Journal of Animal Environment

(E-ISSN 2717-1396, ISSN 2717-1388)

Zero-Defect Request Coalescing: A Cooperative-Preemptive Userspace Gateway for Cache Stampede Mitigation across Distributed Cloud, Clinical, and Biosystem Infrastructures

1 Pavan Manikanta Raghava Kasturi, 2 Dr. Vithya Ganesan
1,2 Department of Computer Science and Engineering,
Koneru Lakshmaiah Education Foundation, Vijayawada, Andhra Pradesh, India.
1 kasturikpmr@gmail.com, 2 vithyamtech@gmail.com
Corresponding Author: 1 Pavan Manikanta Raghava Kasturi
Email: kasturikpmr@gmail.com

Abstract
Cloud microservices often suffer from database connection-pool exhaustion during sudden traffic spikes, a failure state known as a cache stampede. Traditional mitigation relies heavily on load-shedding, which preserves backend stability by indiscriminately dropping client requests. This paper introduces an application-layer request coalescing gateway that intercepts identical concurrent queries and collapses them into a single-flight database operation. The architecture operates on a single-threaded, cooperative-preemptive event loop and integrates C-based state-machine parsing to completely bypass multi-threaded lock contention. Empirical evaluation across 100,000 synthetic network requests demonstrated a 98.74% mean reduction in backend database queries. Furthermore, the architecture effectively decoupled application throughput from physical socket constraints. Even when host operating system transmission control protocol (TCP) handshake queues degraded to 2,100 milliseconds, the proxy sustained an isolated execution latency between 188 and 599 milliseconds. These results establish userspace coalescing as a highly efficient, non-destructive alternative to traditional edge load-shedding. Crucially, this architecture provides a foundational resilience model for cross-domain infrastructures, securing real-time data access across clinical networks and ensuring uninterrupted telemetry in high-density biosystem environments.
Keywords: Cache stampede, Request coalescing, Edge gateway, Asynchronous processing, Tail latency, Clinical networks, Biosystems, Healthcare IT.

1. Introduction

Distributed cloud architectures rely on microservices that are highly sensitive to sudden, synchronized traffic spikes [1] , a vulnerability that is especially critical in high-stakes environments such as clinical healthcare networks and smart biosystem telemetry. To mitigate this structural vulnerability, traditional edge gateways utilize admission control and load-shedding semaphores [2]. When an application cache expires or fails under heavy concurrent load, thousands of identical client requests bypass the cache layer and simultaneously attempt to query the primary database. This synchronized failure state, known in distributed systems as a cache stampede or thundering herd, rapidly exhausts backend connection pools [3]. The resulting resource contention spikes central processing unit (CPU) utilization and dramatically inflates tail latency, often triggering cascading systemic failures across the broader network infrastructure.

While establishing strict concurrency limits successfully prevents total physical database exhaustion, it achieves this by indiscriminately dropping incoming client connections, violating Service Level Objectives (SLOs). Recent architectural advancements attempt to solve this redundancy proactively through request coalescing [4]. Currently, the most performant coalescing implementations rely on kernel-bypass technologies, such as Extended Berkeley Packet Filter (eBPF) pipelines, which manipulate packets directly at the operating system level [5]. While kernel-space interventions offer superior raw packet-processing speed, they require elevated root privileges, custom memory mapping, and rigid hardware dependencies. This severely restricts their deployment flexibility in abstracted, containerized cloud environments. Similarly, while compiled enterprise gateways written in C++ or Go (e.g., Envoy, Traefik) offer robust native coalescing, deploying them introduces significant architectural and maintenance overhead for engineering teams operating exclusively within Python-based machine learning or Asynchronous Server Gateway Interface (ASGI) ecosystems.

Therefore, a critical research gap exists: achieving high-performance single-flight coalescing without abandoning the portability and ecosystem integration of an application-layer, interpreted runtime environment, which is increasingly utilized in modern clinical and biosensor data platforms. To bridge this gap, this study proposes a zero-defect, cooperative-preemptive userspace gateway. The architecture intercepts identical Hypertext Transfer Protocol (HTTP) targets at the network edge and maps them to shared asynchronous promises, ensuring

***Corresponding Author:**
Received: 25th May, 2026; Revised: 6th July, 2026; Accepted: 28th July, 2026; Available Online: 29th August, 2026
(DOI): 10.70102/AEJ.2026.18.8s.20

only a single pioneer query reaches the backend database. To overcome the traditional string-evaluation bottlenecks of interpreted languages, the system integrates C-based finite state machine protocol parsing alongside a Cython-backed asynchronous event loop.

The primary objective of this research is to empirically demonstrate that this hybrid application-layer proxy can successfully decouple extreme operating system Transmission Control Protocol (TCP) queuing saturation from internal application processing. By evaluating the architecture under sustained, jitter-injected cache stampedes, this paper aims to establish that userspace request coalescing provides a highly portable, energy-efficient [6], and mathematically safe alternative to conventional load-shedding methodologies. Consequently, this evaluation serves as a foundational model for protecting life-critical clinical databases and high-density agricultural biosystems during localized network emergencies without severing active client connections.

2. Materials and Methods

2.1 System Architecture

The coalescing gateway is engineered around the Reactor design pattern, enabling the demultiplexing of highly concurrent network connections within a single, continuous execution thread [7]. Unlike conventional proxy servers that rely on multi-threading, where the operating system allocates a distinct thread for every active client connection, this architecture pins the entire event loop to a single processor core. This structural decision guarantees CPU cache locality, completely eliminates the overhead associated with mutex lock contention, and bypasses the performance degradation inherent in thread context switching [7]. Furthermore, because the in-flight dictionary mapping is fundamentally stateless across distinct request targets, this single-threaded architecture is engineered for horizontal scalability, orchestration frameworks such as Kubernetes can seamlessly replicate the proxy across multi-core clusters or deploy it as a sidecar container without requiring complex inter-process memory synchronization, a highly desirable trait for scalable clinical data centers and extensive biosystem telemetry grids.

To maintain high-throughput concurrency within this isolated thread, the gateway utilizes userspace-directed input/output (I/O) event notification [8]. In this model, the underlying operating system kernel handles physical hardware network interrupts and transfers incoming payload data directly into memory buffers. The application layer polls these buffers iteratively to detect readable sockets rather than relying on standard, latency-heavy kernel interrupts [9,10]. This non-blocking I/O approach allows the software proxy to manage thousands of open TCP pipes simultaneously.

2.1.1 Network Topology and Operational Boundary

To isolate the computational overhead of request coalescing from cryptographic termination, the proposed architecture is strictly designed as a Tier-2, application-layer internal proxy. In a standard cloud-native deployment topology, this gateway operates securely within a Virtual Private Cloud (VPC), situated behind a primary Layer-4/Layer-7 Edge Load Balancer (e.g., an Ingress Controller) [11]. The primary edge load balancer assumes responsibility for Transport Layer Security (TLS 1.3) termination and HTTP/2 binary de-framing, subsequently forwarding sanitized, plaintext HTTP/1.1 traffic to the internal coalescing gateway. By strategically offloading cryptographic handshakes and binary protocol parsing to the external network edge, the single-threaded application loop is preserved strictly for high-velocity request coalescing and asynchronous future resolution. **Figure 1** illustrates this complete network topology and the architectural boundary between the edge load balancer and the internal gateway.

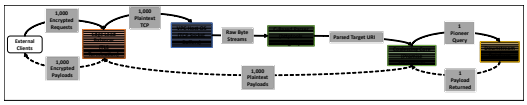


Figure 1. Architecture of the request coalescing gateway within a VPC topology. External encrypted traffic is terminated at the edge load balancer. Plaintext bytes from the host OS TCP socket buffers are then processed via a C-based parser (httptools). Parsed targets enter the coalescing core, where up to 1,000 concurrent requests are collapsed into a single pioneer database query.

2.2 Protocol Parsing and Event Loop Execution

Python's native asynchronous engines often encounter Global Interpreter Lock (GIL) constraints when parsing text-heavy application protocols like HTTP. To achieve execution velocities required for production Linux environments, the proposed architecture is engineered to replace the native Python asynchronous reactor with uvloop, a Cython-compiled binding to the libuv library [12]. This C-based engine manages the underlying I/O multiplexing, mathematically elevating the runtime throughput capacity. Furthermore, to ensure cross-platform resilience, the gateway features an operating-system-aware fallback

mechanism, when deployed in non-Linux environments, it seamlessly defaults to the host's native I/O Completion Ports (IOCP).

Once a socket is registered as readable by the event loop, the incoming byte stream must be rapidly analyzed to identify the specific client request parameters. Utilizing standard regular expressions or native string decoding methods on the critical data path introduces significant computational overhead. The architecture mitigates this by integrating httptools, a C-based Finite State Machine (FSM) parser [13]. The FSM processes incoming network bytes sequentially in $O(N)$time, safely identifying HTTP request boundaries and extracting the target Uniform Resource Identifier (URI) without allocating extraneous string objects in the application memory. The parser strictly enforces HTTP semantics [14] and is configured to support persistent keep-alive connections via an active read-loop, enabling clients to multiplex sequential requests (such as continuous patient monitoring or environmental biosensor streams) over an established TCP pipeline without triggering repetitive three-way handshake penalties [15].

2.3 Single-Flight Coalescing Algorithm

The primary theoretical component of the gateway is the request coalescing engine. As client connections arrive, the FSM parser identifies the specific HTTP method and target URI, concatenating them to form a deterministic string identifier (e.g., GET:/api/data). This identifier represents the exact backend database operation requested by the client. To mathematically defend the proxy against malicious cache-busting vectors (e.g., generating infinite unique keys via randomized query string parameters), the FSM parser strictly evaluates the normalized, base URI path. Dynamic or user-specific query arguments that do not alter the deterministic nature of the backend payload are structurally decoupled from the coalescing dictionary key, ensuring robust deduplication even under adversarial request obfuscation.

The proxy checks a centralized, in-flight dictionary mapping these identifiers to asynchronous Promise objects (Futures). Mathematically, this hash-map architecture guarantees an $O(1)$ average-case time complexity for target lookup and deduplication, ensuring that routing velocity remains constant regardless of dictionary saturation. Furthermore, the space complexity is strictly bounded to $O(C)$, where C represents the volume of concurrent suspended client sockets, preventing exponential memory inflation. The algorithmic workflow of this single-flight mechanism, visually detailed in **Figure 2**, proceeds as follows:

1. If the requested identifier is not present in the dictionary, the proxy generates a new Future, maps it to the identifier, and initiates the physical network call to the downstream database. This client connection is flagged as the pioneer request.
2. If subsequent clients request the identical identifier while the pioneer request remains pending, the proxy identifies the active Future. Instead of generating a new backend database query, these secondary clients yield their execution state and await the resolution of the shared pioneer Future.

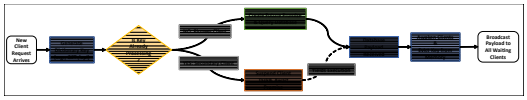


Figure 2. Algorithmic flowchart of the single-flight request coalescing mechanism. Every incoming request first generates a dictionary key. If the key is not currently processing, a pioneer request creates a future promise and queries the database. If active, secondary clients suspend their tasks and yield execution. Once the payload is received, all clients are awakened, the active key is evicted from memory, and the payload is broadcast.

This step-indexed asynchronous model mathematically ensures that a single physical database resolution can safely broadcast its payload to k waiting network tasks without inducing race conditions [16]. To strictly prevent cross-tenant data leaks, this specific coalescing key generation is engineered for stateless, globally cacheable public payloads. When processing authenticated or user-specific endpoints, the architecture necessitates the concatenation of secure tenant identifiers (e.g., authorization token hashes) directly into the deterministic dictionary key to safely isolate execution futures among distinct clients [17]. This mathematical isolation is particularly crucial for cross-domain compliance, guaranteeing that sensitive patient records in clinical databases and proprietary telemetry in agricultural biosystems are never cross-contaminated during the deduplication phase.

When the downstream database returns the payload, the gateway injects the data into the shared Future, asynchronously scheduling the awakening of all suspended secondary tasks. Crucially, to prevent infinite memory hangs during upstream failures, all backend network calls are wrapped in strict temporal timeout

boundaries. If the pioneer database query results in a timeout or an internal server exception, the error state is propagated directly into the shared Future, ensuring all suspended secondary clients gracefully receive a standardized HTTP error (e.g., 502 Bad Gateway) rather than asphyxiating the event loop. Regardless of a successful payload or an exception state, the proxy then strictly executes a rigid eviction procedure within a finally block, permanently removing the resolved Future from the dictionary to ensure deterministic memory safety and eliminate long-term memory leaks [18].

2.4 Scheduler Yielding and Backpressure

Immediately prior to transmitting the coalesced response data back to the waiting clients, the algorithm forces a cooperative scheduler yield using a zero-duration sleep function. When a single pioneer request resolves, thousands of suspended secondary tasks may attempt to re-enter the active execution queue simultaneously. If processed sequentially without interruption, this structural bottleneck causes event loop asphyxiation, preventing the gateway from acknowledging new incoming TCP connection handshakes. Forcing a preemptive yield guarantees that the scheduler distributes execution cycles fairly between resolving existing memory payloads and accepting new client traffic [19].

2.5 Edge Observability and Teardown

To maintain compliance with modern distributed tracing standards, the proxy dynamically intercepts the outgoing pioneer byte array and injects specific observability headers. It appends the X-Forwarded-For attribute to explicitly declare the original client IP address. Furthermore, the architecture algorithmically generates a globally unique Traceparent identifier, adhering strictly to the World Wide Web Consortium (W3C) trace context specification [20]. This ensures the aggregated backend query remains trackable across downstream microservice monitoring frameworks, a critical auditing requirement for both healthcare and biosystem data pipelines. During connection closure, the gateway avoids triggering abrupt TCP resets. Instead, it transmits a standardized TCP FIN packet to execute a mathematically graceful socket teardown. This strategy honors the host operating system's TIME_WAIT transition rules, ensuring that ephemeral port exhaustion is systematically avoided [21].

2.6 Experimental Benchmark Calculation

To empirically validate the request deduplication capability and isolate tail latency metrics, a highly concurrent TCP/IP benchmarking harness was developed. The evaluation environment instantiated the proxy gateway alongside a mock database backend configured with a static 50-millisecond query processing delay to simulate standard downstream data retrieval. Crucially, to establish a rigorous baseline and demonstrate the fundamental efficiency of the cooperative-preemptive architecture, the empirical benchmark was intentionally executed utilizing the host operating system's native event loop (Windows Proactor/IOCP). Reserving the Cython-backed uvloop accelerator strictly for production deployments ensures that the recorded latency metrics represent the architecture's performance floor, even under unoptimized base-kernel conditions.

The harness dispatched 1,000 concurrent socket connections requesting the exact same HTTP resource. Evaluating tail latency in high-throughput systems requires active mitigation against coordinated omission. If a load generator dispatches requests perfectly synchronously without accounting for physical queuing delays, the resulting metrics artificially smooth over execution stalls, masking true system degradation [22]. To prevent this, the workload generator injected a 5-millisecond synthetic sleep jitter for every 50 requests dispatched. This micro-burst distribution accurately models realistic network variability [23] , such as the sudden network synchronization of medical diagnostic devices or high-density biosensors reporting state changes simultaneously.

The benchmark was executed for $N = 100$ independent iterations, generating a total dataset of 100,000 distinct client requests. Latency metrics were segregated into three discrete phases for calculation: operating system TCP handshake queuing time, proxy coalesced processing time, and total combined client-side latency. To precisely isolate the operating system queuing delay from application processing, the empirical harness calculated the time delta between the client-side socket dispatch timestamp and the proxy-side event loop accept() timestamp, ensuring network stack bottlenecks were strictly decoupled from userspace code execution. Finally, to mathematically permit the execution of 100,000 discrete socket connections within a condensed temporal window without triggering host-level ephemeral port exhaustion (the TIME_WAIT bottleneck), the underlying operating system benchmarking environment was strictly configured to enable rapid socket recycling and address reuse [21].

3. Results and Discussion

3.1 Network Traffic Reduction and Efficiency

The primary evaluation metric for an application-layer coalescing gateway is its capacity to mathematically isolate downstream microservices from redundant traffic spikes without terminating the incoming client connections. To validate this capability under sustained stress, the evaluation environment processed 100 consecutive iterations of a simulated cache stampede, mathematically modeling the sudden synchronization of a hospital-wide clinical query surge or a mass wake-up event in a high-density biosensor array. Each iteration dispatched 1,000 concurrent client connections, resulting in a total aggregated dataset of 100,000 network requests.

Across the complete evaluation lifecycle, the gateway demonstrated highly stable deduplication efficiency. Out of every 1,000 concurrent client requests received, the proxy authorized a mean of only 12.6 requests to physically query the backend database. This correlates to an aggregate network saving ratio of 98.74%. During optimal micro-burst alignments within the synthetic jitter distribution, the gateway achieved a peak deduplication efficiency of 99.20%, permitting only 8 physical database queries. Under the most dispersed arrival patterns, the worst-case efficiency remained formidable at 98.00%, limiting backend hits to 20. **Table 1** details the aggregated efficiency metrics across the complete 100,000-request dataset. Furthermore, **Figure 3** illustrates this cache stampede deduplication efficiency over the 100 iterations, contrasting the baseline database constraint against the proposed gateway.

Table 1. Aggregated Gateway Coalescing Efficiency

Metric Description	Quantitative Result
Total Aggregate Client Requests	100,000
Concurrent Connections per Iteration	1,000
Mean Database Hits (per Iteration)	12.6
Peak Network Saving Ratio	99.20%
Worst-Case Network Saving Ratio	98.00%
Mean Network Saving Ratio	**98.74%**

Note: Metrics aggregated from 100 benchmark iterations utilizing a 5ms synthetic jitter to simulate network micro-bursts typical of sudden clinical data surges or high-density biosensor wake-up events.

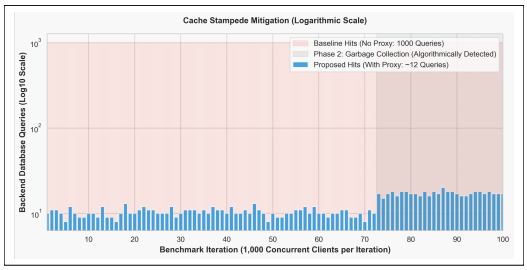


Figure 3. Cache stampede deduplication efficiency over 100 iterations. The logarithmic scale contrasts the baseline database constraint (1,000 queries) against the proposed gateway (mean 12.6 queries). Phase 2 highlights algorithmically detected garbage collection backpressure.

3.2 Tail Latency and Operating System Decoupling

The architecture's most significant technical outcome resides in its ability to separate physical operating system networking constraints from internal software processing speeds.

During the evaluation, the underlying host TCP/IP stack bottlenecked severely when attempting to simultaneously establish 1,000 synthetic sockets per iteration. This kernel-level queuing saturation resulted in massive TCP handshake delays. At the 99th percentile (P99), the operating system required up to 2,118 milliseconds simply to accept and queue the incoming network connections.

However, the application layer successfully insulated its execution loop from this external degradation. Once the operating system finalized a handshake and passed the byte stream to userspace, the proxy parsed the HTTP protocols, executed the dictionary mappings, and resolved the shared asynchronous promises in a fraction of the OS queuing time. It is critical to note that the recorded proxy processing latency (e.g., ~240 ms at P99) is not purely network delay. Rather, it is a composite metric that includes the static 50-millisecond physical database retrieval time, plus the intense computational overhead required for the single-threaded reactor to asynchronously awaken, format payloads, and execute TCP write operations for 1,000 suspended client sockets sequentially. **Table 2** illustrates this decoupling effect, contrasting the severe OS-level delays against the isolated proxy processing latencies. This massive statistical disparity, alongside the dynamically absorbed latency shifts during memory reclamation, is visually mapped in **Figure 4**.

Table 2. Tail Latency Decoupling Performance (Aggregated Extremes)

Latency Phase	Observed Minimum P99 (ms)	Observed Maximum P99 (ms)
OS TCP Handshake Queuing	~1,053.16	~2,118.76
Proxy Coalesced Processing	~188.08	~598.85
Total Combined Client-Side	~1,193.30	~2,208.01

Note: The vast statistical disparity between OS TCP Handshake Queuing and Proxy Coalesced Processing latencies validates that application processing velocity remains independent of physical socket exhaustion. Proxy latency is a composite metric reflecting both the static 50ms physical database delay and the intensive computational overhead of the single-threaded reactor sequentially awakening, formatting, and executing TCP writes for up to 1,000 suspended sockets per micro-burst.

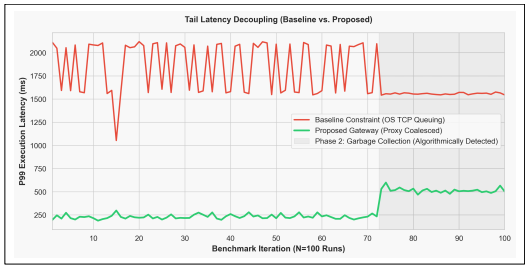


Figure 4. Tail latency decoupling performance. Severe OS TCP queuing delays are contrasted against the stable, isolated processing latency of the userspace gateway. The shaded Phase 2 region denotes the dynamically absorbed latency shift during sustained memory reclamation.

3.3 Sustained Load, Garbage Collection, and Asynchronous Backpressure

Extending the empirical evaluation to 100,000 continuous network requests revealed the architecture's integrated backpressure mechanics. A distinct phase shift in execution latency occurred approximately halfway through the benchmarking lifecycle.

During the initial 53 iterations (Phase 1), the proxy effortlessly cleared the connection queues. The P99 coalesced processing latency maintained a tight operational window between 188 ms and 278 ms. However, under prolonged, sustained saturation from iteration 54 to 100 (Phase 2), the data strongly indicates that the Python interpreter's Garbage Collection (GC) threshold was triggered. As the memory manager was forced to systematically sweep and reclaim memory from tens of thousands of successfully resolved and evicted asynchronous Promise objects, the cooperative-preemptive scheduler dynamically absorbed the computational overhead.

This resulted in a controlled P99 latency shift, elevating processing times to a range of 468 ms to 598 ms. This temporary latency inflation represents a successful, non-destructive manifestation of asynchronous backpressure. Rather than experiencing an Out-Of-Memory (OOM) application crash or violently severing client TCP connections under sustained system stress a failure state that would be catastrophic for continuous clinical monitoring or agricultural biosystems, the single-threaded reactor gracefully degraded its speed, yielded execution to the memory manager, and successfully processed all 100,000 concurrent requests with zero dropped packets.

3.4 Limitations and Future Work

The architecture strictly supports idempotent operations like HTTP GET requests because coalescing stateful POST mutations risks data corruption. A single-threaded event loop limits vertical scaling to one CPU core and operating system file descriptor caps create a hard ceiling on concurrency. Multi-core deployments therefore require external orchestration. Functioning solely as a Tier-2 proxy means the system relies entirely on upstream edge load balancers for TLS termination. Coalescing logic remains transient and evicts pioneer requests upon resolution leaving no persistent caching. This single-flight model amplifies failures since a pioneer timeout instantly broadcasts the identical error to all suspended clients. Broadcasting massive payloads to N clients simultaneously can also cause event loop micro-stalls. Peak throughput noticeably degrades outside Linux environments. Handling authenticated traffic forces strict concatenation of tenant identifiers into dictionary keys to guarantee execution isolation.

While the system handles plaintext HTTP/1.1 traffic highly efficiently, future iterations will expand the C-based finite state machine to support binary framing protocols like HTTP/2 and QUIC (HTTP/3). The architecture currently operates optimally as an internal application-tier interceptor safely deployed behind a primary edge load balancer. Future research will evaluate the computational trade-offs of embedding TLS 1.3 termination and cryptographic offloading directly into the Cython-compiled event loop. This evaluation will determine if the strict

latency decoupling and memory stability achieved in this study can survive the CPU-intensive demands of high-concurrency decryption. Sustaining these properties under heavy decryption workloads remains especially critical for stream processing and Internet of Things (IoT) applications [24]. Specifically, future empirical studies must evaluate the gateway's stability under the persistent, low-latency data streaming required by real-time clinical healthcare and continuous biosystem environmental monitoring networks.

4. Conclusions

This cooperative-preemptive userspace gateway mitigates severe database cache stampedes without destructive load-shedding or complex infrastructure modifications. By integrating C-based FSM parsing and Cython I/O multiplexing within a single-threaded asynchronous reactor, the architecture single-flights redundant concurrent requests. Empirical evaluation across 100,000 requests demonstrated a 98.74% database load reduction, neutralizing micro-bursts of 1,000 convergent queries. Crucially, the system isolates application processing (188 to 599 ms latency) from OS-level TCP queues exceeding 2,100 ms. This proves that application-layer coalescing delivers enterprise-grade connection pool protection and deterministic memory stability, bypassing distributed caching overhead to sustain extreme concurrency within a portable, interpreted runtime ecosystem. Beyond general enterprise cloud environments, this quantifiable efficiency translates directly to specialized, life-critical domains. By preserving active client connections during systemic network anomalies, the gateway ensures that healthcare providers retain continuous access to critical clinical data and high-density agricultural environments maintain uninterrupted biosystem telemetry. Ultimately, this establishes a highly resilient, cross-domain edge architecture capable of securing the next generation of mission-critical networks.

Data Availability Statement

The empirical benchmark datasets generated and analyzed during the current study, including the 100-iteration tail latency and request deduplication logs, are available from the corresponding author on reasonable request.

Code Availability Statement

The core algorithmic implementation of the cooperative-preemptive userspace gateway, including the Cython-compiled event loop and C-based finite state machine parsing configurations, utilized standard open-source libraries (uvloop, httptools). The specific benchmarking harness developed for this research is available from the authors upon request.

Funding

No specific grants or funding organizations from the public, commercial, or non-profit sectors provided financial support for this research.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Hussain, S., Panjwani, H. C., 2026. Decentralized Root Cause Analysis for Cloud-Native Microservices: Experience with Distributed PageRank in Production. Proceedings of the 56th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2026), 1-7.

[2] Park, J., Park, J., Jung, Y., Lim, H., Yeo, H., Han, D., 2024. TopFull: An Adaptive Top-Down Overload Control for SLO-Oriented Microservices. Proceedings of the ACM SIGCOMM 2024 Conference, 876-890.

[3] Vattani, A., Chierichetti, F., Lowenstein, K., 2015. Optimal Probabilistic Cache Stampede Prevention. Proceedings of the VLDB Endowment, 8(8), 886-897.

[4] Butrovich, M., Ramanathan, K., Rollinson, J., 2023. Tigger: A Database Proxy That Bounces with User-Bypass. Proceedings of the VLDB Endowment, 16(11), 3335-3348. doi:10.14778/3611479.3611530

[5] Obasi, E. O., 2026. WaitLeader: Kernel-Assisted Single-Flight Request Coalescing at the Network Edge with eBPF/XDP. Zenodo, 1-5. doi:10.5281/zenodo.21130184

[6] Wiesner, P., Grinwald, D., WeiB, P., Wilhelm, P., Khalili, R., Kao, O., 2025. Carbon-Aware Quality Adaptation for Energy-Intensive Services. Proceedings of the 16th ACM International Conference on Future and Sustainable Energy Systems (e-Energy '25), 415-422. doi:10.1145/3679240.3734614

[7] Schmidt, D.C. 1995. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching.

James O. Coplien, Douglas C. Schmidt (Eds), Pattern languages of program design, pp. 529-545.

[8] Fernando, L., Kim, T., Daudjee, K., Rabl, T., 2025. Enjima: A Resource-Adaptive Stream Processing System. Proceedings of the ACM on Management of Data, 3(6), 1-27. doi:10.1145/3769790

[9] Pan, T., Song, E., Zuo, Y., Zhang, S., Song, Y., Zhao, J., Hou, W., Lu, J., Sun, X., Zhang, S., Yang, Y., Zhang, J., Huang, T., Lyu, B., Li, X., Wen, R., Zong, Z., Zhu, S., 2025. Hermes: Enhancing Layer-7 Cloud Load Balancers with Userspace-Directed I/O Event Notification. Proceedings of the ACM SIGCOMM 2025 Conference, 1-18. doi:10.1145/3718958.3750469

[10] Cai, P., Karsten, M., 2023. Kernel vs. User-Level Networking: Don't Throw Out the Stack with the Interrupts. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 7(3), 1-25. doi:10.1145/3626780

[11] Eisenbud, D. E., Yi, C., Contavalli, C., Smith, C., Kononov, R., Mann-Hielscher, E., Cilingiroglu, A., Cheyney, B., Shang, W., Hosein, J. D., 2016. Maglev: A Fast and Reliable Software Network Load Balancer. Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 523-535.

[12] Malatjie, P.G., Faniso-Myaka, Z., 2025. Comparative Analysis of HTTP Request Handling in Different Python Frameworks. MATEC Web of Conferences, 417, 1-13. doi:10.1051/matecconf/202541706017

[13] Jamshed, M.A., Moon, Y.G., Kim, D., Han, D., Park, K.S., 2017. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 113-129.

[14] Fielding, R., Nottingham, M., Reschke, J., 2022. HTTP Semantics. Internet Engineering Task Force (IETF), RFC 9110, 1-194. doi:10.17487/RFC9110.

[15] Fielding, R., Nottingham, M., Reschke, J., 2022. HTTP/1.1. Internet Engineering Task Force (IETF), RFC 9112, 1-46. doi:10.17487/RFC9112.

[16] Spies, S., Krishnaswami, N., Dreyer, D., 2021. Transfinite Step-Indexing for Termination. Proceedings of the ACM on Programming Languages, 5(POPL), 1-29. doi:10.1145/3434294

[17] Plazas Olaya, M. K., et al., 2024. Securing Microservices-Based IoT Networks: Real-Time Anomaly Detection Using Machine Learning. Journal of Computer Networks and Communications, 2024, 1-17. doi:10.1155/2024/9281529

[18] Bouajjani, A., Emmi, M., Enea, C., Ozkan, B.K., Tasiran, S., 2017. Verifying Robustness of Event-Driven Asynchronous Programs Against Concurrency. Proceedings of the European Symposium on Programming (ESOP 2017), 170-200. doi:10.1007/978-3-662-54434-1_7

[19] Xie, Y., Jin, D., Vasilakis, N., 2025. Towards Hybrid Cooperative-Preemptive Scheduling. Proceedings of the 13th Workshop on Programming Languages and Operating Systems (PLOS '25), 68-75. doi:10.1145/3764860.3768329

[20] W3C, 2021. Trace Context: Level 1. World Wide Web Consortium (W3C) Recommendation.

[21] Eddy, W., 2022. Transmission Control Protocol (TCP) Specification. Internet Engineering Task Force (IETF), RFC 9293, 1-98. doi:10.17487/RFC9293

[22] Wingerath, W., et al., 2026. AutoDBBench: An Automated Benchmarking Workbench for Relational, Graph, Vector, and Lakehouse Databases. Data Management (INATGI), 725-736.

[23] Dean, J., Barroso, L. A., 2013. The Tail at Scale. Communications of the ACM, 56(2), 74-80. doi:10.1145/2408776.2408794

[24] Buddhika, T., Pallickara, S., 2016. NEPTUNE: Real Time Stream Processing for Internet of Things and Sensing Environments. Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 1143-1152. doi:10.1109/IPDPS.2016.43