ANALISIS PERFORMA RESTFUL API DENGAN REDIS CACHING PADA BACKEND GOLANG

PERFORMANCE ANALYSIS OF RESTFUL API WITH REDIS CACHING ON GOLANG BACKEND

Krisna Eko Prasetyo[1*], Abdul Rezha Efrat Najaf[2], Iqbal Ramadhani Mukhlis[3]
*E-mail: 22082010149@student.upnjatim.ac.id

[1,2,3]Sistem Informasi, Fakultas Ilmu Komputer, UPN "Veteran" Jawa Timur

Abstrak

RESTful API pada sistem transaksi berbasis web umumnya menerima permintaan baca berulang terhadap data yang relatif jarang berubah, seperti katalog produk. Jika setiap permintaan langsung diarahkan ke basis data relasional melalui ORM, proses tersebut berpotensi meningkatkan latensi dan menurunkan *throughput* ketika jumlah pengguna bertambah. Penelitian ini bertujuan menganalisis pengaruh penerapan Redis sebagai *in-memory caching layer* terhadap performa RESTful API pada *backend* Golang berbasis Go Fiber dan GORM dengan PostgreSQL. Metode yang digunakan adalah eksperimen kuantitatif komparatif terkontrol melalui *load testing* menggunakan K6. Pengujian dirancang pada tiga skenario, yaitu tanpa cache, Redis *cold cache*, dan Redis *warm cache* dengan pola *cache-aside*. Metrik evaluasi meliputi *average response time*, p90, p95, *throughput*, *error rate*, *cache hit ratio*, dan *database hit rate*. Hasil pengujian menunjukkan bahwa skenario tanpa cache menghasilkan rata-rata *response time* 66.8231 ms, p95 160.8055 ms, *throughput* 1388.4441 req/s, dan *error rate* 0.0014%. Pada skenario Redis *cold cache*, rata-rata *response time* menjadi 9.5016 ms, p95 21.4390 ms, *throughput* 9507.3111 req/s, dan *error rate* 0.0000%. Pada skenario Redis *warm cache*, rata-rata *response time* menjadi 9.5162 ms, p95 21.5285 ms, *throughput* 9445.0949 req/s, dan *error rate* 0.0000%. Temuan ini menunjukkan bahwa Redis *caching* berpotensi menjadi strategi optimasi yang efektif untuk *endpoint* RESTful API dengan pola akses baca berulang pada *backend* Golang.

Kata kunci*: *Redis, caching, RESTful API, Golang, load testing

Abstract

RESTful APIs in web-based transactional systems commonly receive repeated read requests for relatively stable data, such as product catalogs. When each request is directed to a relational database through an ORM, the process may increase latency and reduce throughput as the number of users grows. This study aims to analyze the effect of Redis as an in-memory caching layer on the performance of RESTful APIs in a Golang backend built with Go Fiber, GORM, and PostgreSQL. The method used is a controlled comparative quantitative experiment through load testing with K6. The test is designed under three scenarios: no cache, Redis cold cache, and Redis warm cache using the cache-aside pattern. The evaluated metrics include average response time, p90, p95, throughput, error rate, cache hit ratio, and database hit rate. The test results show that the no-cache scenario produced an average response time of 66.8231 ms, p95 of 160.8055 ms, throughput of 1388.4441 req/s, and error rate of 0.0014%. In the Redis cold cache scenario, the average response time was 9.5016 ms, p95 was 21.4390 ms, throughput was 9507.3111 req/s, and error rate was 0.0000%. In the Redis warm cache scenario, the average response time was 9.5162 ms, p95 was 21.5285 ms, throughput was 9445.0949 req/s, and error

rate was 0.0000%. These findings indicate that Redis caching can be an effective performance optimization strategy for read-heavy RESTful API endpoints in the Golang backend ecosystem.

***Keywords**: Redis, caching, RESTful API, Golang, load testing.*

1. PENDAHULUAN

RESTful API merupakan komponen penting dalam sistem informasi modern karena berperan sebagai kontrak komunikasi antara *client* dan *backend*. Pada aplikasi transaksi berbasis web, *endpoint* baca seperti katalog produk, detail produk, dan informasi ketersediaan sering diakses secara berulang oleh banyak pengguna. Apabila setiap permintaan diarahkan langsung ke basis data relasional, maka lapisan *database* berpotensi menjadi *bottleneck*, terutama ketika *query* yang sama dieksekusi berulang pada kondisi beban tinggi.

Caching menjadi salah satu strategi optimasi yang banyak digunakan untuk mengurangi latensi akses data dan menekan frekuensi *query* ke basis data utama. Redis relevan digunakan karena bekerja sebagai *in-memory data store* yang menyimpan data pada memori sehingga waktu aksesnya lebih rendah dibandingkan akses berulang ke penyimpanan relasional berbasis disk [1]. Pada level aplikasi, strategi *caching* berbasis in-memory menggunakan Redis juga dilaporkan mampu mempercepat akses data relasional melalui penyimpanan sementara hasil *query* [2].

Pada konteks API *e-commerce*, penerapan Redis pada kondisi *cold cache* dan *warm cache* terbukti dapat menurunkan *response time* dan meningkatkan *throughput* dibandingkan API tanpa Redis [3]. Pada RESTful API sistem akademik, Redis juga dilaporkan mampu menjaga waktu respons tetap rendah pada beban konkuren tinggi [4]. Dalam arsitektur *microservices*, Redis Cluster menunjukkan potensi peningkatan *throughput* sekaligus penurunan beban *database* [5]. Temuan-temuan tersebut memperlihatkan bahwa *caching* berbasis Redis berpotensi menurunkan waktu respons, meningkatkan *throughput*, dan mengurangi ketergantungan API terhadap *query* basis data pada beban akses baca berulang.

Meskipun penelitian terdahulu telah menunjukkan efektivitas Redis dalam meningkatkan performa API, ruang kajian yang dekat dengan topik ini masih memiliki keterbatasan konteks teknologi. Studi pada API *e-commerce* menguji Redis *caching* dalam ekosistem Express.js dengan membandingkan kondisi tanpa Redis, *cold cache*, dan *warm cache* [3]. Studi lain pada RESTful API sistem akademik juga masih berfokus pada Express.js dengan pengujian beban menggunakan Apache JMeter [4]. Sementara itu, penelitian mengenai Redis Cluster lebih menitikberatkan pada arsitektur *microservices*, bukan pada *backend* monolitik atau modular berbasis Golang [5]. Dengan demikian, masih terdapat celah penelitian untuk menguji efektivitas Redis *caching* pada RESTful API berbasis Golang, khususnya pada kombinasi Go Fiber, GORM, PostgreSQL, dan pola *cache-aside*.

Konteks Golang menjadi penting karena studi komparatif *backend framework* menunjukkan bahwa ekosistem Go memiliki potensi performa yang kompetitif untuk REST API [6]. Selain itu, implementasi API menggunakan Golang dengan pemisahan *layer* seperti *handler*, *use case*, dan *repository* telah digunakan dalam penelitian terkait Clean Architecture [7]. Implementasi REST API berbasis Golang juga telah diterapkan pada sistem *sandbox* pihak ketiga, sehingga memperkuat relevansi Golang untuk pengembangan layanan *backend* berbasis HTTP [8]. Namun, penelitian-penelitian tersebut belum secara khusus mengevaluasi dampak Redis *caching* terhadap *response time*, *throughput*, dan *error rate* pada RESTful API berbasis Golang dengan akses data melalui GORM dan PostgreSQL.

Berdasarkan celah tersebut, penelitian ini menganalisis performa RESTful API dengan Redis *caching* pada *backend* Golang berbasis Go Fiber dan GORM dengan PostgreSQL. Fokus

penelitian diarahkan pada *endpoint* baca berulang agar dampak cache terhadap *response time*, *throughput*, *error rate*, *cache hit ratio*, dan *database hit rate* dapat diamati secara terukur. Kebaruan penelitian ini terletak pada pengujian komparatif Redis *caching* dalam ekosistem *backend* Golang, bukan pada Express.js atau arsitektur *microservices*, dengan membandingkan skenario tanpa cache, Redis *cold cache*, dan Redis *warm cache*. Kontribusi penelitian ini adalah menyediakan bukti empiris serta rancangan eksperimen yang dapat direplikasi untuk mengevaluasi efektivitas Redis sebagai strategi optimasi performa pada RESTful API berbasis Golang.

2. METODOLOGI

2.1 Desain Penelitian

Penelitian ini menggunakan pendekatan eksperimen kuantitatif dengan desain komparatif terkontrol. Variabel perlakuan dalam eksperimen adalah penggunaan Redis *caching*, sedangkan variabel yang diukur adalah performa RESTful API. Tiga skenario pengujian dibandingkan, yaitu skenario tanpa cache, skenario Redis *cold cache*, dan skenario Redis *warm cache*. Seluruh skenario dijalankan pada *endpoint*, dataset, konfigurasi server, durasi, dan pola beban yang sama agar perbedaan hasil dapat dikaitkan secara lebih objektif dengan penerapan *caching*.

Desain eksperimen ini dipilih karena tujuan penelitian bukan hanya membangun *backend*, tetapi mengukur perubahan performa secara empiris. K6 dipilih karena mendukung penulisan skenario uji berbasis *script* dan telah digunakan pada evaluasi performa *web server* dengan metrik *response time*, *throughput*, *error rate*, dan *requests per second* [9]. Secara konseptual, *load testing* digunakan untuk mengukur respons sistem pada kondisi beban yang dirancang secara sistematis [10].

2.2 Objek dan Lingkungan Pengujian

Objek penelitian adalah RESTful API *backend* Golang pada sistem Nusantara Kuliner. *Backend* dikembangkan menggunakan Go Fiber sebagai HTTP *framework*, GORM sebagai ORM, PostgreSQL sebagai basis data utama, dan Redis sebagai lapisan cache. *Endpoint* yang diuji difokuskan pada *endpoint* baca berulang, misalnya *endpoint* daftar produk atau katalog produk, karena jenis *endpoint* tersebut memiliki karakteristik *read-heavy* dan relatif sesuai untuk mekanisme *cache-aside*. GORM digunakan sebagai ORM karena menyediakan abstraksi pemetaan model, *query*, migrasi, dan transaksi pada ekosistem Go [11].

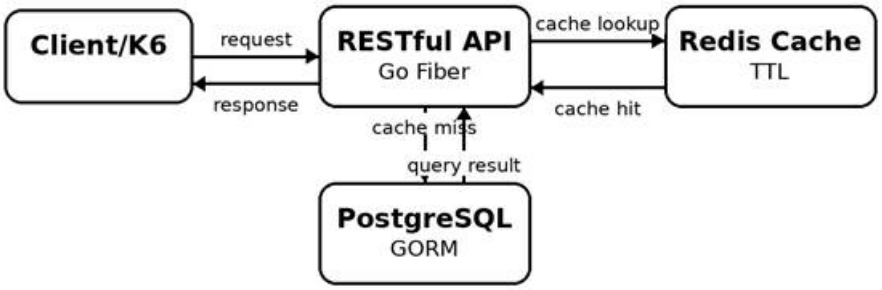
Table 1. Komponen Lingkungan Pengujian

Komponen	Spesifikasi/Keterangan
Backend	Golang dengan *framework* Go Fiber
ORM	GORM untuk akses PostgreSQL [11]
Database	PostgreSQL
Cache	Redis dengan pola *cache-aside*
Load testing tool	K6
Endpoint uji	GET *endpoint* katalog/produk *read-heavy*
Jumlah data uji	10000 record data produk/katalog
Spesifikasi server	CPU 12 vCPU (12th Gen Intel(R) Core(TM) i5-12450HX), RAM 7.6 GiB, OS Ubuntu 24.04.4 LTS
Backend	Golang dengan *framework* Go Fiber

2.3 Mekanisme Redis Cache-aside

Penerapan Redis pada penelitian ini menggunakan pola *cache-aside*. Pada pola ini, API tidak selalu langsung mengakses *database*. Setiap request terlebih dahulu memeriksa ketersediaan

data pada Redis menggunakan *cache key* tertentu. Jika data ditemukan atau *cache hit*, API langsung mengembalikan data dari Redis. Jika data tidak ditemukan atau *cache miss*, API mengambil data dari PostgreSQL melalui GORM, menyimpan hasil *query* ke Redis dengan *time-to-live* (TTL), kemudian mengembalikan respons ke *client*. Pola ini selaras dengan studi pada RESTful API yang menggunakan Redis untuk mengalihkan sebagian beban *query* relasional ke penyimpanan in-memory [4]. Pendekatan serupa pada API *e-commerce* juga menunjukkan bahwa Redis dapat mengurangi ketergantungan API terhadap *query* basis data berulang [3].



Gambar 1. Alur *cache-aside* Redis pada RESTful API

Secara operasional, alur *cache-aside* pada Gambar 1 dimulai ketika *client* mengirimkan request ke RESTful API untuk memperoleh data yang dibutuhkan. Setelah request diterima, API terlebih dahulu melakukan pencarian data pada Redis berdasarkan *cache key* yang telah ditentukan. Apabila data tersedia di Redis, sistem langsung mengembalikan respons dari cache sehingga proses akses ke basis data tidak diperlukan. Sebaliknya, apabila data tidak tersedia di Redis, API akan mengambil data dari PostgreSQL melalui GORM, kemudian hasil *query* tersebut disimpan kembali ke Redis dengan TTL tertentu agar dapat digunakan pada request berikutnya. Dengan alur tersebut, Redis berperan sebagai lapisan penyimpanan sementara yang mengurangi frekuensi akses langsung ke basis data dan mempercepat respons pada pola permintaan data yang berulang.

2.4 Skenario Pengujian

Pengujian dilakukan dalam tiga skenario agar efek cache dapat diamati secara lebih rinci. Skenario tanpa cache digunakan sebagai *baseline* karena seluruh request diarahkan langsung ke PostgreSQL. Skenario Redis *cold cache* digunakan untuk melihat perilaku sistem ketika Redis aktif tetapi cache dikosongkan sebelum pengujian. Skenario Redis *warm cache* digunakan untuk melihat performa ketika data telah tersedia pada Redis sebelum pengujian dimulai.

Table 2. Skenario Pengujian Performa

Skenario	Kondisi Cache	Tujuan Pengujian
S1	Tanpa Redis; *query* langsung ke PostgreSQL melalui GORM	Menetapkan *baseline* performa API tanpa *caching*
S2	Redis aktif tetapi cache dikosongkan sebelum pengujian	Mengamati performa *cold cache* dan proses cache warm-up
S3	Redis aktif dan data telah dipanaskan sebelum pengujian	Mengamati performa *warm cache* saat *cache hit* dominan

Skenario beban disusun menggunakan K6 dengan pola *ramp-up*, *steady load*, dan *ramp-down*. Pola ini dipilih untuk mensimulasikan kenaikan jumlah pengguna, kondisi beban stabil, dan penurunan beban secara terkendali. K6 dipilih karena mendukung penulisan skenario uji

berbasis *script* dan telah digunakan pada evaluasi performa *web server* dengan metrik *response time*, *throughput*, *error rate*, dan *requests per second* [9].

Table 3. Rancangan Beban K6

Tahap	Durasi	Target *Virtual user*	Tujuan
Ramp-up	30 detik	0–100 VU	Menaikkan beban secara bertahap
Steady load	3 menit	100 VU	Mengukur performa pada beban stabil
Ramp-down	30 detik	100–0 VU	Menutup pengujian secara terkendali

2.5 Metrik Evaluasi

Metrik utama yang dianalisis meliputi *average response time*, median *response time*, p90, p95, *throughput*, *error rate*, dan total request. *Average response time* digunakan untuk melihat kecenderungan umum waktu respons. Nilai p90 dan p95 digunakan untuk mengevaluasi stabilitas layanan pada sebagian besar request, sedangkan throughput menunjukkan jumlah request yang dapat diproses per detik. *Error rate* digunakan untuk mengukur proporsi request gagal selama pengujian.

Selain metrik K6, penelitian ini juga mencatat *cache hit ratio*, *cache miss ratio*, dan *database hit rate*. *Cache hit ratio* menunjukkan proporsi request yang berhasil dilayani dari Redis, *cache miss ratio* menunjukkan proporsi request yang tidak menemukan data di Redis, sedangkan *database hit rate* menunjukkan proporsi request yang tetap harus mengakses PostgreSQL. Ketiga metrik tambahan tersebut penting untuk memastikan bahwa perubahan performa benar-benar berkaitan dengan efektivitas *caching*, bukan hanya variasi beban sesaat.

Persentase perubahan performa dihitung menggunakan rumus berikut:

Penurunan response time (%) = ((RT tanpa cache - RT Redis) / RT tanpa cache) x 100%

Peningkatan throughput (%) = ((Throughput Redis - Throughput tanpa cache) / Throughput tanpa cache) x 100%

2.6 Prosedur Validasi Eksperimen

Untuk menjaga validitas internal, seluruh skenario dijalankan pada lingkungan yang sama. Sebelum skenario tanpa cache dan *cold cache* dijalankan, Redis dikosongkan dan aplikasi dikembalikan ke kondisi awal. Untuk skenario *warm cache*, *endpoint* dipanggil terlebih dahulu hingga data tersimpan di Redis. Setiap skenario dijalankan minimal tiga kali sehingga nilai rata-rata dan variasi hasil dapat diamati. Log hasil K6 disimpan dalam format JSON atau CSV untuk memudahkan rekapitulasi metrik. Pada penelitian ini, setiap skenario pengujian dijalankan sebanyak tiga kali, yaitu run-1, run-2, dan run-3, kemudian hasilnya dirata-ratakan untuk memperoleh nilai agregat yang lebih stabil serta mengurangi pengaruh variasi sesaat selama pengujian.

3. HASIL DAN PEMBAHASAN

3.1 Hasil Pengujian Performa API

Hasil pengujian K6 pada ketiga skenario disajikan pada Tabel 4. Tabel tersebut memuat metrik *average response time*, median, p90, p95, throughput, *error rate*, dan total request untuk skenario tanpa cache, Redis *cold cache*, dan Redis *warm cache*.

Seluruh nilai performa yang disajikan pada Table 4 dan Table 5 merupakan hasil agregasi dari tiga kali pengujian pada setiap skenario. Pendekatan ini digunakan untuk memperoleh gambaran performa yang lebih representatif serta meminimalkan bias akibat fluktuasi sesaat pada lingkungan pengujian.

Table 4. Hasil Pengujian Menggunakan K6

Skenario	Tanpa cache	Redis *cold cache*	Redis *warm cache*
Avg RT (ms)	66.8231	9.5016	9.5162
Median (ms)	56.1245	7.9007	7.9506
p90 (ms)	129.9636	17.9118	17.9643
p95 (ms)	160.8055	21.4390	21.5285
Throughput (req/s)	1388.4441	9507.3111	9445.0949
***Error rate* (%)**	0.0014	0.0000	0.0000
Total Request	313707.6667	2154071.0000	2138619.3333

3.2 Analisis Perbandingan Antarskenario

Perbandingan performa dilakukan dengan menghitung perubahan metrik utama antara skenario tanpa cache dan skenario Redis. Hasil perhitungan persentase perubahan disajikan pada Table 5. Nilai positif pada *throughput* menunjukkan peningkatan kapasitas pemrosesan request, sedangkan nilai positif pada penurunan *response time* menunjukkan perbaikan waktu respons API.

Table 5. Persentase Perubahan Performa

Perbandingan	Penurunan Avg RT (%)	Penurunan p95 (%)	Peningkatan *Throughput* (%)	Perubahan *Error rate*
Tanpa cache vs Redis *cold cache*	85.7810	86.6677	584.7457	0.0014% ke 0.0000%
Tanpa cache vs Redis *warm cache*	85.7591	86.6121	580.2647	0.0014% ke 0.0000%

Metrik cache pada Tabel 6 digunakan untuk memperkuat interpretasi performa. *Cache hit ratio* pada skenario Redis *cold cache* dan Redis *warm cache* sama-sama mencapai 99.9865%, sedangkan *cache miss ratio* dan DB hit rate sama-sama 0.0135%. Nilai tersebut menunjukkan bahwa mayoritas request pada skenario Redis dilayani dari cache, sehingga penurunan latensi dan peningkatan *throughput* berkaitan langsung dengan berkurangnya akses ke PostgreSQL.

Table 6. Rekapitulasi Efektivitas Cache

Skenario	*Cache hit ratio* (%)	*Cache miss ratio* (%)	DB Hit Rate (%)
Redis *cold cache*	99.9865	0.0135	0.0135
Redis *warm cache*	99.9865	0.0135	0.0135

3.3 Pembahasan

Berdasarkan Tabel 4 dan Tabel 5, penerapan Redis *caching* menunjukkan penurunan *response time* dan peningkatan *throughput* yang signifikan dibandingkan skenario tanpa cache. Pada skenario Redis *cold cache*, avg RT turun 85.7810% dan p95 turun 86.6677%, sedangkan *throughput* meningkat 584.7457%. Pada skenario Redis *warm cache*, avg RT turun 85.7591% dan p95 turun 86.6121%, sedangkan *throughput* meningkat 580.2647%. Peningkatan tersebut terjadi karena sebagian besar request tidak lagi mengeksekusi *query* ke PostgreSQL; pada

kondisi *cache hit*, API cukup mengambil data dari Redis yang berada di memori sehingga jalur akses data menjadi lebih pendek.

Perbedaan antara *cold cache* dan *warm cache* pada hasil penelitian ini relatif kecil. Redis *cold cache* mencatat avg RT 9.5016 ms dan *throughput* 9507.3111 req/s, sedangkan Redis *warm cache* mencatat avg RT 9.5162 ms dan *throughput* 9445.0949 req/s. Selisih tersebut tidak perlu ditafsirkan sebagai bukti bahwa *cold cache* selalu lebih unggul daripada *warm cache*, melainkan sebagai variasi eksperimen yang masih berada pada rentang sangat dekat. Secara mekanisme, *cold cache* tetap mengalami miss awal ketika data belum tersedia di Redis, sedangkan *warm cache* merepresentasikan kondisi cache-hit dominan karena data telah dipanaskan sebelum pengujian.

Hasil penelitian ini perlu dibandingkan secara hati-hati dengan penelitian terdahulu karena perbedaan stack teknologi, dataset, lingkungan server, durasi pengujian, dan jumlah pengguna dapat memengaruhi hasil. Pola penurunan latency dan peningkatan *throughput* yang diperoleh pada penelitian ini sejalan dengan studi Redis *caching* pada API *e-commerce* berbasis Express.js [3]. Kecenderungan peningkatan reliabilitas API melalui Redis juga ditemukan pada RESTful API sistem akademik dengan beban pengguna tinggi [4]. Pada arsitektur *microservices*, Redis Cluster dilaporkan mampu menurunkan beban *database* sekaligus meningkatkan *throughput* layanan [5].

Rekapitulasi efektivitas cache pada Tabel 6 mendukung interpretasi tersebut. *Cache hit ratio* sebesar 99.9865% pada kedua skenario Redis menunjukkan bahwa hampir seluruh request baca berhasil dilayani dari Redis. *Cache miss ratio* dan DB hit rate sebesar 0.0135% menunjukkan bahwa akses ke PostgreSQL hanya terjadi pada sebagian sangat kecil request, sehingga Redis efektif menekan beban *query* langsung ke *database* selama pengujian.

Dari sisi arsitektur, penggunaan Redis tidak menghilangkan kebutuhan terhadap PostgreSQL sebagai sumber kebenaran utama. Redis berfungsi sebagai lapisan akselerasi baca, sehingga desain TTL dan strategi invalidasi tetap perlu dirancang secara hati-hati. Untuk data yang sering berubah, cache yang tidak diperbarui berpotensi menghasilkan *stale data*. Oleh karena itu, *endpoint* yang paling sesuai untuk *caching* adalah *endpoint* dengan karakteristik data relatif stabil dan intensitas baca tinggi, seperti katalog produk, detail produk, atau data referensi.

Keterbatasan penelitian ini terletak pada ruang lingkup pengujian yang berfokus pada *endpoint* baca dan metrik performa sisi *client*. Apabila pengukuran CPU, memori, dan konsumsi sumber daya Redis belum dilakukan, maka kesimpulan tentang efisiensi infrastruktur harus dinyatakan secara terbatas. Pengujian lebih lanjut pada *endpoint* tulis, strategi invalidasi cache, serta skenario *multi-node* Redis dapat memperluas validitas eksternal penelitian.

4. KESIMPULAN DAN SARAN

Penelitian ini merancang dan menjalankan eksperimen komparatif untuk menganalisis pengaruh Redis *caching* terhadap performa RESTful API pada *backend* Golang berbasis Go Fiber, GORM, dan PostgreSQL. Berdasarkan hasil pengujian aktual, penerapan Redis *caching* dengan pola *cache-aside* pada skenario *warm cache* menghasilkan penurunan rata-rata *response time* sebesar 85.7591%, penurunan p95 sebesar 86.6121%, peningkatan *throughput* sebesar 580.2647%, dan *error rate* turun dari 0.0014% menjadi 0.0000% dibandingkan *baseline* tanpa cache. Dengan demikian, Redis *caching* dapat menjadi strategi optimasi yang relevan untuk *endpoint* RESTful API dengan pola akses baca berulang.

Saran untuk penelitian selanjutnya adalah memperluas pengujian pada variasi jumlah *virtual user*, durasi pengujian yang lebih panjang, pengukuran CPU dan memori, serta pengujian strategi invalidasi cache. Selain itu, evaluasi terhadap Redis Cluster atau *deployment* pada lingkungan *cloud* dapat dilakukan untuk menilai *trade-off* antara peningkatan performa, penggunaan memori, kompleksitas operasional, dan biaya infrastruktur.

5. DAFTAR RUJUKAN

[1] M. V. Privalov and M. V. Stupina, "Improving web-oriented information systems efficiency using Redis caching mechanisms," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 33, no. 3, pp. 1667–1675, Mar. 2024, doi: 10.11591/ijeecs.v33.i3.pp1667-1675.

[2] M. I. Zulfa, A. Fadli, and A. W. Wardhana, "Application caching strategy based on in-memory using Redis server to accelerate relational data access," *Jurnal Teknologi dan Sistem Komputer*, vol. 8, no. 2, pp. 157–163, Apr. 2020, doi: 10.14710/jtsiskom.8.2.2020.157-163.

[3] Y. Prilyansyah and Susanto, "Optimasi Performa API Aplikasi *E-commerce* Menggunakan Redis Caching," *STRING (Satuan Tulisan Riset dan Inovasi Teknologi)*, vol. 10, no. 2, pp. 227–234, 2025, doi: 10.30998/prxard20.

[4] M. Adhiyasa, S. Putranto, and A. Pinandito, "Analisis Kinerja Cache Redis pada RESTful API Berbasis Express.js," *Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer*, vol. 10, no. 2, 2026, [Online]. Available: https://j-ptiik.ub.ac.id/index.php/j-ptiik/article/view/15978

[5] F. Dipraja and A. Rahman, "Penerapan Redis Cluster Meningkatkan Efisiensi Caching Arsitektur *Microservices*," *Intellect : Indonesian Journal of Learning and Technological Innovation*, vol. 4, no. 1, pp. 171–179, Oct. 2025, doi: 10.57255/intellect.v4i1.1445.

[6] A. S. Azzahidi, B. Wijayanto, and A. Darmawan, "Performance Evaluation of Backend *Frameworks* for REST API: A Comparative Study of Spring Boot, Flask, Express.js, Laravel FrankenPHP, and Gin," *Jurnal Teknik Informatika (Jutif)*, vol. 6, no. 4, pp. 2405–2419, Aug. 2025, doi: 10.52436/1.jutif.2025.6.4.4811.

[7] F. PAMUNGKAS and H. Setiaji, "Implementasi Clean Architecture pada Pembuatan API Menggunakan Golang," *Jurnal INSTEK (Informatika Sains dan Teknologi)*, vol. 9, pp. 80–86, Feb. 2024, doi: 10.24252/instek.v9i1.46409.

[8] Y. Harjoseputro, Albertus Ari Kristanto, and Joseph Eric Samodra, "Golang and NSG Implementation in REST API Based Third-Party *Sandbox* System," *Jurnal RESTI (Rekayasa Sistem dan Teknologi Informasi)*, vol. 4, no. 4, pp. 745–750, Aug. 2020, doi: 10.29207/resti.v4i4.2218.

[9] I. W. R. Pinastawa, M. G. Pradana, and N. Maulana, "*Web server* Performance Evaluation: Comparing Nginx and Apache Using K6 Testing Methods for Load, Spike, Soak, and Performance," in *2024 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*, IEEE, Nov. 2024, pp. 1002–1007. doi: 10.1109/ICIMCIS63449.2024.10956402.

[10] Z. M. Jiang and A. E. Hassan, "A Survey on Load Testing of Large-Scale Software Systems," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1091–1118, Nov. 2015, doi: 10.1109/TSE.2015.2445340.

[11] GORM, "GORM Official Documentation." Accessed: Oct. 30, 2025. [Online]. Available: https://gorm.io/docs/