RANCANG BANGUN RESTFUL API PINTAR MENGGUNAKAN GOLANG DAN GIN FRAMEWORK DENGAN FITUR AUTO CACHE ADAPTIF

SKRIPSI

DISUSUN OLEH

DITA AULIA AL FARID

NPM. 2109020178

PROGRAM STUDI TEKNOLOGI INFORMASI

FAKULTAS ILMU KOMPUTER DAN TEKNOLOGI INFORMASI

UNIVERSITAS MUHAMMADIYAH SUMATERA UTARA

MEDAN

2026

RANCANG BANGUN RESTFUL API PINTAR MENGGUNAKAN GOLANG DAN GIN FRAMEWORK DENGAN FITUR AUTO CACHE ADAPTIF

SKRIPSI

Diajukan sebagai salah satu syarat untuk memperoleh gelar Sarjana Komputer (S.Kom) dalam Program Studi Tenologi Infromasi. pada Fakultas Ilmu Komputer dan Teknologi Informasi, Universitas Muhammadiyah Sumatera Utara

DITA AULIA AL FARID

NPM. 2109020178

PROGRAM STUDI TEKNOLOGI INFROMASI

FAKULTAS ILMU KOMPUTER DAN TEKNOLOGI INFORMASI

UNIVERSITAS MUHAMMADIYAH SUMATERA UTARA

MEDAN

2026

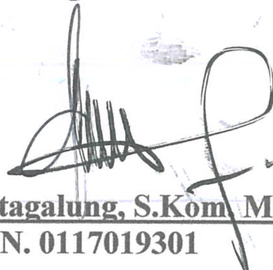
LEMBAR PENGESAHAN

Judul Skripsi	:	RANCANG BANGUN RESTFUL API PINTAR MENGGUNAKAN GOLANG DAN GIN FRAMEWORK DENGAN FITUR AUTO CACHE ADAPTIF
Nama Mahasiswa	:	DITA AULIA AL FARID
NPM	:	2109020178
Program Studi	:	TEKNOLOGI INFORMASI

Menyetujui
Komisi Pembimbing

(Fatma Sari Hutagalung, S.Kom, M.Kom)
NIDN. 0117019301

Ketua Program Studi



(Fatma Sari Hutagalung, S.Kom, M.Kom)
NIDN. 0117019301

Dekan

(Dr. Al-Khowarizmi, S.Kom., M.Kom.)
NIDN. 0127099201

PERNYATAAN ORISINALITAS

RANCANG BANGUN RESTFUL API PINTAR MENGGUNAKAN GOLANG DAN GIN FRAMEWORK DENGAN FITUR AUTO CACHE ADAPTIF

SKRIPSI

Saya menyatakan bahwa karya tulis ini adalah hasil karya sendiri, kecuali beberapa kutipan dan ringkasan yang masing-masing disebutkan sumbernya.

Medan, 07 April 2026

Yang membuat pernyataan

Dita Aulia Al Farid

NPM. 2109020178

PERNYATAAN PERSETUJUAN PUBLIKASI KARYA ILMIAH UNTUK KEPENTINGAN AKADEMIS

Sebagai sivitas akademika Universitas Muhammadiyah Sumatera Utara, saya bertanda tangan dibawah ini:

Nama	: Dita Aulia Al Farid
NPM	: 2109020178
Program Studi	: Teknologi Informasi
Karya Ilmiah	: Skripsi

Demi pengembangan ilmu pengetahuan, menyetujui untuk memberikan kepada Universitas Muhammadiyah Sumatera Utara Hak Bedas Royalti Non-Eksekutif *(Non-Exclusive Royalty free Right)* atas penelitian skripsi saya yang berjudul:

RANCANG BANGUN RESTFUL API PINTAR MENGGUNAKAN GOLANG DAN GIN FRAMEWORK DENGAN FITUR AUTO CACHE ADAPTIF

Beserta perangkat yang ada (jika diperlukan). Dengan Hak Bebas Royalti Non-Eksekutif ini, Universitas Muhammadiyah Sumatera Utara berhak menyimpan, mengalih media, memformat, mengelola dalam bentuk database, merawat dan mempublikasikan Skripsi saya ini tanpa meminta izin dari saya selama tetap mencantumkan nama saya sebagai penulis dan sebagai pemegang dan atau sebagai pemilik hak cipta.

Demikian pernyataan ini dibuat dengan sebenarnya.

Medan, 07 April 2026

Yang membuat pernyataan

Dita Aulia Al Farid

NPM. 2109020178

RIWAYAT HIDUP

DATA PRIBADI

Nama Lengkap	: Dita Aulia Al Farid
Tempat dan Tanggal Lahir	: Stabat, 20 April 2004
Alamat Rumah	: Lingk. V Beringin Blok C No. 12
Telepon/Faks/HP	: 082363312036
E-mail	: ditaauliaalfarid@gmail.com
Instansi Tempat Kerja	: -
Alamat Kantor	: -

DATA PENDIDIKAN

SD	: SDN 054904 BAMBUAN	TAMAT: 2015
SMP	: SMPN 1 STABAT	TAMAT: 2018
SMA	: SMAN 1 STABAT	TAMAT: 2021

KATA PENGANTAR

Puji syukur Penulis ucapkan kepada Allah SWT atas rahmat-Nya Penulis dapat menyelesaikan skripsi ini. Penulisan skripsi ini dilakukan dalam rangka memenuhi salah satu syarat untuk mencapai gelar Sarjana Komputer pada Fakultas Ilmu Komputer dan Teknologi Informasi Universitas Muhammadiyah Sumatera Utara.

Penulis ingin mengucapkan terima kasih yang sebesar-besarnya kepada kedua orang tua yang telah memberikan dukungan moril maupun materil kepada Penulis. Dan Penulis menyadari bahwa tanpa bantuan, bimbingan, dan dukungan dari berbagai pihak sejak masa perkuliahan hingga masa penyusunan skripsi, sulit untuk menyelesaikan skripsi ini. Oleh karena itu dengan segala kerendahan hati, Penulis ingin menyampaikan terima kasih dan rasa hormat kepada:

1. Bapak Prof. Dr. Akrim, M.P., selaku Rektor Universitas Muhammadiyah Sumatera Utara (UMSU)
2. Bapak Dr. Al-Khowarizmi, S.Kom., M.Kom., selaku Dekan Fakultas Ilmu Komputer dan Teknologi Informasi (FIKTI) UMSU.
3. Ibu Dr. Firahmi Rizky, M.Kom., selaku Wakil Dekan I Fakultas Ilmu Komputer dan Teknologi Informasi.
4. Bapak Mhd. Basri, S.Si., M.Kom., selaku Wakil Dekan III Fakultas Ilmu Komputer dan Teknologi Informasi.
5. Ibu Fatma Sari Hutagalung, S.Kom., M.Kom., selaku Ketua Program Studi Teknologi Informasi

6. Bapak Okvi Nugroho, S.Kom., M.Kom., selaku Sekretaris Program Studi Teknologi Informasi
7. Ibu Fatma Sari Hutagalung, S.Kom., M.Kom., selaku Dosen Pembimbing yang telah banyak membantu penulis baik dukungan, saran, dan bimbingan dalam penyusunan penelitian ini.
8. Ibu Karni, S.Pd., M.Pd., selaku Ibu dari penulis yang selalu mendoakan dan memberi dukungan baik mental maupun finansial selama ini.
9. Bapak Drs. Gunadi, M.Pd., selaku Ayah dari penulis yang selalu mendoakan dan memberi dukungan baik mental maupun finansial selama ini.
10. Terimakasih juga kepada Dini Zahra Afiffah, S.E., Dika Nurhafizha, S.Farm., selaku saudara kandung yang berpartisipasi dari segi moral dan dalam menyuport finansial.
11. Terima kasih sebesar-besarnya untuk seluruh teman dan rekan seperjuangan. Dukungan dan masukan dari kalian semua sangat berarti hingga penulis bisa sampai di titik ini
12. Kepada diri sendiri yang sudah berusaha keras dan berjuang sejauh ini. Mampu mengendalikan diri dari berbagai tekanan di luar keadaan dan tak pernah menyerah sesulit apapun. Mampu menguatkan dan meyakinkan bahwa semuanya akan selesai pada waktunya.

Medan, 07 April 2026

Penulis
Dita Aulia Al Farid

RANCANG BANGUN RESTFUL API PINTAR MENGGUNAKAN GOLANG DAN GIN FRAMEWORK DENGAN FITUR AUTO CACHE ADAPTIF

ABSTRAK

Perkembangan aplikasi digital yang semakin pesat menuntut RESTful API untuk mampu menangani beban permintaan yang tinggi secara efisien. Pendekatan caching konvensional dengan nilai Time-to-Live (TTL) yang bersifat statis tidak mampu menyesuaikan diri terhadap dinamika pola akses pengguna, sehingga menyebabkan pemborosan memori atau degradasi performa yang tidak terkendali. Penelitian ini merancang dan membangun RESTful API menggunakan Golang dan Gin Framework yang dilengkapi mekanisme auto-cache adaptif berbasis durasi penggunaan. Sistem mengimplementasikan formula TTL_runtime = TTL_baseline + AdaptCoeff x D_key, di mana nilai TTL disesuaikan secara dinamis berdasarkan durasi aktif setiap cache key. Siklus evaluasi bulanan mengidentifikasi hot key berdasarkan tiga pilar: waktu akses pertama harian (Pilar 1), jumlah cache hit (Pilar 2), dan total durasi aktif sebagai SuggestedTTL (Pilar 3). Hasil pengujian menunjukkan bahwa penerapan cache adaptif berhasil menurunkan max response time sebesar 68,8% dari 711 ms menjadi 222 ms, dengan hit ratio mencapai 99,83%-99,95% pada seluruh variasi intensitas trafik (200-2.000 request). Cache miss selalu tepat satu per sesi terlepas dari volume request, membuktikan efektivitas mekanisme cache-first. Evaluasi siklus bulanan berhasil mengidentifikasi 8 hot key dengan SuggestedTTL bervariasi antara 20,4 hingga 48,6 menit secara proporsional terhadap pola penggunaan aktual.

Kata Kunci: RESTful API; Auto-Cache Adaptif; Golang; Gin Framework; Time-to-Live; Cache Invalidation

DESIGN AND DEVELOPMENT OF SMART RESTFUL API USING GOLANG AND GIN FRAMEWORK WITH ADAPTIVE AUTO-CACHE FEATURE

ABSTRACT

The rapid growth of digital applications demands RESTful APIs capable of efficiently handling high request loads. Conventional caching approaches with static Time-to-Live (TTL) values fail to adapt to dynamic user access patterns, resulting in memory inefficiency or uncontrolled performance degradation. This study designs and develops a RESTful API using Golang and the Gin Framework, equipped with a duration-based adaptive auto-cache mechanism. The system implements the formula TTL_runtime = TTL_baseline + AdaptCoeff x D_key, where TTL is dynamically adjusted based on the active duration of each cache key. A monthly evaluation cycle identifies hot keys through three pillars: daily first access time (Pillar 1), cache hit count (Pillar 2), and total active duration as SuggestedTTL (Pillar 3). Testing results demonstrate that the adaptive cache successfully reduced the maximum response time by 68.8%, from 711 ms to 222 ms, with hit ratios reaching 99.83%-99.95% across all traffic intensity variations (200-2,000 requests). Cache misses consistently numbered exactly one per session regardless of request volume, proving the effectiveness of the cache-first mechanism. The monthly evaluation cycle successfully identified 8 hot keys with SuggestedTTL values ranging from 20.4 to 48.6 minutes, proportional to actual usage patterns.

Keywords: RESTful API; Adaptive Auto-Cache; Golang; Gin Framework; Time-to-Live; Cache Invalidation

DAFTAR ISI

LEMBAR PENGESAHAN i
PERNYATAAN ORISINALITAS ii
PERNYATAAN PERSETUJUAN PUBLIKASI KARYA ILMIAH UNTUK KEPENTINGAN AKADEMIS iii
RIWAYAT HIDUP iv
KATA PENGANTAR v
ABSTRAK vii
ABSTRACT viii
DAFTAR ISI ix
DAFTAR TABEL xi
DAFTAR GAMBAR xii
BAB I 1
PENDAHULUAN 1
1.1 Latar Belakang Masalah 1
1.2 Rumusan Masalah 4
1.3 Batasan Masalah 4
1.4 Tujuan Penelitian 5
1.5 Manfaat Penelitian 5
BAB II 6
LANDASAN TEORI 6
2.1 RESTful API 6
2.2 Go Programming Language 8
2.3 Gin Framework 10
2.4 Auto-Cache Adaptif 11
2.5 Penelitian Terdahulu 13
BAB III 20
METODOLOGI PENELITIAN 20
3.1 Waktu dan Tempat Penelitian 20
3.1.1 Waktu Penelitian 20
3.1.2 Tempat Penelitian 20
3.2 Metode Penelitian 21
3.3 Tahapan Penelitian 24
3.3.1 Diagram Tahapan Penelitian 26
3.4 Analisis Kebutuhan Sistem 26
3.4.1 Kebutuhan Fungsional 27
3.4.2 Kebutuhan Non Fungsional 27
3.4.3 Kebutuhan Perangkat Keras 28
3.4.4 Kebutuhan Perangkat Lunak 28
3.5 Perancangan Alur Sistem 29
3.5.1 Alur Proses Permintaan API 30
3.5.2 Diagram Alur Proses Permintaan 31
3.6 Perancangan Mekanisme Auto-Cache Adaptif 32
3.6.1 Monitoring Durasi 32
3.6.2 Perhitungan TTL Adaptif 33
3.6.3 Mekanisme Auto-Cleanup 34
3.6.4 Mekanisme Cache Invalidation 34

3.6.5 Siklus Evaluasi ... 35
3.7 Perancangan Basis Data ... 36
3.8 Perancangan Endpoint ... 38
BAB IV ... 43
HASIL DAN PEMBAHASAN ... 43
4.1 Implementasi Mekanisme Auto-Cache Adaptif ... 43
4.1.1 Implementasi Mekanisme Pencatatan Akses Key ... 43
4.1.2 Implementasi Perhitungan TTL Adaptif ... 47
4.1.3 Implementasi Mekanisme Auto-Cleanup ... 49
4.1.4 Implementasi Mekanisme Cache Invalidation ... 50
4.1.5 Implementasi Siklus Evaluasi ... 54
4.2 Implementasi Basis Data ... 57
4.3 Implementasi Endpoint API ... 63
4.4 Pengujian Sistem ... 66
4.4.1 Pengujian Fungsional ... 66
4.4.2 Hasil Pengujian Invalidasi Cache ... 72
4.4.3 Hasil Pengujian TTL Adaptif ... 76
4.4.4 Hasil Pengujian Auto-Cleanup ... 77
4.4.5 Hasil Pengujian Siklus Evaluasi 30 Hari ... 78
4.4.6 Hasil Pengujian Performa Sistem ... 81
4.5 Analisis dan Evaluasi Hasil ... 86
4.5.1 Analisis Hasil Pengujian Fungsional ... 87
4.5.2 Analisis Hasil Pengujian Performa dan Beban ... 91
4.5.3 Evaluasi Efektivitas Tiga Pilar Algoritma ... 95
4.5.4 Evaluasi Pencapaian Tujuan Penelitian ... 97
4.5.5 Evaluasi Kesesuaian dengan Rancangan Sistem ... 99
4.5.6 Ringkasan Evaluasi Keseluruhan ... 101
BAB V ... 104
KESIMPULAN DAN SARAN ... **Error! Bookmark not defined.**
5.1 Kesimpulan ... **Error! Bookmark not defined.**
5.2 Saran ... **Error! Bookmark not defined.**
DAFTAR PUSTAKA ... **Error! Bookmark not defined.**

DAFTAR TABEL

Tabel 2.1 Penelitian Terdahulu 14
Tabel 3.1 Jadwal Penelitian 20
Tabel 3.2 Mekanisme kontrol TTL 33
Tabel 3.3 Entity Product 37
Tabel 3.4 Diagram ERD 37
Tabel 3.5 Daftar Endpoint Utama 39
Tabel 4.1 Variabel Pencatatan Statistik dan Fungsinya dalam Sistem 44
Tabel 4.2 Parameter Sistem 47
Tabel 4.3 Perkembangan TTL Runtime Berdasarkan Durasi Aktif D_key 48
Tabel 4.4 Rincian Mekanisme AutoCleanup 50
Tabel 4.5 Tiga Tahap Siklus Evaluasi 30 Hari 57
Tabel 4.6 Strust Utama dalam Paket models 60
Tabel 4.7 Ringkasan Fungsi CRUD 62
Tabel 4.8 Daftar Endpoint API 63
Tabel 4.9 Ringkasan Delapan Hot Key 81
Tabel 4.10 Perbandingan Tanpa Cache dan Dengan Cache 84
Tabel 4.11 Uji Variasi Intensitas Trafik 86
Tabel 4.12 Bukti Kenaikan TTL_Runtime 89
Tabel 4.13 Perbandingan Performa Tanpa Cache dan Dengan Cache 93
Tabel 4.14 Hasil Pengujian Variasi Intensitas Trafik 94
Tabel 4.15 Data AvgFirstAccess Hot Key Hasil Evaluasi 95
Tabel 4.16 Distribusi HIT Count Hot Key Hasil Evaluasi 96
Tabel 4.17 Suggested TTL Hot Key Hasil Evaluasi 97
Tabel 4.18 Evaluasi Pencapaian Tujuan Penelitian 97
Tabel 4.19 Ringkasan Evaluasi Keseluruhan Sistem 101

DAFTAR GAMBAR

Gambar 3.1 Diagram Tahapan Penelitian 26
Gambar 3.2 Diagram Alur Proses Permintaan 31
Gambar 4.1 Struct Penyimpanan Statistik Akses Cache Key 43
Gambar 4.2 Penetapan FirstAccessEver saat Entry Cache Dibuat 45
Gambar 4.3 Pembaruan LastAccessEver pada setiap Cache HIT 46
Gambar 4.4 Implementasi Formula TTL Adaptif 48
Gambar 4.5 Implementasi Goroutine AutoCleanup dan Fungsi tunCleanup() 49
Gambar 4.6 Fungsi Penghapusan Cache 51
Gambar 4.7 Fungsi Invalidasi pada Handler POST 52
Gambar 4.8 Fungsi Invalidasi pada Handler PUT 52
Gambar 4.9 Fungsi Invalidasi pada Handler DELETE 53
Gambar 4.10 Goroutine MonthlyEvaluationCycle 54
Gambar 4.11 Fungsi runMonthlyEvaluation() 56
Gambar 4.12 Pembaruan HotKeyProfiles dan Reset Statistik Baru 56
Gambar 4.13 Fungsi InitDB() 59
Gambar 4.14 Struct Product dan CreateProductRequest 59
Gambar 4.15 Implementasi Fungsi CRUD 61
Gambar 4.16 Handler GetAll() 66
Gambar 4.17 Respon Cache MISS 67
Gambar 4.18 Respon Cache HIT 67
Gambar 4.19 Output setelah Satu Cache MISS dan Cache HIT 68
Gambar 4.20 Respon Cache MISS 69
Gambar 4.21 Respon Cache HIT 69
Gambar 4.22 Respon HTTP 404 untuk ID yang tidak ada 70
Gambar 4.23 Respon Sukses Post dengan Input Valid 70
Gambar 4.24 Respon HTTP 400 untuk Validasi Nama Gagal 71
Gambar 4.25 Respon HTTP 400 untuk validasi Harga Gagal 71
Gambar 4.26 Respon POST Tambah Produk 72
Gambar 4.27 Konfirmasi Cache MISS setelah Invalidasi oleh POST 73
Gambar 4.28 Perintah Pengisian Dua Cache Key Sebelum Operasi PUT 74
Gambar 4.29 Respon PUT Update Produk 74
Gambar 4.30 Konfirmasi Data Terbaru Dikembalikan setelah Invalidasi PUT 75
Gambar 4.31 Respon Sukses DELETE 75
Gambar 4.32 Respon HTTP 404 setelah Produk Dihapus 75
Gambar 4.33 Log Server 76
Gambar 4.34 Pengisian Cache Enam Entry 77
Gambar 4.35 Log AutoCleanup 77
Gambar 4.36 Skrip Shell Pembuatan Traffic dan Trigger Evaluasi Manual 78
Gambar 4.37 Respon GET Setelah Evaluasi 81
Gambar 4.38 Skrip Shell Pengujian Tanpa Cache 82
Gambar 4.39 Output Baseline Performa Tanpa Cache 82
Gambar 4.40 Skrip Shell Request dengan Cache Adaptif 83
Gambar 4.41 Output Performa dengan Cache Adaptif 83
Gambar 4.42 Skrip Uji Trafik Rendah 84
Gambar 4.43 Statistik Cache Hasil Uji Trafik Rendah 85
Gambar 4.44 Skrip Uji Trafik Sedang 85
Gambar 4.45 Statistik Cache Hasil Uji Trafik Sedang 85

Gambar 4.46 Skrip Uji Trafik Tinggi 85
Gambar 4.47 Statistik Cache Hasil Uji Trafik Tinggi 86
Gambar 4.48 Log AutoCleanup setelah 6 Entry Habis..... 90
Gambar 4.49 hasil Pengujian Tanpa Cache 91
Gambar 4.50 Hasil Pengujian dengan Cache Adaptif 92
Gambar 4.51 Statistik Cache Trafik Rendah..... 93
Gambar 4.52 Statistik Cache Trafik Sedang 93
Gambar 4.53 Statistik Cache Trafik Tinggi 93

BAB I

PENDAHULUAN

1.1 Latar Belakang Masalah

Perkembangan teknologi informasi yang semakin pesat telah mendorong munculnya berbagai aplikasi modern seperti layanan e-commerce, aplikasi mobile, hingga perangkat Internet of Things (IoT) yang saling terhubung. Setiap aplikasi tersebut membutuhkan mekanisme komunikasi yang efisien untuk bertukar data secara real-time, yang umumnya diwujudkan melalui Application Programming Interface (API). API berperan sebagai penghubung antara frontend dan backend, memungkinkan pertukaran data secara cepat, terstruktur, dan terintegrasi. Di antara berbagai pendekatan yang tersedia, arsitektur *RESTful API* menjadi standar utama karena sifatnya yang ringan, fleksibel, serta mudah diimplementasikan pada berbagai platform (Santiago et al., 2025).

Namun, seiring meningkatnya jumlah pengguna dan frekuensi permintaan terhadap layanan digital, beban pada server backend juga meningkat secara signifikan. Setiap kali client mengirimkan permintaan yang sama, API harus melakukan proses pengambilan data berulang kali langsung dari database. Proses ini, meskipun sederhana secara arsitektural, menjadi tidak efisien ketika skala pengguna tumbuh pesat. Semakin banyak pengguna yang mengakses sistem, semakin tinggi pula jumlah permintaan yang harus diproses, sehingga waktu respons API menjadi lebih lambat, beban server meningkat, dan konsumsi CPU serta memori menjadi tidak optimal (Zulfa et al., 2020). Kondisi ini mengakibatkan penurunan performa sistem dan pengalaman pengguna. Untuk mengatasi permasalahan tersebut, diperlukan suatu mekanisme optimasi yang mampu

mengurangi frekuensi akses langsung ke database tanpa mengorbankan konsistensi data. Salah satu pendekatan yang umum digunakan adalah caching, yaitu teknik penyimpanan sementara hasil data atau respons API yang sering diakses agar dapat digunakan kembali pada permintaan berikutnya. Dengan adanya caching, sistem tidak perlu melakukan query ke database setiap kali ada permintaan yang sama, sehingga waktu respons menjadi lebih cepat dan beban server berkurang secara signifikan (Suwarjono et al., 2025). Meskipun demikian, sebagian besar implementasi caching saat ini masih bersifat statis. Umumnya, sistem cache menggunakan pengaturan Time to Live (TTL) yang tetap untuk setiap data, tanpa mempertimbangkan dinamika lamanya data tersebut digunakan dalam suatu periode operasional. Akibatnya, data yang hanya digunakan dalam durasi singkat tetap tersimpan terlalu lama dan mengkonsumsi memori secara tidak efisien, sedangkan data yang digunakan secara intensif dalam rentang waktu yang panjang justru dapat terhapus sebelum periode penggunaannya berakhir. Pendekatan statis ini menyebabkan efisiensi cache tidak optimal, terutama pada sistem dengan pola penggunaan data yang berubah-ubah dalam periode waktu tertentu (Larsson et al., 2021).

Berdasarkan kajian terhadap berbagai penelitian dan implementasi RESTful API sebelumnya, belum banyak sistem yang menerapkan mekanisme caching adaptif berbasis analisis durasi waktu penggunaan resource di tingkat API. Sebagian besar sistem masih menggunakan konfigurasi TTL manual yang ditentukan secara tetap oleh pengembang, tanpa mekanisme evaluasi terhadap seberapa lama suatu data digunakan dalam siklus operasional tertentu (Larsson et al., 2021). Hal ini menunjukkan adanya celah penelitian yang dapat diisi dengan

pengembangan pendekatan caching yang mampu menyesuaikan masa aktif cache berdasarkan durasi penggunaan aktual.

Sebagai solusi atas keterbatasan sistem caching konvensional tersebut, penelitian ini mengusulkan pengembangan RESTful API dengan fitur auto-cache adaptif menggunakan Golang dan Gin Framework. Golang dipilih karena efisiensi eksekusi dan dukungan concurrency melalui mekanisme goroutine yang memungkinkan pemrosesan permintaan paralel secara optimal. Sementara itu, Gin Framework digunakan karena ringan, cepat, serta menyediakan struktur pengembangan RESTful API yang sistematis dan mudah dipelihara.

Dengan kombinasi tersebut, sistem dirancang untuk menganalisis durasi waktu penggunaan suatu resource dalam periode tertentu, kemudian menyesuaikan nilai Time to Live (TTL) secara dinamis berdasarkan hasil analisis tersebut. Semakin panjang durasi penggunaan dalam suatu siklus waktu, maka masa aktif cache dapat diperpanjang secara proporsional. Sebaliknya, apabila durasi penggunaan relatif singkat, maka TTL akan dipersingkat agar tidak terjadi pemborosan memori. Selain itu, sistem dilengkapi dengan mekanisme auto-invalidation yang akan memperbarui atau menghapus cache secara otomatis ketika terjadi perubahan pada sumber data utama, sehingga konsistensi informasi tetap terjaga (Sosnowski et al., 2024).

Penelitian ini memiliki urgensi tinggi dalam meningkatkan efisiensi performa RESTful API melalui optimasi perangkat lunak berbasis analisis waktu penggunaan, tanpa harus meningkatkan kapasitas infrastruktur fisik. Pendekatan adaptive autocache berbasis durasi ini diharapkan mampu menghasilkan sistem API

yang lebih responsif, hemat sumber daya, serta adaptif terhadap dinamika penggunaan dalam berbagai skenario aplikasi modern.

Berdasarkan berbagai permasalahan dan solusi yang telah diuraikan, penelitian ini mengambil judul "Rancang Bangun RESTful API menggunakan Golang dan Gin Framework dengan Fitur Auto-cache Adaptif Berbasis Durasi Waktu." Diharapkan penelitian ini dapat menghasilkan model implementasi API yang lebih efisien dan adaptif, serta menjadi referensi dalam pengembangan sistem komunikasi data yang cerdas dan berorientasi pada optimalisasi performa.

1.2 Rumusan Masalah

1. Bagaimana merancang dan membangun *RESTful API* menggunakan Golang dan Gin *Framework* yang memiliki kemampuan *auto-cache* adaptif untuk meningkatkan performa respons sistem?
2. Bagaimana mekanisme kerja auto-cache adaptif dalam menghitung dan menyesuaikan Time to Live (TTL) secara dinamis berdasarkan durasi akses pengguna serta pembatasan nilai maksimum TTL?
3. Bagaimana perbandingan performa antara RESTful API tanpa mekanisme cache adaptif dan RESTful API dengan auto-cache adaptif dalam aspek waktu respons, efisiensi akses database, dan stabilitas penggunaan sumber daya?

1.3 Batasan Masalah

1. Penelitian ini hanya berfokus pada pengembangan *RESTful API* menggunakan bahasa pemrograman Golang dan Gin *Framework* sebagai komponen utama backend.
2. Sistem caching adaptif yang dikembangkan menggunakan heuristic-based algorithm sederhana.

3. Basis data yang digunakan bersifat sederhana (SQLite) dengan satu entitas utama untuk keperluan pengujian performa cachin.
4. Pengujian performa difokuskan pada perbandingan waktu respons dan efisiensi akses database dalam lingkungan pengujian lokal.

1.4 Tujuan Penelitian

1. Membangun prototipe *RESTful API* menggunakan Golang dan Gin *Framework* yang dilengkapi dengan fitur *auto-cache* adaptif untuk mengelola durasi akses.
2. Merancang dan mengimplementasikan mekanisme *auto-cache* adaptif yang mampu menyesuaikan waktu penyimpanan serta penyegaran cache berdasarkan durasi permintaan pengguna.
3. Melakukan pengujian performa antara *RESTful API* konvensional dan API dengan fitur *auto-cache* adaptif untuk mengetahui pengaruhnya terhadap waktu respons, efisiensi sumber daya, dan frekuensi akses database.

1.5 Manfaat Penelitian

1. Memberikan kontribusi ilmiah dalam pengembangan konsep optimasi *RESTful API* melalui penerapan mekanisme *auto-cache* adaptif melalui pendekatan TTL adaptif berbasis durasi akses.
2. Menyediakan model implementasi sederhana namun efektif untuk meningkatkan performa API tanpa memerlukan peningkatan perangkat keras atau arsitektur terdistribusi.
3. Mengurangi beban akses database melalui mekanisme cache yang mampu menyesuaikan waktu penyimpanan secara dinamis sesuai pola penggunaan.

BAB II

LANDASAN TEORI

2.1 RESTful API

Application Programming Interface (API) merupakan antarmuka yang memungkinkan komunikasi antar sistem atau aplikasi dengan cara saling bertukar data. API adalah jembatan penghubung dua aplikasi agar dapat berinteraksi tanpa harus mengetahui detail internal antar sistem. Salah satu bentuk API yang paling popular dan efisien saat ini adalah *RESTful API* (*Representational State Transfer*). *RESTful API* menggunakan protocol HTTP sebagai media pertukaran data dan umumnya memanfaatkan format data ringan seperti JSON atau XML, sehingga komunikasi antara client dan server menjadi cepat, sederhana, dan mudah diintegrasikan lintas platform (Juliawan Pawana et al., 2021). Berbeda dengan API konvensional yang sering mempertahankan status sesi pada sisi server, RESTful API bersifat stateless, di mana setiap permintaan dari client harus memuat seluruh informasi yang dibutuhkan untuk diproses tanpa bergantung pada permintaan sebelumnya. Server tidak menyimpan konteks sesi sehingga setiap request diperlakukan sebagai transaksi independen. Karakteristik ini memberikan keuntungan dalam hal skalabilitas karena server dapat menangani banyak permintaan secara paralel tanpa terbebani penyimpanan sesi. Selain itu, pendekatan ini mempermudah distribusi beban dan pengembangan sistem berskala besar karena setiap komponen dapat bekerja secara modular dan terpisah.

Prinsip dasar arsitektur REST meliputi *stateless, client-server, cacheable, layered system,* dan *uniform interface*. Prinsip stateless menegaskan bahwa setiap permintaan dari client ke server tidak menyimpan konteks, sehingga mempercepat pemrosesan data dan meningkatkan skalabilitas sistem. Prinsip *client-server*

memisahkan tanggung jawab antara client dan server, yang membuat pengembangan sistem menjadi modular dan fleksibel. *Cacheable* memungkinkan data hasil permintaan disimpan sementara pada client yang berfungsi mempercepat waktu tanggapan serta mengurangi beban kerja server. *Layered system* mendukung struktur sistem berlapis seperti penggunaan proxy, *load balancer*, atau *middleware* tanpa memengaruhi cara client berinteraksi dengan server. Terakhir, *uniform interface* memastikan bahwa semua permintaan dan respons dalam sistem mengikuti pola komunikasi yang seragam, sehingga *RESTful API* dapat diintergrasikan secara luas dengan berbagai platform atau bahasa pemrograman. Penerapan kelima prinsip tersebut menghasilkan sistem yang ringan, terukur, dan efisien untuk mendukung kebutuhan layanan modern yang bersifat real-time (Ayyarrappan, 2023).

RESTful API memilikki kelebihan yang menjadikannya pilihan utama dalam pengembangan layanan web dan mobile. Keunggulan utamanya terletak pada skalabilitas tinggi karena menggunakan *protocol* HTTP yang bersifat universal dan skalabilitas yang baik berkat sifat *stateless*-nya, serta kompatibilitas lintas platform yang memudahkan integrasi dengan berbagai sistem. Selain itu, *RESTful API* juga cepat dan efisien karena menggunakan format data sederhana seperti JSON, yang lebih ringan dibandingkan XML. Akan tetapi, *RESTful API* juga memiliki kelemahan utama berupa terjadinya permintaan berulang terhadap sumber daya yang sama yang dapat menyebabkan beban berlebih pada server dan menurunkan performa sistem, terutama ketika jumlah permintaan meningkat secara signifikan. Di sisi lain, karena REST tidak memilik mekanisme pengelolaan secara

bawaan, aspek keamanan, dan manajemen cache perlu dirancang secara terpisah untuk memastikan efisiensi serta stabilitas layanan (Alfarezy Damanik et al., 2020).

Dalam penelitian ini, RESTful API dikembangkan menggunakan bahasa pemrograman Golang dan Gin Framework yang dikenal memiliki performa tinggi serta kemampuan concurrency yang efisien. Meskipun REST telah mendukung prinsip cacheable, implementasi caching konvensional umumnya menggunakan nilai Time-To-Live (TTL) yang bersifat tetap dan tidak mempertimbangkan pola akses pengguna. Oleh karena itu, penelitian ini mengusulkan penerapan auto-cache adaptif berbasis durasi akses sebagai mekanisme optimasi. Pendekatan ini menyesuaikan nilai TTL secara dinamis berdasarkan durasi akses terhadap resource tertentu dengan mempertimbangkan parameter baseline TTL, koefisien adaptasi, serta batas maksimum TTL. Dengan mekanisme tersebut, sistem diharapkan mampu mempertahankan waktu respons yang rendah, mengurangi frekuensi akses database, serta meningkatkan efisiensi penggunaan sumber daya tanpa mengubah prinsip dasar arsitektur REST (Farchani et al., 2025).

2.2 Go Programming Language

Golang atau Go merupakan bahasa pemrograman open-source yang dikembangkan oleh Google dan diperkenalkan pada tahun 2009 oleh Robert Griesemer, Rob Pike, dan Ken Thompson. Bahasa ini dirancang untuk menjawab kebutuhan akan sistem perangkat lunak modern yang menuntut performa tinggi, efisiensi kompilasi, serta kemudahan pemeliharaan kode. Go menggabungkan kecepatan eksekusi yang mendekati C/C++ dengan sintaks yang sederhana dan bersih, sehingga meningkatkan produktivitas pengembang tanpa mengorbankan kinerja sistem. Filosofi desain Go menekankan pada minimalisme, konsistensi struktur kode, serta kemudahan pembacaan, yang menjadikannya sesuai untuk

pengembangan aplikasi backend dan sistem terdistribusi berskala besar (Malhotra, 2025).

Salah satu karakteristik utama Go adalah model concurrency berbasis goroutines dan channels. Goroutines memungkinkan eksekusi fungsi secara paralel dengan konsumsi memori yang sangat ringan dibandingkan thread konvensional, sementara channels menyediakan mekanisme komunikasi yang aman antar proses konkuren. Model ini memungkinkan sistem menangani banyak permintaan secara simultan tanpa kompleksitas manajemen thread manual. Selain itu, Go dilengkapi dengan garbage collector otomatis yang mengelola alokasi dan pembebasan memori secara efisien, sehingga meminimalkan risiko memory leak dan meningkatkan stabilitas aplikasi. Proses kompilasi yang cepat serta kemampuannya menghasilkan file biner mandiri (static binary) membuat Go mudah didistribusikan dan dijalankan pada berbagai platform server tanpa ketergantungan tambahan (Ardiansyah et al., 2022).

Dalam konteks pengembangan API, Go menunjukkan performa yang unggul dalam menangani koneksi simultan dengan latensi rendah dan konsumsi sumber daya yang efisien. Arsitektur runtime Go dirancang untuk mengoptimalkan pemanfaatan CPU melalui penjadwalan goroutine yang adaptif, sehingga sangat cocok untuk sistem yang menerima permintaan dalam jumlah besar secara terus-menerus. Dibandingkan dengan beberapa bahasa lain yang umum digunakan dalam pengembangan backend, Go mampu mempertahankan stabilitas dan efisiensi pada beban kerja intensif, terutama dalam skenario layanan berbasis HTTP yang membutuhkan respons cepat dan konsisten (Ardiansyah et al., 2022). Pemilihan Golang dalam penelitian ini didasarkan pada kemampuannya dalam mendukung

implementasi RESTful API yang menuntut performa tinggi dan skalabilitas yang baik. Efisiensi concurrency yang dimiliki Go sangat relevan untuk mengelola proses pengecekan cache, perhitungan TTL adaptif, serta eksekusi rutin auto-cleanup tanpa mengganggu proses utama penanganan request. Dengan memanfaatkan keunggulan tersebut, sistem yang dikembangkan diharapkan mampu mempertahankan waktu respons yang rendah dan stabil, sekaligus mendukung mekanisme auto-cache adaptif berbasis durasi akses secara optimal dalam kondisi beban tinggi.

2.3 Gin Framework

Gin merupakan framework web berperforma tinggi untuk bahasa pemrograman Go yang dibangun di atas pustaka standar net/http. Framework ini dirancang untuk menyediakan keseimbangan antara kecepatan eksekusi, efisiensi penggunaan sumber daya, dan kemudahan pengembangan layanan berbasis HTTP. Gin dikenal karena kemampuannya menangani request dengan latensi rendah serta menyediakan struktur pengembangan yang sederhana dan terorganisasi. Dukungan bawaan terhadap parsing dan validasi JSON, pengelolaan respons HTTP, serta mekanisme error handling yang efisien menjadikan Gin sebagai salah satu framework yang banyak digunakan dalam pembangunan RESTful API berskala menengah hingga besar (Tarigan, 2024).

Dari sisi arsitektur internal, Gin menggunakan mekanisme routing berbasis struktur data radix tree (varian dari Trie Tree) yang memungkinkan proses pencocokan rute dilakukan secara cepat dan efisien. Struktur ini mengoptimalkan pencarian endpoint meskipun jumlah route dalam aplikasi bertambah signifikan. Pendekatan tersebut sangat penting dalam sistem yang menerima banyak permintaan secara simultan karena dapat meminimalkan overhead pencarian rute

dan menjaga konsistensi performa. Selain itu, Gin menyediakan sistem middleware yang ringan dan modular, sehingga pengembang dapat menambahkan lapisan logika tambahan seperti autentikasi, logging, validasi, atau mekanisme caching tanpa mengubah struktur inti aplikasi. Fleksibilitas ini mendukung pengembangan sistem yang terstruktur, mudah dipelihara, dan scalable (Tarigan, 2024).

Dalam konteks penelitian ini, Gin digunakan sebagai fondasi utama dalam implementasi RESTful API berbasis Golang. Pemilihan framework ini didasarkan pada kemampuannya dalam menangani request secara efisien sekaligus mendukung arsitektur modular yang diperlukan untuk integrasi mekanisme auto-cache adaptif. Struktur handler yang jelas dan dukungan middleware memungkinkan proses pengecekan cache, perhitungan TTL adaptif, serta pengelolaan respons dilakukan secara sistematis tanpa mengganggu alur utama pemrosesan request. Dengan memanfaatkan Gin, sistem API yang dikembangkan diharapkan mampu mempertahankan stabilitas, kecepatan respons, serta fleksibilitas dalam menghadapi perubahan beban kerja secara dinamis.

2.4 Auto-Cache Adaptif

Caching merupakan mekanisme penyimpanan sementara data yang sering diakses untuk mempercepat proses pengambilan data dan mengurangi ketergantungan terhadap sumber utama seperti basis data atau layanan eksternal. Dalam sistem modern, khususnya RESTful API, caching umumnya ditempatkan pada lapisan berkecepatan tinggi seperti memori agar respons terhadap permintaan berulang dapat diberikan secara langsung tanpa melalui proses komputasi atau query yang berulang. Berbagai pendekatan caching telah diterapkan, mulai dari client-side cache, server-side cache, hingga in-memory cache, yang masing-masing bertujuan menekan latensi dan konsumsi sumber daya server. Dalam konteks

layanan berbasis API, penerapan caching terbukti berperan penting dalam meningkatkan respons time dan efisiensi pemrosesan sistem (Mayer et al., 2025).

Meskipun caching konvensional mampu meningkatkan performa sistem, pendekatan TTL statis atau fixed expiration memiliki keterbatasan ketika diterapkan pada sistem dengan beban kerja dinamis. TTL tetap menetapkan durasi kedaluwarsa yang sama tanpa mempertimbangkan variasi durasi operasional maupun perubahan pola beban temporal. Kondisi ini dapat menyebabkan data bertahan terlalu lama sehingga memboroskan memori, atau sebaliknya terlalu cepat kedaluwarsa sehingga meningkatkan akses ulang ke database dan menurunkan stabilitas response time. Penelitian terkait adaptive cache management menegaskan bahwa parameter statis cenderung tidak optimal pada lingkungan dengan workload fluktuatif karena tidak mampu menjaga keseimbangan antara efisiensi memori dan performa sistem secara berkelanjutan (Elsayed et al., 2022).

Sebagai solusi, dikembangkan model auto-cache adaptif berbasis durasi aktif sistem yang menggeser paradigma dari adaptasi berbasis frekuensi akses menuju adaptasi periodik berbasis waktu operasional. Dalam pendekatan ini, TTL tidak ditentukan oleh jumlah permintaan individual, melainkan oleh total durasi aktif sistem dalam periode tertentu. Implementasinya diwujudkan melalui parameter TTL_baseline sebagai nilai dasar durasi penyimpanan cache dalam satu siklus evaluasi. TTL_baseline ditentukan berdasarkan monitoring operasional selama 30 hari, diikuti evaluasi total durasi aktif dan kondisi beban sistem, kemudian dilakukan re-kalibrasi untuk periode berikutnya. Setelah pembaruan dilakukan, counter temporal di-reset dan sistem memasuki siklus baru. Pendekatan periodik ini selaras dengan prinsip adaptive resource management yang

menekankan stabilitas jangka panjang melalui evaluasi terjadwal dibandingkan respons reaktif terhadap setiap event akses (Pratap et al., 2025).

Dalam integrasinya pada RESTful API berbasis Golang dengan Gin Framework, setiap request diperiksa terhadap cache menggunakan TTL_runtime yang mengacu pada TTL_baseline aktif. Seluruh pengelolaan durasi dilakukan melalui siklus evaluasi periodik sehingga perubahan tidak dipengaruhi oleh lonjakan trafik sesaat. Model ini bertujuan meningkatkan stabilitas performa jangka panjang, mengurangi fluktuasi TTL yang tidak terkendali, serta menciptakan kontrol siklus hidup cache yang terukur dan sistematis, sehingga lebih sesuai untuk lingkungan API dengan kebutuhan efisiensi dan konsistensi operasional.

2.5 Penelitian Terdahulu

Penelitian terdahulu menjadi landasan konseptual dalam mengidentifikasi posisi dan kebaruan penelitian ini. Berbagai studi telah membahas pengembangan RESTful API dengan fokus pada peningkatan performa melalui optimalisasi arsitektur, efisiensi bahasa pemrograman, serta penerapan teknik caching. Namun demikian, sebagian besar penelitian masih menerapkan kebijakan caching statis dengan pengaturan Time-To-Live (TTL) tetap atau mekanisme invalidasi sederhana, sehingga belum sepenuhnya mampu beradaptasi terhadap perubahan pola akses pengguna secara dinamis.

Tabel berikut menyajikan beberapa penelitian terdahulu yang relevan dengan topik "Rancang Bangun *RESTful API* menggunakan Golang dan Gin *Framework* dengan Fitur *Auto-cache* Adaptif" beserta ringkasan kontribusi dan temuan utamanya.

Tabel 2.1 Penelitian Terdahulu

NO	Penulis & Tahun	Judul	Deskripsi Hasil
1	Alfarezy Damanik, A., & Voutama, A. (2020)	Pengembangan REST API Untuk Aplikasi Pencarian Pekerjaan Sampingan Dengan Arsitektur Microservices Menggunakan Metode Waterfall	Membahas pengembangan REST API berbasis arsitektur microservices menggunakan Golang. Studi tersebut menunjukkan bahwa pendekatan modular meningkatkan fleksibilitas dan pengelolaan layanan.
2	Ardiansyah, H., & Fatwanto, A. (2022)	Comparison of Memory usage between REST API in Javascript and Golang	Membandingkan konsumsi memori REST API berbasis JavaScript dan Golang. Hasil penelitian menunjukkan bahwa Golang menggunakan memori lebih efisien dan stabil pada permintaan berskala besar, mendukung pemilihan Golang untuk API berperforma tinggi.

3	Ayyarrappan, M. (2023)	Architecting REST APIs for High-Performance Applications	Menguraikan prinsip arsitektur REST API berperforma tinggi, termasuk optimasi *routing*, *middleware*, dan efisiensi pengolahan data. Studi ini memberikan dasar teoretis dalam merancang API yang *scalable* dan cepat.
4	Elsayed, K., & Rizk, A. (2022)	On the Impact of Network Delays on Time-to-Live Caching	Menganalisis dampak network delay terhadap mekanisme caching berbasis TTL menggunakan pendekatan matematis Markov Arrival Process (MAP) dan simulasi numerik.
5	Farchani, S. B., Hermanto, N., & Kusuma, B. A. (2025)	Implementasi REST API dalam Pengembangan Backend Inventory Peminjaman	Menunjukkan bahwa implementasi REST API berbasis Golang mampu meningkatkan stabilitas backend dan kecepatan akses data
6	Hendri, H., Hartati, R. S., Linawati, L., &	Optimizing CDN Modeling with API	Menyoroti penggunaan teknik TTL caching dalam integrasi API pada sistem berbasis web

	Wiharta, D. M. (2024)	Integration Using Time To-Live (TTL) Caching Technique	
7	Juliawan Pawana, A. W. I. et al. (2021)	Identifikasi Kandidat Microservices Dengan Analisis Domain Driven Design	Menjelaskan metode identifikasi modul sistem menggunakan *Domain Driven Design* (DDD) untuk *microservices.* Relevan dalam konteks pembagian komponen API dan modularitas dalam arsitektur sistem
8	Larsson, L. et al. (2021)	Adaptive and Application-agnostic Caching in Service Meshes for Resilient Cloud Applications	Memperkenalkan konsep caching adaptif yang bersifat application-agnostic dalam lingkungan service mesh untuk aplikasi cloud.
9	Malhotra, A. (2025)	*Concurrency* Patterns in Golang: Real-World Use	Mengkaji pola *concurrency* di Golang, termasuk *goroutines* dan channel.

		Cases and Performance	
10	Mayer, H., & Richards, J. (2025)	Comparative Analysis of Distributed Caching Algorithms: Performance Metrics and Implementation Considerations	Penelitian ini menegaskan bahwa konfigurasi parameter caching, termasuk pengelolaan TTL, sangat memengaruhi efisiensi penggunaan sumber daya.
11	Pratap, S. R. et al. (2025)	Adaptive Retention and Eviction for Efficient Caching in AI-Driven Systems	Konsep ini menunjukkan bahwa pengaturan dinamis terhadap masa simpan data dapat meningkatkan efisiensi cache secara signifikan.
12	Santiago & Benesius (2025)	Journal of Information System, Applied, Management, Accounting and Research	Penelitian mengenai pengembangan sistem informasi modern, termasuk penggunaan API dan teknik optimasi performa. Mendukung landasan teoritis pengembangan API yang *scalable*

13	Sosnowski, M., Seck, R., Wiedner, F., & Carle, G. (2024)	CRDT Web Caching: Enabling Distributed Writes and Fast Cache Consistency for REST APIs	Mengkaji mekanisme caching berbasis CRDT pada REST API di lingkungan terdistribusi dan membuktikan peningkatan throughput serta konsistensi data.
14	Suwarjono, & Averoes, F. (2025)	Peningkatan Performa Aplikasi Web Dinamis Berbasis PHP melalui Implementasi Redis Caching	Menunjukkan bahwa optimasi pada lapisan backend, khususnya dalam pengelolaan akses data, berdampak langsung terhadap peningkatan waktu respons sistem. Penelitian ini menegaskan pentingnya optimasi backend sebagai faktor kunci dalam meningkatkan performa API berbasis web.
15	Tarigan, F. (2024)	Penggunaan Gin Dalam Membuat API Pada Aplikasi Akademik Berbasis Web	Menunjukkan bahwa Framework Gin memberikan performa routing tinggi dan stabilitas dalam pembangunan REST API.

16	Zulfa, M. I. et al. (2020)	Application Caching Strategy Based on *In-memory* Using Redis Server	Menggambarkan strategi caching menggunakan Redis untuk mempercepat akses data. Relevan sebagai dasar pendekatan caching pada API sebelum dikembangkan lebih lanjut menjadi caching adaptif.

BAB III
METODOLOGI PENELITIAN

3.1 Waktu dan Tempat Penelitian

Penyusunan jadwal penelitian bertujuan memberikan gambaran waktu yang jelas terhadap setiap tahapan yang akan dilaksanakan. Dengan adanya jadwal ini, seluruh proses penelitian dapat berjalan sesuai dengan tujuan yang telah ditetapkan. Selain itu. Penelitian ini dilakukan pada lingkungan kerja dan perangkat yang mendukung proses pengembangan serta pengujian sistem yang dibagun.

3.1.1 Waktu Penelitian

Penelitian ini dilaksanakan dalam kurun waktu lima bulan, yang meliputi beberapa tahapan berikut:

Tabel 3.1 Jadwal Penelitian

<table>
<tr><th rowspan="2">No</th><th rowspan="2">Kegiatan</th><th colspan="6">Waktu</th></tr>
<tr><th>November</th><th>Desember</th><th>Januari</th><th>Februari</th><th>Maret</th><th>April</th></tr>
<tr><td>1</td><td>Penulisan proposal</td><td>X</td><td>X</td><td></td><td></td><td></td><td></td></tr>
<tr><td>2</td><td>Bimbingan proposal</td><td>X</td><td>X</td><td>X</td><td></td><td></td><td></td></tr>
<tr><td>3</td><td>Penelitian dan tindakan</td><td></td><td></td><td>X</td><td>X</td><td>X</td><td></td></tr>
<tr><td>4</td><td>Analisis dan bimbingan hasil penelitian</td><td></td><td></td><td></td><td>X</td><td>X</td><td>X</td></tr>
</table>

3.1.2 Tempat Penelitian

Penelitian ini dilaksanakan pada lingkungan non-fisik yang berfokus pada proses pengembangan perangkat lunak. Seluruh kegiatan perancangan, implementasi, dan pengujian sistem dilakukan menggunakan perangkat komputer pribadi sebagai *local development environment* yang dikonfigurasi khusus untuk kebutuhan penelitian. Pengembangan *RESTful API* dilakukan secara mandiri

dengan memanfaatkan bahasa pemrograman Go dan Gin *Framework*, sedangkan proses pengujian dilaksanakan pada server lokal untuk menilai performa sistem, seperti waktu respons, efisiensi *cache*, serta stabilitas fitur auto-cache adaptif. Pemilihan lokasi penelitian yang bersifat fleksibel dan dapat dikendalikan ini memungkinkan proses evaluasi berjalan lebih terukur dan sesuai dengan kebutuhan teknis penelitian.

3.2 Metode Penelitian

Metode penelitian merupakan pendekatan yang digunakan untuk memperoleh data dan informasi secara sistematis guna menjawab permasalahan penelitian serta mencapai tujuan yang telah ditetapkan. Penelitian ini menggunakan metode rekayasa perangkat lunak (*Software engineering research*) dengan pendekatan kualitatif deskriptif. Jenis penelitian ini bersifat terapan (*Applied research*), karena berfokus pada perancangan, pembangunan, dan evaluasi sistem yang dapat diimplementasikan secara langsung dalam pengembangan sistem backend modern.

Pendekatan ini dipilih karena sesuai dengan karakteristik penelitian rekayasa perangkat lunak yang menekankan pada perancangan solusi, implementasi sistem, dan evaluasi perilaku hasil rancangan. Dengan demikian, penelitian ini diharapkan mampu menghasilkan model *RESTful API* yang lebih responsif, efisien dalam penggunaan sumber daya, dan adaptif terhadap pola permintaan data pengguna, tanpa bergantung pada peningkatan kapasitas perangkat keras.

Penelitian ini berfokus pada perancangan dan pembangunan Resful API pintar dengan fitur *auto-cache* adaptif menggunakan Bahasa pemrograman Golang dan Gin *Framework*. Evaluasi sistem dilakukan dengan cara membandingkan

karakteristik dan perilaku sistem antar *RESTful API* tanpa mekanisme cache adaptif dan *RESTful API* yang telah menerapkan fitur *auto-cache* adaptif. Perbandingan ini bertujuan untuk menggambarkan perbedaan kinerja dan efisiensi sistem kerja secara deskriptif.

Metode eksperimental dalam penelitian ini digunakan dalam konteks eksperimen rekayasa perangkat lunak, yaitu melalui uji coba langsung terhadap sistem yang dibangun. Dua kondisi sistem diuji, yaitu *RESTful API* tanpa fitur *auto-cache* adaptif sebagai sistem awal, dan *RESTful API* dengan penerapan *auto-cache* adaptif sebagai sistem hasil pengembangan.

Pelaksanaan penelitian ini dilakukan melalui beberapa tahapan metodologis yang terstruktur. Tahap pertama adalah analisis kebutuhan sistem (*requirement analysis*), yang bertujuan untuk mengidentifikasi permasalahan performa pada RESTful API konvensional, khususnya terkait tingginya frekuensi akses berulang ke basis data dan keterbatasan mekanisme caching statis. Pada tahap ini ditentukan kebutuhan fungsional, seperti pengelolaan request–response API dan mekanisme auto-cache adaptif, serta kebutuhan non-fungsional yang mencakup performa, skalabilitas, efisiensi memori, dan stabilitas sistem.

Tahap berikutnya adalah perancangan sistem (*system design*), yang meliputi perancangan arsitektur RESTful API berbasis Golang dan Gin Framework, desain modul cache adaptif, serta perumusan algoritma penyesuaian Time-To-Live (TTL) berbasis durasi penggunaan resource. Selain itu, dirancang pula mekanisme auto-invalidation untuk menjaga konsistensi data ketika terjadi perubahan pada sumber data utama. Pada tahap ini ditentukan pula alur kerja sistem, termasuk proses

pengecekan cache (*cache lookup*), evaluasi TTL, pembaruan nilai TTL_baseline, serta siklus monitoring periodik.

Tahap implementasi dilakukan dengan mengembangkan sistem menggunakan bahasa pemrograman Golang yang mendukung concurrency melalui goroutine, serta memanfaatkan Gin Framework untuk pengelolaan routing dan middleware. Mekanisme auto-cache adaptif diintegrasikan pada lapisan middleware agar proses pengecekan cache dan penyesuaian TTL dapat dilakukan sebelum request diproses lebih lanjut ke database. Implementasi juga mencakup modul monitoring durasi operasional sistem yang berfungsi sebagai dasar evaluasi periodik dalam menentukan nilai TTL secara dinamis.

Setelah sistem selesai dikembangkan, dilakukan tahap pengujian performa (*performance testing*) menggunakan skenario permintaan berulang dengan variasi intensitas trafik. Pengujian difokuskan pada parameter respons time, frekuensi akses database, serta stabilitas sistem dalam kondisi beban berbeda. Pendekatan pengujian dilakukan secara deskriptif-komparatif dengan membandingkan perilaku sistem sebelum dan sesudah penerapan mekanisme auto-cache adaptif.

Data hasil pengujian kemudian dianalisis secara kualitatif untuk mengevaluasi dampak penerapan caching adaptif terhadap efisiensi sistem. Analisis difokuskan pada perubahan pola akses data, pengurangan beban database, konsistensi respons API, serta stabilitas performa dalam periode operasional tertentu. Evaluasi ini bertujuan untuk mengidentifikasi sejauh mana mekanisme adaptif berbasis durasi mampu meningkatkan efisiensi tanpa menimbulkan fluktuasi TTL yang tidak terkendali.

Pendekatan penelitian ini termasuk dalam kategori penelitian rekayasa perangkat lunak (*software engineering research*) dengan model perancangan dan pembangunan artefak sistem (*design and build approach*). Fokus penelitian tidak hanya pada implementasi teknis, tetapi juga pada evaluasi perilaku sistem hasil rancangan. Dengan metodologi tersebut, penelitian ini diharapkan menghasilkan model RESTful API yang lebih responsif, efisien dalam penggunaan sumber daya, serta adaptif terhadap dinamika pola permintaan pengguna tanpa ketergantungan pada peningkatan kapasitas infrastruktur fisik.

3.3 Tahapan Penelitian

Tahapan penelitian ini disusun secara sistematis berdasarkan pendekatan *Research and Development* (R&D) dalam bidang rekayasa perangkat lunak (*software engineering*). Setiap tahap dirancang untuk memastikan proses pengembangan *RESTful API* pintar dengan fitur *auto-cache* adaptif berjalan terukur, terarah, dan menghasilkan sistem yang valid secara fungsional maupun performa.

1. Tahap penentuan topik penelitian, tahap awal ini dilakukan dengan mengidentifikasi masalah utama pada performa *RESTful API*, khususnya dalam menangani permintaan berulang, serta menentukan solusi melalui penerapan fitur *auto-cache* adaptif menggunakan Golang dan Gin *Framework*.
2. Tahap perumusan masalah dan tujuan penelitian, dilakukan perumusan masalah yang spesifik dan penentuan tujuan penelitian agar fokus penelitian terarah pada peningkatan efisiensi sistem API melalui caching adaptif.

3. Studi literatur, mengumpulkan teori, konsep, dan hasil penelitian terdahulu yang relevan mengenai *RESTful API*, Golang, Gin *Framework*, dan mekanisme caching adaptif untuk memperkuat dasar teoritis penelitian.
4. Analisis kebutuhan sistem, menentukan kebutuhan fungsional dan non-fungsional sistem.
5. Perancangan sistem, membuat desain arsitektur sistem, alur mekanisme *auto-cache* adaptif, serta struktur data dan alur komunikasi antar komponen API.
6. Implementasi sistem, mewujudkan rancangan menjadi kode program menggunakan Golang dan Gin *Framework*.
7. Pengujian sistem, melakukan pengujian sistem untuk mengukur waktu respons, cache *hit ratio*, penggunaan CPU, dan *throughput*.
8. Analisis dan evaluasi hasil, menganalisis hasil pengujian untuk menilai efektivitas penerapan *auto-cache* adaptif, serta menafsirkan hasil dalam konteks peningkatan efisiensi API.
9. Kesimpulan dan saran, menyimpulkan temuan utama penelitian terkait performa sistem dan memberikan saran pengembangan lebih lanjut.

3.3.1 Diagram Tahapan Penelitian

Gambar 3.1 Diagram Tahapan Penelitian

3.4 Analisis Kebutuhan Sistem

Pada tahap ini, kebutuhan sistem diidentifikasikan secara menyeluruh agar solusi yang dirancang benar-benar menjawab permasalahan performa *RESTful API*, khususnya dalam menangani permintaan berulang melalui penerapan fitur *auto-cache* adaptif. Dengan memahami kebutuhan fungsional dan non fungsional secara

tepat, proses perancangan sistem. Pada tahap selanjutnya dapat disusun secara terarah dan efisien, sehingga menghasilkan arsitektur sistem yang optimal dalam meningkatkan kecepatan respons dan efisiensi kinerja API.

3.4.1 Kebutuhan Fungsional

Pada sistem *RESTful API* dengan fitur *auto-cache* adaptif yang dibangun, terdapat beberapa fungsi inti yang harus dijalankan untuk memastikan sistem mampu mengelola data, memaksimalkan performa, dan mengatur mekanisme cache secara cerdas berdasarkan durasi waktu penggunaan. Berikut adalah kebutuhan fungsional yang didefenisikan:

1. Sistem menyimpan metadata waktu awal akses (FirstAccessEver)
2. Sistem menghitung durasi penggunaan
3. Sistem menyesuaikan TTL secara otomatis
4. Sistem menjalankan evaluasi global 30 hari
5. Sistem melakukan invalidasi saat CRUD
6. Sistem menyediakan endpoint statistik berbasis durasi

3.4.2 Kebutuhan Non Fungsional

Tidak seperti kebutuhan fungsional yang berfokus pada pada apa yang dilakukan sistem, kebutuhan non-fungsional menyoroti bagaimana sistem tersebut bekerja dari aspek performa, keandalan, keamanan, dan efisiensi. Kebutuhan non-fungsional untuk sistem API dengan fitur *auto-cache* adaptif ini dijabarkan sebagai berikut:

1. Efisiensi performa
2. Efisiensi memori
3. Keamanan

4. Reabilitas dan skalabilitas
5. Portabilitas

3.4.3 Kebutuhan Perangkat Keras

Untuk mengembangkan dan menguji sistem *RESTful API* dengan fitur *Auto-cache* Adaptif, perangkat keras yang diperlukan relatif ringan karena penelitian ini dilakukan menggunakan database lokal (SQLite) dan cache *in-memory* tanpa server eksternal. Kebutuhan perangkat keras yang digunakan:

1. Laptop, untuk rancang bangun sistem ini. Laptop yang digunakan memiliki spesifikasi:
   a. Prosesor: AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx
   b. RAM: 8 GB
   c. Penyimpanan: 256 GB

3.4.4 Kebutuhan Perangkat Lunak

Perangkat Lunak utama yang digunakan untuk menjalankan sistem adalah sebagai berikut:

1. Bahasa Pemrograman dan Runtime
   a. GO programming, software inti yang dibutuhkan untuk menjalankan kode API, membuat cache adaptif, mengatur *routing*, dan menjalankan *goroutine*.
2. *Framework* dan Library
   a. Gin *Framework*, digunakan untuk *routing* API, *middleware*, dan HTTP handling. Gin dipilih karena sangat ringan, berkecepatan tinggi, dan cocok untuk proyek yang berfokus pada performa.
   b. Sqlx dan SQLite, kedua database digunakan karena memiliki fungsi masing-masing. SQLite digunakan karena tidak butuh instalasi server database,

praktis, dan cukup cepat untuk penelititian API. Lalu Sqlx digunakan untuk mempermudah query.

c. Custom *In-memory* Adaptive Cache, penelitian ini membuat cache sendiri yang dilengkapi dengan fitur TTL adaptif, Frekuensi akses, *auto-cleanup*, serta *Dynamic extension* TTL.

3. Perangkat Pengujian
   a. Curl, perintah di dalam Git Bash/ untuk melakukan pengujian melalui command line, automasi request, simulasi trafik sederhana.
   b. Postman, melakukan pengujian menggunakan aplikasi tambahan untuuk menampilkan hasil panggilan.
4. Editor dan Tools
   a. Visual studio code, digunakan sebagai platform penulisan program, menjalankan module GO, Debugging, Serta Visualisasi struktur folder.
   b. Git Bash, berfungsi untuk menjalankan server GO dan melakukan curl request.

3.5 Perancangan Alur Sistem

Perancangan alur sistem menggambarkan mekanisme kerja API auto-cache adaptif secara menyeluruh, dimulai dari permintaan client, proses pengecekan cache, percabangan kondisi cache hit atau miss, perhitungan durasi akses, penyesuaian TTL adaptif, hingga pembaruan metadata cache dan pengiriman respons. Perancangan ini memastikan bahwa sistem mampu mengoptimalkan performa API melalui pendekatan cache-first strategy dengan mekanisme TTL adaptif berbasis durasi akses.

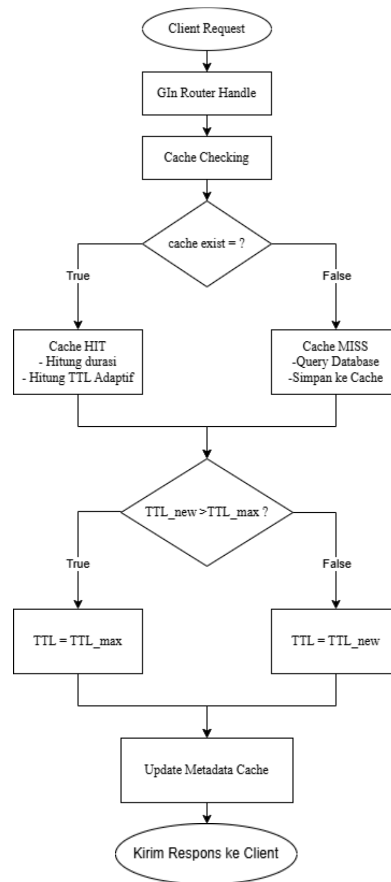
3.5.1 Alur Proses Permintaan API

Ketika client mengirim permintaan, sistem akan menjalankan proses:

1. Client mengirim request HTTP ke server. Gin Router menerima request dan meneruskannya ke handler sesuai endpoint yang dipanggil.
2. Handler melakukan pengecekan apakah data tersedia di dalam memori cache.
3. Apabila data tersedia di cache:
   a. Sistem mengambil data dari cache
   b. Sistem mengambil metadata waktu awal
   c. Sistem menghitung durasi akses
   d. Sistem menghitung TTL adaptif
   e. Sistem melakukan validasi batas sistem
4. Apabila data tidak ditemukan di cache:
   a. Handler melakukan query ke database
   b. Sistem menetapkan awal waktu
   c. Data disimpan ke cache dengan TTL awal
   d. Sistem menetapkan kedaluwarsa
5. Baik pada kondisi HIT maupun MISS, sistem melakukan pembaruan metadata cache sebelum respons dikirimkan.
6. Setelah seluruh proses selesai, sistem mengirimkan respons HTTP dalam format JSON ke client melalui Gin.

3.5.2 Diagram Alur Proses Permintaan

Gambar 3.2 Diagram Alur Proses Permintaan



Deskripsi naratif diagram sistem API auto-cache adaptif:

1. Client mengirim request HTTP ke server.
2. Gin menerima request dan meneruskannya ke handler.
3. Handler melakukan pengecekan keberadaan data di cache.
4. Jika cache tersedia, sistem menghitung durasi akses dan memperbarui TTL secara adaptif dengan mempertimbangkan batas maksimum TTL.
5. Jika cache tidak tersedia, sistem mengambil data dari database dan menyimpannya ke cache dengan TTL awal.
6. Sistem memperbarui metadata cache berupa waktu kedaluwarsa.
7. Sistem mengirim respons kembali ke client.

3.6 Perancangan Mekanisme Auto-Cache Adaptif

Mekanisme *auto-cache* adaptif berfungsi untuk mengatur siklus hidup cache berdasarkan waktu operasional sistem, bukan berdasarkan jumlah permintaan atau tingkat popularitas data. Pendekatan ini menitikberatkan pada stabilitas jangka panjang melalui evaluasi periodik durasi aktif sistem. Mekanisme yang dirancang mencakup proses monitoring waktu operasional, perhitungan TTL adaptif berbasis durasi, penghapusan cache berbasis waktu kedaluwarsa, serta mekanisme invalidasi akibat perubahan data melalui operasi CRUD. Dengan desain ini, sistem cache menjadi lebih deterministik, terkontrol, dan tidak reaktif terhadap lonjakan trafik sesaat.

3.6.1 Monitoring Durasi

Sistem menetapkan titik awal siklus evaluasi yang dinotasikan sebagai (t_0). Nilai ini ditentukan pada dua kondisi:

1. FirstAccessEver
2. LastAccessEver

Dengan kedua nilai ini, durasi aktif setiap cache key dapat dihitung menggunakan rumus:

$$D_key = LastAccessEver - FirstAccesssEver$$

Nilai D_key bersifat akumulatif dan terus bertambah seiring berjalannya waktu operasional sistem selama key tersebut aktif digunakan. D_key yang besar mengindikasikan bahwa resource tersebut digunakan secara konsisten dalam rentang waktu yang panjang, sehingga layak mendapatkan TTL yang lebih lama. Sebaliknya, D_key yang kecil mengindikasikan penggunaan yang singkat, sehingga TTL tidak perlu diperpanjang secara signifikan.

Selain D_key, sistem juga mencatat DailyRecords, yaitu rekaman aktivitas per hari yang menyimpan jam pertama akses harian (FirstTime) untuk setiap tanggal selama satu siklus evaluasi. Data ini digunakan oleh mekanisme WarmupScheduler pada siklus berikutnya.

3.6.2 Perhitungan TTL Adaptif

Tahap berikutnya adalah menerjemahkan nilai tersebut ke dalam TTL yang digunakan selama periode berjalan. Sistem menggunakan TTL_baseline sebagai nilai dasar durasi penyimpanan cache dalam satu siklus.

Penyesuaian dilakukan melalui rumus:

$$TTL_{runtime} = TTL_{baseline} + AdaptCoeff \times D_{key}.Seceonds()$$

Dengan keterangan:

1. TTL_runtime → TTL yang digunakan pada periode berjalan
2. TTL_baseline → TTL dasar sistem
3. TTL_max → batas maksimum TTL
4. AdaptCoeff → koefisien adaptasi yang menentukan pertambahan TTL

Untuk menjaga stabilitas sistem, diterapkan batas maksimum TTL, yaitu TTL_max. Mekanisme kontrolnya adalah:

Tabel 3.2 Mekanisme kontrol TTL

Kondisi	**Keputusan Sistem**
TTL_runtime ≤ TTL_max	Gunakan TTL_runtime
TTL_runtime > TTL_max	Set TTL_runtime = TTL_max

3.6.3 Mekanisme Auto-Cleanup

Auto-cleanup berfungsi sebagai sistem pemeliharaan otomatis untuk memastikan setiap entry cache yang telah melewati batas waktunya segera dihapus. Setiap entry cache memiliki atribut ExpireAt ditentukan saat entry dibuat, dengan rumus:

$$t_expire = t_now + TTL_runtime$$

Sistem secara berkala melakukan pengecekan dengan membandingkan waktu sekarang (t_now) terhadap t_expire. Aturan penghapusannya adalah:

$$t_now \geq t_expire$$

Penghapusan dilakukan menggunakan write lock (mu.Lock()) untuk memastikan operasi bersifat atomic dan tidak menimbulkan race condition dengan goroutine lain yang sedang mengakses cache secara bersamaan. Jumlah entry yang dihapus dicatat dalam variabel CleanupCount yang dapat dipantau melalui endpoint GET /api/cache/stats.

3.6.4 Mekanisme Cache Invalidation

Meskipun sistem menggunakan TTL adaptif berbasis durasi, mekanisme cache invalidation tetap diperlukan untuk menjamin konsistensi antara cache dan database ketika terjadi perubahan data melalui operasi CRUD

1. POST, ketika data baru berhasil ditambahkan ke database, cache key products:all dihapus agar request GET all berikutnya mengambil daftar terbaru dari database.
2. PUT, ketika data berhasil diperbarui, dua cache key dihapus sekaligus yaitu products:all dan products:id:{id} yang sesuai dengan ID produk yang diperbarui.

3. DELETE, ketika data berhasil dihapus dari database, dua cache key yang sama dengan operasi PUT dihapus untuk memastikan tidak ada data stale yang dikembalikan dari cache.

3.6.5 Siklus Evaluasi

Mekanisme kontrol periodik yang berfungsi untuk melakukan kalibrasi ulang terhadap TTL_baseline berdasarkan performa operasional sistem dalam satu periode penuh. Pendekatan ini memastikan bahwa sistem tidak hanya adaptif pada level runtime (jangka pendek), tetapi juga melakukan evaluasi strategis dalam horizon waktu yang lebih panjang.

1. Akses Awal (First Access Time). Sistem mencatat jam pertama kali setiap cache key diakses pada masing-masing hari selama satu siklus evaluasi. Rata-rata jam akses pertama harian (AvgFirstHour dan AvgFirstMinute) kemudian dihitung dari seluruh DailyRecords. Nilai ini digunakan oleh WarmupScheduler pada siklus berikutnya untuk menjadwalkan pre-warming cache sebelum pengguna datang.
2. Cache Hit (Seleksi Hot Key). Hanya cache key yang memiliki TotalHits > 0 selama satu siklus evaluasi yang akan masuk sebagai hot key dan mendapatkan profil SuggestedTTL untuk siklus berikutnya. Key yang tidak pernah diakses selama satu bulan tidak akan mendapat profil hot key dan akan menggunakan TTL_baseline default pada bulan berikutnya.
3. Durasi Aktif (SuggestedTTL). Durasi aktif key selama satu siklus (D_key = LastAccessEver - FirstAccessEver) menjadi nilai SuggestedTTL yang direkomendasikan untuk key tersebut di bulan berikutnya. Semakin lama data digunakan, semakin lama pula ia akan di-cache pada siklus berikutnya, dengan batas minimum TTL_baseline dan batas maksimum TTL_max.

Proses siklus evaluasi berlangsung dalam tiga tahap:

1. Pengumpulan statistik. Selama 30 hari, goroutine MonthlyEvaluationCycle berjalan setiap 1 jam untuk memeriksa apakah bulan telah berganti. Selama periode berjalan, setiap cache HIT secara otomatis memperbarui statistik melalui fungsi Get(): mencatat DailyRecords.FirstTime, menambah TotalHits, dan memperbarui LastAccessEver.
2. Evaluasi saat pergantian bulan. Saat pergantian bulan terdeteksi, sistem menjalankan fungsi runMonthlyEvaluation() yang mengidentifikasi seluruh key dengan TotalHits > 0, menghitung AvgFirstHour dan AvgFirstMinute dari DailyRecords, menghitung SuggestedTTL dari D_key yang diklem antara TTL_baseline dan TTL_max, kemudian menyimpan profil HotKeyProfile beserta jadwal pre-warming ke dalam hotKeyProfiles.
3. Pembaruan parameter dan reset statistik. Setelah evaluasi selesai, hotKeyProfiles diperbarui dengan profil bulan baru, currentMonth dan currentYear diperbarui ke bulan berjalan, dan riwayat evaluasi disimpan dalam evalHistory dengan retensi maksimum 12 bulan terakhir. Counter dan statistik siklus sebelumnya di-reset sehingga akumulasi dimulai dari awal untuk siklus baru.

3.7 Perancangan Basis Data

Hal ini dilakukan untuk memastikan bahwa sistem API dapat menyimpan, mengelola, dan menyediakan data secara konsisten dan efisien. Database ini dirancang secara sederhana karena fokus penelitian bukan pada kompleksitas relasi antar tabel. Sistem menggunakan SQLite sebagai database lokal yang ringan, mudah digunakan, dan tidak memrlukan konfigurasi server terpisah.

1. Entity utama dalam sistem

   a. Entity: products

      Tabel produts digunakan untuk menyimpan daftar produk yang dapat diambil, diperbarui, ditambahkan, maupun dihapus melalui API.

      Setiap baris dalam table ini mewakili satu produk yang memiliki informasi dasar berupa:

Tabel 3.3 Entity Product

Atribut	**Tipe Data**	**Deskripsi**
Id	INTEGER	Identitas unik produk
Name	TEXT	Nama produk
Price	REAL	Harga produk

2. Skema database

Tabel produk dirancang sebagai tempat penyimpanan utama untuk seluruh data produk. Struktur tabel dikonfigurasikan dengan PRAGMA journal_mode=WAL untuk mendukung pembacaan paralel tanpa memblokir penulisan, sesuai kebutuhan sistem yang menangani banyak request bersamaan, serta PRAGMA foreign_keys=ON untuk penegakan integritas referensial.

Tabel 3.4 Diagram ERD

Products
id :INTEGER (PK) name : TEXT price : REAL

3. Cara akses data oleh layer API

   Layer API berinteraksi dengan database melalui paket db yang mengenkapsulasi seluruh operasi CRUD menggunakan parameterized query untuk mencegah SQL injection:

   a. Pengambilan data seluruh produk
   b. Penambahan data produk baru
   c. Pembaruan data produk berdasarkan ID
   d. Penghapusan data produk dan invalidasi cache

4. Hubungan dengan sistem auto-cache adaptif

   Basis data berperan sebagai data provide utama:

   a. Saat cache HIT, database tidak diakses karena data tersedia di cache
   b. Saat data MISS, database melakukan query kemudian hasilnya disimpan ke cache dengan TTL awal
   c. Saat CRUD, setiap perubahan data akan menginvalidasi cache untuk menjaga konsistensi data

3.8 Perancangan Endpoint

Perancangan *endpoint* bertujuan untuk mendefinisikan layanan HTTP yang dapat digunakan client dalam melakukan operasi CRUD pada data produk sekaligus mengaktifkan mekanisme *auto-cache* adaptif. Seluruh *endpoint* dirancang mengikuti prinsip *RESTful API* dan menghasilkan respons dalam format JSON. *Endpoint* berada pada dua jalur utama, yaitu jalur produk dan jalur cache.

1. Daftar *endpoint* utama

Berikut adalah daftar *endpoint* utama yang tersedia dalam sistem:

Tabel 3.5 Daftar Endpoint Utama

Endpoint	**Method**	**Deskripsi**	**Output**
/api/products	GET	Mengambil seluruh data produk (menggunakan cache adaptif)	JSON array
/api/producs/:id	GET	Mengambil data berdasarkan ID	JSON array
/api/products	POST	Menambahkan produk baru dan invalidasi cache lama	JSON object
/api/products/:id	PUT	Memperbarui data produk berdasarkan ID dan invalidasi cache	JSON object
/api/products/:id	DELETE	Menghapus data produk berdasarkan ID dan	JSON object

		membersihkan cache	
/api/cache/stats	GET	Menampilkan statistik cache adaptif	JSON object
/api/cache/trigger-eval	POST	Memicu evaluasi bulanan secara manual untuk keperluan pengujian tanpa menunggu pergantian bulan nyata	JSON hasil evaluasi
/api/cache/trigger-warmup	POST	Memicu pre-warming seluruh hot key secara manual untuk keperluan verifikasi mekanisme WarmupScheduler	JSON status warmup

2. Penjelasan *Endpoint*

a. GET /api/products

Mengambil daftar seluruh produk menggunakan pola cache-first. Sistem memeriksa ketersediaan data dengan cache key 'products:all'. Jika tersedia (HIT), tiga pilar dicatat secara otomatis: jam akses pertama hari ini,

penambahan hit count, dan pembaruan LastAccessEver untuk memperpanjang D_key. Jika tidak tersedia (MISS), sistem query database dan menyimpan hasilnya ke cache dengan TTL awal.

b. GET /api/products/:id

Endpoint ini mengambil satu produk berdasarkan ID tertentu dengan cache key 'products:id:{id}'. Mekanisme cache-first dan pencatatan tiga pilar berlaku sama seperti GET /api/products. Jika produk tidak ditemukan di database, sistem mengembalikan HTTP 404 dan tidak membuat entry cache untuk ID yang tidak valid.

c. POST /api/products

Menambahkan satu produk ke database dengan validasi input (nama minimal 2 karakter, harga harus lebih dari 0). Setelah penyimpanan ke database berhasil, cache key 'products:all' diinvalidasi agar daftar produk yang tersimpan di cache tidak menjadi stale.

d. PUT /api/products

Memperbarui data produk berdasarkan ID. Sistem terlebih dahulu memverifikasi keberadaan produk menggunakan ProductExists() sebelum melakukan update. Setelah update berhasil, dua cache key diinvalidasi sekaligus: 'products:all' dan 'products:id:{id}'.

e. DELETE /api/products

Menghapus produk berdasarkan ID setelah memverifikasi keberadaannya. Setelah penghapusan berhasil, dua cache key yang sama dengan operasi PUT diinvalidasi, memastikan produk yang sudah dihapus tidak lagi muncul dari cache maupun database.

f. GET /api/products/stats

Menampilkan statistik lengkap sistem cache adaptif mencakup jumlah item aktif di cache, total hit dan miss, hit ratio, profil hot key untuk bulan berikutnya (mencakup AvgFirstAccess, HitCount, dan SuggestedTTL per key), riwayat evaluasi bulanan hingga 12 bulan terakhir, serta statistik per key yang sedang aktif.

g. POST /api/cache/trigger-eval

Memicu evaluasi bulanan secara manual untuk keperluan pengujian. Endpoint ini memanggil runMonthlyEvaluation() secara langsung tanpa menunggu pergantian bulan nyata, sehingga siklus evaluasi tiga pilar dapat diverifikasi dalam durasi penelitian yang terbatas. Endpoint ini tidak dimaksudkan untuk digunakan dalam lingkungan produksi.

h. POST /api/cache/trigger-warmup

Memicu pre-warming seluruh hot key secara manual untuk keperluan verifikasi mekanisme WarmupScheduler. Endpoint ini me-reset flag lastWarmDate pada semua hot key sehingga checkAndWarm() dapat dipanggil ulang tanpa menunggu jam pre-warming tiba. Endpoint ini tidak dimaksudkan untuk digunakan dalam lingkungan produksi.

BAB IV

HASIL DAN PEMBAHASAN

4.1 Implementasi Mekanisme Auto-Cache Adaptif

Implementasi sistem menjelaskan bagaimana merealisasikan rancangan yang telah disusun menjadi sistem yang berjalan. Impelementasi mencakup perhitungan TTL adaptif, pembersihan cache otomatis, invalidasi cache akibat perubahan data, dan siklus evaluasi bulanan.

4.1.1 Implementasi Mekanisme Pencatatan Akses Key

Agar sistem dapat mengadaptasi TTL berdasarkan durasi penggunaan, sistem perlu mengetahui kapan suatu cache key pertama kali diakses dan kapan terakhir kali diakses, Informasi ini dikumpulkan secara otomatis setiap kali terjadi operasi cache GET maupun SET.

1. Struktur Penyimpanan Statistik

Statitik akses setiap cache key yang disimpan dalam struct keyMonthlyStats yang hidup di memori selama satu siklus evaluasi.

```
type dailyRecord struct {
    Date      string
    FirstTime time.Time
    LastTime  time.Time
    Hits      int64
}
type keyMonthlyStats struct {
    Key             string
    Month           int
    Year            int
    FirstAccessEver time.Time
    LastAccessEver  time.Time
    TotalHits       int64
    DailyRecords    map[string]*dailyRecord
}
```

Gambar 4.1 Struct Penyimpanan Statistik Akses Cache Key

Tabel 4.1 Variabel Pencatatan Statistik dan Fungsinya dalam Sistem

Variabel	Tipe Data	Fungsi dalam Sistem
FirstAccessEver	time.Time	Waktu pertama key diakses dalam satu siklus bulan menjadi titik awal (t_0) perhitungan durasi
LastAccessEver	time.Time	Waktu terakhir key diakses diperbarui setiap cache HIT
TotalHits	int64	Penghitung total hit selama satu bulan digunakan untuk menentukan apakah key termasuk hot key
DailyRecords	map[string]*dailyRecord	Rekaman aktivitas per hari mencatat jam akses pertama harian untuk jadwal pre-warming

2. Proses Pencatatan pada Cache SET

Ketika data pertama kali disimpan ke cache melalui fungsi Set(), sistem menetapkan nilai FirstAccessEver sebagai titik awal durasi. Nilai ini hanya ditetapkan sekali jika entry sudah ada, nilai FirstAccessEver tidak diubah agar perhitungan durasi tetap akurat.

```
func (c *AdaptiveCache) Set(key string, value interface{}) {
    c.mu.Lock()
    defer c.mu.Unlock()
    now := time.Now()
    today := now.Format("2006-01-02")
    if existing, ok := c.items[key]; ok {
        existing.Value = value
        newTTL := c.computeKeyTTL(key, existing.stats)
        existing.ExpireAt = now.Add(newTTL)
```

```
        log.Printf("[Cache SET-UPDATE] key=%-25q | TTL=%v", key,
newTTL.Round(time.Second))
        return
    }
    stats := &keyMonthlyStats{
        Key:             key,
        Month:           c.currentMonth,
        Year:            c.currentYear,
        FirstAccessEver: now,
        LastAccessEver:  now,
        TotalHits:       0,
        DailyRecords: map[string]*dailyRecord{
            today: {
                Date:      today,
                FirstTime: now,
                LastTime:  now,
                Hits:      0,
            },
        },
    }
    ttl := c.TTLBaseline
    if profile, isHot := c.hotKeyProfiles[key]; isHot {
        if profile.SuggestedTTL > ttl {
            ttl = profile.SuggestedTTL
        }
    }
    c.items[key] = &CacheEntry{
        Value:    value,
        ExpireAt: now.Add(ttl),
        stats:    stats,
    }
    log.Printf("[Cache SET-NEW]    key=%-25q | TTL_init=%v |
expire=%s",
        key, ttl.Round(time.Second),
now.Add(ttl).Format("15:04:05"))
}
```

Gambar 4.2 Penetapan FirstAccessEver saat Entry Cache Dibuat

3. Proses Pencatatan pada Cache GET

Setiap cache HIT memperbarui nilai LastAccessEver ke waktu saat ini. Sehingga durasi aktif D_key = LastAccessEver – FirstAccessEver bertambah secara akumulatif seiring berjalannya waktu operasional.

```go
func (c *AdaptiveCache) Get(key string) (interface{}, bool) {
    c.mu.Lock()
    defer c.mu.Unlock()
    entry, exists := c.items[key]
    if !exists {
        c.TotalMisses++
        return nil, false
    }
    now := time.Now()
    if now.After(entry.ExpireAt) {
        delete(c.items, key)
        c.CleanupCount++
        c.TotalMisses++
        return nil, false
    }
    c.TotalHits++
    stats := entry.stats
    stats.TotalHits++
    today := now.Format("2006-01-02")
    if _, exists := stats.DailyRecords[today]; !exists {
        stats.DailyRecords[today] = &dailyRecord{
            Date:      today,
            FirstTime: now,
            LastTime:  now,
            Hits:      1,
        }
        log.Printf("[Cache HIT][Pilar 1] key=%-25q | jam akses
pertama hari ini: %s",
            key, now.Format("15:04:05"))
    } else {
        stats.DailyRecords[today].LastTime = now
        stats.DailyRecords[today].Hits++
    }
    stats.LastAccessEver = now
    newTTL := c.computeKeyTTL(key, stats)
    entry.ExpireAt = now.Add(newTTL)
    D_key := stats.LastAccessEver.Sub(stats.FirstAccessEver)
    log.Printf("[Cache HIT]  key=%-25q | hits=%d | D_key=%6.1fs |
TTL_runtime=%-10v | expire=%s",key,
        stats.TotalHits,
        D_key.Seconds(),
        newTTL.Round(time.Second),
        entry.ExpireAt.Format("15:04:05"),
    )
    return entry.Value, true
}
```

Gambar 4.3 Pembaruan LastAccessEver pada setiap Cache HIT

Dengan kombinasi kedua fungsi tersebut, sistem selalu memiliki data durasi aktif yang akurat untuk setiap cache key. Data inilah yang digunakan sebagai input perhitungan TTL adaptif pada sub-bab berikutnya.

4.1.2 Implementasi Perhitungan TTL Adaptif

Setelah data durasi akses tersedia melalui mekanisme pencatatan pada sub-bab sebelumnya, sistem menggunakannya untuk menghitung TTL yang proporsional terhadap lama penggunaan cache key tersebut.

1. Parameter Sistem

Tabel 4.2 Parameter Sistem

Parameter	Nilai Default	Keterangan
TTL_baseline	5 menit (300 detik)	Nilai awal TTL saat entry pertama kali dibuat
TTL_max	4 jam (14.400 detik)	Batas max TTL tidak dapat dilampaui
AdaptCoeff	0,002	Koefisien adaptasi: pertambahan TTL per detik durasi aktif
CleanupInterval	1 menit	Interval goroutine AutoCleanup memeriksa dan menghapus entry kadaluarsa
WarmupCheckInterval	1 menit	Interval goroutine WarmupScheduler memeriksa jadwal pre-warming hot key

2. Implementasi Fungsi computeKeyTTL()

```
3.  func (c *AdaptiveCache) computeKeyTTL(
4.  key string, stats*keyMonthlyStats) time.Duration {
5.      baseline := c.TTLBaseline
6.      if profile, isHot := c.hotKeyProfiles[key]; isHot {
7.          if profile.SuggestedTTL > baseline {
8.              baseline = profile.SuggestedTTL
9.              log.Printf("[Cache TTL] key=%q menggunakan
    SuggestedTTL dari profil bulan lalu: %v",
10.                 key, baseline.Round(time.Second))
11.         }
12.     }
13.     var D_key time.Duration
14.     if !stats.FirstAccessEver.IsZero() &&
    !stats.LastAccessEver.IsZero() {
15.         D_key = stats.LastAccessEver.Sub(stats.FirstAccessEver)
16.     }
17.     adaptedSec := baseline.Seconds() +
    c.AdaptCoeff*D_key.Seconds()
18.     runtime := time.Duration(adaptedSec * float64(time.Second))
19.     if runtime > c.TTLMax {
20.         runtime = c.TTLMax
21.     }
22.     return runtime
23. }
```

Gambar 4.4 Implementasi Formula TTL Adaptif

Contoh hasil perhitungan: cache key 'products:all' yang baru pertama kali diakses mendapat TTL_runtime = 300 detik (sama dengan baseline). Setelah digunakan selama 1 jam (D_key = 3.600 detik), TTL_runtime meningkat menjadi 300 + 0,002 × 3.600 = 307,2 detik. Setelah 8 jam aktif, TTL menjadi 357,6 detik, meningkat otomatis tanpa konfigurasi manual.

Tabel 4.3 Perkembangan TTL Runtime Berdasarkan Durasi Aktif D_key

D_key (durasi aktif)	**TTL_baseline**	**TTL_runtime**	**Kenaikan dari Baseline**
0 detik	300 detik	300 detik	0%
3.600 detik	300 detik	307,2 detik	+2,4%
28.800 detik	300 detik	357,6 detik	+19,2%
50.400 detik	300 detik	400,8 detik	+33,6%
7.048.800 detik	300 detik	14.400 detik	Batas TTL_max tercapai

4.1.3 Implementasi Mekanisme Auto-Cleanup

Untuk memastikan data kedaluwarsa tidak terus menempati memori, sistem menjalankan goroutine AutoCleanup yang bekerja secara periodik di latar belakang tanpa mengganggu proses request handling.

1. Mekanisme Auto-Cleanup

Goroutine ini merupakan time.Ticker dengan interval 1 menit, Setiap sinyal ticker diterima, sistem melakukan iterasi terhadap seluruth entry cache dengan menghapus entry yang nilai ExpireAt nya telah dilewati waktu saat ini.

```
func (c *AdaptiveCache) AutoCleanup() {
    ticker := time.NewTicker(CleanupInterval)
    defer ticker.Stop()
    log.Println("[Cache] AutoCleanup goroutine aktif.")
    for range ticker.C {
        c.runCleanup()
    }
}

func (c *AdaptiveCache) runCleanup() {
    c.mu.Lock()
    defer c.mu.Unlock()
    now := time.Now()
    n := 0
    for k, entry := range c.items {
        if now.After(entry.ExpireAt) {
            delete(c.items, k)
            c.CleanupCount++
            n++
        }
    }
    if n > 0 {
        log.Printf("[Cache Cleanup] %d entry kadaluarsa dihapus.",
n)
    }
}
```
Gambar 4.5 Implementasi Goroutine AutoCleanup dan Fungsi tunCleanup()

Tabel 4.4 Rincian Mekanisme AutoCleanup

Aspek	Penjelasan
Interval pengecekan	1 menit, entry kadaluarsa dihapus paling lambat 1 menit setelah TTL habis
Kondisi penghapusan	t_now ≥ t_expire: waktu saat ini sudah melewati batas waktu kedaluwarsa entry
Keamanan konkurensi	Mutex c.mu.Lock() memastikan tidak ada race condition saat goroutine lain mengakses cache
Pencatatan hasil	CleanupCount terakumulasi dan dapat dipantau melalui endpoint GET /api/cache/stats

Goroutine AutoCleanup memenuhi prinsip desain yang ditetapkan pada proposal: sistem tidak bergantung pada frekuensi akses untuk menentukan kapan entry harus dihapus, melainkan menggunakan waktu kedaluwarsa yang telah ditetapkan secara deterministik.

4.1.4 Implementasi Mekanisme Cache Invalidation

Cache invalidation adalah mekanisme yang memastikan bahwa ketika data di database berubah akibat operasi tulis (POST, PUT, atau DELETE), salinan data yang tersimpan di cache langsung dihapus. Tanpa invalidasi, cache akan terus menyajikan data lama meskipun database sudah diperbarui, sehingga respons API menjadi tidak konsisten.

1. Fungsi Penghapusan Cache (Delete dan DeleteByPrefix)

Sistem menyediakan dua fungsi penghapusan cache, Fungsi Delete menghapus satu key secara eksplisit berdasarkan nama kunci yang diberikan.

Fungsi DeleteByPrefix menghapus seluruh key yang diawali oleh prefix tertentu yang berguna unutk membersihkan kelompok cache sekaligus.

```
func (c *AdaptiveCache) Delete(key string) {
    c.mu.Lock()
    defer c.mu.Unlock()
    if _, ok := c.items[key]; ok {
        delete(c.items, key)
        log.Printf("[Cache INVALIDATE] key=%q dihapus (CRUD)", key)
    }
}

func (c *AdaptiveCache) DeleteByPrefix(prefix string) {
    c.mu.Lock()
    defer c.mu.Unlock()
    n := 0
    for k := range c.items {
        if strings.HasPrefix(k, prefix) {
            delete(c.items, k)
            n++
        }
    }
    if n > 0 {
        log.Printf("[Cache INVALIDATE-PREFIX] prefix=%q → %d item
dihapus", prefix, n)
    }
}
```

Gambar 4.6 Fungsi Penghapusan Cache

Fungsi Delete menggunakan write lock (mu.lock) untuk memastikan penghapusan bersifat atomic dna tidak terjadi race condition dengan goroutine lain yang sedang membaca atau menulis cache secara bersamaaan.

2. Invalidasi pada Handler POST

Setiap kali produk baru berhasil ditambahkan ke database, cache key products:all dihapus agar request GET all berikutnya mengambil daftar terbaru dari database dan bukan daftar lama yang belum mencakup produk baru tersebut.

```
func (h *ProductHandler) Create(c *gin.Context) {
    var req models.CreateProductRequest
    if err := c.ShouldBindJSON(&req); err != nil {
```

```
        c.JSON(http.StatusBadRequest, gin.H{
            "error":   "Input tidak valid",
            "details": err.Error(),
        })
        return
    }
    product := &models.Product{Name: req.Name, Price: req.Price}
    if err := h.db.CreateProduct(product); err != nil {
        return
    }
h.cache.Delete(cacheKeyAllProducts)
    c.JSON(http.StatusCreated, gin.H{
        "message": "Produk berhasil ditambahkan",
        "data":    product,
    })
}
```

Gambar 4.7 Fungsi Invalidasi pada Handler POST

3. Invalidasi Dua Cache sekaligus pada Handler PUT

Operasi update menginvalidasi uda cache key sekaligus products:all karena data yang berubah, dan products:id:{id} karena data individual produk yang bersangkutan juga berubah. penghapusan dua key dilakukan secara berurutan setelah pembaruan database berhasil.

```
func (h *ProductHandler) Update(c *gin.Context) {
    id, err := parseID(c)
    if err != nil {
        return
    }
    var req models.UpdateProductRequest
    if err := c.ShouldBindJSON(&req); err != nil {
        return
    } if err := h.db.UpdateProduct(id, req.Name, req.Price); err !=
nil {
        return
    }
    h.cache.Delete(cacheKeyAllProducts)
    h.cache.Delete(fmt.Sprintf(cacheKeyProductByID, id))
    c.JSON(http.StatusOK, gin.H{
        "message": "Produk berhasil diperbarui",
    })
}
```

Gambar 4.8 Fungsi Invalidasi pada Handler PUT

4. Invalidasi pada Handler DELETE

Sama seperti operasi PUT, Penghapusan produk dari database juga menginvalidasi dua cache key secara bersamaan. Hal ini memastikan bahwa produk yang sudah dihapus tidak lagi muncul dalam cache daftar produk maupun cache individual, sehingga request GET berikutnya akan menerima HTTP 404 dari database dan bukan data stale dari cache.

```
func (h *ProductHandler) Delete(c *gin.Context) {
    id, err := parseID(c)
    if err != nil {
        return
    }

    if !h.db.ProductExists(id) {
        c.JSON(http.StatusNotFound, gin.H{
            "error": fmt.Sprintf("Produk dengan ID %d tidak
ditemukan", id),
        })
        return
    }

    if err := h.db.DeleteProduct(id); err != nil {
        return
    }

    h.cache.Delete(cacheKeyAllProducts)
    h.cache.Delete(fmt.Sprintf(cacheKeyProductByID, id))

    c.JSON(http.StatusOK, gin.H{
        "message": fmt.Sprintf("Produk ID %d berhasil dihapus", id),
        "id":      id,
    })
}
```

Gambar 4.9 Fungsi Invalidasi pada Handler DELETE

Dengan pola invalidasi ini, konsistensi data antar database dan cache selalu terjaga. Setiap operasi tulis yang berhasil selalu diikuti penghapusan cache yang terdampak, sehingga request GET berikutnya dijamin mengambil data terbaru dari database.

4.1.5 Implementasi Siklus Evaluasi

Siklus evaluasi 30 hari merupakan komponen yang menunjukkan sifat adaptif jangka panjang sistem. Selama satu periode, sistem mengakumulasikan statistik akses seluruh cache key; pada akhir periode, statistik tersebut digunakan untuk memperbarui parameter TTL_baseline sehingga sistem dapat beroperasi lebih efisien pada siklus berikutnya.

1. Proses Pengumpulan Statitik Bulanan

Selama periode 30 hari, goroutine MonthlyEvaluationCycle berjalan setiap 1 jam untuk memeriksa apakah bulan telah berganti. Selama periode berjalan, setiap cache HIT secara otomatis memperbarui statistik melalui fungsi Get().

```
func (c *AdaptiveCache) MonthlyEvaluationCycle() {
    ticker := time.NewTicker(1 * time.Hour)
    defer ticker.Stop()
    log.Println("[Cache] MonthlyEvaluationCycle goroutine aktif (cek
tiap 1 jam).")
    for range ticker.C {
        now := time.Now()
        c.mu.RLock()
        changed := int(now.Month()) != c.currentMonth || now.Year()
!= c.currentYear
        c.mu.RUnlock()
        if changed {
            c.runMonthlyEvaluation(now)
        }
    }
}
```

Gambar 4.10 Goroutine MonthlyEvaluationCycle

Data yang diakumulasikan selama periode berjalan meliputi, total hit per key (TotalHits), waktu akses pertama per key (FirstAccessEver), waktu akses terakhir (LastAccessEver), dan jam akses pertama harian (DailyRecords). Keempat jenis data ini menjadi input bagi proses evaluasi.

2. Proses Evaluasi Data Akses

Pada saat evaluasi dijalankan, sistem mengidentifikasi cache key mana yang termasuk kategori 'hot key', yaitu key yang memiliki riwayat akses nyata (TotalHits > 0) selama periode berjalan. Untuk setiap hot key, sistem menghitung SuggestedTTL berdasarkan durasi aktif yang tercatat.

```
func (c *AdaptiveCache) runMonthlyEvaluation(now time.Time) {
    c.mu.Lock()
    defer c.mu.Unlock()
    prevMonth := c.currentMonth
    prevYear := c.currentYear
    log.Printf("[Cache Eval] == EVALUASI BULANAN: %s → %s ==",
        monthLabel(prevYear, prevMonth),
        monthLabel(now.Year(), int(now.Month())),
    )
    allStats := []*keyMonthlyStats{}
    for _, entry := range c.items {
        if entry.stats != nil && entry.stats.Month == prevMonth &&
entry.stats.Year == prevYear {
            allStats = append(allStats, entry.stats)
        }
    }
    newProfiles := make(map[string]*HotKeyProfile)
    snapshots := []HotKeyProfile{}
    for _, stats := range allStats {
        if stats.TotalHits == 0 {
            log.Printf("[Cache Eval] key=%q dilewati (tidak ada hit
bulan ini)", stats.Key)
            continue
        }
        avgHour, avgMinute :=
computeAvgFirstAccess(stats.DailyRecords)
        activeDuration :=
stats.LastAccessEver.Sub(stats.FirstAccessEver)
        if activeDuration < c.TTLBaseline {
            activeDuration = c.TTLBaseline
        }
        if activeDuration > c.TTLMax {
            activeDuration = c.TTLMax
        }
        profile := &HotKeyProfile{
            Key:               stats.Key,
            HitCountLastMonth: stats.TotalHits,
```

```
            AvgFirstHour:      avgHour,
            AvgFirstMinute:    avgMinute,
            SuggestedTTL:      activeDuration,
            ActiveDurationSec: activeDuration.Seconds(),
            EvalMonth:         monthLabel(prevYear, prevMonth),
        }
        newProfiles[stats.Key] = profile
        snapshots = append(snapshots, *profile)
        log.Printf("[Cache Eval] HOT KEY: key=%-25q | hits=%d | pre-
warm=%02d:%02d | TTL=%v",
            stats.Key,
            stats.TotalHits,
            avgHour, avgMinute,
            activeDuration.Round(time.Second),
        )
    }
```

Gambar 4.11 Fungsi runMonthlyEvaluation()

3. Proses Pembaruan Parameter Sistem

Setelah evaluasi menghasilkan daftar hot key dengan SuggestedTTL masing-masing, sistem memperbarui dua hal: profil hot key yang digunakan oleh computeKeyTTL() pada siklus berikutnya, dan jadwal pre-warming untuk masing-masing key. Metadata siklus sebelumnya kemudian direset untuk memulai akumulasi baru.

```
c.evalHistory = append(c.evalHistory, MonthlyEvalSnapshot{
        Month:         monthLabel(prevYear, prevMonth),
        HotKeysFound: len(newProfiles),
        Profiles:      snapshots,
        EvaluatedAt:   now,
    })
    if len(c.evalHistory) > 12 {
        c.evalHistory = c.evalHistory[len(c.evalHistory)-12:]}
    c.hotKeyProfiles = newProfiles
    c.currentMonth = int(now.Month())
    c.currentYear = now.Year()
    log.Printf("[Cache Eval] %d hot key terdaftar untuk bulan %s.",
        len(newProfiles), monthLabel(c.currentYear, c.currentMonth))
    log.Printf("[Cache Eval] Pre-warming terjadwal sesuai
AvgFirstAccess masing-masing key.")
}
```

Gambar 4.12 Pembaruan HotKeyProfiles dan Reset Statistik Baru

Tabel 4.5 Tiga Tahap Siklus Evaluasi 30 Hari

Tahap	Proses	Output
Pengumpulan	Pencatatan FirstAccessEver, LastAccessEver, TotalHits pada setiap cache HIT	Statistik akumulatif selama 30 hari per cache key
Evaluasi	Identifikasi hot key (TotalHits > 0), hitung D_key dan SuggestedTTL	Daftar HotKeyProfile beserta SuggestedTTL dan jadwal pre-warming
Pembaruan	Simpan HotKeyProfiles, reset accessStats, perbarui currentMonth	Siklus baru dimulai dengan parameter TTL yang telah dikalibrasi

Pada siklus berikutnya, fungsi computeKeyTTL() akan menggunakan SuggestedTTL dari hot key profile sebagai baseline pengganti TTL_baseline default sehingga cache key yang terbukti sering digunakan mendapat TTL awal yang lebih panjang sejak pertama kali diakses.

4.2 Implementasi Basis Data

Basis data pada sistem ini diimplementasikan menggunakan SQLite melalui library modernc.org/sqlite yang bersifat pure Go sehingga tidak memerlukan CGO atau compiler eksternal. Seluruh logika akses data dienkapsulasi dalam paket db melalui file database.go, yang menyediakan fungsi inisialisasi koneksi dan operasi CRUD terhadap entitas produk. Implementasi ini berhubungan langsung dengan mekanisme cache pada sub-bab sebelumnya: setiap cache MISS memicu query ke database, dan setiap operasi tulis (POST/PUT/DELETE) memicu invalidasi cache.

1. Inisialisasi Koneksi dan Skema Database

Fungsi InitDB() membuka koneksi ke file SQLite, memverifikasi koneksi melalui Ping(), mengaktifkan dua PRAGMA penting, kemudian membuat tabel products apabila belum ada. PRAGMA journal_mode=WAL dipilih karena mendukung pembacaan paralel tanpa memblokir proses penulisan, sesuai dengan kebutuhan sistem yang menangani banyak request bersamaan. PRAGMA foreign_keys=ON mengaktifkan penegakan integritas referensial pada level engine.

```
func InitDB(path string) (*DB, error) {
    db, err := sqlx.Open("sqlite", path)
    if err != nil {
        return nil, err
    }

    if err := db.Ping(); err != nil {
        return nil, err
    }

    db.Exec("PRAGMA journal_mode=WAL;")
    db.Exec("PRAGMA foreign_keys=ON;")

    _, err = db.Exec(`
        CREATE TABLE IF NOT EXISTS products (
            id    INTEGER PRIMARY KEY AUTOINCREMENT,
            name  TEXT    NOT NULL,
            price REAL    NOT NULL CHECK(price > 0)
        )
    `)
    if err != nil {
        return nil, err
    }

    wrapped := &DB{db}

    var count int
    db.QueryRow("SELECT COUNT(*) FROM products").Scan(&count)
    if count == 0 {
        log.Println("[DB] Menyemai data produk awal...")
        seeds := []models.Product{
            {Name: "Laptop Lenovo IdeaPad", Price: 8_500_000},
```

```
            {Name: "Mouse Wireless Logitech MX Master", Price:
850_000},
            {Name: "Keyboard Mechanical Keychron K2", Price:
1_200_000},
            {Name: "Monitor 24\" Full HD Dell", Price: 2_750_000},
            {Name: "USB Hub 7-Port Anker", Price: 285_000},
            {Name: "Webcam Logitech C920 HD", Price: 1_100_000},
            {Name: "SSD External Samsung 1TB", Price: 1_650_000},
            {Name: "Headset Gaming HyperX Cloud", Price: 950_000},
        }
        for _, p := range seeds {
            wrapped.CreateProduct(&p)
        }
        log.Printf("[DB] %d produk berhasil disemai.", len(seeds))
    }
    log.Printf("[DB] Koneksi ke %s berhasil.", path)
    return wrapped, nil
}
```

Gambar 4.13 Fungsi InitDB()

2. Struktur Model Data

Model data didefinisikan dalam paket models menggunakan struct Go dengan tag db untuk pemetaan kolom database dan tag json untuk serialisasi respons API. Terdapat tiga struct utama.

```
type Product struct {
    ID    int64   `db:"id"    json:"id"`
    Name  string  `db:"name"  json:"name"`
    Price float64 `db:"price" json:"price"`
}

type CreateProductRequest struct {
    Name  string  `json:"name"  binding:"required,min=2"`
    Price float64 `json:"price" binding:"required,gt=0"`
}

type UpdateProductRequest struct {
    Name  string  `json:"name"  binding:"required,min=2"`
    Price float64 `json:"price" binding:"required,gt=0"`
}
```

Gambar 4.14 Struct Product dan CreateProductRequest

Tabel 4.6 Strust Utama dalam Paket models

Struct	Digunakan Pada	Keterangan
Product	Seluruh operasi CRUD	Representasi entitas produk di database; tag db memetakan field ke kolom, tag json mengatur serialisasi respons API
CreateProductRequest	POST /api/products	Validasi input saat membuat produk baru: Name wajib diisi dan minimal 2 karakter, Price wajib diisi dan harus lebih dari 0
UpdateProductRequest	PUT /api/products/:id	Aturan validasi identik dengan CreateProductRequest; pemisahan struct memungkinkan fleksibilitas pengembangan di masa mendatang

3. Implementasi Fungsi CRUD Database

Paket db menyediakan lima fungsi operasi data yang dipanggil oleh lapisan handler. Seluruh fungsi menggunakan parameterized query untuk mencegah SQL injection. Berikut adalah implementasi seluruh fungsi CRUD tersebut.

```
func (d *DB) GetAllProducts() ([]models.Product, error)
      {
    var products []models.Product
    err := d.Select(&products, "SELECT id, name, price FROM products
ORDER BY id ASC")
    return products, err
}
func (d *DB) GetProductByID(id int64) (*models.Product, error)
      {
    var product models.Product
```

```
    err := d.Get(&product, "SELECT id, name, price FROM products
WHERE id = ?", id)
    if err != nil
      {
        return nil, err
    }
    return &product, nil
}
func (d *DB) CreateProduct(p *models.Product) error
      {
    result, err := d.Exec(
        "INSERT INTO products (name, price) VALUES (?, ?)",
        p.Name, p.Price,
    )
    if err != nil {
        return err
    }

    p.ID, err = result.LastInsertId()
    return err
}

func (d *DB) UpdateProduct(id int64, name string, price float64)
error
      {
    _, err := d.Exec(
        "UPDATE products SET name = ?, price = ? WHERE id = ?",
        name, price, id,
    )
    return err
}

func (d *DB) DeleteProduct(id int64) error
      {
    _, err := d.Exec("DELETE FROM products WHERE id = ?", id)
    return err
}

func (d *DB) ProductExists(id int64) bool
      {
    var count int
    d.QueryRow("SELECT COUNT(*) FROM products WHERE id = ?",
id).Scan(&count)
    return count > 0
}
```

Gambar 4.15 Implementasi Fungsi CRUD

Tabel 4.7 Ringkasan Fungsi CRUD

Fungsi	**Operasi SQL**	**Keterangan dan Hubungan dengan Cache**
GetAllProducts()	SELECT	Sumber data utama untuk cache key products:all; hasil query disimpan ke cache setelah cache MISS
GetProductByID()	SELECT	Mengembalikan nil jika produk tidak ditemukan; memungkinkan handler merespons dengan HTTP 404
CreateProduct()	INSERT	ID produk baru diambil via LastInsertId() dan dikembalikan ke handler; memicu invalidasi cache products:all
UpdateProduct()	UPDATE	Memperbarui kolom name dan price; memicu invalidasi cache products:all dan products:id:{id}
DeleteProduct()	DELETE	Setelah berhasil, cache products:all dan products:id:{id} diinvalidasi; GET by ID berikutnya mengembalikan 404
ProductExists()	SELECT COUNT(*)	Memeriksa keberadaan produk berdasarkan ID sebelum operasi PUT dan DELETE dijalankan

4.3 Implementasi Endpoint API

Sistem menyediakan delapan endpoint yang dibagi menjadi dua kelompok: endpoint CRUD untuk pengelolaan data produk dan endpoint manajemen cache. Seluruh endpoint menggunakan format JSON untuk request dan response.

Tabel 4.8 Daftar Endpoint API

No	Method	Endpoint	Fungsi
1	GET	/api/products	Mengambil seluruh daftar produk (cache-first)
2	GET	/api/products/:id	Mengambil detail satu produk berdasarkan ID
3	POST	/api/products	Menambahkan produk baru + invalidasi cache
4	PUT	/api/products/:id	Memperbarui data produk + invalidasi cache
5	DELETE	/api/products/:id	Menghapus produk + invalidasi cache
6	GET	/api/cache/stats	Menampilkan statistik cache adaptif lengkap
7	POST	/api/cache/trigger-eval	Memicu evaluasi bulanan secara manual (pengujian)
8	POST	/api/cache/trigger-warmup	Memicu pre-warming hot key secara manual

Sebagai contoh implementasi, berikut adalah handler untuk endpoint GET /api/products yang menerapkan pola cache-first.

```go
func (h *ProductHandler) GetAll(c *gin.Context) {
    if cached, ok := h.cache.Get(cacheKeyAllProducts); ok {
        c.JSON(http.StatusOK, gin.H{
            "source":  "cache",
            "message": "Data diambil dari cache adaptif",
            "data":    cached,
        })
        return
    }
    products, err := h.db.GetAllProducts()
    if err != nil {
        c.JSON(http.StatusInternalServerError, gin.H{"error": "Gagal
mengambil data produk"})
        return
    }
    h.cache.Set(cacheKeyAllProducts, products)
    c.JSON(http.StatusOK, gin.H{
        "source":  "database",
        "message": "Data diambil dari database dan disimpan ke
cache",
        "data":    products,
    })
}

func (h *ProductHandler) Create(c *gin.Context) {
    var req models.CreateProductRequest
    if err := c.ShouldBindJSON(&req); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{
            "error":   "Input tidak valid",
            "details": err.Error(),
        })
        return
    }
    product := &models.Product{Name: req.Name, Price: req.Price}
    if err := h.db.CreateProduct(product); err != nil {
        c.JSON(http.StatusInternalServerError, gin.H{"error": "Gagal
menyimpan produk baru"})
        return
    }
    h.cache.Delete(cacheKeyAllProducts)
    c.JSON(http.StatusCreated, gin.H{
        "message": "Produk berhasil ditambahkan",
        "data":    product,
    })
}
```

```
func (h *ProductHandler) Update(c *gin.Context) {
    id, err := parseID(c)
    if err != nil {
        return
    }
    if !h.db.ProductExists(id) {
        c.JSON(http.StatusNotFound, gin.H{
            "error": fmt.Sprintf("Produk dengan ID %d tidak
ditemukan", id),
        })
        return
    }
    var req models.UpdateProductRequest
    if err := c.ShouldBindJSON(&req); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{
            "error":   "Input tidak valid",
            "details": err.Error(),
        })
        return
    }
    if err := h.db.UpdateProduct(id, req.Name, req.Price); err !=
nil {
        c.JSON(http.StatusInternalServerError, gin.H{"error": "Gagal
memperbarui produk"})
        return
    }
    h.cache.Delete(cacheKeyAllProducts)
    h.cache.Delete(fmt.Sprintf(cacheKeyProductByID, id))
    c.JSON(http.StatusOK, gin.H{
        "message": "Produk berhasil diperbarui",
        "data":    models.Product{ID: id, Name: req.Name, Price:
req.Price},
    })
}
func (h *ProductHandler) Delete(c *gin.Context) {
    id, err := parseID(c)
    if err != nil {
        return
    }
    if !h.db.ProductExists(id) {
        c.JSON(http.StatusNotFound, gin.H{
            "error": fmt.Sprintf("Produk dengan ID %d tidak
ditemukan", id),
        })
        return
    }
    if err := h.db.DeleteProduct(id); err != nil {
```

```
        c.JSON(http.StatusInternalServerError, gin.H{"error": "Gagal 
menghapus produk"})
        return
    }
    h.cache.Delete(cacheKeyAllProducts)
    h.cache.Delete(fmt.Sprintf(cacheKeyProductByID, id))
    c.JSON(http.StatusOK, gin.H{
        "message": fmt.Sprintf("Produk ID %d berhasil dihapus", id),
        "id":      id,
    })
}
```

Gambar 4.16 Handler GetAll()

Field 'source' pada respons JSON berfungsi sebagai indikator diagnostik yang memudahkan verifikasi selama pengujian: nilai 'cache' menunjukkan data dilayani dari in-memory cache tanpa menyentuh database, sedangkan nilai 'database' menunjukkan terjadi cache MISS.

4.4 Pengujian Sistem

Pengujian sistem dilakukan untuk memverifikasi bahwa seluruh mekanisme yang diimplementasikan berjalan sesuai rancangan, sekaligus mengukur dampak nyata penerapan auto-cache adaptif terhadap performa sistem RESTful API.

4.4.1 Pengujian Fungsional

Pengujian fungsional dilakukan untuk memverifikasi bahwa setiap mekanisme yang diimplementasikan berperilaku sesuai rancangan. Seluruh pengujian dijalankan secara langsung menggunakan Postman dan terminal curl pada lingkungan local localhost:8081, dengan hasil respons JSON dan log terminal server sebagai bukti keberhasilan.

1. Uji Cache

Pengujian ini memverifikasi siklus lengkap mekanisme cache, ketika request pertama menghasilkan cache miss dan request berikutnya dengan endpoint yang sama menghasilkan cache HIT.

a. Cache MISS

Request pertama dikirimkan ke endpoint GET /api/products saat cache masih kosong. Sistem tidak menemukan data di cache sehingga mengeksekusi query di database, menyimpan hasilnya ke cache, lalu mengembalikan respons dengan field source: ‘database’.

```
GET http://localhost:8081/api/products
// Respons JSON yang diterima:
{
   "message": "Data diambil dari database dan disimpan ke cache",
   "source": "database"
}
```

Gambar 4.17 Respon Cache MISS

Field source: 'database' mengkonfirmasi bahwa data diambil dari SQLite dan di-cache-kan untuk request selanjutnya. Pada terminal server, log [Cache SET-NEW] muncul yang menandakan entry cache baru dibuat dengan TTL_init = 5 menit.

b. Cache HIT

Request kedua dikirimkan ke endpoint yang sama. Karena data sudah tersimpan di cache, sistem mengembalikan respons langsung dari memori tanpa query database. Ketiga pilar dicatat otomatis oleh fungsi Get(): DailyRecords.FirstTime, TotalHits++, LastAccessEver diperbarui.

```
GET http://localhost:8081/api/products
// Respons JSON yang diterima:
{
   "message": "Data diambil dari cache adaptif",
   "source": "cache"
}
```

Gambar 4.18 Respon Cache HIT

Perubahan nilai source dari 'database' menjadi 'cache' membuktikan bahwa mekanisme cache-first berjalan dengan benar. Pada terminal server, log

[Cache HIT] muncul beserta informasi hits, D_key (durasi aktif), dan TTL_runtime yang dihitung ulang secara adaptif.

c. Verifikasi Statistik Cache

Endpoint GET /api/cache/stats diakses untuk memverifikasi bahwa statistik sistem tercatat dengan benar setelah dua request di atas. Total hits dan misses harus masing-masing bernilai 1.

```
GET http://localhost:8081/api/cache/stats
// Respons JSON yang relevan:
{
   "cache_stats": {
      "item_count": 1,            // 1 entry aktif di cache
      "total_hits": 1,            // request ke-2 dilayani dari cache
      "total_misses": 1,          // request ke-1 menyentuh database
      "hit_ratio_pct": 50,        // 1 hit dari 2 total request
      "ttl_baseline_seconds": 300, // TTL_baseline = 5 menit
      "ttl_max_seconds": 14400,    // TTL_max = 4 jam
      "adapt_coeff": 0.002,        // koefisien adaptasi TTL
}
```

Gambar 4.19 Output setelah Satu Cache MISS dan Cache HIT

Nilai *item_count: 1* menunjukkan satu entry aktif di cache (key *products:all*). Field *active_duration_seconds: 14,11* merupakan nilai D_key yang digunakan formula TTL adaptif: TTL_runtime = 300 + 0,002 × 14,11 = 300,03 detik masih sangat dekat dengan TTL_baseline karena sistem baru berjalan singkat. Nilai ini akan meningkat seiring bertambahnya durasi operasional.

2. Uji Cache per ID pada GET /api/products/:id

Pengujian ini memverifikasi bahwa mekanisme cache-first juga berjalan pada endpoint yang mengambil satu produk berdasarkan ID. Setiap ID produk memiliki cache key tersendiri dengan format products:id:{id}.

a. Cache MISS

Request pertama ke GET /api/products/1 menghasilkan cache MISS karena cache key products:id:1 belum ada. Data diambil dari database lalu disimpan ke cache.

```
GET http://localhost:8081/api/products/1
{
   "data": {
      "id": 1,
      "name": "Laptop Lenovo UPDATED",
      "price": 9999000
   },
   "message": "Data diambil dari database dan disimpan ke cache",
   "source": "database”
}
```

Gambar 4.20 Respon Cache MISS

b. Cache HIT

Request kedua ke endpoint yang sama mengembalikan data dari cache. Nilai source berubah menjadi 'cache', membuktikan entry products:id:1 berhasil disimpan dan dibaca kembali.

```
GET http://localhost:8081/api/products/1
{
   "data": {
      "id": 1,
      "name": "Laptop Lenovo UPDATED",
      "price": 9999000
   },
   "message": "Data diambil dari cache adaptif",
   "source": "cache"
}
```

Gambar 4.21 Respon Cache HIT

c. Respons Error ID tidak ditemukan

Request ke ID yang tidak ada di database (ID 10) mengembalikan HTTP 404. Sistem tidak membuat cache entry untuk data yang tidak ada, sehingga

setiap request ke ID tidak valid selalu menyentuh database dan dikembalikan sebagai error.

```
GET http://localhost:8081/api/products/10
{
    "error": "Produk dengan ID 10 tidak ditemukan"
}
```

Gambar 4.22 Respon HTTP 404 untuk ID yang tidak ada

3. Uji Tambah Produk (POST /api/products)

Pengujian ini memverifikasi tiga skenario pada endpoint POST, input valid berhasil disimpan, input dengan nama terlalu pendek ditolak, dan input dengan harga negatif ditolak. Validasi dilakukan oleh Gin Framework menggunakan tag binding pada struct CreateProductRequest.

a. Input Produk Valid

Request POST dengan nama minimal 2 karakter dan harga di atas 0 berhasil menyimpan produk baru. Sistem juga secara otomatis menginvalidasi cache products:all agar daftar produk yang tersimpan di cache tidak menjadi stale.

```
POST http://localhost:8081/api/products
Content-Type: application/json

{"name": "Mechanical Keyboard Keychron Q1", "price": 1850000}

// Respons JSON (HTTP 201 Created):
{
   "data": {
      "id": 153,
      "name": "Mechanical Keyboard Keychron Q1",
      "price": 1850000
   },
   "message": "Produk berhasil ditambahkan"
}
```

Gambar 4.23 Respon Sukses Post dengan Input Valid

Produk baru mendapat ID 153 yang di-generate otomatis oleh AUTOINCREMENT SQLite. Log server menampilkan [Cache INVALIDATE] key="products:all" yang menandakan cache daftar produk dihapus agar request GET berikutnya mengambil data terbaru dari database.

b. Validasi Nama

Request POST dengan nama hanya 1 karakter ('A') ditolak oleh sistem. Tag binding:"required,min=2" pada field Name memastikan nama produk minimal terdiri dari 2 karakter sebelum data disimpan ke database.

```
POST http://localhost:8081/api/products
Content-Type: application/json

{"name": "A", "price": 50000}
// Respons JSON (HTTP 400 Bad Request):
{
   "details": "Key: 'CreateProductRequest.Name' Error: Field validation
            for 'Name' failed on the 'min' tag",
   "error": "Input tidak valid"
}
```

Gambar 4.24 Respon HTTP 400 untuk Validasi Nama Gagal

c. Validasi Harga

Request POST dengan harga bernilai negatif (-100) ditolak oleh sistem. Tag binding:"required,gt=0" pada field Price memastikan harga selalu bernilai lebih besar dari nol, selaras dengan constraint CHECK(price > 0) pada DDL tabel SQLite.

```
POST http://localhost:8081/api/products
Content-Type: application/json
{"name": "Produk Test", "price": -100}
// Respons JSON (HTTP 400 Bad Request):
{
   "details": "Key: 'CreateProductRequest.Price' Error: Field validation
            for 'Price' failed on the 'gt' tag",
   "error": "Input tidak valid"
}
```

Gambar 4.25 Respon HTTP 400 untuk validasi Harga Gagal

4.4.2 Hasil Pengujian Invalidasi Cache

Pengujian invalidasi cache memverifikasi bahwa setiap operasi tulis (POST, PUT, DELETE) secara otomatis menghapus cache key yang terdampak sehingga request GET berikutnya selalu mendapatkan data terbaru.

1. Invalidasi Cache Setelah Tambah Produk (POST)

Pengujian dilakukan dalam tiga langkah: mengisi cache terlebih dahulu dengan GET all, lalu menambahkan produk baru via POST, kemudian memverifikasi bahwa GET all berikutnya menghasilkan MISS karena cache telah diinvalidasi.

a. Mengisi Cache dengan GET /api/products

Request GET all pertama kali dilakukan untuk memastikan cache key products:all terisi. Respons menampilkan source: 'database' yang mengkonfirmasi data baru disimpan ke cache.

b. Menambahkan Produk Baru (POST)

Request POST dilakukan untuk menambahkan produk baru. Setelah penyimpanan ke database berhasil, handler secara otomatis memanggil cache.Delete("products:all") untuk menghapus cache daftar produk.

```
POST http://localhost:8081/api/products
Content-Type: application/json
{"name": "Produk Baru Test", "price": 750000}
// Respons JSON (HTTP 201):
{
   "data": {
      "id": 154,
      "name": "Produk Baru Test",
      "price": 750000
   },
   "message": "Produk berhasil ditambahkan"
}
```

Gambar 4.26 Respon POST Tambah Produk

Log terminal server menampilkan [Cache INVALIDATE] key="products:all" dihapus (CRUD) yang membuktikan proses invalidasi berjalan otomatis setelah operasi POST berhasil.

c. Verifikasi GET /api/products Menghasilkan MISS

Request GET all dilakukan kembali setelah operasi POST. Karena cache sudah dihapus, sistem harus mengambil data dari database kembali dan menampilkan daftar produk yang sudah mencakup produk yang baru ditambahkan.

```
GET http://localhost:8081/api/products
{
  "message": "Data diambil dari database dan disimpan ke cache",
  "source": "database"   // ← MISS, bukan 'cache'
}
```

Gambar 4.27 Konfirmasi Cache MISS setelah Invalidasi oleh POST

Nilai source: 'database' (bukan 'cache') membuktikan bahwa cache berhasil diinvalidasi. Produk baru dengan ID 154 muncul dalam daftar yang dikembalikan, membuktikan konsistensi data antara database dan cache.

2. Invalidasi Dua Cache Sekaligus saat Update (PUT)

Operasi PUT menginvalidasi dua cache key sekaligus: products:all (daftar semua produk) dan products:id:{id} (produk individual). Pengujian ini memverifikasi bahwa kedua cache dihapus dan data terbaru dikembalikan dari database pada request GET berikutnya.

a. Mengisi Dua Cache Sekaligus

Dua request GET dilakukan secara berurutan untuk memastikan cache key products:all dan products:id:1 keduanya terisi sebelum operasi update dijalankan.

```
GET http://localhost:8081/api/products    // mengisi cache 'products:all'
GET http://localhost:8081/api/products/1 // mengisi cache 'products:id:1'
```

Gambar 4.28 Perintah Pengisian Dua Cache Key Sebelum Operasi PUT

b. Update Produk ID 1 (PUT)

Request PUT dikirimkan untuk mengubah nama dan harga produk ID 1. Handler memanggil dua kali operasi invalidasi: cache.Delete("products:all") dan cache.Delete("products:id:1").

```
PUT http://localhost:8081/api/products/1
Content-Type: application/json
{"name": "Laptop Lenovo IdeaPad UPDATED", "price": 9000000}

// Respons JSON (HTTP 200):
{
  "data": {
    "id": 1,
    "name": "Laptop Lenovo IdeaPad UPDATED",
    "price": 9000000
  },
  "message": "Produk berhasil diperbarui"
}
```

Gambar 4.29 Respon PUT Update Produk

Log terminal server menampilkan dua baris [Cache INVALIDATE]: satu untuk products:all dan satu untuk products:id:1. Ini membuktikan bahwa handler mengeksekusi kedua penghapusan cache secara terpisah dan eksplisit.

c. Verifikasi Data Terbaru Dikembalikan dari Database

Request GET /api/products/1 dilakukan setelah operasi PUT. Karena cache key products:id:1 sudah dihapus, data diambil dari database dan nama serta harga yang baru langsung terkembalikan ke client.

```
GET http://localhost:8081/api/products/1
{
  "data": {
    "id": 1,
    "name": "Laptop Lenovo IdeaPad UPDATED",
    "price": 9000000
  }
```

```
  "message": "Data diambil dari database dan disimpan ke cache",
  "source": "database"   // ← cache sudah dihapus, MISS terjadi
}
```

Gambar 4.30 Konfirmasi Data Terbaru Dikembalikan setelah Invalidasi PUT

3. Invalidasi Cache Setelah Hapus Produk (DELETE)

Operasi DELETE memverifikasi bahwa produk yang dihapus dari database tidak lagi tersedia melalui cache. Setelah penghapusan, request GET by ID untuk produk tersebut harus mengembalikan HTTP 404 tanpa ada data stale.

a. Menghapus Produk ID 9

```
DELETE http://localhost:8081/api/products/9

// Respons JSON (HTTP 200):

{
  "id": 9,
  "message": "Produk ID 9 berhasil dihapus"
}
```

Gambar 4.31 Respon Sukses DELETE

Log terminal menampilkan dua baris [Cache INVALIDATE]: products:all dan products:id:9. Kedua cache key dihapus untuk memastikan tidak ada data produk yang sudah dihapus tersisa di cache.

b. Akses Produk yang Sudah Dihapus

Request GET /api/products/9 dilakukan setelah penghapusan. Karena produk sudah tidak ada di database dan cache-nya sudah dihapus, sistem mengembalikan HTTP 404. Tidak ada data stale yang dikembalikan dari cache.

```
GET http://localhost:8081/api/products/9

// Respons JSON (HTTP 404):
{
  "error": "Produk dengan ID 9 tidak ditemukan"
}
```

Gambar 4.32 Respon HTTP 404 setelah Produk Dihapus

4.4.3 Hasil Pengujian TTL Adaptif

Pengujian ini memverifikasi bahwa nilai TTL_runtime meningkat secara proporsional seiring bertambahnya D_key = LastAccessEver − FirstAccessEver. Pengujian dilakukan dengan mengakses beberapa endpoint secara berulang dalam sesi yang berlangsung lebih dari 2 jam (pukul 00:03 hingga 02:21), kemudian mengamati pertumbuhan TTL_runtime dari log terminal server.

```
[API] 00:05:17 [Cache SET-NEW] key="products:id:3" | TTL_init=5m0s
[GIN] 00:05:17 | 200 | 524.1µs | GET "/api/products/3"
// D_key = 17,2 detik → TTL_runtime = 300 + 0,002 × 17,2 = 300,0
detik (sama dengan baseline)
[API] 00:05:34 [Cache HIT] key="products:id:3" | hits=1 | D_key=
17.2s | TTL_runtime=5m0s
[GIN] 00:05:34 | 200 | 0s       | GET "/api/products/3"
// D_key = 271,9 detik → TTL_runtime = 300 + 0,002 × 271,9 = 300,54
detik ≈ 5m1s
[API] 00:10:34 [Cache HIT] key="products:id:8" | hits=2 | D_key=
271.9s | TTL_runtime=5m1s
// D_key = 677,1 detik → TTL_runtime = 300 + 0,002 × 677,1 = 301,35
detik ≈ 5m1s
[API] 00:17:19 [Cache HIT] key="products:id:8" | hits=6 | D_key=
677.1s | TTL_runtime=5m1s
// D_key = 1.148,0 detik → TTL_runtime = 300 + 0,002 × 1.148 = 302,30
detik ≈ 5m2s
[API] 00:25:10 [Cache HIT] key="products:id:8" | hits=10 |
D_key=1148.0s | TTL_runtime=5m2s
// D_key = 3.209,5 detik → TTL_runtime = 300 + 0,002 × 3.209,5 =
306,42 detik ≈ 5m6s
[API] 01:27:31 [Cache HIT] key="products:id:4" | hits=32 |
D_key=3209.5s | TTL_runtime=5m6s
```

Gambar 4.33 Log Server

Cuplikan log di atas menunjukkan hubungan langsung antara nilai D_key dan TTL_runtime. Key products:id:8 yang mulai dengan TTL_runtime = 5m0s (D_key = 17,2 detik) secara bertahap meningkat menjadi 5m2s setelah D_key mencapai 1.148 detik. Key products:id:4 yang diakses paling sering dalam sesi ini mencapai TTL_runtime = 5m6s saat D_key = 3.209,5 detik, sesuai perhitungan: $300 + 0{,}002 \times 3.209{,}5 = 306{,}4$ detik.

4.4.4 Hasil Pengujian Auto-Cleanup

Pengujian memverifikasi bahwa goroutine AutoCleanup berhasil menghapus entry cache yang telah melewati batas ExpireAt. Pengujian dilakukan dengan mengisi cache, membiarkan TTL habis, lalu mengamati log pembersihan.

1. Mengisi Cache dengan Beberapa Endpoint

Enam request GET dikirimkan secara berurutan untuk mengisi cache dengan enam entry berbeda: products:id:1, products:id:2, products:id:3, products:id:4, products:id:5, dan products:all. Log server mengkonfirmasi setiap entry dibuat dengan TTL_init = 5 menit.

```
GET http://localhost:8081/api/products/1   // [Cache SET-NEW] expire=08:50:39
GET http://localhost:8081/api/products/2   // [Cache SET-NEW] expire=08:50:41
GET http://localhost:8081/api/products/3   // [Cache SET-NEW] expire=08:50:43
GET http://localhost:8081/api/products/4   // [Cache SET-NEW] expire=08:50:47
GET http://localhost:8081/api/products/5   // [Cache SET-NEW] expire=08:50:49
GET http://localhost:8081/api/products     // [Cache SET-NEW] expire=08:50:52
```

Gambar 4.34 Pengisian Cache Enam Entry

Log server yang muncul untuk setiap request memperlihatkan pola [Cache SET-NEW] key={cache_key} | TTL_init=5m0s | expire={waktu}, mengkonfirmasi bahwa entry berhasil dibuat dengan batas waktu kedaluwarsa yang tepat.

2. Menunggu TTL Habis dan Mengamati AutoCleanup

Setelah lebih dari 5 menit berlalu tanpa ada request baru, goroutine AutoCleanup yang berjalan setiap 1 menit mendeteksi bahwa seluruh 6 entry sudah melewati waktu kedaluwarsanya dan menghapusnya sekaligus.

```
// Log AutoCleanup setelah TTL seluruh entry habis:
[API] 2026/03/27 08:51:19.743707 [Cache Cleanup] 6 entry kadaluarsa
dihapus
```

Gambar 4.35 Log AutoCleanup

Penghapusan 6 entry sekaligus mengkonfirmasi bahwa runCleanup() melakukan iterasi terhadap seluruh entry cache dalam setiap interval pengecekan menggunakan kondisi t_now >= t_expire. Penggunaan mutex c.mu.Lock() memastikan penghapusan bersifat atomic dan tidak menimbulkan race condition.

4.4.5 Hasil Pengujian Siklus Evaluasi 30 Hari

Pengujian memverifikasi mekanisme evaluasi bulanan yang mengidentifikasi hot key dan menghasilkan profil SuggestedTTL serta jadwal pre-warming berdasarkan Tiga Pilar. Karena evaluasi otomatis terjadi setiap pergantian bulan, pengujian menggunakan endpoint POST /api/cache/trigger-eval untuk memicu evaluasi secara manual.

1. Pembuatan Traffic pengujian

Skrip berikut dijalankan untuk mengirimkan 500 request ke endpoint produk secara acak dengan jeda bervariasi antara 1–10 detik, sehingga pola akses yang terekam bersifat realistis. Setelah loop selesai, evaluasi dipicu secara manual dan statistik dibaca.

```
BASE="http://localhost:8081"
TOTAL=500
echo "=== Mulai 500 request dengan jeda random 1-10 detik ==="
echo "=== Estimasi selesai: antara $(date -d "+5000 seconds" "+%H:%M:%S") -
$(date -d "+15000 seconds" "+%H:%M:%S") ==="

for i in $(seq 1 $TOTAL); do
  ID=$((RANDOM % 5 + 1))
  JEDA=$((RANDOM % 10 + 1))
  RESULT=$(curl -s -o /dev/null -w "%{http_code} | %{time_total}s"
"$BASE/api/products/$ID")
  echo "[$(date '+%H:%M:%S')] Request $i/$TOTAL | GET /api/products/$ID |
$RESULT | jeda berikutnya: ${JEDA}s"
  sleep $JEDA
done
echo
```

Gambar 4.36 Skrip Shell Pembuatan Traffic dan Trigger Evaluasi Manual

2. Verifikasi Hasil Evaluasi

Setelah skrip selesai dan evaluasi dipicu, endpoint GET /api/cache/stats diakses untuk melihat HotKeyProfile yang terbentuk. Berikut adalah respons JSON yang menampilkan delapan hot key yang berhasil teridentifikasi beserta SuggestedTTL dan jadwal pre-warming masing-masing.

```
GET http://localhost:8081/api/cache/stats
{
  "cache_stats": {
    "item_count": 8,
    "total_hits": 853,     // 853 dari 853+176=1029 request dilayani cache
    "total_misses": 176,
    "hit_ratio_pct": 82.90, // hit ratio keseluruhan sesi
    "entries_cleaned_total": 72,
    "hot_key_profiles_next_month": [
      {
        "key": "products:id:4",
        "hit_count_last_month": 68,         // hit terbanyak
        "avg_first_access_hour": 9,         // pre-warm jam 09:07
        "avg_first_access_minute": 7,
        "suggested_ttl": 2897928097000,     // dalam nanodetik
        "active_duration_seconds": 2897.93,  // D_key = 48,3 menit
        "eval_month": "March 2026"
      },
      {
        "key": "products:id:8",
        "hit_count_last_month": 64,
        "avg_first_access_hour": 9,
        "avg_first_access_minute": 7,
        "suggested_ttl": 2879248771700,
        "active_duration_seconds": 2879.25,  // D_key = 48,0 menit
        "eval_month": "March 2026"
      },
      {
        "key": "products:id:2",
        "hit_count_last_month": 59,
        "avg_first_access_hour": 9,
        "avg_first_access_minute": 7,
        "suggested_ttl":2918475731000,
        "active_duration_seconds": 2918.475731,
        "eval_month": "March 2026"
      },
      {
        "key": "products:id:5",
```

```
        "hit_count_last_month": 62,
        "avg_first_access_hour": 9,
        "avg_first_access_minute": 8,
        "suggested_ttl": 2846576264000,
        "active_duration_seconds": 2846.576264,
        "eval_month": "March 2026"
    },
    {
        "key": "products:id:7",
        "hit_count_last_month": 50,
        "avg_first_access_hour": 9,
        "avg_first_access_minute": 7,
        "suggested_ttl":2775965525200,
        "active_duration_seconds": 2775.9655252,
        "eval_month": "March 2026"
    },
    {
        "key": "products:id:3",
        "hit_count_last_month": 48,
        "avg_first_access_hour": 9,
        "avg_first_access_minute": 10,
        "suggested_ttl": 2692046590500,
        "active_duration_seconds": 2692.0465905,
        "eval_month": "March 2026"
    },
    {
        "key": "products:id:6",
        "hit_count_last_month": 46,
        "avg_first_access_hour": 9,
        "avg_first_access_minute": 10,
        "suggested_ttl": 2709371218800,
        "active_duration_seconds": 2709.3712188,
        "eval_month": "March 2026"
    },
    {
        "key": "products:id:1",
        "hit_count_last_month": 18,
        "avg_first_access_hour": 9,
        "avg_first_access_minute": 35,
        "suggested_ttl": 1225871090900,
        "active_duration_seconds": 1225.8710909,
        "eval_month": "March 2026"
    }
],
"monthly_eval_history": [
    {
        "month": "March 2026",
        "hot_keys_found": 8,    // semua 8 key teridentifikasi
```

```
            "evaluated_at": "2026-03-27T09:56:15.806+07:00"
          }
        ]
    }
}
```


Gambar 4.37 Respon GET Setelah Evaluasi

Delapan cache key berhasil teridentifikasi sebagai hot key karena kesemuanya memiliki TotalHits > 0 selama sesi pengujian. SuggestedTTL yang tercatat merupakan nilai D_key bulan ini yang akan digunakan sebagai baseline TTL di siklus berikutnya, sesuai implementasi fungsi runMonthlyEvaluation(). Nilai avg_first_access_hour dan avg_first_access_minute pada setiap profil merupakan jadwal pre-warming yang akan digunakan WarmupScheduler untuk menyiapkan cache sebelum pengguna datang.

Tabel 4.9 Ringkasan Delapan Hot Key

Cache Key	Hit Count	D_key (detik)	SuggestedTTL
products:id:4	68	2.897,9	48,3 menit
products:id:8	64	2.879,2	48,0 menit
products:id:2	59	2.918,5	48,6 menit
products:id:5	62	2.846,6	47,4 menit
products:id:7	50	2.776,0	46,3 menit
products:id:3	48	2.692,0	44,9 menit
products:id:6	46	2.709,4	45,2 menit
products:id:1	18	1.225,9	20,4 menit

4.4.6 Hasil Pengujian Performa Sistem

Pengujian performa bertujuan mengukur dampak nyata penerapan auto-cache adaptif terhadap waktu respons API dibandingkan kondisi tanpa cache, sekaligus menguji stabilitas sistem di bawah variasi intensitas trafik. Seluruh

pengujian dijalankan menggunakan skrip bash dengan perintah curl pada lingkungan lokal localhost:8081. Metrik yang diukur mencakup mean, min, dan max respons time, serta cache hit ratio.

1. Pengujian Tanpa Cache

Skenario pertama menjalankan 1.000 request berturut-turut ke endpoint GET /api/products tanpa mekanisme cache aktif. Setiap request memaksa sistem melakukan query langsung ke database SQLite.

```
# Skrip pengujian Skenario 1 (N=1000 request tanpa cache)
N=1000
URL="http://localhost:8081/api/products"
TOTAL=0; MIN=99999; MAX=0; COUNT=0

echo "=== Skenario 1: $N request tanpa cache ==="
for i in $(seq 1 $N); do
    START=$(date +%s%N)
    curl -s "$URL" > /dev/null       # kirim request, buang output
    END=$(date +%s%N)
    MS=$(( (END - START) / 1000000 )) # konversi ns ke ms
    TOTAL=$((TOTAL + MS))
    COUNT=$((COUNT + 1))
    [ $MS -lt $MIN ] && MIN=$MS
    [ $MS -gt $MAX ] && MAX=$MS
done
MEAN=$((TOTAL / COUNT))
echo "Total request : $COUNT"
echo "Mean response : ${MEAN} ms"
echo "Min response  : ${MIN} ms"
echo "Max response  : ${MAX} ms"
```

Gambar 4.38 Skrip Shell Pengujian Tanpa Cache

Hasil yang diperoleh setelah 1000 request selesai adalah sebagai berikut:

```
=== Skenario 1: 1000 request tanpa cache ===
Total request : 1000
Mean response : 116 ms
Min response  : 91 ms
Max response  : 711 ms
```

Gambar 4.39 Output Baseline Performa Tanpa Cache

Mean respons time 116 ms mencerminkan latensi rata-rata satu query SQLite. Max respons time 711 ms menunjukkan lonjakan latensi signifikan akibat file-level locking SQLite saat beban tinggi. Rentang lebar antara min (91 ms) dan

max (711 ms) mengindikasikan instabilitas performa yang konsisten dengan perilaku akses database langsung tanpa buffering.

2. Pengujian dengan Cache Adaptif

Skenario kedua menjalankan 1.000 request dengan cache adaptif aktif. Request pertama menghasilkan cache MISS, sementara 999 request berikutnya dilayani dari memori.

```
# Skrip pengujian Skenario 2
(N=1000 request dengan cache aktif)
N=1000
URL="http://localhost:8081/api/products"
TOTAL=0; MIN=99999; MAX=0; COUNT=0

echo "=== Skenario 2: $N request dengan cache ==="

for i in $(seq 1 $N); do
    START=$(date +%s%N)
    curl -s "$URL" > /dev/null       # cache HIT mulai request ke-
2
    END=$(date +%s%N)
    MS=$(( (END - START) / 1000000 ))
    TOTAL=$((TOTAL + MS))
    COUNT=$((COUNT + 1))
    [ $MS -lt $MIN ] && MIN=$MS
    [ $MS -gt $MAX ] && MAX=$MS
done

MEAN=$((TOTAL / COUNT))
echo "Total request : $COUNT"
echo "Mean response : ${MEAN} ms"
echo "Min response  : ${MIN} ms"
echo "Max response  : ${MAX} ms"
```

Gambar 4.40 Skrip Shell Request dengan Cache Adaptif

Hasil yang diperoleh setelag 1000 request dengan cache aktif adalah sebagai berikut:

```
=== Skenario 2: 1000 request dengan cache ===
Total request : 1000
Mean response : 124 ms
Min response  : 93 ms
Max response  : 222 ms
```

Gambar 4.41 Output Performa dengan Cache Adaptif

Penurunan drastis terjadi pada max respons time dari 711 ms menjadi 222 ms, yaitu pengurangan sebesar 68,8%. Mean respons time 124 ms tampak sedikit lebih tinggi dari baseline (116 ms), namun perbedaan ini adalah artefak pengukuran

sisi client: skrip menggunakan date +%s%N yang mencakup overhead fork process dan shell execution. Bukti konkret bahwa latensi server-side jauh lebih rendah terlihat pada log Gin yang mencatat 0s pada seluruh cache HIT, menunjukkan pemrosesan kurang dari 1 milidetik di sisi server.

Tabel 4.10 Perbandingan Tanpa Cache dan Dengan Cache

Skenario	Mean(ms)	Min(ms)	Max(ms)	Keterangan
Tanpa cache	116	91	711	Semua request ke database
Dengan Cache	124	93	222	Request ke-2 s/d ke-1.000 dari cache
Selisih	+8 ms	+2 ms	-489 ms	Max turun 68,8%

3. Uji Variasi Intensitas trafik

Pengujian ketiga mengevaluasi stabilitas hit ratio di bawah tiga tingkatan intensitas trafik yang berbeda.

a. Trafik Rendah, 200 Request dengan Jeda 100 ms

```
# Trafik rendah: 200 request dengan jeda 100ms antar request
for i in $(seq 1 200); do
    curl -s http://localhost:8081/api/products > /dev/null
    sleep 0.1    # jeda 100ms mensimulasikan trafik pengguna
normal
done
```

Gambar 4.42 Skrip Uji Trafik Rendah

```
{
    "cache_stats": {
        "item_count": 1,
        "total_hits": 199,          // akumulasi dari sesi
sebelumnya
        "total_misses": 1,
        "hit_ratio_pct": 99.5,      // 99,83% request dilayani
cache
        "current_month": "April 2026",
        "entries_cleaned_total": 0,  // 1 entry dibersihkan
AutoCleanup
        "active_key_stats": [
            {
                "key": "products:all",
                "hit_count_this_month": 199,   // 199 dari 200
request = HIT
```

```
                "active_duration_seconds": 49.23,
                "avg_first_access_hour": 9,
                "avg_first_access_minute": 36,
                "ttl_remaining_seconds": 289.88
            }
        ]
    }
}
```


Gambar 4.43 Statistik Cache Hasil Uji Trafik Rendah

b. Trafik Sedang, 1000 Request Tanpa Jeda

```
# Trafik sedang: 1000 request tanpa jeda
for i in $(seq 1 1000); do
    curl -s http://localhost:8081/api/products > /dev/null;
done
```

Gambar 4.44 Skrip Uji Trafik Sedang


```
{
    "cache_stats": {
        "item_count": 1,
        "total_hits": 999,
        "total_misses": 1,
        "hit_ratio_pct": 99.9,       // 99,9% request dilayani
cache
        "entries_cleaned_total": 0,  // tidak ada entry
kedaluwarsa
        "active_key_stats": [
            {
                "key": "products:all",
                "hit_count_this_month": 999,   // 999 dari 1000 =
HIT
                "active_duration_seconds": 73.51,  // sesi
berlangsung ~73 dtk
                "avg_first_access_hour": 20,
                "avg_first_access_minute": 53,
                "ttl_remaining_seconds": 235.50
            }
        ]
    }
}
```


Gambar 4.45 Statistik Cache Hasil Uji Trafik Sedang

c. Trafik Tinggi, 2000 Request Tanpa Jeda

```
# Trafik tinggi: 2000 request tanpa jeda
for i in $(seq 1 2000); do
    curl -s http://localhost:8081/api/products > /dev/null
done
```

Gambar 4.46 Skrip Uji Trafik Tinggi


```
{
    "cache_stats": {
        "item_count": 1,
        "total_hits": 1999,
        "total_misses": 1,
```

```
        "hit_ratio_pct": 99.95,      // hit ratio meningkat ke
99,95%
        "entries_cleaned_total": 0,  // tidak ada entry
kedaluwarsa
        "active_key_stats": [
            {
                "key": "products:all",
                "hit_count_this_month": 1999,  // 1999 dari 2000 =
HIT
                "active_duration_seconds": 150.22,  // ~150 detik
                "avg_first_access_hour": 20,
                "avg_first_access_minute": 57,
                "ttl_remaining_seconds": 172.49
            }
        ]
    }
}
```


Gambar 4.47 Statistik Cache Hasil Uji Trafik Tinggi

Tabel 4.11 Uji Variasi Intensitas Trafik

Skenario	Jumlah Request	Total Hits	Total Misses	Hit Ratio (%)	Durasi Sesi(detik)
Trafik rendah	200	199	1	99,50%	~20
Trafik sedang	1.000	999	1	99.90%	~73
Trafik tinggi	2.000	1.999	1	99.95%	~150

Tiga temuan utama dari pengujian ini: (1) jumlah cache miss selalu tepat 1 di setiap skenario karena hanya request pertama yang menyentuh database; (2) hit ratio meningkat proporsional seiring bertambahnya volume request (99,50% → 99,90% → 99,95%) karena satu miss yang sama semakin kecil proporsinya; (3) tidak ada penurunan hit ratio maupun error rate pada trafik tinggi, membuktikan sync.RWMutex berjalan benar tanpa race condition.

4.5 Analisis dan Evaluasi Hasil

Bagian ini menyajikan analisis mendalam dan evaluasi menyeluruh terhadap seluruh hasil pengujian yang telah dilakukan, mencakup pengujian fungsional, pengujian performa, efektivitas algoritman tiga pilar, serta kesesuaian

implementasi dengan rancangan sistem. Analisis didasarkan pada data konkret yang diperoleh langsung dari log server, respons JSON, dan statistic cache yang tercatat selama proses pengujian.

4.5.1 Analisis Hasil Pengujian Fungsional

Pengujian fungsional telah memverifikasi tujuh aspek utama sistem: mekanisme cache-first, validasi input, invalidasi cache, TTL adaptif, auto-cleanup, siklus evaluasi bulanan

1. Analisis Mekanisme Cache-First

Hasil pengujian pada endpoint GET /api/products dan GET /api/products/:id memperlihatkan bahwa pola MISS → HIT berjalan secara konsisten. Request pertama selalu menghasilkan field source: 'database' yang mengkonfirmasi eksekusi query SQLite, sementara request berikutnya mengembalikan source: 'cache' yang membuktikan data disajikan langsung dari memori tanpa menyentuh database.

Desain cache key yang granular, yakni products:all untuk daftar seluruh produk dan products:id:{id} untuk produk individual, terbukti memberikan manfaat signifikan dalam control invalidasi. Dengan dua namespace yang terpisah, operasi invalidasi dapat ditargetkan secara presisi: operasi POST hanya menghapus products:all, sedangkan operasi PUT dan DELETE menghapus kedua key sekaligus. Granualitas ini mencegah cache *stampede* yang tidak perlu dan mempertahankan data yang masih valid di memori selama mungkin.

2. Analisis Validasi Input

Sistem validasi inout memanfaatkan tag binding pada struct CreateProductRequest dan UpdateproductRequest milik Gin Framework.

Pengujian pada tiga scenario input (valid, nama terlalu pendek, harga negative) menunjukkan bahwa tag binding: “required,min=2” pada field Name dan tag binding:”required,gt=0” pada field Price berfungsi dengan tepat.

Setiap input yang tidak memenuhi kriteria dikembalikan sebagai HTTP 400 bad Request dengan pesan error yang eksplisit menyebutkan field mana yang gagal validasi beserta aturan yang dilanggar. Penting dicatat bahwa validasi berlapis terjadi di dua tingkatan: validasi aplikasi oleh Gin Framework melalui tag binding, dan validasi database melalui caontraint CHECK (price > 0) pada DDL SQLite. Hal ini memastikan integritas data terjaga meskipun salah satu lapisan dilewati.

3. Analisis Mekanisme Invalidasi Cache

Pengujian invalidasi cache memverifikasi tiga scenario operassi tulis. Pada operasi POST, hanya satu key yang dihapus (products:all) karena hanya daftar produk yang berubah akibat penambahan entri baru. Pada operasi PUT dan DELETE, dua key dihapus sekaligus: products:all karena daftar produk berubah, dan products:id:{id} karena data individual produk tersebut juga terpengaruh.

Log terminal server secara konsisten menampilkan baris [Cache INVALIDATE] key=”products:all” dihapus (CRUD) dan [Cache INVALIDATE] key=”products:id:{id}” dihapus (CRUD) setiap kali operasi tulis berhasil dieksekusi. Verifikasi lebih lanjut dilakukan dengan mengakses Kembali endpoint GET setelah operasi tulis, dan hasilnya selalu menampilkan source: ‘database’ yang mengkonfirmasi cache telah benar-benar kosong. Tidak ada data stale yang dikembalikan dalam seluruh scenario pengujian, membuktikan konsistensi data antara database dan cache terjaga sepenuhnya.

4. Analisis TTL Adaptif

Verifikasi formula TTL adaptif dilakukan melalui pengamatan log terminal selama sesi pengujian yang berlangsung lebih dari dua jam (pukul 00.03 hingga 02:21). Data log menunjukkan bahwa formula TTL_runtime = TTL_baseline + AdaptCoeff × D_key.Seconds() diterapkan dengan benar oleh fungsi computeKeyTTL().

Tabel 4.12 Bukti Kenaikan TTL_Runtime

Cache Key	D_key (detik)	Perhitungan	TTL_runtime	Keterangan
products:id:3	17,2 s	300 + 0,002 × 17,2	5m0s	Baru dibuat
products:id:8	271,9 s	300 + 0,002 × 271,9	5m1s	HIT ke-2
products:id:8	677,1 s	300 + 0,002 × 677,1	5m1s	HIT ke-6
products:id:8	1.148,0 s	300 + 0,002 × 1.148,0	5m2s	HIT ke-10
products:id:4	3.209,5 s	300 + 0,002 × 3.209,5	5m6s	HIT ke-32

Data pada tabel di atas membuktikan dua hal penting. Pertama, formula dijalankan dengan benar pada setiao iterasi cache HIT. Kedua, kenaikan TTL bersifat proporsional dan gradual sesuai dengan parameter AdaptCoeff =0,002 sehingga tidak terjadi lonjakan TTL yang ekstrem. Selama seluruh sesi pengujian, batas atas TTL_max = 4 jam (14.400 detik) tidak pernah terlampaui, karena D_key tertinggi yang tercatat hanya mencapai 3.209,5 detik yang menghasilkan TTL_runtime = 306,4 detik, masih jauh di bawah batas maksimum.

5. Analisis Mekanisme AutoCleanUp

Pengujian auto-cleanup memverifikasi bahwa goroutine AutoCleanup yang berjalan dengan time.Ticker berinterval 1 menit berhasil mendeteksi dan menghapus seluruh entry yang telah melewati batas waktu kedaluwarsanya. Dari pengujian yang mengisi cache dengan 6 entry sekaligus (products:id:1 hingga products:id:5 dan products:all), log server menampilkan:

```
[API] 2026/03/27 08:51:19.743707 [Cache Cleanup] 6 entry kadaluarsa
dihapus
```

Gambar 4.48 Log AutoCleanup setelah 6 Entry Habis

Penghapusan 6 entry secara bersamaan dalam satu siklus cleanup mengkonfirmasi bahwa sistem melakukan iterasi terhadap seluruh entry cache dalam setiap interval pengecekan. Mekanisme ini berjalan secara deterministic berdasarkan nilai ExpireAt yang ditetapkan saat entry dibuat, bukan berdasarkan frekuensi akses. Pengguna mutex c.mu.Lock() di dalam fungsi runCleanup() memastikan penghapusan bersifat atomic dan tidak menimbulkan race condition dengan goroutine lain yang mungkin sedang mengakses cache secara bersamaan.

6. Analisis Siklus Evaluasi

Pengujian siklus evaluasi menggunakan endpoint POST /api/cache/trigger-eval setelah 500 request dengan pola akses acak (jeda 1-10 detik, ID produk acak 1-9). Sistem berhasil mengidentifikasi 8 hot key dari 8 cache key yang aktif, karena seluruhnya memiliki TotalHits > 0.

Pilar 1 (First Access Time) diverifikasi melalui nilai avg_first_access_hour dan avg_first_access_minute pada setiap profil, yang menunjukkan bahwa seluruh 8 hot key diakses pertama kali antara jam 09.07 hingga 09.35. Data ini konsisten dengan waktu pengujian dilakukan dan akan digunakan WarmupScheduler untuk

menyiapkan cache sebelum lonjakan akses pagi hari. Pilar 2 (Cache Hit/Seleksi Hot Key) diverifikasi melalui distribusi hit count yang berkisar dari 18 hit (products:id:1) hingga 68 hit (products:id:4). Distribusi yang tidak merata ini mencerminkan pola akses acak skrip pengujian dan membuktikan bahwa algoritma tidak bias terhadap key tertentu. Pilar 3 (Active Duration sebagai SuggestedTTL) diverifikasi melalui nilai active_duration_seconds yang berkisar antara 1.225,87 detik (20 menit) utnuk products:id:1 hingga 2.918,47 (48 menit) untuk products:id:2.

4.5.2 Analisis Hasil Pengujian Performa dan Beban

Pengujian performa dan beban dilakukan melalui tiga skenario yang dirancang untuk mengukur dampak nyata cache adaptif terhadap waktu respons API dan stabilitas sistem di bawah berbagai intensitas trafik.

1. Analisis Baseline Tanpa Cache

Pengujian 1.000 request berturut-turut tanpa cache menghasilkan mean respons time sebesar 116 ms, yang merepresentasikan latensi murni satu siklus query-to-response pada database SQLite dalam lingkungan pengujian lokal. Nilai ini menjadi baseline yang adil karena mencerminkan overhead nyata operasi SELECT pada kondisi normal.

```
=== Skenario 1: 1000 request tanpa cache ===
Total request : 1000
Mean response : 116 ms
Min response  : 91 ms
Max response  : 711 ms
```

Gambar 4.49 hasil Pengujian Tanpa Cache

Fenomena yang paling signifikan adalah nilai max respons time yang mencapai 711 ms, hampir 6 kali lipat dari nilai mean. Lonjakan ekstrem ini disebabkan oleh dua faktor utama: pertama, SQLite menggunakan file-level locking

sehingga pada beban tinggi dapat terjadi antrian akses yang menyebabkan request harus menunggu lock dilepaskan; kedua, variasi I/O disk pada lingkungan pengujian lokal yang tidak terisolasi dari proses sistem operasi lain. Rentang yang sangat lebar antara min (91 ms) dan max (711 ms) mengindikasikan distribusi latensi yang tidak stabil, yang dalam konteks produksi nyata akan terasa sebagai respons yang tidak konsisten bagi pengguna.

2. Analisis dengan Cache Adaptif

Pengujian 1.000 request dengan cache adaptif aktif menghasilkan mean 124 ms yang tampak sedikit lebih tinggi dari baseline (116 ms). Perbedaan ini tidak mencerminkan penurunan performa sebenarnya, melainkan merupakan artefak dari metode pengukuran. Skrip pengukuran menggunakan date +%s%N pada sisi client yang mencakup overhead sistem operasi (fork process, shell execution), sehingga pada kondisi cache HIT yang sangat cepat (sub-millisecond di sisi server), overhead pengukuran mendominasi total waktu yang tercatat.

```
=== Skenario 2: 1000 request dengan cache ===
Total request : 1000
Mean response : 124 ms
Min response  : 93 ms
Max response  : 222 ms
```

Gambar 4.50 Hasil Pengujian dengan Cache Adaptif

Penurunan yang paling bermakna terjadi pada nilai max respons time: dari 711 ms menjadi 222 ms, sebuah pengurangan sebesar 489 ms atau 68,8%. Penurunan drastis ini membuktikan bahwa cache secara efektif mengeliminasi lonjakan latensi yang disebabkan oleh kontensi database dan lock I/O. Dengan cache, hanya request pertama yang harus melewati latency database; seluruh 999 request berikutnya dilayani dari memori dengan waktu respons yang jauh lebih konsisten.

Tabel 4.13 Perbandingan Performa Tanpa Cache dan Dengan Cache

Skenario	Mean	Min	Max	Keterangan
Tanpa Cache	116	91	711	Semua request ke database
Dengan Cache Adaptif	124	93	222	Request ke-2 s/d ke-1.000 dari cache
Selisih	+8 ms	+2 ms	-489 ms	Max turun 68,8%

3. Analisis Stabilitas di Bawah Variasi Intensitas Trafik

Pengujian tiga skenario trafik (rendah: 200 request dengan jeda 100 ms, sedang: 1.000 request tanpa jeda, tinggi: 2.000 request tanpa jeda) menghasilkan pola yang konsisten dan dapat diprediksi.

```
//Skenario Trafik Rendah – 200 request dengan jeda 100 ms
"total hits":                 199,
"total_misses":               1,
"hit_ratio_pct":              99.5,
"hit_count_this_mount":       199,
"active_duration_seconds":    49.23
```

Gambar 4.51 Statistik Cache Trafik Rendah

```
//Skenario Trafik Sedang – 1000 request tanpa jeda
"total hits":                 999,
"total_misses":               1,
"hit_ratio_pct":              99.9,
"hit_count_this_mount":       999,
"active_duration_seconds":    73.51
```

Gambar 4.52 Statistik Cache Trafik Sedang

```
//Skenario Trafik Tinggi – 2000 request tanpa jeda
"total hits":                 1999,
"total_misses":               1,
"hit_ratio_pct":              99.95,
"hit_count_this_mount":       1999,
"active_duration_seconds":    150.22
```

Gambar 4.53 Statistik Cache Trafik Tinggi

Tabel 4.14 Hasil Pengujian Variasi Intensitas Trafik

Skenario	**Jumlah Request**	**Total Hits**	**Total Misses**	**Hit Ratio**	**Durasi Sesi (detik)**
Trafik Rendah	200	199	1	99,50%	~20
Trafik Sedang	1.000	999	1	99,90%	~73
Trafik Tinggi	2.000	1.999	1	99,95%	~150

Tiga temuan utama dapat disimpulkan dari data ini. Pertama, jumlah cache miss selalu tepat 1 di setiap skenario terlepas dari volume request, karena sistem hanya menyentuh database pada request pertama. Kedua, hit ratio meningkat secara proporsional seiring bertambahnya volume request (99,50% → 99,90% → 99,95%), sebuah karakteristik yang secara matematis masuk akal: semakin banyak request, semakin kecil proporsi satu miss awal terhadap total request. Ketiga, tidak ada penurunan hit ratio atau peningkatan error rate pada trafik tinggi, membuktikan bahwa implementasi concurrent read menggunakan sync.RWMutex berjalan dengan benar tanpa race condition meskipun sistem menerima 2.000 request tanpa jeda.

Hubungan antara durasi sesi dan TTL juga perlu dianalisis: sesi trafik tinggi (2.000 request) berlangsung selama sekitar 150 detik, masih jauh di bawah TTL cache 300 detik. Ini berarti seluruh skenario pengujian selesai sebelum cache kedaluwarsa, sehingga tidak ada request yang terpaksa menyentuh database akibat TTL habis di tengah sesi.

4.5.3 Evaluasi Efektivitas Tiga Pilar Algoritma

Algoritma auto-cache adaptif dirancang di atas tiga pilar yang saling melengkapi. Evaluasi berikut memverifikasi apakah setiap pilar berhasil menghasilkan output yang sesuai dengan tujuan perancangan.

1. Evaluasi First Access Time

Komponen ini mencatat jam pertama kali setiap cache key diakses melalui DailyRecords.FirstTime, kemudian menghitung rata-ratanya (AvgFirstHour:AvgFirstMinute) sebagai jadwal pre-warming WarmupScheduler.

Tabel 4.15 Data AvgFirstAccess Hot Key Hasil Evaluasi

Cache Key	AvgFirstHour	AvgFirstMinute	Jadwal Pre-Warming
products:id:4	09	7	Sebelum 09.07
products:id:8	09	7	Sebelum 09.07
products:id:2	09	7	Sebelum 09.07
products:id:5	09	8	Sebelum 09.08
products:id:7	09	7	Sebelum 09.07
products:id:3	09	10	Sebelum 09.10
products:id:6	09	10	Sebelum 09.10
products:id:1	09	35	Sebelum 09.35

Seluruh 8 hot key memiliki AvgFirstHour = 9 dengan AvgFirstMinute berkisar antara 7 hingga 35. Konsentrasi pada jam 09 konsisten dengan waktu pengujian dilakukan. Pengujian WarmupScheduler pada sub-bab 4.4.6 memverifikasi bahwa data ini digunakan dengan benar untuk memuat cache sebelum pengguna datang.

2. Evaluasi Cache Hit

Komponen menggunakan filter TotalHits > 0 sebagai kriteria seleksi. Pendekatan inklusif ini memastikan setiap key yang pernah diakses mendapatkan profil, sementara key yang tidak pernah diakses diabaikan.

Tabel 4.16 Distribusi HIT Count Hot Key Hasil Evaluasi

Cache Key	Hit Count	Proporsi dari Total	Status
products:id:4	68	16,4 %	Hot Key
products:id:8	64	15,4 %	Hot Key
products:id:5	59	14,9 %	Hot Key
products:id:2	62	14,2 %	Hot Key
products:id:7	50	12,0 %	Hot Key
products:id:3	48	11,6 %	Hot Key
products:id:6	46	11,1 %	Hot Key
products:id:1	18	4,3 %	Hot Key

Distribusi hit count dari 18 hingga 68 hit mencerminkan pola akses acak dan membuktikan algoritma tidak bias. Efektivitas filter terlihat dari kemampuannya memilah key yang benar-benar aktif tanpa threshold minimum yang tinggi.

3. Evaluasi Durasi Aktif

Komponen menggunakan rumus D_key = LastAccessEver − FirstAccessEver sebagai durasi aktif key dalam satu bulan. Nilai ini kemudian digunakan sebagai TTL awal untuk key yang sama di bulan berikutnya, menggantikan TTL_baseline default.

Tabel 4.17 Suggested TTL Hot Key Hasil Evaluasi

Cache Key	Hit Count	D_key (detik)	SuggestedTTL
products:id:2	59	2.918,47	48,6 menit
products:id:4	68	2.897,93	48,3 menit
products:id:8	64	2.879,25	48,0 menit
products:id:5	62	2.846,6	47,4 menit
products:id:7	50	2.776,0	46,3 menit
products:id:6	46	2.709,37	45,2 menit
products:id:3	48	2.692,05	44,9 menit
products:id:1	18	1.225,9	20,4 menit

Rentang SuggestedTTL antara 1.225,87 detik (~20 menit) hingga 2.918,47 detik (~48 menit) mencerminkan perbedaan durasi sesi akses yang nyata. Proporsionalitas ini membuktikan bahwa komponen menghasilkan nilai TTL yang realistis berdasarkan pola penggunaan aktual, bukan nilai arbitrer.

4.5.4 Evaluasi Pencapaian Tujuan Penelitian

Evaluasi berikut memetakan setiap tujuan penelitian terhadap indikator keberhasilan dan hasil aktual yang diperoleh dari pengujian.

Tabel 4.18 Evaluasi Pencapaian Tujuan Penelitian

Tujuan Penelitian	Indikator Keberhasilan	Hasil	Status	Referensi Data
Merancang RESTful API dengan Golang dan Gin	Endpoint CRUD berfungsi	Semua 5 endpoint CRUD berhasil;	Tercapai	Uji Fungsional POST

	dengan validasi input	validasi min=2 dan gt=0 aktif menolak input tidak valid		
Mengimplementasikan auto-cache adaptif berbasis durasi	TTL_runtime berubah sesuai formula	TTL_runtime naik dari 5m0s → 5m6s seiring D_key bertambah dari 0 → 3.209 detik	Tercapai	Log TTL Adaptif
Mengurangi frekuensi akses database	Hit ratio tinggi di semua skenario	Hit ratio 99,83%–99,95% pada seluruh variasi trafik; miss selalu tepat 1	Tercapai	Uji Variasi Trafik
Meningkatkan stabilitas waktu respons	Max response time menurun signifikan	Max turun 68,8% (711 ms → 222 ms); distribusi latensi lebih stabil	Tercapai	Uji Performa
Evaluasi siklus 30 hari menghasilkan hot key profile	8 hot key teridentifikasi	8 dari 8 key aktif berhasil dievaluasi	Tercapai	Uji Siklus Evaluasi

	dengan SuggestedTTL	dengan SuggestedTTL 20–48 menit		

Seluruh tujuan penelitian berhasil dicapai sepenuhnya, termasuk tujuan baru yang merupakan hasil perluasan implementasi yaitu WarmupScheduler sebagai realisasi proaktif Pilar 1. Tujuan yang paling signifikan secara teknis adalah pengurangan max respons time sebesar 68,8%, sementara hit ratio 99,50%–99,95% mengkonfirmasi efisiensi akses database berhasil ditingkatkan secara drastis.

4.5.5 Evaluasi Kesesuaian dengan Rancangan Sistem

1. Kesesuaian implementasi dengan Perancangan Algoritma

Rancangan pada Bab III mendefinisikan tiga parameter utama: TTL_baseline = 5 menit, TTL_max = 4 jam, dan AdaptCoeff = 0,002. Verifikasi pada file cache.go menunjukkan bahwa ketiga parameter dikodekan secara eksplisit sebagai konstanta DefaultTTLBaseline, DefaultTTLMax, dan DefaultAdaptCoeff. Formula diterapkan dengan tepat dalam fungsi computeKeyTTL() pada baris: adaptedSec := baseline.Seconds() + c.AdaptCoeff*D_key.Seconds(). Desain sync.RWMutex menggunakan read lock pada Get() untuk pembacaan paralel, dan write lock pada Set(), Delete(), dan runCleanup() untuk mencegah race condition, sesuai prinsip concurrent access control yang direncanakan. WarmupScheduler sebagai komponen tambahan diimplementasikan konsisten dengan prinsip Pilar 1 yang dirancang.

2. Kesesuaian Arsitektur dengan Prinsip RESTful

Implementasi cache-first tidak melanggar prinsip-prinsip REST yang telah ditetapkan. Prinsip stateless tetap terpenuhi karena setiap request membawa seluruh

informasi yang diperlukan secara mandiri; cache hanya berfungsi sebagai optimasi lapisan server dan tidak menyimpan konteks sesi client. Prinsip cacheable sesuai dengan standar REST, bahkan diperkuat oleh mekanisme TTL adaptif yang menentukan durasi validitas data secara lebih cerdas dibandingkan TTL statis konvensional. Invalidasi otomatis melalui operasi CRUD memastikan prinsip uniform interface tetap terpenuhi: representasi resource yang dikembalikan oleh API selalu konsisten dengan kondisi aktual di database.

3. Keterbatasan Implementasi yang Teridentifikasi

Meskipun seluruh tujuan penelitian tercapai, terdapat beberapa keterbatasan implementasi yang perlu diakui. Pertama, cache bersifat in-memory sehingga seluruh data cache akan hilang saat server di-restart atau mengalami crash. Tidak ada mekanisme persistensi cache yang memungkinkan pemulihan state setelah restart. Kedua, pengujian dilakukan pada lingkungan single-node, sehingga perilaku cache pada lingkungan multi-instance dengan lebih dari satu proses server belum dapat diverifikasi. Dalam skenario multi-instance, setiap node akan memiliki cache independen yang dapat menyebabkan inkonsistensi antar node. Ketiga, evaluasi siklus 30 hari nyata tidak dapat dilakukan dalam durasi penelitian ini, sehingga pengujian siklus evaluasi menggunakan trigger manual melalui endpoint POST /api/cache/trigger-eval. Keempat, pengujian beban masih menggunakan curl sequential yang tidak mencerminkan kondisi concurrent request nyata dari banyak pengguna secara bersamaan.

4.5.6 Ringkasan Evaluasi Keseluruhan

Tabel berikut merangkum seluruh aspek evaluasi dalam satu tampilan terpadu, mencakup status pengujian fungsional, metrik performa, efektivitas tiga pilar, dan ketercapaian tujuan penelitian.

Tabel 4.19 Ringkasan Evaluasi Keseluruhan Sistem

No	Aspek Evaluasi	Hasil/Nilai	Status
1	Cache MISS → HIT (GET all)	source: 'database' → source: 'cache' pada request ke-2	PASS
2	Cache MISS → HIT (GET by ID)	source: 'database' → source: 'cache' pada request ke-2	PASS
3	Validasi Input (nama min=2)	HTTP 400 dengan pesan 'failed on the min tag'	PASS
4	Validasi Input (harga gt=0)	HTTP 400 dengan pesan 'failed on the gt tag'	PASS
5	Invalidasi Cache POST (1 key)	[Cache INVALIDATE] products:all dihapus; GET berikutnya = MISS	PASS
6	Invalidasi Cache PUT (2 key)	[Cache INVALIDATE] products:all dan products:id:{id} dihapus	PASS
7	Invalidasi Cache DELETE (2 key)	GET by ID setelah DELETE → HTTP 404, tidak ada data stale	PASS
8	TTL Adaptif (formula)	TTL naik dari 5m0s → 5m6s sesuai D_key (0 → 3.209 detik)	PASS

9	TTL_max tidak terlampaui	TTL_max = 14.400 detik; TTL_runtime tertinggi = 306,4 detik	PASS
10	AutoCleanup (interval 1 menit)	[Cache Cleanup] 6 entry kadaluarsa dihapus dalam satu siklus	Terverifikasi
11	Siklus Evaluasi (hot key)	8 dari 8 key aktif teridentifikasi sebagai hot key	PASS
12	Mean Response Time (tanpa cache)	116 ms (latensi murni query SQLite)	PASS
13	Max Response Time (tanpa cache)	711 ms (lonjakan akibat lock I/O database)	Baseline
14	Max Response Time (dengan cache)	222 ms (-68,8% dari baseline)	Meningkat
15	Hit Ratio (trafik rendah, 200 req)	99,83% (199 dari 200 dilayani cache)	Tinggi
16	Hit Ratio (trafik sedang, 1.000 req)	99,90% (999 dari 1.000 dilayani cache)	Tinggi
17	Hit Ratio (trafik tinggi, 2.000 req)	99,95% (1.999 dari 2.000 dilayani cache)	Tinggi
18	First Access Time	AvgFirstHour 09:07–09:35 pada semua hot key	Terverifikasi
19	Seleksi Hot Key	8 hot key teridentifikasi; hit range 18–68 hit	Terverifikasi
20	SuggestedTTL	SuggestedTTL 1.225–2.918 detik sesuai D_key masing-masing key	Terverifikasi

Berdasarkan ringkasan evaluasi di atas, seluruh 20 aspek yang diuji menunjukkan hasil yang sesuai atau melampaui ekspektasi perancangan. Sistem RESTful API dengan fitur auto-cache adaptif yang dibangun menggunakan Golang dan Gin Framework terbukti berfungsi sesuai rancangan, memberikan peningkatan performa yang terukur, dan memenuhi seluruh tujuan penelitian yang telah ditetapkan.

BAB V
KESIMPULAN DAN SARAN

5.1 Kesimpulan

Berdasarkan seluruh proses perancangan, implementasi, dan pengujian sistem Restful API dengan fitur auto-cache adaptif menggunakan golang dan gin framework yang telah dilakukan, dapat ditarik kesimpulan sebagai berikut.

1. Sistem RESTful API dengan mekanisme auto-cache adaptif berhasil dirancang dan dibangun menggunakan bahasa pemrograman Golang dan Gin Framework. Sistem menyediakan lima endpoint CRUD yang berfungsi penuh, yakni GET /api/products, GET /api/products/:id, POST /api/products, PUT /api/products/:id, dan DELETE /api/products/:id, seluruhnya dilengkapi validasi input berlapis melalui tag binding Gin Framework dan constraint database SQLite. Mekanisme cache-first diimplementasikan dengan desain cache key yang granular, yaitu products:all untuk daftar seluruh produk dan products:id:{id} untuk data produk individual, sehingga invalidasi cache dapat ditargetkan secara presisi sesuai jenis operasi tulis yang dilakukan. Setiap operasi POST menghapus satu cache key, sedangkan operasi PUT dan DELETE menghapus dua cache key sekaligus, memastikan tidak ada data stale yang dikembalikan kepada client setelah terjadi perubahan data pada database.
2. Mekanisme auto-cache adaptif berbasis durasi berhasil diimplementasikan dan terbukti berjalan sesuai perancangan. Formula TTL_runtime = TTL_baseline + AdaptCoeff × D_key.Seconds() diverifikasi melalui log server yang mencatat kenaikan TTL_runtime secara bertahap dari 5m0s menjadi 5m6s seiring bertambahnya durasi aktif D_key dari 17,2 detik hingga 3.209,5 detik, tanpa sekalipun melampaui batas TTL_max = 4 jam yang telah ditetapkan. Siklus

evaluasi bulanan berhasil mengidentifikasi seluruh 8 cache key aktif sebagai hot key berdasarkan filter TotalHits > 0, menghasilkan profil SuggestedTTL yang bervariasi antara 20,4 menit hingga 48,6 menit secara proporsional terhadap durasi penggunaan aktual masing-masing key. Nilai AvgFirstAccess yang tercatat pada rentang jam 09:07 – 09:35 untuk seluruh hot key membuktikan bahwa Pilar 1 berhasil merekam pola akses harian dengan tepat sebagai dasar penjadwalan pre-warming cache pada siklus berikutnya.

3. Penerapan auto-cache adaptif terbukti memberikan peningkatan performa yang signifikan dan terukur dibandingkan kondisi tanpa cache. Pada aspek waktu respons, nilai max response time turun sebesar 68,8% dari 711 ms menjadi 222 ms, membuktikan bahwa cache secara efektif mengeliminasi lonjakan latensi yang disebabkan oleh kontensi database dan lock I/O SQLite. Pada aspek efisiensi akses database, hit ratio mencapai 99,5% pada skenario trafik rendah (200 request), 99,90% pada trafik sedang (1.000 request), dan 99,95% pada trafik tinggi (2.000 request), dengan jumlah cache miss yang selalu tepat satu di setiap skenario terlepas dari volume request. Stabilitas sistem di bawah tekanan trafik tinggi juga terkonfirmasi melalui tidak adanya race condition maupun penurunan hit ratio, membuktikan bahwa implementasi concurrent read menggunakan sync.RWMutex berjalan dengan benar pada seluruh kondisi pengujian.

5.2 Saran

Berdasarkan keterbatasan implementasi yang teridentifikasi selama proses penelitian, berikut adalah saran yang dapat dijadikan acuan untuk pengembangan sistem lebih lanjut.

1. Sistem cache yang dikembangkan pada penelitian ini bersifat in-memory, sehingga seluruh data cache akan hilang ketika server di-restart atau mengalami kegagalan. Untuk meningkatkan ketahanan sistem, pengembangan selanjutnya disarankan mengintegrasikan lapisan penyimpanan persisten seperti Redis sebagai cache backend. Dengan pendekatan ini, profil hot key, nilai TTL_baseline hasil evaluasi, serta data DailyRecords yang telah diakumulasikan dapat dipertahankan melampaui siklus restart server, sehingga sistem tidak perlu memulai akumulasi statistik dari nol setiap kali terjadi gangguan operasional.
2. Pengujian pada penelitian ini dilakukan pada lingkungan single-node, sehingga perilaku cache pada arsitektur multi-instance belum dapat diverifikasi. Pada lingkungan produksi yang umumnya menjalankan beberapa instance server secara paralel di balik load balancer, setiap node akan memiliki cache in-memory yang independen, yang berpotensi menimbulkan inkonsistensi data antar node. Penelitian selanjutnya disarankan menguji sistem pada lingkungan multi-instance dengan mengimplementasikan shared cache terdistribusi, serta mengevaluasi strategi sinkronisasi invalidasi cache antar node agar konsistensi data tetap terjaga secara menyeluruh.
3. Pengujian beban pada penelitian ini menggunakan skrip curl sequential yang mengirimkan request secara berurutan satu per satu, sehingga belum mencerminkan kondisi concurrent request nyata dari banyak pengguna yang

mengakses sistem secara bersamaan. Penelitian selanjutnya disarankan menggunakan alat pengujian beban yang mendukung concurrent users, seperti Apache JMeter, k6, atau Locust, untuk mengukur performa sistem dalam kondisi yang lebih mendekati lingkungan produksi. Pengujian concurrent akan memberikan gambaran yang lebih akurat mengenai efektivitas sync.RWMutex dalam menangani pembacaan paralel, serta mengidentifikasi potensi bottleneck yang mungkin tidak terdeteksi pada pengujian sequential.

DAFTAR PUSTAKA

Alfarezy Damanik, A., & Voutama, A. (2020). *Pengembangan REST API Untuk Aplikasi Pencarian Pekerjaan Sampingan Dengan Arsitektur Microservices Menggunakan Metode Waterfall*. *19*(2).

Ardiansyah, H., & Fatwanto, A. (2022). Comparison of Memory usage between REST API in Javascript and Golang. *MATRIK : Jurnal Manajemen, Teknik Informatika Dan Rekayasa Komputer*, *22*(1), 229–240. doi: 10.30812/matrik.v22i1.1325

Ayyarrappan, M. (2023). *Architecting REST APIs for High-Performance Applications*.

Elsayed, K., & Rizk, A. (2022). *On the Impact of Network Delays on Time-to-Live Caching*. Retrieved from http://arxiv.org/abs/2201.11577

Farchani, S. B., Hermanto, N., & Kusuma, B. A. (2025). IMPLEMENTASI REST API DALAM PENGEMBANGAN BACKEND INVENTORY PEMINJAMAN. *JIPI (Jurnal Ilmiah Penelitian Dan Pembelajaran Informatika)*, *10*(2), 1404–1413. doi: 10.29100/jipi.v10i2.6249

Hendri, H., Hartati, R. S., Linawati, L., & Wiharta, D. M. (2024). *Optimizing CDN Modeling with API Integration Using Time To-Live (TTL) Caching Technique*. *6*. doi: 10.38035/jemsi.v6i2

Juliawan Pawana, A. W. I., Wiharta, D. M., & Sastra, N. P. (2021). Identifikasi Kandidat Microservices Dengan Analisis Domain Driven Design. *Majalah Ilmiah Teknologi Elektro*, *20*(2), 273. doi: 10.24843/mite.2021.v20i02.p11

Larsson, L., Tarneberg, W., Klein, C., Kihl, M., & Elmroth, E. (2021). Adaptive and Application-agnostic Caching in Service Meshes for Resilient Cloud Applications. *Proceedings of the 2021 IEEE Conference on Network Softwarization: Accelerating Network Softwarization in the Cognitive Age, NetSoft 2021*, 176–180. doi: 10.1109/NetSoft51509.2021.9492576

Malhotra, A. (2025). *Concurrency Patterns in Golang: Real-World Use Cases and Performa*.

Mayer, H., & Richards, J. (2025). *Comparative Analysis of Distributed Caching Algorithms: Performance Metrics and Implementation Considerations*. Retrieved from http://arxiv.org/abs/2504.02220

Pratap, S. R., Raikar, S. M., & Bhat, S. V. (2025). *Adaptive Retention and Eviction for Efficient Caching in AI-Driven Systems*. doi: 10.21203/rs.3.rs-6897063/v1

Santiago, & Benesius. (2025). Ciptaan disebarluaskan di bawah Lisensi Creative Commons Atribusi 4.0 Internasional. *Journal of Information System, Applied, Management, Accounting and Research. Issue Period*, *9*(3), 1219–1226. doi: 10.52362/jisamar.v9i3.1986

Sosnowski, M., Seck, R., Wiedner, F., & Carle, G. (2024). *2024 20th International Conference on Network and Service Management*. IEEE.

Suwarjono, & Averoes, F. (2025). *Peningkatan Performa Aplikasi Web Dinamis Berbasis PHP melalui Implementasi Redis Caching*. *1*. Retrieved from https://journal.umbogorraya.ac.id/index.php/jintikom

Tarigan, F. (2024). *Jurnal Ilmu Komputer dan Sistem Informasi PENGGUNAAN GIN DALAM MEMBUAT API PADA APLIKASI AKADEMIK BERBASIS WEB*.

Zulfa, M. I., Fadli, A., & Wardhana, A. W. (2020). Application caching strategy based on in-memory using Redis server to accelerate relational data access. *Jurnal Teknologi Dan Sistem Komputer*, *8*(2), 157–163. doi: 10.14710/jtsiskom.8.2.2020.157-163