IJSES
Volume 9, Issue 8, pp. 63-68, 2025.
International Journal of Scientific Engineering and Science
ISSN (Online): 2456-7361

Approaches to Reducing REST API Response Time in High-Traffic Banking Systems

Rushikesh Anantrao Deshpande
Sr IT Developer, First Horizon Bank
Memphis, TN, USA

Abstract— *This article examines a comprehensive approach to reducing REST API response time in high-load banking systems, based on the synergy of architectural, protocol, and application optimizations. The study aims to identify and comparatively analyze effective methods for reducing the latency of critical API endpoints under request volumes measured in hundreds of millions per month, to achieve acceptable p50 and p95 response-time metrics within fractions of a second. The relevance of this work stems from the high sensitivity of banking-service users to delays: even a 0.1-second improvement increases conversion in transactional scenarios by 8–10%, while delays exceeding one second erode trust and may lead to transaction abandonment. The novelty of the research lies in the integration of results from more than twenty sources, including recent experiments on virtual threads (JEP 444), analysis of microservice patterns Circuit Breaker and Bulkhead, protocol optimizations HTTP/2 and ALPN, as well as the application of hybrid sharding strategies and CQRS architecture in banking platforms. A three-tier caching model is proposed, comprising edge-CDN, in-memory storage, and conditional HTTP requests, which complements protocol and architectural measures. The main findings demonstrate that migrating from a monolith to a microservice topology combined with an API Gateway reduces median and tail latency through parallel processing and rapid failure on component faults; protocol techniques HTTP/2, Keep-Alive, gzip and Brotli bring transmission time close to the physical limits of the network; multilevel caching minimizes network hops; and database optimization via PgBouncer connection pooling, indexing, replication, sharding and non-blocking I/O maintains p95 and p99 latency at single- to tens-of-milliseconds levels. This article will be beneficial for engineers and architects of banking IT systems, application performance specialists, and researchers of distributed services.*

Keywords— *REST API, response time, banking systems, microservices, HTTP/2, caching.*

I. Introduction

Digital channels have become the central point of interaction between banks and their clients; thus, REST API latency directly influences perceived service quality and, consequently, competitiveness. User sessions in mobile and web applications generate a continuous stream of requests. In large European banks, their volume is measured in hundreds of millions of calls per month, and for balance-check or payment-initiation operations, the permissible response time must not exceed fractions of a second. Industry-wide research confirms the insatiable demand for speed: improving page-load time by just 0.1 seconds raises conversion in transactional scenarios by 8–10%, while any delay above one second noticeably diminishes user focus and increases the likelihood of transaction abandonment [1]. Although these data derive from the e-commerce sector, latency sensitivity is even greater in banking, since waiting is associated with financial risk and brand trust.

Actual open-banking metrics reveal that even leading UK providers strive to keep the average response time of critical endpoints within a narrow range; yet, specific calls, such as account statement retrieval, still exceed two seconds [2]. Such figures illustrate that the race for milliseconds remains critical: regulatory reports publish comparative tables, and each additional delay instantly becomes a public performance indicator for the bank.

Within organizations, the importance of the issue is reflected in the level of investment. According to a New Relic report, over 55% of financial firms have already implemented APM systems, and 36% cite customer-experience improvement as the primary motivation for enhancing observability [3]. These solutions do more than record p95 latency; they provide the basis for SLO discipline, enabling the alignment of performance with business metrics such as transaction resilience and user retention.

Thus, low REST API latency in banking systems is not an abstract characteristic, but a concrete factor that determines client trust, compliance with regulatory requirements, and the bank's ability to monetize digital channels.

II. Materials and Methodology

This study is based on the analysis of 22 sources, including academic articles, industry reports, technical specifications, and results from public benchmarks. The theoretical foundation comprises works on microservice architecture and resilience patterns: Ramu's research [4] and the experiments by Meijer et al. [6] demonstrated reductions in median and p99 latency when migrating from a monolith to microservices; the HTTP/2 analysis in Ramadan and Mohamed [10] and RFC 9113 [11] described gains from multiplexing and header compression; JEP 444 recommendations [20] revealed the potential of virtual threads to serve millions of concurrent connections with minimal overhead. As starting points for business metrics and SLO targets for p50 and p95 latency in transactional scenarios, industry reports from Think with Google [1], Open Banking [2], and New Relic [3] were utilized.

Methodologically, the research combined the following components: comparative analysis of architectural patterns—contrasting monoliths and microservice topologies with assessments of internal contention and network hops [4,6]; protocol analysis—measuring TCP/TLS overhead based on Mad Packets data [9], evaluating HTTP/1.1 Keep-Alive, ALPN and HTTP/2 migration according to Ramadan and Mohamed

[10] and RFC 9113 [11]; compression and JSON-format evaluation—comparing gzip and Brotli in the Verdigris infrastructure [7,8]; connection-management study—analysis of PgBouncer pool performance and its impact on connection-establishment time from AWS RDS data [17]; multilevel caching—comparison of Cloudflare edge-CDN [12], Redis in-memory cache [13] and HTTP caching with Cache-Control, ETag and Last-Modified headers [14,15]; database optimization—testing connection pools and indexing per MoldStud [16], evaluating replication and Multi-AZ AWS RDS topologies [17,18], and experimentally comparing sharding strategies [19]; resilience-pattern assessment for Circuit Breaker [5] and Bulkhead [6]; and the study of asynchronous and batch approaches—JEP 444 virtual threads [20], Kafka batching [21] and CQRS with Event Sourcing [22].

III. Results and Discussion

Architectural decisions exert the greatest influence on latency because they set the minimum possible time for a request to traverse the system. Migrating to microservice segmentation decomposes the monolith into independent functional domains, thereby removing lok-al contention on memory and transaction locks, yet introducing interservice interaction. A 2023 empirical study confirms that, with appropriate protocol selection and service-discovery schema, a microservice topology reduces median response time by almost 25% compared to an equivalent monolith due to parallel elasticity and improved failure isolation, while the increase in network hops is offset by reduced internal contention and the ability to scale critical services selectively [4]. Domain boundaries should follow the single-responsibility principle: for example, the payment service should not contain credit-scoring logic, so updating the latter does not entail redeploying the entire container ensemble.

To consolidate heterogeneous calls and reduce redundant RTTs, an API Gateway is deployed above the microservice suite, performing terminal authentication, response aggregation, and dynamic route rewriting. Intelligent routing further enhances this: when a specific service degrades, the gateway can temporarily divert a portion of traffic to a replica with a lower consistency SLA or return a cached response, thereby preserving perceived responsiveness.

Even with thoughtful routing, individual components may unexpectedly slow down, making the Circuit Breaker pattern indispensable. The algorithm measures the proportion of errors or timeouts over a short window and, upon exceeding a threshold, opens the circuit to return a predictable error code in micro-constant time, preventing the frontend from waiting indefinitely. Microsoft Azure documentation reports that systems employing active Circuit Breakers observe reductions in p99 tail latency from hundreds of milliseconds to tens, owing to the elimination of cascading retries [5]. For further fault isolation, the Bulkhead pattern is applied, where each critical service is allocated its own thread and connection pool, ensuring that the blockage of one domain does not deplete resources from others. An experimental comparison of resilience architectures revealed that Bulkhead isolation maintains overall pool CPU utilization above 85% under load without a significant increase in latency. In contrast, a shared pool exhibits exponential timeout growth beyond the same threshold [6].

The combined effect is synergistic: microservices deliver scalability, the gateway minimizes network transactions and enables fine-grained traffic control, and Circuit Breaker, together with Bulkhead, transforms unpredictable failures into fast, localized rejections.

Low-level HTTP techniques constitute the next layer of optimization, building on the above microservice architecture and gateway level, as it is at the protocol boundary that the total time spent transmitting the payload between the client and the banking core is determined. The first and most accessible measure is response-body compression and judicious JSON structuring. Measurements for public REST endpoints indicate that enabling gzip reduces typical financial payload sizes by 60–80%, and migrating to Brotli yields an additional approximately 20% improvement [7]. In Verdigris's production infrastructure, enforcing the Accept-Encoding: gzip header resulted in a traffic reduction of 30–70% without altering server-side computation, directly lowering the median load latency of extensive reports by tens of milliseconds, thanks to shorter packet transmission times over the operator's network [8]. Combined with the omission of null fields and the replacement of verbose property names with concise aliases, this technique brings response time closer to the network speed lower bound without affecting business logic.

The next source of latency is connection establishment itself. Under HTTP/1.1, each new TCP + TLS session incurs two to three full RTTs, which in mobile networks can easily amount to hundreds of milliseconds. A field trace published by MadPackets records that, with a baseline RTT of 68 ms, the three-way handshake overhead is approximately 158 ms, and this overhead grows proportionally with longer routes [9]. Therefore, the mandatory use of HTTP Keep-Alive and a well-tuned client-side connection pool are critical. By paying the handshake cost once, the application reuses the same socket and avoids both TCP slow-start and repeated certificate verification. On the server side, the pool size must be sufficient, with separate limits for internal services to prevent artificial queue saturation and thus avoid growth in p99 latency.

Even when using persistent sockets, HTTP/1.1 remains susceptible to head-of-line blocking; therefore, migrating to HTTP/2 provides a further order of optimization. Thanks to multiplexing multiple logical streams over a single TCP channel and header compression via HPACK, the need for sharding and domain-specific sprite domains is eliminated. In contrast, a single TLS tunnel reduces the total number of handshakes. An eleven-month observation of one million websites showed that among 80% of those who enabled HTTP/2, page-load times decreased, with gains more pronounced under mobile-network conditions; in a laboratory demonstration, sequential loading of two hundred small objects was 6.1 times faster than HTTP/1.1, due to concurrent frame retrieval without additional pipeline blocking [10]. For banking REST APIs, the effect manifests not only in a median reduction; the tail of the distribution is more significant, as stream

parallelism smoothes latency spikes that arise during the encryption of extensive reports or when servicing slow clients.

Optimization is achieved by the proper use of ALPN within TLS 1.2 or 1.3: the client and server negotiate the application-layer protocol during the initial handshake phase, rather than via a separate Upgrade, thereby eliminating another network round-trip, which is particularly notable for intercontinental transactions [11]. Taken together, these techniques yield double-digit reductions in both average and p95 latency without major changes to business code, and they establish a robust foundation for subsequent caching and scaling strategies.

Caching moves data closer to the point of consumption, thereby eliminating entire segments of the network path and complementing the aforementioned architectural and protocol measures. At the edge layer, banks use CDNs with points of presence in dozens of countries, where static and pseudo-dynamic responses are stored for seconds or minutes. A practical measurement by a cloud provider found that, after enabling edge caching, the average time to serve a request from the nearest node fell to twenty-three milliseconds, while origin-server requests dropped by sixty-five percent, yielding a sixty-percent reduction in outbound traffic [12]. For banks handling personal data, such a scheme requires encryption at the perimeter and a strict TTL policy; however, the benefits of reduced long-distance RTTs and offloaded internal channels outweigh the overhead of key management.

Within the cluster, the principal tool is an in-memory cache with an LRU or LFU eviction policy. Benchmark results publication indicated that a single Redis instance on a cloud node can handle 1.2 million operations per second with latency under one millisecond [13]. Such capacity enables the storage of hot account records, authentication profiles, and complex-computation results with minimal delay. A cache-aside strategy ensures consistency: the application first attempts to read from the cache; upon a miss, it queries the database and asynchronously updates the cache key. It is crucial to segregate keyspaces by domain, so that high-frequency balance checks do not evict rarely updated credit-limit entries.

Even with edge and in-memory caches present, browsers and intermediate proxies continue to request resources; hence, correct HTTP headers remain necessary. Standard use of Cache-Control, ETag, and Last-Modified allows returning 304 Not Modified responses with no response body, thereby saving bandwidth and CPU cycles. A proxy-server traffic study showed that twenty to thirty percent of all requests are conditional GETs ending with 304 codes [14]. An HTTP caching best-practices guide emphasizes that, when properly configured, such responses eliminate payload transmission, leaving only compact headers, as illustrated in Figure 1 [15].

For banking APIs, this is especially critical when serving mobile clients, where each kilobyte increases both data-transfer cost and wait time, and retransmitting unchanged data is counterproductive.

Thus, a three-tier model—comprising a global edge cache, an in-cluster in-memory store, and correctly configured HTTP headers—forms a sequential defense against latency. It minimizes network hops, unloads the database, and reduces transmitted bytes without risking the integrity of financial operations.

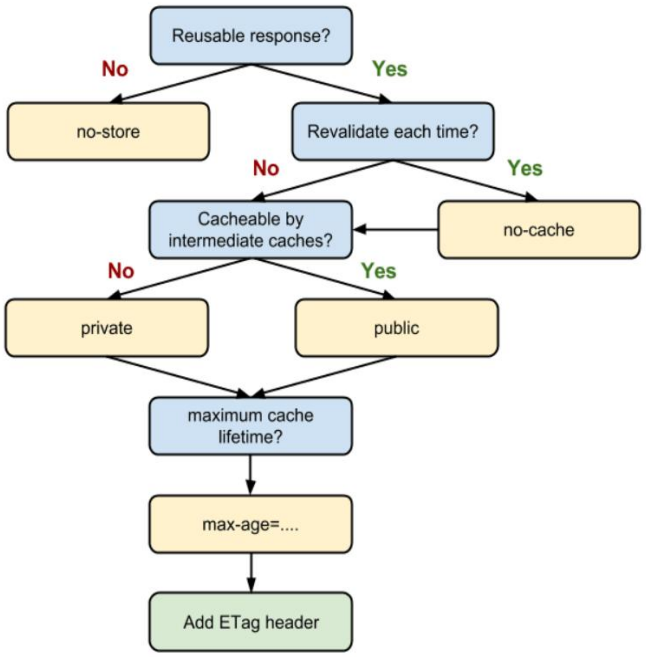

Fig. 1. The decision process for setting your Cache-Control headers [15]

Once caching has shortened the network path, the database becomes the bottleneck; hence, minimizing its response time determines whether the system can maintain the target budget under peak traffic. The first step is to minimize the costs associated with establishing connections. A dedicated proxy, such as PgBouncer, maintains a persistent pool and redistributes queries over existing channels. This demonstrates that even without changes to query logic, a well-configured pool yields immediate gains, and pool saturation serves as an early indicator that application instances are insufficient.

The next optimization level concerns the data schema itself. Analysis of large request samples shows that appropriately chosen indexes reduce execution time by up to 70% due to decreased random I/O and a shift from full-table scans to point-lookup reads [16]. The effect is amplified when an index covers the query: the engine satisfies the request without accessing the base table, thus eliminating extra B-tree traversals and saving tens of milliseconds per access. To preserve benefits under increasing load, index structures should be reviewed against real traffic profiles, and unnecessary secondary indexes, which slow down writes, should be minimized.

AWS documentation records direct savings: routing read-only traffic away from the master node removes a significant portion of load and enables horizontal scaling while remaining within the same SLA, as shown in Fig. 2 [17].

Practical reports confirm that a correct replica configuration reduces the proportion of operations hitting the primary instance to 30%. Further, decreases the median read latency by 30–50% compared to a master-only setup [18]. Asynchronous replication requires lag monitoring; however, in typical

IJSES
Volume 9, Issue 8, pp. 63-68, 2025.
International Journal of Scientific Engineering and Science
ISSN (Online): 2456-7361

financial scenarios, absolute consistency is only needed at write time.

Finally, when even a vertically scaled database and a replica pool hit hardware limits, data must be partitioned into independent fragments. A recent comparative experiment demonstrated that adaptive sharding, utilizing a hybrid of range and hash parameters, maintains a response time of approximately 90 ms under dynamic load. In contrast, classic range-based distribution reaches 135 ms under the same volume, and uniform hashing exceeds 180 ms, as illustrated in Fig. 3 [19].

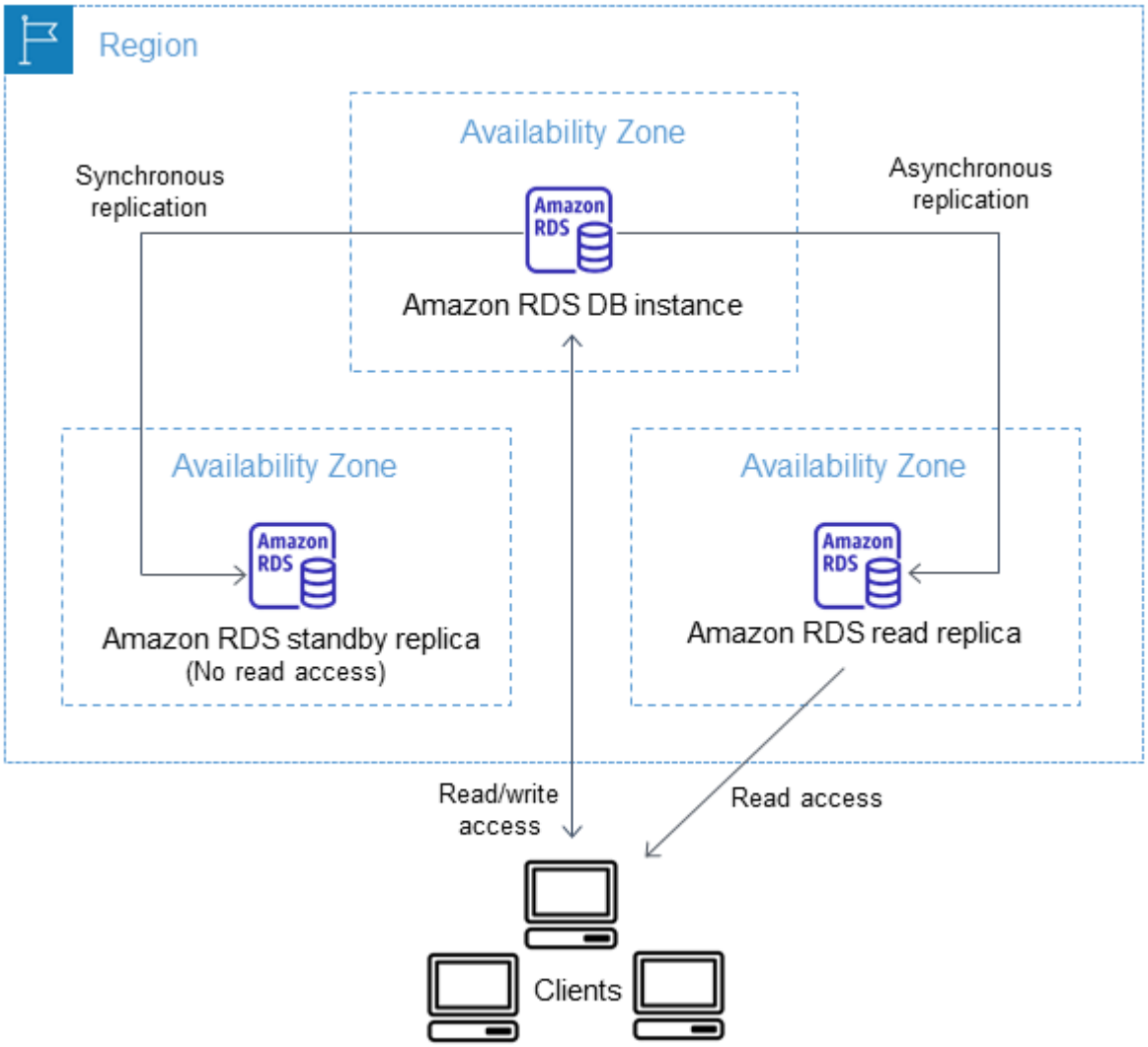

Fig. 2. Multi-AZ RDS Replication Topology [17]

Transitioning from an optimized database to asynchronous and parallel patterns eliminates residual delays that occur under concurrent access, when load is so high that even replication and sharding periodically create queues. The primary measure is to switch I/O to non-blocking mode, so that compute threads never idle while awaiting disk or network access. In recent Java releases, this is implemented as virtual threads: a single server process can maintain millions of concurrent connections on commodity hardware, since each virtual thread occupies roughly 300 bytes and is not bound to an OS kernel thread [20]. Such a scale reduces p95 latency to single-digit milliseconds, because the overhead of thread creation and context switching becomes statistically negligible.

When many requests are aggregated into homogeneous batches, batch processing is applied. The idea is simple: group several operations and send them into the cluster as one transport unit, thereby amortizing network and cryptographic overhead. Basic Kafka experiments demonstrate that with a batch size of 200 messages, producer throughput reaches 50 MB/s, whereas with batching disabled, throughput falls by an order of magnitude [21]. In banking REST APIs, this technique is helpful for tasks such as mass balance checks or commission calculations, where in-memory grouping time (a few milliseconds) is less than the cumulative time required to establish hundreds of separate HTTP connections.

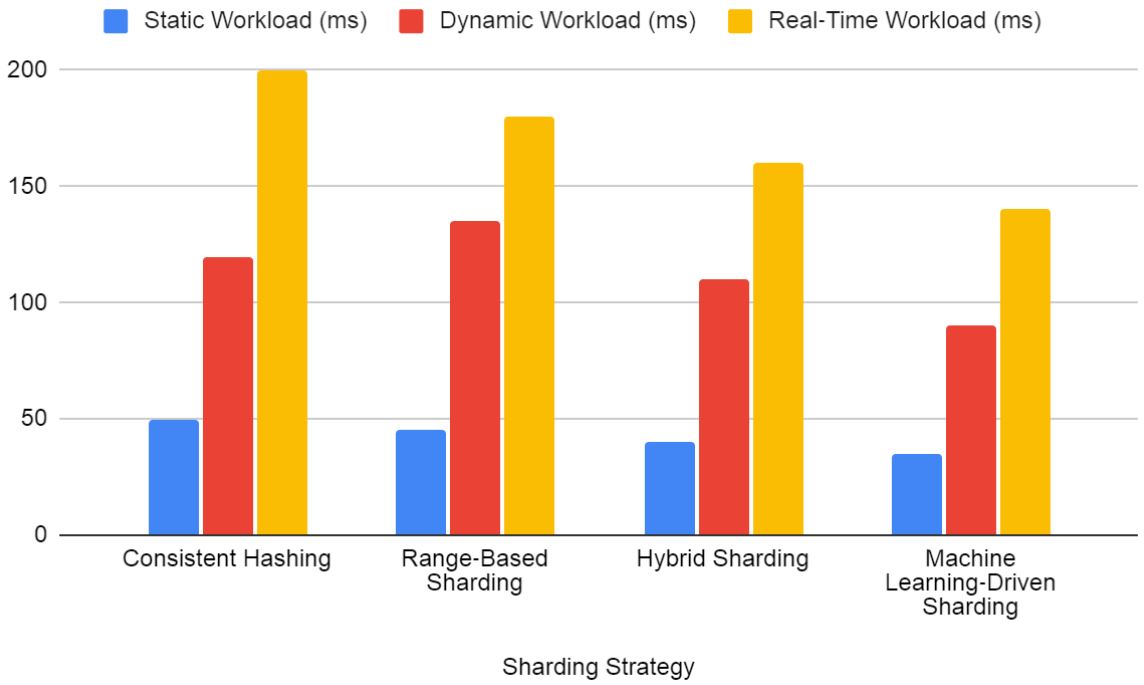

Fig. 3. Query Response Time Across Sharding Strategies [19]

For scenarios in which reads vastly outnumber writes and a flexible data model is required, the architecture is augmented with a separation of commands and queries. In the CQRS pattern, a command is recorded as an immutable event, and materialized projections serve as the source of reads; thus, writes complete instantly—merely appending a record to the log—while reads scale independently, as forecasts can be replicated horizontally. A report on implementing CQRS + Event Sourcing in an instant-payment system demonstrates that such a scheme can withstand high throughput while maintaining stable low latency, thanks to the absence of locks and the complete isolation of the read model from the write model [22]. Simultaneously, each domain step generates its event, simplifying audit and accelerating recovery from failures by replaying the log.

The combination of non-blocking I/O, batch pipelines, and CQRS converts remaining sequential segments into parallel ones, so the peak load is now distributed across many lightweight tasks, and each computation executes precisely when its data are ready. This closes the optimization loop: after network-, cache-, and database-level improvements, the system gains additional resilience, maintaining REST API responses within budget even under sudden multiple surges in client traffic.

Banking platforms benefit when technological solutions are designed in tandem with regulatory requirements. When selecting an architecture of microservices and asynchronous patterns, it is essential to incorporate data governance controls from the outset, as national personal information laws regulate data protection. Encryption key registries should reside in the same region as the client; otherwise, any delay in the decryption process negates the gains from an optimized request path.

Performance is inseparable from reliability; therefore, scaling plans must be accompanied by continuous testing procedures. Load tests should be run on production-like clones using realistic scenarios to observe not only average speed but also tail latency behavior under peak traffic. Identified bottlenecks should be addressed in small increments, employing blue-green deployment strategies to avoid downtime and enable rapid rollback without customer impact.

Every optimization point must be measured by business-understandable metrics: payment completion time, proportion of successfully served requests, and time to recovery after failure. Target values are assigned to these metrics, and any breach automatically triggers root-cause analysis. A blameless-postmortem culture helps infrastructure and development teams jointly resolve the underlying cause without assigning fault.

Fine-tuning of caching and sharding mechanisms yields the maximum effect only with an up-to-date request profile. Thus, access logs should be aggregated and analyzed daily to evict seldom-accessed keys and to rebuild indexes according to actual read frequencies. Continuous monitoring of schema changes enables the pre-scaling of hot shards, thereby preventing latency growth.

Security remains a fundamental prerequisite. Every optimization is vetted against least-privilege policies, and token and certificate lifetimes are aligned with the stated risk level. In such a framework, even sub-second operations occur under strict control, without opening additional attack surfaces. Consequently, the bank obtains a platform that combines low latency, resilience to load surges, and regulatory compliance, providing a reliable foundation for further deployment of digital services.

IV. Conclusion

The analysis demonstrates that a comprehensive approach to reducing REST API response time in high-load banking systems must be based on the synergy of architectural, protocol, and application-level optimizations. First, migrating from a monolith to a microservice topology, along with the implementation of an API Gateway and the Circuit Breaker and Bulkhead patterns, ensures scalability, failure isolation, and predictable system behavior under load, enabling reductions in median and tail latency through parallel processing and rapid failure handling.

Second, at the network level, the adoption of HTTP/2 and ALPN within TLS, the use of persistent HTTP Keep-Alive connections, and judicious response compression (gzip, Brotli) alongside JSON-format optimization allow data transmission time to approach physical network limits, reducing RTTs and minimizing handshake and header transport overhead.

The third optimization tier is multilevel caching, which includes edge-CDN for geographically distributed clients, an in-cluster in-memory cache (Redis with LRU/LFU), and conditional requests with properly configured HTTP headers (Cache-Control, ETag, Last-Modified). It lowers the load from the original services. It decreases traffic volume. It ensures quick response time by delivering data at the closest point of presence.

The fourth layer of enhancement involves databases, comprising connection pooling (PgBouncer), optimization of data structures and indexes, horizontal replication and multi-zone topologies, hybrid strategy sharding, and a transition to non-blocking I/O with virtual threads. All these factors, in conjunction, help maintain extremely high peak loads, with p95 and p99 latency staying at single or double-digit milliseconds.

Collect and analyze access logs, execute load tests, and configure SLO discipline based on business metrics, as well as regulatory and security requirements. Combine technical optimizations with process-driven quality and reliability practices to systematically ensure fast and resilient REST API responses from a banking digital platform that sustains trust at the highest levels of business, while also promoting competitiveness.

References

[1] "Milliseconds make Millions," *Think with Google*, 2020. https://www.thinkwithgoogle.com/_qs/documents/9757/Milliseconds_Make_Millions_report_hQYAbZJ.pdf (accessed Jun. 26, 2025).

[2] "API performance stats," *Open Banking*, 2025. https://www.openbanking.org.uk/api-performance/ (accessed Jun. 27, 2025).

[3] New Relic, "State of Observability for Financial Services and Insurance," New Relic, 2025. Accessed: Jul. 26, 2025. [Online]. Available: https://newrelic.com/sites/default/files/2025-04/new-relic-state-of-observability-fsi-2025_0425.pdf

[4] V. B. Ramu, "Performance Impact of Microservices Architecture," *The Review of Contemporary Scientific and Academic Studies*, vol. 3, no. 6, Jun. 2023, doi: https://doi.org/10.55454/rcsas.3.06.2023.010.

[5] Azure, "Circuit Breaker pattern," *Azure*, 2025. https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker (accessed Jun. 28, 2025).

[6] W. Meijer, C. Trubiani, and A. Aleti, "Experimental evaluation of architectural software performance design patterns in microservices," *Journal of Systems and Software*, vol. 218, p. 112183, Dec. 2024, doi: https://doi.org/10.1016/j.jss.2024.112183.

[7] MoldStud, "Designing API Payloads for Optimal Performance and Efficiency," *MoldStud*, 2024. https://moldstud.com/articles/p-designing-api-payloads-for-performance-and-efficiency (accessed Jun. 29, 2025).

[8] "API performance improvements," *Verdigris*, 2023. https://support.verdigris.co/en/articles/8746973-december-12-2023-api-performance-improvements (accessed Jun. 30, 2025).

[9] Mad Packets, "TCP 3-Way Handshake Delay Calculation," *Mad Packets*, May 05, 2020. https://madpackets.com/2020/05/04/tcp-3-way-handshake-delay-calculation/ (accessed Jul. 01, 2025).

[10] N. Ramadan and I. Mohamed, "Impact of Implementing HTTP/2 in Web Services," *International Journal of Computer Applications*, vol. 147, no. 9, pp. 27–32, Aug. 2016, doi: https://doi.org/10.5120/ijca2016911182.

[11] M. Thomson and C. Benfield, "RFC 9113: HTTP/2," *IETF Datatracker*, Jun. 2022. https://datatracker.ietf.org/doc/html/rfc9113?utm_source=chatgpt.com (accessed Jul. 02, 2025).

[12] "Cloudflare CDN," *Cloudflare*. https://www.cloudflare.com/resources/images/slt3lc6tev37/6mpY5zjlPP3b2twznUQOki/b221bc7b49723deb2c09a4b30747f685/cloudflare-cdn-whitepaper-19Q4.pdf (accessed Jul. 05, 2025).

[13] Redis, "Redis Labs Breaks Database Transaction and Latency Performance over Cloud Instance," *Redis*, 2024. https://redis.io/press/redis-labs-breaks-database-transaction-and-latency-performance-over-cloud-instance/ (accessed Jul. 06, 2025).

[14] P. Cao, "Characterization of Web Proxy Traffic and Wisconsin Proxy Benchmark 2.0," *W3*. https://www.w3.org/1998/11/05/WC-workshop/Papers/cao.html (accessed Jul. 07, 2025).

[15] "Prevent unnecessary network requests with the HTTP Cache," *Web dev*, 2018. https://web.dev/articles/http-cache (accessed Jul. 09, 2025).

[16] "Smart Query Design Best Practices," *Moldstud*, 2025. https://moldstud.com/articles/p-smart-query-design-best-practices-optimize-sql-rewrite-for-enhanced-performance (accessed Jul. 10, 2025).

[17] "Working with DB instance read replicas," *AWS*. https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_ReadRepl.html (accessed Jul. 11, 2025).

[18] "Enhancing Read Performance in AWS RDS," *MoldStud*. https://moldstud.com/articles/p-enhancing-read-performance-in-aws-rds-when-to-use-read-replicas-for-optimal-efficiency (accessed Jul. 14, 2025).

[19] S. Jayaraman and D. Borada, "Efficient Data Sharding Techniques for High-Scalability Applications," *Integrated Journal for Research in Arts and Humanities*, vol. 4, no. 6, pp. 323–351, Nov. 2024, doi: https://doi.org/10.55544/ijrah.4.6.25.

[20] "JEP 444: Virtual Threads," *Openjdk*, 2023. https://openjdk.org/jeps/444 (accessed Jul. 14, 2025).

[21] Apache Kafka, "Documentation," *Apache Kafka*, 2024. https://kafka.apache.org/07/documentation.html (accessed Jul. 17, 2025).

[22] "Why is CQRS-ES a good option for instant payments?" *Icon Solutions*. https://iconsolutions.com/wp-content/uploads/2020/12/IPF_Technology_Series_2.pdf (accessed Jul. 19, 2025).