Queuing systems

Verified 2026-08-03

Acknowledgement, delivery guarantees, retries, ordering, and choosing between BullMQ, RabbitMQ, Kafka, and SQS.

Contents

Mental model

A queue turns a synchronous call into a durable handoff: the request records the intent and returns, and a separate worker does the slow part later. That buys a fast response and **load leveling** — a traffic spike lands in the queue instead of on the server. Everything hard about queues follows from one fact: a broker cannot tell a slow consumer from a dead one, so it must choose between delivering a message twice and losing it. It chooses twice, and the rest is your problem.

Receive, acknowledge, delete

Event	What the broker does
Producer sends	Stores the message durably
Consumer receives	Hides it from others — “in flight”
Consumer acknowledges	Deletes it

Event	What the broker does
Consumer dies or times out	Makes it visible again

Reading a message does not remove it; it hides it. SQS calls that window the **visibility timeout** — 30 seconds by default, 12 hours maximum. AMQP calls the reply an acknowledgement: if a consumer dies without sending one, RabbitMQ redelivers to another consumer, or waits until one registers. A job slower than the window is a job that runs twice.

```
# hide it for 5 minutes, then delete on success
aws sqs receive-message --queue-url "$Q" \
  --visibility-timeout 300

aws sqs delete-message --queue-url "$Q" \
  --receipt-handle "$H"
```

> **Gotcha:** Acknowledging after the visibility timeout expires is too late — the message went back on the queue and another worker is already running it. Size the window against your slowest job, not the average one.

Delivery guarantees

Guarantee	You get	Use for
At most once	Loss, never duplicates	Metrics, telemetry
At least once	Duplicates, never loss	Almost everything
Exactly once	Neither, at a price	Ledger-style writes

Exactly-once *delivery* over a network does not exist; what tools sell is exactly-once *processing* — at-least-once delivery plus deduplication. The docs are candid about it: BullMQ “attempts to deliver every message exactly

one time, but it will deliver at least once in the worst case scenario", and SQS standard queues warn that "more than one copy of a message might be delivered". Correctness lives in the consumer, not the broker.

```
-- idempotent consumer: a replay hits the
constraint
INSERT INTO processed (job_id) VALUES ($1)
ON CONFLICT (job_id) DO NOTHING;
```

> **Gotcha:** Duplicates are normal operation, not an incident. A handler that charges a card or sends mail needs a deduplication key with a unique constraint — "retries are rare" is not a design.

Retries, backoff, dead letters

Setting	Controls
`attempts` / `maxReceiveCount`	Tries before giving up

Setting	Controls
`backoff`	How fast the delay grows
`jitter`	Spread across a failing batch
Dead-letter queue	Where exhausted messages land

A job that fails and retries immediately, forever, is how one bad message saturates a worker pool — a **poison message**. Cap the attempts, grow the delay (BullMQ's `exponential` waits `2 ^ (attempts - 1) * delay` ms), add jitter so a batch that failed together doesn't retry in lockstep, and send whatever exhausts its attempts to a dead-letter queue.

```
import { Queue } from 'bullmq';

const queue = new Queue('billing');

queue.add(
```

```
  'send-invoice',
  { userId: 42 },
  {
    attempts: 3,
    backoff: { type: 'exponential', delay: 1000 },
  },
);
```

> **Gotcha:** A dead-letter queue nobody watches is a silent data-loss channel. Alert on its depth — worker error rates go quiet precisely when messages stop being retried.

Ordering

System	Ordering unit	Parallelism
Kafka	Partition	One consumer per partition
SQS FIFO	Message group	One in flight per group

System	Ordering unit	Parallelism
SQS standard	None; best effort	Unbounded

Global ordering and parallel consumption are mutually exclusive, so every system shards order into independent lanes. Kafka guarantees “any consumer of a given topic-partition will always read that partition’s events in exactly the same order as they were written” — and promises nothing across partitions. SQS FIFO does the same per `MessageGroupId`, which is required on every FIFO send. Choose the key so things that must be ordered share it, and nothing else does.

```
user:42  -> partition 3   ordered
user:99  -> partition 1   ordered
between partitions:       no order
```

> **Gotcha:** One ordering key for everything makes the queue serial. A single group or partition is consumed by exactly one worker, so throughput stops improving no matter how many you run.

Queues versus logs

Aspect	Queue	Log (Kafka)
Reading	Removes the message	Advances an offset
Replay	Gone once acknowledged	Any offset, any time
Retention	Until consumed	Time or size, regardless

A queue holds *work to be done* — a consumer takes a message and it is gone. A log holds *a record of what happened*: in Kafka "events are not deleted after consumption", each consumer group keeps its own offset, and a partition is consumed by exactly one

consumer within each group. Two teams read the same stream without coordinating, and a fixed bug replays from yesterday.

```
offset:  0    1    2    3    4
group A             ^ (reading)
group B  ^ (replaying from 0)
```

> **Gotcha:** Retention is a deadline. Past it, events are discarded whether or not anyone consumed them — "we can always replay" holds only inside the window you configured.

Choosing a tool

Tool	Model	Reach for it when
BullMQ	Redis-backed job queue	Node app, you run Redis
RabbitMQ	AMQP broker with routing	Services need routing rules

Tool	Model	Reach for it when
Kafka	Partitioned, replayable log	Volume, replay, many readers
SQS	Managed queue	On AWS, want no ops

Match the model, not the popularity. BullMQ is “a Node.js library that implements a fast and robust queue system built on top of Redis”, with delayed jobs, cron repeats, priorities, and parent/child flows. RabbitMQ routes: publishers send to an exchange, which uses bindings to select queues — `direct` by exact routing key, `topic` by pattern, `fanout` to everything bound. SQS is the one with nothing to operate: 1 MiB messages, 4-day default retention, 14 maximum.

```
publisher -> exchange -> binding -> queue
             (topic)     order.*    orders
```

> **Gotcha:** Kafka is not a drop-in job queue. There is no per-message acknowledgement and no per-job retry, and one slow message holds up its whole partition. Use it for streams you replay, not for sending email.

Further reading

- Amazon SQS Developer Guide
- RabbitMQ — AMQP 0-9-1 concepts
- Apache Kafka — Introduction
- BullMQ documentation