International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(20s) 2026 pp. 1497-1504
https://doi.org/10.70917/ijcisim-2026-6000
Article

# In-Memory Metadata Distribution for Low-Latency Schema Access in High-Throughput Data Platforms

**Rohit Raravi**

Independent Researcher

**Abstract:** A metadata service that fetches from the primary database on every request is borrowing capacity it does not own. This article argues that neither of the two default fixes, scaling that primary store horizontally or bolting on a conventional cache, is architecturally matched to what metadata actually is: high-read, low-write, and changed in discrete, versionable increments rather than continuously. In-memory, producer-consumer snapshot distribution, where versioned metadata snapshots are built and published independently of live request traffic and loaded directly by serving nodes, decouples freshness from availability in a way neither alternative manages. The evidence comes from a production redesign of a high-throughput metadata platform supporting an enterprise experience-data model, where schema-fetch latency and request volume had both grown past what request-time database access could sustain cleanly. The redesigned architecture separates snapshot production, which assembles a versioned, internally consistent metadata state on a scheduled cadence, from snapshot consumption, where serving nodes load that state directly into memory, refreshing it out of band from the request path and including explicit safeguards for staleness, failed builds, and consumer cold starts. Deployed in production, the architecture cut schema-fetch latency from approximately 240 milliseconds to approximately 40 milliseconds and scaled metadata access from hundreds to thousands of requests per second, while measurably reducing load on the primary metadata store. Metadata, this article suggests, is a distinct enough workload category that it deserves its own architectural default, one neither general-purpose caching nor database scaling was built to provide.

**Keywords:** In-memory data distribution, metadata management, distributed systems performance, producer-consumer architecture, schema registry, low-latency retrieval, high-throughput platforms

## 1. Introduction

High-throughput data platforms expose more metadata every year: schema definitions, field types, dataset associations, business context that downstream services and end users query constantly just to understand what data exists and how to use it. Once a platform's catalog grows to hundreds of schemas and thousands of fields, and request volume climbs into the thousands per second, metadata access stops being an occasional lookup. It becomes a load-bearing part of the request path. Fetch that metadata from a primary transactional or document store on every request, and the metadata path inherits that store's latency, and more consequentially, its capacity limits. A spike in metadata reads now competes directly with the writes and application queries the same store also has to serve.

Two responses dominate current practice when this becomes a bottleneck. Scale the primary store horizontally, adding read replicas or partitioning data to spread load. Or insert a conventional request-time cache, typically Redis or something like it, between the application and the store, serving cached responses until a time-to-live expires or an explicit invalidation fires. Both treat metadata the way they'd treat any other frequently accessed application data. Both also carry a coupling metadata's actual access pattern does not require: read latency and throughput stay tied, in one form or another, to the primary store's liveness and load, or to the cache's hit rate under shifting traffic.

Metadata is not, in fact, a generic caching problem, and treating it as one leaves gains on the table. It is read far more often than it is written. It changes in discrete, versioned increments rather than drifting continuously. And critically, it does not need to reflect a write the instant that write happens, so long as staleness stays bounded and predictable. That combination makes metadata a strong candidate for producer-consumer snapshot distribution: a producer assembles a complete, versioned, immutable snapshot of current metadata state, publishes it, and consumer processes on serving nodes load that snapshot directly into memory, refreshing it on a schedule entirely decoupled from the request path. Under this architecture, read latency and throughput become properties of local memory access on the serving node, not of the primary store's load. Freshness becomes an explicit, tunable parameter instead of an incidental byproduct of eviction policy.

The argument here is grounded in a real deployment, not a synthetic benchmark. A high-throughput metadata platform supporting an enterprise experience-data model was redesigned around exactly this pattern, replacing a primary-store-backed metadata service with in-memory snapshot distribution. What follows situates that argument within the distributed-systems literature on consistency, caching, and snapshot semantics, walks through the producer-consumer architecture and its safeguards, reports latency and throughput findings from production, and closes with what this means for how data platform teams should classify and architect around metadata, rather than reaching for whatever caching or scaling pattern is already lying around.

## 2. Literature Review

### *2.1 Caching and Horizontal Scaling as the Default Response to Read-Latency Bottlenecks*

The ceiling on this entire problem class was set well before modern data platforms existed. Gilbert and Lynch's formal proof of the CAP theorem established that no distributed system can guarantee consistency, availability, and partition tolerance all at once, forcing every distributed architecture to choose which guarantee gives way under load or failure [1]. The distributed key-value systems built in the years after made that choice explicit, and mostly chose availability and eventual consistency over strict correctness. Amazon's Dynamo, built on consistent hashing and horizontal partitioning, is the canonical reference design for that choice, and it is exactly the architecture teams reach for when metadata latency gets bad enough to justify scaling the primary store [2]. Vogels later generalized the same tradeoff into a vocabulary engineers still reach for by default when reasoning about distributed data freshness [3].

Conventional request-time caching, the second common response, inherits a related but distinct set of tradeoffs. Shi and colleagues' study of Redis-based caching in production reports peak write-operation volumes ranging from roughly 100 per second to 200,000 per second across real deployment clusters, a range that illustrates both the scale caching layers operate at and the operational weight of managing cache consistency across that range [4]. Jafari and Bayat's work on distributed cache performance via middleware session management reports an 86 percent cache hit rate for their proposed architecture, and read carefully, that figure concedes the core limitation of request-time caching: even a well-tuned cache leaves a meaningful share of requests falling straight through to the primary store [5]. That residual miss rate is not a tuning failure waiting to be optimized away. It is a structural property of any cache that reacts to request patterns instead of tracking the underlying data's own versioning.

### *2.2 In-Memory and Snapshot-Based Data Distribution Architectures for High-Read Workloads*

A separate strand of the literature treats data distribution as a snapshot problem rather than a caching problem, and this is the strand the article's architecture builds on directly. Chandy and Lamport's foundational snapshot algorithm established that a consistent point-in-time view of a distributed system can be captured without halting the system being observed, a property any producer-consumer metadata distribution architecture depends on when assembling a snapshot from a live, continuously updated source [6]. Sun and colleagues extend this into modern multi-versioned index structures for hybrid transactional and analytical workloads, showing that keeping multiple versions of the same data in memory can support efficient snapshot isolation without the coordination overhead naive locking would impose, a design principle directly analogous to how versioned metadata snapshots avoid coordinating with live request traffic at all [7]. Alhomssi and Leis push this further with an in-memory storage engine sustaining roughly

two million transactions per second even under a concurrent long-running analytical scan, scaling near-linearly to sixty-four threads when transactions carry warehouse affinity. Evidence, in other words, that snapshot-isolated, multi-versioned in-memory architectures do not trade throughput away to get consistency guarantees [8].

Publish-subscribe and producer-consumer messaging architectures supply the distribution mechanism once a snapshot exists. Longo and Redondi's work on advanced message-broker design for distributed publish-subscribe scenarios establishes patterns for reliably distributing discrete, versioned units of data to many independent consumers, patterns this article's snapshot-publication step generalizes directly [9]. Sawadogo and Darmont's review of data lake metadata architectures makes the motivating point most explicitly: metadata management deserves architectural treatment distinct from general application data management, because its structure, change cadence, and consumption pattern differ systematically from the data it describes [10]. Brahmia, Grandi, and Oliboni's companion reviews of schema versioning and schema evolution reinforce this from the schema side, cataloguing how schema change is typically discrete and versioned rather than continuous drift, exactly the property that makes metadata snapshot-friendly in a way a live transactional order table, say, is not [11], [12]. Overeem and colleagues corroborate this from industry data, finding that real-world schema changes in production event-sourced systems cluster into discrete, identifiable versions rather than accumulating as continuous change [13].

Read together, this body of work reaches a conclusion the caching and horizontal-scaling literature never quite states: metadata's specific mix of high read frequency, low write frequency, and discrete versioning fits producer-consumer snapshot distribution better than either database scaling or request-time caching, both of which were designed for access patterns metadata does not actually have.

## 3. Materials and Methods

### *3.1 Metadata Versioning and Snapshot Production*

The producer side of the architecture has one job: assemble a complete, internally consistent, versioned representation of the platform's metadata state and publish it as an immutable artifact. Schema, dataset, and field-level metadata are read from the primary store on a scheduled cadence, not per request, transformed into a compact in-memory representation, and versioned as a discrete snapshot rather than a continuously mutating structure. This mirrors the consistent-snapshot principle Chandy and Lamport established, applied here to a metadata source instead of general distributed system state [6], and it leans on the same multi-versioned indexing logic that lets snapshot construction proceed without locking out concurrent writes to the primary store [7].

### *3.2 Snapshot Distribution and Consumer-Side Refresh*

Once built, a snapshot publishes to a distribution layer that serving nodes, acting as consumers, poll or subscribe to independently of the request path. Each consumer loads the full snapshot into local memory and serves every metadata read from that in-memory copy until the next version lands. This is the architectural inversion the whole argument rests on: metadata reads are served from local memory on the same host handling the request, zero network calls to a shared store, while keeping that in-memory copy current happens entirely out of band. Serving those reads through a non-blocking, event-loop-based I/O layer rather than a thread-per-request model matters here specifically because in-memory lookups are cheap enough that thread-management overhead, not the lookup itself, becomes the dominant cost under high concurrency; empirical comparisons of reactive versus traditional blocking server architectures report throughput gains as high as 6.5 times under comparable concurrent-request loads, alongside substantially lower average response times [14]. That advantage compounds specifically in cache-heavy, high-read scenarios resembling in-memory metadata serving, where reactive architectures have been measured delivering markedly higher throughput and a meaningfully smaller memory footprint than their blocking counterparts [15].

Explicit safeguards address the failure modes this inversion introduces. Stale data is bounded by a maximum acceptable snapshot age; past that age, a consumer flags its own data as degraded rather than silently serving arbitrarily old metadata. Failed snapshot builds do not propagate downstream; a consumer that fails to validate an incoming

snapshot keeps serving its last known-good version instead of falling back to an empty or partial state, a resilience pattern in the same family as adaptive circuit-breaker designs that fail toward a known-safe behavior rather than a hard failure, and that have been shown to cut recovery-relevant latency substantially in comparable microservice settings [16]. Cold-start scenarios load the most recent valid snapshot synchronously before accepting any traffic (a lesson learned specifically from watching a node come up cold against a slow snapshot source during early testing), avoiding a window where a freshly started node serves requests against incomplete metadata.

Three checkpoints. Three separate failure modes. No single point where the whole system quietly degrades.

### *3.3 Evaluation Approach*

The evaluation compares this architecture against the baseline it replaced: metadata fetched from the primary store on every request, over a synchronous request-response protocol rather than the snapshot-push model this article proposes, a protocol-level distinction that itself carries measurable throughput and latency consequences independent of caching strategy [17]. Three criteria structure the comparison: schema-fetch latency, measured end-to-end from the moment a request enters the metadata service to the moment a validated response leaves it, tracked at the p50 and p95 percentiles rather than as a single mean figure, since mean latency alone hides exactly the tail behavior that determines whether users notice a slow lookup; throughput, measured in requests served per second sustained before response times began to degrade past acceptable bounds, not a peak burst figure; and primary-store load, measured as the reduction in direct query volume against the primary metadata store attributable to the architectural change. Benchmarking followed the rigor Negi and colleagues argue production database benchmarking actually needs, drawing workload patterns from real platform traffic distributions rather than synthetic uniform load, since synthetic benchmarks routinely understate how bursty and skewed real metadata traffic gets [18].

The workload itself mirrored production traffic shape rather than a uniform synthetic distribution: metadata read requests arriving in the bursty, unevenly distributed pattern characteristic of an enterprise platform where request volume tracks customer business hours and batch-processing windows rather than arriving smoothly around the clock, with a long tail of infrequently accessed schemas alongside a small number of hot schemas responsible for a disproportionate share of traffic. Both the request-time-fetch baseline and the in-memory snapshot-distribution architecture were evaluated against this same traffic shape, replayed against each configuration during its respective production window rather than in a side-by-side synthetic comparison, so that the reported latency and throughput figures reflect what each architecture actually sustained under live enterprise load rather than under a controlled but unrepresentative benchmark. Measurement spanned a sustained observation window covering multiple full business cycles rather than a single peak-traffic snapshot, specifically to avoid the common benchmarking error of reporting a best-case interval as if it were representative; the figures in Section 4 reflect that full window, not an isolated favorable sample.

The deployment environment was the production infrastructure supporting the enterprise experience-data platform itself, not an isolated staging replica, meaning both the before and after measurements absorbed whatever normal production variance, co-located workloads, and infrastructure conditions the platform experienced during each respective period. That choice trades some measurement cleanliness for ecological validity: the comparison reflects what changed for real users under real operating conditions, not an idealized isolated benchmark environment that might not survive contact with production traffic.

One methodological complication surfaced during benchmarking, and it is worth stating plainly rather than smoothing over. Schemas carrying an unusually large number of labels produced snapshot-construction times an order of magnitude longer than typical schemas, a complication traced back to a specific production stability incident in a European region during rollout and resolved through targeted snapshot-construction optimizations for high-label-count schemas specifically, not a general architectural change.

## 4. Results

## *4.1 Schema-Fetch Latency and Throughput Scaling*

The in-memory snapshot-distribution architecture cut schema-fetch latency from approximately 240 milliseconds to approximately 40 milliseconds (author's primary research data), and scaled metadata access capacity from hundreds to thousands of requests per second (author's primary research data). This magnitude of improvement lines up with the throughput ceiling in-memory, multi-versioned architectures show more broadly: Alhomssi and Leis report their storage engine sustaining roughly two million transactions per second on a sixty-four-core server even under concurrent analytical scan load, which suggests the theoretical throughput headroom of in-memory, snapshot-isolated architectures runs well past what this deployment's metadata workload alone demanded [8]. The comparison baseline's ceiling shows up in Shi and colleagues' production Redis figures, where peak operation volumes reaching 200,000 per second still required real cluster-level operational management to sustain, underscoring that even a well-run conventional cache carries operational weight that in-memory snapshot serving, once loaded, simply does not [4].

*Table 1. Safeguard Matrix: Failure Modes and Corresponding Safeguard Mechanisms*

| Failure Mode | Safeguard Mechanism | Observable Behavior |
|---|---|---|
| Stale snapshot (age exceeds bound) | Maximum acceptable snapshot age enforced per consumer | Consumer flags its own data as degraded rather than silently serving arbitrarily old metadata |
| Failed snapshot build | Validation check on incoming snapshot before load | Consumer continues serving last known-good version rather than falling back to empty or partial state |
| Consumer cold start | Synchronous load of most recent valid snapshot before accepting traffic | No window in which a newly started node serves requests against incomplete metadata |
| High-label-count schema snapshot delay | Targeted snapshot-construction optimization for high-label-count schemas | Snapshot construction time returns to typical range; isolated fix, not a general architectural change |

*Source: Author's production architecture (Section 3.2-3.3).*

## *4.2 Primary-Store Load Reduction and Operational Stability*

Pulling per-request metadata fetches out of the primary store's query path reduced direct load against that store correspondingly (author's primary research data), consistent with the architectural intent of decoupling metadata read volume from primary-store capacity entirely. This reduction in coupling is the mechanism behind the latency and throughput gains above, not a side effect of them. Once metadata reads stop competing with the primary store's other query and write traffic, their latency stops being a function of that store's instantaneous load. Ghandour, El Kafhali, and El Mir's work on queuing-theoretic overload prediction supports treating this kind of load decoupling as a first-order reliability concern rather than an incidental optimization; their framework shows overload conditions in shared-resource systems are predictable in advance when load sources are properly modeled, and removing an entire class of load source, as this architecture does for metadata reads against the primary store, makes that prediction problem considerably easier [19]. Zhang and colleagues' cost analysis of periodically replicated systems reinforces the same point from a scheduling angle, arguing that the interval at which replicated state is resynchronized should itself be modeled and optimized rather than fixed arbitrarily, precisely the logic that governs how this architecture's own snapshot-refresh cadence is set [20].

Figure 1. Producer-Consumer Metadata Snapshot Distribution Pipeline

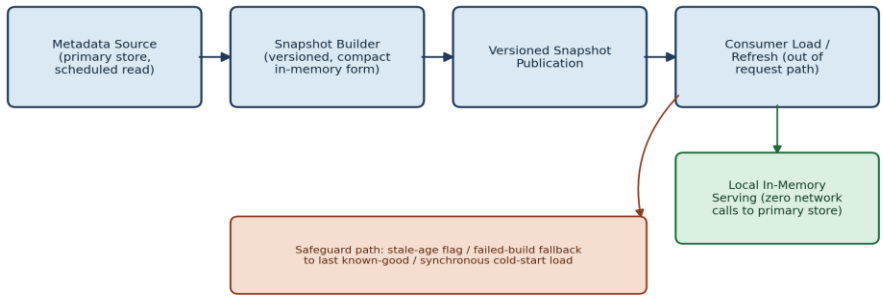



*Figure 1. Producer-consumer metadata snapshot distribution pipeline, with safeguard path for stale/failed-snapshot handling. Author's system architecture.*

Figure 2. Schema-Fetch Latency and Throughput: Author's Production Deployment

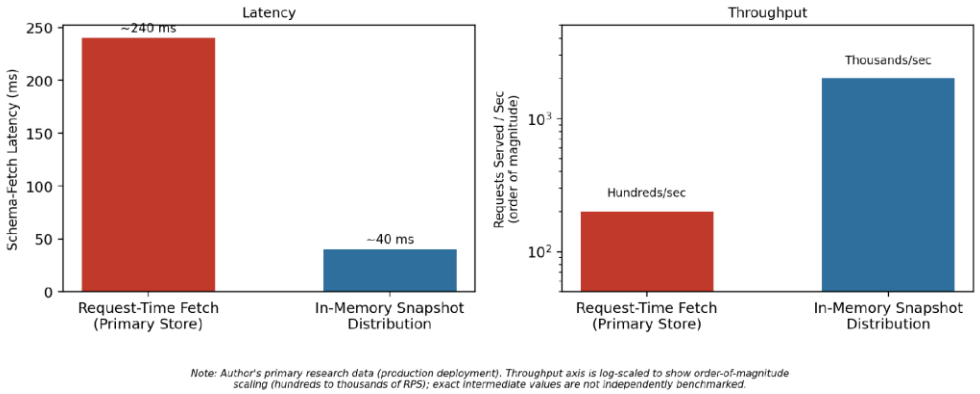



*Figure 2. Schema-fetch latency and throughput, request-time-fetch baseline vs. in-memory snapshot-distribution architecture. Author's production deployment data.*

## 5. Discussion

The central finding, that decoupling metadata freshness from metadata availability produces gains beyond what either horizontal scaling or conventional caching achieves here, carries an implication the caching and scaling literature doesn't fully engage: the right architecture for a read-heavy workload should depend on whether the underlying data changes continuously or in discrete, versionable increments, not just on its read-to-write ratio. A conventional cache suits data that changes unpredictably and continuously, where a time-to-live bound is the best guarantee realistically available. Metadata changes in discrete events instead, schema additions, field revisions, sandbox provisioning, that can be captured, versioned, and distributed as complete units. Treat metadata like generic cacheable data and you discard that structure, settling for a staleness model, cache eviction timing, that fits the data's actual change pattern worse than an explicit, versioned snapshot schedule would.

The safeguard design in Section 3.2 carries a broader implication for how staleness gets reasoned about in systems built this way. Rather than treating staleness as an undesirable side effect to minimize indiscriminately, the architecture treats it as an explicit, bounded, monitored parameter. A consumer degrades its own reported freshness rather than silently serving indefinitely stale data. A failed snapshot build fails safe to the last known-good state rather than failing open to an empty one. This approach differs meaningfully from how conventional caching typically

handles staleness, where an expired entry either blocks on a synchronous refetch, reintroducing the exact latency the cache exists to avoid, or serves stale data with no explicit staleness signal attached at all.

One genuine limitation constrains how far this generalizes. The snapshot-construction anomaly for high-label-count schemas, described in Section 3.3, shows the architecture's benefits depend on snapshot construction staying fast relative to the platform's tolerance for staleness. A metadata source whose structure grows pathologically complex in specific edge cases can degrade snapshot freshness in ways a request-time cache, which fetches reactively rather than proactively, would not experience the same way. That's a manageable constraint where such edge cases stay rare and identifiable, which this deployment's experience suggests, but it would need re-examination in environments where metadata complexity growth is less predictable. Cold-start behavior for a newly provisioned metadata source, where no prior valid snapshot exists to fall back on, remains a related and only partially resolved constraint.

There's a broader question underneath this specific deployment: whether a given read-latency problem calls for scaling the store, adding a cache, or restructuring how the data gets distributed in the first place. These findings suggest metadata, and by extension other high-read, low-write, versioned data categories, are systematically underserved by reaching straight for the first two options.

## 6. Conclusions

In-memory, producer-consumer snapshot distribution beats both horizontal database scaling and conventional request-time caching for metadata specifically, because it is the only one of the three architecturally matched to what metadata actually does: high read frequency, low write frequency, discrete and versionable updates. The production deployment's latency drop from approximately 240 milliseconds to approximately 40 milliseconds, and its throughput scaling from hundreds to thousands of requests per second, show that architectural fit translating into gains that are substantial, not marginal. The safeguard design, treating staleness as an explicit, bounded, monitored parameter rather than an incidental cache-eviction side effect, offers a template that should generalize past this one platform to other metadata-heavy systems under similar latency and scaling pressure. What this article offers the field is a sharper vocabulary for deciding when metadata earns its own architectural treatment, rather than inheriting whatever caching or scaling pattern a team already happens to have lying around for general application data.

**REFERENCES**


1. S. Gilbert, N. A. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services," ACM SIGACT News, vol. 33, no. 2, pp. 51-59, 2002.
2. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," ACM SIGOPS Operating Systems Review, vol. 41, no. 6, pp. 205-220, 2007.
3. W. Vogels, "Eventually Consistent," Communications of the ACM, vol. 52, no. 1, pp. 40-44, 2009.
4. L. Shi, H. Qiao, C. Yang, Y. Jiang, K. Yu, C. Chen, "Research and Application of Distributed Cache Based on Redis," Journal of Software, vol. 19, no. 1, pp. 1-8, 2024.
5. B. Jafari, P. Bayat, "Performance Improvement of Distributed Cache Using Middleware Session," The Journal of Supercomputing, vol. 80, no. 8, pp. 10818-10862, 2024.
6. K. M. Chandy, L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," ACM Transactions on Computer Systems, vol. 3, no. 1, pp. 63-75, 1985.
7. Y. Sun, G. E. Blelloch, W. S. Lim, A. Pavlo, "On Supporting Efficient Snapshot Isolation for Hybrid Workloads with Multi-Versioned Indexes," Proceedings of the VLDB Endowment, vol. 13, no. 2, pp. 211-225, 2019/2020.
8. A. Alhomssi, V. Leis, "Scalable and Robust Snapshot Isolation for High-Performance Storage Engines," Proceedings of the VLDB Endowment, vol. 16, no. 6, pp. 1426-1438, 2023.
9. E. Longo, A. E. C. Redondi, "Design and Implementation of an Advanced MQTT Broker for Distributed Pub/Sub Scenarios," Computer Networks, vol. 224, art. 109601, 2023.
10. P. Sawadogo, J. Darmont, "On Data Lake Architectures and Metadata Management," Journal of Intelligent Information Systems, vol. 56, no. 1, pp. 97-120, 2021.
11. Z. Brahmia, F. Grandi, B. Oliboni, "Schema Versioning in Databases: A Literature Review," Computing Open, vol. 2, art. 2430002, 2024.
12. Z. Brahmia, F. Grandi, B. Oliboni, "A Literature Review on Schema Evolution in Databases," Computing Open, vol. 2, art. 2430001, 2024.

13. M. Overeem, M. Spoor, S. Jansen, S. Brinkkemper, "An Empirical Characterization of Event Sourced Systems and Their Schema Evolution: Lessons from Industry," Journal of Systems and Software, vol. 178, art. 110970, 2021.
14. A. Zbarcea, C. Tudose, A. Boicea, "Extending the Migration from Asynchronous to Reactive Programming in Java: A Performance Analysis of Caching, CPU-Bound, and Blocking Scenarios," Applied Sciences, vol. 16, no. 1, art. 90, 2026.
15. H. Li, X. Guo, "Performance Analysis of Web Server Side Reactive Programming," in 6GN for Future Wireless Networks, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 554, pp. 96-108, 2024.
16. E. Punithavathy, N. Priya, "Auto Retry Circuit Breaker for Enhanced Performance in Microservice Applications," International Journal of Electrical and Computer Engineering, vol. 14, no. 2, pp. 2274-2281, 2024.
17. M. Z. Ain, R. Ardiansyah, S. A. Pratama, M. Akbar, N. T. Lapatta, "Comparative Performance Analysis of GRPC and Rest API Under Various Traffic Conditions and Data Sizes Using a Quantitative Approach," Journal of Applied Informatics and Computing, vol. 9, no. 2, 2025.
18. P. Negi, L. Bindschaedler, M. Alizadeh, T. Kraska, J. Leeka, A. Gruenheid, M. Interlandi, "Unshackling Database Benchmarking from Synthetic Workloads," 2023 IEEE 39th International Conference on Data Engineering (ICDE), pp. 3659-3662, 2023.
19. O. Ghandour, S. El Kafhali, I. El Mir, "Scalable Overload Prediction in Cloud Computing Using a Hybrid Queuing-Theoretic and Machine Learning Framework," Computing, vol. 107, no. 12, art. 231, 2025.
20. C. Zhang, L. Wang, L. Xiao, S. Jiang, M. Han, J. Wang, B. Wei, G. Qin, "Minimizing the Cost of Periodically Replicated Systems via Model and Quantitative Analysis," Frontiers of Computer Science, vol. 18, no. 5, art. 185206, 2024.