PERANCANGAN ARSITEKTUR CACHE-ASIDE MENGGUNAKAN REDIS PADA SISTEM INFORMASI AKADEMIK: STUDI PROTOTIPE DAN SIMULASI BEBAN

Yoga Ari Tofan[1], Rizky Parlika[2], Steffanuel Pranatalie Krispriyanto[3]

[1,2,3] Informatika, Fakultas Ilmu Komputer, Universitas Pembangunan Nasional Veteran Jawa Timur, Indonesia
[1] yoga.if@upnjatim.ac.id, [2] rizkyparlika.if@upnjatim.ac.id, [3] 23081010059@student.upnjatim.ac.id

Abstrak

Sistem informasi akademik, khususnya pada modul Kartu Rencana Studi (KRS), kerap mengalami lonjakan trafik ekstrem di awal semester. Arsitektur konvensional yang sepenuhnya bergantung pada basis data relasional berpotensi mengalami masalah latensi akibat antrean input/output (I/O) disk pada kondisi konkurensi tinggi. Penelitian ini merancang dan mengimplementasikan prototipe arsitektur *Cache-Aside* menggunakan Redis sebagai *cache layer* pada sistem backend berbasis Python (Flask). Untuk mengevaluasi perilaku arsitektur, dibangun model simulasi di mana latensi akses basis data (500 ms) dan latensi akses *cache* (5 ms) ditetapkan berdasarkan karakteristik tipikal yang dilaporkan dalam literatur. Pengujian beban dilakukan menggunakan Locust dengan simulasi 1.000 pengguna konkuren dan *spawn rate* 100 pengguna per detik. Hasil simulasi menunjukkan bahwa arsitektur *Cache-Aside* mampu menurunkan waktu respons rata-rata dari 507,52 ms (tanpa *cache*) menjadi 9,12 ms (dengan *cache*), meningkatkan *throughput* dari 385,71 menjadi 483,03 *request* per detik, serta mempertahankan *zero failure rate* pada kedua skenario. Distribusi persentil menunjukkan konsistensi performa: p95 turun dari 520 ms menjadi 13 ms. Hasil ini mengonfirmasi bahwa pola *Cache-Aside* secara arsitektural efektif dalam mengalihkan beban kerja dari penyimpanan berbasis disk ke memori pada skenario lonjakan trafik akademik. Validasi lebih lanjut dengan infrastruktur MySQL dan Redis sesungguhnya diperlukan untuk mengonfirmasi parameter simulasi.

Kata kunci: cache-aside, Redis, sistem informasi akademik, simulasi beban, Locust, KRS

1. Pendahuluan

Sistem informasi akademik memegang peranan krusial dalam operasional perguruan tinggi modern, terutama dalam pengelolaan data mahasiswa, administrasi nilai, dan proses registrasi mata kuliah (Latifurrahman et al., 2023; Noviyana & Nasution, 2024). Efisiensi dan akurasi dalam pengelolaan data menjadi tuntutan utama untuk mendukung pengambilan keputusan yang cepat dan tepat, serta meminimalkan kesalahan redundansi yang sering terjadi pada sistem konvensional (Ramli et al., 2024; Yindrizal, 2021). Di antara berbagai modul yang tersedia, modul Kartu Rencana Studi (KRS) merupakan salah satu komponen yang paling rentan terhadap lonjakan beban, karena ribuan mahasiswa mengaksesnya secara bersamaan dalam periode yang sangat singkat di awal semester (Azhar et al., 2024).

Pada skenario konkurensi tinggi seperti periode pengisian KRS, arsitektur yang sepenuhnya bergantung pada basis data relasional (RDBMS) berpotensi mengalami penurunan performa yang signifikan. Hal ini disebabkan oleh antrian *input*/*output* (I/O) pada penyimpanan berbasis disk yang menjadi *bottleneck* ketika volume permintaan melonjak secara drastis (Ju et al., 2024; Papon & Athanassoulis, 2021). Kondisi ini dapat menyebabkan peningkatan latensi respons hingga tingkat yang tidak dapat diterima pengguna, bahkan berpotensi menimbulkan kegagalan layanan (*service failure*) (Li et al., 2022).

Salah satu pendekatan yang banyak digunakan untuk mengatasi permasalahan tersebut adalah implementasi mekanisme *caching* berbasis *in-memory*. Redis, sebagai *in-memory key-value store*, telah terbukti mampu meningkatkan performa aplikasi web secara signifikan melalui berbagai studi. Pramudia dkk. (2025) mengonfirmasi bahwa sistem cache berbasis IMDB seperti Redis mampu meningkatkan performa akses data hingga empat kali lebih cepat dibandingkan akses langsung ke RDBMS berbasis disk pada aplikasi web (Pramudia et al., 2025). Privalov dan Stupina (2024) menunjukkan bahwa penggunaan gabungan MySQL dan Redis dengan strategi *caching* yang tepat dapat meningkatkan performa aplikasi web secara substansial (Privalov & Stupina, 2024). Weerasinghe dan Perera (2023) juga menunjukkan bahwa Redis mampu mengurangi latensi komunikasi antar layanan secara signifikan melalui pendekatan asinkron berbasis Redis Stream (Weerasinghe & Perera, 2023). Dalam konteks data akademik, Zulfa dkk. (2020) melaporkan bahwa Redis mampu mempercepat akses data relasional hingga 3,3 kali lipat pada operasi *join query* untuk menampilkan data mahasiswa (Zulfa et

al., 2020). Pada skala yang lebih besar, Kaptosv (2025) menguji Redis dengan pola *Cache-Aside*, *Write-Through*, dan Redis Cluster pada beban hingga 10.000 pengguna dan melaporkan penurunan waktu respons rata-rata dari 1.146 ms menjadi 323 ms pada mode *Cache-Aside* (Kaptosv, 2025).

Meskipun efektivitas Redis sebagai mekanisme *caching* telah banyak dibuktikan pada infrastruktur produksi, terdapat tiga celah yang belum ditangani secara terpadu oleh studi-studi tersebut. Pertama, studi yang ada umumnya dilakukan pada sistem yang sudah berjalan di lingkungan produksi, sehingga belum tersedia pendekatan evaluasi desain arsitektur berbasis simulasi terkontrol yang dapat digunakan oleh pengembang pada tahap awal perancangan sistem, sebelum infrastruktur tersedia. Kedua, studi-studi tersebut umumnya hanya melaporkan metrik rata-rata (*average response time*), sementara analisis distribusi persentil lengkap (p50 hingga p99,9) pada kondisi lonjakan trafik akademik belum tersedia sebagai acuan kuantitatif untuk mengevaluasi konsistensi performa *tail latency*. Ketiga, skenario lonjakan trafik yang spesifik pada periode pengisian KRS dengan pola kedatangan 1.000 pengguna konkuren dalam 10 detik belum dimodelkan secara eksplisit dalam studi yang ada, meskipun pola ini memiliki karakteristik beban yang berbeda dari trafik web umum karena bersifat serentak dan terikat jadwal akademik.

Berdasarkan ketiga celah tersebut, penelitian ini bertujuan untuk: (1) merancang arsitektur *Cache-Aside* menggunakan Redis pada prototipe modul KRS berbasis Python (Flask) sebagai model referensi yang dapat direplikasi oleh pengembang sistem informasi akademik; (2) memvalidasi efektivitas arsitektur tersebut melalui simulasi beban terkontrol dengan parameter latensi yang diturunkan dari literatur, sebagai pendekatan *design-time evaluation* sebelum implementasi produksi; dan (3) menyajikan analisis distribusi persentil lengkap (p50, p95, p99, p99,9) sebagai bukti konsistensi performa *tail latency* pada arsitektur *Cache-Aside* di bawah beban konkuren tinggi.

Kontribusi utama penelitian ini adalah: (1) desain arsitektur *Cache-Aside* yang dapat direplikasi untuk modul akademik berbasis web; (2) analisis komparatif kuantitatif antara skenario tanpa *cache* dan dengan *cache* melalui simulasi beban terkontrol menggunakan Locust; serta (3) rekomendasi untuk pengembangan lebih lanjut menuju implementasi dengan infrastruktur basis data dan *cache* sesungguhnya.

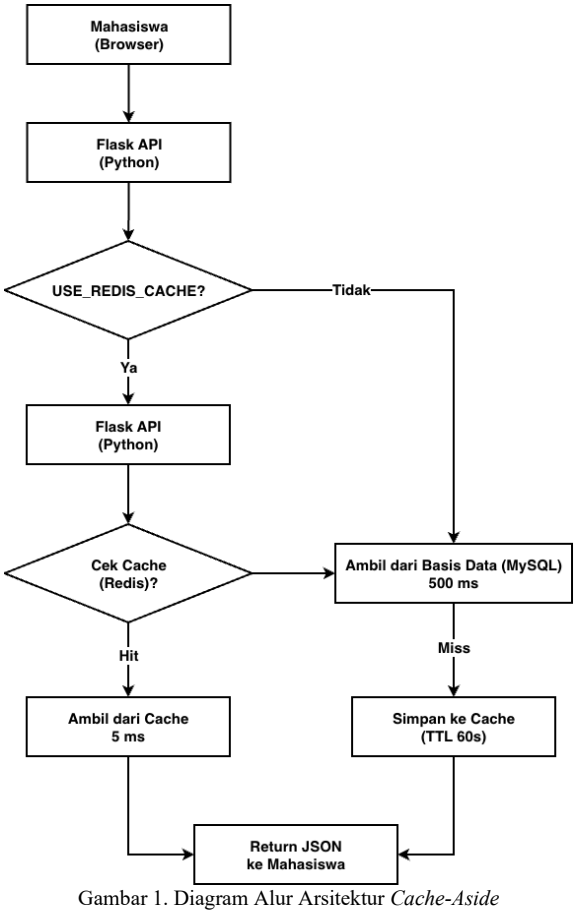
2. Metode

Penelitian ini menggunakan pendekatan pengembangan prototipe (*prototyping*) dengan simulasi beban terkontrol. Tujuan utama metode ini adalah membangun model arsitektur *Cache-Aside* pada modul Kartu Rencana Studi (KRS) dan mengevaluasi perilakunya di bawah skenario lonjakan trafik melalui parameter latensi yang diturunkan dari literatur. Tahapan penelitian terdiri atas tiga fase: (1) perancangan arsitektur dan struktur data, (2) implementasi prototipe *backend*, dan (3) pengujian beban menggunakan simulasi.

2.1 Perancangan Arsitektur

Arsitektur yang dirancang mengikuti pola *Cache-Aside* (juga dikenal sebagai *Lazy Loading*), di mana aplikasi bertanggung jawab untuk membaca dan menulis *cache* secara eksplisit (Privalov & Stupina, 2024; Zulfa et al., 2020). Alur kerja arsitektur terdiri atas dua skenario seperti pada Gambar 1, yaitu:

1. Skenario tanpa *cache* (*cache miss*): Setiap permintaan dari pengguna diteruskan langsung ke basis data utama untuk mengambil data jadwal kuliah. Pada kondisi ini, latensi didominasi oleh waktu akses I/O disk.
2. Skenario dengan *cache* (*cache hit*): Pada permintaan pertama, data diambil dari basis data dan disimpan ke *cache* dengan kunci unik (`krs:jadwal:semester_ganjil`) dan *Time-To-Live* (TTL) selama 60 detik. Permintaan berikutnya yang menggunakan kunci yang sama akan dilayani langsung dari *cache* tanpa mengakses basis data.



Gambar 1. Diagram Alur Arsitektur *Cache-Aside*

2.2 Implementasi Prototipe

Prototipe backend dibangun menggunakan bahasa pemrograman Python dengan *framework* Flask. Sistem menyediakan satu *endpoint* REST API (`GET /api/jadwal`) yang mengembalikan data jadwal kuliah dalam format JSON. Struktur data yang digunakan merepresentasikan tabel jadwal mata kuliah dengan atribut kode mata kuliah, nama mata kuliah, jumlah SKS, dan hari perkuliahan.

Karena penelitian ini bersifat simulasi untuk memvalidasi arsitektur, latensi akses data dimodelkan menggunakan fungsi *delay* (`time.sleep`) dengan parameter sebagai berikut:

- Latensi basis data (*cache miss*): 500 ms. Nilai ini merepresentasikan latensi tipikal akses basis data relasional pada kondisi antrian I/O disk di bawah beban tinggi. Kaptosv (2025) melaporkan latensi rata-rata 1.146 ms pada kondisi tanpa *cache* dengan beban 10.000 pengguna (Kaptosv, 2025), sementara Privalov & Stupina (2024) juga mengonfirmasi bahwa latensi basis data meningkat signifikan pada kondisi konkurensi tinggi (Privalov & Stupina, 2024). Nilai 500 ms dipilih sebagai estimasi konservatif untuk beban moderat (1.000 pengguna).
- Latensi *cache* (*cache hit*): 5 ms. Nilai ini merepresentasikan latensi tipikal operasi GET pada Redis server. Zulfa, et al (2020) melaporkan waktu akses Redis pada orde mikrodetik hingga milidetik rendah (Zulfa et al., 2020), dan benchmark resmi Redis melaporkan latensi sub-milidetik pada operasi sederhana.

Mekanisme *caching* diimplementasikan melalui *in-process* dictionary yang menyimulasikan perilaku penyimpanan *key-value* Redis, dengan operasi *get* (baca *cache*) dan *setex* (tulis *cache* dengan TTL). Pengaturan skenario dikendalikan melalui variabel konfigurasi `USE_REDIS_CACHE` yang memungkinkan pergantian antara mode tanpa cache dan mode dengan *cache* tanpa mengubah kode utama.

Penggunaan *in-process dictionary* sebagai representasi *cache* dan dataset tiga record merupakan pilihan metodologis yang disengaja, bukan keterbatasan teknis. Pendekatan ini mengikuti paradigma simulasi terkontrol (*controlled simulation*) yang lazim digunakan dalam evaluasi desain arsitektur pada tahap awal pengembangan, di mana tujuan utamanya adalah memvalidasi logika alur arsitektur (*architectural flow*) dan mengukur dampak latensi secara terisolasi, bukan mereplikasi kondisi produksi secara penuh. Dengan mengisolasi variabel latensi melalui `time.sleep`, penelitian ini dapat mengukur dampak murni dari perbedaan latensi *cache hit* versus *cache miss* terhadap distribusi respons di bawah beban konkuren, tanpa kontaminasi dari variabel eksternal seperti kondisi jaringan, beban CPU, atau variasi ukuran data. Strategi serupa digunakan dalam berbagai studi evaluasi arsitektur awal untuk memperoleh pemahaman baseline sebelum implementasi pada infrastruktur penuh (Ju et al., 2024). Konsekuensi dari pendekatan ini — yaitu bahwa hasil yang diperoleh merupakan estimasi batas atas (*upper bound*) dan belum memperhitungkan dinamika sistem nyata — dibahas secara eksplisit pada sub-bab 3.6.

Tabel 1. Spesifikasi Lingkungan Pengembangan dan Pengujian

Komponen	Spesifikasi
Bahasa pemrograman	Python 3.12
Web framework	Flask 3.1.2
Simulasi *cache*	*In-process key-value store* (Python *dictionary*)
Parameter latensi basis data	500 ms (`time.sleep`)
Parameter latensi *cache*	5 ms (`time.sleep`)
Format respons API	JSON
Endpoint yang diuji	GET `/api/jadwal`
Jumlah record data	3 mata kuliah (validasi logika arsitektur)
Pendekatan simulasi cache	In-process dictionary (isolasi variabel latensi)
CPU	Apple Silicon M4
RAM	16 GB
OS	MacOS Tahoe 26.2

2.3 Pengujian Beban (Load Testing)

Pengujian beban dilakukan menggunakan Locust, sebuah *open-source load testing tool* berbasis Python yang memungkinkan simulasi perilaku pengguna secara *scriptable* (Borge et al., 2025). Setiap *virtual user* dikonfigurasi untuk mengakses *endpoint* `/api/jadwal` dengan jeda antar-permintaan (*wait time*) antara 1 hingga 3 detik, yang merepresentasikan perilaku mahasiswa saat menjelajahi halaman KRS. Konfigurasi pengujian beban ditetapkan seperti pada Tabel 2.

Tabel 2. Konfigurasi Pengujian Beban (Parameter Locust)

Parameter	Nilai
Jumlah pengguna konkuren	1.000
Spawn rate	100 pengguna/detik
Durasi *ramp-up*	10 detik (1.000 / 100)
Wait time antar-*request*	1–3 detik (*random*)
Endpoint target	`GET /api/jadwal`

Skenario ini merepresentasikan kondisi traffic spike di mana 1.000 mahasiswa masuk ke sistem dalam waktu 10 detik, sebuah skenario yang realistis pada saat pembukaan periode pengisian KRS. Pengujian dilakukan dalam dua skenario terpisah:

- Skenario A (tanpa *cache*): `USE_REDIS_CACHE = False`. Seluruh permintaan melewati jalur simulasi akses basis data (latensi 500 ms).
- Skenario B (dengan *cache*): `USE_REDIS_CACHE = True`. Permintaan pertama melewati jalur basis data (*cache miss*), permintaan selanjutnya dilayani dari *cache* (*cache hit*, latensi 5 ms).

2.4 Metrik Evaluasi

Data kinerja yang dikumpulkan dari Locust meliputi metrik-metrik seperti pada Tabel 3. Analisis komparatif dilakukan dengan membandingkan seluruh metrik antara Skenario A dan Skenario B untuk mengevaluasi dampak arsitektur *Cache-Aside* terhadap latensi, throughput, dan stabilitas sistem.

Tabel 3. Deskripsi Metrik Evaluasi

Metrik	Deskripsi
Average response time	Rata-rata waktu respons seluruh permintaan (ms)
Median response time (p50)	Nilai tengah distribusi waktu respons (ms)
Persentil p95 dan p99	Waktu respons pada persentil ke-95 dan ke-99 (ms)
Min/Max response time	Waktu respons tercepat dan terlambat (ms)
Requests per second (RPS)	Jumlah permintaan yang berhasil dilayani per detik
Failure count	Jumlah permintaan yang gagal
Total request count	Total permintaan yang terkirim selama pengujian

3. Hasil dan Pembahasan

3.1 Hasil Pengujian Respons Tunggal

Sebelum pengujian beban, dilakukan verifikasi fungsional terhadap *endpoint* `/api/jadwal` untuk memastikan bahwa arsitektur *Cache-Aside* bekerja sesuai rancangan. Gambar 2 menunjukkan respons sistem pada kondisi *cache miss*, sedangkan Gambar 3 menunjukkan respons pada kondisi *cache hit*.

Pada kondisi *cache miss* (Skenario A), sistem mengembalikan data dengan sumber "MYSQL (*Cache Miss*)" dan waktu respons 505,03 ms. Pada kondisi *cache hit* (Skenario B), sistem mengembalikan data yang identik dengan sumber "REDIS (*Cache Hit*)" dan waktu respons 5,46 ms. Kedua respons menghasilkan data yang sama, yang mengonfirmasi bahwa mekanisme *caching* berfungsi dengan benar tanpa mengubah konten data yang disajikan.


```
{
  "data": [
    {
      "hari": "Senin",
      "kode": "MK001",
      "matkul": "Kecerdasan Buatan",
      "sks": 3
    },
    {
      "hari": "Selasa",
      "kode": "MK002",
      "matkul": "Pemrograman Web",
      "sks": 4
    },
    {
      "hari": "Rabu",
      "kode": "MK003",
      "matkul": "Basis Data",
      "sks": 3
    }
  ],
  "status": "berhasil",
  "sumber_data": "MYSQL (Cache Miss)",
  "waktu_respon_ms": 505.03
}
```


Gambar 2. Respons Sistem Ketika *Cache Miss*


```
{
  "data": [
    {
      "hari": "Senin",
      "kode": "MK001",
      "matkul": "Kecerdasan Buatan",
      "sks": 3
    },
    {
      "hari": "Selasa",
      "kode": "MK002",
      "matkul": "Pemrograman Web",
      "sks": 4
    },
    {
      "hari": "Rabu",
      "kode": "MK003",
      "matkul": "Basis Data",
      "sks": 3
    }
  ],
  "status": "berhasil",
  "sumber_data": "REDIS (Cache Hit)",
  "waktu_respon_ms": 5.46
}
```


Gambar 3. Respons Sistem Ketika *Cache Hit*

3.2 Hasil Pengujian Beban (Load Testing)

Pengujian beban dilakukan pada kedua skenario dengan 1.000 pengguna konkuren. Tabel 4 menyajikan ringkasan hasil pengujian pada Skenario A (tanpa *cache*) dan Skenario B (dengan *cache*) yang diperoleh dari laporan statistik Locust.

Tabel 4. Perbandingan Metrik Kinerja Skenario A vs B

Metrik	Skenario A	Skenario B	Selisih
Total *request*	46.038	57.632	+25,2%
Failure count	0	0	–
Rata-rata waktu respons	507,52 ms	9,12 ms	−98,2 %
Median (p50)	510 ms	8 ms	−98,4 %
Min waktu respons	500,79 ms	5,75 ms	−98,9 %
Maks waktu respons	600,37 ms	64,84 ms	−89,2 %
p95	520 ms	13 ms	−97,5 %
p99	560 ms	42 ms	−92,5 %
p99,9	570 ms	55 ms	−90,4 %
Throughput (RPS)	385,71	483,03	+25,2%
Rata-rata ukuran respons	434,91 B	431,86 B	~sama

3.3 Analisis Latensi Rata-rata dan Median

Hasil simulasi menunjukkan penurunan waktu respons yang sangat signifikan ketika arsitektur *Cache-Aside* diaktifkan. Rata-rata waktu respons turun dari 507,52 ms menjadi 9,12 ms, yang merepresentasikan penurunan sebesar 98,2%. Median waktu respons menunjukkan pola yang serupa, turun dari 510 ms menjadi 8 ms.

Penurunan latensi ini konsisten dengan parameter simulasi yang ditetapkan (500 ms untuk akses basis data, 5 ms untuk akses *cache*), dengan tambahan overhead dari pemrosesan Flask dan serialisasi JSON. Pada Skenario A, rata-rata 507,52 ms sedikit di atas parameter *delay* 500 ms, yang mengindikasikan overhead pemrosesan sekitar 7,5 ms. Pada Skenario B, rata-rata 9,12 ms juga sedikit di atas parameter 5 ms, dengan overhead serupa sekitar 4,1 ms. Perbedaan overhead antara kedua skenario kemungkinan disebabkan oleh operasi `json.loads` tambahan pada jalur *cache hit*.

Hasil ini sejalan dengan temuan studi sebelumnya yang menggunakan infrastruktur nyata. Privalov & Stupina (2024) melaporkan peningkatan performa yang substansial pada aplikasi web dengan strategi *caching* Redis dan MySQL (Privalov & Stupina, 2024). Zulfa, et al (2020) melaporkan percepatan hingga 3,3 kali lipat pada akses data akademik relasional menggunakan Redis (Zulfa et al., 2020). Pada skala yang lebih besar, Kaptosv (2025) menunjukkan penurunan waktu respons dari 1.146 ms menjadi 323 ms (penurunan ~72%) menggunakan pola *Cache-Aside* dengan Redis dan PostgreSQL pada beban 1.000 pengguna (Kaptosv, 2025). ada stack teknologi berbeda, Sakti dkk. (2025) melaporkan penurunan latensi drastis dari 30,32 detik menjadi 1,66 detik pada backend Node.js dengan PostgreSQL menggunakan Redis caching, mengonfirmasi bahwa pola ini efektif lintas platform (Sakti et al., 2025).

3.4 Analisis Distribusi Persentil (Tail Latency)

Selain rata-rata, distribusi persentil memberikan gambaran yang lebih lengkap tentang konsistensi performa. Tabel 5 menyajikan perbandingan distribusi persentil antara kedua skenario.

Tabel 5. Distribusi Persentil Waktu Respons (ms)

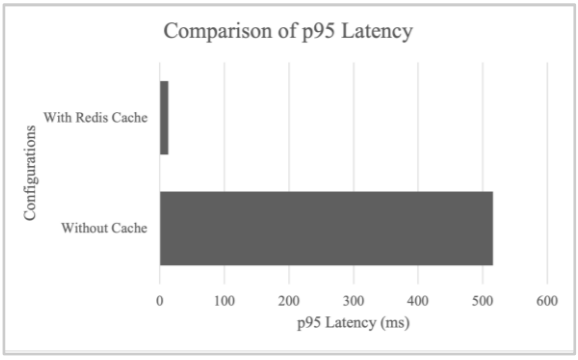
Persentil	Skenario A	Skenario B	Rasio
p50 (median)	510	8	63,8x
p66	510	8	63,8x
p75	510	9	56,7x
p80	510	9	56,7x
p90	510	10	51,0x
p95	520	13	40,0x
p98	550	35	15,7x
p99	560	42	13,3x
p99,9	570	55	10,4x
p99,99	590	64	9,2x
p100 (maks)	600	65	9,2x

Dari Tabel 5 terlihat dua pola yang menarik:

1. Skenario A menunjukkan distribusi yang sangat sempit. Hampir seluruh permintaan (p50 hingga p90) memiliki waktu respons 510 ms, dengan *spread* total hanya ~100 ms (500–600 ms). Hal ini konsisten dengan karakteristik fungsi `time.sleep(0.5)` yang menghasilkan *delay* deterministik, dengan variasi kecil yang berasal dari *scheduling* sistem operasi dan overhead jaringan.
2. Skenario B menunjukkan distribusi yang lebih lebar secara proporsional. Meskipun mayoritas permintaan (p50–p80) dilayani dalam 8–9 ms, terdapat *tail* yang meningkat di persentil atas: p98 = 35 ms, p99 = 42 ms, p99,9 = 55 ms, dan maks = 65 ms. Lonjakan pada persentil atas ini kemungkinan disebabkan oleh faktor-faktor seperti *garbage collection* Python, *context switching* pada tingkat sistem operasi, atau antrian pada *event loop* Flask saat menangani 1.000 koneksi konkuren.

Meskipun *tail latency* pada Skenario B mencapai 65 ms pada kasus terburuk, angka ini masih jauh di bawah rata-rata Skenario A (507,52 ms), yang menunjukkan bahwa arsitektur *Cache-Aside* tetap memberikan keunggulan performa bahkan pada kondisi *worst case*.

Gambar 4 memvisualisasikan perbandingan latensi pada persentil ke-95 (p95) antara Skenario A dan Skenario B. Terlihat perbedaan yang sangat signifikan: p95 pada Skenario A tercatat 520 ms, sementara pada Skenario B hanya 13 ms, penurunan sebesar 97,5%.



Gambar 4. Grafik Batang Perbandingan p95 Latency

3.5 Analisis Throughput dan Stabilitas

Skenario B berhasil melayani 57.632 *request* dibandingkan 46.038 pada Skenario A dalam durasi pengujian yang sama, merepresentasikan peningkatan *throughput* sebesar 25,2%. Dalam satuan *request per second* (RPS), Skenario A mencapai 385,71 RPS sementara Skenario B mencapai 483,03 RPS.

Peningkatan *throughput* ini terjadi karena setiap request pada Skenario B melepaskan *thread* lebih cepat (rata-rata 9,12 ms vs 507,52 ms), sehingga server Flask dapat menerima dan memproses *request* berikutnya lebih segera. Dalam konteks pengisian KRS, peningkatan kapasitas ini berarti sistem dapat melayani lebih banyak mahasiswa dalam periode waktu yang sama. Hasil ini konsisten dengan temuan Dipraja dan Rahman (2025) yang melaporkan peningkatan throughput sebesar 30,6% dan penurunan beban basis data hingga 58%

menggunakan Redis Cluster pada arsitektur microservices (Dipraja & Rahman, 2025).

Aspek stabilitas juga patut dicatat: kedua skenario mencatat *zero failure* (0 kegagalan dari total seluruh *request*). Ini menunjukkan bahwa prototipe Flask mampu menangani 1.000 koneksi konkuren tanpa *error* pada kedua konfigurasi, meskipun dengan perbedaan latensi yang sangat besar.

3.6 Keterbatasan dan Validitas Hasil Simulasi

Hasil yang disajikan pada sub-bab 3.2 hingga 3.5 perlu diinterpretasikan secara hati-hati dengan mempertimbangkan karakteristik pendekatan simulasi yang digunakan. Terdapat empat keterbatasan mendasar yang memengaruhi derajat generalisasi hasil penelitian ini.

Pertama, parameter latensi bersifat deterministik dan merepresentasikan batas atas teoritis (*upper bound*). Penurunan waktu respons sebesar 98,2% yang diperoleh dalam simulasi ini tidak mencerminkan performa yang dapat diharapkan secara langsung pada sistem produksi. Pada sistem nyata, latensi basis data relasional bervariasi tergantung pada kompleksitas query, ukuran data yang diambil, jumlah koneksi aktif, kondisi jaringan, dan beban CPU server. Begitu pula latensi Redis pada kondisi produksi dipengaruhi oleh latensi jaringan antara aplikasi dan Redis server, tekanan memori, dan ukuran value yang disimpan. Sebagai perbandingan, Kaptosv (2025) yang menguji Redis *Cache-Aside* pada infrastruktur nyata dengan beban 1.000 pengguna melaporkan penurunan latensi rata-rata dari 1.146 ms menjadi 323 ms, atau sekitar 71,8% — jauh lebih rendah dari 98,2% yang diperoleh dalam simulasi ini (Kaptosv, 2025). Demikian pula, Privalov & Stupina (2024) melaporkan peningkatan performa yang substansial namun tidak mendekati nilai teoritis, karena variabilitas latensi jaringan dan *query processing* pada sistem nyata menghasilkan distribusi yang lebih lebar (Privalov & Stupina, 2024). Dengan demikian, angka 98,2% dalam penelitian ini lebih tepat dipahami sebagai estimasi potensi maksimal dari arsitektur Cache-Aside, bukan sebagai prediksi performa pada lingkungan produksi.

Kedua, dataset berskala sangat kecil. Prototipe hanya menggunakan 3 record data jadwal dalam struktur data tunggal. Pada sistem produksi, tabel jadwal KRS umumnya melibatkan ratusan hingga ribuan record dengan operasi join ke tabel lain seperti tabel dosen, ruang kuliah, dan slot waktu. Ukuran payload yang lebih besar akan memengaruhi baik latensi pengambilan data dari basis data maupun latensi serialisasi dan deserialisasi value di Redis.

Ketiga, mekanisme *caching* tidak menggunakan Redis server sesungguhnya. Implementasi menggunakan in-process dictionary Python yang hanya menyimulasikan perilaku *key-value store*. Akibatnya, aspek-aspek penting seperti latensi jaringan antara aplikasi dan Redis, manajemen memori (*memory eviction*), persistensi data, serta thread-safety pada konkurensi tinggi tidak tercakup dalam simulasi ini.

Keempat, skenario *cache miss* hanya terjadi satu kali selama pengujian Skenario B, yaitu pada request pertama sebelum data tersimpan di cache. Seluruh request berikutnya adalah *cache hit* karena TTL 60 detik tidak kedaluwarsa dalam durasi pengujian. Pada sistem produksi, *cache miss* dapat terjadi berulang akibat TTL kedaluwarsa, *key eviction* karena tekanan memori, atau variasi parameter query yang menghasilkan *cache key* berbeda. Skenario *cache miss* berulang ini akan menurunkan performa Skenario B dan mempersempit selisih latensi antara kedua skenario.

Meskipun keempat keterbatasan di atas membatasi generalisasi langsung ke lingkungan produksi, hasil simulasi ini tetap memiliki nilai ilmiah sebagai evaluasi desain arsitektur pada tahap awal pengembangan (*design-time evaluation*). Pendekatan simulasi terkontrol memungkinkan isolasi variabel dan pengukuran yang tidak mungkin dilakukan dengan presisi setara pada sistem produksi yang memiliki banyak variabel eksternal. Temuan bahwa arsitektur *Cache-Aside* secara konsisten memberikan keunggulan performa di seluruh metrik — rata-rata, median, distribusi persentil, dan throughput — selaras dengan konsensus literatur yang menggunakan infrastruktur nyata (Kaptosv, 2025; Privalov & Stupina, 2024; Zulfa et al., 2020), sehingga memperkuat validitas konseptual arsitektur yang dirancang. Validasi empiris lebih lanjut menggunakan MySQL dan Redis server sesungguhnya diperlukan untuk mengonfirmasi besaran performa yang sesungguhnya dapat dicapai.

3.7 Diskusi: Dinamika Caching pada Sistem Produksi

Sub-bab ini mendiskusikan tiga dinamika penting dalam sistem caching nyata yang tidak tercakup dalam skenario simulasi, namun krusial untuk dipahami sebelum arsitektur ini diimplementasikan pada lingkungan produksi.

3.7.1 Cache Miss Berulang dan Dampaknya terhadap Latensi

Pada Skenario B dalam simulasi ini, seluruh request setelah yang pertama adalah *cache hit* karena TTL 60 detik tidak kedaluwarsa selama durasi pengujian, sehingga rasio cache hit mendekati 100%. Kondisi ini merupakan skenario ideal yang tidak representatif untuk sistem produksi.

Pada sistem produksi, *cache miss* dapat terjadi berulang karena tiga sebab utama: (1) TTL kedaluwarsa secara periodik, (2) Redis mengeluarkan *key* dari memori (*eviction*) akibat tekanan memori, dan (3) variasi parameter query yang menghasilkan *cache key* berbeda. Pada sistem KRS, misalnya,

mahasiswa yang mengakses jadwal berdasarkan program studi atau semester yang berbeda akan menghasilkan *cache key* yang berbeda, sehingga cache miss pertama terjadi untuk setiap kombinasi unik.

Dampak kuantitatif dari *cache miss* berulang dapat diestimasi menggunakan formula latensi rata-rata tertimbang pada Persamaan (1).

$$L = r_{hit} \cdot L_{hit} + (1 - r_{hit}) \cdot L_{miss} \qquad (1)$$

di mana L adalah latensi rata-rata yang diharapkan, r_{hit} adalah rasio *cache hit*, L_{hit} adalah latensi *cache hit* (5 ms), dan L_{miss} adalah latensi *cache miss* (500 ms). Tabel 6 menyajikan estimasi latensi rata-rata pada berbagai skenario rasio *cache hit*.

Tabel 6. Estimasi latensi rata-rata berdasarkan variasi rasio *cache hit*

Rasio *Cache Hit*	Estimasi Latensi Rata-rata
100% (simulasi ini)	5 ms
95%	29,75 ms
90%	54,5 ms
80%	104 ms
70%	153,5 ms
50%	252,5 ms

Dari Tabel 6 terlihat bahwa meskipun rasio *cache hit* turun menjadi 90%, latensi rata-rata yang diharapkan (54,5 ms) masih jauh di bawah latensi tanpa *cache* (507,52 ms). Arsitektur *Cache-Aside* tetap memberikan manfaat performa yang signifikan selama rasio *cache hit* berada di atas 50%.

3.7.2 *Cache Invalidation* dan Strategi TTL

Cache invalidation adalah proses penghapusan atau pembaruan data di cache ketika data sumber (basis data) berubah, untuk mencegah pengguna menerima data yang sudah kedaluwarsa (*stale data*). Pada pola *Cache-Aside*, tanggung jawab invalidation sepenuhnya berada pada aplikasi.

Pada konteks KRS, data jadwal kuliah relatif statis dalam satu semester, sehingga risiko *stale* data tergolong rendah. Namun, perubahan jadwal yang terjadi di tengah semester — misalnya perubahan ruang atau waktu kuliah — harus segera dicerminkan di *cache*. Dua strategi yang dapat diterapkan adalah:

1. TTL pendek (misalnya 60—300 detik): Data *cache* kedaluwarsa secara otomatis dalam waktu singkat. Strategi ini sederhana namun meningkatkan frekuensi *cache miss* dan beban basis data.
2. Invalidasi aktif (*active invalidation*): Setiap kali data jadwal diperbarui di basis data, aplikasi secara eksplisit menghapus atau memperbarui key terkait di Redis. Strategi ini mempertahankan konsistensi data secara *real-time* namun memerlukan koordinasi antara modul penulisan dan pembacaan data.

Falkevych dan Lisniak (2025) mengusulkan pendekatan deklaratif untuk cache invalidation yang memisahkan logika bisnis dari mekanisme pembaruan cache, sehingga meningkatkan fleksibilitas manajemen cache pada sistem yang dinamis (Falkevych & Lisniak, 2025). Pemilihan strategi TTL yang optimal untuk modul KRS merupakan area yang perlu dieksplorasi lebih lanjut pada penelitian berikutnya, dengan mempertimbangkan frekuensi perubahan data dan toleransi terhadap *stale data* di lingkungan akademik.

3.7.3 *Cache Stampede* pada Kondisi *Traffic Spike*

Pada skenario lonjakan trafik seperti pembukaan periode KRS, terdapat risiko terjadinya *cache stampede* (juga dikenal sebagai *thundering herd problem*): kondisi di mana TTL dari sebuah *key* yang populer kedaluwarsa tepat saat trafik sedang tinggi, sehingga ratusan request secara bersamaan mengalami *cache miss* dan semuanya langsung menekan basis data secara simultan. Kondisi ini dapat menyebabkan lonjakan beban basis data yang justru lebih berat dibandingkan kondisi tanpa *cache* sama sekali.

Beberapa teknik mitigasi yang dapat diterapkan antara lain: (1) probabilistic early expiration, di mana *cache* diperbarui sebelum TTL benar-benar kedaluwarsa berdasarkan probabilitas tertentu; (2) mutex lock pada saat *cache miss*, sehingga hanya satu request yang mengakses basis data sementara *request* lainnya menunggu; dan (3) staggered TTL, yaitu pemberian nilai TTL yang sedikit bervariasi secara acak untuk mencegah kedaluwarsa massal secara bersamaan. Eksplorasi teknik-teknik ini dalam konteks sistem KRS merupakan arah penelitian lanjutan yang relevan.

4. Kesimpulan

Penelitian ini memberikan tiga kontribusi terhadap literatur optimasi performa sistem informasi akademik. Pertama, penelitian ini memperkenalkan pendekatan simulasi terkontrol sebagai metode design-time evaluation arsitektur Cache-Aside, yang memungkinkan pengembang memvalidasi desain dan mengukur potensi dampak performa sebelum infrastruktur produksi tersedia. Kedua, penelitian ini menyajikan analisis distribusi persentil lengkap (p50 hingga p99,9) yang tidak tersedia pada studi-studi sebelumnya, sehingga memberikan gambaran yang lebih komprehensif tentang konsistensi performa tail latency di bawah beban konkuren tinggi. Ketiga, penelitian ini secara eksplisit memodelkan skenario lonjakan trafik pengisian KRS dengan pola kedatangan 1.000 pengguna konkuren dalam 10 detik, yang merupakan karakteristik beban spesifik sistem akademik yang belum dieksplorasi secara terfokus pada studi sebelumnya.

Hasil simulasi menunjukkan bahwa arsitektur *Cache-Aside* secara signifikan menurunkan waktu respons rata-rata dari 507,52 ms (tanpa *cache*)

menjadi 9,12 ms (dengan *cache*), meningkatkan *throughput* dari 385,71 menjadi 483,03 *request* per detik, serta mempertahankan *zero failure rate* pada kedua skenario. Analisis distribusi persentil juga mengonfirmasi konsistensi performa: latensi p95 turun dari 520 ms menjadi 13 ms, yang menunjukkan bahwa 95% permintaan pada arsitektur *cache* dilayani dalam waktu di bawah 15 ms.

Temuan ini mengonfirmasi bahwa pola *Cache-Aside* secara arsitektural efektif dalam mengalihkan beban kerja dari penyimpanan berbasis disk ke memori, khususnya pada skenario lonjakan trafik yang tipikal terjadi saat periode pengisian KRS. Hasil simulasi juga konsisten dengan studi-studi sebelumnya yang menggunakan infrastruktur MySQL dan Redis sesungguhnya (Kaptosv, 2025; Privalov & Stupina, 2024; Zulfa et al., 2020), yang memperkuat validitas konseptual arsitektur yang dirancang.

Namun, karena penelitian ini menggunakan pendekatan simulasi dengan parameter latensi deterministik dan dataset berskala kecil, hasil yang diperoleh belum dapat digeneralisasi secara langsung ke sistem produksi. Penelitian selanjutnya disarankan untuk: (1) mengimplementasikan arsitektur ini dengan MySQL dan Redis server sesungguhnya, (2) menggunakan dataset akademik yang lebih besar dan realistis dengan operasi *join* antar-tabel, serta (3) menguji skenario *cache invalidation* dan variasi TTL untuk mengevaluasi efektivitas *caching* pada kondisi data yang dinamis.

Daftar Pustaka:

Azhar, M. H., Pradnyana, I. W. W., & Irzavika, N. (2024). Optimization of Microservice-Based Academic Services with the Use of Message Brokers (Case Study: Business Process of Submitting Krs). *2024 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*, 827–832. https://doi.org/10.1109/ICIMCIS63449.2024.10956829

Borge, F. V., López-de-Ipiña, D., Manrique, M. E., Olivares-Rodríguez, C., Wolosiuk, D., & Vuckovic, M. (2025). Stress-Testing Citizen Science at Scale: Performance Insights from the GREENCROWD Platform. *2025 10th International Conference on Smart and Sustainable Technologies (SpliTech)*, 1–8. https://doi.org/10.23919/SpliTech65624.2025.11091673

Dipraja, F., & Rahman, A. (2025). Penerapan Redis Cluster Meningkatkan Efisiensi Caching Arsitektur Microservices. *Intellect : Indonesian Journal of Learning and Technological Innovation*, *4*(1), 171–179. https://doi.org/10.57255/intellect.v4i1.1445

Falkevych, V., & Lisniak, A. (2025). Cache invalidation based on a declarative approach for separating business logic of microservices from cache update rules. *Eastern-European Journal of Enterprise Technologies*, *2*(2 (134)), 68–74. https://doi.org/10.15587/1729-4061.2025.325932

Ju, L., Yadav, A., Khan, A., Sah, A. P., & Yadav, D. (2024). Using Asynchronous Frameworks and Database Connection Pools to Enhance Web Application Performance in High-Concurrency Environments. *2024 8th International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, 742–747. https://doi.org/10.1109/I-SMAC61858.2024.10714639

Kaptosv, L. (2025). Using Redis for Caching Optimization in High-Traffic Web Applications. *International Journal of Advanced Multidisciplinary Research and Studies*, *5*(4), 1714–1722. https://doi.org/10.62225/2583049X.2025.5.4.4839

Latifurrahman, A., Imilda, & Salam, A. (2023). Sistem Informasi Akademik menggunakan PHP dan MySQL pada Sekolah Tinggi Manajemen Informatika Komputer (STMIK) Indonesia Banda Aceh. *Jurnal Sistem Komputer (SISKOM)*, *3*(2), 74–83. https://doi.org/10.35870/siskom.v3i2.796

Li, N., Jiang, H., Che, H., Wang, Z., & Nguyen, M. Q. (2022). Improving scalability of database systems by reshaping user parallel I/O. *Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22*, 592–609. https://doi.org/10.1145/3492321.3519570

Noviyana, N., & Nasution, M. I. P. (2024). Penerapan Teknologi Informasi Untuk Efektivitas dan Efisiensi Pengolahan Data Mahasiswa. *Jurnal Ilmiah Research and Development Student*, *2*(1), 152–160. https://doi.org/10.59024/jis.v2i1.578

Papon, T. I., & Athanassoulis, M. (2021). *The Need for a New I/O Model*. Proceedings of the Annual Conference on Innovative Data Systems Research (CIDR).

Pramudia, F. A., Zulfa, M. I., & Aliim, M. S. (2025). The Effect In-memory Based Cache System On Web Applications In Improving Data Access Performance. *Jurnal Ilmiah Dinamika Rekayasa*, *21*(2), 166–174. https://doi.org/10.20884/1.jidr.2025.21.2.12

Privalov, M. V., & Stupina, M. V. (2024). Improving web-oriented information systems efficiency using Redis caching mechanisms. *Indonesian Journal of Electrical Engineering and Computer Science*, *33*(3), 1667–1675. https://doi.org/10.11591/ijeecs.v33.i3.pp1667-1675

Ramli, H., Akbar, P. I. M., Aisah, A. A. N., Alfiani, & Labenu, T. (2024). Meningkatkan Efisiensi dan Kualitas Layanan Akademik: Pendekatan Sistem Informasi di Universitas Taal: INDONESIA. *Journal of Renewable Energy and Smart Device*, 44–63. https://doi.org/10.61220/joresd.v1i2.241

Sakti, M. D. I., Dirgantara, D., Ramadhan, A. K., & Ibrahim, M. A. (2025). Optimizing Asynchronous Performance in Node.js with Express and PostgreSQL. *Procedia Computer Science, The 10th International Conference on Computer Science and Computational Intelligence 2025*, *269*, 172–181. https://doi.org/10.1016/j.procs.2025.08.270

Weerasinghe, S., & Perera, I. (2023). Optimized Strategy for Inter-Service Communication in Microservices. *International Journal of Advanced Computer Science and Applications (IJACSA)*, *14*(2). https://doi.org/10.14569/IJACSA.2023.0140233

Yindrizal, Y. (2021). Dampak Penggunaan Sistem Informasi Akademik Untuk Meningkatkan Kualitas Pelayanan Akademik Mahasiswa Universitas Andalas Padang. *Jurnal Manajemen Publik Dan Kebijakan Publik*, *3*(1), 1–13. https://doi.org/10.36085/jmpkp.v3i1.1433

Zulfa, M. I., Fadli, A., & Wardhana, A. W. (2020). Strategi caching aplikasi berbasis in-memory menggunakan Redis server untuk mempercepat akses data relasional. *Jurnal Teknologi dan Sistem Komputer*, *8*(2), 157–163. https://doi.org/10.14710/jtsiskom.8.2.2020.157-163

Halaman ini sengaja dikosongkan