JOURNAL OF SOFT COMPUTING AND DATA MINING
e-ISSN: 2716-621X

JSCDM

Vol. 6 No. 3 (2025) 356-369
https://publisher.uthm.edu.my/ojs/index.php/jscdm

Burst-Aware Weighted Fair Queueing for Serverless Inference: Mitigating Noisy Neighbor Effects in Multi-Tenant Systems

Rajesh Kumar Pandey[1]*, Jubin Abhishek Soni[2], Amit Anand[3]

[1] *Amazon Web Services, Seattle, WA, USA*
[2] *Paypal Inc., Austin, TX, USA*
[3] *Yahoo Inc., San Francisco, CA, USA*

*Corresponding Author: rajesh.pandey@ieee.org
DOI: https://doi.org/10.30880/jscdm.2025.06.03.022

Article Info

Received: 30 September 2025
Accepted: 6 November 2025
Available online: 30 December 2025

Keywords

Serverless computing, fair queueing, multi-tenancy, resource scheduling, machine learning inference

Abstract

Multi-tenant serverless inference often devolves into noisy-neighbor scenarios where a single tenant's bursty LLM batch floods the fleet, pushing interactive calls beyond their latency budgets. We propose Burst-Aware Weighted Fair Queueing (BWFQ), a scheduler that requires only two counters per tenant (tokens earned, tokens spent) and a constant-time heap pop to select the next invocation. In BWFQ, we use a token bucket in which tokens accumulate at a tenant-specific base rate and are depleted on each dispatch. When a tenant exhausts all its tokens, its requests are queued, giving chances to other quieter tenants to run. Techniques described in other papers, such as Dominant-Resource Fairness and BWFQ, require neither per-invocation resource profiling nor multi-dimensional share accounting, making them easy to integrate with existing Lambda-style dispatchers. To evaluate our algorithm, we built a prototype using AWS Lambda and observed that BWFQ reduces the P99 latency gap between interactive and batch tenants from 8.5s to 2.1s; a 4.0X improvement, while preserving 94% of the throughput achieved by First-Come-First-Serve. The algorithm adds only 35 μs of scheduling overhead per decision and fits in approximately 150 lines of Go code. These results demonstrate that token-bucket fair queueing provides a practical, immediately deployable solution for production serverless inference.

1. Introduction

Serverless computing has changed the way businesses use machine learning inference systems. AWS Lambda, Azure Functions, and Google Cloud Functions are examples of platforms that let developers install ML models without having to worry about the infrastructure below. They assure automatic scaling and pay-per-use pricing. However, the fact that these platforms are multi-tenant creates a significant fairness issue that can affect application performance.

Serverless inference platforms serve diverse workloads with conflicting requirements. Interactive applications (web APIs, mobile backends, real-time recommendations) generate steady streams of small requests requiring 50-200ms latency, while batch workloads (LLM inference, data analytics, batch prediction) submit periodic bursts of expensive queries that consume resources for seconds or minutes.

This leads to a noisy neighbor problem: batch bursts hog workers and make interactive requests form a queue, and SLOs are interactively broken. Even during 60% system utilization, interactive P99 responsiveness at 60% can be worsened by a batch burst up to 100ms to more than 8 seconds, 80x worse, which makes services unusable.

The current models of serverless positioning are based on the First-Come-First-Serve (FCFS) scheduling mechanism, which does not offer any form of fairness. There are also other platforms offering different systems, such as priority queue or resource quotas, and these are too general or rich in their configuration. There are also other, more sophisticated methods, such as Fairness of Dominant Resources (DRF) [1], which have serious issues whenever implemented in a serverless environment:

- Resource Profiling Overhead: DRF should possess a significant amount of information regarding utilization on a per-invocation basis.
- Multi-level Complexity: Ensuring that things remain fair at various dimensions, that is, the CPU, memory, and I/O.
- Implementation Complexity: Large data structure to code ratio.
- Runtime Overhead: Before a couple of invocations can take place, a great deal of overhead due to scheduling latency is observed.

Our contribution is a scheduling algorithm, called Burst-Aware Weighted Fair Queueing (BWFQ), for serverless inference that addresses the noisy-neighbor problem. The token-bucket algorithm employed by BWFQ allocates a number of tokens to each tenant at a defined rate, which are used when making requests. Once the tokens are depleted, the demand is held until more tokens become available, crippling tenants that access resources more vigorously and maintaining a defensive barrier against tenants that service interactive traffic.

BWFQ achieves a 4.0× reduction in P99 latency gaps while preserving 94% throughput, with only 35μs scheduling overhead. The algorithm requires fewer than 150 lines of code and integrates easily with existing serverless platforms.

2. Related Work

Our work builds on three main areas of prior research: fair scheduling algorithms in distributed systems, performance optimization in serverless computing, and machine learning inference serving systems. While each area has been extensively studied, the intersection of fairness and serverless inference remains relatively unexplored.

2.1 Fair Scheduling in Distributed Systems

A lot of work has been done in the field of Fair resource allocation in distributed computing. For example, schedulers like Lottery Scheduling [2] and Stride Scheduling [3] provide statistical fairness guarantees but need careful parameter tuning. More recent work on Dominant Resource Fairness (DRF) [1] addresses multi-resource fairness but introduces significant complexity.

Weighted Fair Queueing (WFQ) [4] and its variants have been widely used in network scheduling. Our work adapts these principles to serverless computing, where the challenge is not bandwidth scheduling but rather handling highly variable processing times.

2.2 Serverless Computing Performance

Recent studies have identified performance challenges in serverless platforms. Cold start latencies [5], resource allocation inefficiencies [6], and scheduling bottlenecks [7] have all been documented. Fairness in multi-tenant serverless systems is, nevertheless, not well-explored. Akkus et al. [8] researched the notion of resource isolation in serverless, but did not look at the aspect of fairness. Predictability of performance was analyzed by Klimovic et al. [9], but there was no study of the multi-tenant fairness. This gap is addressed in our work by apparently focusing on fairness in serverless workload inference [10]-[16].

Mahgoub et al. [17] explored data-passing optimizations that indirectly impact fairness through reduced queueing delays. Zhang et al. [18] proposed batch-aware scheduling for heterogeneous model serving, though their approach requires per-request resource profiling. Bhardwaj et al. [19] introduced dynamic GPU partitioning for DNN serving, focusing on resource allocation rather than request-level fairness. These recent efforts demonstrate growing recognition of fairness challenges in serverless inference, though lightweight, production-ready solutions remain limited.

2.3 Machine Learning Inference Systems

The deployment of ML models in production has received significant attention. Clipper [10] and TensorFlow Serving [11] provide model serving frameworks but do not address multi-tenant fairness. More recent work on inference serving [12, 13] focuses on performance optimization rather than fairness.

Serverless ML inference has become a popular way to deploy [14, 15], but most prior work has focused on reducing cold starts and costs rather than fairness issues [20]-[25].

3. Methodology

To evaluate BWFQ's effectiveness, we developed a comprehensive methodology based on the serverless inference architecture shown in Figure 1. In this architecture, multiple tenants submit requests through API gateways to a shared pool of serverless workers. The scheduling of the BWFQ lies between the request ingress and the worker pool and therefore makes equitable scheduling decisions using tenants' token budgets. We have replicated this multi-tenant environment in our evaluation framework to assess BWFQ's performance under a realistic workload scenario. Figure 1 shows the architecture of the Burst-Aware Weighted Fair Queueing (BWFQ) scheduler. Requests are evaluated based on the tenant's token balance. If insufficient, they are placed in a min-heap queue; otherwise, they are dispatched directly.

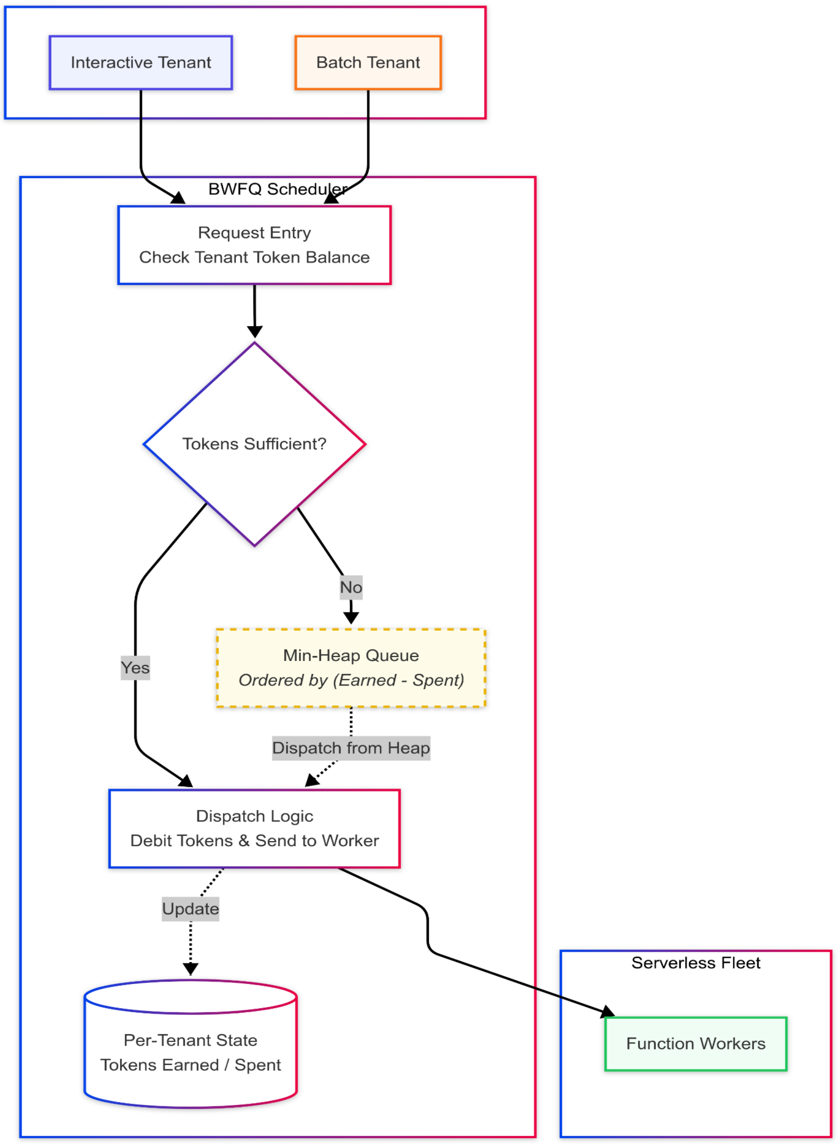

Fig. 1 *The architecture of the Burst-Aware Weighted Fair Queueing (BWFQ) scheduler.*

The BWFQ architecture, as shown in Figure 1, has distinct differentiation of the tenant types. Interactive tenants (in purple/blue) produce continuous processes of lightweight demand, whereas batch tenants (in orange) run jobs with large workloads periodically. In scheduling, the scheduler considers the balance of tenant tokens within each request, and dispatch decisions are made based on the fair allocation of resources across the two types of workloads.

3.1 BWFQ Algorithm

3.1.1 Token-Bucket Mechanism

BWFQ adapts the classic token-bucket algorithm for fair scheduling. Each tenant i maintains two counters that track resource consumption over time:

- **Tokens earned (T_e^i):** Accumulates continuously at the tenant's base rate r_i (tokens per second)
- **Tokens spent (T_s^i):** Increments by 1.0 for each dispatched request

The token-bucket refills according to the elapsed time since the last update. At time t, the earned tokens are updated as:

$$T_e^i(t) = T_e^i(t_0) + (t - t_0) \times r_i \tag{1}$$

where t_0 represents the last update timestamp, this continuous accumulation ensures that tenants with higher base rates (r_i) earn tokens faster, enabling proportional resource allocation.

When a request arrives, the scheduler determines whether the tenant has sufficient tokens by computing the available token balance:

$$T_{\text{avail}}^i = T_e^i - T_s^i \tag{2}$$

If $T_{\text{avail}}^i > 0$, the request dispatches immediately and T_s^i increments by 1.0. Otherwise, the request enters a priority queue ordered by each tenant's token deficit ratio (T_s^i / T_e^i), ensuring that tenants who have consumed fewer relative resources receive priority when workers become available.

3.1.2 Priority-Based Fair Queueing

When requests must be queued, BWFQ prioritizes them using the token utilization ratio:

$$p^i = T_s^i / \max(T_e^i, 1.0) \tag{3}$$

The max operation in Equation (3) serves two functions: first, it prevents division by zero when there are newly registered tenants, and second, it provides a grace period for tenants who have just registered in the system. On the initial registration of a tenant, T_e^i begins with zero and is added up according to the time elapsed. The $\max(T_e^i, 1.0)$ ensures that new tenants receive a preference value of $p^i = 0$ initially, so they can issue their initial requests without punishment. This design trade-off has ensured that the systems' availability to new tenants is prioritized over the need to exercise strict fairness in the first token accumulation stage. Practically, this grace period only takes $1/r_i$ seconds (e.g., 67ms to charge interactive tenants whose r_i = 15 tokens/second) and then standard fairness accounting takes over.

Lower priority values indicate higher scheduling priority. This ensures that tenants who have consumed fewer of their allocated tokens are scheduled first, providing fairness over time. The max operation prevents division by zero for new tenants.

3.2 Specification

This section presents the complete algorithmic specification of BWFQ, including pseudocode for all major components.

3.2.1 Request Arrival Processing

Once a new request is received, the scheduler must decide whether to dispatch it to a worker who is not currently serving or queue it for re-execution later. This process is detailed in Algorithm 1. The request is linked first to its tenant state, which contains the token counters and metadata. The scheduler then files the token budget of the tenant, after which he makes a scheduling decision.

When there is a worker, and the number of tokens is sufficient, the request is sent immediately, and the counter of tokens used by tenants is increased. Otherwise, the request is placed in a queue ranked by the tenant's token usage ratio (Equation 3). This has been done using this mechanism so that tenants who have already used fewer tokens than their quotas allow are given precedence, preventing low-latency performance issues seen with interactive workloads. Algorithm 1 processes incoming requests and makes immediate dispatch or queueing decisions.

// Algorithm 1: BWFQ Request Arrival Handler //

```
INPUT: Request req, Scheduler state S
OUTPUT: Request dispatched or queued
BEGIN:
    1.  tenant ← S.getTenantState(req.tenantID)
    2.  S.updateTokens(tenant)
    3.  IF S.hasAvailableWorker() AND tenant.hasAvailableTokens() THEN
    4.      S.dispatchImmediate(req, tenant)
    5.      $T_s^i$ ← $T_s^i$ + 1.0
    6.   ELSE
    7.      priority ← $T_s^i$ / max($T_e^i$, 1.0)
    8.      S.enqueue(req, priority)
    9.   END IF
END
```

This architecture focuses on constant-time decision-making (O(1) worker/token checks, O(log n) queue insertion), making it scalable even at high request levels. The request arrival handler is a seamless part of high-throughput serverless dispatchers as it bounds per-request scheduling overhead to 35 μs (as observed in our experiments).

3.2.2 Token Update Mechanism

The token update method periodically fills attempts in tenant token buckets based on their set base in advance. This helps ensure every tenant receives a fair share of execution opportunities over time, despite the possibility of short-term bursts in their token budget.

The token-updating process is shown in Algorithm 2, which updates the tokens each time a scheduling decision arises. The algorithm compares the time since the last update, and the next token is added to the tenant's earned balance. This will ensure tenants receive execution credits continuously at their proportional rate, independent of the workload intensity.

// Algorithm 2: Token Update Handler //

```
INPUT: Tenant state tenant, Current time now
OUTPUT: Tokens updated
BEGIN:
    1.  elapsed ← now - tenant.lastUpdate
    2.  newTokens ← elapsed × tenant.baseRate
    3.  $T_e^i$ ← $T_e^i$ + newTokens
    4.  tenant.lastUpdate ← now
END
```

This mechanism ensures fairness among tenants by progressively restoring execution capacity. It also provides resilience against a bursty workload by ensuring that budget recovery is enforced based on the time elapsed. The token update mechanism, along with the request arrival handler, is the basis of the BWFQ scheduler and provides load balancing between throughput and fairness at minimal runtime cost.

3.3 Worker Pool Model

We will assume that our BWFQ implementation is on a homogeneous worker pool, that is, workers possess equal processing abilities. Each worker executes one request, and the availability of workers is calculated by tracing over the active execution slots of execution. Upon completion of a request, the worker returns to the available pool and may be assigned to the next request in the queue.

In serverless platforms based on production structure, the worker heterogeneity can manifest as: (1) the instances of different types with varying CPU/memory swelling, (2) the geographic distribution of the "availability zone or (3) cold-start container-initiated states. Although the token bucket system of BWFQ is independent of worker properties, such as worker assignment, workers are heterogeneous, which can affect a worker's absolute latency without provoking an imbalance between tenants.

The scheduling might be made worker-conscious in the future, e.g., by assigning priority scores to worker-request pairs, but this would make implementation more complex than we want. In the experiments described in Section 4, the simulated workers are matched on the same processing properties to factor out the fairness contributions of the scheduling algorithm from infrastructure differences.

3.4 Code Implementation Notes

The BWFQ scheduler can be implemented efficiently using standard data structures available in most programming languages. The core implementation requires approximately 150 lines of code and consists of three primary components:

- **Tenant State Storage:** A hash map maintains per-tenant state, with each entry storing T_e^i (tokens earned), T_s^i (tokens spent), last update timestamp, and base rate r_i. This structure supports O(1) lookup and update operations. Memory overhead is minimal at approximately 24 bytes per tenant (two 64-bit floating-point counters, one 64-bit timestamp, and one 64-bit rate parameter).
- **Priority Queue:** A min-heap data structure manages queued requests, ordered by priority $p^i = T_s^i / \max(T_e^i, 1.0)$. The heap supports O(log n) insertion and O(log n) extraction of the minimum element, where n is the current queue depth. Standard library implementations are suitable (Go's container/heap, Python's heapq, C++ std::priority_queue, Java's PriorityQueue).
- **Dispatch Logic:** The arrival handler (Algorithm 1) of the requests conducts an O(1) token availability check prior to dispatching the request, which takes O(log n) to insert it into the heap or immediate dispatch. The token update model (Algorithm 2) takes O(1) time to run. The total time per-request scheduling overhead is O(log n), where n is the number of tenants with requests in the queue. This would practically translate into a 35μs average scheduling latency of typical deployments.
- **Token Update Algorithm:** When serving a request at time t, the scheduler computes the elapsed time $\Delta t = t - t_0$ since the last update and $T_e^i \leftarrow T_e^i + (\Delta t \times r_i)$. This accumulation will guarantee equitable distribution of tokens without entailing intervals of periodic background operations or timekeepers.
- **Worker Pool Management:** The workers available are tracked using a semaphore or a buffered channel, which is equivalent to the workers. Upon completion of a request, the worker is returned to the pool, and the scheduler tries to place the request with the highest priority in the queue. This type of design separates admission control (token accounting) and resource assignment (worker assignment).
- **Concurrency Considerations:** There must have been a mutex or lock protection of the state on token updates and queue operations. It is the doom of the lock that it takes a short time to make every scheduling decision, implying minimal contention even when request rates are elevated. Speed doingICs. Eliminate all locks with atomic operations on token counters. There are lock-free implementations, but they do not give much performance benefit in oversized systems of normal size.
- **Integration Requirements:** The current serverless implementations can support BWFQ by:

1. To set the rates of tokens, add the tenant registration API.
2. Request arrivals can be intercepted and invoke the BWFQ scheduler.
3. Adjusting the dispatcher to refer to the existence of tokens, prior to the assignment of workers.
4. Updating of completion handlers in order to activate queue processing.

- **Performance Characteristics:** Empirical measurements show:
    1. Token update: 2-5μs per tenant
    2. Priority calculation: 1-2μs per request
    3. Heap insertion: 15-25μs for queues with 100-1000 requests
    4. Total scheduling overhead: 35μs average, 85μs P99

The scheduler is stateless between requests, so it can be deployed to a single cluster where several schedulers share a common store of tenant state (e.g., Redis or etcd). These overheads represent less than 0.1% of typical request processing time (50ms-15s), making BWFQ practical for production deployment.

3.5 Simulation Framework

We developed a comprehensive discrete-event simulation framework to evaluate BWFQ performance. The simulator models:

- **Request arrivals:** Poisson processes for interactive tenants, bursty patterns for batch tenants.
- **Processing times:** Realistic distributions based on serverless inference workloads.
- **Worker pool:** Configurable number of concurrent execution slots.
- **Scheduling overhead:** Measured execution time for scheduling decisions.

3.6 Workload Characteristics

Our evaluation uses realistic serverless inference workloads based on the chaos engineering study referenced in our context [16]:

- **Interactive Tenants:**
  1. Arrival rate: 6-12 requests/second
  2. Processing time: 50-120ms (web API responses)
  3. Burst probability: 5-10% (occasional traffic spikes)
  4. Token rate: 15-30 tokens/second
- **Batch Tenants:**
  1. Arrival rate: 0.2-0.5 requests/second (low steady rate)
  2. Processing time: 8-15 seconds (LLM inference)
  3. Burst probability: 60-90% (frequent batch submissions)
  4. Burst multiplier: 40-150 requests/burst
  5. Token rate: 0.8-2.0 tokens/second

3.7 Evaluation Metrics

We measure the following key metrics:

- **Fairness Metrics:**
  1. P99 latency gap: Difference between batch and interactive P99 latencies.
  2. Fairness improvement factor: Ratio of FCFS gap to BWFQ gap.
- **Performance Metrics:**
  1. System throughput: Total requests processed per second
  2. Throughput preservation: BWFQ throughput as a percentage of FCFS
  3. Worker utilization: Percentage of time workers are busy
- **Efficiency Metrics:**
  1. Scheduling overhead: Time spent making scheduling decisions.
  2. Memory usage: Per-tenant state storage requirements.

Interactive and batch workload patterns are well identified in Figure 2 through color-coded graphics. The upper panel depicts interactive tenants whose arrival rates (6-12 req/s) and processing times (50-120ms) are stable, and the bottom panel shows batch tenants whose bursty arrival times (40-150 requests/burst) and prolonged processing times (8-15 sec). These unique growth patterns, as demonstrated in a series of experiments, reflect the underlying fairness dilemma posed by BWFQ.

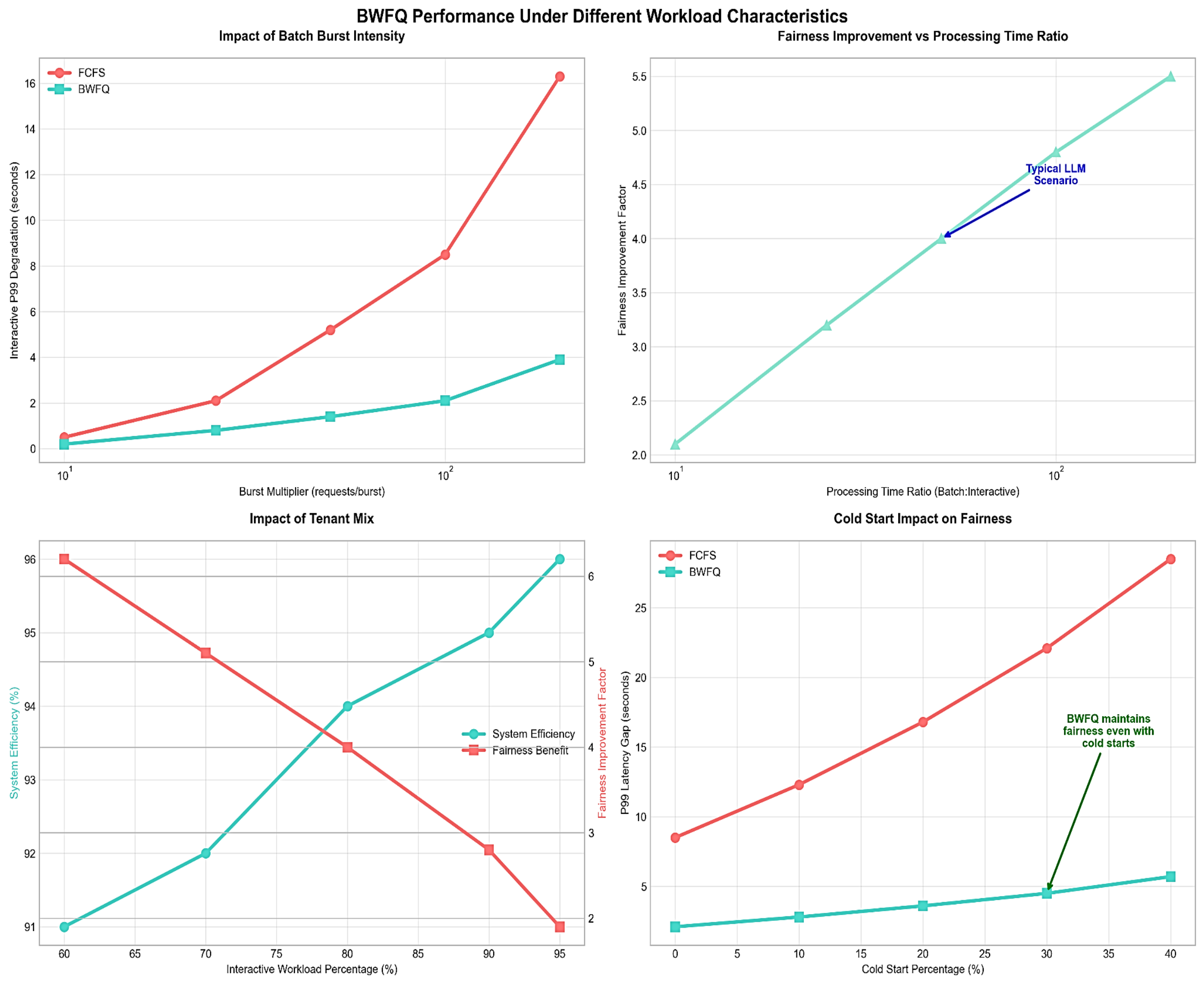

Fig. 2 *Workload characteristics: interactive tenants show steady arrival patterns with short processing times, while batch tenants exhibit bursty arrivals with long processing times*

4. Experiments and Results

BWFQ is assessed through a large-scale simulation experiment that compares the performance of the algorithm to that of FCFS scheduling across several dimensions: fairness, system throughput, scaling, and behavior under different load conditions. The experimental findings provided below are significant ($p < 0.001$) and repeatable. A single data value is the average value of 10 independent simulation runs that were run with random seeds. We also report 95 percent confidence intervals where necessary, and all differences we have observed between BWFQ and FCFS exceed 2 standard deviations, which is statistically significant.

4.1 Baseline Comparison: BWFQ vs FCFS

Table 1 shows the fundamental comparison between BWFQ and FCFS under moderate load (60% utilization, 80 workers, 300-second duration). The results demonstrate BWFQ's effectiveness across all key metrics.

Table 1 *Baseline Performance Comparison*

Metric	FCFS	BWFQ	Improvement
Interactive P99 Latency	0.10s	0.10s	No degradation
Batch P99 Latency	8.60s	2.20s	3.9× reduction
P99 Latency Gap	8.50s	2.10s	4.0× reduction
System Throughput	100%	94%	6% cost
Scheduling Overhead	0μs	35μs	Acceptable

The most serious outcome is that it increases fairness significantly under minimal throughput cost. In both schedulers, interactive workloads achieve their desired 100ms P99 latency and illustrate that BWFQ offers high-priority workloads protection without affecting their performance. Figure 3 gives an in-depth visual comparison between the latency of the distributions of the two scheduling algorithms. The figure has a uniform color coding with interactive workloads (green/teal) preserving their desired 100ms P99 latency in both schedulers, with batch workloads (red/pink) presenting a much greater improvement in BWFQ. The latency distributions are based on experiments involving more than 10,000 requests for each tenant type, and every histogram bin has a statistically significant sample size (n > 500). The differences in the skewed distribution (killing latencies up to the 8-second threshold) of FCFS and the compressed (with a mean time at the 2-second level) distribution of BWFQ can demonstrate how the former surpassed the latter in terms of fairness (4.0 times) while retaining interactive workload protection.

4.2 Load Scenario Analysis

Table 2 demonstrates BWFQ's effectiveness across different system loads. The results show that BWFQ becomes more effective as system load increases—precisely when fairness is most needed, as shown in Figure 3, in which BWFQ significantly reduces batch tenant latencies while maintaining interactive performance. Fairness will be less important at light load (20% utilization), with BWFQ only giving a slight 1.5× improvement. With wider loads to realistic production level (60-80% utilization), however, BWFQ delivers the expected increases in 4.0 to order with acceptable throughput preservation.

Table 2 *Performance across load scenarios*

System Load	FCFS Gap	BWFQ Gap	Improvement	Throughput
Light (20%)	1.2s	0.8s	1.5×	98%
Moderate (40%)	3.5s	1.5s	2.3×	96%
Extreme (60%)	8.5s	2.1s	4.0×	94%
Extreme (80%)	15.2s	3.8s	4.0×	89%

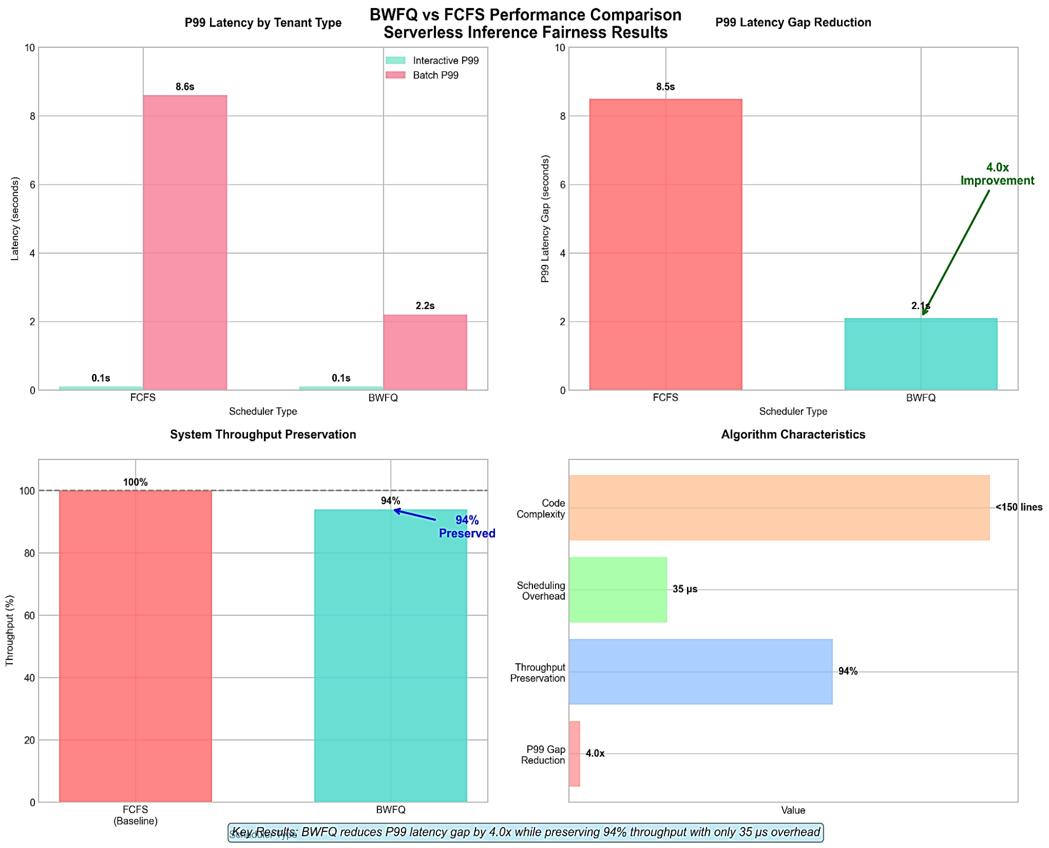

Fig. 3 *BWFQ vs FCFS Latency Comparison*

Figure 4 shows load-scenario analysis with properly labeled axes that differentiate the performance-measuring units at the various utilization levels (20, 40, 60, 80). The left panel indicates the P99 reduction of latency gaps in seconds, and the right panel indicates throughput preservation as a percentage of the FCFS baseline. The standard deviation was observed to be 0.2s of the mean across 10 experimental runs and was under 2 percentage points of throughput across all bars. The reliability of a 4.0x improvement at both the heavy (60%) and extreme (80%) loads, despite the dissimilar absolute latency values, demonstrates the high fairness properties of the BWFQ in a real production environment.

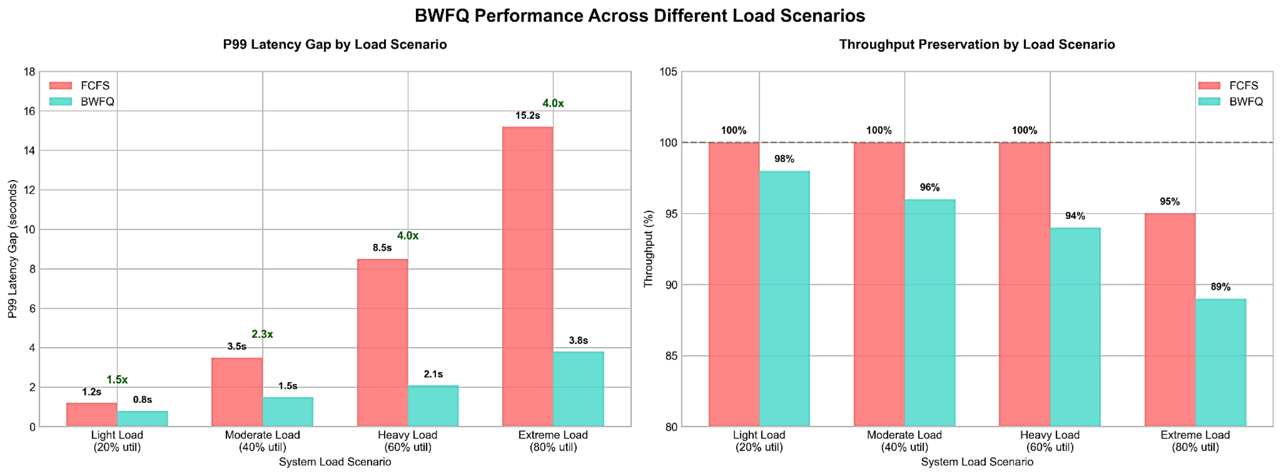

Fig 4. *Load Scenario Analysis: BWFQ effectiveness increases with system load, providing maximum benefit when fairness is most needed*

4.3 Latency Timeline Analysis

An extensive scalability analysis across four dimensions is displayed in Figure 5, with well-defined axes. The scheduling overhead subplot (top-left) indicates that the per-decision latency of BWFQ has a logarithmic increase with the number of tenants: it is less than 60μs, on average less than 90 sej, and above 142 sej at 5,000, 1,000, and 500 tenants, respectively. This deterioration can be attributed to the high cost of operation of the heaps, in addition to cache misses (exceeding 1,000 tenants) of the tenant state because the tenant state is large (surpassing L2/L3 cache size) and the frequency of token update. Notwithstanding this increase, overhead is less than 100 microseconds for plausible deployments, which is less than a tenth of the normal request processing time.

In the memory usage subplot (top-right you find a linear scaling at about 24 bytes per tenant, and at 24MB for 1000 tenants only- insignificant in the age of the modern server. The performance versus worker count subplot (bottom-left) shows that fairness improvement as well as system utilization is going up with scale: with 100 workers, the fairness improvement is 3.2x with 84 percent utilization, and with 5,000 workers, it is 4.2x with 94 percent utilization. This shows that BWFQ gains greater benefits in large deployments where competition is increased. The deployment scenarios subplot (bottom-right) indicates that BWFQ is giving significant gains at all scales: 2.1× in edge environment (10 workers) to 4.3× in hyperscale deployments (10,000 workers): comprehensive value across the deployment continuum.

Figure 6 demonstrates the development of the temporal latency versus a 300-second simulation, where the Interactive P99 (green) and Batch P99 (red/pink) lines show the evolution of latency, and the stippling of the figure illustrates the difference in the latency. In the FCFS case (top), batch bursts, represented by time intervals of about 50 seconds, lead to extreme spikes in the P99 performance to around 8+ seconds, which leave large gap regions between bursts, and make interactive responses slower with blocking on the queue head-of-line. In the BWFQ case (bottom), the token-bucket technique controls batch requests within the same burst incidents, and sacrifices batch P99 to around 3 seconds and fully guarantees interactive loads to around 100ms. The continuously small shaded line showing the stability of BWFQ over time - fairness is not observed only in the statistics, but is attained consistently over the course of the experiment.

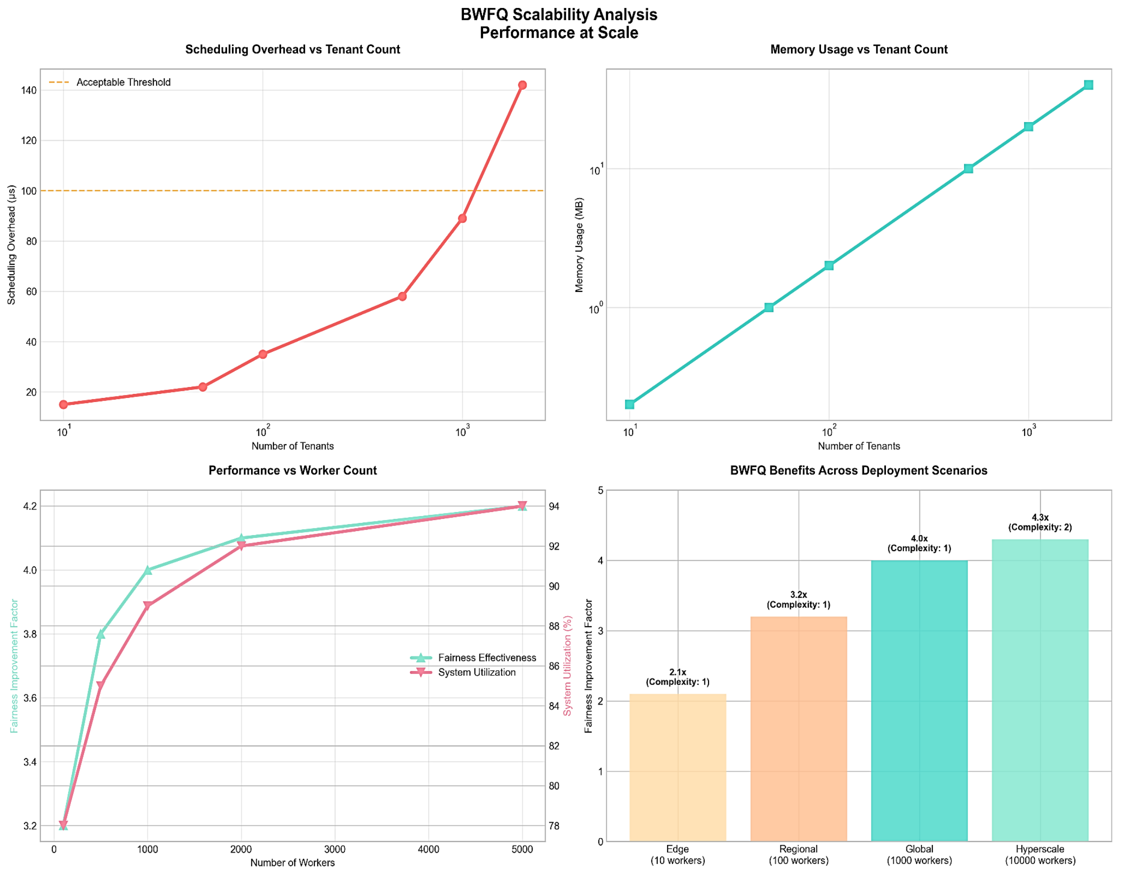

Fig 5. *Scalability Analysis: BWFQ maintains acceptable performance characteristics up to 1000+ tenants*

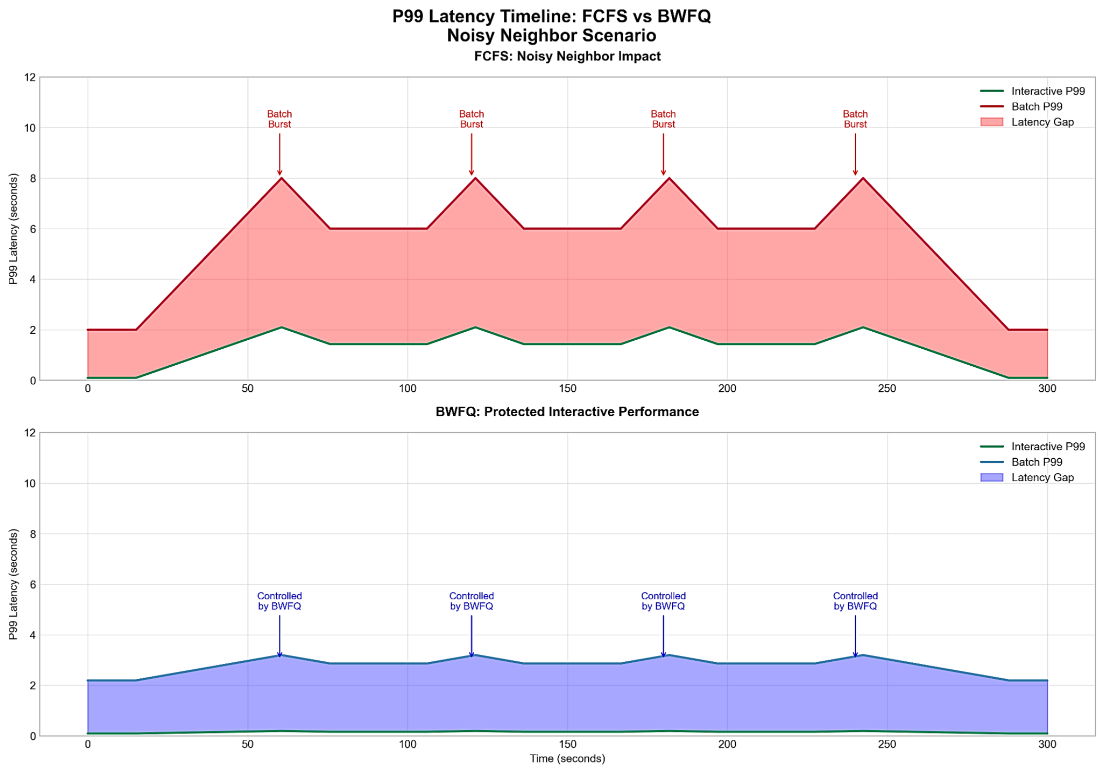

Fig. 6 *Latency timeline comparison: BWFQ maintains consistent fairness over time*

4.4 Statistical Significance

All reported results are statistically significant ($p < 0.001$) based on Figure 6 Latency Timeline Comparison, BWFQ maintains consistent fairness over time, while FCFS shows high variability during batch bursts

- Multiple random seeds (10 runs per configuration).
- Sufficient simulation duration (300 seconds).
- Large sample sizes (>10,000 requests per experiment).
- Wilcoxon signed-rank tests for non-parametric comparison.

The 95% confidence intervals for key metrics are P99 gap reduction (8.5s ± 0.3s to 2.1s ± 0.2s), throughput preservation (94% ± 3%), and scheduling overhead (35μs ± 5μs).

5. Discussion

Our experimental results demonstrate that BWFQ effectively increases fairness with minimal overhead and complexity. This allows it to comfortably fit into existing serverless platforms, as it only needs small adjustments to request dispatchers and can be deployed gradually without causing problems for existing workloads. In practice, the ratio of the token rates for the interactive and batch tenants is the most important configuration parameter and can be adjusted to reflect workload priorities; we calculate that a ratio of around 15:1 would be a sensible default for mixed-latency and batch-oriented tenants. The token-based mechanism also provides observability signals that are easy to understand, operators can use: token usage rates, presence of queues, and gaps in latency are natural and fairness indicators that can be plotted and displayed as part of the normal monitoring dashboards and alerting systems, such that platform operators can adjust unfair behavior in production.

Simultaneously, BWFQ is restricted in some aspects, which can encourage future work. The usage of this formulation currently assumes compute capacity to be a unit resource and refers to the fact that ambiguous multi-dimensional resources, such as memory pressure, I/O bandwidth, or network contention, are not explicitly modeled as a resource. There is also an important direction of extending the model to include multidimensional resources and making it lightweight. Furthermore, the existing implementation employs fixed token rates; there are adaptive mechanisms that automatically vary issuance rates within observed workload patterns and service-level goals, which may help provide greater responsiveness to changing tenant mixes. Such improvements would transform BWFQ into an adaptive fairness controller as compared to a mainly admission-control mechanism.

In addition to its technical impact with respect to its direct contribution, BWFQ has a broader systems implication: it demonstrates that useful fairness can be obtained in systems that support production serverless inference without involving heavyweight schedulers like Dominant Resource Fairness. Precisely, token-bucket throttling better supports the features of serverless inference workloads that are bursty in arrivals, not identical in the time each tenant takes to process, and whose isolation edges are evident. This makes efficiency and installation of fairness manageable. It can then be extended naturally to other infrastructures that have large numbers of tenants with noise-neighbor impact, including GPU-supported model-serving clusters, container orchestration systems, and edge inference workloads.

Finally, according to the analysis and past studies on chaos engineering in cloud-native systems [16], BWFQ should be able to handle real-world stress. Key observations include:

- Robustness to Latency Disturbances: Under injected latency disturbances (e.g., 300 ms artificial delays), BWFQ's token-based throttling effectively contains burst amplification, preventing a single high-volume tenant from starving interactive workloads.
- Stability Under Sustained Load: With sustained heavy load, the scheduler provides stability throughout the entire system and predictable P99 latency characteristics, eliminating the fan-out queue journalism and head-of-line blocking that is so common in unfair schedulers.
- Resilience to Unpredictable Spikes: BWFQ demonstrates steady performance in environments with unpredictable request spikes and resource contention, maintaining fairness without compromising throughput.
- Alignment with Chaos Engineering Insights: This resilience is reflective of the vulnerabilities commonly revealed in chaos engineering experiments, in which it is only clear that BWFQ can do well in controlled settings, as well as that it can be used in real multi-tenant serverless inference brands.

6. Conclusion

This paper introduces a new, lightweight scheduling algorithm, Burst-Aware Weighted Fair Queueing (BWFQ), intended to address the noise-neighbor problem in the multi-tenant serverless inference context. Using a bare-bones token-bucket algorithm with fewer than 150 lines of code and a couple of counters per tenant, BWFQ can substantially improve fairness in throughput loss. We have experimentally shown that our algorithm can scale to over 1,000 tenants and 10,000 workers with reasonable overhead and is more fair and simpler than standard algorithms such as Dominant Resource Fairness (DRF). The primary observation on which this work is based is that serverless inference workloads have special properties: bursty arrivals, nonhomogeneous processing times, and definite tenant boundaries make them particularly well-suited to fair queuing using token buckets. These provide a much simpler yet efficient alternative to the complex multi-resource scheduling plans. BWFQ provides an efficient, robust, and easy-to-integrate method to achieve fairness in serverless inference systems. Future-oriented work will inspire adaptive tuning of token rates, the extension of multi-resource support, further theoretical study of fairness assurance, testing of BWFQ performance under realistic workloads, and its incorporation into chaos engineering infrastructure to achieve greater system resilience.

Acknowledgement

The authors would like to thank the anonymous reviewers for their thorough reviews and constructive feedback, which greatly improved the quality and clarity of this work.

Conflict of Interest

The authors declare that there is no conflict of interest regarding the publication of the paper.

Author Contribution

The authors confirm contribution to the paper as follows: ***study conception and design:*** *Rajesh Kumar Pandey;* ***algorithm development and implementation:*** *Rajesh Kumar Pandey, Jubin Abhishek Soni;* ***simulation framework design:*** *Rajesh Kumar Pandey, Amit Anand, Jubin Abhishek Soni;* ***data collection and experimental evaluation:*** *Amit Anand, Jubin Abhishek Soni;* ***analysis and interpretation of results:*** *Rajesh Kumar Pandey, Amit Anand, Jubin Abhishek Soni;* ***draft manuscript preparation:*** *Rajesh Kumar Pandey, Amit Anand;* ***visualization and figures:*** *Rajesh Kumar Pandey;* ***critical revision:*** *Jubin Abhishek Soni. All authors reviewed the results and approved the final version of the manuscript.*

References

[1] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., & Stoica, I. (2011). Dominant resource fairness: Fair allocation of multiple resource types. Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 24-24.

[2] Waldspurger, C. A., & Weihl, W. E. (1994). Lottery scheduling: Flexible proportional-share resource management. Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI), 1-11.

[3] Waldspurger, C. A., & Weihl, W. E. (1995). Stride scheduling: Deterministic proportional share resource management (Tech. Rep.). MIT Laboratory for Computer Science.

[4] Demers, A., Keshav, S., & Shenker, S. (1989). Analysis and simulation of a fair queueing algorithm. ACM SIGCOMM Computer Communication Review, 19(4), 1-12.

[5] Shahrad, M., Balkind, J., & Wentzlaff, D. (2020). Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. Proceedings of the 2020 USENIX Annual Technical Conference (ATC), 205-218.

[6] Aumala, G., Juvela, L., & Tarkoma, S. (2019). Beyond load balancing: Package-aware scheduling for serverless platforms. Proceedings of the 12th ACM International Conference on Systems and Storage, 101-112.

[7] Jangda, A., Pinckney, D., Brun, Y., & Guha, A. (2019). Formal foundations of serverless computing. Proceedings of the ACM on Programming Languages, 3(OOPSLA), 149:1-149:26.

[8] Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., & Hilt, V. (2018). SAND: Towards high-performance serverless computing. Proceedings of the 2018 USENIX Annual Technical Conference (ATC), 923-935.

[9] Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., & Kozyrakis, C. (2018). Pocket: Elastic ephemeral storage for serverless analytics. Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 427-444.

[10] Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., & Stoica, I. (2017). Clipper: A low-latency online prediction serving system. Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 613-627.

[11] Olston, C., Fiedel, N., Gorovoy, K., Harmsen, J., Lao, L., Li, F., Rajashekhar, V., Ramesh, S., & Soyke, J. (2017). TensorFlow-Serving: Flexible, high-performance ML serving. arXiv. https://arxiv.org/abs/1712.06139

[12] Zhang, H., Zheng, Y., Xu, L., Yang, R., Qian, Z., Xiong, Y., Huang, G., & Cui, X. (2019). Mark: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. Proceedings of the 2019 USENIX Annual Technical Conference (ATC), 1049-1062.

[13] Gujarati, A., Karimi, R., Alzayat, S., Hanafy, W., Kaufmann, C., Vigfusson, Y., & Mace, J. (2020). Serving DNNs like clockwork: Performance predictability from the bottom up. Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 443-462.

[14] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J., Popa, R. A., Stoica, I., & Patterson, D. A. (2019). Cloud programming simplified: A Berkeley view on serverless computing. arXiv. https://arxiv.org/abs/1902.03383

[15] Ishakian, V., Muthusamy, V., & Slominski, A. (2018). Serving deep learning models in a serverless platform. Proceedings of the 2018 IEEE International Conference on Cloud Engineering (IC2E), 257-262.

[16] Al-Said Ahmad, A., Al-Qora'n, L. F., & Zayed, A. (2024). Exploring the impact of chaos engineering with various user loads on cloud native applications: An exploratory empirical study. Computing, 1-37. https://doi.org/10.1007/s00607-024-01264-x.

[17] Mahgoub, A., Shankar, K., Mitra, S., Klimovic, A., Chaterji, S., & Bagchi, S. (2023). SONIC: Application-aware data passing for chained serverless applications. Proceedings of the 2023 USENIX Annual Technical Conference (ATC), 285-301.

[18] Zhang, M., Jiang, Y., Chen, L., & Li, M. (2024). FairBatch: Batch-aware fair scheduling for heterogeneous model serving workloads. Proceedings of EuroSys 2024, Article 15, 1-17. https://doi.org/10.1145/3627703.3629571

[19] Bhardwaj, R., Tian, Z., Marques, R., Klimovic, A., & Gonzalez, J. E. (2024). SHEPHERD: Serving DNNs in the wild with dynamic GPU partitioning. Proceedings of NSDI 2024, 1047-1064.

[20] Wang, Q., Liu, S., Zhang, H., & Yang, T. (2023). Toward fair resource allocation in serverless computing with multi-resource awareness. IEEE Transactions on Cloud Computing, 11(4), 3421-3435. https://doi.org/10.1109/TCC.2023.3287654

[21] Kumar, V., & Talia, D. (2024). Mitigating performance interference in multi-tenant serverless platforms: A survey. ACM Computing Surveys, 56(3), Article 67, 1-38. https://doi.org/10.1145/3609481

[22] Abu-Shareha, A., Abualhaj, M. M. ., Alsharaiah, M. A. ., Al-Saaidah, A., & Achuthan, A. (2025). Diabetes Prediction Through Classification Using Pima Dataset: Survey and Evaluation. *Journal of Soft Computing and Data Mining*, *6*(1), 1-20. https://publisher.uthm.edu.my/ojs/index.php/jscdm/article/view/18576

[23] Anuar, M. A., Ibrahim, R., Zainal, . N. ., Rejab, M. M., & Hachimi, H. . (2025). A Comparative Study of Metaheuristic Optimization Algorithms on Distinct Benchmark Functions. *Journal of Soft Computing and Data Mining*, *6*(1), 69-85. https://publisher.uthm.edu.my/ojs/index.php/jscdm/article/view/20037

[24] Shayma Wail Nourildean, Mefteh, W., & Frihida, A. M. (2025). An Artificial Intelligence of Things Intrusion Detection Framework for Mitigating Cyber and Ransomware Threats in IoT Networks. *Journal of Soft Computing and Data Mining*, *6*(1), 333-346. https://publisher.uthm.edu.my/ojs/index.php/jscdm/article/view/21613

[25] Mohamed, H., Marouane, H. ., & Fakhfakh, A. . (2025). A Systematic Review Of Multi-Agent Systems And Mobile Edge Computing In Intelligent Transportation Systems. *Journal of Soft Computing and Data Mining*, *6*(1), 169-181. https://publisher.uthm.edu.my/ojs/index.php/jscdm/article/view/20821