JOURNAL OF
Emerging Paradigms
in Computing Systems
JEPCS

Journal of Emerging Paradigms in Computing Systems (JEPCS) | Volume 02, Issue No. 01, 2026

ARTICLE TYPE: Research Article

Article ID: JEPCS-2026-0201-0059

Performance and Fairness of Web API Rate-Limiting Algorithms under Bursty and Heterogeneous Traffic: A Controlled Event-Driven Study

Sadia Aslam[1,*], Muhammad Umar Hameed[1]

[1] Faculty of Information Technology & Computer Science, University of Central Punjab, Lahore, Pakistan

*Corresponding author: sadia.aslam@ucp.edu.pk

Contributing authors: sadia.aslam@ucp.edu.pk,umar.hameed@ucp.edu.pk

Received: October 11, 2025 **Accepted:** December 19, 2025

Abstract- An API quota such as 100 requests per second does not, by itself, specify how a limiter behaves around a burst or how admitted capacity is divided among competing clients. We examine that ambiguity with five implementations: fixed-window counting, an approximate sliding-window counter, an exact sliding-window log, a global token bucket, and an equal-share per-client token bucket. The same event traces are replayed through every limiter. The workloads include Poisson arrivals, boundary-aligned and synchronized bursts, heterogeneous clients, and a flash crowd. The global policies use a nominal rate of 100 requests per second. In the boundary experiment, two 100-request bursts were placed 20 ms apart on opposite sides of a fixed-window boundary. Fixed window admitted 200 requests within one rolling second. The exact sliding log and per-client token bucket admitted 100, whereas the approximate counter admitted 101 and the global token bucket 102. A second experiment used 16 clients at 6 requests/s and four clients at 25 requests/s. Jain's index remained between 0.6234 and 0.6258 for the global limiters, while the equal-share policy reached 0.9995 with 97.20 requests/s of admitted throughput. The same static allocation was inefficient when the 16 lower-rate clients had only 2 requests/s, with admitted throughput dropping to 52.20 requests/s as their unused shares were not reallocated. The measurements thus consider rolling-window compliance, client allocation and utilization as separate features of an API rate-limit policy.

Keywords- API rate limiting; token bucket; sliding window; fairness; burst traffic

1. Introduction

An API provider might impose a limit, for example, "100 requests per second," but that statement leaves some part of the policy unspecified. A client can exceed a limit, as defined by RFC 6585 [1] and the server can then return status code 429. The RFC does not specify how the client is identified, nor how requests are counted. Two implementations can therefore advertize the same numerical quota, but still pass different amounts of traffic to the protected service.

This difference is more evident for burst traffic. Fixed window counting is tied to a clock boundary, so a request arriving just before and one just after a reset will be charged to different counters even if they are temporally close. Sliding-window schemes avoid that particular reset effect, but an exact log keeps more state, as recent timestamps must be kept. Token buckets have yet another rule: stored tokens are allowed to absorb a short burst while the refill rate controls sustained admission. The basic ideas stem from prior work on leaky-bucket regulation, generalized processor sharing and fair resource allocation [2–5]. However , in an API , the regulated unit is usually a request , not a packet or a virtual circuit .

Capacity sharing causes another problem. A global limiter can keep aggregate admission near a target, while allowing a few high-rate clients to consume a large fraction of that capacity. Such concentration is important in service chains, where overload and long tail behavior can propagate across dependent requests [6]. Per-client limits can isolate callers. But

a fixed partition also has a cost: capacity assigned to an idle client cannot be used elsewhere unless the policy includes redistribution. Related problems are studied with more general system feedback in [7] (Distributed enforcement), [8] (fairness-throughput scheduling), [9, 10] (adaptive Web admission), and [11–14] (microservice overload control). A traditional API limiter is typically less complex than these mechanisms, but the allocation choice still exists.

In a few studies the setting of the API is approximated. El Malki et al. [15] model the impact of rate limits on microservice reliability, while Serbout et al. [16] discuss design choices for rate-limit method, scope and granularity. Noah uses reinforcement learning [17] to adjust the limits as the microservice behavior changes. Other works statistically estimate the throttling limits [18] or adapt clients such that repeated HTTP 429 responses are reduced [19]. The studies are concerned with the selection, adaptation or response to a boundary. We are more concerned with a more basic question: what traffic is counted under the underlying counting rule and scope before adaptation is applied?

We evaluate five configurations on the same timestamped traces: fixed window, approximate sliding counter, exact sliding log, global token bucket, and equal-share per-client token bucket. Each limiter has the same nominal rate, only the admission semantics or policy scope varies. The experiments differ in the offered load and traffic shape. We record acceptance ratio, maximum count admitted in a rolling one second interval, Jain's allocation index and admitted throughput. The analysis also follows the share taken by four high-rate clients, recovery on a digital flash-crowd trace, and processing cost of the provided Python implementation. The per-client token bucket is intentionally simple and is used to reveal the impact of static isolation, not as a comprehensive fairness scheduler.

The experiments are designed to address three questions. Does the limiter roll one second ceiling during a burst? How does a global quota divide when client demand varies and how much capacity is stranded by fixed equal shares? The event-driven design is motivated by the fact that these questions concern admission semantics. Network transfer, HTTP parsing, backend execution and gateway latency are outside the model, and no end-to-end latency claim is made.

Organization of the paper is given below: Section 2 reviews related work. Section 3 describes the limiter model and measures. The workload protocol is described in Section 4 and the measurements in Section 5. Limitations of the study and design implications are discussed in Sections 6 and 7.

2. Related Work

Early rate-control work was from packet networks. Turner studied leaky-bucket regulation for high speed networking [2] and Sidi et al. [3] studied the effect of input regulation on smoothing and delay. Parekh and Gallager used generalized processor sharing combined with leaky-bucket constraints to provide service guarantees [4]. Demers et al. [5] addressed fair sharing among contending flows. Jain et al. [20] presented the bounded fairness index that we use later in our experiments. Kim [21] studied token-bucket control with minimum and maximum constraints. Abendroth and Jensen [8] studied fairness and throughput in bucket-based scheduling. Carousel additionally measured the CPU and memory costs for accurate end-host rate shaping [22] at larger scale.

With the emergence of extremely concurrent Internet services, admission control was applied to protect the service instead of only shaping a flow. To this end, SEDA used explicit queues and dynamic resource control [23]. Later, Welsh and Culler adapted the admission according to observed server conditions [9] and Bartolini et al. studied self-configuration in distributed Web overload and flash crowds [10]. Raghavan et al. [7] considered a different deployment problem of coordinating rate enforcement across sites in an effort to enforce a global policy without a single central limiter.

Microservice overload control adds information from service graph and server state. DAGOR [11] coordinates the load shedding in the WeChat backend, and the fairness is included in the control logic. Breakwater controls RPC demand using credits that are issued by the server [12]. TopFull employs per-API control based on global observations [13] and DRACO considers resource conditions in multi-tier admission decisions [24]. Rajomon coordinates overload control with distributed tokens and prices [14]. Their goal is to provide complete protection for a service system. The current benchmark, instead, ignores the service model and compares the local semantics of the limiter itself.

APIs work on the server's side limit and client's response to the server's side limit. El Malki et al. [15] examine the impact of API rate limits on reliability, and Serbout et al. [16] list choices such as method, scope, and granularity. Arman et al. [18] investigate the choice of a throttling value, whereas Farkiani et al. [19] study retries in the presence of multiple clients sharing an HTTP quota. The protocol standards define signaling but not a counting algorithm. RFC 6585 defines status code 429 [1], and RFC 9110 defines fields including Retry-After [25].

We compare these studies to the controlled comparison presented here in Table 1. We do not add an adaptive controller. Instead, the limiter rule or scope is changed with the nominal rate and input trace fixed. Thus, the rolling-window behavior, client allocation, utilization, and state or processing cost can be compared without the need to change the offered workload.

Table 1: *Representative related work and its relationship to the controlled comparison in this study.*

Study	Primary concern	Relation to this paper
Raghavan et al. [7]	Distributed global rate enforcement	Addresses coordination across sites rather than burst/fairness behavior of local counting semantics.
Welsh and Culler [9]	Adaptive Web-server admission	Controls server overload using measured response time.
DAGOR [11]	Microservice overload and fairness	Uses system-level collaborative shedding in a service graph.
Breakwater [12]	Credit-based RPC admission	Regulates demand using server feedback and queueing delay.
El Malki et al. [15]	API reliability under rate limits	Models success/failure rates and adaptive limit selection.
Serbout et al. [16]	API rate-limit design patterns	Describes method, scope, and granularity choices for API designers.
TopFull [13]	Adaptive per-API overload control	Optimizes goodput under microservice SLO constraints.
Arman et al. [18]	Statistical selection of API limits	Chooses the rate value rather than comparing elementary limiter semantics.
Farkiani et al. [19]	Client-side throttling behavior	Reduces failed retries under a shared HTTP API quota.
This study	Limiter semantics and scope	Compares rolling burst behavior, allocation equality, utilization, and structural/processing cost under identical traces.

3. Rate-Limiting Model and Methods

3.1. System Model

We model one admission point in front of an API. Each event contains an arrival time and a client identifier and the identifier is consulted only by the per-client baseline. Once an event reaches the limiter, it is either *admit*ted or *reject*ed. Backend service time is not simulated. The global policies are configured with $R = 100$ requests/s over $W = 1$ s, the token bucket holds $B = 100$ requests, and the default workload contains $N = 20$ clients.

This benchmark is intentionally narrower than a gateway benchmark. For any seed, each limiter replays the identical timestamped events. Thus, the comparison includes no transport timing, scheduler variation, or backend execution. What is measured is the rule of admission. Latency of HTTP gateway is not measured.

3.2. Fixed-Window Counter

In fixed-window counting, time is divided into intervals of width W. A request that arrives at time t belongs to window

$$k(t) = \left\lfloor \frac{t}{W} \right\rfloor. \tag{1}$$

When $k(t)$ changes, the counter of the new interval is set to zero. The count is less than $L = RW = 100$, so the request is accepted. It does depend on where the fixed boundary falls . But you only need a constant state . So, two dense groups of requests can be counted separately even if both groups are within the same rolling one second period.

3.3. Sliding-Window Counter and Log

The sliding counter is approximate and keeps two counts, one for the current fixed window and one for the previous window. Let e be the fraction of the current window that has passed. The implementation estimates recent demand as

$$\widehat{C}(t) = C_{\text{cur}} + (1 - e)C_{\text{prev}}. \tag{2}$$

We only admit if $\widehat{C}(t) + 1 \le L$. The state is of constant size. This implementation can be conservative too if an earlier burst is dense since part of the previous count is carried into the estimate.

The exact sliding log uses a different procedure: It maintains timestamps of accepted requests, and when a new event at t is considered, it removes timestamps at or before $t - W$. Accept the event only if there are fewer than L admitted timestamps left. So the implemented rule is exactly enforced by the previous rolling window. The stored state increases with the quota and interval.

3.4. Global and Per-Client Token Buckets

The global token bucket is filled with B tokens. Tokens are added to the bucket at rate R continuously, until the bucket is full. where $T(t^-)$ is the previous level before an arrival. If Δt is the time since the last update, the new level is

$$T(t) = \min\{B,\, T(t^-) + R\Delta t\}. \tag{3}$$

One token is removed for each admitted request. Accumulated tokens may be spent in a short interval, so the rule is not intended to impose the same rolling-window ceiling as the sliding log. Sustained admission is instead governed by the refill rate.

For the per-client baseline, N independent token buckets are created. Each bucket receives a refill rate of R/N and capacity B/N. With 20 clients, this corresponds to 5 requests/s and five stored tokens per client, while the nominal sum remains 100 requests/s. A client's share is isolated from the others and if that client does not use it, the unused capacity is not transferred.

3.5. Evaluation Measures

Let O be the number of offered requests and A the number admitted. We report A/O as the acceptance ratio and A/T as admitted throughput over duration T. Burst behavior is measured from M_{1s}, the largest number of admitted requests found in any rolling one-second interval. We normalize that count by the nominal one-second quota:

$$\Gamma = \frac{M_{1s}}{L}. \tag{4}$$

When $\Gamma = 1$, no observed rolling second exceeds the nominal quota. Values above one have to be interpreted according to the limiter. In a token bucket, for example, stored tokens are meant to permit a temporary burst, so a value greater than one is not necessarily an error.

In the saturated heterogeneous workload, every client asks for more than its equal share, which lets us compare allocation equality directly. Jain's fairness index is used for that purpose [20]. If client i receives x_i admitted requests,

$$J = \frac{\left(\sum_{i=1}^{N} x_i\right)^2}{N \sum_{i=1}^{N} x_i^2}. \tag{5}$$

The index lies in $(0, 1]$ and reaches one when admitted counts are equal. We also calculate the fraction of admitted requests received by the four aggressive clients. Those clients make up 20% of the population, so a capture share close to 20% corresponds to equal allocation in the saturated experiment.

Processing cost is timed in a separate microbenchmark. A pre-generated trace is passed through each Python implementation nine times and the first execution is treated as warm-up and removed. From the remaining runs we report median nanoseconds per request together with the 25th and 75th percentiles. These timings belong to the supplied Python decision path. They do not include network transfer, HTTP parsing, serialization, persistence, or synchronization across replicas.

4. Experimental Setup

Workload generation is completed before any limiter is run. For a given seed, every algorithm therefore receives the same ordered event trace. Poisson arrivals are generated independently per client and merged by timestamp. We use twenty independent seeds for the load sweep and flash crowd, and 30 seeds for each heterogeneous-client experiment. Where repeated runs are available, means are reported with 95% confidence intervals from the Student-t distribution.

The six traffic scenarios are listed in Table 2. Aggregate Poisson demand is swept from $0.5R$ to $2R$. For the boundary test, 100 requests are placed 10 ms before a fixed-window reset and another 100 are placed 10 ms after it. Thirty such pairs are generated with enough separation between pairs. Table 3 reports one pair, and aggregate counts over all pairs are given in the results. The synchronized workload sends 160 requests at the start of each second for 60 seconds. Heterogeneous demand is examined with sixteen normal and four aggressive clients. All clients exceed 5 requests/s in the saturated case, whereas the 16 normal clients offer only 2 requests/s in the underloaded case so that unused static shares can be observed. The flash-crowd trace runs at 60 requests/s, rises to 250 requests/s for 20 seconds, and then returns to 60 requests/s.

Table 2: *Traffic scenarios and repeated-run protocol.*

Scenario	Offered traffic	Runs	Purpose
Load sweep	20 equal Poisson clients; $0.5R, 0.8R, 1.0R, 1.2R, 1.5R, 2.0R$	20	Admission behavior as load crosses capacity.
Boundary burst	Two 100-request bursts separated by 20 ms across a window boundary; 30 pairs	1	Rolling-window amplification.
Periodic burst	160 synchronized requests/s for 60 s	1	Repeated burst sensitivity.
Heterogeneous saturated	16 clients at 6 requests/s; 4 at 25 requests/s	30	Equal-share fairness when all clients exceed 5 requests/s.
Heterogeneous underloaded	16 clients at 2 requests/s; 4 at 30 requests/s	30	Fairness-utilization trade-off when static shares are unused.
Flash crowd	60 requests/s for 30 s; 250 requests/s for 20 s; 60 requests/s for 30 s	20	Surge admission and recovery after overload.

Python 3.13.5 was used on x86-64 Linux running under KVM. The environment reported an AMD EPYC 9V74 processor with five virtual CPUs. Trace generation and analysis used NumPy 2.3.5, pandas 2.2.3, and SciPy 1.17.0. Hardware timing affects only the Python microbenchmark in Section 5.6. The event-driven admission results are determined by the generated timestamps and limiter code. Random seeds are fixed in the experiment script, and raw per-run CSV files are written before the summaries and figures are produced.

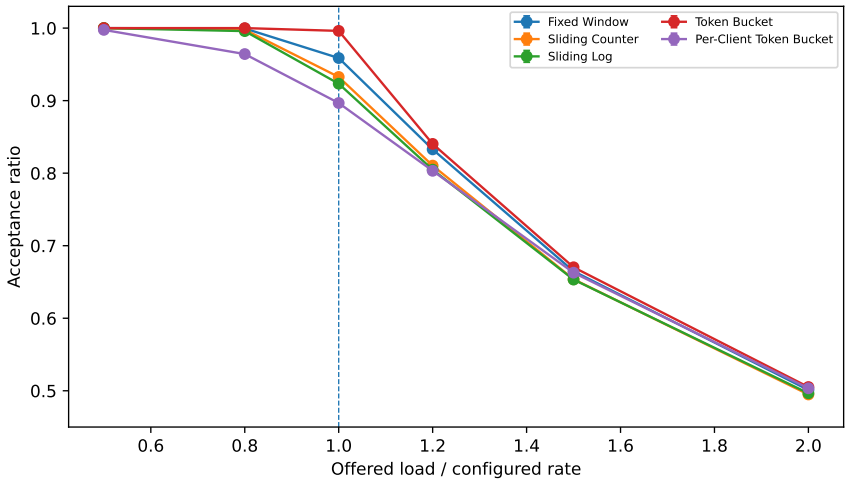


Figure 1: *Mean acceptance ratio under increasing Poisson offered load. Error bars show 95% confidence intervals over 20 seeds.*

5. Results and Discussion

5.1. Admission under Increasing Load

The global policies accept almost all requests when the load is below the configured aggregate rate. Figure 1 illustrates the full sweep. At $0.8R$, acceptance is 0.9994 for the fixed window, 0.9978 for the sliding counter, 0.9958 for the sliding log and 1.0000 for the global token bucket. The token bucket per client is smaller at 0.9642. A small client-side burst can empty its independent 5-requests/s buckets even when another client's allocation is unused.

When demand at any point exceeds the limit, the acceptance ratios tend toward the share implied by the 100 requests/s service budget. The results are 0.8328, 0.8104 and 0.8048 for fixed window, sliding counter and sliding log, respectively. The global token bucket and the per-client token bucket are 0.8403 and 0.8032 respectively. At $2R$ they are 0.5011, 0.4949, 0.4965, 0.5052 and 0.5034 respectively. These small differences have to do with window placement, initial bucket contents, and the finite run length. The long-run configured rate is the same.

Aggregate acceptance does not describe the biggest short term burst. For the implemented rule at $2R$ the exact sliding log remains at $\Gamma = 1.00$. The mean maximum amplification is 1.347 for fixed window and 1.046 for the sliding counter . The per-client and global token buckets reach 1.588 and 1.969 . The token-bucket values are derived from stored capacity. Requests may use tokens that were already available as the refill proceeds, so the policy allows a burst that a strict rolling-window limit would deny.

5.2. Boundary and Synchronized Bursts

The boundary trace was built to show the reset of the fixed window. 100 requests arrive just before the boundary and another 100 just after. For the pair reported in Table 3 fixed window accepts all 200 and gets $\Gamma = 2.00$. The sliding log allows 100. The per-client token bucket with the chosen equal shares allows 100. The approximate sliding counter admits 101. The global token bucket admits 102 because 20 ms of refill supplies two additional tokens between the bursts.

Table 3: *Admission for one 200-request boundary pair separated by 20 ms.*

Algorithm	Admitted	Γ
Fixed Window	200	2.00
Sliding Counter	101	1.01
Sliding Log	100	1.00
Token Bucket	102	1.02
Per-Client Token Bucket	100	1.00

The same behavior persists across the 30 separated pairs. Fixed window accepts all 6000 offered requests. The totals are 3001 for the sliding counter, 3000 for the sliding log, 3060 for the global token bucket, and 3000 for the per-client token bucket. These counts do not imply additional backend capacity for fixed window. They are a direct consequence of assigning the two bursts to different fixed counters despite their 20 ms separation.

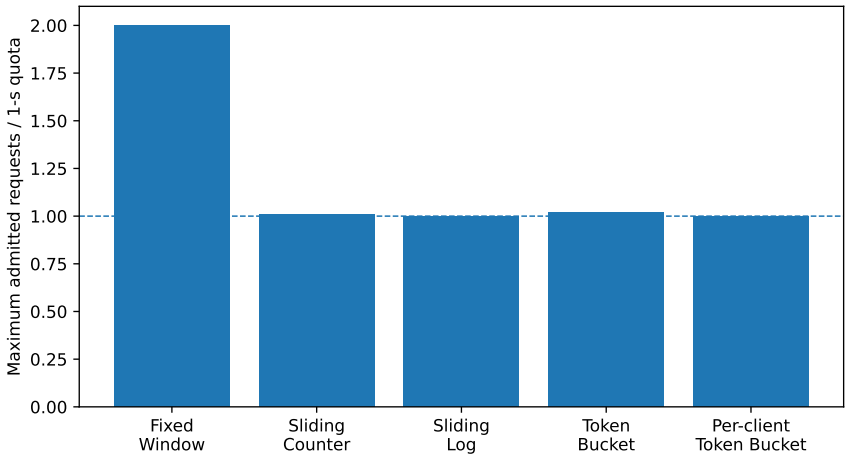


Figure 2: *Maximum admitted requests in a rolling one-second interval, normalized by the nominal one-second quota, for the boundary-burst experiment.*

Synchronized bursts produce a different result. When 160 requests arrive at the start of every second, fixed window, sliding log, and global token bucket each admit 6000 of 9600 requests. Their acceptance ratio is 0.6250, while the per-client token bucket admits 5998. The approximate sliding counter admits 3000, or 0.3125. At the start of a new interval, the previous full window still receives a large weight in Eq. (2), so this particular estimator rejects more aggressively. The finding belongs to this two-window approximation and should not be extended to every implementation called a sliding-window limiter."

5.3. Fairness under Saturated Heterogeneous Demand

In the saturated heterogeneous workload, 16 clients offer 6 requests/s and four offer 25 requests/s. Since every client is above the 5 requests/s equal share, equal allocation can be compared without an idle-client effect. The global policies constrain the total but do not reserve capacity per client. Their Jain indices are 0.6246 for fixed window, 0.6243 for the sliding counter, 0.6234 for the sliding log, and 0.6258 for the global token bucket. The equal-share per-client policy reaches 0.9995, as shown in Figure 3.

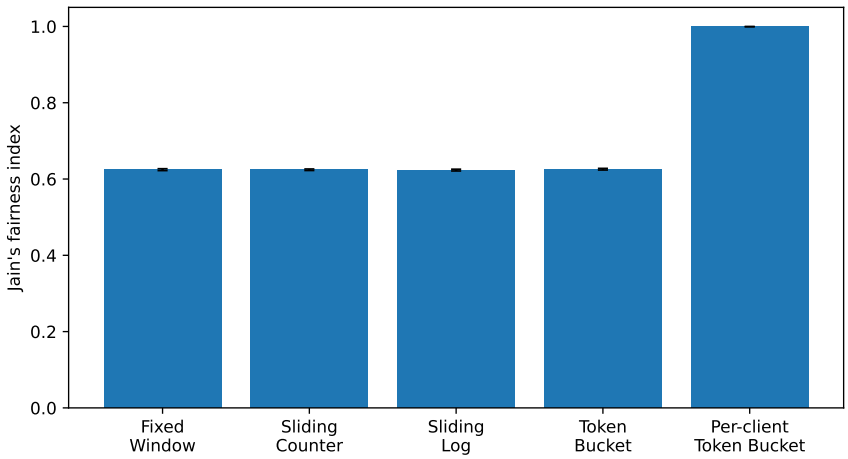


Figure 3: *Jain's fairness index under saturated heterogeneous demand. Bars show means and 95% confidence intervals over 30 seeds.*

The client shares make the imbalance easier to read. Four aggressive clients represent 20% of the population, but under global policies they get around 51% of all admitted requests. The shares of the fixed window, the sliding counter, the sliding log, and the global token bucket are 50.98%, 51.00%, 51.06%, and 50.90% respectively. The mean share is 20.71% with equal share separate buckets. These values are plotted in figure 4 .

Equalizing the shares also changes the degree to which the nominal capacity is utilized. The mean admitted throughput is 100.00 requests/sec for fixed window, 98.74 for the sliding counter and 99.05 for the sliding log. The global and per-client token buckets can reach 100.82 and 97.20 requests/s. The global token bucket can average slightly above 100 requests/s

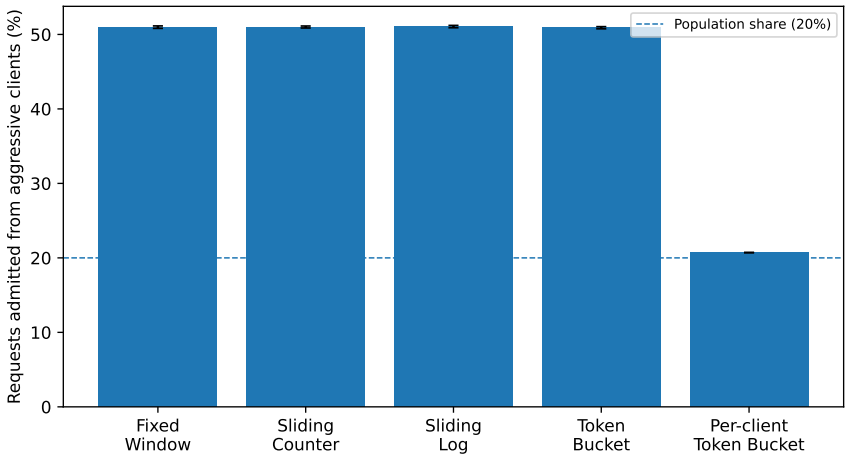


Figure 4: *Share of admitted traffic obtained by the four aggressive clients in the saturated heterogeneous workload. The dashed reference corresponds to their 20% population share.*

over the finite 120 second run since it starts with a full token inventory. In the per-client case a client can run out of its five token bucket, while another client has tokens that have not been used, and those tokens cannot be borrowed.

5.4. Uneven Demand and Quota Fragmentation

The underloaded heterogeneous case affects only the pattern of demand. You now have 16 clients with 2 requests/s, 4 clients with 30 requests/s and total demand of 152 requests/s. The system is still overloaded in aggregate, but most normal clients leave part of their 5 requests/s static allocation unused. Under the global policies, admitted throughput ranges from 98.31 to 100.80 requests/s. Jain's index stays near 0.316 because the admitted traffic follows the unequal offered demand.

Separate equal-share buckets produce a very different operating point. Jain's index rises to 0.8213, but throughput falls to 52.20 requests/s. Normal clients are almost never throttled, with an acceptance ratio of 0.9996, whereas the aggressive clients are admitted at 0.1679. Their share of admitted traffic falls from about 78.9% under the global policies to 38.6%. The remaining capacity is stranded because unused normal-client quota is not reassigned.

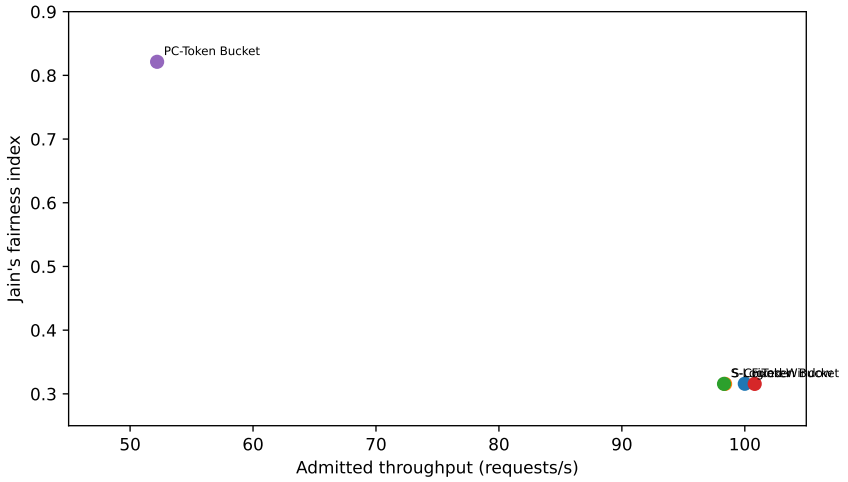


Figure 5: *Allocation equality versus admitted throughput when 16 clients are below their static equal share and four clients offer high demand.*

The underloaded case makes the cost of static isolation explicit. Fixed per-client quotas can protect one client from another, but they are not work conserving when demand is uneven. Redistributing spare capacity would require a different mechanism, such as weighted sharing or a scheduler above the individual buckets. Related fairness-throughput trade-offs have been studied in queueing and bucket-based network schedulers [4, 8]. Here the same tension appears at the API admission point.

5.5. Flash-Crowd Behavior

The flash-crowd trace begins at 60 requests/s for 30 seconds, rises to 250 requests/s for 20 seconds, and then returns to 60 requests/s for 30 seconds. Before the rise, the four global policies admit the offered load on average. During the hot interval, admitted throughput is 100.00 requests/s for fixed window, 98.89 for the sliding counter, and 99.73 for the sliding log. The global token bucket reaches 104.92 requests/s. The per-client token bucket admits 103.07 requests/s over the same interval.

The token-bucket averages above 100 requests/s are produced by stored capacity, not by a higher refill rate. Across seeds, the largest one-second count during the hot interval averages 197.8 for the global token bucket and 159.1 for the per-client version. For the one-second bins used in this trace, fixed window averages 100.0, the sliding counter 99.0, and the sliding log 100.0. One representative run is plotted in Figure 6.

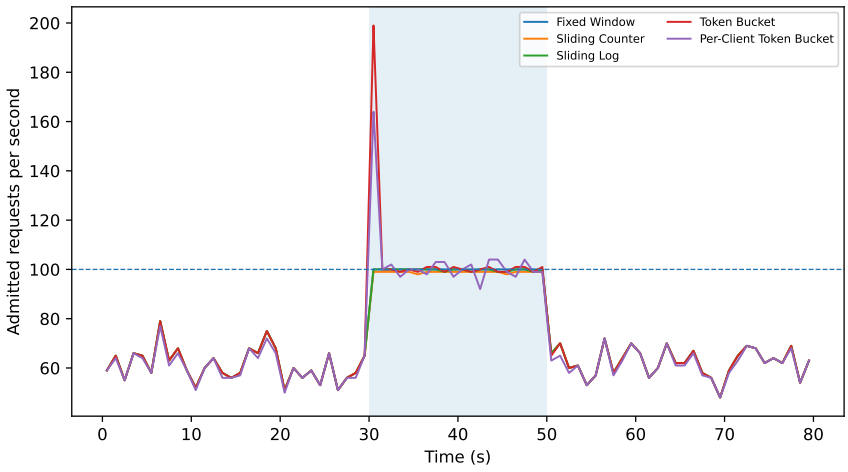


Figure 6: *Admitted requests per second for one flash-crowd trace.*

When the offered rate drops back to 60 requests/s, fixed window meets the chosen recovery criterion immediately in every run. Mean recovery is 0.05 s for the sliding counter and global token bucket, 0.45 s for the sliding log, and 1.50 s for the per-client token bucket. Recovery is declared only after three consecutive one-second bins admit at least 95% of the requests offered in those bins. Since the model contains no backend queue, these values describe the digital admission trace and they are not measurements of service recovery after queue buildup.

5.6. Implementation and State Cost

Table 4 reports timing from the Python decision path in the supplied environment. State size also differs across implementations. Fixed window and the global token bucket each keep two scalar items and the sliding counter keeps three. The exact log can retain up to 100 admitted timestamps. With 20 clients, the per-client token bucket maintains two scalar values for each client.

Table 4: *Python decision-path microbenchmark on a 299,587-request pre-generated trace. The first of nine runs is discarded.*

Algorithm	Median ns/req.	P25	P75	State items
Fixed Window	413.43	404.23	415.28	2
Sliding Counter	543.50	537.28	549.24	3
Sliding Log	459.95	456.11	464.08	100
Token Bucket	413.50	411.59	416.49	2
Per-Client Token Bucket	477.71	472.54	482.87	40

The timing order should not be generalized beyond this code. In our Python implementation, the approximate sliding counter is slower than the log despite storing less state. Interpreter overhead and the executed operations determine that result. A production implementation could produce another ordering. The state requirement is less platform dependent: the exact log grows with admitted events retained inside the window, whereas fixed window and the global token bucket keep constant scalar state.

5.7. Implications for API Rate-Limit Design

The measurements make one thing clear: the number in a rate-limit policy doesn't fully define the policy. The fixed window in the tested boundary trace allows for twice the nominal one second quota, because the reset is between the bursts. The sliding log maintains the tested rolling interval to its quota, but retains recent admitted timestamps. The token bucket

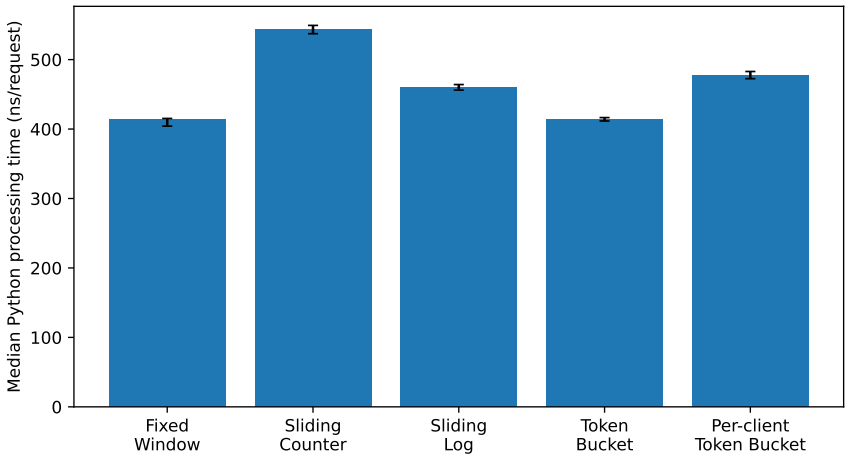


Figure 7: *Median Python processing time per request. Error bars span the 25th to 75th percentile across eight timed repetitions after one warm-up run.*

sacrifices strict rolling ceiling for burst capacity by design, while the approximate sliding counter has constant state and may reject conservatively if bursts happen to fall on the window structure.

The scope of the policy is relevant in its own right, and not as a matter of counting rule. In the saturated mixed workload, the four high-rate clients constitute one fifth of the population but they account for about half of the admitted requests under every global limiter. Equal per-client buckets reduce a lot of this imbalance when all clients are saturated. But with uneven demand, the same fixed shares leave capacity idle. Then the provider has to decide which of the requirements is the priority: aggregate backend protection, client isolation, burst tolerance or a rolling-window ceiling. You will not optimize all four with one limiter configuration.

RFC 6585 does not specify a rule for counting requests or a model for client identity [1]. Documentation should describe the scope of the quota and time semantics for clients that rely on predictable throttling. It should also indicate if bursts are allowed. A number alone is insufficient. Another part of the problem is the retry behavior. The Retry-After [25] header is defined in RFC 9110 and client-side adaptation has been studied as a mechanism to reduce repeated 429 responses [19].

6. Limitations and Threats to Validity

The benchmark isolates the admission logic as it is not a production gateway experiment. The harness produces event timestamps, and no request goes through a socket, HTTP parser, TLS stack, Redis instance, shared-memory path, or backend service. The reported comparisons thus refer to the semantics of the implemented limiter. They should not be used as end-to-end throughput or latency results for NGINX, Kong, Envoy, a cloud gateway or any other deployed product.

We consider only one nominal rate, one one-second interval and default population of 20 clients with the synthetic workload families in Table 2. Changing the interval length, bucket size, number of clients, or cost per request can change both the overhead and the observed behavior. The fixed-window boundary effect is a consequence of the tested reset rule, but the numerical values obtained from the Poisson, heterogeneous and flash-crowd traces are within the stated parameters.

Our measure of fairness also has a limited interpretation. Equal admitted counts make sense in the saturated experiment since all clients ask for more than the same nominal share. The underloaded case is not used to argue that clients with different legitimate demand should receive equal throughput and it is used to measure the capacity stranded by static partitioning. Tenant weights, paid tiers, endpoint costs, grouped identities, dynamic borrowing, etc. are all real policies but none of that is modeled here.

The per-client baseline is intentionally non-work-conserving. It never steals unused tokens from other client, hence the underloaded experiment drops to 52.20 requests/s. Some isolation could be preserved with a fair scheduler or a hierarchical bucket that re-allocates excess capacity. Likewise, the sliding-counter implementation is just an approximation with two windows, and its behavior should not be considered to be representative of all sliding-window algorithms.

The timing numbers are the most non-portable piece of the study. They depend on the Python version, the virtual machine, and details of the code path. We include them so that the supplied implementations have a measured processing cost, not to rank production gateways by nanoseconds. A deployment-oriented extension would need a real gateway or middleware implementation, concurrent state access, persistence, and measurements of end-to-end and tail latency, memory allocation, and multi-node consistency cost.

7. Conclusion and Future Work

Five rate-limit configurations were evaluated with the same event traces and a nominal capacity of 100 requests/s. Their behavior was not interchangeable. In the boundary test, fixed window admitted 200 requests inside one rolling second. The exact sliding log and equal-share per-client token bucket admitted 100, the approximate sliding counter 101, and the global token bucket 102. The difference comes from the policy semantics applied to the same 20 ms-separated bursts rather than from a change in offered traffic.

Client demand changed the allocation result as well. With all 20 clients above their equal share, Jain's index stays near 0.62 for the global policies, and the four high-rate clients receive about 51% of admitted traffic. The static per-client token bucket changes those values to 0.9995 and 20.71% respectively, while admitting 97.20 requests/s. The same policy allows 52.20 requests/s only when most clients are below their allocated share, because unused quota remains with the low-demand clients.

There is no configuration that is preferred by every metric in this benchmark. Which is right depends on what the API policy is meant to control: a strict rolling-window count, a permitted burst budget, aggregate backend admission, or separation among clients. The provided traces and code make those differences reproducible at the admission rule level. The next logical step is to replay these workloads through production gateways, then add work-conserving redistribution, weighted tenant shares, and replicated rate-limit state so we can measure the costs of consistency directly.

Author Declarations

Author Contributions (CRediT): Sadia Aslam: Conceptualization, Methodology, Software, Investigation, Data Curation, Formal Analysis, Validation; Muhammad Umar Hameed: Visualization, Writing - Original Draft, Writing - Review & Editing. The authors have read and approved the submitted version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The study uses digitally generated request traces.

Acknowledgments: No acknowledgments are reported.

Conflicts of Interest: The authors declares no conflict of interest.

Ethics Statement: Not applicable.

Declaration of Generative AI: During manuscript preparation, the authors used LLM AI for literature organization, data-analysis support, and language editing. The authors reviewed the manuscript and remains responsible for the accuracy, integrity, and content of the manuscript.

References

[1] M. Nottingham and R. T. Fielding, "Additional http status codes," Internet Engineering Task Force, Tech. Rep. RFC 6585, Apr. 2012.
[2] J. S. Turner, "New directions in communications (or which way to the information age?)," *IEEE Communications Magazine*, vol. 24, no. 10, pp. 8–15, Oct. 1986.
[3] M. Sidi, W.-Z. Liu, I. Cidon, and I. Gopal, "Congestion control through input rate regulation," *IEEE Transactions on Communications*, vol. 41, no. 3, pp. 471–477, Mar. 1993.
[4] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: The single-node case," *IEEE/ACM Transactions on Networking*, vol. 1, no. 3, pp. 344–357, Jun. 1993.
[5] A. J. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *Proceedings of ACM SIGCOMM*, 1989, pp. 1–12.
[6] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013.
[7] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren, "Cloud control with distributed rate limiting," in *Proceedings of ACM SIGCOMM*, 2007, pp. 337–348.
[8] D. Abendroth, M. E. Eckel, and U. Killat, "Solving the trade-off between fairness and throughput: Token bucket and leaky bucket-based weighted fair queueing schedulers," *AEU - International Journal of Electronics and Communications*, vol. 60, no. 5, pp. 404–407, 2006.
[9] M. Welsh and D. Culler, "Adaptive overload control for busy internet servers," in *4th USENIX Symposium on Internet Technologies and Systems (USITS 03)*. USENIX Association, Mar. 2003, pp. 43–57.
[10] N. Bartolini, G. Bongiovanni, and S. Silvestri, "Self-* through self-learning: Overload control for distributed web systems," *Computer Networks*, vol. 53, no. 5, pp. 727–743, Apr. 2009.
[11] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang, "Overload control for scaling wechat microservices," in *Proceedings of the 9th ACM Symposium on Cloud Computing*, 2018, pp. 149–161.
[12] I. Cho, A. Saeed, J. Fried, S. J. Park, M. Alizadeh, and A. Belay, "Overload control for microsecond-scale rpcs with breakwater," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020, pp. 299–314.
[13] J. Park, J. Park, Y. Jung, H. Lim, H. Yeo, and D. Han, "Topfull: An adaptive top-down overload control for slo-oriented microservices," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024, pp. 876–890.
[14] J. Xing, A. Giannoukos, P. Loh, S. Wang, J. Qiu, H. M. Demoulin, K. Kallas, and B. C. Lee, "Rajomon: Decentralized and coordinated overload control for latency-sensitive microservices," in *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Apr. 2025, pp. 21–36.

[15] A. E. Malki, U. Zdun, and C. Pautasso, "Impact of api rate limit on reliability of microservices-based architectures," in *2022 IEEE 16th International Conference on Service-Oriented System Engineering (SOSE)*, 2022, pp. 19–28.
[16] S. Serbout, A. E. Malki, C. Pautasso, and U. Zdun, "Api rate limit adoption: A pattern collection," in *Proceedings of the 28th European Conference on Pattern Languages of Programs*, 2023, pp. 1–20.
[17] Z. Li, H. Sun, Z. Xiong, Q. Huang, Z. Hu, D. Li, S. Ruan, H. Hong, J. Gui, J. He, Z. Xu, and Y. Fang, "Noah: Reinforcement-learning-based rate limiter for microservices in large-scale e-commerce services," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 9, pp. 5403–5417, Sep. 2023.
[18] A. Arman, M. Monticelli, D. Firmani, F. Leotta, and M. Mecella, "An adaptive toolbox for computing throttling rate limits in web apis," *IEEE Transactions on Services Computing*, vol. 19, no. 1, pp. 141–152, 2026.
[19] B. Farkiani, F. Liu, and P. Crowley, "Rethinking http api rate limiting: A client-side approach," in *2026 IEEE 23rd Consumer Communications and Networking Conference (CCNC)*, Jan. 2026, pp. 1–7.
[20] R. Jain, D.-M. Chiu, and W. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer systems," Digital Equipment Corporation, Tech. Rep. DEC-TR-301, Sep. 1984.
[21] H. S. Kim, E.-C. Park, and S. W. Heo, "A token-bucket based rate control algorithm with maximum and minimum rate constraints," *IEICE Transactions on Communications*, vol. E91-B, no. 5, pp. 1623–1626, May 2008.
[22] A. Saeed, N. Dukkipati, V. Valancius, V. T. Lam, C. Contavalli, and A. Vahdat, "Carousel: Scalable traffic shaping at end hosts," in *Proceedings of ACM SIGCOMM*, 2017, pp. 404–417.
[23] M. Welsh, D. Culler, and E. Brewer, "Seda: An architecture for well-conditioned, scalable internet services," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 230–243, Oct. 2001.
[24] D. Cotroneo, R. Natella, and S. Rosiello, "Draco: Distributed resource-aware admission control for large-scale, multi-tier systems," *Journal of Parallel and Distributed Computing*, vol. 192, p. 104935, Oct. 2024.
[25] R. T. Fielding, M. Nottingham, and J. Reschke, "Http semantics," Internet Engineering Task Force, Tech. Rep. RFC 9110, Jun. 2022.