International Journal of Advanced Reasoning and Computational Intelligence (IJARCI)
Vol. 1, No. 1, Juy 2026, pp. xx~xx
ISSN: -, DOI: -

Smart RESTful API Design with Adaptive Auto-Cache Feature Using Golang and Gin Framework

Dita Aulia Al Farid[1], Ahmad Rahmatika[2]
[1]Faculty of Computer Science and Information Technology, Study Program of Information Technology,
Universitas Muhammadiyah Sumatera Utara, Medan, Indonesia
[2]Faculty of eacher Training and Education, Study Program of Mathematics Education,
Universitas Muhammadiyah Sumatera Utara, Medan, Indonesia

Email: [1]ditaauliaalfarid@gmail.com, [2]ahmadrahmatika@umsu.ac.id

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.

Article Info

Article history:

Received May, 30, 2026
Revised June 21, 20026
Accepted July 12, 2026

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

Corresponding Author:
Name: Dita Aulia Al Farid
Department: Information Technology Study Program
Institution: Universitas Muhammadiyah Sumatera Utara
Address: Medan
Email: ditaauliaalfarid@gmail.com

1. INTRODUCTION

The rapid advancement of information technology has driven the proliferation of modern applications including e-commerce platforms, mobile applications, and Internet of Things (IoT) devices that rely on efficient communication mechanisms for real-time data exchange. Application Programming Interfaces (APIs) serve as the backbone for this communication, with RESTful API (Representational State Transfer) emerging as the dominant architectural standard due to its lightweight, flexible, and platform-agnostic nature [1]-[3].

As user bases grow and request frequencies increase, backend servers face significant performance challenges. Each time a client submits an identical request, the API must repeatedly query the database, leading to increased response times, elevated server loads, and suboptimal CPU and memory consumption [4]-[5]. This degradation of system performance directly impacts user experience, particularly in high-traffic environments.

Caching temporarily storing frequently accessed data or API responses is a well-established optimization technique that reduces direct database access and significantly improves response times [6]-[8]. However, most existing caching implementations rely on static Time-to-Live (TTL) configurations that do not account for the dynamic nature of user access patterns. A fixed TTL causes either premature cache eviction of

ISSN: -

frequently accessed data or unnecessarily prolonged retention of rarely accessed data, both of which lead to suboptimal memory utilization [9]-[11].

A review of prior research reveals a gap in RESTful API systems incorporating adaptive caching based on temporal usage duration analysis. Most systems still rely on developer-defined static TTL values without mechanisms to evaluate how long data is actively used within an operational cycle [12]-[15]. This gap motivates the development of an adaptive caching approach that adjusts cache lifetime based on actual usage duration patterns.

This study proposes a smart RESTful API built using Golang and the Gin Framework, equipped with a duration-based adaptive auto-cache mechanism. Golang was selected for its high-performance concurrency model via goroutines, and the Gin Framework was chosen for its lightweight, high-speed HTTP handling and modular middleware architecture. The system implements the formula TTL_runtime = TTL_baseline + AdaptCoeff x D_key, dynamically adjusting TTL based on the active duration of each cache key. A monthly evaluation cycle identifies hot keys through three analytical pillars: first daily access time, total cache hit count, and active duration as SuggestedTTL [16]-[18].

2. METHOD

2.1. Research Approach

This study employs a Software Engineering Research methodology with a qualitative descriptive approach, classified as applied research due to its focus on designing, building, and evaluating a deployable backend system. The experimental method is applied through software engineering experiments comparing two system conditions: a baseline RESTful API without adaptive cache and the developed API with the adaptive auto-cache feature [19]-[21].

2.2. System Architecture

The system architecture integrates Golang and the Gin Framework as the core backend, an in-memory custom adaptive cache, and a SQLite database for data persistence. The cache operates on a cache-first strategy: incoming requests are checked against the in-memory cache before reaching the database. Granular cache keys (*products:all* and *products:id:{id}*) enable precise cache invalidation targeted to the specific operation type.

2.3. Adaptive TTL Mechanism

The adaptive auto-cache mechanism is built upon three core components: duration monitoring, adaptive TTL calculation, and a periodic evaluation cycle.

2.3.1. Duration Monitoring

The system records two temporal markers for each cache key: *FirstAccessEver* (set when a cache entry is first created via SET) and *LastAccessEver* (updated on every cache HIT via GET). The active duration is computed as [22], [23]:

$$D\_key = LastAccessEver - FirstAccessEver \quad (1)$$

Additionally, *DailyRecords* captures the first access time per day for each cache key, enabling the WarmupScheduler to pre-warm cache entries before peak usage periods in subsequent cycles.

2.3.2. Adaptive TTL Calculation

The system uses the following formula to compute the runtime TTL [24]:

$$TTL\_runtime = TTL\_baseline + AdaptCoeff\ x\ D\_key.Seconds \quad (2)$$

System parameters are set as: TTL_baseline = 5 minutes (300 seconds), TTL_max = 4 hours (14,400 seconds), and AdaptCoeff = 0.002. If TTL_runtime exceeds TTL_max, it is capped at TTL_max to ensure system stability.

2.3.3. Monthly Evaluation Cycle

A goroutine *MonthlyEvaluationCycle* checks every hour for a month transition. Upon detection, the function *runMonthlyEvaluation()* processes accumulated statistics through three analytical pillars [25]:

Pillar 1 - First Access Time: Records the daily first access hour (AvgFirstHour/AvgFirstMinute) per hot key for WarmupScheduler pre-warming scheduling.

Pillar 2 - Cache Hit Selection: Identifies hot keys by filtering cache keys with TotalHits > 0 during the evaluation period.

Pillar 3 - Active Duration as SuggestedTTL: Uses D_key as the recommended TTL baseline for the subsequent cycle, clamped between TTL_baseline and TTL_max.

2.4. Cache Invalidation Strategy

To maintain data consistency, CRUD operations trigger targeted cache invalidation: POST invalidates *products:all*; PUT and DELETE invalidate both *products:all* and *products:id:{id}*. The AutoCleanup goroutine (1-minute interval) removes expired entries based on the *ExpireAt* condition (t_now >= t_expire), using mutex locking to prevent race conditions.

2.5. Testing Methodology

Performance testing used bash scripts with curl commands on a local environment (localhost:8081), measuring mean, min, and max response times across 1,000 sequential requests. Traffic variation tests evaluated hit ratios under three load levels: low (200 requests, 100 ms delay), medium (1,000 requests, no delay), and high (2,000 requests, no delay). The monthly evaluation cycle was tested by triggering manual evaluation via POST /api/cache/trigger-eval after 500 requests with random access patterns (1-10 second delays, random product IDs 1-9).

3. RESULTS AND DISCUSSION

3.1. Functional Testing

Functional testing verified the complete cache lifecycle, input validation, cache invalidation, adaptive TTL behavior, auto-cleanup, and monthly evaluation. The cache-first mechanism consistently produced the MISS → HIT transition: the first request returned *"source": "database"* while subsequent identical requests returned *"source": "cache"*, confirming that data was served from in-memory cache without touching the database.

Input validation via Gin Framework's binding tags (*required,min=2* for Name and *required,gt=0* for Price) successfully rejected invalid inputs with HTTP 400 responses specifying the failed validation rule. Two-layer validation (Gin Framework and SQLite DDL constraint CHECK (price > 0) ensures data integrity is maintained even if one layer is bypassed.

Cache invalidation testing confirmed that each POST operation invalidated exactly one cache key (*products:all*), while PUT and DELETE operations invalidated two keys simultaneously. Server logs consistently showed [Cache INVALIDATE] entries after each write operation, and subsequent GET requests always returned *"source": "database"*, confirming zero stale data was returned across all test scenarios.

3.2. Adaptive TTL Verification

Server log observation over a 2-hour session verified the proportional growth of TTL_runtime with increasing D_key values. Table 1 presents representative measurements from this session.

Table 1. TTL_runtime Growth with Increasing D_key

Cache Key	D_key (sec)	Formula	TTL_runtime
products:id:3	17.2	300 + 0.002x17.2	5m0s
products:id:8	271.9	300 + 0.002x271.9	5m1s
products:id:8	677.1	300 + 0.002x677.1	5m1s
products:id:8	1148.0	300 + 0.002x1148.0	5m2s
products:id:4	3209.5	300 + 0.002x3209.5	5m6s

The results demonstrate that TTL_runtime grows gradually and proportionally with D_key, from 5m0s at initialization (D_key = 17.2 s) to 5m6s at D_key = 3,209.5 s. At no point did TTL_runtime exceed the TTL_max ceiling of 4 hours, confirming the stability control mechanism functions correctly.

3.3. Auto-Cleanup Verification

Six cache entries were created simultaneously with TTL_init = 5 minutes. After the TTL elapsed, the AutoCleanup goroutine successfully detected and removed all six expired entries within a single cleanup cycle, as evidenced by the server log: *[Cache Cleanup] 6 expired entries removed*. This confirms that the goroutine

ISSN: -

iterates all cache entries in each 1-minute interval using the deterministic condition t_now >= t_expire, with mutex locking ensuring atomic deletion without race conditions.

3.4. Monthly Evaluation Cycle

After 500 requests with randomized access patterns, the manual evaluation trigger via POST /api/cache/trigger-eval successfully identified 8 hot keys. Table 2 summarizes the profiling results.

Table 2. Summary of 8 Hot Keys Identified in Monthly Evaluation

Cache Key	Hit Count	D_key (sec)	SuggestedTTL
products:id:4	68	2,897.9	48.3 min
products:id:8	64	2,879.2	48.0 min
products:id:2	59	2,918.5	48.6 min
products:id:5	62	2,846.6	47.4 min
products:id:7	50	2,776.0	46.3 min
products:id:6	46	2,709.4	45.2 min
products:id:3	48	2,692.0	44.9 min
products:id:1	18	1,225.9	20.4 min

All 8 active cache keys were correctly identified as hot keys since each had TotalHits > 0. SuggestedTTL values ranging from 20.4 to 48.6 minutes reflect proportional differences in actual usage duration, validating Pillar 3. The AvgFirstAccess times ranging from 09:07 to 09:35 for all hot keys, consistent with the testing session timing, demonstrate that Pillar 1 accurately captures daily access patterns for pre-warming scheduling.

3.5. Performance Testing

Table 3 presents the comparison between API performance without cache (Scenario 1: 1,000 sequential requests directly to SQLite) and with adaptive cache (Scenario 2: 1,000 requests utilizing the cache-first mechanism).

Table 3. Performance Comparison: Without Cache vs. With Adaptive Cache

Scenario	Mean (ms)	Min (ms)	Max (ms)	Note
Without Cache	116	91	711	All requests to DB
With Cache	124	93	222	Request 2-1000 from cache
Difference	+8 ms	+2 ms	-489 ms	Max reduced 68.8%

The most significant improvement is the 68.8% reduction in maximum response time, from 711 ms to 222 ms. The baseline maximum of 711 ms is attributed to SQLite's file-level locking under high sequential load, which causes request queuing. With adaptive caching, only the first request incurs database latency; all subsequent requests are served from in-memory cache. The mean response time of 124 ms (vs. 116 ms baseline) appears slightly higher due to client-side measurement overhead from the bash scripting (date +%s%N), not from actual server-side processing. Gin Framework logs confirm cache HIT responses were processed in sub-millisecond time.

3.6. Traffic Variation Testing

Table 4 summarizes the cache performance across three traffic intensities.

Table 4. Cache Performance Under Traffic Intensity Variations

Scenario	Requests	Hits	Misses	Hit Ratio	Duration (s)
Low Traffic	200	199	1	99.50%	~20
Medium Traffic	1,000	999	1	99.90%	~73
High Traffic	2,000	1,999	1	99.95%	~150

Three key findings emerge from this data. First, cache misses were consistently exactly 1 across all scenarios regardless of request volume, since only the initial request accesses the database. Second, hit ratio improves proportionally with request volume (99.50% → 99.90% → 99.95%), reflecting the diminishing proportion of the single miss against total requests. Third, no performance degradation or increased error rate was observed under high traffic (2,000 requests without delay), confirming that the sync.RWMutex implementation prevents race conditions under concurrent load. These results are consistent with findings by Zulfa et al. [2] and Larsson et al. [4], which demonstrate that well-configured in-memory caching significantly reduces database access frequency and stabilizes response time distribution.

3.7. Three Pillars Effectiveness Analysis

Pillar 1 (First Access Time) generated WarmupScheduler schedules with AvgFirstHour = 9 and AvgFirstMinute ranging from 7 to 35 for all 8 hot keys, consistent with the testing session timing and demonstrating accurate daily access pattern capture. Pillar 2 (Cache Hit Selection) produced a non-biased distribution of hit counts from 18 to 68, reflecting the randomized access script behavior. Pillar 3 (Active Duration as SuggestedTTL) yielded proportional SuggestedTTL values from 1,225.87 seconds (~20 minutes) to 2,918.47 seconds (~48 minutes), replacing the 5-minute TTL_baseline default for hot keys in the next cycle. This proportionality confirms that the algorithm generates realistic TTL values based on actual usage patterns rather than arbitrary thresholds.

3.8. Discussion of Limitations

Several implementation limitations should be acknowledged. The in-memory cache loses all data upon server restart, lacking a persistence mechanism for recovery. Performance testing used sequential curl requests rather than true concurrent multi-user simulation, which may not fully represent production load conditions. The 30-day evaluation cycle could not be tested in real-time; manual trigger via endpoint was used instead. Multi-instance deployment was not evaluated; independent in-memory caches per node could cause inter-node inconsistency. Future work should address these limitations by integrating Redis as a distributed cache backend, employing tools such as Apache JMeter or k6 for concurrent load testing, and validating the system in multi-instance environments.

4. CONCLUSION

This study successfully designed and developed a smart RESTful API using Golang and the Gin Framework with a duration-based adaptive auto-cache mechanism. The system provides five fully functional CRUD endpoints with layered input validation and a cache-first strategy using granular cache keys for precise invalidation control.

The adaptive TTL formula TTL_runtime = TTL_baseline + AdaptCoeff x D_key was verified through server log observation, demonstrating proportional TTL growth from 5m0s to 5m6s as D_key increased from 17.2 to 3,209.5 seconds, without exceeding the TTL_max ceiling of 4 hours.

Performance testing demonstrated a 68.8% reduction in maximum response time (711 ms → 222 ms), with cache hit ratios of 99.50%-99.95% across traffic volumes of 200-2,000 requests. Cache misses were consistently exactly one per session, proving the effectiveness of the cache-first mechanism. The monthly evaluation cycle successfully identified all 8 active cache keys as hot keys with SuggestedTTL values ranging from 20.4 to 48.6 minutes, proportional to actual usage duration.

ACKNOWLEDGMENTS

The author thanks Ibu Fatma Sari Hutagalung, S.Kom., M.Kom. for her guidance and support throughout this research, and the Faculty of Computer Science and Information Technology, Universitas Muhammadiyah Sumatera Utara.

ISSN: -

REFERENCES

[1] A. Santiago and L. Benesius, "Development of modern information systems using API and performance optimization techniques," *J. Inf. Syst. Appl. Manage. Account. Res.*, vol. 9, no. 3, pp. 1219-1226, 2025.

[2] M. I. Zulfa, A. Fadli, and A. W. Wardhana, "Application caching strategy based on in-memory using Redis server to accelerate relational data access," *J. Teknologi Dan Sist. Komputer*, vol. 8, no. 2, pp. 157-163, 2020.

[3] S. Suwarjono and F. Averoes, "Peningkatan performa aplikasi web dinamis berbasis PHP melalui implementasi Redis caching," 2025.

[4] L. Larsson *et al.*, "Adaptive and application-agnostic caching in service meshes for resilient cloud applications," in *Proc. IEEE NetSoft*, 2021, pp. 176-180.

[5] K. Elsayed and A. Rizk, "On the impact of network delays on time-to-live caching," *arXiv:2201.11577*, 2022.

[6] H. Ardiansyah and A. Fatwanto, "Comparison of memory usage between REST API in JavaScript and Golang," *MATRIK*, vol. 22, no. 1, pp. 229-240, 2022.

[7] M. Ayyarrappan, "Architecting REST APIs for high-performance applications," 2023.

[8] F. Tarigan, "Penggunaan Gin dalam membuat API pada aplikasi akademik berbasis web," *J. Ilmu Komputer Sist. Inf.*, 2024.

[9] H. Mayer and J. Richards, "Comparative analysis of distributed caching algorithms: Performance metrics and implementation considerations," *arXiv:2504.02220*, 2025.

[10] S. R. Pratap, S. M. Raikar, and S. V. Bhat, "Adaptive retention and eviction for efficient caching in AI-driven systems," 2025.

[11] M. Sosnowski, R. Seck, F. Wiedner, and G. Carle, "CRDT web caching: Enabling distributed writes and fast cache consistency for REST APIs," in *Proc. IEEE NOMS*, 2024.

[12] S. B. Farchani, N. Hermanto, and B. A. Kusuma, "Implementasi REST API dalam pengembangan backend inventory peminjaman," *JIPI*, vol. 10, no. 2, pp. 1404-1413, 2025.

[13] A. Malhotra, "Concurrency patterns in Golang: Real-world use cases and performance," 2025.

[14] A. Alfarezy Damanik and A. Voutama, "Pengembangan REST API untuk aplikasi pencarian pekerjaan sampingan dengan arsitektur microservices menggunakan metode waterfall," vol. 19, no. 2, 2020.

[15] H. Hendri *et al.*, "Optimizing CDN modeling with API integration using time-to-live (TTL) caching technique," *JEMSI*, vol. 6, 2024.

[16] A. W. I. Juliawan Pawana, D. M. Wiharta, and N. P. Sastra, "Identifikasi kandidat microservices dengan analisis domain driven design," *Majalah Ilmiah Teknologi Elektro*, vol. 20, no. 2, pp. 273-280, 2021.

[17] V. Repin and A. Sidorov, “Distributed caching system with strong consistency model,” Frontiers in Computer Science, vol. 7, 2025, doi: 10.3389/fcomp.2025.1511161.

[18] M. Sosnowski, R. von Seck, F. Wiedner, and G. Carle, “CRDT Web Caching: Enabling Distributed Writes and Fast Cache Consistency for REST APIs,” in Proceedings of the 20th International Conference on Network and Service Management (CNSM), 2024, pp. 1–7, doi: 10.23919/CNSM62983.2024.10814315.

[19] J. Liu, Y. Chen, and H. Ding, “CacheSim: A cache simulation framework for evaluating caching algorithms on resource-constrained edge devices,” SoftwareX, vol. 29, p. 102018, 2025, doi: 10.1016/j.softx.2024.102018.

[20] D. Kim and S. Park, “Blockchain-Based Caching Architecture for DApp Data Security and Delivery,” Sensors, vol. 24, no. 14, p. 4559, 2024, doi: 10.3390/s24144559.

[21] M. Chowdhury, “Trailblazer: A proactive caching, blockchain, minimum content retrieval latency, and energy cost-aware resource and job orchestration scheme for publisher-subscriber based 6G ICN services,” Journal of High Speed Networks, vol. 31, no. 1, 2025, doi: 10.3233/JHS-240059.

[22] “CACHE-IT: A distributed architecture for proactive edge caching in heterogeneous IoT scenarios,” Ad Hoc Networks, vol. 156, p. 103413, 2024, doi: 10.1016/j.adhoc.2024.103413.

[23] “Distributed service caching with deep reinforcement learning for sustainable edge computing in large-scale AI,” Digital Communications and Networks, vol. 11, no. 5, pp. 1447–1456, 2025, doi: 10.1016/j.dcan.2024.11.009.

[24] F. Z. Junaedy and U. Surapati, “Optimisasi Web Service REST API Menggunakan Load Balancer dan Cache dengan Algoritma Round Robin,” Jurnal Indonesia Manajemen Informatika dan Komunikasi, vol. 5, no. 3, pp. 3158–3169, 2024.

[25] C. Gu, X. L. Li, R. Kuditipudi, P. Liang, and T. Hashimoto, “Auditing Prompt Caching in Language Model APIs,” arXiv preprint arXiv:2502.07776, 2025.