AWS SQS and SNS

Verified 2026-11-02

SQS queues and SNS topics — standard vs FIFO, the receive loop, DLQs, Lambda partial batch failure, fan-out, filter policies and limits.

Contents

Mental model 2

Standard vs FIFO queues 2

The receive loop and visibility timeout 3

FIFO ordering and deduplication 5

Dead-letter queues and redrive 7

Lambda event source mapping 9

SNS topics and fan-out 10

Filter policies and raw delivery 12

Size, encryption and pricing 13

Further reading 14

Mental model

SNS pushes each message to every subscriber right now and keeps nothing; SQS stores each message until one consumer takes it and deletes it. A queue subscribed to a topic combines both: SNS fans out, and each queue holds its own copy until its consumer is ready. Both deliver at least once, so every consumer must be idempotent.

Standard vs FIFO queues

Property	Standard	FIFO
Ordering	Best-effort	Strict per message group
Delivery	At least once	Deduplicated within 5 minutes
Throughput	Nearly unlimited	300 calls/s per partition
Name	Any	Must end in `.fifo`

Pick standard unless the consumer genuinely needs order or deduplication: it scales without tuning and costs less. FIFO's limit is per API action; batching 10 messages per call raises it to 3,000 messages per second, and high-throughput mode raises it further.

```
import {
  SQSClient, SendMessageCommand,
} from "@aws-sdk/client-sqs";

const sqs = new SQSClient({});
await sqs.send(new SendMessageCommand({
  QueueUrl: process.env.QUEUE_URL,
  MessageBody: JSON.stringify({ orderId: "o-9" }),
}));
```

The receive loop and visibility timeout

Setting	Default	Range
`VisibilityTimeout`	30 s	0 s to 12 hours
`WaitTimeSeconds`	0 (short poll)	Up to 20 s

Setting	Default	Range
`MaxNumberOfMessages`	1	1–10
`MessageRetentionPeriod`	4 days	1 minute to 14 days

Receiving a message hides it for the visibility timeout; it is not removed. You delete it with its `ReceiptHandle` after processing, or it reappears for another consumer. Long polling (`WaitTimeSeconds: 20`) waits for messages instead of returning empty at once, which cuts empty receives and cost.

```
import { ReceiveMessageCommand,
  DeleteMessageCommand } from "@aws-sdk/client-
sqs";

const { Messages = [] } = await sqs.send(
  new ReceiveMessageCommand({
    QueueUrl: url,
    MaxNumberOfMessages: 10,
```

```
    WaitTimeSeconds: 20,
  }));
for (const m of Messages) {
  await handle(JSON.parse(m.Body));
  await sqs.send(new DeleteMessageCommand({
    QueueUrl: url, ReceiptHandle: m.ReceiptHandle,
  }));
}
```

> **Gotcha:** If processing takes longer than the visibility timeout, the message becomes visible again and a second consumer processes it in parallel. Set the timeout above your worst-case processing time, or extend it with `ChangeMessageVisibility`.

FIFO ordering and deduplication

Attribute	Purpose
`MessageGroupId`	Required; order is kept within a group

Attribute	Purpose
`MessageDeduplicationId`	Same ID within 5 minutes is dropped
`ContentBasedDeduplication`	Uses a SHA-256 of the body as the ID
`DeduplicationScope`	`messageGroup` for high throughput

Messages in one group are delivered in order, one batch at a time, so a group is also a unit of serialization. Use many group IDs, such as one per customer, to get parallelism and spread load across partitions.

```
await sqs.send(new SendMessageCommand({
  QueueUrl: fifoUrl,
  MessageBody: JSON.stringify({ step: 2 }),
  MessageGroupId: "customer-42",
  MessageDeduplicationId: "order-9-step-2",
}));
```

> **Gotcha:** One shared `MessageGroupId` serializes the whole queue: a single slow or failing message blocks every message behind it.

Dead-letter queues and redrive

Setting	Meaning
`maxReceiveCount`	Receives before a message moves to the DLQ
`deadLetterTargetArn`	The DLQ, same type as the source
`RedriveAllowPolicy`	Which sources may use this DLQ
`StartMessageMoveTask`	Moves messages back after a fix

The redrive policy lives on the source queue, as a JSON string inside the attributes map. A message that fails `maxReceiveCount` times is parked in the DLQ instead of

looping forever. After fixing the bug, redrive moves messages back to the source queue.

```
POLICY=$(jq -nc --arg arn "$DLQ_ARN" \
  '{deadLetterTargetArn: $arn,
    maxReceiveCount: "5"}')
aws sqs set-queue-attributes --queue-url "$URL" \
  --attributes "$(jq -nc --arg p "$POLICY" \
    '{RedrivePolicy: $p}')"
aws sqs start-message-move-task \
  --source-arn "$DLQ_ARN"
```

> **Warning:** The DLQ's own retention clock keeps the original enqueue time for standard queues. Give the DLQ a longer retention than the source, or messages can expire soon after arriving.

Lambda event source mapping

Setting	Value
`BatchSize`	Default 10; max 10,000 standard, 10 FIFO
Batching window	Required ≥ 1 s when batch > 10
Queue visibility timeout	6 × function timeout + window
`ReportBatchItemFailures`	Retry only failed messages

Lambda polls the queue, invokes your function with a batch, and deletes the batch on success. Without partial batch response, one failed message makes the whole batch visible again, so the successful ones are processed twice.

```
aws lambda create-event-source-mapping \
  --function-name worker \
  --event-source-arn "$QUEUE_ARN" \
```

```
  --batch-size 10 \
  --function-response-types
ReportBatchItemFailures
```

> **Gotcha:** With `ReportBatchItemFailures` on, the function must return `{ batchItemFailures: [...] }`. Throwing an error still fails the entire batch.

SNS topics and fan-out

Subscriber type	Notes
SQS queue	Durable fan-out; needs a queue policy
Lambda	Invoked asynchronously
HTTP/S	Must confirm the subscription
Email, SMS, mobile push	For people, not systems

Publish once, and SNS delivers a copy to every subscription. Subscribing queues rather than functions directly gives each consumer its own buffer, retries and DLQ. The queue must allow the topic to send to it, scoped with `aws:SourceArn`.

```
{
  "Effect": "Allow",
  "Principal": { "Service": "sns.amazonaws.com" },
  "Action": "sqs:SendMessage",
  "Resource": "arn:aws:sqs:eu-
west-1:1234:billing",
  "Condition": { "ArnEquals": {
    "aws:SourceArn":
      "arn:aws:sns:eu-west-1:1234:orders"
  } }
}
```

Filter policies and raw delivery

Attribute	Effect
`FilterPolicy`	Only matching messages reach this subscriber
`FilterPolicyScope`	`MessageAttributes` (default) or `MessageBody`
`RawMessageDelivery`	Drop the SNS JSON envelope

A filter policy moves routing into SNS, so a subscriber stops receiving and discarding messages it does not want. The policy matches message attributes by default; set the scope to `MessageBody` to match fields in a JSON payload.

```
{
  "eventType": ["OrderPlaced"],
  "total": [{ "numeric": [">=", 100] }],
  "region": [{ "anything-but": ["test"] }]
}
```

> **Gotcha:** Without raw delivery, the SQS body is an SNS envelope and your payload is the escaped string in its `Message` field. A consumer that parses `Body` directly gets the envelope, not your object.

Size, encryption and pricing

Item	Value
SQS message size	Up to 1 MiB
SNS message size	256 KiB default, 1 MiB by topic setting
SQS encryption	SSE-SQS on by default for new queues
Billing unit	Each 64 KB chunk is one request
Free tier	1M SQS and 1M SNS requests per month

Raise an SNS topic's `MaximumMessageSize` to send up to 1 MiB; such a topic is limited to 100 subscriptions on SQS,

Lambda or Firehose. For anything larger, store the payload in S3 and send its key.

```
SQS 1 MiB message   -> 16 chunks -> 16 requests
SNS to KMS queue    -> customer managed key,
                       key policy allows SNS
```

> **Gotcha:** SNS cannot deliver to a queue encrypted with the AWS managed SQS key. Use a customer managed KMS key whose policy grants `sns.amazonaws.com` `kms:GenerateDataKey*` and `kms:Decrypt`.

Further reading

- SQS Developer Guide
- SNS Developer Guide
- Using Lambda with SQS
- SNS message filtering