International Journal of Computer Techniques–IJCT Volume 12 Issue 6, November 2025
Open Access and Peer Review Journal ISSN 2394-2231 https://ijctjournal.org/

Comparative Study of Caching Strategies for Lookup Data in Web Applications

Artsiom Lazouski
Microsoft Solutions Departement
LeverX International
Minsk, Republic of Belarus
lazouskiartsiom@gmail.com

Abstract — Static lookup data is a common type of data widely used in modern web applications. It is frequently accessed while changing infrequently. Despite its simplicity, inefficient handling of such data can introduce unnecessary database load, increased latency, and other performance issues. This paper presents a comparative study of different caching mechanisms for delivering static lookup data. Several approaches are evaluated, including a plain database no-cache approach, distributed caching using Redis, application-level in-memory caching, and browser-based caching using Cache-Control headers. The approaches are analyzed under cold, warm, and update scenarios to provide a comprehensive view of their behavior in realistic operating conditions. The study highlights practical trade-offs between the evaluated strategies based on experimental results and observed performance characteristics.

Keywords — *web caching, lookup data, performance evaluation, distributed cache, in-memory cache, HTTP caching, latency optimization, web APIs*

I. Introduction

Modern web applications operate with vast amounts of data of different types: dynamic data (such as product listings or shopping carts), user-specific data and others. One of them is *static lookup data* – datasets that define application configuration and background data required for many operations but that rarely change. Some typical examples include lists of countries, currencies, regions, ZIP codes, payment types or any kind of dropdown configurations. Despite being often small in size and infrequently updated, lookup data at the same time is repeatedly requested by all clients across sessions and devices. This effect becomes especially visible across distributed environments and microservice architectures, where there is a high demand for data communication within one system. At large scale, load and latency that appears as a consequence of repeated retrieval of this data from the server can become a serious issue for user experience, infrastructure cost, and system reliability.

A common practice to solve these problems is server-side caches (e.g., Redis) or rely on edge caching and Content Delivery Networks (CDNs). While these solutions can reduce database load and response time, they still require round trips across the network, add cache invalidation complications, and do not use browser built-in capabilities. Edge and CDN caching partially solve these concerns, placing data closer to the user. However, they typically are used for static files rather than structured lookup data that is consumed by Application Programming Interfaces (APIs). There are other validator-based strategies and service workers that introduce further options, but there is limited guidance on how these methods can be specifically applied to static lookup data. Existing studies focus primarily on management and delivery of dynamic data, leaving a research gap in caching strategies optimized specifically for static lookup data.

This paper aims to bridge this gap by providing a comprehensive comparative study of caching strategies for managing and transmitting static lookup data in web applications. To achieve this, we implemented and evaluated several caching mechanisms such as server-side distributed caching (using Redis), edge and CDN caching, browser caching via Hypertext Transfer Protocol (HTTP) cache-control headers and others. We evaluated and compared them under identical workloads with a focus on several key performance indicators that reflect both user experience and infrastructure efficiency: average latency, origin load reduction, bandwidth usage, cache hit ratio, and freshness after version updates across cold, warm, and update scenarios.

To reflect realistic application states, we implemented three operational scenarios – *cold*, *warm*, and *update*. The cold scenario simulates the first user interaction with a fresh environment and no pre-existing cache. The warm scenario represents steady-state performance when the server already has cached data. The update scenario evaluates system behavior when data version has changed and cache invalidation with refresh must occur.

II. Related Work

A. Server-Side Caching Studies

Recent research has analyzed the impact of server-side caching mechanisms on database and application performance. Studies show that full-page and in-memory caching can significantly reduce response times and database load in large-scale systems [1]. Other articles indicate that database throughput can be improved through architectural enhancements such as caching proxies and delayed persistence [2]. Additional works go beyond optimization of data retrieval, applying caching principles to computation-heavy contexts such as deep learning inference, reducing redundant GPU operations and latency [3]. Overall, current studies demonstrate that server-side caching is an important part of modern application modern web applications, evolving from simple data reuse to more adaptive and work-specific optimization strategies.

B. Edge and CDN Caching

For large-scale web systems, reducing end-to-end latency and origin load can be achieved through edge caching and CDNs. Latest research indicates that placing frequently requested data at geographically distributed edge nodes can shorten round-trip times and improve latency [6]. Effectiveness of this strategy often depends on several factors, such as placement, policies and delivery [5]. Other studies highlight that some implementation details like cache-key normalization and improved validation using

entity tags (ETags), can stabilize performance. Despite their effectiveness, most studies focus on static content and media, leaving a room for further analysis of how CDN caching behaves for structured data consumed by APIs.

C. Browser and HTTP Caching

Browser caching techniques can significantly reduce the number of partially overlapping requests in the system and introduce reuse mechanisms for static content. This leads to a minimization of redundant data transfers between client and server [7]. Some studies focus on optimizing HTTP caching policies through dynamic and adaptive refresh intervals [8]. These approaches use modern validation mechanisms such as HTTP/2 protocol and ETags to improve cache coherence and overall system efficiency.

D. Cache Invalidation and Freshness Models

Modern cache mechanisms in large web applications must balance between two key goals: server load minimization and data freshness maintenance. Recent research introduces expiration-based caching models that improve this balance by optimizing how and when cached data should be invalidated and refreshed [9]. The results show that lightweight validation and adaptive expiration policies can achieve higher cache-hit ratio and lower response times compared to static time-to-live (TTL) mechanisms. This is especially valuable in complex distributed architectures with many nodes, where static lookup data changes infrequently but must remain consistent across multiple services.

III. Methodology

The main purpose of this study is to evaluate different caching strategies and approaches for delivering static lookup data in web applications. Our goal is to measure latency, throughput, and freshness retention under different operational conditions, such as cold, warm, and update. It will help us to understand which caching mechanisms are the most effective and suitable for structured, infrequently changing datasets accessed through APIs.

To keep experiments controlled and repeatable, we built a minimal but realistic setup for different types of tests:

- Single-request latency experiments using Developer Tools and Postman.
- Load tests using k6.
- Cold/warm/update scenarios.
- Consistency verification for update scenarios.

Across all experiments, we evaluated how each caching mechanism affects these metrics:

- **Latency** – Time to First Byte (TTFB) and total response time.
- **Throughput** – number of requests per second under load.
- **Efficiency** – reduction in external database calls and remote round-trip operations.
- **Scalability** – how quickly system adapts under concurrent traffic.

The following subsections describe the test system, data model, and API endpoints used in our evaluation.

A. System Under Test

To perform controlled and accurate experiments across all caching mechanisms, we designed a minimal lookup-service environment that reflects common patterns in modern web applications.

One of the principles for the fair and comparable testing was maintaining equal hosting conditions for each service. All experiments were executed on the same machine with identical environment settings, including initial system load, background processes, and power configuration.

The test application is lightweight and intentionally minimal to isolate the effect of caching mechanisms. It is an ASP.NET 8 Web API built specifically for this research. Each caching mechanism is exposed through a dedicated GET endpoint to avoid cross-interference and ensure repeatable measurements.

Azure Cosmos DB was selected as the primary data source, allowing experiments to be executed not only against a local emulator but also against a cloud-hosted database to reflect realistic production scenarios. Redis (running in a Docker container) was used to implement the distributed cache approach. For in-memory and HTTP-based caching, default application-level storage and Cache-Control headers were used. Since the API and caching layers ran locally while Cosmos DB remained remote, network round-trip latency was intentionally preserved to match real-world cloud conditions.

Azure Cosmos DB is used as a source of truth for all lookup data. Key configuration:

- Region: the closest to local testing machine.
- Provisioned throughput: sufficient to avoid RU throttling.
- Partitions: each lookup document has its own partition key equal to its ID.

B. Lookup Data Model

Static lookup datasets used in this study represent data that changes infrequently but is requested repeatedly in production systems. Performance of caching solutions strongly depends on dataset size, and while small payloads often produce similar results across caching methods, increasing document size may cause significant divergence in latency and throughput [1].

To evaluate this effect, two lookup documents were created: a small **0.2 KB** entry and a large **1.5 MB** entry. Each was stored as an independent document in a single Cosmos DB container, sharing identical indexing and partitioning configuration. This ensured that observed performance differences originate from payload size rather than database configuration or storage layout.

This dataset design allowed controlled measurement of size-dependent caching behavior and helped eliminate unintended variation from container settings or indexing strategy.

C. Endpoints Under Test

For each caching strategy, a separate API endpoint was implemented in order to evaluate its behavior independently. Every endpoint interacts exclusively with its designated caching layer and therefore prevents interference between approaches. Business logic was intentionally minimized so

that execution flow consists only of the caching mechanism and Cosmos DB access, without additional processing.

To ensure purity of measurement, typical application concerns such as request validation, DTO or domain mappings, and exception handling were excluded, preventing them from affecting latency or throughput results.

The evaluated endpoints are listed below:

- */api/static/{name}*
- */api/static/redis/{name}*
- */api/static/memory/{name}*
- */api/static/etag/{name}*
- */api/static/header-versioned/{name}?v={number}*

D. Measurement Procedure

Almost every caching strategy was tested under three scenarios: cold, warm and update. For every scenario, measurements were collected on the same machine, network and processor load conditions, using the same datasets and execution flow. Each strategy was tested independently via their corresponding endpoints to avoid overlaps and unnecessary artifacts.

1) Cold scenario

This scenario represents state of the system when no cache is populated. For Redis and Cosmos DB scenarios, this was achieved by restarting of the application and clearing Redis memory before measuring the first request.

2) Warm scenario

In this scenario measurements were collected on a steady-state system where cached data has already been loaded. Application was not restarted and cache was not flushed. For every strategy tests were executed 15 times to eliminate anomalies and get more accurate averages. For this scenario, latency was expected to decrease due to cache usage.

3) Update scenario

This scenario was evaluated and measured only for strategies that support cache invalidation and freshness retention. For each test, the lookup document in database was modified. After that client sent new request to observe how quickly cache was invalidated and refreshed. This scenario is important for analyzing real-world behavior, where lookup data changes occasionally and must be handled accordingly.

Load and scalability test were executed using k6 performance benchmarking tool. Each test was evaluated under identical user settings and execution duration. Following results were recorded:

- Average requests per second
- 95th and 90th percentile latencies
- Failure rate (timeouts and errors)
- Throughput stability

These measurements allowed us to evaluate both single request and high-load scenarios.

IV. Caching Strategies Evaluated

A. Baseline: Cosmos DB without Caching

The baseline represents the simplest request-response flow, without any caching mechanism applied. In this approach, each client makes a call to the server, which retrieves lookup data directly from the database. This makes such approach the slowest among all, but at the same time provides vital measurement data for comparing other caching methods.

The execution flow is as follows:

- Clients sends HTTP GET request to */api/static/{name}*
- The API forwards the request to Azure Cosmos DB
- Cosmos retrieves a document by ID and partition key (*name*)
- The response is returned to the client without intermediate caching

To simulate real-world conditions, the application and database were hosted in different environments, introducing natural network round-trips delays. This configuration demonstrates why caching becomes essential as the frequency of lookup data retrieval increases.

B. Server-Side Distributed Cache (Redis)

The second approach extends the baseline by adding a new layer – distributed cache using Redis. In this method, data is retrieved from the database only once and stored in Redis. During subsequent requests, it is served directly from there, without the need to access the database again. This approach significantly reduces the number of round-trips and improves latency.

The execution flow is as follows:

- Clients sends HTTP GET request to */api/static/redis/{name}*
- The API checks whether data exists in Redis
- If present, the cached value is returned immediately
- If absent, the API fetches the value from Cosmos DB, stores it in Redis, and returns the response

Redis was hosted in a local Docker container to avoid network inconsistencies and simulate a real-world deployment scenario. Cache entries were stored without TTL to ensure they remained available for the entire duration of warm-scenario measurements.

Expected advantages of this approach include database load reduction and improved latency. However, it also introduces new complications typical for distributed caching systems.

C. In-memory caching

The third approach introduces application-level caching using in-memory storage inside the ASP.NET runtime. Instead of storing cache data in Redis or any other distributed cache, the application keeps it directly within the process memory of a running instance.

The execution flow is as follows:

- A client sends an HTTP GET request to */api/static/memory/{name}*
- The API checks whether the key exists in the local in-memory dictionary
- If present, the cached value is returned immediately

- If absent, the value is loaded from Cosmos DB, stored in memory, and sent back to the client

This approach, while fast, also introduces trade-offs. Since the cache is stored per instance and not shared across multiple nodes, it becomes more complex to use in distributed environments. Cache invalidation and synchronization must be handled manually to ensure consistency between instances. Additionally, cache is cleared whenever the process or application restarts, which may result in temporary cold-start penalties unless warm-up mechanisms are implemented.

D. HTTP Caching with Etag (Conditional Requests)

This strategy introduces HTTP-based caching using entity tags, allowing the browser to validate whether data has changed since the last request. Instead of fetching a fresh copy of lookup data from the server, the client first checks if cached whether the cached version is still valid.

- A client sends an HTTP GET request to */api/static/etag/{name},* storing the returned ETag and response body in the browser cache
- During subsequent requests, the client includes the stored ETag in an *If-None-Match* header
- If the lookup document remains unchanged, the server responds with **304 Not Modified** without sending the payload again
- If the value has changed, the server returns a **200 OK** response with the updated document and a new ETag

This approach still requires network calls, buy can significantly reduce traffic volume, as repeated requests may transfer only headers rather than actual payload with data. HTTP validation caching offers a balance between freshness and bandwidth efficiency, making it suitable for systems where lookup updates occur occasionally but strict consistency is still necessary.

E. Header-Versioned Caching

The final evaluated approach uses HTTP caching using Cache-Control headers, allowing the browser to store responses indefinitely as long as session remains unchanged. In this model, the browser treats each version of the resource as a unique file, meaning cached data could be retrieved almost instantly.

The execution flow is as follows:

- Client first requests a version map from */api/static/versions*, storing it locally
- The application then retrieves lookup data using */api/static/header-versioned/{name}?v={number}*
- If the requested *{name}* and version *{number}* were previously loaded, the browser serves the cached copy immediately
- On version change, the URL also changes, forcing the browser to fetch a new response but leaving the old one is cached separately

This approach is highly effective for large static datasets because it eliminates heavy round-trips. Fetches are skipped entirely when cache exists, and response time drops to near-zero on the client side. However, this method does not allow cache invalidation to be triggered directly from the server, which introduces additional complexity. One option is to use a TTL configuration of the Cache-Control headers to limit staleness and balance freshness with performance.

V. Test Results and Observations

A. Overview of Collected Measurements

All caching strategies were evaluated under the same execution environment, datasets and test conditions. The main purpose of this section is to present and analyze collected measurements and compare the effectiveness of each method for lookup data retrieval.

Two types of experiments were conducted:

1) Latency-oriented: signle-request timings for cold, warm, and update scenarios. Metrics:

a) Time to First Byte (TTFB)

b) Total response time

2) Load and scalability: conducted using k6 benchmarking tool only for warm scenario. Metrics:

a) Average response duration

b) p90 and p95 latencies

c) Requests-per-second throughput

These measurements form the basis for comparison in the following subsections, where results are analyzed across data sizes and caching strategies.

B. Small Payload Evaluation

This subsection focuses on the retrieval performance for the smallest dataset used in testing (0.2 KB). Although lookup entries of this size are common in real-world web applications, they do not challenge memory or storage throughout limits. For such small payloads, overall latency is dominated not by data transfer time but by cache lookup overhead, network round-trip delays, code-level processing costs, and the presence or absence of database calls.

TABLE I presents the cold-scenario results for each caching strategy. It is important to note that the cold scenario for ETag validation, HTTP Cache-Control headers, and in-memory caching differs from the cold scenarios of the first two approaches. This was intentional: for these three mechanisms, cold measurements were taken after warming up the database with several direct retrieval calls. This was necessary to highlight the difference between database-dependent and cache-dependent strategies. Measurements for these strategies were recorded only after database warm-up to avoid including Cosmos DB cold-start latency, which would blur their actual behavior.

TABLE I.
EVALUATED RESULT FOR COLD SCENARIO / 0.2 KB CASE

Strategy (0.2 KB / cold)	*TTBF*	*Total Response*
Cosmos DB only	~ 2498 ms	~ 2510 ms
Redis cache	~ 2712 ms	~ 2724 ms
ETag validation	~ 124 ms	~ 125 ms
HTTP Cache-Control headers	~ 114 ms	~ 116 ms
In-memory cache	~ 117 ms	~ 118 ms

Similar results were observed for the warm scenario (TABLE II). Strategies that still require a Cosmos DB round-trip, for example Cosmos DB only and ETag validation, show nearly identical performance between cold and warm

states. However, other strategies that do not include database round-trips achieve significantly faster response times once cached data is available.

TABLE II.
EVALUATED RESULT FOR WARM SCENARIO / 0.2 KB CASE

Strategy (0.2 KB / warm)	*TTBF*	*Total Response*
Cosmos DB only	~ 121 ms	~ 124 ms
Redis cache	~ 22 ms	~ 24 ms
ETag validation	~ 118 ms	~ 120 ms
HTTP Cache-Control headers	~ 0 ms	~ 13 ms
In-memory cache	~ 14 ms	~ 17 ms

As previously mentioned, the update scenario was measured only for two strategies – ETag validation and HTTP Cache-Control headers (TABLE III). Their results match the cold scenario measured after database warm-up. This is expected, because in the update case the server must query the database again to retrieve fresh data. This database round-trip takes up most of the total response time. However, in practice, update operations occur far less frequent than warm-scenario requests, which are visibly faster.

TABLE III.
EVALUATED RESULT FOR UPDATE SCENARIO / 0.2 KB CASE

Strategy (0.2 KB / update)	*TTBF*	*Total Response*
ETag validation	~ 124 ms	~ 126 ms
HTTP Cache-Control headers	~ 110 ms	~ 115 ms

To evaluate scalability and behavior under concurrent access, load tests using the k6 performance benchmarking tool was conducted. The warm state was used in this scenario for all strategies to simulate a realistic steady-state system. Each test executed for 30 second with 20 virtual users, applying identical load across all strategies (TABLE IV).

During the test, all strategies sustained stable execution without errors or failures. As expected, average latency increased compared to single-request measurements for every strategy, and some calls showed nearly double the response time under load. These measurements provide a foundation for analyzing scalability properties, which will be discussed in the observation section.

TABLE IV.
LOAD-TEST RESULTS FOR 0.2 KB PAYLOAD

Strategy (0.2 KB)	*Avg. duration*	*p90*	*p95*	*Req / s*
Cosmos only	196 ms	288 ms	353 ms	16
Redis	81 ms	155 ms	174 ms	18
In-memory	21 ms	32 ms	48 ms	20

C. *Large Payload Evaluation*

This subsection evaluates performance of caching strategies with the largest dataset used in this study (1.5 MB). In real-world systems, lookup data of this size is relatively uncommon, yet it appears in certain complex web applications. Additionally, this scenario may represent a case where the client needs to retrieve data aggregated from several different lookup sources.

The dominant factor in response time for such datasets shifts from code-level to network bandwidth, serialization/deserialization cost, and database to API transport time. This makes large payloads a better stress case for understanding the practical limitations for each caching strategy.

TABLE V represents the cold scenario measurements for the large dataset. As expected, every strategy exhibited a substantial increase in response time compared to the 0.2 KB case. This difference is explained not only by slower transfer speeds for larger payloads but also by additional model transformation and memory-allocation overheads.

TABLE V.
EVALUATED RESULT FOR COLD SCENARIO / 1.5 MB CASE

Strategy (1.5 MB / cold)	*TTBF*	*Total Response*
Cosmos DB only	~ 3940 ms	~ 3984 ms
Redis cache	~ 4322 ms	~ 4356 ms
ETag validation	~ 1168 ms	~ 1180 ms
HTTP Cache-Control headers	~ 1214 ms	~ 1226 ms
In-memory cache	~ 1218 ms	~ 1226 ms

TABLE VI summarizes the results for the warm scenario with 1.5 MB dataset. As expected, strategies that bypass database entirely, for example Redis caching, in-memory caching, and HTTP Cache-Control headers, demonstrate visible improvements compared to the cold scenario due to elimination of costly database round-trips. The warm measurements clearly highlight the distinction between strategies that still require database access and those that remove this stage from the execution path.

TABLE VI.
EVALUATED RESULT FOR WARM SCENARIO / 1.5 MB CASE

Strategy (1.5 MB / warm)	*TTBF*	*Total Response*
Cosmos DB only	~ 1262 ms	~ 1280 ms
Redis cache	~ 74 ms	~ 85 ms
ETag validation	~ 1178 ms	~ 1180 ms
HTTP Cache-Control headers	~ 0 ms	~ 15 ms
In-memory cache	~ 20 ms	~ 32 ms

The update scenario for the 1.5 MB dataset was measured only for the strategies that support freshness validation. TABLE VII shows that ETag validation produces results almost identical to the warm state because both require a database call. In contrast, the HTTP Cache-Control mechanism demonstrates a dramatic increase in response time due to the absence of a database round-trip in warm scenario. While update operations are more time-consuming that warm-state lookups, they occur far less frequently in practice.

TABLE VII.
EVALUATED RESULT FOR UPDATE SCENARIO / 1.5 MB CASE

Strategy (1.5 MB / update)	*TTBF*	*Total Response*
ETag validation	~ 1194 ms	~ 1204 ms

Strategy (1.5 MB / update)	*TTBF*	*Total Response*
HTTP Cache-Control headers	~ 1200 ms	~ 1215 ms

Similar to the small payload evaluation, the warm state was used for the load test with the large dataset. The main purpose was to reflect the real production scenario in which the system remains running for an extended period before receiving load. TABLE VIII presents result for the 1.5 MB dataset retrieval under the load. It is clear that the volume of data is a key performance factor, amplifying the differences between strategies. The Cosmos DB approach demonstrates a dramatical increase in response time, both averaging and p90 with p95. This leads to less than 1 request per second for this approach, which is unacceptable for the modern web applications. Redis and in-memory approaches successfully reduce response times under the load.

TABLE VIII.
LOAD-TEST RESULTS FOR 1.5 MB PAYLOAD

Strategy (1.5 MB)	*Avg. duration*	*p90*	*p95*	*Req / s*
Cosmos only	19000 ms	44000 ms	51000 ms	0.74
Redis	414 ms	561 ms	611 ms	14
In-memory	149 ms	210 ms	237 ms	27

D. Cross-Approach Observations

The experimental measurements demonstrate clear and visible differences in the performance profiles of evaluated caching strategies. The baseline no-cache approach using plain Cosmos DB shows the slowest performance in both cold and warm scenarios. Database calls and network round-trips have a huge impact on performance. Large payload amplifies these delays significantly. While this approach always guarantees data freshness without additional cost of cache management, it is non-optimal for lookup data that is frequently accessed and infrequently changes.

Redis caching improves response times considerably, particularly in warm scenarios and under the load. It eliminates repetitive database retrievals and provides a solid middle ground between performance and architectural complexity. However, this approach introduces additional concerns such as distributed cache management, infrastructure cost, cache invalidation considerations, and potential synchronization problems in multi-region deployments. Based on the results, Redis is well suited for medium-sized datasets when system architecture can accommodate distributed caching.

The in-memory caching approach achieves the lowest latency among all evaluated strategies. It eliminates all connections to external storages, enabling near-instant retrieval times for both evaluated payloads. Nevertheless, s, this approach introduces risks related to data consistency, as cache resides only within the memory of a single application instance. Due to nature of this approach, it is suitable primarily to single-node applications. Multi-instance or autoscaling environments require additional coordination to avoid stale data or inconsistent states.

The HTTP-based strategies shift responsibility from the server to the client, reducing redundant downloads and minimizign bandwidth consumption. Cache-Control headers completely trust client in cache management, but bring complications of cache invalidation and its freshness. ETag-based validation addresses some of these concerns by enabling reduced payload transfer while maintaining freshness validation; however, it still requires a full network round-trip, which may affect performance for large datasets.

Overall, it is evident from these results that there is no universal approach to solve all problems of lookup data management with one ultimate strategy. Instead, the optimal choice depends on system architecture, dataset size, frequency of certain operations, and consistency requirements. Trade-offs emerge between performance, infrastructure complexity, and data freshness guarantees, making caching strategy selection a design decision rather than a purely technical optimization.

VI. SUMMARY

A. Practical Summary

The experimental results demonstrate that selecting an efficient caching mechanism for lookup data should be based on dataset size, operational scenarios, system architecture, and consistency requirements, rather than attempting to identify a universal solution. For small lookup payloads, performance differences between approaches are limited, and direct database access may be sufficient in some cases. As dataset size increases, the cost of network round-trips and payload transfer becomes the dominant performance factor, making server-side caching mechanisms such as Redis significantly more effective.

Application-level in-memory caching consistently achieved the lowest latency in warm scenarios by eliminating external dependencies completely, though it is best suited for single-instance deployments due to synchronization complications. Overall, the results emphasize that caching strategy selection is a design decision that balances performance gains against architectural complexity and data freshness guarantees.

B. Limitations

This study has several limitations. The evaluations and measurements were conducted using one application node and one database, which eliminates scenarios of multi-regional and geographically distributed deployments, where latency and consistency behavior may differ substantially. Only read-heavy access patterns were examined, excluding pre-warming strategies, proactive cache refresh mechanisms, and write-intensive workloads. Additionally, experiments were conducted under stable network and database connections, which almost eliminates the presence of failed calls and inconsistency in responses.

C. Conclusion

This study presented a comparative evaluation of several caching mechanisms for lookup data in web applications, considering characteristics such as latency, scalability, consistency, and network load. The results demonstrated that there is no universal solution suitable for every scenario or system architecture. Instead, different mechanisms provide advantages depending on workload patterns and deployment constraints.

Future work may extend this research by evaluating hybrid caching models that combine multiple strategies. Additionally, future studies may explore multi-region deployments, CDN caching, and other approaches that were not examined in this research. Evaluating system behavior under unstable network conditions and write-heavy

workloads would also provide further insight into real-world system behavior.

VII. REFERENCES

[1] M. Olumide, "Reducing latency with caching: An in-depth analysis of effective caching strategies for modern web applications," 2024. [Online]. Available: https://www.researchgate.net/publication/385737667_Reducing_Latency_With_Caching_An_In-Depth_Analysis_Of_Effective_Caching_Strategies_For_Modern_Web_Applications

[2] I.-A. Moise and A. Băicoianu, "Optimizing intensive database tasks through caching proxy mechanisms," April 2024. [Online]. Available: https://arxiv.org/abs/2404.12128

[3] D. Akinyele and S. Adeoye, "Caching and Content Delivery Networks (CDNs) for Performance Optimization,", December 2023. [Online]. Available: https://www.researchgate.net/publication/389814256_Caching_and_Content_Delivery_Networks_CDNs_for_Performance_Optimization

[4] H. Wu, Y. Fan, Y. Wang, H. Ma, and L. Xing, "A comprehensive review on edge caching from the perspective of total process: Placement, policy and delivery," Sensors, vol. 21, no. 5033, pp. 1–28, 2021, doi: 10.3390/s21155033.

[5] M. P. Noura and J. Zhang, "Optimizing CDN architectures: Multi-metric algorithmic breakthroughs for edge and distributed performance," arXiv preprint, December 2024. [Online]. Available: https://arxiv.org/abs/2412.09474

[6] H. Akbar, D. Athar, M. A. F. Rana, C. H. Javed, Z. A. Uzmi, I. A. Qazi, and Z. A. Qazi, "Semantic Caching for Improving Web Affordability," 2025, arXiv preprint. [Online]. Available: https://arxiv.org/abs/2506.20420v1

[7] M. Hosseini, S. Darabi, P. Eugster, M. Choopani, and A. H. Jahangir, "Rethinking Web Caching: An Optimization for the Latency-Constrained Internet," in Proc. 23rd ACM Workshop on Hot Topics in Networks (HotNets'24), Irvine, CA, USA, Nov. 18–19, 2024, pp. 326–332. doi: 10.1145/3696348.3696873

[8] F. Gessert, M. Schaarschmidt, W. Wingerath, S. Friedrich, and N. Ritter, "The Cache Sketch: Revisiting expiration-based caching in the age of cloud data management," in Proc. 40th IEEE Int. Conf. Data Eng. (ICDE '24), Macao, China, Apr. 15–19, 2024, pp. 120-131. [Online]. Available: https://www.baqend.com/paper/btw-cache.pdf