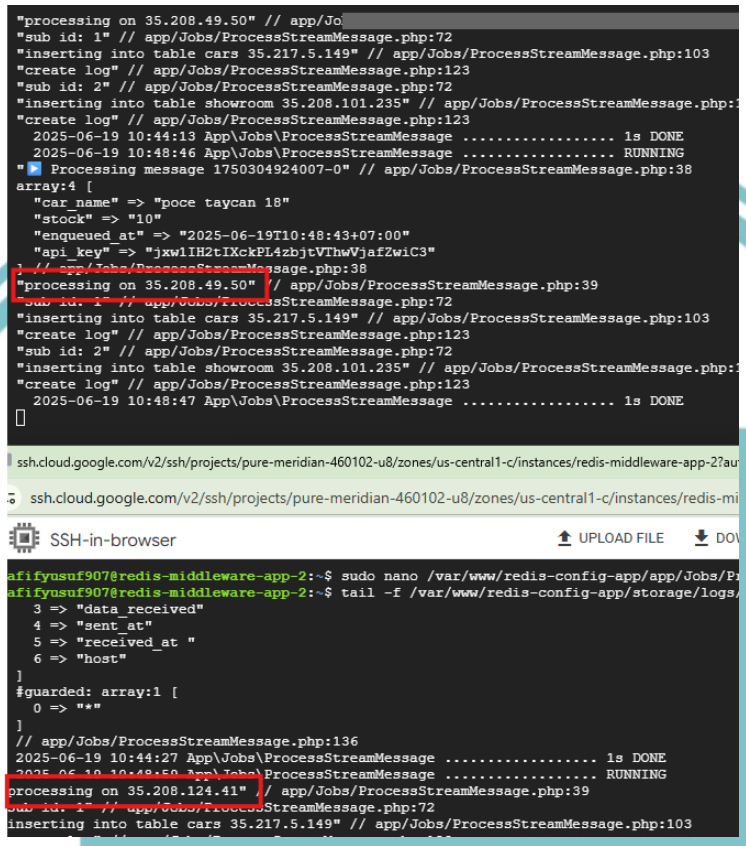
35.198.204.5	OK	{"unit": "Ferrari F8", "stock": "5"}	{"stock": "5"}	Jul 15, 2025 02:09:04	Jul 15, 2025 02:09:04	retried
35.197.135.27	OK	{"unit": "Nissan GTR", "stock": "16"}	{"stock": "16"}	Jul 15, 2025 02:09:03	Jul 15, 2025 02:09:03	retried

Gambar 4.15 hasil log pengiriman data

Pada Gambar 4.14 adalah hasil log ketika *database* tujuan dengan IP 35.198.204.5 dan 35.197.135.27 tidak tersedia. Kolom Status berkode RETRY dan kolom *Data Received* kosong, menandakan bahwa *payload* berhasil diambil dari Redis Stream tetapi *connection check* ke database gagal. Aplikasi *middleware* secara otomatis menandai pesan dengan keterangan "*held for retry: database unreachable*", lalu disimpan di kunci *retry* pada redis server. Sebaliknya, Gambar 4.15 menunjukkan kondisi setelah *database* tujuan dengan IP 35.198.204.5 dan 35.197.135.27 kembali *up*. Data log akan bertambah dan tidak mengubah status *retry* sebelumnya, kolom *Data Received* terisi *payload* dengan pesan "*retried*", yang berarti bahwa mekanisme *retry worker* berhasil menarik ulang entri dari kunci Redis, membuka koneksi ke database, serta mengirim data.

4.4.3.3 Hasil Pengujian *Load-Balancing*

Pada Gambar 4.16 menunjukan log dua VM yaitu redis-middleware-app1 dan redis-middleware-app2



Gambar 4.16 hasil load-balancing

Kedua VM tersebut secara bergantian menampilkan baris “processing on (IP)”, yang berarti *load balancer* berhasil mendistribusikan permintaan ke *instance* dengan utilisasi CPU terendah pada saat itu. Pola penyebaran trafik ini memastikan beban kerja tidak menumpuk pada satu VM saja, sehingga latensi terjaga dan risiko *throttling* berkurang.

Pada Gambar 4.17 ditampilkan hasil pengujian performa *load balancer* menggunakan Apache Benchmark dengan skenario 1000 permintaan konkuren (concurrency = 100) menuju endpoint /api/data

```
Benchmarking 34.117.154.57 (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests


Server Software:        Apache/2.4.52
Server Hostname:        34.117.154.57
Server Port:            80

Document Path:          /api/data
Document Length:        165 bytes

Concurrency Level:      100
Time taken for tests:   29.119 seconds
Complete requests:      1000
Failed requests:        0
Total transferred:      418000 bytes
Total body sent:        224000
HTML transferred:       165000 bytes
Requests per second:    34.34 [#/sec] (mean)
Time per request:       2911.872 [ms] (mean)
Time per request:       29.119 [ms] (mean, across all concurrent requests)
Transfer rate:          14.02 [Kbytes/sec] received
                        7.51 kb/s sent
                        21.53 kb/s total

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   0.8      0       4
Processing:  1202 2680 1470.3   2209    8403
Waiting:     1200 2673 1466.8   2206    8392
Total:       1202 2680 1470.7   2209    8406
WARNING: The median and mean for the initial connection time are not within a normal deviation
        These results are probably not that reliable.

Percentage of the requests served within a certain time (ms)
  50%   2209
  66%   2534
  75%   2802
  80%   3084
  90%   4248
  95%   6855
  98%   7424
  99%   7956
 100%   8406 (longest request)
```

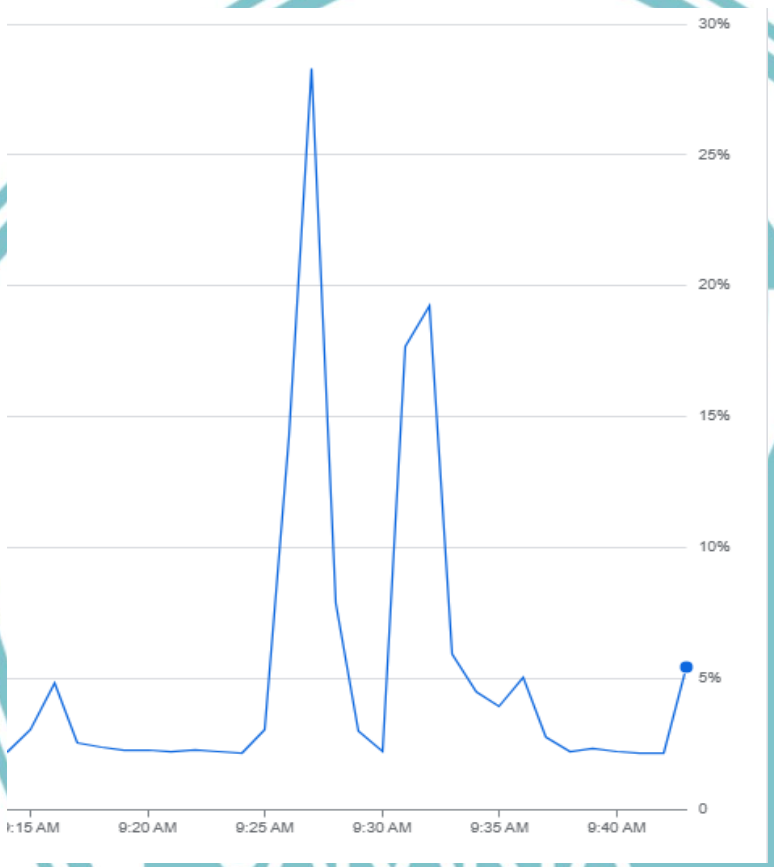
Gambar 4.17 hasil *benchmark* menggunakan Apache Benchmark

Dengan 1 000 *request* dan 100 konkurensi, server *load-balancing* menyelesaikan uji selama ≈ 29 detik. Rata-rata "requests per second" tercatat 34,34 req/s, latensi rata-rata tiap permintaan (individu) 29,12 ms, sedangkan waktu rata-rata untuk satu "batch" berisi 100 permintaan 2 911 ms. Tidak ada permintaan yang gagal, dan laju

transfer tercatat ≈ 14 KB/s diterima. Nilai-nilai ini menunjukkan bahwa aplikasi sanggup melayani beban sedang tanpa kesalahan.

4.4.3.4 Hasil Pengujian *Resource*

Pada Gambar 4.18 diperlihatkan grafik penggunaan *resource* CPU pada kedua VM aplikasi *middleware* selama uji *load-balancing*.



Gambar 4.18 penggunaan CPU *load-balancing*

Rata-rata pemanfaatan CPU VM *load-balancing* di GCP tercatat di bawah 10 %, dengan satu puncak 28 % dengan data uji sebesar 1000 request dan 100 konkurensi.

Tabel 4.3 adalah tabel pembanding penggunaan CPU yang merujuk pada penelitian yang dilakukan oleh Olejnik.

Tabel 4.3 perbandingan hasil uji CPU dengan yang dilakukan oleh Olejnik et al, 2021.

Parameter (–n 1000, –c 100)	**LB *Google Cloud***	**HAProxy (Olejnik et al.)**	**NGINX (Olejnik et al.)**	**LVS (Olejnik et al.)**
CPU peak	~ 28 %	~ 22 %	~ 27 %	~ 10 %
CPU average	< 10 %	≈ 18 %	≈ 23 %	≈ 7 %

Nilai puncak ini masih sebanding dengan utilisasi tertinggi NGINX (≈ 27 %) yang diteliti oleh Olejnik et al., sementara lebih tinggi ± 6 poin dibanding HAProxy (≈ 22 %) dan hampir tiga kali lipat dibanding LVS (≈ 10 %). Dengan demikian, konfigurasi GCP menunjukkan performa CPU yang masih berada dalam batas efisiensi. Dengan demikian, konfigurasi load balancer Google Cloud tetap dinilai efisien karena puncak utilisasi CPU-nya (~ 28 %) sebanding dengan NGINX (~ 27 %) dan masih lebih rendah secara relatif dibanding akumulasi rata-rata beban HAProxy (~ 22 %) maupun LVS (~ 10 %), sedangkan rata-rata pemakaiannya berada di bawah 10 %.

Gambar 4.19 menunjukan grafik penggunaan *resource memory* pada server redis yang diambil dari monitoring *Google cloud* saat pengujian skenario normal.

Gambar 4.19 grafik penggunaan memory di server redis pada skenario normal

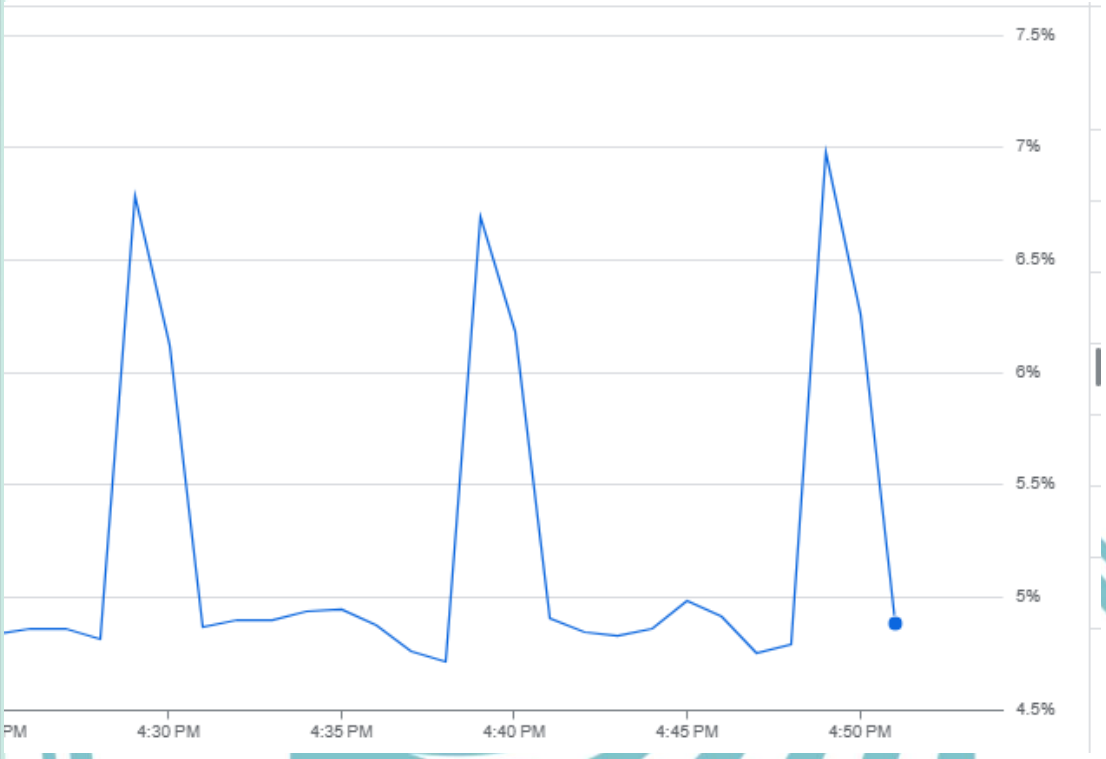
Tabel 4.4 adalah tabel pembanding yang merujuk pada penelitian yang dilakukan oleh Al-Allawee.

Tabel 4.4 perbandingan hasil uji dengan yang dilakukan oleh Al-Allawee.

No	Basis data	Penggunaan memori (persentase terhadap RAM)
1	Local cache (cachedb_local)	7,6 %
2	Memcached	8,8 %
3	Redis	8,9 %
4	Redis – hasil uji GCP (VM 8 GB RAM, 2 vCPU, 1 000 pesan)	5,9 %

Berdasarkan gambar 4.19 dan tabel 4.3 yang disajikan, pengujian resource server redis dilakukan dengan mengirim 1000 *request,* penggunaan memori pada skenario normal dilaporkan berada pada kisaran 7,6 % untuk *local cache*, 8,8 % untuk Memcached, dan 8,9 % untuk Redis pada penelitian terdahulu. Ketika skenario serupa diuji ulang dalam lingkungan GCP dengan *instance* Redis berkapasitas 8 GB RAM dan 2 vCPU, konsumsi memori terukur hanya 5,9 %, sehingga tercatat lebih rendah 1,7–3 poin persentase dibanding ketiga konfigurasi pada jurnal yang dibuat oleh Al-Allawee. Selisih ini mengindikasikan bahwa optimasi konfigurasi, perbedaan versi perangkat lunak, serta karakteristik beban kerja di platform GCP telah memungkinkan pemanfaatan sumber daya yang lebih efisien pada sisi memori, walaupun jenis basis data dan pola operasinya tetap sama, yakni penulisan serempak oleh banyak klien.

Gambar 4.20 menunjukan grafik penggunaan *resource memory* pada server redis yang diambil dari monitoring *Google cloud* pada pengujian *partial failure*.



Gambar 4.20 grafik penggunaan memory di server redis pada skenario *partial-failure*

Pada gambar 4.20 menunjukan grafik fluktuasi yang puncak nya ada pada 7% karena setelah server redis melakukan *write* untuk menyimpan data sementara dan *read* untuk dibaca datanya, pada skenario *partial failure* aplikasi *middleware* akan kembali mengakses data di server redis untuk melakukan pengiriman kembali kepada *database* tujuan yang sebelumnya *down* menjadi *up* yang berarti terdapat 3 proses yang dilakukan kepada server redis pada skenario *partial failure* yang menyebabkan lonjakan penggunaan RAM..

BAB V
PENUTUP

5.1 Kesimpulan

1. Berdasarkan implementasi dan hasil uji, *load-balancing* berhasil menyebarkan *traffic* ke 5 VM untuk memecah beban yang diproses oleh VM.
2. Berdasarkan implementasi dan hasil uji, penggunaan *resource* memory pada server redis saat uji skenario normal menunjukan hasil penggunaan sebesar 5,9 % sehingga tercatat lebih rendah 1,7–3 poin persentase dibanding ketiga konfigurasi pada jurnal pembanding dan menunjukan hasil sebesar 7% saat uji skenario *partial-failure.*
3. Berdasarkan implementasi dan hasil uji, data akan tetap tersimpan dalam server redis dan akan dikirim kembali apabila *database* tujuan sudah dalam keadaan *up.*
4. Berdasarkan implementasi dan hasil uji, sistem dapat mendistribusikan data dengan lengkap ke 5 *database* MySQL

5.2 Saran

1. Tambahkan fitur *update* ke *database* tujuan.
2. Tambahkan jumlah VM sebagai server *middleware*.
3. Terapkan *autoscaling* untuk *load-balancing.*

DAFTAR PUSTAKA

Yohanssen Pratama (2021) *SISTEM TERDISTRIBUSI - Yohanssen Pratama, S.Si., M.T. - Google Books*. Edited by Masyrifatul Khairiyyah. Available at: https://books.google.co.id/books?hl=en&lr=&id=bbFCEAAAQBAJ&oi=fnd&pg=PP1&dq=sistem+terdistribusi&ots=sX_spVNbLd&sig=5CYKNBS4CHoaBVhbq53L6wEESEw&redir_esc=y#v=onepage&q=sistem%20terdistribusi&f=false.

S Zhu *et al.* (2023) "A Redis Cluster Fault Resolution Method Based on Sentinel Mechanism for IoT Management Platform." Available at: https://iopscience.iop.org/article/10.1088/1742-6596/2476/1/012047/pdf.

Febriyani, F., Sakti Pramukantoro, E. and Bakhtiar, F.A. (2019) *Perbandingan Kinerja Redis, Mosquitto, dan MongoDB sebagai Message Broker pada IoT Middleware*. Available at: http://j-ptiik.ub.ac.id.

Rachel, T. and Susetyo, Y.A. (no date) "Implementasi Pengiriman Pesan Broadcast dengan Redis Pub/Sub dan Bahasa Pemrograman Nim," 7(1), p. 2022.

Eddelbuettel, D. (2022) "A Brief Introduction to Redis." Available at: https://doi.org/10.1080/00031305.2017.1375990.

Claudiyap, J.M., Ocsa, P. and Saian, N. (2022) *IMPLEMENTASI SISTEM BROADCAST MESSAGE MENGGUNAKAN PYTHON DAN REDIS PUB/SUB*.

Doshi, R. *et al.* (2024) "Distributed MQTT Broker: A Load-Balanced Redis-Based Architecture," in *2024 International Conference on Emerging Smart Computing and Informatics, ESCI 2024*. Institute of Electrical and Electronics Engineers Inc. Available at: https://doi.org/10.1109/ESCI59607.2024.10497427.

Weerasinghe, S. and Perera, I. (2023) *Optimized Strategy for Inter-Service Communication in Microservices*, *IJACSA) International Journal of*

Advanced Computer Science and Applications. Available at: www.ijacsa.thesai.org.

Muniswamaiah, M., Agerwala, T. and Tappert, C.C. (2023) "Comparison of SQL, NoSQL, and NewSQL Database Technologies," in *Proceedings - 2023 IEEE International Conference on Big Data, BigData 2023*. Institute of Electrical and Electronics Engineers Inc., pp. 6230–6232. Available at: https://doi.org/10.1109/BigData59044.2023.10386522.

Chao-Hsien Hsieh *et al.* (no date) *Development of laravel digital platform based on MVC design pattern for compii cated data structure-take the Bible for example | IEEE conference publication | IEEE Xplore*. Available at: https://ieeexplore.ieee.org/document/9232045.

Akanbi, A. & Masinde, M., 2020. A Distributed Stream Processing Middleware Framework for Real-Time Analysis of Heterogeneous Data on Big Data Platform: Case of Environmental Monitoring. *Sensors*, 20(11), p.3166.

Shahakar, M.S., Patil, L. & Mahajan, S., 2023. A survey on load balancing in distributed systems. *International Journal on Recent and Innovation Trends in Computing and Communication*, 11(2).

Liu, G. et al., 2025. ClusterBalance: Adaptive Load Balancing in Distributed Systems via Real-Time Clustering. *International Journal of Research Publication and Reviews*, 6(4), pp.12795–12803.

Saputra, R. B. and Subektiningsih, S. (2023) "Comparative Analysis of Apache 2 Performance in Docker Containers vs Native Environment", *Jurnal Ilmiah Teknik Elektro Komputer dan Informatika*, 9(4), pp. 1024–1034. doi: 10.26555/jiteki.v9i4.27220.

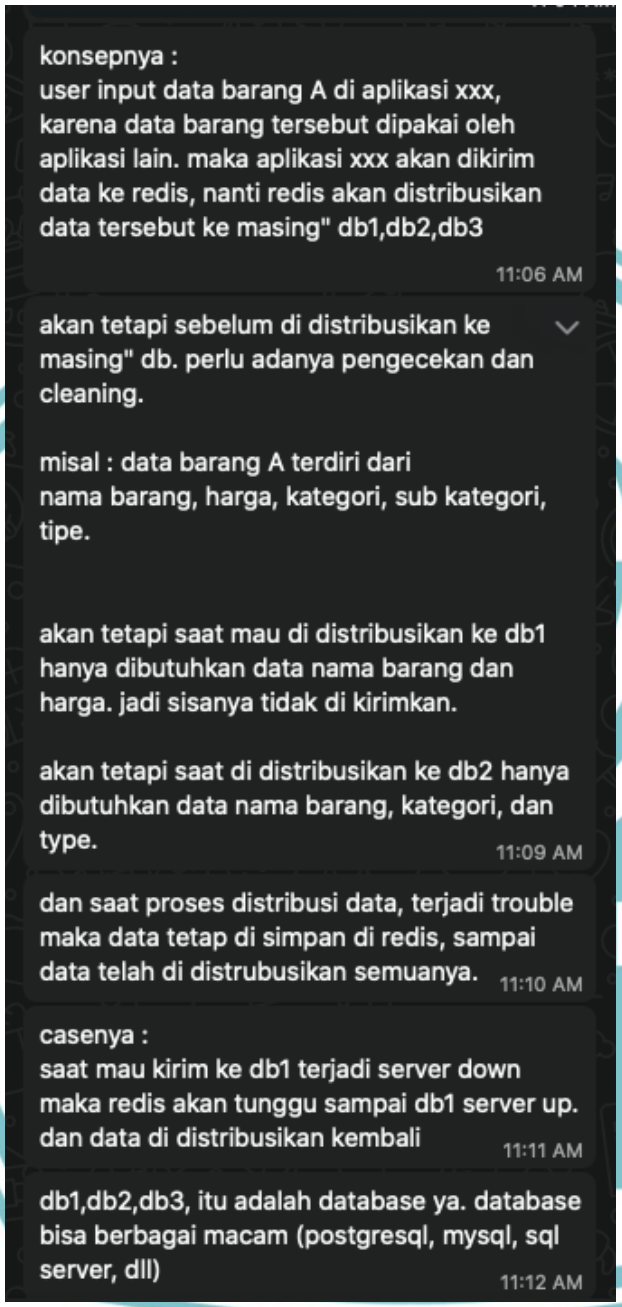
Narsipuram, M., Rai, A. and Tiwari, A. (2024) "A Comprehensive Study of Load Balancing Architectures in Cloud Computing Citation," *Smart. Internet.*

Things, 1(2), pp. 115–127. Available at: https://doi.org/10.22105/SA.2021.281500.1061.

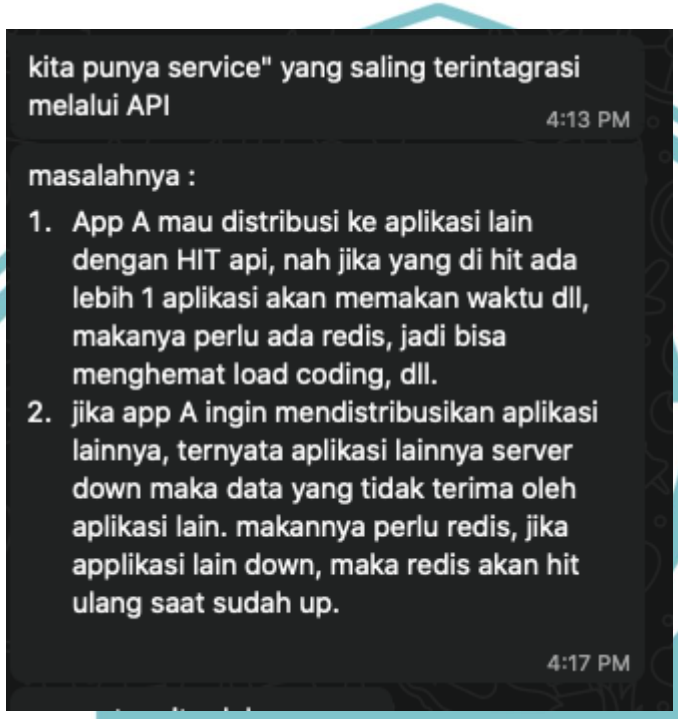
Al-Allawee, A., Lorenz, P., Abouaissa, A. and Abualhaj, M. (2023) 'A performance evaluation of in-memory databases operations in Session Initiation Protocol', *Network*, 3(1), pp. 1–14. doi:10.3390/network3010001.

Olejnik, **P.**, Zajac, M. & Franuskiewicz, A., 2021. *Comparative Analysis of Selected Open-Source Solutions for Traffic Balancing in Server Infrastructures Providing WWW Service*. **Energies**, 14(22), p.7719. DOI: 10.3390/en14227719.

Lampiran



(lanjutan)



Middleware Database

- Tambah fitur update (saat ini baru action insert)

Menu DB Configuration:

- Support: pgsql & mysql
- Password harus diinput sebelum di save
- Tambah fitur: Check connection DB sebelum save

Menu Database Tables

- Tambah fitur Select query tables dari database, agar tidak input manual menghindari human error

Phase II:

- Buat fungsi heartbeat untuk cek koneksi in case ada issue koneksi/downtime database, bisa di system berbeda
- Optimasi queue karena ada 1 job bisa 5-8 detik
- Pengolahan data