International Journal of AI, BigData, Computational and Management Studies
Noble Scholar Research Group | Volume 6, Issue 4, PP 161-169, 2025
ISSN: 3050-9416 | https://doi.org/10.63282/3050-9416.IJAIBDCMS-V6I4P118

Original Article

Priority Queues in Large-Scale Distributed Systems under Heavy Workloads

Milan Gupta
Independent Researcher, USA.

Received On: 28/09/2025 Revised On: 31/10/2025 Accepted On: 11/11/2025 Published On: 22/11/2025

Abstract: *Large-scale distributed systems often face periods of heavy workloads where a surge in tasks can overwhelm processing capacity. In such scenarios, critical operations risk being delayed as they compete with a backlog of routine tasks. Priority queues offer a solution by ensuring that high-priority work is serviced ahead of less critical tasks, thereby preserving system responsiveness and meeting service-level objectives. This paper examines the importance of priority queuing in distributed systems handling massive workloads, highlighting how the Priority Queue pattern can prevent important tasks (such as monitoring queries or high-value data updates) from being starved of resources. We discuss real-world scenarios—including an overloaded database where monitoring queries get stuck, and a catalog processing pipeline where urgent updates languish behind bulk jobs—and illustrate design patterns that mitigate these issues. We explore implementations ranging from cloud infrastructure (e.g., AWS SQS and workload management) to industry-scale solutions like Facebook's FOQS distributed priority queue. Through diagrams and case studies, we show how priority-based task scheduling improves observability, data freshness, and overall system reliability under heavy load. The paper provides an academic discussion in a formal tone, supported by references to established patterns and recent industry and research insights.*

Keywords: *Priority Queuing, Overload Mitigation, Monitoring Query Prioritization, Catalog Update Pipelines, Fast-Lane Processing, Multi-Tenant Workload Isolation, Resource Starvation Prevention, Distributed Scheduling, Backpressure Mechanisms, High-Value Data Updates, Workload Imbalance.*

1. Introduction

In large-scale distributed systems, managing workloads under heavy load is a critical challenge. When a system is inundated with a high volume of tasks, all tasks are not equally time-sensitive. Some operations (e.g. user-facing or control-plane tasks) may require immediate attention, while others (e.g. batch processing or background jobs) can tolerate delays. If the system processes tasks strictly in a first- in-first-out (FIFO) manner or without differentiation, high-priority tasks can become stuck behind a backlog of lower-priority work, leading to unacceptable delays or system issues. This is where priority queues and priority-based scheduling become essential. By assigning priorities to tasks and ensuring that higher-priority tasks are serviced first, distributed systems can maintain responsiveness for critical operations even under heavy workloads.

In this paper, we explore the importance of priority queue mechanisms in large-scale distributed systems, especially under heavy load. We discuss scenarios where lack of priority handling causes problems, examine design patterns and architectures that incorporate priority queues, and review industry and academic examples. We also address the challenges and trade-offs involved in implementing priority scheduling (such as fairness and starvation). Throughout, we take an academic perspective with references to both research and real-world systems (with a focus on widely-used technologies, including AWS-based stacks).

2. Background: Priority Queues and Distributed Scheduling

Priority queue is a fundamental abstract data type in which each element is associated with a "priority" and the element with highest priority is served before others. In practice, a priority queue can be implemented with data structures like heaps to allow efficient insertion and retrieval of the highest-priority item. Many scheduling algorithms in computer science leverage priority queues to decide the order of task execution. For example, operating systems often maintain ready threads in priority queues, and network routers use priority queues to forward high-priority packets first. The general idea is that by ordering tasks according to priority, the system can give preference to urgent tasks and improve responsiveness for those tasks.

In distributed systems and cloud services, priority-based scheduling is a common strategy to achieve Quality of Service (QoS) guarantees. Many platforms support some notion of priority. For instance, Amazon Redshift's workload manager allows configuring query queues with different priorities, *"so that short, fast-running queries don't get stuck in queues behind long-running queries."*. Similarly, real-time systems and high-performance computing (HPC) clusters permit high-priority jobs to preempt lower-priority jobs when necessary. The goal in all cases is to prevent critical or time-sensitive tasks from being delayed by large volumes of less critical work.

Priority Queue Pattern: In a distributed application, the priority queue pattern refers to designing the task workflow such that tasks are prioritized either by using a single prioritized queue or multiple separate queues for different priority levels. In a single prioritized queue model, producers assign a priority value to each task/message, and the queue (or broker) internally orders messages by priority. Consumers then receive higher-priority messages before lower-priority ones. Figure 2.1 illustrates this concept: an application tags outgoing messages with a priority, a priority queue reorders them (so, for example, "High Priority" messages will be dequeued ahead of "Low Priority" ones), and consumers process tasks in priority order.

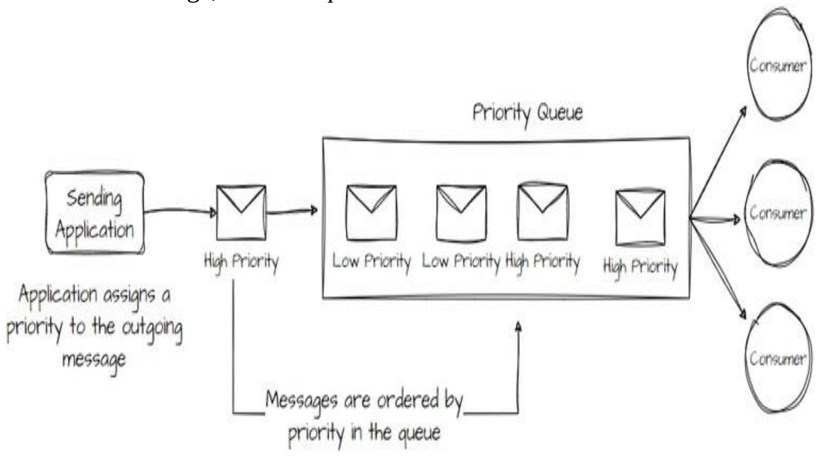

Fig 1: Conceptual Illustration of a Single Priority Queue in a Messaging System.

The Sending Application Attaches Priority Labels to Messages (E.G. High vs Low Priority). The Priority Queue Orders Messages Internally by Priority, Ensuring Consumers Receive and Process Higher-Priority Messages First (Ahead of Lower Priority Messages).

In contrast, an alternative implementation is to use multiple queues partitioned by priority. In a multi- queue design, there might be (for example) one queue for high-priority tasks and another for normal (or low-priority) tasks. The producers direct each message to the appropriate queue based on priority. Consumers can then be allocated to each queue in a way that high-priority queues get more processing resources or faster service. This pattern is effectively a "fast lane" for urgent tasks: high-priority tasks bypass the congestion in the normal queue by going into their own dedicated queue (or lane). An example is using multiple AWS SQS queues or separate Kafka topics for each priority level, as we will discuss later. Figure 2 shows an architecture with separate queues for high and low priorities, each possibly served by its own pool of consumers.

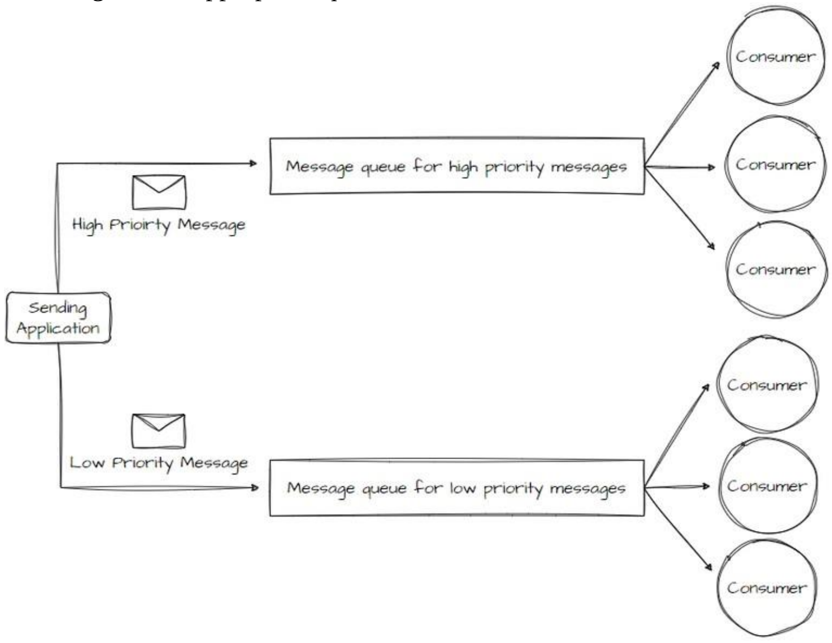

Fig 2: Example of a Multi-Queue Priority Architecture.

The sending application places high-priority messages into a dedicated high-priority queue, and low-priority messages into a low-priority queue. Separate consumer processes (or threads) may service each queue. This separation ensures that a flood of low-priority tasks does not delay the processing of high-priority tasks. In practice, more consumers or compute resources can be allocated to the high-priority queue to expedite important tasks.

Priority Scheduling in Practice: Many real-world systems implement one of these patterns. As mentioned, Amazon Redshift allows defining multiple query queues with a Workload Management (WLM) configuration. This enables, for example, isolating quick operational queries or monitoring queries into their own queue so they won't be blocked behind heavy analytical queries. Under Redshift WLM, *“short queries don’t get stuck in queues behind long-running queries,”* illustrating the benefit of priority-based isolation. Another example is message queuing systems: AWS Simple Queue Service (SQS) does not natively support message prioritization in a single queue, so AWS architects often implement priority by using multiple SQS queues for different priority levels, and assigning more consumer Lambda functions or EC2 workers to the high-priority queue. In fact, AWS recommends allocating separate concurrency limits for consumer processes on each queue, such as heavily weighting concurrency toward the high-priority queue (e.g. 90% of workers on the high-priority queue and 10% on the low-priority) to ensure critical tasks are processed first. Microsoft Azure’s service bus also doesn’t directly support priority on a single queue, so a common solution is to use multiple subscriptions or queues filtered by priority, achieving a similar effect.

Finally, it is worth noting that priority mechanisms must be applied carefully to avoid issues like starvation of lower-priority tasks. Uncontrolled priority scheduling can lead to situations where low- priority work never gets done because new high- priority tasks keep arriving (we will discuss this challenge in a later section). Techniques such as aging (gradually increasing the priority of a task the longer it waits) are sometimes employed to ensure fairness.

3. Motivation: Challenges under Heavy Workloads without Priority Queues

To appreciate the importance of priority queues, we consider two illustrative scenarios drawn from industry experience where heavy workloads without appropriate prioritization caused significant problems:

Scenario 1: Monitoring and Control Queries in an Overloaded Database Cluster. In a distributed database under heavy transactional load, all compute resources may be occupied by user transactions (e.g., large analytical queries or heavy write operations). Monitoring queries – such as health checks, system status queries, or administrative commands – are typically lightweight and short-running. However, if they are queued along with regular workload, they can be delayed behind long-running user queries. In the worst case, an admin or monitoring query (which might be needed urgently to diagnose an ongoing incident) could be stuck waiting for minutes or hours, rendering observability and control nearly impossible when it’s needed most. This is essentially a form of priority inversion at the system level – the system’s insight into itself is blocked by the very workload it is running.

For example, suppose the database is maxed out on CPU and threads due to complex analytical queries. An operator trying to run a SHOW STATUS or a diagnostic query might find it enqueued and only serviced after those heavy queries finish. The lack of a “fast lane” for critical monitoring tasks is a serious risk. Modern systems address this by carving out reserved resources or priority queues for such tasks. As an analogy, Redshift’s manual WLM could be configured with a dedicated system queue for monitoring or small queries with higher priority. In general, database workload management or query schedulers should separate low-latency, short queries from long-running ones. By using priority scheduling (or segregated resource pools), the database ensures that, say, a simple monitoring query can execute promptly even when the system is under heavy load. Indeed, *“internal system queries”* or telemetry queries are often assigned high priority or run on separate threads to avoid being blocked by user workload. This scenario underlines the need for priority handling: without it, observability and control degrade exactly when they are most needed.

Scenario 2: Catalog Data Backfill vs. Real-Time Updates. Consider a large-scale e-commerce catalog or content index system that must process millions of items. Periodically, a backfill or full reprocessing job might be run (for example, re-indexing all products or recomputing recommendations). This backfill could involve processing an enormous volume of items (possibly millions) and could take many hours or days to complete. Meanwhile, there are *high-priority update events* — for instance, a critical price update for a product, or a new item that just got added and should be searchable quickly. If the system processes all items in one common pipeline without prioritization, those urgent updates will end up waiting behind the entire backlog of the backfill. In effect, an urgent item update may not be applied until the backfill finishes or the item’s turn comes in the FIFO queue, which might be days later. During that time, the catalog data for that item is stale or inconsistent (e.g., the price shown to users remains outdated). This is obviously unacceptable for business, as it can lead to lost revenue or compliance issues.

In such scenarios, implementing a priority queue (or multiple-tier processing) is vital. High-priority items (say, real-time updates or VIP customer data) should be placed into a faster lane for immediate processing, bypassing the bulk backlog. For example, one could maintain two queues: one for normal bulk processing and one for urgent updates. The system can always drain the urgent-update queue first so that critical changes propagate quickly. This pattern has been noted in industry; for instance, multi-queue designs are recommended when *“different tasks have distinct* *performance requirements that must be independently met."* In other words, if certain items must be processed within minutes while others can take hours, they should be separated and given dedicated processing capacity. Failing to do so can leave high- priority data in an out-of-sync state for extended periods, as was observed in the scenario above.

Both scenarios demonstrate a common theme: without a priority mechanism, critical but lightweight tasks get stuck behind heavy, bulk tasks. Under heavy workload, this can impair system usability, data freshness, and even stability. These challenges motivate the use of priority queues and smart scheduling as a fundamental design consideration in large-scale systems.

4. Priority Queue Design Patterns and Architectures

Having established the need, we now discuss how to incorporate priority handling into system architectures. The two basic design patterns were already introduced: (1) a single priority queue that reorders tasks by priority, and (2) multiple queues separated by priority level (with either a unified or separate consumer pools). There are variations and nuances to these patterns in practice, and the choice depends on the use-case and infrastructure capabilities:

4.1. Single Priority Queue, Single Consumer Pool:

In this design, the system has one work queue that supports prioritization internally. Producers assign a priority to each work item, and the queue (e.g., a priority heap or a message broker that supports priority) always delivers the highest-priority item next. All workers/consumers pull from this same queue. This design is simple for developers – you have one pipeline and the infrastructure ensures ordering by priority. For example, some message brokers (like RabbitMQ) support priority queues natively, and will order messages by a priority field. However, the downside is that if high-priority tasks keep arriving, lower priority tasks may never get served (starvation). Azure architects warn that with a single consumer pool servicing multiple priorities, *"lower priority messages [could] be continually delayed and potentially never processed."*. Thus, while this pattern ensures strict preference for urgent tasks, it can starve the background tasks unless the arrival rate of high priority messages is inherently limited or additional policies (like aging or max wait times) are implemented. Some systems address this by boosting the priority of tasks that have waited too long – effectively ensuring that no task waits indefinitely.

4.2. Multiple Priority Queues, Multiple Consumer Pools:

This pattern involves physically isolating the workloads. For each priority level, there is a separate queue and a dedicated set of consumers for that queue. For instance, one could have a *critical queue* with its own fleet of worker instances, a *normal queue* with a separate fleet, etc. Higher priority queues can be given a larger or faster set of consumers. The Azure Architecture Guide recommends this approach when different priority tasks have "strict performance requirements" or need fault isolation (so that issues processing low- priority tasks do not interfere with high- priority tasks). The benefit here is strong isolation: high-priority tasks are entirely in their own lane with guaranteed resources. This was historically a common approach in systems like print job schedulers or batch processing clusters (e.g., separate job classes). The drawback is cost and resource fragmentation – you must provision separate capacity for each priority level, and if the high-priority queue is often empty, its reserved resources might sit idle while low- priority queues backlog (resource inefficiency). Nonetheless, for critical workloads, this isolation is often worth the cost.

4.3. Multiple Priority Queues, Single Unified Consumer Pool:

A hybrid approach is to have separate queues for priorities, but a shared set of worker processes that pull tasks from all queues. The workers implement logic to always check the higher-priority queue first. This can be thought of as a software scheduling policy on top of multiple queues: e.g., a worker could poll the high- priority queue, and only if it's empty, then poll the low-priority queue. This ensures high-priority work is preferred, while still allowing a single scalable pool of workers (which can simplify autoscaling and reduce idle capacity issues). However, implementing this correctly can be complex – one must be careful about race conditions and fairness. Also, as the Azure guidance notes, a single pool always serving high priority first is effectively the same as strict priority scheduling, thus starvation of low priority is possible if high priority load is continuous. Some implementations allow configuring a maximum ratio of time or resources to high vs low priority tasks to ensure the lower priority queue makes progress (a form of weighted fair scheduling instead of strict priority).

An example of the unified pool approach in practice is a solution on AWS using Lambda: one can have a single Lambda function subscribed to both the high and low priority SQS queues, and in the function's code, always process high-priority messages first (by checking the high-priority queue for messages and only processing low-priority if none high remain). This was demonstrated by Van Donkersgoed (2022) using SQS and Lambda – effectively building a priority scheduler in the Lambda logic. Another example is the *round-robin listener* approach described by Mark Heath for Azure Service Bus, where a single process alternates between queues, prioritizing the high-priority queue on each cycle.

Yet another variant arises in systems like Kafka which do not support priority on a single topic. One solution seen in industry is to use *multiple topics* (one per priority level) and then have a custom consumer that merges them by priority. We will see an example of this from Klaviyo's event processing system later.

To summarize, **Table 1** compares these patterns:

- Single Priority Queue: Easiest logically; risk of low-priority starvation; requires broker support for priority or a custom priority data structure.

- Multi-Queue, Multi-Pool: Strong isolation; higher resource cost; suits strict SLO separation.
- Multi-Queue, Single-Pool: Balanced resource usage; requires custom scheduling logic in consumers; needs careful design to avoid starvation.

All these patterns aim to ensure that in overload conditions, the important work gets done first. The optimal choice depends on factors like the available messaging infrastructure, workload characteristics, and how much complexity one is willing to manage.

5. Case Studies and Examples

5.1. Facebook's Distributed Priority Queue (FOQS)

One prominent real-world system built around priority queuing is Facebook's FOQS (Facebook Ordered Queueing Service). FOQS is described as *"a fully managed, horizontally scalable, multi-tenant distributed priority queue built on sharded MySQL."* .It was designed to handle the enormous scale of background jobs at Facebook, which involve trillions of queue operations per day. The motivation for FOQS directly ties to our discussion: Facebook has many types of asynchronous tasks, and they realized that some tasks need to happen quickly (e.g. sending a notification to a user), whereas others can be deferred (e.g. generating translations of posts, or processing large analytics jobs). By introducing priority levels, they ensure that time-sensitive tasks (like notifications or real-time updates) are delivered and processed promptly, without being bottlenecked by heavy but less urgent workloads (like batch translations).

FOQS's architecture is quite advanced. Producers enqueue tasks with an assigned priority (numerical, where lower number = higher priority). The service is multi-tenant, meaning different teams at Meta use FOQS as a central queueing service and each gets isolated capacity and namespaces. Internally, FOQS stores queued items in MySQL tables, taking advantage of indexes to sort by priority and by deliver- after timestamps. Consumers then dequeue tasks in priority order. FOQS supports features like **delayed tasks**, retries, and **capacity quotas** to ensure one tenant's high load doesn't overwhelm others.

To give a simplified view, consider Figure 5.1, which illustrates FOQS conceptually (adapted from descriptions in ByteByteGo's analysis). Applications enqueue items with varying priority levels into FOQS; the FOQS system orders tasks such that consumers always receive the highest-priority ready tasks first. This is essentially the single priority queue model on a massive scale, with FOQS acting as a distributed priority broker. By deploying FOQS, Facebook achieved a decoupling of producers and consumers with guarantees on ordering by priority and time. This allows, for example, the "async workload" at Facebook to offload non-urgent computations so they don't clog the main interaction path. It also allows critical workflows (like video uploads spawning encoding jobs) to reliably fan-out tasks that will be handled with appropriate priority.

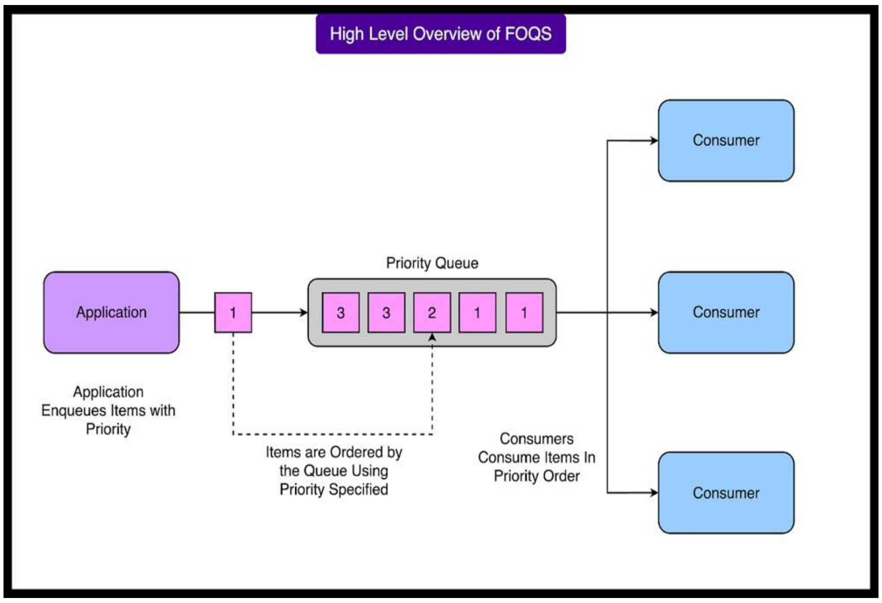
High Level Overview of FOQS
Application
1
Priority Queue
3
3
2
1
1
Consumer
Consumer
Consumer
Application Enqueues Items with Priority
Items are Ordered by the Queue Using Priority Specified
Consumers Consume Items In Priority Order

Fig 3: High-Level Overview of a Priority Queue Service Like FOQS (Facebook Ordered Queueing Service).

The producer application enqueues items with an associated priority value. The central queueing service stores tasks and ensures that items are dequeued in order of priority (here, tasks labeled "1" are highest priority, processed before lower priority numbers). Multiple consumers can pull from the queue, and they will receive tasks in global priority order. This design guarantees that critical tasks (e.g. priority 1) are handled before less critical tasks (e.g. priority 3), improving latency for the former even at very large scales. FOQS demonstrates that priority queuing can be implemented at extreme scale – leveraging database sharding to store and sort tasks – and underlines the benefits: **resilience and control over task processing**. Facebook's engineers have credited FOQS with enabling them to manage throughput and ordering without each team writing custom queue logic. It is essentially an "internal AWS SQS with priority" tailored to Facebook's needs.

5.2. Priority Scheduling in Event Processing (Klaviyo's Kafka-based System)

Another illuminating example comes from Klaviyo, which in 2025 described how they re-architected their event processing pipeline to handle events with different SLO (Service Level Objective) requirements. Their system processes billions of events per day (up to 170k events/sec) with some events requiring near- real-time processing (few seconds) and others being less urgent (many minutes permissible). Initially, they had separate infrastructure for each priority tier (e.g., separate Kafka consumer groups or separate processing clusters for "P10, P20, P30" levels). This corresponds to the multi-queue, multi-pool pattern – effective but resource-inefficient, as some clusters would be underutilized. As load grew, they found that approach unsustainable.

Klaviyo's solution was to unify the processing fleet while keeping priority segregation. Concretely, they implemented a unified priority-aware scheduler on top of Kafka. They maintained separate Kafka topics for each priority class (so producers still categorize events by priority, sending to different topics), but they introduced a consumer proxy layer that pulls from all priority topics and merges events into a single internal priority queue in memory. This in-memory queue is ordered by each event's deadline or priority (they describe using an earliest-deadline-first, EDF, scheduling approach). The processing workers then consume from this proxy via a pull-based interface, always getting the highest priority (earliest deadline) event available. Importantly, their scheduler allows preemption: a newly arrived high-priority event can effectively overtake lower priority ones in the queue, *"high-priority events can preempt lower-priority processing."*. At the same time, they built fairness controls to *"prevent starvation of background workloads."* – meaning if low-priority events start to lag too far behind, the system will ensure they eventually get CPU time (for example, by limiting how many high-priority events can preempt consecutively, or by monitoring queue age).

This design gave Klaviyo the best of both worlds: a single shared processing cluster (improving resource utilization by 30% and simplifying operations), and the ability to meet strict SLOs for high priority events (improving on-time performance by 20%). It's a modern example of implementing the priority queue pattern in a distributed streaming context. The proxy- based architecture also showcases a technique to introduce priority scheduling when the underlying messaging system (Kafka) does not natively support it. By using separate topics (which act as separate queues) and a custom merging layer, they in effect created a global priority queue. This is analogous to having multiple input queues and a single smart consumer pool as discussed earlier. Their approach also used pull-based consumption to avoid overloading consumers – processors pull the next event when ready, ensuring that slow processing of a low item doesn't block the proxy from handing off a high item to another free processor. This decoupling further reduces head-of-line blocking in the system.

Klaviyo's case study reinforces a key point: introducing priority scheduling can significantly improve tail latency and SLO adherence in high- throughput systems. By carefully designing the scheduler (using EDF and fairness in this case), they avoided starving lower priority tasks while still prioritizing the critical ones. It also highlights that priority queue concepts are being actively applied in industry (beyond the traditional message queue use- case) to solve modern large-scale problems.

5.3. AWS Priority Queue Patterns

While not a single product, it is worth summarizing how priority queue concepts manifest in common AWS architectures, since many large-scale systems use AWS components:

5.3.1. Amazon SQS:

As mentioned, SQS standard queues do not preserve any ordering (and specifically have *best-effort* ordering) and do not support priority fields. A well-known pattern to implement prioritization is using multiple SQS queues for different priorities. One can configure consumers to poll the high-priority queue preferentially. Alternatively, if using Lambda triggers, one can set a higher concurrency for the high- priority queue's Lambda consumer so that it can process many messages in parallel, whereas the low-priority queue is throttled to fewer concurrent executions. This ensures that under load, most resources go to high- priority tasks. The Priority Queue Pattern in AWS architecture literature explicitly suggests: "Use SQS to prepare multiple queues for individual priority levels. Place high-priority tasks in the high priority queue, and allocate more processing power to that queue" (Ref jayendrapatil.com). This pattern has been used in scenarios like video processing services where premium users' videos are processed on a high-priority queue and free users on a standard queue – ensuring premium customers get faster results.

5.3.2. Amazon SNS + SQS or EventBridge:

Another approach for AWS users is to use SNS topics or EventBridge with filtering to route messages to different SQS queues or Lambda functions by priority. This is analogous to Azure's topic subscription filtering. The producer sends all events to one SNS topic with a message attribute "Priority". Then one can set up two SQS subscriptions on that SNS topic, each with a filter policy (e.g. Priority = High vs Priority= Low). The rest of the architecture remains as above (separate or weighted consumers). This pattern offloads the routing logic to AWS's managed filtering rather than in application code.

5.3.3. Workload Management in Databases:

For AWS analytics databases like Redshift (and third-party ones like Snowflake), a form of priority queue is implemented in their scheduling of queries. Redshift's manual WLM allows defining up to 8 queues with a mix of memory and concurrency limits, effectively letting certain queues host only small queries with higher priority (Ref dishanka.medium.com). More recently, Redshift's automatic WLM will dynamically route short queries to a special "concurrency scaling" cluster so that they aren't impeded by long queries on the main cluster. This can be viewed as creating an implicit fast lane for short queries by provisioning extra capacity. The principle is the same: *don't let the big jobs monopolize everything.* Ensure there's always capacity for small, urgent tasks.

5.3.4. AWS Step Functions / Batch:

In AWS Step Functions (or AWS Batch), one can also manage tasks with priority. AWS Batch jobs have a priority parameter which the scheduler uses to decide which job to run next when there are resource constraints. Higher priority jobs are dispatched first (though Batch also tries to ensure lower priority jobs eventually run, to avoid starvation). This is an example of a managed service implementing priority queuing internally.

In summary, AWS provides building blocks but it often falls to the system designer to implement the priority logic using those blocks. The prevalent solution is multiple queues or topics for priorities, which maps well to the patterns discussed. Importantly, AWS's own guidance and customer experiences have repeatedly highlighted that such patterns are necessary for high-performance systems. If one were to treat all messages or tasks equally in a cloud workload, it is very easy for a heavy batch of work to degrade the responsiveness for critical tasks (as our scenarios showed). Thus, **the priority queue pattern is a recommended best practice in system design** for achieving differentiated service levels in the cloud.

6. Challenges and Considerations

While priority queues and priority scheduling bring clear benefits, they also introduce challenges that must be managed:

6.1. Starvation of Low-Priority Tasks:

The most obvious risk is that a continuous stream of high- priority tasks can starve the lower priority ones, which might then never get service. This can lead to unbounded wait times for some tasks, which may be unacceptable. To address this, systems often implement aging or priority boost mechanisms: if a low-priority task has waited too long, its priority is gradually increased. For example, Azure Service Bus does not have this built-in, but the Azure Architecture Center recommends that *"in queues that support message prioritization, dynamically increase the priority of aged messages to ensure low-priority messages eventually get processed."* (*Ref* learn.microsoft.com). Another approach is to ensure that high-priority traffic is capped or bounded. For instance, one might decide that at most 90% of processing capacity will go to high priority, leaving 10% always for low priority. This is effectively a weighted fair scheduling rather than strict priority. Some cluster schedulers (and even CPU schedulers) use such weighting to ensure progress for all classes. In networks, the analogous concept is providing a minimum bandwidth guarantee to each traffic class to avoid complete starvation. Designers must carefully consider the expected load patterns; if high priority load could ever be unending, then starvation prevention must be in place.

6.2. Priority Inversion and Resource Contention:

Priority inversion is a classic problem where a high-priority task is waiting indirectly on a low-priority one due to resource locking. In distributed systems, this might occur if a high-priority job needs access to a resource (say a database row or a file) currently locked by a low-priority job. Simply using a priority queue won't solve that; one needs additional mechanisms (like priority inheritance in locking, or timeouts to force release). While this is more of a concurrency control issue, it is relevant – introducing priorities doesn't magically eliminate dependencies. Operationally, one should monitor if high-priority tasks are being delayed by any such issues (for example, if a high-priority queue consumer is consistently waiting on something held by low- priority processes, that's a problem).

Additionally, if high and low priority tasks share infrastructure, a heavy low-priority task could consume some critical shared resource (e.g. memory, I/O bandwidth) and impact the high-priority task. This is why true isolation (as in separate consumer pools or separate hardware) is sometimes necessary for the most critical tasks. Engineers must ensure that any common resources (threads, database connections, etc.) are not bottlenecked by lower priority work when needed by high priority work – often achieved via token buckets or simply by segregating those resources.

6.3. Complexity of Implementation:

Priority scheduling can add complexity to the system. If the underlying messaging middleware does not support priority, developers must implement the logic themselves (as we saw with the Lambda or Kafka examples). This increases the chances of bugs. Care must be taken to handle edge cases like bursts of high- priority messages, switching costs between queues, and maintaining fairness. Testing priority systems under load is important to verify that the expected ordering is happening and performance is stable. Observability is also key: systems like Klaviyo's included metrics like *"queue depth by priority lane"* and *"age skew"* to monitor if any priority band is lagging too much.

6.4. Throughput vs. Latency Trade-offs:

Introducing priority can sometimes slightly reduce overall throughput or increase complexity in load balancing. For instance, if workers are constantly context- switching to higher priority tasks, there could be overhead that wouldn't exist if they just plowed through tasks in arrival order. In practice, this is usually negligible compared to the latency gains for high-priority tasks, but it's something to consider. Also, using multiple queues can complicate load balancing – e.g., one queue might become very large while another is nearly empty, which is easier to handle in a single queue model. Techniques like dynamic scaling of consumer pools per queue or merging queues when idle can help mitigate this.

6.5. Correct Priority Assignment:

The system is only as good as the priority definitions. If too many tasks are labeled high priority, then effectively nothing is low priority and the benefit is lost (all tasks will contend in the high lane). Conversely, if a critical task is mistakenly not marked high priority, it could get stuck. Thus, clear definitions and governance of what constitutes a high priority task are important. In an academic sense, this relates to scheduling theory: you want to design your scheduling classes such that they align with business importance and performance goals.

6.6. Cost and Resource Over-Provisioning:

Especially for the multi-queue multi-pool pattern, maintaining separate capacity for high priority tasks that might be sporadic can mean paying for idle resources. There is a cost trade-off in ensuring low latency for critical tasks. Solutions like serverless computing or elastic scaling can alleviate this by not having dedicated always-on workers for the high-priority queue, but one must ensure the spin-up time is fast enough to still meet latency requirements. For example, AWS Lambda cold start might be a factor if a high-priority queue suddenly receives a burst – one might keep a minimal warm pool ready.

6.7. Verification and Testing:

From an academic perspective, verifying that a priority scheduling system meets all deadlines or latency targets can be non-trivial, especially in probabilistic heavy-load scenarios. Queueing theory or simulations are often used to ensure that the system design (with certain arrival rates and service rates per priority) will behave as expected. In critical systems (e.g., real-time systems), one might derive analytical bounds for worst-case latency given the scheduling algorithm. In less formal terms, teams should test their systems under synthetic load where, for instance, low priority load is high and see if the high priority tasks still meet their SLAs (Service Level Agreements).

Despite these challenges, the consensus in industry and research is that the benefits of priority-aware scheduling far outweigh the downsides for systems that have heterogeneous task requirements. Techniques exist to mitigate each of the issues above, and many are well-studied (for instance, starvation avoidance has been studied in operating systems for decades).

7. Conclusion

Under heavy workloads, distributed systems must be intelligent about how they schedule work. The use of **priority queues** and priority-based scheduling is a proven approach to ensure that critical tasks maintain low latency and important data remains fresh, even when the system is processing massive amounts of background work. We saw that without such prioritization, systems can suffer from *monitoring blind spots* (when admin tasks get stuck behind user tasks) and *stale or delayed data* (when urgent updates languish in backlogs). By introducing priority lanes – whether via a single prioritized queue or multiple queues with dedicated resources – systems can provide differentiated service levels, honoring the needs of high-priority operations.

We discussed architectural patterns for implementing priority queues, from the simple single-queue approach to more complex multi-queue designs. Real- world case studies, like Facebook's FOQS service and Klaviyo's Kafka-based scheduler, demonstrate that priority-based designs are not just theoretical, but in fact crucial in large-scale production environments handling billions of events. These systems achieved better performance and reliability by ensuring that urgent work preempts or bypasses less urgent work when necessary, all while striving to keep lower- priority work from starving completely.

Priority scheduling does introduce considerations such as fairness, complexity, and resource allocation trade- offs. However, with careful design – including techniques like aging, separate resource pools, and fine-grained monitoring – these challenges are manageable. In practice, many cloud and data platforms (AWS, Azure, etc.) provide tools to implement these patterns, but the responsibility is on system designers to apply them correctly to meet their specific workload requirements.

In an academic sense, the priority queue approach in distributed systems ties into broader topics of scheduling theory, real-time systems, and QoS management. As systems continue to scale and as workloads become more varied (with mixes of real- time streams and offline batches sharing infrastructure), the importance of efficient priority scheduling will only grow. Future research and development may focus on more autonomous prioritization (using AI to dynamically adjust priorities based on observed behavior or SLAs) and on robust theoretical guarantees for complex distributed schedulers.

In summary, priority queues are a fundamental tool in the arsenal of large-scale system design. They bring order (literally) to chaos under heavy load, ensuring that when resources are scarce, they are applied where they matter most. By learning from the patterns and examples discussed, practitioners can avoid the pitfalls of unmanaged queues and build systems that remain responsive and correct, even at massive scale.

References

[1] Amazon Web Services, *"Amazon Redshift Workload Management,"* 2024. [Online]. Available: https://docs.aws.amazon.com/redshift/latest/dg/c_workload_mngmt_classification.html#:~:text=Amazon%20Redshift%20workload%20managemen t%20,well%20as%20a%20runtime%20queue

[2] Microsoft Azure, *"Priority Queue Pattern,"* Azure Architecture Center, 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/patterns/priority-queue

[3] ByteByteGo, *"How Facebook's Distributed Priority Queue Handles Trillions of Items,"* Oct. 2025. [Online]. Available: https://blog.bytebytego.com/p/how-facebooks-distributed-priority

[4] J. Patil, *"AWS SQS Design Patterns – Priority Queue,"* 2023. [Online]. Available: https://jayendrapatil.com/tag/sqs-design patterns/#:~:text=Priority%20Queue%20Pattern

[5] Rohit Pathak, *"Building a Distributed Priority Queue on Kafka,"* Sep. 2025. [Online]. Available: https://klaviyo.tech/building-a-distributed-priority-queue-in-kafka-1b2d8063649e

[6] L. van Donkersgoed, *"Implement the Priority Queue Pattern with SQS and Lambda,"* 2022. [Online]. Available: https://lucvandonkersgoed.com/2022/04/25/impl ement-the-priority-queue-pattern-with-sqs-and-lambda/#:~:text=In%20the%20previous%20secti on%2C%20we,github%20source

[7] W. Velida, *"The Priority Queue Pattern,"* DEV Community, 2024. [Online]. Available: ttps://dev.to/willvelida/the-priority-queue- pattern-23g8#:~:text=The%20Priority%20Queue%20patte rn%20is,faster%20than%20lower%20priority%20r equests

[8] Meta (Facebook) Engineering, "FOQS Use Cases and Distributed Priority Queue Behavior," 2025. [Online]. Available: https://blog.bytebytego.com/p/how-facebooks-distributed-priority and https://engineering.fb.com/2021/02/22/productio n-engineering/foqs-scaling-a-distributed-priority- queue/

[9] Amazon Web Services, "Create and Prioritize Query Queues in Amazon Redshift," AWS re:Post, 2024. [Online]. Available: https://docs.aws.amazon.com/redshift/latest/dg/ c_workload_mngmt_classification.html#:~:text= Workload%20management%20,don%27t%20get %20stuck%20in%20queues

[10] Ratan Bajpai, Krishna Dhara, B. Chandramouli *et al.*, *"QPID: A Distributed Priority Queue with Item Locality,"* 2009. [Online]. Available: https://www.researchgate.net/publication/220946085 _QPID_A_Distributed_Priority_Queue_with_Item_L ocality

[11] H. Rihani, P. Sanders, and R. Dementiev, *"MultiQueues: Simpler, Faster, and Better Relaxed Concurrent Priority Queues,"* arXiv:1411.1209, 2014. https://arxiv.org/abs/1411.1209

[12] L. Kleinrock, *Queueing Systems, Volume 1: Theory*, Wiley, 1975.

[13] A. Federgruen and H. Groenevelt, *"M/G/c Queueing Systems with Multiple Customer Classes,"* Columbia Business School Working Paper, 1988.

[14] D. Bertsimas and J. Niño-Mora, *"Multiclass Queueing Systems in Heavy Traffic: An Asymptotic Approach,"* MIT Working Paper, 1997.

[15] J. F. C. Kingman, *"Queue Disciplines in Heavy Traffic," Journal of the Royal Statistical Society B*, 1982.

[16] S. Lee *et al.*, *"Analysis of a Priority Queueing System with Enhanced Priority," Journal of Ambient Intelligence and Humanized Computing*, 2024.

[17] D. Alistarh *et al.*, *"The SprayList: A Scalable Relaxed Priority Queue,"* PPoPP 2015 / Microsoft Research Tech Report, 2015.

[18] H. Rihani, P. Sanders, and R. Dementiev, *"MultiQueues: Simpler, Faster, and Better Relaxed Concurrent Priority Queues,"* arXiv:1411.1209, 2014.

[19] M. Williams *et al.*, *"Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues,"* arXiv:2107.01350, 2021.

[20] S. Walzer and M. Williams, *"A Simple yet Exact Analysis of the MultiQueue,"* arXiv:2410.08714, 2024.

[21] M. Feldmann, M. Henzinger, *et al.*, *"Skeap & Seap: Scalable Distributed Priority Queues,"* SPAA 2019.

[22] T. Zhou *et al.*, *"A Practical, Scalable, Relaxed Priority Queue (ZMSQ),"* ICPP, 2019.

[23] B. S. Lin and X. Liang, *"A Scalable Relaxed Distributed Priority Queue,"* Stanford CS244B Project Report, 2020.