

THE LITTLE BOOK OF

QueenMQ

*Partitions are for ordering.
Not for scaling.*



Contents

About this book	v
1 Introduction	1
1.1 What is QueenMQ?	1
1.2 Why QueenMQ	1
1.3 The model	3
1.3.1 Partitions	3
1.3.2 Leases, not static assignment	4
1.3.3 Transactions	4
1.3.4 A simple interface	5
1.3.5 Wrap up	6
2 Writing applications with QueenMQ	7
2.1 Setup a local broker	7
2.2 Create and configure queues	8
2.2.1 Every option	9
2.2.2 Three traps	11
2.3 Push messages	12
2.3.1 Every message gets its own answer	12
2.3.2 Buffering, for many small messages	13
2.3.3 When several messages must land together	14
2.3.4 Binary push, compression and framed pop	14
2.4 Pop messages	16
2.4.1 Queue mode and consumer groups	16
2.4.2 One message at a time, or a whole batch	17
2.4.3 How many messages a pop takes: autopilot	19
2.4.4 Leases and long work	19
2.4.5 Where a new group starts	19
2.4.6 Only the newest message: conflation	20
2.4.7 Faster, with no ack: commitOnDelivery	20
2.4.8 Running a consumer for real	21
2.4.9 Read it again	22
2.5 Acknowledge messages	22
2.5.1 Four answers	22
2.5.2 One message or a whole batch	23

2.5.3	Three things to know	24
2.6	Retries and the dead-letter queue	25
2.6.1	What a dead letter keeps	26
2.6.2	Read it	26
2.6.3	Replay it	26
2.6.4	Skip it, or turn it off	26
2.7	Advanced transactional usage	27
2.8	KV, Locks and Timers	28
2.9	Ephemeral queues	30
2.10	Stream processing	31
2.10.1	Four kinds of window	32
2.10.2	When it happened, or when it arrived	33
2.10.3	Shaping the data	34
2.10.4	Two ways to end a stream	34
2.10.5	Rate limits with a gate	34
2.10.6	Running a stream	35
3	Real applications examples	36
3.1	Work queue	36
3.2	Publish/subscribe and fan-out	37
3.3	Request and reply	39
3.4	Ordering per key, and batching	40
3.5	Workflows: one step, one commit	41
3.6	Windowed aggregation	43
3.7	Rate limiting	44
3.8	Last value	45
3.9	Delayed and scheduled messages	46
3.10	Retries and dead letters	47
3.11	One worker at a time	49
4	Integrations	51
4.1	How to use with Kafka clients	51
4.1.1	Turn it on	51
4.1.2	One consumer group for both	52
4.1.3	How Kafka maps onto Queen	53
4.1.4	Five things to know	54
4.2	S3 Sink	55
4.2.1	Turn it on	55

4.2.2	How it works	56
4.2.3	Four things to know	58
4.3	Postgres Source and Sink	58
4.3.1	The source	58
4.3.2	The sink	61
4.3.3	Four things to know	61
4.4	The Laravel Worker Supervisor	62
4.4.1	Two engines, one configuration	63
4.4.2	How to use it	63
4.4.3	What it does with a burst	65
4.4.4	Control it	66
4.4.5	Four things to know	66
5	Manage clusters	67
5.1	Run a cluster with Docker	68
5.1.1	Try it on one machine	68
5.1.2	Stop the leader	70
5.1.3	Three machines	71
5.1.4	Five things to know	72
5.2	Run a cluster inside Kubernetes	73
5.2.1	The secret	73
5.2.2	The manifest	74
5.2.3	The operator	75
5.2.4	Four things to know	76
5.3	Configuration for different use cases	76
5.3.1	To try Queen, or to develop	77
5.3.2	A small production	77
5.3.3	To survive the loss of a machine	77
5.3.4	To keep strangers out	78
5.3.5	Many tenants on one cluster	79
5.3.6	Payloads that must not sit on disk in clear	80
5.3.7	Disk that does not fill up	81
5.3.8	Many short-lived partitions	81
5.3.9	Kafka clients	82
5.3.10	Thousands of queues	82
5.3.11	To survive the loss of a whole site	83
5.4	Upgrading and monitoring	83
5.4.1	Upgrade one node at a time	83

5.4.2	The way back	84
5.4.3	What to watch	85
5.5	Recovery	88
5.5.1	Start with a write	88
5.5.2	Rules for a bad day	89
5.5.3	Find your case	89
5.5.4	Replace a node	90
5.5.5	A majority is down, the disks are fine	91
5.5.6	A disk is full	91
5.5.7	A majority of the disks is gone for good	91
5.5.8	Every node stopped on the same entry	92
5.5.9	Backups	92
6	Architecture	94
6.1	Leader and followers	94
6.2	Admission, Batch, Plan, Apply	95
6.3	Raft internal protocol	96
6.4	Memory state	97
6.5	Consumption engine	97
6.6	LMDB	99
6.7	Storage	99
6.7.1	The data directory	99
6.7.2	What is inside a log file	100
6.7.3	How retention works	101
6.7.4	Rewriting a file	102
	Conclusion	103

About this book

This book is written by the authors of QueenMQ. In it, we have tried to explain in simple terms what QueenMQ does, and why. Here is what each chapter covers.

- 1. Introduction.** What QueenMQ is, why we built it (the story that went from a Postgres table to a log on disk replicated with Raft), and the model behind it, in four ideas: partitions, leases, transactions and a simple interface.
- 2. Writing applications with QueenMQ.** Every feature of the client API, one at a time, in JavaScript: a local broker, queues and their options, push, pop and consume, acks, retries and the dead-letter queue, transactions, KV, locks and timers, ephemeral queues and stream processing. Each feature comes with the reason it exists.
- 3. Real applications examples.** The common messaging patterns, one page each: work queue, publish/subscribe, request and reply, ordering and batching, workflows, windowed aggregation, rate limiting, last value, delayed messages, retries and dead letters, and one worker at a time. Each one solves a real problem in a few lines.
- 4. Integrations.** How QueenMQ works with the systems around it: Kafka clients through its built-in Kafka port, an S3 bucket kept up to date by the sink, PostgreSQL tables read and written by the source and the sink, and Laravel applications, whose workers the QueenMQ supervisor runs.
- 5. Manage clusters.** The settings that matter most, a cluster with Docker and inside Kubernetes, the configuration for different needs (tenants included), upgrades and monitoring, and what to do when something breaks.
- 6. Architecture.** How the broker works inside: one leader per tenant, the path of a write from the request to the disk, Raft, the memory state, the consumption engine, LMDB, and the files on disk down to their bytes.

Every piece of code in this book comes from a program that runs it against QueenMQ 2.6.0 and checks the result. The examples are in JavaScript; the website has them in every language of the SDKs.

1

Introduction

1.1 What is QueenMQ?

QueenMQ is a distributed message broker, built in Rust, replicated with Raft, and developed at Smartness. With it you can write applications that decouple producing from consuming, do request and reply, and process streams, using transactional features that let you remove a lot of logic from the application and offload it to QueenMQ.

Every feature the broker has is there because we needed it at Smartness, where it powers almost every product.

1.2 Why QueenMQ

The world is full of message brokers, and some of them are really powerful, with tons of useful features. So why build another one?

Back in 2025, while we were developing a product called Smartchat, we suffered heavily from head-of-line blocking in Kafka. The product was made of several services that talked through Kafka, and some of them processed messages with LLMs. Processing a message with an LLM can be slow, and how slow depends on the size and type of the message (plain text, XML...), so in production we saw messages wait seconds, sometimes minutes, behind heavier ones. This was expected when we chose Kafka for Smartchat, but there was no easy alternative (also because we used Kafka in every other product), so we stuck with it.

To solve the issue, we started with a hack. For the heavy topics, we consumed the messages and wrote them into a Postgres table with a `chat_id` column (one conversation between A and B, for example). Then a custom poller read that table, processed the messages of each chat in order, and pushed them again to another Kafka topic.

It was easy, and it worked very well. So in our minds a new model took shape, one that SQL represents very well, with these entities:

- the queue table
- the partition table
- the partition consumer table
- the message table

Like this:

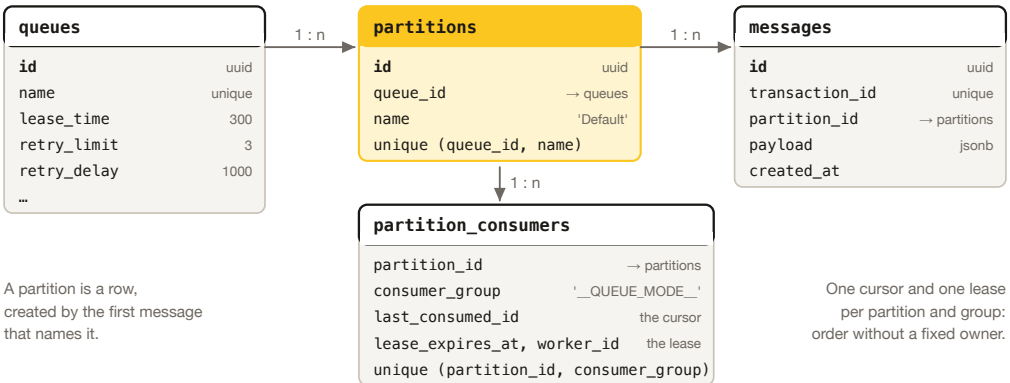


Figure 1.1: The model as the broker's first schema wrote it (October 2025): four tables.

Because we had to solve the same problem in another part of the application, we started to think about a generic solution. The idea was simple: an HTTP API on a Node.js server, which writes to and reads from Postgres tables. After the first MVP, we started to think that something good could come out of it. So we rewrote the MVP in C++, to get a statically typed language with stronger correctness guarantees, and replaced the SQL queries with stored procedures in Postgres. One month after the first idea, that custom broker was running in production, first for the heavy LLM queue, and one week later for all the other original Kafka topics too.

Then, in July 2026, we redesigned the way data was saved in Postgres: no more a row for each message, but segments with offsets. To make it fast, we put a lot of in-memory data structures in the broker to speed up consuming. The new storage was so fast that our C++ design became the bottleneck. After some thought, we decided to rewrite it in Rust, because fast async I/O is much easier to write there than in C++ (no more custom libuv loops: we relied on the Tokio async runtime).

This version was faster than anything we ever needed, but it had one structural issue: the more data you store in the broker, the slower the system becomes. Postgres is fast and configurable, but it is a system you cannot fully tune from the outside: you have to follow its rules.

So we started to think about a new class of queue, stored on disk next to the ones in the database. The idea was sound, and after some tests we decided to remove the Postgres queue engine completely, write the storage on disk ourselves, and replicate it with Raft.

The result of this rewrite is Queen today: a message broker replicated with Raft, with a single leader per tenant, able to sustain millions of messages per second at low latency, with a virtually unlimited number of partitions.

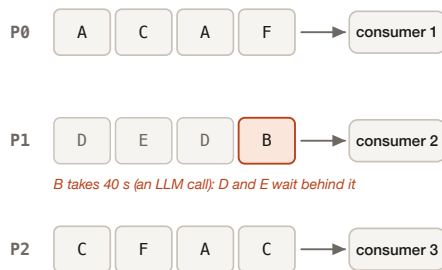
1.3 The model

The model of QueenMQ is easy to understand. It rests on four ideas: partitions, leases, transactions and a simple interface.

1.3.1 Partitions

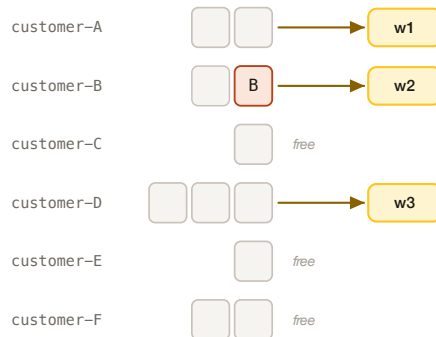
The core of the system are the partitions. A QueenMQ partition is a FIFO sub-queue inside a queue, created on the fly, and a queue is a named collection of those sub-queues. The first push that names a partition creates it, so there is no partition count to choose and nothing to plan: you give each entity its own partition, a customer, an order, a conversation, and one queue has held ten million of them.

Kafka: keys hashed onto 3 partitions



Head of the partition on the right. One slow message holds up every key that hashes to the same partition.

Queen: one partition per customer



Each customer is its own ordered partition, created by its first push. B's slow message delays only customer B.

Figure 1.2: Hashed partitions share one queue per partition; Queen gives every entity its own.

QueenMQ partitions are not physical entities like in Kafka: there is no dedicated thread or file for each partition. A partition is a name, a few rows in the broker's memory and a position for each consumer group, and its messages live in the log files of its queue.

WHY *order matters per entity, not across the whole queue. In Kafka you choose a number of partitions and hash your keys onto them, so a slow key blocks every key that shares its partition: the head-of-line blocking of the Smartchat story. With one partition per entity, a slow conversation waits only for itself.*

1.3.2 Leases, not static assignment

A consumer does not own partitions. When it asks for work, the broker gives it a batch from a partition that is free, and leases that partition to it for a while (`leaseTime`). Until the consumer acks the batch, or the lease runs out, no other consumer of the same group gets that partition. Then the partition is free again, for whoever asks next.

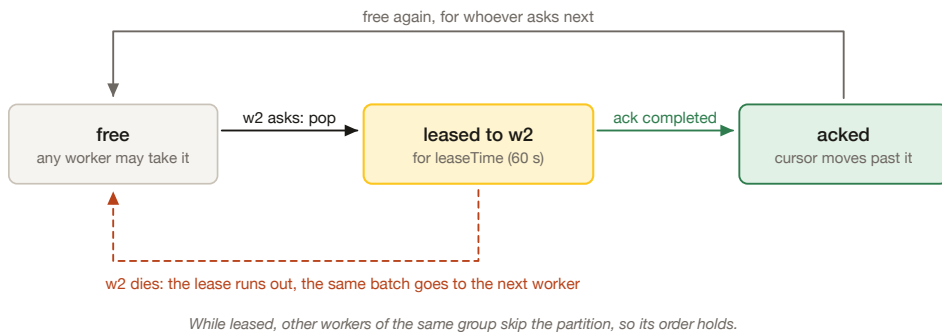


Figure 1.3: A partition's cycle for one consumer group: free, leased, acked, free again.

Consumer groups give every reader its own position: each group gets every message, and the workers inside a group share its work. And reading does not delete: an ack only moves the group's position, and the messages stay until retention removes them, so a new group can read the history again.

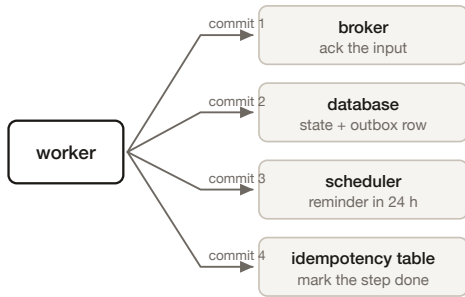
WHY *with no static assignment there is no rebalance. You add a worker and it starts taking work; a worker dies and its lease runs out, and someone else continues from the same message. The order of each partition is kept, because a partition has one holder at a time.*

1.3.3 Transactions

A worker's step usually does several things: it finishes the message it read, pushes the next events, saves some state, and maybe schedules something for later. In QueenMQ that is one call: a transaction that acks, pushes to any queue, writes to the key-value store and schedules timers, and commits as a single entry of the log, all or nothing.

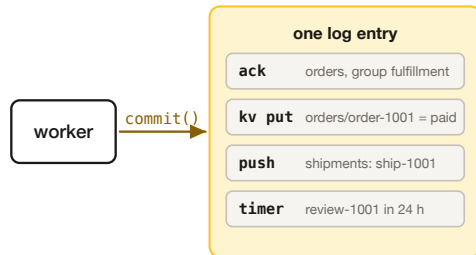
WHY *without it, every step needs an outbox table, an idempotency table and a cron job, plus the code that keeps them in step. With it, a crash can never leave a step half done.*

Without: four systems, four commits



a crash between two commits: the step is half done

With Queen: one transaction, one entry



all or nothing

Figure 1.4: One worker step without and with a transaction.

1.3.4 A simple interface

QueenMQ chose a simple API over HTTP: JSON on one port, the same for pushing, consuming, transactions, KV, timers and locks. The SDKs (JavaScript, Python, Go, Rust, PHP, C++) are thin layers over it, `curl` is enough to try everything, and Kafka clients can talk to the broker through its built-in Kafka port.

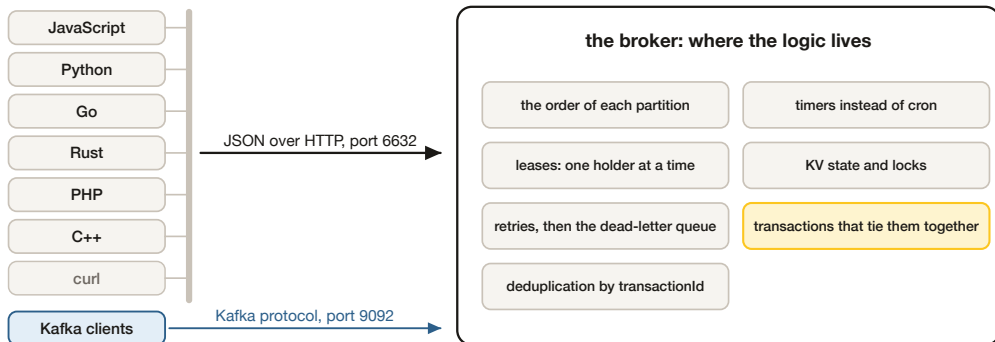


Figure 1.5: Thin clients; the logic lives in the broker.

WHY any language with an HTTP client can use QueenMQ, and what you see on the wire is easy to read and debug.

1.3.5 Wrap up

In QueenMQ, clients are dumb and the broker is where all the logic lives: the order of each partition, the leases, the retries and the dead-letter queue, deduplication, timers, state, and the transactions that tie them together. Your application keeps only its own business.

2

Writing applications with QueenMQ

For this book, we chose to show code with the QueenMQ JavaScript SDK, because we think it is one of the easiest languages for understanding how the system works. On the website, you will find all the examples of this book in every QueenMQ SDK.

2.1 Setup a local broker

One binary serves the HTTP API and the dashboard on one port:

```
docker run -p 6632:6632 ghcr.io/queen-mq/queen:2.6.0
```

Open <http://localhost:6632> and watch every step of this part land in the dashboard (Figure 2.1). Then add the client:

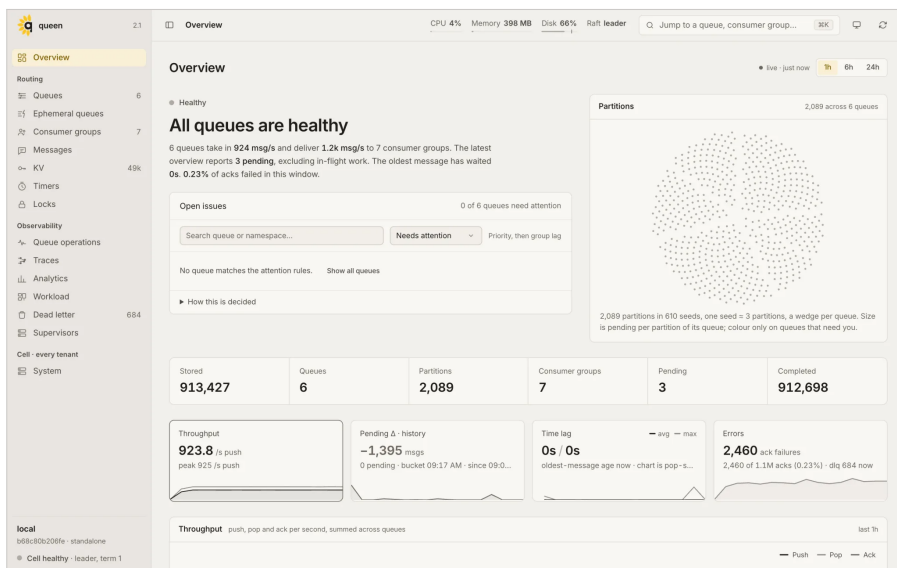


Figure 2.1: The dashboard, served on the same port as the API.

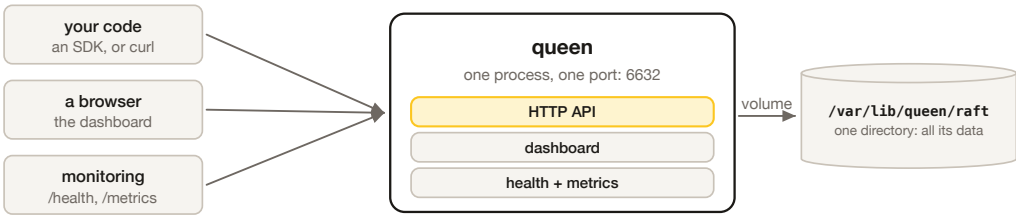


Figure 2.2: One process serves the API, the dashboard, health and metrics on port 6632.

```
npm install queen-mq
```

2.2 Create and configure queues

A queue is a named set of partitions that share one configuration. You never have to create one: the first push that names a queue creates it, with the defaults. Configure it when you want other settings.

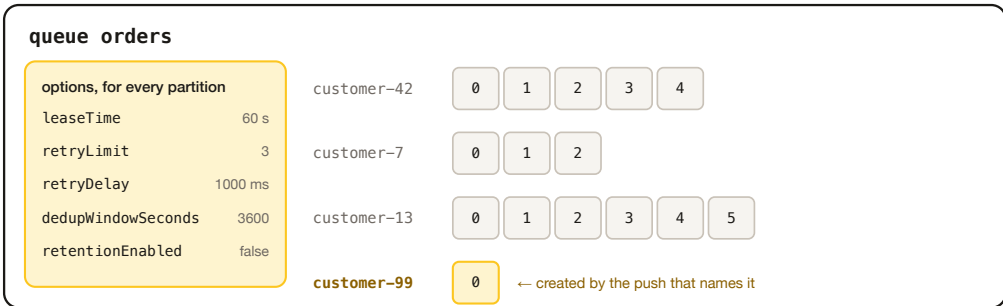


Figure 2.3: A queue: options shared by its partitions, each an ordered list of messages.

```
import { Queen } from 'queen-mq'

const queen = new Queen({ url: 'http://localhost:6632' })

await queen.queue('orders')
  .namespace('shop')           // groups queues in the dashboard
  .config({
    leaseTime: 30,              // seconds a worker holds a message before it is redelivered
    retryLimit: 3,              // failures before the message goes to the dead-letter queue
    retentionEnabled: true,     // retention runs only when this is on
    retentionSeconds: 86400,    // then messages older than a day are removed
  })
  .create()
```

WHY *a queue created by a push is good enough to start, and most queues never need more. When they do, the settings live on the queue, in one place, instead of in every worker.*

2.2.1 Every option

Here is a queue that sets most of them, and the table below says what each one does.

```
await queen.queue('jobs')
  .config({
    leaseTime: 120,              // 2 minutes per job, then it goes to another worker
    retryLimit: 5,              // after 5 failures, the dead-letter queue
    retryDelay: 10000,          // a failed job comes back after 10 s, not at once
    delayedProcessing: 30,      // a new job waits 30 s before anyone can take it
    windowBuffer: 5,            // a burst waits 5 s, then is handed over at once
    dedupWindowSeconds: 86400,  // a transactionId is remembered for a day
    retentionEnabled: true,     // retention runs only when this is on:
    completedRetentionSeconds: 3600, // jobs every group has finished go after an hour
    maxWaitTimeSeconds: 604800, // anything older than a week goes, finished or not
  })
  .create()
```

Option	Default	What it does
leaseTime	60 s (300 s from create())	How long a worker holds the messages it took before they go to another worker.

Option	Default	What it does
<code>retryLimit</code>	3	How many failed attempts a message gets. After the last one it goes to the dead-letter queue.
<code>retryDelay</code>	1000 ms	How long a failed message waits before it comes back. Its partition waits with it, for that consumer group.
<code>deadLetterQueue, dlqAfterMaxRetries</code>	true	File a message that ran out of retries in the dead-letter queue. With both off, it is dropped.
<code>delayedProcessing</code>	0 s	How long every new message waits after its push before anyone can take it.
<code>windowBuffer</code>	0 s	How long new messages wait in their partition, so a burst is handed over as one batch.
<code>dedupWindowSeconds</code>	3600 s	How long a partition remembers <code>transactionIds</code> . 0 turns deduplication off.
<code>retentionEnabled</code>	false	Must be true for the two options below to do anything.
<code>retentionSeconds</code>	0	Remove messages older than this, consumed or not.
<code>completedRetentionSeconds</code>	0	Remove messages older than this that every consumer group has already acked.
<code>maxWaitTimeSeconds</code>	0	Remove messages older than this, consumed or not. Works without <code>retentionEnabled</code> .
<code>encryptionEnabled</code>	false	Encrypt payloads on disk. Every node needs <code>QUEEN_ENCRYPTION_KEY</code> .
<code>namespace, task</code>	from the name	Labels that group queues in the dashboard and in a discovery pop.

Leases. A worker that takes messages holds them under a lease. Until it acks them or the lease runs out, no other worker of the same consumer group gets that partition. **WHY** a worker can crash in the middle of its work. The lease is how its messages get back to someone else without a human noticing. Set it longer than your slowest handler, or renew it while the handler runs (see *Pop messages*).

Retries. A failed message comes back `retryLimit` times, each time after `retryDelay`, and then goes to the dead-letter queue.

WHY *most failures are temporary: a database that restarts, a provider that times out. Trying again later fixes them. Waiting between tries gives the other side time to recover, and a message that can never succeed stops after a few tries instead of blocking its partition forever.*

Delays. `delayedProcessing` holds every new message of the queue for a while. `windowBuffer` holds a group's new messages until the burst is over, so the handler gets them together.

WHY *some work should not start at once (a reminder sent five minutes after a signup), and some work is cheaper in bulk (one database write for a burst of fifty updates). For a delay that is different for each message, use a timer instead (see KV, Locks and Timers).*

Deduplication. Inside the dedup window, a second push with the same `transactionId` in the same partition is not written.

WHY *producers retry when they do not get an answer, and a retry must not create a second order.*

Retention. By default Queen keeps every message, and the disk grows. Retention runs only when `retentionEnabled` is true; then `retentionSeconds` removes old messages, read or not, and `completedRetentionSeconds` removes old messages that every group has finished. `maxWaitTimeSeconds` removes old messages without needing `retentionEnabled`. A node whose disk is 85% full refuses the writes that would grow it.

WHY *an acked message is not deleted: it stays in its partition so that a new consumer group, an audit or a fixed bug can read it again. Retention is how you decide how long that history is worth its disk.*

Encryption. With `encryptionEnabled`, payloads are encrypted on disk.

WHY *some payloads hold personal data that must not sit on a disk in clear text.*

2.2.2 Three traps

- `config(...).create()` in the JavaScript client sends your options together with the client's own defaults (a 300-second lease, encryption off, no retention). Calling it on an existing queue resets every option you leave out. Name every option you care about.

- On a node without `QUEEN_ENCRYPTION_KEY`, an encrypted queue stores plain text, and nothing warns you.
- `priority`, `ttr`, `maxSize` and `minPopWaitTime` are accepted and stored, and do nothing.

On the site, with the code in all six SDK languages: queenmq.com/writing-applications/queues.

2.3 Push messages

In Kafka you choose a number of partitions and hash keys onto them, so a slow customer holds up every customer that shares its partition. Here a partition is one entity: a customer, an order, a conversation. The push that names it creates it, order holds inside it, and one queue has held ten million.

The `transactionId` makes a retry safe: the same id within the dedup window (an hour by default) is not written twice.

```
await queen.queue('orders')
  .partition('customer-42')    // one ordered partition per customer, created by this push
  .push([
    {
      transactionId: 'order-1001', // the dedup key: pushing the same id again is a no-op
      data: { orderId: 1001, customer: 42, total: 59.9 },
    }
  ])
```

WHY *order matters per entity (an order's events must be handled in sequence), not across the whole queue. One partition per entity keeps that order without one slow entity holding up the others. A push with no partition goes to a partition called `Default`.*

2.3.1 Every message gets its own answer

One push can carry many messages, and the broker answers each one: queued (stored, with its offset), duplicate (that `transactionId` is already there, nothing was written) or error (refused, nothing written). The JavaScript client throws when any message is refused, unless you handle it yourself, and never throws for a duplicate.

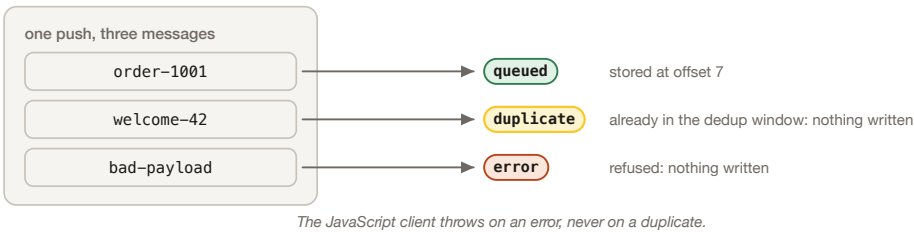


Figure 2.4: One push, one answer per message.

```
await queen.queue('emails')
  .partition('customer-42')
  .push([
    { transactionId: 'welcome-42', data: { to: 'ada@example.com' } }
  ])
  .onSuccess((items) => console.log('stored', items.length))
  .onDuplicate((items) => console.log('already stored', items[0].transactionId))
  .onError((items, error) => console.error('refused', error.message))
```

WHY a duplicate is usually good news: it means an earlier attempt already got through. A refusal is not. Telling them apart per message lets one bad item in a batch of a hundred be retried alone, and lets a producer that retried after a timeout know that its first attempt worked.

2.3.2 Buffering, for many small messages

Each push is one HTTP request. A producer that pushes thousands of small messages a second can let the client collect them and send them in batches instead.

```
for (let i = 0; i < 1000; i++) {
  await queen.queue('events')
    .partition(`device-${i % 10}`)
    .buffer({ messageCount: 100, timeMillis: 1000 }) // 100 at a time, or every second
    .push([
      { transactionId: `reading-${i}`, data: { device: i % 10, value: i } }
    ])
}
await queen.flushAllBuffers() // before exiting, send what is left
```

A buffer is sent when it holds `messageCount` messages or when `timeMillis` has passed, whichever comes first. If the producer gets far ahead of the network (four times `messageCount` waiting), a buffered push waits instead of letting memory grow.



Figure 2.5: Client buffering: many small pushes, a few requests.

WHY one request per message spends most of its time on the round trip, not on the message. Batching gets far more messages through the same connection. The price: a message is stored only when its batch is sent, so a process that crashes loses what was still in its buffer. Call `flushAllBuffers()` before you exit, and do not buffer messages you cannot afford to lose.

2.3.3 When several messages must land together

A push of several messages is not all-or-nothing: each message gets its own answer, so some can be stored while others are refused. When messages must be stored together or not at all (an order and its payment request, say), push them in a transaction (see Advanced transactional usage).

2.3.4 Binary push, compression and framed pop

By default the client and the broker speak JSON. Three options of the client change how messages travel, and nothing else: the calls, their results, deduplication and the guarantees stay the same.

Option	Default	What it does
<code>binaryPush</code>	false	A push, direct or buffered, goes as one binary body instead of a JSON document. A transaction that pushes sends its pushes the same way.
<code>pushCompression</code>	false	Needs <code>binaryPush</code> . The client packs each partition's batch the way the broker stores it, and compresses it with <code>zstd</code> once. The broker keeps those bytes as they are.

Option	Default	What it does
framedPop	false	pop() and consume() get the records as they are stored, and the client unpacks them into the same messages.

Turn them on when you create the client:

```
const fast = new Queen({
  url: 'http://localhost:6632',
  binaryPush: true,      // a push travels as one binary body, not JSON
  pushCompression: true, // with binaryPush: each partition's batch is zstd-compressed here
  framedPop: true,       // a pop gets the records as stored, and unpacks them here
})

await fast.queue('invoices').partition('customer-42')
  .push([
    { transactionId: 'invoice-2001', data: { orderId: 2001, total: 12.5 } }
  ])

await fast.queue('invoices').each().limit(1).consume(async (msg) => {
  console.log(msg.data) // { orderId: 2001, total: 12.5 }, as with JSON
})
```

zstd comes with Node.js, so there is nothing to install. Some requests stay JSON by themselves: a push with a traceId, a pop of a conflating group, and a pop of an encrypted queue. Acks, KV, timers and locks do not change.

WHY *at a high message rate, the broker spends CPU turning JSON into stored records and back, and the network carries every message uncompressed. These options move that work into the clients, where many processes share it, and send the batches compressed. When neither the broker's CPU nor the network is the limit, leave them off: JSON is easier to read when you debug.*

One trap: binaryPush and pushCompression need broker 2.3.0 or later on every node. An older broker answers 400, and the client does not fall back to JSON, so turn them on after the last node is upgraded. framedPop is safe with any broker: an older one ignores it and answers JSON.

On the site, with the code in all six SDK languages: queenmq.com/writing-applications/push. More there: [the binary forms in every SDK](#).

2.4 Pop messages

There are two ways to read. `consume()` is a loop: it takes messages, calls your handler, acks them when the handler returns, and goes on. `pop()` is one call: it hands you the messages and leaves the ack to you.

WHY *most workers are loops, and `consume()` writes the loop for you, with the error handling done right. `pop()` is for the cases a loop does not fit: a scheduled job that reads once, a script, or code that wants to decide itself when to ack.*

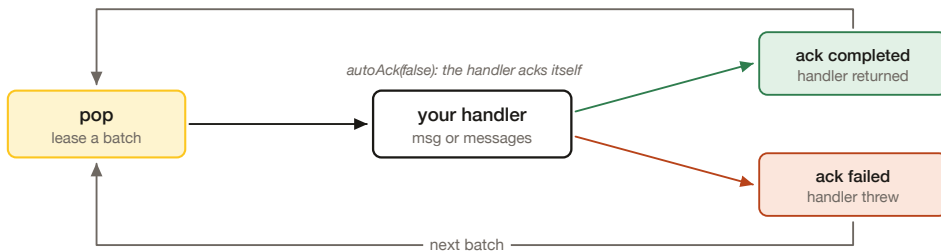


Figure 2.6: What `consume()` does for you.

2.4.1 Queue mode and consumer groups

Without a consumer group, each message goes to one worker, and the broker sizes every pop for you.

```

await queen.queue('orders')
  .each() // the handler gets one message at a time
  .limit(1) // stop after one message; a worker leaves this out
  .consume(async (msg) => { // acked when the handler returns, retried if it throws
    console.log(msg.data)
  })

```

With a consumer group, every group gets every message and the workers of one group share them. A new group starts at the tail, so this one asks for everything already in the queue. Here every option is set by hand, and the handler acks.

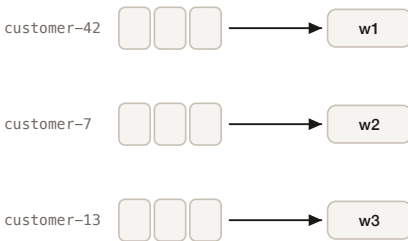
```

await queen.queue('orders')
  .group('billing')           // every group gets every message
  .subscriptionMode('all')    // a new group starts at the tail unless told otherwise
  .batch(100)                 // up to 100 messages per pop...
  .partitions(10)             // ...from up to 10 partitions
  .renewLease(true, 2000)     // heartbeat every 2 s while the handler runs
  .autoAck(false)            // the handler decides the outcome
  .each()
  .limit(1)
  .consume(async (msg) => {
    const ok = msg.data.total > 0
    await queen.ack(msg, ok ? 'completed' : 'failed', { group: 'billing' })
  })
    
```

Ack failed and the message comes back, up to the queue's `retryLimit`; after the last retry it goes to the dead-letter queue.

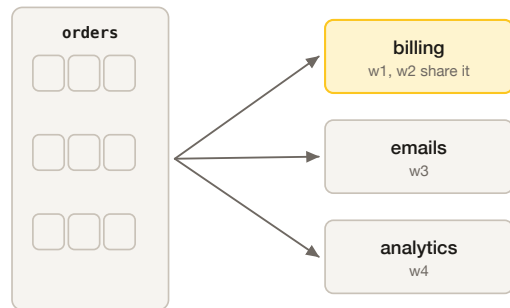
WHY *one queue often feeds several services: billing, emails and analytics all want every order. Each service is a consumer group, reading at its own pace, and none of them can slow the others down. Inside a group, the workers compete, so you add workers to go faster.*

No group: each message to one worker



Workers compete. A partition has one holder at a time.

Groups: each group gets every message



own position, own pace

Figure 2.7: Queue mode shares the messages; every consumer group gets all of them.

2.4.2 One message at a time, or a whole batch

With `.each()`, the handler gets one message at a time. Without it, the handler gets the whole batch as an array.

```
await queen.queue('events')
  .group('archiver')
  .subscriptionMode('all')
  .batch(100) // up to 100 messages per call
  .limit(1000)
  .consume(async (messages) => { // no .each(): the handler gets the whole batch
    console.log('archiving', messages.length)
  }) // all acked if it returns, all nacked if it throws
```

What the client does with your handler's result (autoAck, on by default):

- **It returns:** the message (or the whole batch) is acked.
- **It throws:** the message (or the whole batch) is nacked as failed, which spends one retry, and the loop goes on with the next messages.
- **It throws under .each():** the rest of that partition's messages in the same batch are skipped, because they will be delivered again after the failed one, in order. Messages of other partitions in the batch are handled as usual.
- **With .autoAck(false):** the handler acks itself (in a transaction, for example). A handler that throws is still nacked, so its partition is not stuck until the lease runs out.

WHY a batch is cheaper (one ack for a hundred messages, one database write for a hundred rows), and one at a time is simpler and fails one message instead of a hundred. Pick per worker.

If you add .onSuccess() or .onError() to a consumer, the client stops acking and nacking for you: you decide in those callbacks what happens to each message.

```
await queen.queue('emails')
  .group('mailer')
  .subscriptionMode('all')
  .each()
  .limit(1)
  .consume(async (msg) => sendEmail(msg.data))
  .onSuccess((msg) => queen.ack(msg, 'completed', { group: 'mailer' }))
  .onError((msg, error) => queen.ack(msg, error.permanent ? 'dlq' : 'failed',
    { group: 'mailer', error: error.message })))
```

WHY some failures should be retried and some never will succeed (an email address that does not exist). The callbacks let you send the first kind back with failed and the second kind straight to the dead-letter queue with dlq.

2.4.3 How many messages a pop takes: *autopilot*

By default the broker decides how many messages each pop gets (batch, from 100 to 1,000, based on how fast the group acks) and from how many partitions (partitions, the ones that have work, shared among the waiting workers). A value you set yourself is never changed. `partitions(N)` lets one pop take messages from up to `N` partitions (1 to 64), with batch as the total. `.autopilot(false)` goes back to one message from one partition per pop.

WHY *the right batch size depends on your handler and your load, and it changes during the day. The broker sees both, so it can size every pop better than a number fixed in the code.*

2.4.4 Leases and long work

A pop leases its messages to the worker for the queue's `leaseTime`. Work that can take longer renews the lease while it runs: `.renewLease(true, 10000)` renews every 10 seconds, and `queen.renew(message)` renews once. In PHP, the renewal happens between messages, so a single long handler calls the renew callback it is given. A lease that has already expired cannot be renewed: its messages go to the next worker, marked as one delivery attempt more.

WHY *the lease is what protects order and stops two workers from doing the same work. A short lease gets a crashed worker's messages moving again quickly; renewing lets a slow but healthy worker keep them.*

2.4.5 Where a new group starts

A group's starting point is decided on its first pop and then stored: `new` (the default) reads only what is pushed after that first pop, `all` reads everything still kept, and `subscriptionFrom` reads from a point in time. A group that already exists moves with a seek, to a time or to the end.

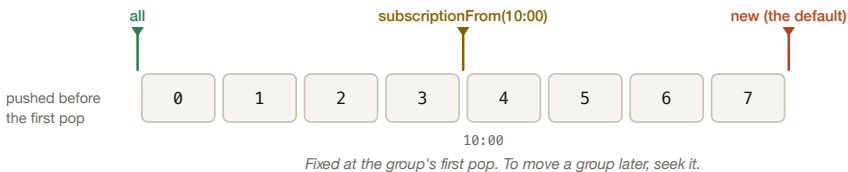


Figure 2.8: *Where a new group starts, fixed at its first pop.*

```
await queen.admin.seekConsumerGroup('archiver', 'events', {
  timestamp: '2026-10-22T00:00:00Z',
})
// the archiver group now reads again everything pushed since that time

await queen.admin.seekConsumerGroup('archiver', 'events', { toEnd: true })
// and now it skips everything it has not read yet
```

WHY *a new service usually wants only new events, a new projection wants the whole history, and a bug fix wants to replay the last day. A seek to the end is how a group skips a backlog it no longer needs.*

2.4.6 Only the newest message: conflation

A group with `.conflation()` gets only the newest message of each partition, and its cursor moves past the older ones.

```
const [latest] = await queen.queue('prices')
  .group('repricer')
  .subscriptionMode('all')
  .conflation() // only the newest message of each partition
  .wait(false)
  .pop()
console.log(latest.data) // { sku: 'A-1', price: 12 }: 10 and 11 are skipped
await queen.ack(latest, 'completed', { group: 'repricer' })
```

WHY *some queues carry commands where only the latest one matters: “recompute the price of A-1”. A worker that is behind should do the work once, with the newest input, instead of three times with inputs that are already old.*

Conflation is decided by the group’s first pop, needs a consumer group, and cannot be combined with `commitOnDelivery`.

2.4.7 Faster, with no ack: commitOnDelivery

A pop with `.commitOnDelivery()` moves the group’s cursor as the messages are handed out. There is no lease and nothing to ack.

```
const readings = await queen.queue('metrics')
  .group('live-chart')
  .subscriptionMode('all')
  .commitOnDelivery() // the cursor moves as they are handed out: no ack
  .batch(100)
  .wait(false)
  .pop()
```

WHY *a live dashboard or a metrics chart can lose a reading now and then, and would rather not pay for leases and acks. This is at-most-once delivery: a worker that crashes after the pop loses that batch. It works with pop() only; consume() refuses it.*

2.4.8 Running a consumer for real

```
const stop = new AbortController()
process.once('SIGTERM', () => stop.abort()) // a deploy stops the worker cleanly

await queen.queue('tasks')
  .group('workers')
  .subscriptionMode('all')
  .concurrency(4) // four handlers at once, on four partitions
  .renewLease(true, 10000) // long tasks keep their lease alive
  .idleMillis(60000) // or stop after a minute with nothing to do
  .each()
  .consume(async (task) => runTask(task.data), { signal: stop.signal })
```

- `concurrency(n)` runs `n` handlers at once, each on its own partitions.
- `limit(n)` stops after `n` messages, and `idleMillis(ms)` stops after that long with nothing to do.
- `wait(true)` (the default) waits for messages instead of returning empty, up to `timeoutMillis(ms)` (30 seconds by default, at most 60).
- A signal stops the loop at once. Messages it had taken but not started go back with a retry ack, which spends no retry, so a deploy does not leave partitions waiting for their leases to run out.

WHY *a worker runs for weeks, through deploys and quiet nights. These options decide how much it does at once, when it stops, and how it gives its work back when it is told to.*

2.4.9 Read it again

Consuming does not delete: the queue is a log, kept for its retention. A new group can start from a point in time and read the same orders again, while the groups that already read them stay where they are.

```
await queen.queue('orders')
  .group('audit') // a new group, with its own cursor
  .subscriptionFrom('2026-10-22T00:00:00Z') // everything pushed from this time on
  .each()
  .limit(1)
  .consume(async (msg) => {
    console.log('again:', msg.data)
  })
```

Or read a partition by offset with `fetch`: no group, no lease, no ack. It suits an audit, rebuilding one entity, or a sink that keeps its own position.

```
const read = await queen.queue('orders').fetch(['customer-42']) // no group, lease or ack
for (const record of read.get('customer-42').records) {
  console.log(record.offset, record.payload)
}
```

WHY *history is useful long after the first reader is done with it: to build a new view, to audit what happened, or to rebuild one customer's state from its events.*

On the site, with the code in all six SDK languages: queenmq.com/writing-applications/pop. More there: [reads by offset in every SDK](#).

2.5 Acknowledge messages

An ack tells the broker what happened to a message. A completed ack moves the consumer group's cursor past it, and that is all: the message stays in its partition until retention removes it.

WHY *the cursor is the group's memory of where it is. Keeping the message after the ack is what lets other groups, and later replays, read it too.*

2.5.1 Four answers



Figure 2.9: An ack moves the group's cursor; the message stays.

Status	Spends a retry	What happens, and when to use it
completed	no	Done. The cursor moves past it. <code>true</code> means the same.
failed	yes	The lease is released and the message comes back after <code>retryDelay</code> . After the last retry it goes to the dead-letter queue. Use it for failures that may pass. <code>false</code> means the same.
retry	no	The lease is released and the message comes back at once. Use it to give work back (a worker that is shutting down).
dlq	no	Filed in the dead-letter queue now. Use it when trying again cannot help (a payload that will never parse).

A failed or dlq ack can carry the reason (error: 'card declined' next to the group), and the dead-letter queue keeps it.

WHY *"it did not work" has different meanings. Sometimes trying again in a while is right, sometimes now, and sometimes never. Each answer sends the message where it should go.*

2.5.2 One message or a whole batch

`queen.ack()` takes one message or an array. With an array, one call acks the whole batch.

```
const messages = await queen.queue('events')
  .group('auditor')
  .subscriptionMode('all')
  .batch(50)
  .wait(true) // long poll: wait for messages, never return empty
  .pop()
console.log('auditing', messages.length)
await queen.ack(messages, 'completed', { group: 'auditor' }) // the whole batch at once
```

Each message of an array can carry its own answer:

```
await queen.ack([
  { ...first, _status: 'completed' },
  { ...second, _status: 'failed', _error: 'timeout' },
], 'completed', { group: 'moderation' })
```

In a batch, the cursor stops at the first message that will come back: a completed message after it is delivered again too.

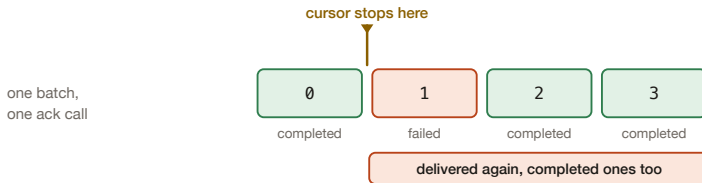


Figure 2.10: In a batch, the cursor stops at the first message that comes back.

WHY one call for a hundred messages is much cheaper than a hundred calls. The cursor rule keeps order: a message is never passed while an earlier one of the same partition is still to be done.

2.5.3 Three things to know

- **Inside `.each()`, throw to fail a message.** When the handler throws, the client nacks it and skips the rest of that partition's batch. When the handler acks `failed` itself and returns, the client does not know: it goes on with the partition's next messages, their acks are refused (that part of the batch is coming back anyway), and they are handled twice.

- **Every ack answers HTTP 200, even a refused one.** Read success in the answer: a refused ack usually means the lease ran out and another worker has the message now.
- **An ack carries its consumer group and its lease.** The client takes both from the message. Over HTTP you send them yourself: an ack without its lease is not checked, and can complete a batch another worker holds.

On the site, with the code in all six SDK languages: queenmq.com/writing-applications/ack.

2.6 Retries and the dead-letter queue

Every failed ack spends one retry. The message comes back after the queue's `retryDelay`, and its partition waits with it, for that consumer group. After `retryLimit` failures, the message goes to the dead-letter queue and the group moves on. You set both on the queue (see Create and configure queues).

WHY *retrying is how temporary failures heal by themselves, and the limit is how a message that can never succeed stops blocking everything behind it in its partition.*

Two cases do not spend a retry: a retry ack, and a lease that runs out. So a message that crashes its worker's process every time comes back forever and never reaches the dead-letter queue. If some input can crash your worker, check for it before the work starts and ack such a message with `dlq`.

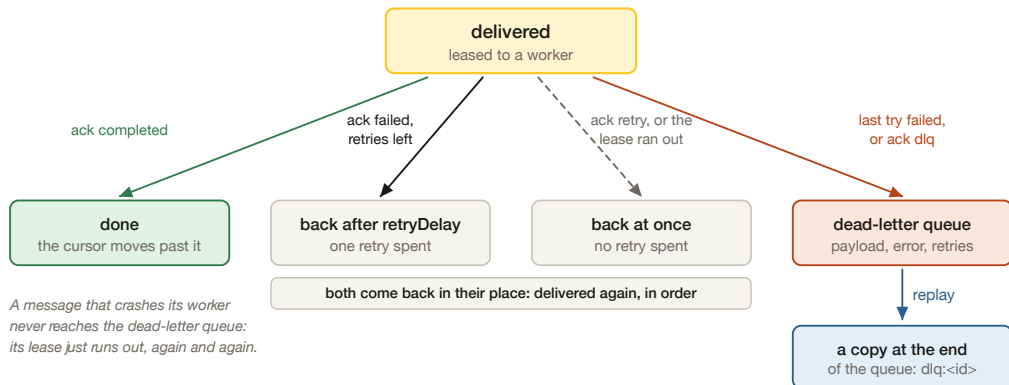


Figure 2.11: *The life of a message: four endings, and the replay.*

2.6.1 What a dead letter keeps

A dead letter keeps the payload, the error the worker sent, the number of retries, and the consumer group that gave up on it. It stays until you replay it or delete it. Each group has its own: a message that failed for billing can be fine for analytics.

WHY *a failure you cannot see is a failure you cannot fix. The dead-letter queue is where they wait for a person, or for a fix in the code.*

2.6.2 Read it

```
const dead = await queen.queue('payments').dlq('charger').limit(20).get()
for (const letter of dead.messages) {
  console.log(letter.transactionId, letter.errorMessage, letter.retryCount)
}
```

`limit()` and `offset()` page through it, newest first. The client also has `from()` and `to()`, but the 2.6.0 broker ignores them.

2.6.3 Replay it

Once the cause is fixed, put the messages back:

```
for (const letter of dead.messages) {
  // back into its partition, and out of the dead-letter queue
  await queen.admin.retryMessage(letter.partitionId, letter.transactionId)
}
```

A replay moves the dead letter back into its partition, at the end, as a new message with the `transactionId` `dlq:<id>`, and removes it from the dead-letter queue in the same step. Doing it twice cannot create a second copy. Every group that reads that partition sees the replayed message, not only the group that gave up on it. Over HTTP the same thing is `POST /api/v1/dlq/:id/replay`.

WHY *the fix usually comes after the failure: a bug deployed, a provider back up. Replay lets the same messages go through again without anyone pushing them by hand.*

2.6.4 Skip it, or turn it off

An ack with `dlq` files a message straight away, without spending its retries. Setting `deadLetterQueue` and `dlqAfterMaxRetries` to false on the queue drops a message that runs out of retries instead of keeping it.

WHY *some messages are worth nothing once they fail (a typing indicator, a stale price), and keeping them would only fill a list nobody reads.*

On the site, with the code in all six SDK languages: queenmq.com/writing-applications/retries. More there: [consuming](#).

2.7 Advanced transactional usage

The ack, the order's new state, the next event and tomorrow's reminder commit as one log entry, or none of them does. That one call replaces the outbox table, the idempotency table and the cron job.

```
await queen.queue('orders')
  .group('fulfillment')
  .subscriptionMode('all')
  .autoAck(false)
  .each()
  .limit(1)
  .consume(async (msg) => {
    const { orderId, customer } = msg.data
    await queen.transaction()
      .ack(msg, 'completed', { consumerGroup: 'fulfillment' }) // the input is done
      .kv.put('orders', `order-${orderId}`, { status: 'paid' }, { ttl: '7d' })
      .queue('shipments').partition(msg.partition) // the next event
      .push([ { transactionId: `ship-${orderId}`, data: { orderId } } ])
      .timer('reminders').key(`review-${orderId}`).delay('24h') // a message tomorrow
      .payload({ orderId, customer }).schedule()
      .commit()
  })
```

Then open the dashboard: its KV page holds the order's new state, and its Timers page holds tomorrow's reminder.

WHY *a worker's step is several writes: it marks its input done, saves what changed, tells the next service and plans what comes later. When those writes go to different systems, a crash between two of them leaves the step half done, and someone writes code to find and repair the halves. One commit has no halves.*

If the worker took too long and its lease ran out, the whole transaction is refused, so a slow worker can never write over the one that has the message now. If the step committed and only the answer got lost, a retry with the same transactionIds is refused whole, and nothing is written twice.

On the site, with the code in all six SDK languages: queenmq.com/writing-applications/transactions.

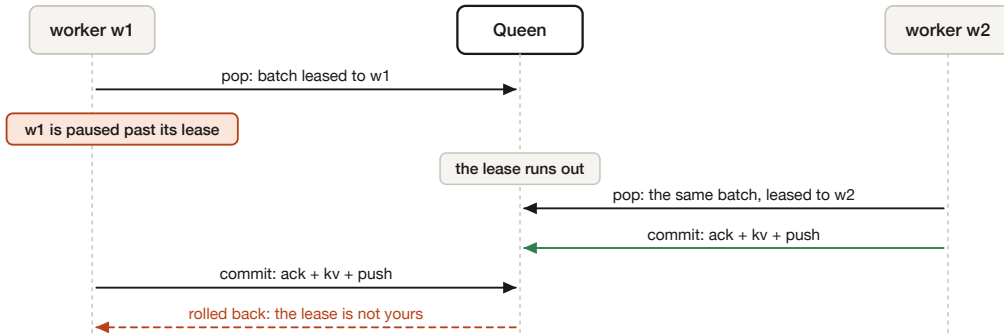


Figure 2.12: A worker whose lease ran out: its whole transaction rolls back.

2.8 KV, Locks and Timers

The transaction above already used two of the three: `kv.put` stored the order's state, and `timer()` scheduled tomorrow's reminder, which arrives as a message.

WHY KV is there so workers keep nothing between messages: the state of each order lives next to its events and changes in the same commit. Timers are there because “do this in 24 hours” is a message that should arrive later, in the order's own partition, not a cron job that scans a table. Scheduling the same timer key again moves it, so a reminder can be pushed back or cancelled.

KV also works on its own, outside a transaction. Every write names its lifetime, and the version a read returns makes the next write conditional:

```

// Every write names its lifetime: a ttl, or forever.
await queen.kv.put('carts', 'customer-42', { items: 3 }, { ttl: '7d' })
const cart = await queen.kv.get('carts', 'customer-42') // { found, value, version, ... }

// Write only if nobody wrote since we read it: the version is the condition.
const saved = await queen.kv.put('carts', 'customer-42', { items: 4 },
    { ttl: '7d', expect: cart.version })
console.log(saved.applied) // false if the version moved

// A counter with a ceiling: past max, the increment does not apply.
const hit = await queen.kv.incr('limits', 'api:customer-42', 1, { max: 100, ttl: '1m' })
if (!hit.applied) console.log('over the limit') // reason: 'limit'
    
```

WHY *two workers can read the same cart and both write it. With expect, the second write finds the version moved and is refused, with the current value in the answer, so nobody overwrites what they never saw. A counter with max is a rate limit in one call.*

The third is a lock. Some work no message triggers: a nightly report, a cleanup, a singleton. Take a lock for it. A lock is a lease, and a slow holder can outlive it, so guard the transaction with it: the push commits only while the lock is still yours.

```
const lock = queen.lock('daily-report', { ttl: '30s' }) // a lease with a fencing token
if (await lock.acquire()) {                             // false: another worker has it
  try {
    await queen.transaction()
      .guard(lock)                                     // commits only while the lock is ours
      .queue('reports').push([
        { transactionId: 'report-2026-10-22', data: { day: '2026-10-22' }
      ]})
      .commit()
  } finally {
    await lock.release()
  }
}
```

WHY *a cron job running on three replicas runs three times. A lock makes it run once, and the guard makes sure a holder that froze and lost its lock cannot commit after another one took over. A semaphore (`queen.semaphore(name, n)`) allows n holders at once.*

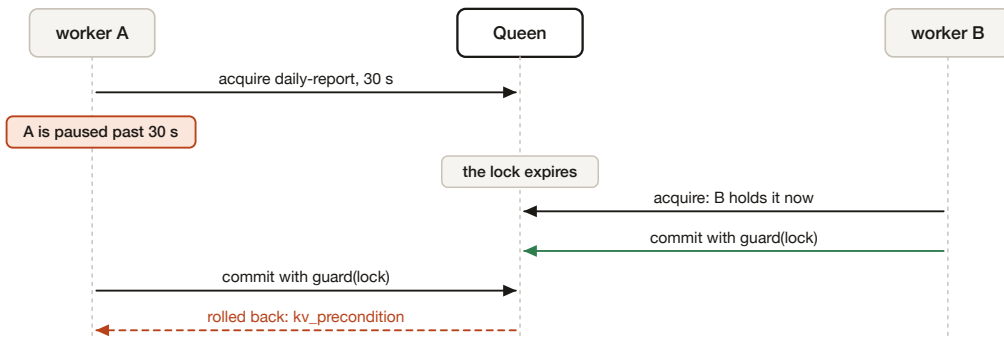


Figure 2.13: A lock expires without telling its holder; the guard keeps it out.

On the site, with the code in all six SDK languages: queenmq.com/writing-applications/kv-locks-timers.

2.9 Ephemeral queues

Ephemeral queues live in memory, outside the log, for request/reply. The worker answers on whatever inbox a request names; the caller makes a private inbox and waits on it. There is no transaction or replay here, and a crash loses what is in flight.

The worker:

```
const { messages } = await queen.ephemeral.pop('price-requests', { group: 'pricers', wait:
  ↳ true })
for (const m of messages) {
  await queen.ephemeral.push(m.payload.replyTo, [{ sku: m.payload.sku, price: 59.9 }])
}
await queen.ephemeral.ack('price-requests', messages, { group: 'pricers' })
```

The caller:

```
const inbox = `inbox-${crypto.randomUUID()}` // a private reply queue
await queen.ephemeral.push('price-requests', [{ sku: 'A-1', replyTo: inbox }])
const reply = await queen.ephemeral.pop(inbox, { wait: true, timeout: 5000, commitOnDelivery:
  ↳ true })
console.log(reply.messages[0]?.payload) // { sku: 'A-1', price: 59.9 }
```

WHY a request that waits for an answer (“what is the price of A-1, now?”) needs speed more than history. Writing it to disk, replicating it and keeping it for days would cost time for nothing. Ephemeral queues keep the same groups, leases and acks, and skip the log. Use a regular queue for anything that must survive a crash.

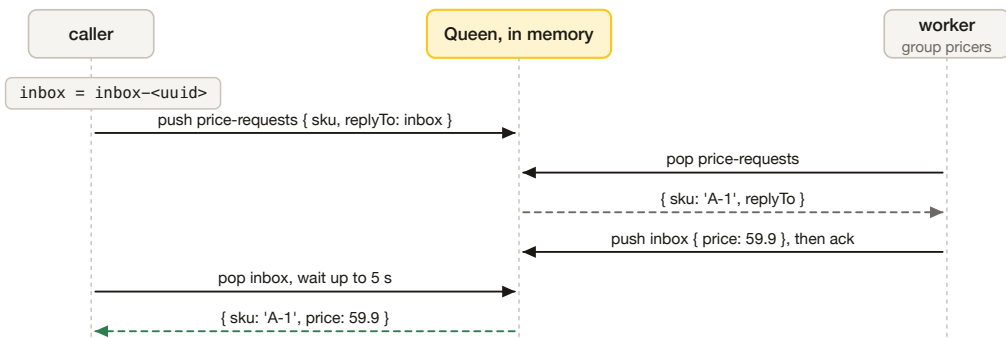


Figure 2.14: Request and reply over ephemeral queues, with a private inbox.

On the site, with the code in all six SDK languages: queenmq.com/writing-applications/ephemeral.

2.10 Stream processing

A stream reads a queue and writes its results to another. The partition is the key, so each customer gets its own windows. Like a new group it starts at the tail: push a few more orders and a result arrives a minute later.

```
import { Stream } from 'queen-mq'

const revenue = await Stream
  .from(queen.queue('orders'))           // one window per partition: per customer
  .windowTumbling({ seconds: 60 })
  .aggregate({ count: () => 1, sum: (order) => order.total })
  .to(queen.queue('revenue-per-minute')) // one result per customer per minute
  .run({ queryId: 'revenue-per-minute', url: 'http://localhost:6632' })
```

A stream reads its queue as a consumer group (streams.<queryId>) and keeps its state, the open windows, in the broker. Each time it closes windows, it writes the new state, pushes the results and acks the messages they came from in one transaction.

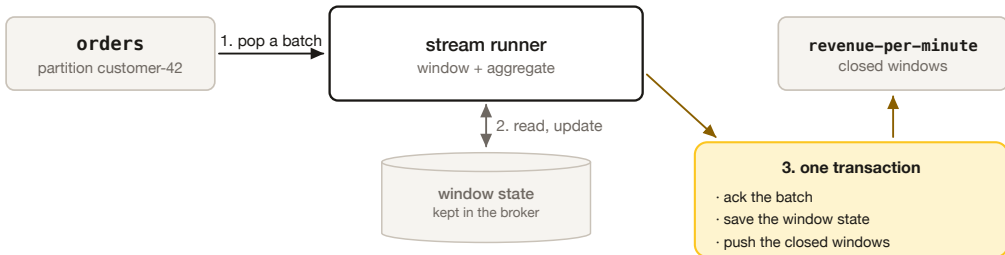


Figure 2.15: One cycle of a stream is one transaction.

WHY counting, summing and grouping events over time is a job most systems hand to a separate stream processor, with its own cluster and its own idea of what was processed. Here it runs in a normal process, its state lives next to the data, and one commit keeps the results and the inputs in step: a crash can neither lose a result nor count an event twice.

2.10.1 Four kinds of window

Window	Example	What it is, and when to use it
windowTumbling	seconds: 60	Fixed windows, one after the other. Revenue per minute.
windowSliding	size: 600, slide: 60	Overlapping windows: the last 10 minutes, every minute. Each event counts in several windows, so it keeps more state.
windowSession	gap: 1800	One window per burst of activity, closed after a quiet gap. A visit, a conversation.
windowCron	every: 'hour'	Windows aligned to the clock (second, minute, hour, day, week). Hourly and daily reports.

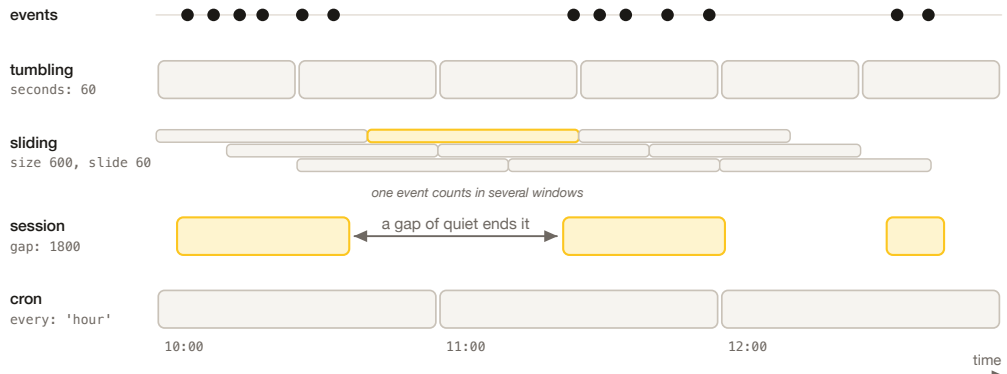


Figure 2.16: *The four kinds of window on one partition's events.*

```
const loginRate = await Stream
  .from(queen.queue('logins'))           // one partition per user
  .windowSliding({ size: 600, slide: 60 }) // the last 10 minutes, every minute
  .aggregate({ count: () => 1 })
  .to(queen.queue('login-rate'))
  .run({ queryId: 'login-rate', url: 'http://localhost:6632' })
```

```
const visits = await Stream
  .from(queen.queue('clicks')) // one partition per visitor
  .windowSession({ gap: 1800 }) // a visit ends after 30 minutes of quiet
  .aggregate({ count: () => 1 })
  .foreach(async (visit) => console.log('visit ended', visit)) // acked after this returns
  .run({ queryId: 'visits', url: 'http://localhost:6632' })
```

WHY *“in the last ten minutes”, “per visit” and “per hour” are different questions, and each window shape answers one of them without code of your own.*

2.10.2 When it happened, or when it arrived

By default a window holds the events that arrived during it. Give `eventTime` a function that reads the time from the message, and windows hold the events that happened during it instead. It gets the whole message, so the event’s own fields are under `data`; the aggregate’s functions get the event itself.

```
const hourly = await Stream
  .from(queen.queue('sales'))
  .windowCron({
    every: 'hour', // on the clock: 10:00 to 11:00, 11:00 to 12:00
    eventTime: (msg) => msg.data.soldAt, // when the sale happened, not when it arrived
    allowedLateness: 120, // a sale up to 2 minutes late still counts
  })
  .aggregate({ count: () => 1, sum: (sale) => sale.amount, max: (sale) => sale.amount })
  .to(queen.queue('sales-per-hour'))
  .run({ queryId: 'sales-per-hour', url: 'http://localhost:6632', subscriptionMode: 'all' })
```

`allowedLateness` says how far behind the newest event an event may be and still count. A later one is dropped, or with `onLate: 'include'` it opens a fresh window, so the window is sent a second time with only the late events. `gracePeriod` keeps a window open a little after its end, and `idleFlushMs` (5 seconds) closes the windows of partitions that went quiet. `subscriptionMode: 'all'` makes a new stream read what the queue already holds.

WHY *events arrive late and out of order: a phone that was offline, a batch upload, a retry. An hourly report should count a sale in the hour it happened, not in the hour its message got through.*

2.10.3 Shaping the data

- `map(fn)` changes each message, `filter(fn)` drops the ones you do not want, and `flatMap(fn)` turns one message into several values.
- `aggregate({ ... })` folds a window into named numbers. The name decides the fold: `count`, `sum`, `min`, `max` and `avg`; any other name adds up. `reduce(fn, initial)` is the general form, for anything else.
- `keyBy(fn)` groups by something other than the partition (per page instead of per user). Several workers can then write the same key, which slows them down; when that matters, push to a queue partitioned by the new key first, and stream from it.

WHY *most stream jobs are the same few steps: drop noise, reshape, group, fold. Having them as building blocks keeps a stream to a few readable lines.*

2.10.4 Two ways to end a stream

`.to(queue)` pushes each result to another queue, in the same commit as the state and the acks, into the partition of the same name unless you pass `{ partition }`. `.foreach(fn)` calls your function for each result, and acks only after it returns: a crash in the middle calls it again.

WHY *a result that stays inside Queen can be exactly once. A call to the outside world (an email, an API) cannot, so foreach makes it at least once and says so. For an external effect that must happen once, send the result to a queue and let a worker do it with an idempotency key.*

2.10.5 Rate limits with a gate

`gate(fn)` decides, message by message, whether a message may go on now, with state kept per partition. Two ready-made gates are included: `tokenBucketGate` and `slidingWindowGate`.

```
import { tokenBucketGate } from 'queen-mq'

const limiter = await Stream
  .from(queen.queue('outgoing-sms'))           // one partition per customer
  .gate(tokenBucketGate({ capacity: 10, refillPerSec: 1 })) // 10 at once, then 1 a second
  .foreach(async (sms) => sendSms(sms))
  .run({ queryId: 'sms-limiter', url: 'http://localhost:6632' })
```

A message the gate refuses stops its partition there: it and the ones after it are delivered again, in order, when the lease runs out.

WHY *an SMS provider, a partner API or a rate-limited email service accepts only so many calls a second. A gate per customer keeps you inside those limits without losing order and without a queue of your own for the waiting messages.*

2.10.6 Running a stream

`run()` takes the stream's `queryId`, its identity: its state is stored under it, so a process restarted with the same id carries on with the same windows, and several processes with the same id share the partitions between them. `batchSize` (200) and `maxPartitions` (4) size each cycle, and `stop()` on the handle `run()` returns stops it cleanly.

WHY *a stream is a long-running worker like any other: it has to survive restarts, scale out, and stop during a deploy without losing its place.*

On the site, with the code in all six SDK languages: queenmq.com/writing-applications/streams.

Push, consume, ack, replay, fetch, transactions, KV, timers, locks, streams and ephemeral queues all combine. Build on them and Queen carries the logic you would otherwise write around a broker.

3

Real applications examples

The common messaging patterns, one page each, with a real case for each one: the problem, the pattern that solves it with Queen, and the code. Every result quoted here comes from one program that checks it, `examples/tutorials/js/book-use-cases.mjs`.

3.1 Work queue

Many workers share one stream of jobs, and each job is done by exactly one of them: resizing photos, sending emails, rendering invoices. This is the work queue, also called competing consumers.

In Queen it is a consumer with no consumer group. Each message goes to one worker. Work runs in parallel across partitions: a partition is held by one worker at a time, so here every user's photos are a partition, and three workers share four users.

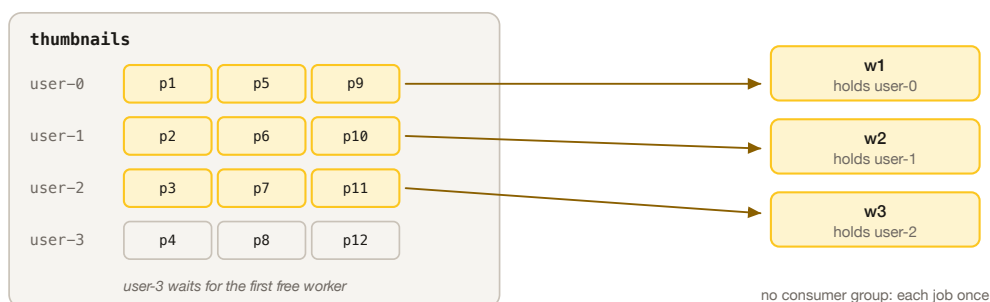


Figure 3.1: *Competing consumers: each job once, one user's partition per worker at a time.*

```
// Twelve photos to resize, from four users.
for (let i = 1; i <= 12; i++) {
  await queen.queue('thumbnails').partition(`user-${i % 4}`)
    .push([ { data: { photo: `p${i}.jpg` } } ])
}

// Three workers compete for the jobs: each job goes to exactly one of them.
const worker = (name) => queen.queue('thumbnails')
  .partitions(1) // one user's photos at a time
  .each()
  .idleMillis(1500) // stop when there is nothing left
  .consume(async (job) => resize(job.data.photo, name))

await Promise.all([worker('w1'), worker('w2'), worker('w3')])
```

All twelve photos were resized once each, shared out as 6, 3 and 3 between the three workers.

WHY *order still holds inside each user's photos, and a slow user holds up only its own partition. To go faster, add workers, up to the number of partitions that have work.*

One thing to know: with a single partition the workers would take turns, one at a time. Give the jobs a partition per user, per customer or per tenant.

3.2 Publish/subscribe and fan-out

One event, many readers: when an order is placed, billing charges it, shipping prepares it, and analytics counts it. Each reader must get every event, at its own pace, and a reader added later must be able to start from the beginning.

Each reader is a consumer group. A group gets every message of the queue and keeps its own position; the workers inside a group share its work, as in a work queue.

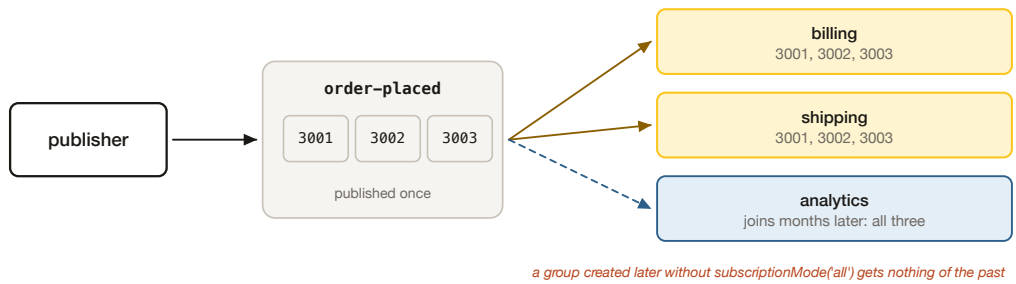


Figure 3.2: Published once, read by every group, also by one that joins later.

```
const subscribe = (group) => queen.queue('order-placed')
  .group(group) // each group gets every message
  .subscriptionMode('all')
  .each()
  .consume(async (event) => {
    console.log(group, event.data.orderId)
  })

for (const orderId of [3001, 3002, 3003]) {
  await queen.queue('order-placed').partition(`order-${orderId}`)
    .push([{ data: { orderId } }])
}
await Promise.all([subscribe('billing'), subscribe('shipping')])

// Months later, analytics subscribes too, and reads every order ever placed.
await subscribe('analytics')
```

Billing and shipping each got the three orders. Analytics, started later, read all three from the start.

WHY the queue is a log: reading does not delete. So a new subscriber costs nothing to the publisher, and a group that falls behind slows down only itself.

One thing to know: a new group starts at the end of the queue unless it asks for `subscriptionMode('all')`, or a point in time with `subscriptionFrom`. Our check proved it: a group created without it found nothing.

3.3 Request and reply

The booking page needs a price from the pricing service, now. If no answer comes in half a second, it shows “ask us”. A durable queue is too heavy for that, and a direct HTTP call ties the two services together.

Ephemeral queues live in memory, outside the log. The caller makes a private inbox, sends it as `replyTo`, and waits for the answer with a timeout. Any instance of the pricing service can answer.

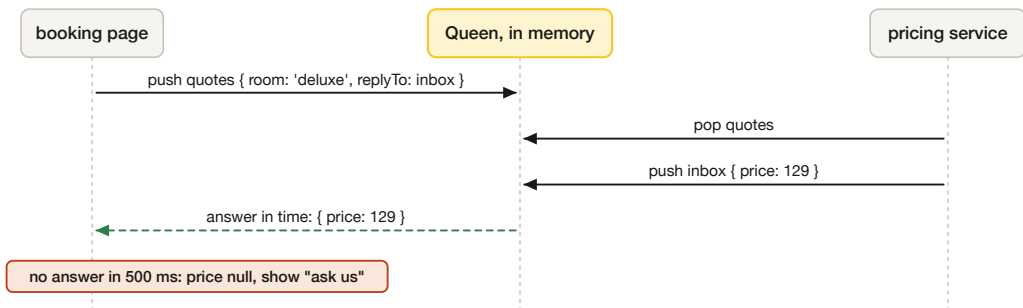


Figure 3.3: *A quote with a deadline: the answer, or “ask us”.*

```

// The pricing service answers quote requests, in memory.
async function pricing() {
  const { messages } = await queen.ephemeral.pop('quotes', { group: 'pricing', wait: true })
  for (const m of messages) {
    await queen.ephemeral.push(m.payload.replyTo, [{ room: m.payload.room, price: 129 }])
  }
  await queen.ephemeral.ack('quotes', messages, { group: 'pricing' })
}

// The booking page asks, and waits at most half a second for the answer.
async function quote(room) {
  const inbox = `inbox-${crypto.randomUUID()}`
  await queen.ephemeral.push('quotes', [{ room, replyTo: inbox }])
  const reply = await queen.ephemeral.pop(inbox, {
    wait: true, timeout: 500, commitOnDelivery: true })
  return reply.messages[0]?.payload ?? { room, price: null } // too slow: show "ask us"
}

const [deluxe] = await Promise.all([quote('deluxe'), pricing()])
console.log(deluxe) // { room: 'deluxe', price: 129 }

```

The answer came back in time: {room: 'deluxe', price: 129}. With no pricing service running, the caller gave up after half a second and got price: null.

WHY *an answer that comes too late is worth nothing, so it does not need a disk. In memory a round trip costs very little, and the service scales like any consumer group.*

One thing to know: a node that crashes loses the ephemeral messages it held. There are no transactions, dead letters or replays on these queues.

3.4 Ordering per key, and batching

A chat: every conversation must be handled in order, and a slow one must not delay the others. And a guest rarely sends one message. They send three in a row: “Hi!”, “Is breakfast included?”, “And is there parking?”. An assistant that answers each message at once answers three times.

Every conversation is a partition of its own, created by its first message, so its messages stay in order and it waits on nobody else. `windowBuffer` adds the batching: new messages of a conversation wait until the first of them is two seconds old, and then go to the handler together.

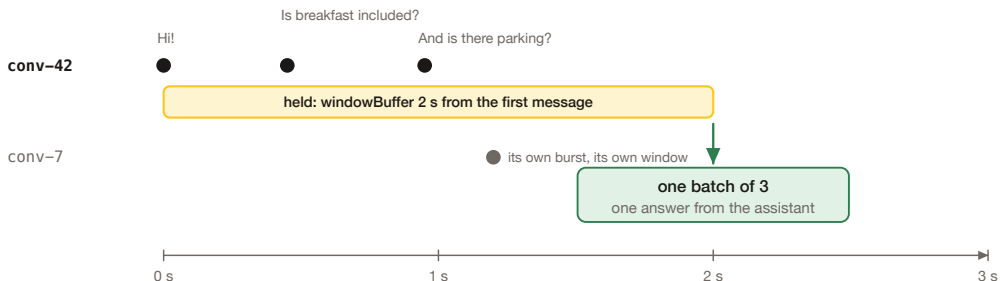


Figure 3.4: *A conversation’s burst waits two seconds, then goes as one batch.*

```

// A burst waits 2 seconds, then goes to the handler as one batch.
await queen.queue('chat').config({ windowBuffer: 2 }).create()

// The guest sends three messages in a row.
for (const text of ['Hi!', 'Is breakfast included?', 'And is there parking?']) {
  await queen.queue('chat').partition('conv-42').push([{ data: { text } }])
}

// The assistant answers the whole burst once.
await queen.queue('chat')
  .group('assistant')
  .subscriptionMode('all')
  .batch(10) // the handler gets every message of the burst
  .consume(async (messages) => {
    question = messages.map((m) => m.data.text).join(' ')
    console.log(question) // Hi! Is breakfast included? And is there parking?
  })

```

The handler ran once, two seconds after the first message, with the three texts.

WHY *a conversation comes in bursts, not one message at a time. Waiting a moment gives the assistant the whole question. Each conversation waits only for its own burst, never for the others.*

One thing to know: the window ends two seconds after the oldest waiting message, even if the guest keeps typing, so a long burst is answered in parts, every two seconds.

3.5 Workflows: one step, one commit

An order moves through states: paid, shipping, shipped. A redelivered event, two workers, or an event that arrives late must never move it twice or backwards, and a step must never be half done.

The order's state lives in KV. Each step is one transaction: it acks the event, writes the new state only if the state is still the version the worker read (expect), and pushes the next event. If the version moved, the whole transaction rolls back, and nothing is written.

```

await queen.queue('order-events')
  .group('workflow')
  .subscriptionMode('all')
  .autoAck(false)
  .each()

```



Figure 3.5: A state machine whose steps cannot race.

```
.consume(async (msg) => {
  const { orderId } = msg.data
  const order = await queen.kv.get('orders', orderId) // { status: 'paid' }
  if (order.value.status !== 'paid') { // already moved: just ack
    return queen.ack(msg, 'completed', { group: 'workflow' })
  }

  // Do the work of this step here: charge the card, book the courier, ...

  const result = await queen.transaction()
  .ack(msg, 'completed', { consumerGroup: 'workflow' })
  .kv.put('orders', orderId, { status: 'shipping' },
    { ttl: '30d', expect: order.version, required: true }) // only from what we read
  .queue('shipping').partition(orderId)
  .push([[{ transactionId: `ship-${orderId}`, data: { orderId } }]])
  .commit()
  if (result.success === false) return // moved by someone else: nothing written
})
```

The step moved order-2001 to shipping and pushed one shipping event. The same step sent again from the old version rolled back: still one event.

WHY the ack, the new state and the next event commit together or not at all. That replaces the outbox table, the idempotency table and the lock you would put around them.

One thing to know: `commit()` returns `success: false` for a rollback, it does not throw. Read it.

3.6 Windowed aggregation

Tracing: an SDK sends one event for each step of a request (the HTTP call, the database query, the cache lookup), and retries when it gets no answer. A trace is finished when no more events arrive, and what you want to keep is one record per trace. The same shape counts visits, sessions and anything else that ends when it goes quiet.

Each trace is a partition, and each event's `transactionId` is the trace and the step, so a retried event is stored once. A stream with a session window closes a trace after two seconds of quiet, adds it up, and pushes one message per finished trace.

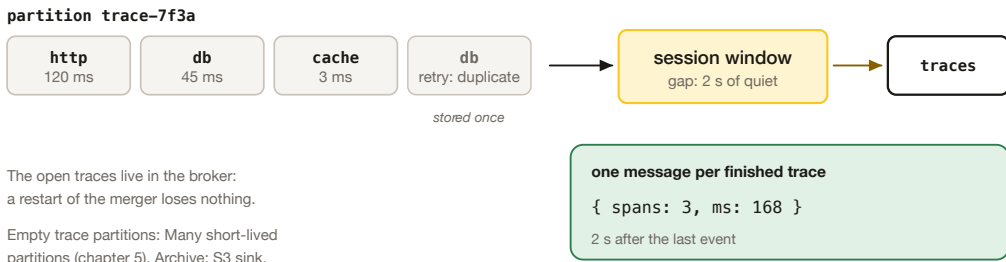


Figure 3.6: Events merged into one record per trace by a session window.

```
// A trace is complete when it has been quiet for 2 seconds.
const merger = await Stream
  .from(queen.queue('trace-events'))
  .windowSession({ gap: 2, idleFlushMs: 500 })
  .reduce((t, e) => ({ spans: t.spans + 1, ms: t.ms + e.ms }), { spans: 0, ms: 0 })
  .to(queen.queue('traces')) // one message per finished trace
  .run({ queryId: 'trace-merger', url, subscriptionMode: 'all' })

// Every SDK call is one event, in the partition of its trace.
const event = (span, ms) => ({ transactionId: `trace-7f3a:${span}`, data: { span, ms } })
await queen.queue('trace-events').partition('trace-7f3a').push([
  event('http', 120), event('db', 45), event('cache', 3),
])
await queen.queue('trace-events').partition('trace-7f3a')
  .push([event('db', 45)]) // a retried call: stored once
```

Two seconds after the last event, the traces queue held one message: 3 spans and 168 ms. The retried database event was counted once.

WHY *every trace is its own ordered stream, so “this partition went quiet” is exactly “this trace is finished”. The open traces live in the broker, so a restart of the merger loses nothing.*

With many traces, delete the empty trace partitions quickly (Many short-lived partitions, in chapter 5), and copy the finished traces to a bucket (the S3 sink, in chapter 4). One thing to know: a trace that pauses longer than the gap becomes two records, and the reader merges them.

3.7 Rate limiting

An SMS provider accepts five messages a second for each customer. Send more and it refuses them, or blocks the account. The messages must still all go out, in order.

A stream with a gate decides, message by message, whether a message may go on now. The token bucket gate keeps its state per partition, in the broker: here five tokens per customer, refilled at five a second. A message the gate refuses stops its partition there, and comes back, with the ones after it, when its lease ends.

```
// The SMS provider accepts 5 messages a second for each customer.
// A refused message comes back when its lease ends: here after 1 second.
await queen.queue('sms-out').config({ leaseTime: 1 }).create()

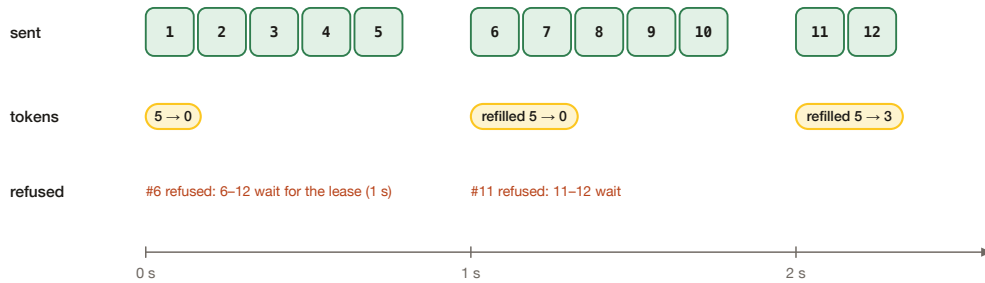
const limiter = await Stream
  .from(queen.queue('sms-out')) // one partition per customer
  .gate(tokenBucketGate({ capacity: 5, refillPerSec: 5 })) // 5 at once, then 5 a second
  .foreach(async (sms) => sendSms(sms)) // only what the gate lets through
  .run({ queryId: 'sms-gate', url, subscriptionMode: 'all' })

// Twelve messages for one customer, all at once.
await queen.queue('sms-out').partition('customer-42')
  .push(Array.from({ length: 12 }, (_, i) => ({ data: { n: i + 1, text: `#${i + 1}` } })))
```

Five messages went at once, and the rest followed at about five a second, so the twelfth left about two seconds after the first: all twelve, in order.

WHY *the bucket lives in the broker, per customer, and survives restarts: no Redis next to it. Waiting messages stay in their queue, in order, and never need a queue of your own.*

One thing to know: the refused messages wait for their lease, so size the two together. The most a gate can let through is the smaller of `refillPerSec` and `capacity` divided by the lease in seconds, so keep `capacity` near `refillPerSec` times the lease.



The most a gate lets through: $\min(\text{refillPerSec}, \text{capacity} \div \text{lease in seconds})$.

Figure 3.7: The token bucket gate: 5, 5, then 2, in order. A refill a few milliseconds short lets 4 through, not 5.

3.8 Last value

The availability of a hotel changes many times a minute. A screen that shows it needs only the current value, and handling every old update on the way there is wasted work.

A consumer group with `.conflation()` gets only the newest message of each partition, and its cursor moves past the older ones.

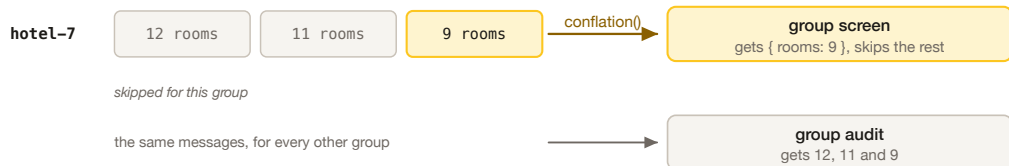


Figure 3.8: Conflation: the newest value of each partition, for this group only.

```
// The availability of a hotel changes often; the screen needs only the last value.
for (const rooms of [12, 11, 9]) {
  await queen.queue('availability').partition('hotel-7')
    .push([ { data: { hotel: 7, rooms } } ])
}

const [latest] = await queen.queue('availability')
  .group('screen')
  .subscriptionMode('all')
  .conflation() // the newest message of each hotel, the rest skipped
  .wait(false)
  .pop()
console.log(latest.data) // { hotel: 7, rooms: 9 }
await queen.ack(latest, 'completed', { group: 'screen' })
```

Three updates were pushed (12, 11 and 9 rooms). The screen got 9, and the older two were never delivered to it.

WHY *for a value that changes, only the last one matters. Conflation lets a slow consumer catch up in one step instead of replaying every update. Other groups of the same queue still get every message.*

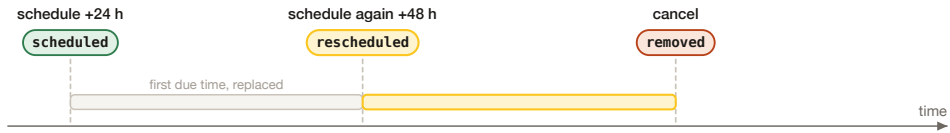
One thing to know: use it only where an old message has no meaning of its own. For events that each count, such as payments, every message must be handled.

3.9 Delayed and scheduled messages

A day after an order is delivered, ask for a review. If the delivery is late, move the reminder. If the guest already wrote a review, cancel it. The usual answer is a cron job that scans a table every minute.

A timer is a message that waits. It is named by its queue and a key: scheduling the same key again moves it, and cancelling the key removes it. When it is due, it arrives in its queue as a normal message, through the log.

review-reminders / review-2001



Not cancelled, a timer fires as a normal message of its queue, through the log.

A cancel can arrive just after it fired: check the order's state before sending.

Figure 3.9: A keyed timer: schedule, move, cancel.

```
// When an order is delivered, ask for a review in 24 hours.
await queen.timer('review-reminders').key('review-2001').delay('24h')
  .payload({ orderId: 2001 }).schedule()           // status: 'scheduled'

// The delivery was late: schedule the same key again, and the reminder moves.
await queen.timer('review-reminders').key('review-2001').delay('48h')
  .payload({ orderId: 2001 }).schedule()           // status: 'rescheduled'

// The guest wrote a review already: no reminder at all.
await queen.timer('review-reminders').key('review-2001').cancel()
```

The first call answered scheduled and the second rescheduled; after the cancel, no reminder was left. A short timer of the same queue arrived as a message, with its payload as data.

WHY the key makes a reminder easy to move or cancel, and a retried schedule is harmless. Put the timer in the transaction of the step that creates it (chapter 2), and it exists exactly when the step committed.

One thing to know: a cancel can arrive just after the timer fired. The consumer of the reminder should check the order's state before it sends anything.

3.10 Retries and dead letters

When a booking is confirmed, send `booking.confirmed` to the customer's URL. Their server fails sometimes, and a crash at the wrong moment can make you call it twice.

One message per call, in a queue with retries (`retryLimit: 5` and a `retryDelay`). The handler throws when the call fails: the message comes back after the delay, and

goes to the dead-letter queue after the last try. The message's `transactionId` travels as the `Idempotency-Key`, so the receiver can drop a second call.

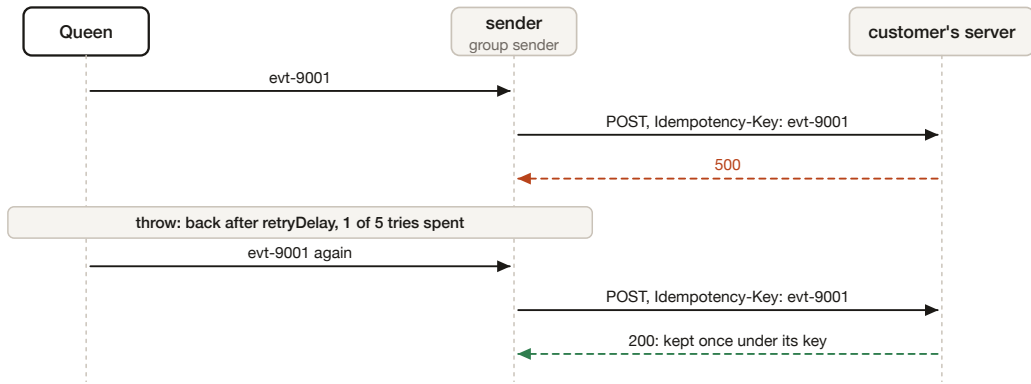


Figure 3.10: A failed call comes back after the delay; the key makes it count once.

```

const stop = new AbortController()
process.once('SIGTERM', () => stop.abort()) // a deploy stops the sender cleanly

await queen.queue('webhooks')
  .group('sender')
  .subscriptionMode('all')
  .each()
  .consume(async (msg) => {
    const res = await fetch(msg.data.url, {
      method: 'POST',
      headers: { 'content-type': 'application/json', 'idempotency-key': msg.transactionId },
      body: JSON.stringify(msg.data.event),
    })
    if (!res.ok) throw new Error(`HTTP ${res.status}`) // retried, then dead-lettered
  }, { signal: stop.signal })
  
```

The receiver failed the first call. The message came back after the delay, the second call worked, and the receiver kept the event once, under `evt-9001`.

WHY the call happens outside any commit, so it can happen twice: a crash after the call and before the ack sends it again. The receiver dropping the second call by its key is what makes it count once. Queen gives the stable key, the retries and the dead-letter queue.

One thing to know: a failed message holds its partition for the delay. Use one partition per customer, so a customer whose server is down waits on its own retries alone.

3.11 One worker at a time

The nightly report must be built once, but every replica of the service runs the same scheduler. This is leader election, or a singleton job.

A lock with a lifetime: the worker that gets it builds the report, and the others skip. The report is committed with `.guard(lock)`: the transaction commits only while the lock is still this worker's. The day in the `transactionId` is a second fence.

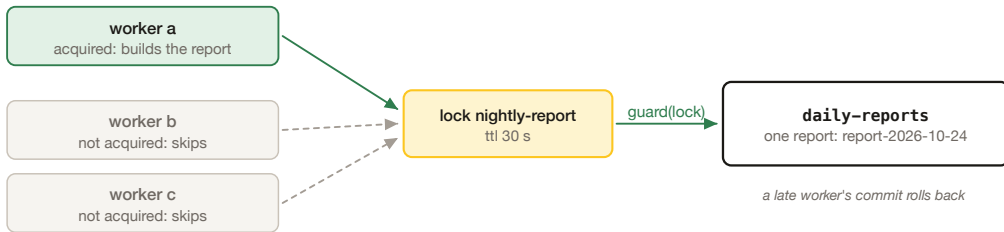


Figure 3.11: One worker builds the report; the guard keeps a late one out.

```

const lock = queen.lock('nightly-report', { ttl: '30s' })
if (!await lock.acquire({ wait: '1s' })) return 'skipped' // another worker has it
try {
  const report = { day: '2026-10-24', bookings: 412 }
  const result = await queen.transaction()
    .guard(lock) // commits only while the lock is ours
    .queue('daily-reports').push([{ transactionId: `report-${report.day}`, data: report }])
    .commit()
  if (result.success === false) return 'lost the lock'
  return 'done'
} finally {
  await lock.release()
}
  
```

Three workers started at once: one built the report and two skipped. There was one report.

WHY a lock expires without telling its holder. A worker paused past the lifetime keeps running, while another one takes the lock. The guard keeps the late worker's work out: its commit rolls back.

One thing to know: the lock renews itself in the background while the worker holds it. For long work, pass `lock.signal` to what the work waits on, so it stops when the

lock is lost. For “at most three at a time”, `queen.semaphore(name, 3)` works the same way.

4

Integrations

Queen talks to the systems around it from inside the broker: Kafka clients through a Kafka port, S3 buckets through a sink, PostgreSQL tables through a source and a sink. There is no extra process to run. Each one is a few settings or one HTTP call, and each one runs on every node of a cluster. The last section is on the application side: Laravel, whose workers the QueenMQ supervisor runs.

4.1 How to use with Kafka clients

Kafka producers, consumers and admin tools can use Queen as they are: change bootstrap.servers, and nothing else. A topic is a Queen queue and a Kafka consumer group is a Queen consumer group, so Kafka clients and Queen clients read and write the same data.

WHY *moving away from Kafka should not mean rewriting every service on the same day. With the Kafka port, you move one service at a time, and the services not moved yet keep working on the same topics.*

4.1.1 Turn it on

```
docker run -d --name queen -p 6632:6632 -p 127.0.0.1:9092:9092 \
-v /var/lib/queen:/var/lib/queen/raft \
-e QUEEN_KAFKA_EMBEDDED=true \
-e QUEEN_KAFKA_ADVERTISED_ADDR=127.0.0.1:9092 \
ghcr.io/queen-mq/queen:2.6.0
```

QUEEN_KAFKA_ADVERTISED_ADDR is the address Kafka clients come back to after their first request, so they must be able to reach it. It has no default: without it the node

refuses to start. Send three keyed records with kcat, and read them back:

```
kcat -P -b 127.0.0.1:9092 -t orders -K: <<'EOF'
customer-123:{"orderId":8891,"total":42}
customer-456:{"orderId":8892,"total":17}
customer-123:{"orderId":8893,"total":5}
EOF
kcat -C -b 127.0.0.1:9092 -t orders -o beginning -e -q -f '%t [%p] offset %o key=%k %s\n'
orders [307] offset 0 key=customer-123 {"orderId":8891,"total":42}
orders [307] offset 1 key=customer-123 {"orderId":8893,"total":5}
orders [400] offset 0 key=customer-456 {"orderId":8892,"total":17}
```

Nobody created orders: kcat's first request did, 1,024 partitions wide. kcat hashed each key onto a partition, and both orders of customer-123 landed in partition 307, in order. The HTTP API reads the same records. A Kafka key and value are bytes, so each message carries them in base64 (k, v), with the producer's timestamp (t):

```
curl -s 'localhost:6632/api/v1/pop/queue/orders/partition/307?batch=10' \
  | jq -c '.messages[].data'
{"k":"Y3VzdG9tZXItMTIz","t":1792837397814,"v":"eyJvcnRlcklkIjo4ODkxLCJ0b3RhbCI6NDJ9"}
{"k":"Y3VzdG9tZXItMTIz","t":1792837397814,"v":"eyJvcnRlcklkIjo4ODkzLCJ0b3RhbCI6NX0="}
```

4.1.2 One consumer group for both

Run a Kafka group consumer, and stop it once it has printed the three records:

```
kcat -b 127.0.0.1:9092 -G billing -o beginning -q -f 'billing [%p] offset %o key=%k\n' orders
billing [400] offset 0 key=customer-456
billing [307] offset 0 key=customer-123
billing [307] offset 1 key=customer-123
```

billing is a Queen consumer group now, with its lag in the dashboard (/api/v1/consumer-groups lists it with "kind":"kafka"). A Queen consumer of billing starts

where the Kafka consumer stopped:

```
curl -s -X POST localhost:6632/api/v1/push -H 'content-type: application/json' -d '{"items":[
  {"queue":"orders","partition":"307","payload":{"orderId":8894,"total":99}}]}' >/dev/null
Q=localhost:6632/api/v1/pop/queue/orders/partition/307
curl -s "$Q?consumerGroup=billing&batch=10" | jq -c '.messages[] | {offset, data}'
{"offset":2,"data":{"orderId":8894,"total":99}}
```

And a Kafka consumer reads what a Queen client pushed: a record with no key, whose value is the JSON.

4.1.3 How Kafka maps onto Queen

Kafka	Queen
Topic	A queue of the same name, created by CreateTopics or by the first request that names it
Partition n	The partition named "n"
Partition count	What CreateTopics asked for, else 1,024. A topic never gets narrower
Offset, committed offset	The partition's offset; the group's position on that partition
acks=1, acks=all	One path: answered when the push is committed in the raft log
Replication factor	The raft voters: every voter holds every partition
Idempotent producer	Queen's dedup: producer id, epoch and sequence are the transactionId
Transaction	Kept in the leader's memory, committed with its offsets as one raft entry
retention.ms	The queue's retention, set when the topic is created
cleanup.policy=compact	Retention off: every record is kept

On a cluster every node runs the Kafka port, and together they are one Kafka cluster. The raft leader leads every partition and coordinates every group; a client sent to another node is told to go to the leader. Give each node its own QUEEN_KAFKA_ADVERTISED_ADDR, and never put the nodes behind one load-balanced address: a client sent to the leader would land on another node, and loop. A load balancer is fine in bootstrap.servers, which a client uses once.

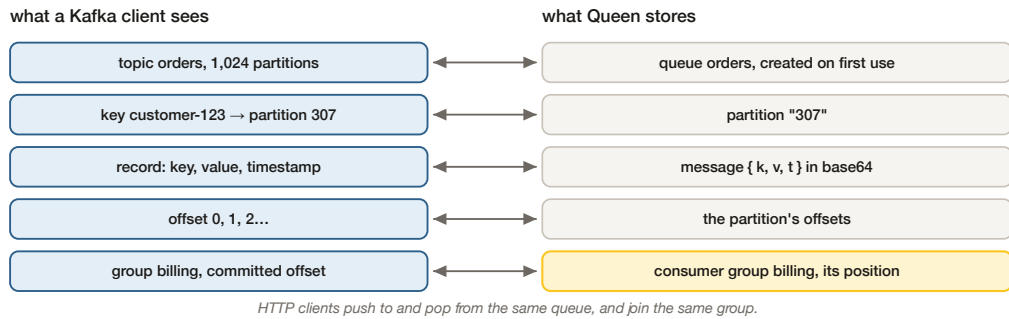
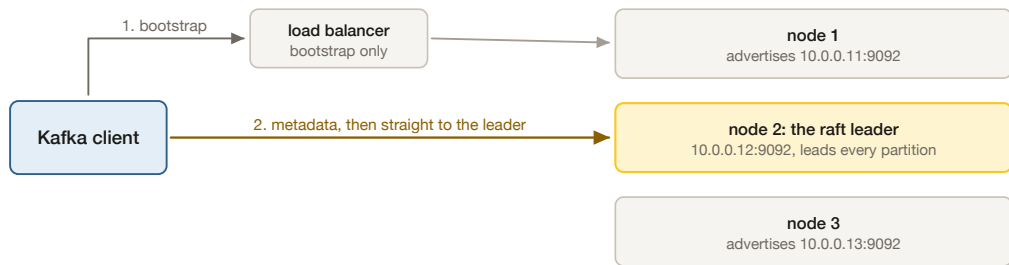


Figure 4.1: What a Kafka client sees, and what Queen stores.



Never advertise the load balancer: a client sent to the leader would land elsewhere, and loop.

Figure 4.2: One Kafka cluster: bootstrap once, then straight to the raft leader.

4.1.4 Five things to know

- **Kafka consumers take no lease.** Queen's retries and dead-letter queue do not apply to them: a Kafka consumer handles its own failures, as on Kafka. Kafka and Queen consumers share a group's position, not its work, so hand a group from one kind to the other, and do not run both at once.
- **Committed offsets never expire.** Kafka forgets an empty group after seven days; here a group keeps its position until you delete it.
- **Transactions live in the leader's memory,** up to 8 MiB, 50,000 records and 1,000 partitions each. A leader change aborts the transactions that were open.
- **The Kafka port has no TLS in 2.6,** and SASL passwords travel in clear: keep it on a private network.
- **Advertise an address the client can reach, as it will try it.** On macOS, localhost is tried as ::1 first, so a port published on 127.0.0.1 only fails with localhost:

use 127.0.0.1.

On the site: queenmq.com/integrations/kafka. More there: [Kafka Streams](#), [Connect](#) and [Flink](#).

4.2 S3 Sink

The broker can copy its queues into an S3 bucket, as JSONL or Parquet files in folders that DuckDB, Spark or ClickHouse read directly. The sink runs inside the broker, on every node of a cluster, and it is exactly once: each record lands in one file, and a crash or a retried upload rewrites the same bytes under the same name instead of adding a copy.

WHY *a queue is a log, and its history is worth keeping for analytics, audits and replays. Disk on the brokers is expensive for that; a bucket is cheap, and the tools that read it are already there. The sink needs no consumer to run and no code to write.*

4.2.1 Turn it on

Any S3-compatible service works. Our test used a versitygw gateway named s3 on the same Docker network:

```
docker run -d --name queen --network lake -p 6632:6632 \
  -v /var/lib/queen:/var/lib/queen/raft \
  -e QUEEN_S3_EMBEDDED=true \
  -e QUEEN_S3_QUEUES=orders \
  -e QUEEN_S3_ENDPOINT=http://s3:7070 \
  -e QUEEN_S3_REGION=us-east-1 \
  -e QUEEN_S3_PATH_STYLE=true \
  -e QUEEN_S3_BUCKET=lake \
  -e QUEEN_S3_ACCESS_KEY=queen \
  -e QUEEN_S3_SECRET_KEY=queen-secret-key \
  -e QUEEN_S3_START=earliest \
  -e QUEEN_S3_MAX_WINDOW_MS=10000 \
  ghcr.io/queen-mq/queen:2.6.0
```

The last two lines are for the demo: `earliest` copies the queue from its first record, and a file closes after ten seconds instead of five minutes. Push three messages:

```
curl -s -X POST localhost:6632/api/v1/push -H 'content-type: application/json' -d '{"items":[
  {"queue":"orders","partition":"cust-1","payload":{"order":1,"status":"new","total":42.00}},
  {"queue":"orders","partition":"cust-2","payload":{"order":2,"status":"new","total":17.50}},
  {"queue":"orders","partition":"cust-1","payload":{"order":1,"status":"paid","total":42.00}}
]}' >/dev/null
curl -s localhost:6632/status \
  | jq -c '.s3.sinks[0].queues[0] | {state, k, windowsCommitted, records}'
{"state":"filling","k":1,"windowsCommitted":1,"records":3}
```

About ten seconds later window 1 was in the bucket, as two files: the data and its manifest.

```
queen/tenant=00000000-0000-0000-0000-000000000001/queue=orders/dt=2026-10-24/hour=10/
w-00000000001-1792836000000000-1792837437745538.jsonl.zst
queen/_queen/tenant=00000000-0000-0000-0000-000000000001/queue=orders/windows/0000000001.json
```

The data file holds one line per record, sorted by partition and offset:

```
{"partition":"cust-1","offset":0,"transactionId":"01a16d70-3e1e-7001-9fb4-299167829f1d",
  "ts":"2026-10-24T10:23:54.910988Z","payload":{"order":1,"status":"new","total":42.00}}
{"partition":"cust-1","offset":1,...,"payload":{"order":1,"status":"paid","total":42.00}}
{"partition":"cust-2","offset":0,...,"payload":{"order":2,"status":"new","total":17.50}}
```

The payload is the producer's own text, byte for byte: `42.00` is still `42.00`.

4.2.2 How it works

The sink cuts each queue into windows of time, on the broker's clock. A window closes when it holds `QUEEN_S3_TARGET_MB` of records, when it is `QUEEN_S3_MAX_WINDOW_MS` old, or at the next hour. Once a window is closed, no record can join it, so building it again gives the same records in the same order, the same bytes and the same file name. That is the whole exactly-once argument, and it asks nothing special of the bucket.

Every node runs the sink, but only one node writes a queue: the one that holds the queue's lease, a row in Queen's KV. A window is committed in three steps: its end is written in KV, the files go up, then a commit pointer moves in KV. A node that dies

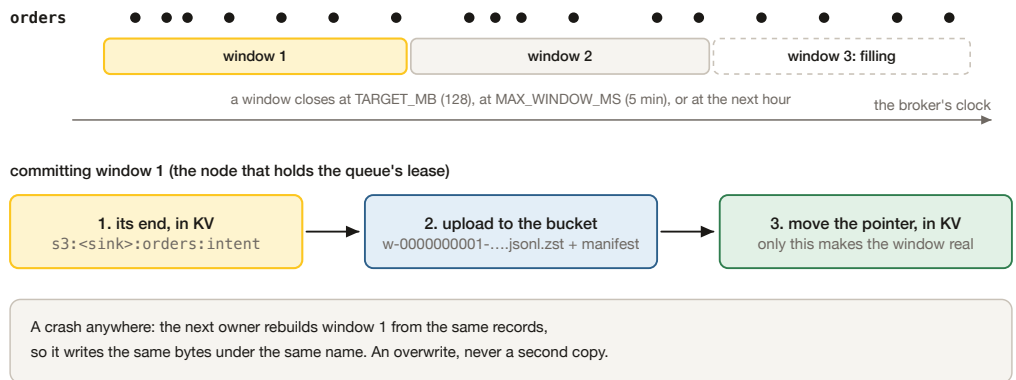


Figure 4.3: *Windows on the broker's clock, and the three-step commit.*

halfway leaves the next one enough to finish the same window. The queues spread over the nodes by themselves.

Setting	Default	What it does
QUEEN_S3_EMBEDDED	false	Starts the sink on this node.
QUEEN_S3_QUEUES	required	The queues to copy, comma-separated, or * for all.
QUEEN_S3_ENDPOINT, _ REGION, _BUCKET	required	Where the bucket is.
QUEEN_S3_ACCESS_KEY, _ SECRET_KEY	required	The credentials. The secret is never logged.
QUEEN_S3_PATH_STYLE	false	true puts the bucket in the URL path, as most self-hosted services want.
QUEEN_S3_FORMAT	jsonl	jsonl or parquet.
QUEEN_S3_START	latest	Where a new queue starts: latest, or earliest for what retention still holds.
QUEEN_S3_TARGET_MB	128	Uncompressed megabytes that close a window.
QUEEN_S3_MAX_WINDOW_MS	300000	The longest a window stays open.
QUEEN_S3_SSE	unset	AES256 or aws:kms: encryption of the files in the bucket.

4.2.3 Four things to know

- **The bucket copies the log, not what consumers did.** A message a group nacked or sent to the dead-letter queue is in the bucket like any other.
- **The bucket is plain text.** The sink decrypts encrypted queues before it writes them, so give the bucket encryption of its own (QUEEN_S3_SSE, or the bucket's default).
- **Retention can delete records before the sink reads them.** The loss is never silent (the manifest lists the lost range), but you can rule it out: with "retention-SinkHold":"default" on the queue, retention waits for the sink, up to seven days.
- **Watch queen_s3_lag_seconds:** how far behind the bucket is. It climbs to a little more than one window on a busy queue, then drops when the window commits.

On the site: queenmq.com/integrations/s3.

4.3 Postgres Source and Sink

The broker can stream a PostgreSQL table into a queue, and a queue into a table, with no other process. The **source** reads committed changes through logical replication and turns each one into a message, in a partition named after the row's key, in commit order. The **sink** is a consumer group the broker runs for you, and it writes every message into a table. Each one is set up with one HTTP call, runs on every node of a cluster, and gives exactly-once effects.

WHY *getting changes out of a database usually means Debezium, a Kafka Connect cluster and Kafka. Here it is one call on the broker you already run. And the sink replaces the consumer you would write to keep a table up to date with a queue.*

4.3.1 The source

The broker needs QUEEN_ENCRYPTION_KEY, because it stores the database password sealed with that key. PostgreSQL needs version 17 or later and wal_level=logical.

Take a table with two rows:

```
docker exec -i pg psql -U postgres <<'SQL'
CREATE TABLE orders (
  id          bigint PRIMARY KEY,
  customer    jsonb NOT NULL,
  status      text NOT NULL,
  total       numeric(12,2) NOT NULL,
  created_at  timestamptz NOT NULL DEFAULT now()
);
INSERT INTO orders (id, customer, status, total) VALUES
(1, '{"name": "Ada", "tier": "gold"}', 'new', 42.00),
(2, '{"name": "Linus", "tier": "silver"}', 'new', 17.50);
SQL
```

Create the source:

```
curl -s -X PUT localhost:6632/api/v1/connectors/orders-src \
-H 'content-type: application/json' -d '{
  "kind": "source",
  "connection": {"host": "pg", "database": "postgres", "user": "postgres",
    "password": "secret", "sslMode": "disable"},
  "source": {"tables": [{"table": "public.orders", "queue": "orders"}]}
}' >/dev/null
curl -s localhost:6632/api/v1/connectors/orders-src \
| jq -c '.status | {phase, pointerLsn, messages}'
{"phase":"streaming","pointerLsn":"0/17F85F0","messages":2}
```

Within a few seconds the source made a publication and a replication slot, copied the two rows already there, and started streaming. Each row is a message in the partition named after its key:

```
Q='batch=10&partitions=10&consumerGroup=reader&subscriptionMode=all'
curl -s "localhost:6632/api/v1/pop/queue/orders?$Q" | jq -c '.messages[0].data'
{"op":"r","table":"public.orders","key":{"id":1},
  "after":{"id":1,"customer":{"name":"Ada","tier":"gold"},"status":"new","total":42.00,
    "created_at":"2026-10-24T10:24:22.156265+00:00"},
  "before":null,"lsn":"0/17F85F0","ts":"2026-10-24T10:24:22.823535Z","seq":0}
```

Field	Example	What it is
op	r	c insert, u update, d delete, r a row copied at the start.
key, after, before		The row's key, the new row (null for a delete), the old key for a delete.
lsn, xid	0/17F85F0	The commit position in the WAL, and the PostgreSQL transaction.
ts, seq		The commit time, and the change's place inside its transaction.

Then we changed the table: one update, then an insert and a delete in one transaction. The insert and the delete arrived with the same `lsn` and `xid`, in one Queen transaction, so no consumer could see one without the other. Each message id is built from the change's place in the WAL (`pg:0a238c4b:000000000017F8830:0`), so it is the same however many times it is sent.

WHY *the source writes the messages and its WAL position in the same Queen transaction. After a crash the next owner restarts from that position and drops what is already in Queen. Only after Queen has committed does PostgreSQL get leave to recycle the WAL.*

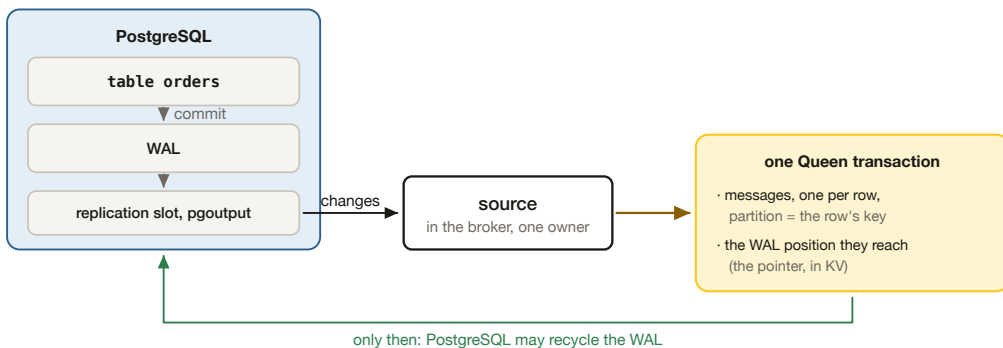


Figure 4.4: *The source: the messages and the WAL position commit together.*

4.3.2 The sink

A sink in cdc mode applies the source's events to another table:

```
docker exec pg psql -U postgres -c 'CREATE TABLE orders_copy (LIKE orders INCLUDING ALL);'
curl -s -X PUT localhost:6632/api/v1/connectors/orders-copy \
-H 'content-type: application/json' -d '{
  "kind": "sink",
  "connection": {"host": "pg", "database": "postgres", "user": "postgres",
                 "password": "secret", "sslMode": "disable"},
  "sink": {"queue": "orders", "table": "public.orders_copy", "mode": "cdc"}
}' >/dev/null
docker exec pg psql -U postgres \
-c 'SELECT id, customer, status, total FROM orders_copy ORDER BY id;'
id |          customer          | status | total
---+-----+-----+-----
 1 | {"name": "Ada", "tier": "gold"} | paid   | 42.00
 3 | {"name": "Grace", "tier": "gold"} | new    | 99.90
```

A sink also writes plain messages. In upsert mode each top-level key of the message goes to the column of the same name, keys with no column are ignored, and a column the message does not name keeps its value. We pushed a message with id 7, a customer, a total of "12.50" and a note with no column, then {"id":7,"total":15}. The row ended with the customer of the first message and a total of 15.00, and a generated column filled the city from the customer's address.

Mode	Each message
append	is inserted as a new row
upsert	is inserted, or updates the row with its key, with the columns it names
cdc	is a source event, applied: insert or update, delete, truncate
sql	runs your statement, with parameters taken from the message

WHY *each batch is one PostgreSQL transaction that also moves a progress row (queen.sink_progress). A message that comes back after a lost ack finds its offset already there, and is skipped. So even a statement that is not idempotent runs exactly once per message.*

4.3.3 Four things to know

- A replication slot keeps WAL while the source is down, and can fill the database's disk. Set max_slot_wal_keep_size, and alert on queen_pg_source_slot_lag_

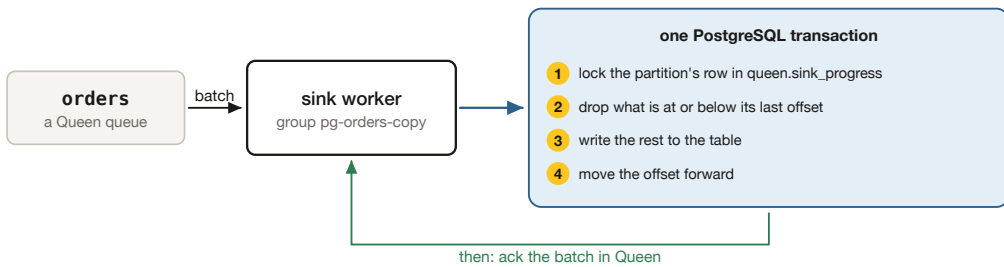


Figure 4.5: The sink: the offsets commit with the rows.

bytes.

- **Every source table needs a key in the WAL:** a primary key, or a unique index as replica identity. A table without one is refused.
- **A message PostgreSQL refuses** (a constraint, a wrong type) is tried a few times, then dead-lettered with the database's error, and its partition goes on.
- **Keep the table's key aligned with the partition** in upsert and cdc mode. Order holds inside a partition, so two partitions writing one row would race.

On the site: queenmq.com/integrations/postgres.

4.4 The Laravel Worker Supervisor

QueenMQ is also a queue driver for Laravel. Your jobs stay Laravel jobs, `dispatch()` does not change, and `php artisan queue:work` is still the worker. Install the package, then point Laravel at the broker in `.env`:

```
composer require queen-mq/php-client
php artisan vendor:publish --tag=queen-config
```

```
QUEUE_CONNECTION=queen
QUEEN_URL=http://127.0.0.1:6632
```

The **supervisor** is what runs those workers. It is a master process that starts ordinary `queue:work` processes, keeps the right number of them on each queue, restarts the ones that exit, and stops them cleanly at a deploy. It does the job of Horizon's master, with no Redis.

WHY *a fixed number of workers is too many at night and too few in a burst. A master that reads the backlog sizes the pool for you, and a clean stop means a deploy never cuts a job in half.*

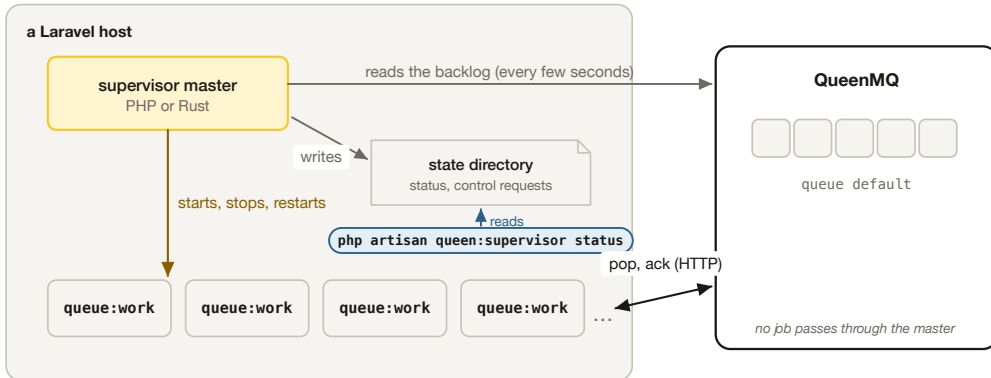


Figure 4.6: *The supervisor starts the workers; the workers talk to QueenMQ.*

4.4.1 Two engines, one configuration

Engine	What it is
PHP	Runs straight from the Composer package, with Laravel loaded in its master. Start it with <code>php artisan queen:supervise</code> .
Rust	A small native master: 7.5 MiB of memory in our run. Install it once with <code>php artisan queen:supervisor-install</code> , then start <code>vendor/bin/queen-supervisor</code> .

Both read the same supervisor section of `config/queen.php`, write the same status and obey the same commands. Start with one and move to the other without touching a pool.

4.4.2 How to use it

The supervisor keeps its state (its status, its lock, the commands you send it) in a private directory, owned by the user that runs it. It refuses a directory others could

write to:

```
install -d -m 0700 /srv/app/queen-supervisor
echo "QUEEN_SUPERVISOR_STATE_DIRECTORY=/srv/app/queen-supervisor" >> .env
```

Then describe a pool in `config/queen.php`. A pool is a set of workers with one configuration, like a Horizon supervisor. This one serves the queue `default` with one to eight workers:

```
'supervisor' => [
    'supervisors' => [
        'default' => [
            'connection' => 'queen',
            'queues' => ['default'],
            'balance' => 'auto',           // the pool follows the backlog
            'min_processes' => 1,
            'max_processes' => 8,
            'target_jobs_per_process' => 10, // one worker per 10 waiting jobs
            'fast_scale_up' => true,       // close half the gap at each step
            'timeout' => 60,               // shorter than retry_after (90 s)
            'tries' => 3,
        ],
    ],
],
```

Start it, with either engine, under the process manager you already use (systemd, Supervisor, Kubernetes), and ask it how it is:

```
php artisan queen:supervise           # the PHP engine
vendor/bin/queen-supervisor --php php --artisan artisan # or the Rust engine

php artisan queen:supervisor status
```

engine	state	live	ready	full capacity	pid	updated
php	running	yes	yes	yes	45516	2026-10-24T15:03:55Z

4.4.3 What it does with a burst

We dispatched 300 jobs of 200 ms each onto this pool. Three seconds later the supervisor had seen 290 waiting jobs and grown the pool from one worker to five. It had eight at nine seconds, and the queue was empty at fifteen. Ten seconds after that it began to give the extra workers back, one at a time, until one was left. The Rust engine did the same, within a second.

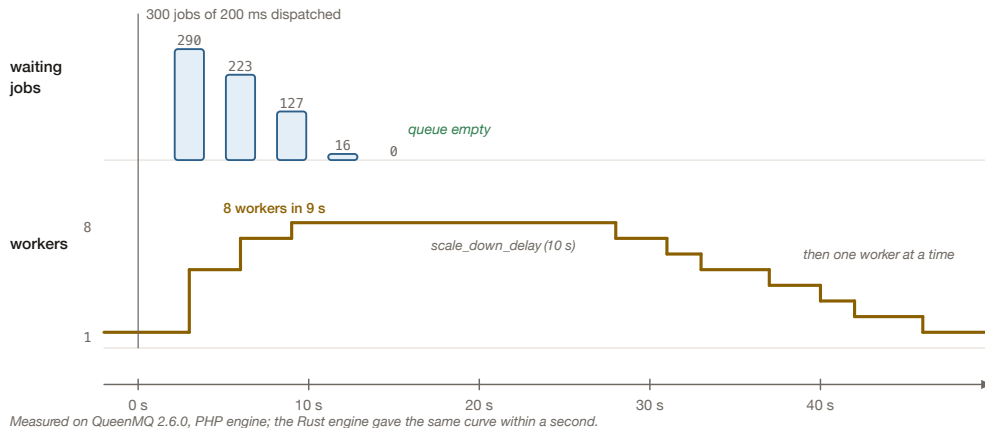


Figure 4.7: A burst of 300 jobs, as the supervisor followed it.

Pool option	Default	What it does
balance	auto	auto: the pool follows the backlog. simple: a fixed number of workers. off: every worker serves the whole ordered list of queues.
min_processes, max_processes	1, 10	The bounds of the pool.
strategy	size	size: one worker per target_jobs_per_process waiting jobs. time: enough workers to clear the backlog within target_clear_seconds.
balance_max_shift, balance_cooldown	1, 3 s	How many workers one step adds or removes, and how often. fast_scale_up closes half the gap at each step instead.

Pool option	Default	What it does
scale_down_delay	10 s	How long a lower target must hold before workers are given back.
timeout, retry_after	60 s, 90 s	A job's time limit, and its lease. The supervisor refuses to start when the timeout is not the shorter one.

4.4.4 Control it

```
php artisan queen:supervisor pause      # workers finish their job, take no new ones
php artisan queen:supervisor continue   # and start again
php artisan queen:supervisor terminate  # drain and exit, at every deploy
php artisan queen:supervisor status --check # a probe: fails unless every pool serves
```

While we paused it, 20 new jobs waited in the queue and `status --check` failed; after `continue` they were done in a few seconds. `terminate` stopped the master and every worker in two seconds.

WHY *a deploy must not cut a job in half, and what runs after it must be the new code. The `terminate` command lets each worker finish its job and hands the jobs it had not started back to the broker. Your process manager then starts a fresh master, which loads the new code.*

4.4.5 Four things to know

- **One master per application and consumer group.** A master sizes its pools from the whole backlog, so two of them would each run the full target. To run several replicas, turn on `QUEEN_SUPERVISOR_COORDINATION=true` in every one: they then share one target.
- **Prefork saves most of the memory.** With `QUEEN_SUPERVISOR_PREFORK=true`, Laravel boots once and every worker is forked from it: in our benchmark, 64 forked workers fit in 183 MiB, where Horizon's took 1,892.
- **It refuses an unsafe pool at the start:** a timeout that is not shorter than the lease, or pools whose maximums do not fit its process budget. A mistake shows at the deploy, not as a job that runs twice.
- **It runs on your worker hosts, never on the broker,** and on Unix only.

On the site: queenmq.com/integrations/laravel. More there: [worker supervisors](#), [migrate from Horizon](#).

5

Manage clusters

A broker starts with no settings at all, and most installations change very few. These are the ones that matter most. Some are read only once, when a node creates its data directory, so choose them before the first start. The sections of this chapter explain each one where it is used.

Setting	Default	What it does
<i>Read only at the first start</i>		
QUEEN_QLOG_SHARDS	0	0: each queue has its own log files. K: all queues share K logs, so a write syncs at most K files. For thousands of queues.
QUEEN_RAFT_PEERS	unset	The nodes of a cluster (see Run a cluster with Docker). After the first start, members change only through the membership API.
QUEEN_RAFT_DIR	/var/lib/queen/raft	The data directory, one per node.
<i>How fast old data leaves</i>		
PARTITION_CLEANUP_SECONDS	30 days	An empty partition is deleted after this long without a write. At least 60.
QUEEN_RAFT_TXN_WINDOW_MIN_S	900	The least time a message's dedup record is kept. Nothing is deleted sooner, partitions included.
QUEEN_RAFT_RETENTION_SCAN_PER_S	50000	Partitions the retention scan visits a second. On 2.6.0, set 40000 when partitions come and go fast (see Many short-lived partitions).
QUEEN_RAFT_TRACE_RETENTION_S	7 days	How long message traces are kept.
<i>Limits</i>		
QUEEN_RAFT_DISK_HIGH_PCT	85	Above this share of the disk, writes that would grow it answer 507.

Setting	Default	What it does
QUEEN_EPHEMERAL_MAX_BYTES	256 MiB	The memory for ephemeral queues. Above it, their pushes answer 503.
<i>Security</i>		
QUEEN_RAFT_TOKEN	unset	The secret the nodes of a cluster share.
JWT_ENABLED	false	Check a token on every request.
QUEEN_ENCRYPTION_KEY	unset	The key of encrypted queues: 64 hex characters, the same on every node.
<i>Upgrades and logs</i>		
QUEEN_RAFT_CLUSTER_VERSION_MS	1000	0 keeps the way back to the previous release open.
LOG_LEVEL, QUEEN_LOG_JSON	info, false	How much the node logs, and whether each line is a JSON object for a log collector.

5.1 Run a cluster with Docker

A cluster is three copies of the same broker, each on its own machine, each told where the other two are. Together they keep one log. The leader puts every write in that log and sends it to the others, and a write is answered only when a majority of the nodes (two of three) has it on disk. Any node serves any client: it prepares the work of its own clients and sends the leader only the prepared command.

WHY *a machine can die, and with it the disk. With three nodes, losing one costs nothing: the other two keep working, and no message they acknowledged is lost. The price is three machines, and a write that waits for a second disk.*

5.1.1 Try it on one machine

The repository has a Compose file that runs three nodes on your laptop, each with its own volume. Give the nodes a shared secret, start them, and ask each one what it is:

```
cd deploy/compose/three-node          # in a clone of github.com/queen-mq/queen
echo "QUEEN_RAFT_TOKEN=$(openssl rand -hex 32)" > .env
docker compose up -d --wait
for n in 1 2 3; do
  echo "queen-$n $(curl -s localhost:${n}6632/health | jq -r .raft.role)"
done
```

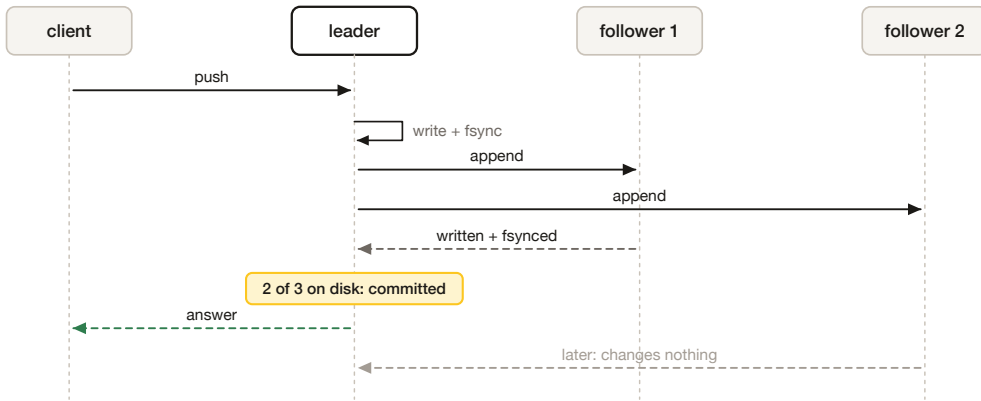


Figure 5.1: A write is answered when two of three nodes have it on disk.

```

queen-1 follower
queen-2 follower
queen-3 leader
  
```

The three nodes started empty, voted, and node 3 won. `--wait` returns when every node is healthy, a few seconds on a laptop. Node N answers on port $N6632$ (16632, 26632, 36632), API and dashboard alike. The raft port, 7400, where the nodes talk to each other, is not published at all.

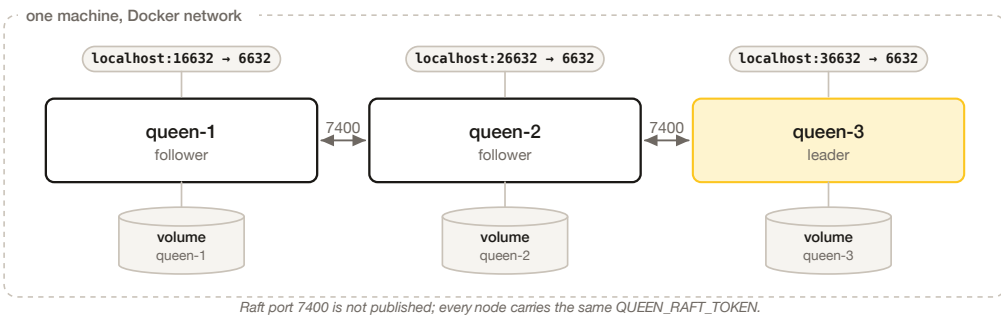


Figure 5.2: The Compose cluster on one machine.

Now give a client all three addresses. It spreads its requests over the nodes, and

when one does not answer it moves to the next:

```
const queen = new Queen({
  urls: ['http://localhost:16632', 'http://localhost:26632', 'http://localhost:36632'],
})
```

With node 1 stopped, ten pushes through this client took 64 ms in total, and a consumer read all twenty messages, the ten pushed before the stop included.

5.1.2 Stop the leader

```
docker compose stop queen-3          # the node that said leader
for n in 1 2; do
  echo "queen-$n $(curl -s localhost:${n}6632/health | jq -r .raft.role)"
done
```

```
queen-1 follower
queen-2 leader
```

Node 2 leads now. Node 3's log says how long the change took:

```
INFO rsm: raft: leadership handed off to=2 ms=28
```

Start node 3 again. It gets the writes it missed from the new leader:

```
docker compose up -d --wait queen-3
curl -s localhost:36632/health | jq -c '.raft | {role, applied, commit, lag}'
```

```
{"role":"follower","applied":56,"commit":56,"lag":0}
```

applied equal to commit means node 3 has caught up. `docker compose down -v` removes the containers and their volumes when you are done.

WHY *a planned stop should cost nothing. When a leader gets SIGTERM, it hands its job to the most up-to-date follower before it exits, so clients see no gap. A crash (kill -9, a lost machine) costs more: the others wait one to two seconds, then elect a new leader, at most about four seconds in our tests.*

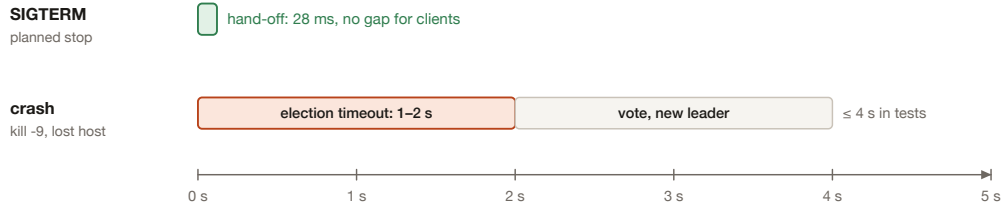


Figure 5.3: *The leader stops: a planned hand-off, or an election after a crash.*

5.1.3 Three machines

Three containers on one laptop all die with the laptop, so the Compose file is only for trying things. In production each node gets its own machine, on a private network. On each machine, run the same command with that machine's address and node number. Here are the commands for node 1:

```
TOKEN=...          # openssl rand -hex 32, made once: the same secret on every node
PEERS="1=10.0.0.11:7400/10.0.0.11:6632"
PEERS="$PEERS,2=10.0.0.12:7400/10.0.0.12:6632"
PEERS="$PEERS,3=10.0.0.13:7400/10.0.0.13:6632"

docker run -d --name queen --hostname queen-1 \
  --restart unless-stopped --stop-timeout 60 \
  -p 10.0.0.11:6632:6632 -p 10.0.0.11:7400:7400 \
  -v /var/lib/queen:/var/lib/queen/raft \
  -e QUEEN_RAFT_REPLICATOR=openraft \
  -e QUEEN_RAFT_NODE_ID=1 \
  -e QUEEN_RAFT_PEERS="$PEERS" \
  -e QUEEN_RAFT_TOKEN="$TOKEN" \
  ghcr.io/queen-mq/queen:2.6.0
```

Setting	Default	What it does
QUEEN_RAFT_REPLICATOR	local	openraft makes the node a member of a cluster.
QUEEN_RAFT_NODE_ID	1	This node's number: 1, 2, 3. A different number on each node.

Setting	Default	What it does
QUEEN_RAFT_PEERS	unset	Every node as id=raft address/HTTP address. The same list on every node.
QUEEN_RAFT_TOKEN	unset	The secret the nodes show each other. Unset, anyone who reaches port 7400 can vote.
QUEEN_RAFT_DIR	/var/lib/queen/raft	This node's data. One directory per node, never shared.

The Docker flags matter too. `--stop-timeout 60` gives a stopping node time to hand over its work (Docker's default of 10 seconds can kill it halfway). `--restart unless-stopped` brings a node back after a crash, and after the one planned exit a node makes: when it was away so long that the leader sent it a snapshot, it exits with code 75 and loads the snapshot when it starts again. `--hostname` gives the node a stable name in the logs and the dashboard.

WHY *the peer list is how a new cluster is born: every node reads the same list on an empty directory, and they elect a leader. After that first start the list lives in the log, and the nodes change it only through the membership API (see Recovery).*

5.1.4 Five things to know

- **The addresses are fixed at the first start.** They are stored in the log. Changing a host name or an IP under a running cluster cuts the nodes off from each other: put the old addresses back, or run a forced recovery. Use addresses that do not change.
- **Keep both ports private.** Port 6632 has no authentication unless you turn it on (see Configuration for different use cases), and port 7400 is plain HTTP with a shared token. Neither should face the internet.
- **One node does not grow into a cluster.** To go from one node to three, start a new cluster beside the old one and move the clients to it.
- **Give every node the same disk size.** A node refuses its own clients' writes at 85% full, but writes accepted by the other nodes keep arriving until its disk is full.
- **Run NTP on every machine.** Leases are timed on the leader's clock, and they stay exclusive across a leader change only while the clocks agree within half a second.

On the site: queenmq.com/manage/docker.

5.2 Run a cluster inside Kubernetes

On Kubernetes a cluster is a StatefulSet of three pods. Each pod gets its own volume, the pods find each other through a headless Service, a second Service carries port 6632 to clients, and a disruption budget lets Kubernetes evict only one pod at a time. There are two ways to run it: a plain manifest you apply yourself, or the operator, which runs the same cluster from one object.

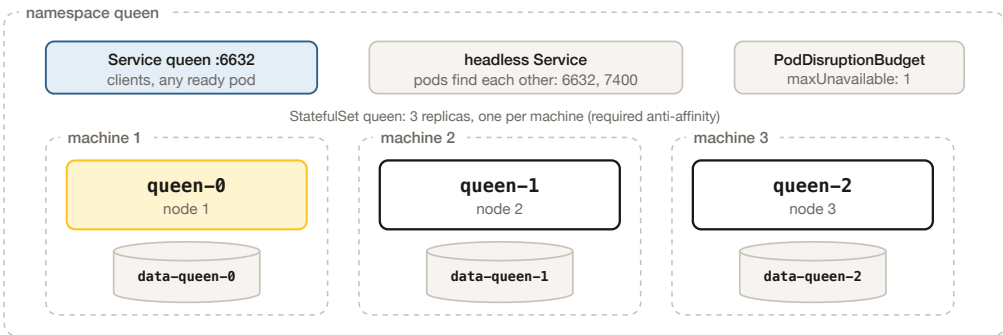


Figure 5.4: The cluster on Kubernetes.

The repository ships no Helm chart. The complete manifest is on the site (queenmq.com/manage/kubernetes), ready to copy or to put in your own chart.

5.2.1 The secret

Both ways start with a namespace and a secret: the token the nodes share, and the key that encrypts payloads on disk.

```
kubectl create namespace queen
kubectl -n queen create secret generic queen \
  --from-literal=QUEEN_RAFT_TOKEN="$(openssl rand -hex 32)" \
  --from-literal=QUEEN_ENCRYPTION_KEY="$(openssl rand -hex 32)"
```

Keep a copy of the encryption key somewhere safe, apart from the cluster. Without it, the payloads of encrypted queues cannot be read, and a backup does not contain it.

5.2.2 The manifest

The manifest is ordinary Kubernetes, but a few fields are there for a reason that is not obvious. Change them only if you know what you lose.

Field	Value	Why
podManagementPolicy	Parallel	With publishNotReadyAddresses, lets all pods start together. Without both, a full restart never ends: no pod is ready without a leader, and no leader is elected until the pods can find each other.
publishNotReadyAddresses	true	On the headless Service, for the same reason.
QUEEN_RAFT_NODE_ID	ordinal	Pod queen-0 is node 1, queen-1 is node 2, and so on.
readinessProbe	/health	A pod that does not know a leader, or is behind, gets no clients, and a rolling update waits for it.
livenessProbe	TCP on 6632	Never /health: an election would make every pod fail it, and Kubernetes would restart them all at once.
terminationGrace-PeriodSeconds	90	A stopping pod hands over its work first, which can take close to a minute. Shorter, and the stop becomes a crash.
CPU limit	none	Throttling turns a burst of work into a missed heartbeat. Memory request equals the limit.
podAntiAffinity	required	One voter per machine: two voters on one machine turn its failure into a lost majority.

After `kubectl apply`, check that the three nodes formed one cluster:

```
kubectl -n queen rollout status statefulset/queen
kubectl -n queen exec queen-0 -- curl -s localhost:6632/api/v1/system/raft/membership \
  | jq -c '.membership | {leader, term, voters, members: [.members[] | {nodeId, live, lag}}}'
```

```
{"leader":1,"term":4,"voters":[1,2,3],"members":[{"nodeId":1,"live":true,"lag":0},...]}
```

Three voters, all live, none behind. Clients use the queen Service and can land on any pod.

WHY *a StatefulSet gives each pod a stable name and its own volume, which is what a node of a cluster needs: the same identity and the same disk after every restart.*

5.2.3 The operator

A StatefulSet cannot add a voter, replace a node whose volume is lost, or grow its volumes, and its rolling update moves on as soon as a pod is Ready. The operator does those steps for you, through the same HTTP API you would use. Install it once:

```
kubectl apply -f \
  https://raw.githubusercontent.com/queen-mq/queen/master/deploy/operator/install.yaml
```

Then, with the secret above, a cluster is one object:

```
apiVersion: queenmq.com/v1alpha1
kind: QueenCluster
metadata:
  name: queen
  namespace: queen
spec:
  version: "2.6.0"
  replicas: 3
  secretName: queen
  storage:
    size: 50Gi
```

```
kubectl -n queen get queencluster
```

NAME	VERSION	VOTERS	COMMITTS	PHASE	AGE
queen	2.6.0	[1,2,3]	true	Ready	40s

COMMITTS is a write the operator makes once a minute: `true` while the cluster can commit. The pods and Services have the same names as in the manifest.

WHY *the steps that keep a cluster whole (remove a node before you wipe it, add it back as a learner, restart the leader last) are easy to get wrong by hand on a bad day. The operator does them in the right order, and stops and says why when only a person can decide. It is young: its API is `v1alpha1`, and a field may still change.*

5.2.4 Four things to know

- **Scaling the StatefulSet does not add a voter.** The members live in the log. Add or remove voters through the membership API, or change `replicas` on the operator's object.
- **Some things are fixed at the first start:** the StatefulSet, Service and namespace names (they are part of the node addresses), and the ports.
- **Delete one volume at a time, and remove the member first.** In a test, deleting two volumes at once left a cluster that could not commit while every pod said Ready.
- **Pin the image version.** A pod that restarts on `latest` pulls whatever is current that day. An upgrade should be a step you take.

On the site: queenmq.com/manage/kubernetes. More there: [the operator](#).

5.3 Configuration for different use cases

Every setting is an environment variable, and most installations need very few of them. When a node starts, it prints the settings it is really using, in lines that start with `config:.` Read them after every change. A yes/no setting accepts `true` or `false`, `1` or `0`, `yes` or `no`, `on` or `off`; any other value stops the node at boot, so a typo cannot pass unnoticed.

You need	What to set
To try Queen, or to develop	Nothing
A small production	One node with a volume, and backups
To survive the loss of a machine	Three nodes; five to survive two
To keep strangers out	Broker JWT, or the built-in proxy
Many teams or customers on one cluster	The proxy, and raft groups
Payloads that must not sit on disk in clear	<code>QUEEN_ENCRYPTION_KEY</code>
Disk that does not fill up	Retention on the queues
Many short-lived partitions	<code>PARTITION_CLEANUP_SECONDS</code> and two more
Kafka clients	<code>QUEEN_KAFKA_EMBEDDED</code>
Thousands of queues	<code>QUEEN_QLOG_SHARDS</code> , before the first start
To survive the loss of a whole site	A standby cluster

5.3.1 To try Queen, or to develop

One command, no settings:

```
docker run -p 6632:6632 ghcr.io/queen-mq/queen:2.6.0
```

Without a volume, the data lives inside the container and is gone when the container is removed. That is usually what you want on a laptop.

5.3.2 A small production

One node with its data on a volume:

```
docker run -d --name queen --restart unless-stopped --stop-timeout 60 \
  -p 6632:6632 -v /var/lib/queen:/var/lib/queen/raft \
  ghcr.io/queen-mq/queen:2.6.0
```

A single node writes every message to disk (fsync) before it answers. A crash, a kill -9 or a power cut loses nothing it acknowledged: it starts again and reads its log. What it cannot survive is losing its disk, so take backups (see Recovery). An upgrade stops it for a few seconds, and its ephemeral queues, which live in memory, are empty after every restart. Port 6632 stays on a private network, as on a cluster (see To keep strangers out).

WHY *many workloads do not need three machines. One node is the simplest thing that keeps every acknowledged message. Choose it knowing that a lost disk means restoring the last backup.*

5.3.3 To survive the loss of a machine

Run three nodes, or five (see Run a cluster with Docker). A majority of three is two, so three nodes survive one failure. Five survive two, but every write then waits for three disks, and the leader feeds four followers. Four nodes survive only one failure, like three, while every write waits for three of them: an even number only costs. Put each node on its own machine, and in its own zone if you have zones.

WHY *a majority is what lets the cluster decide alone, without a person, which node leads. More nodes make a cluster harder to kill, not faster: one leader orders every write, so adding nodes adds no write throughput.*

5.3.4 To keep strangers out

Out of the box a node checks nobody: anyone who reaches port 6632 can read, write, delete queues and change the cluster's members. Choose one of two front doors.

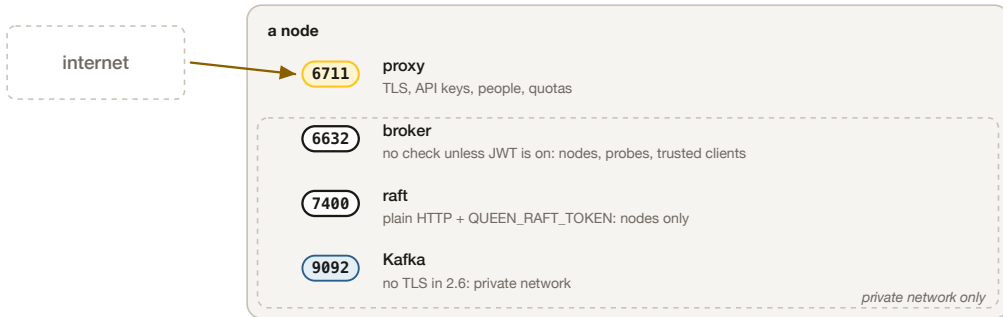


Figure 5.5: What listens where: only the proxy's port faces the internet.

Setting	Default	What it does
JWT_ENABLED	false	The broker checks a token on every request. For services that already get tokens from an identity provider.
JWT_ALGORITHM	HS256	HS256 checks with JWT_SECRET; RS256 and EdDSA with JWT_PUBLIC_KEY or JWT_JWKS_URL.
QUEEN_PROXY_EMBEDDED	false	The built-in proxy: API keys, people who sign in to the dashboard, tenants, quotas.
QUEEN_PROXY_PORT	unset	The proxy's own port. Expose this one, and keep 6632 inside.
QUEEN_TLS_CERT, _KEY	unset	TLS on the proxy's port. The broker port has no TLS: use your load balancer.

A token's roles decide what it may do: read-only, write-only (push, for producers that must not read), read-write and admin (the membership API and deletes). Whatever you choose, keep the raft port (7400) on a private network, with QUEEN_RAFT_TOKEN set on every node.

WHY a broker holds every payload and every key of your system. The default is open so that the first run is one command, and it is the first thing to change before anything else can reach the node.

5.3.5 *Many tenants on one cluster*

One cluster can serve many teams or customers. Each one is a tenant: its queues, KV and timers are invisible to the others. The proxy built into the broker adds API keys and people in front of it. It keeps its own data (tenants, users, key hashes, plans, usage) in the broker's replicated KV, so there is no database to run.

```
SESSION_SECRET=$(openssl rand -hex 32)      # signs console sessions
CP_TOKEN=$(openssl rand -hex 32)            # opens the control plane, /api/cp/*
QUEEN_KEY="qk_live_$(openssl rand -base64 32 | tr '+/' '-' | tr -d '=)'"

docker run -d --name queen -p 6711:6711 -p 127.0.0.1:6632:6632 \
  -v /var/lib/queen:/var/lib/queen/raft \
  -e QUEEN_PROXY_EMBEDDED=true \
  -e QUEEN_PROXY_PORT=6711 \
  -e QUEEN_PROXY_JWT_SECRET="$SESSION_SECRET" \
  -e QUEEN_PROXY_CP_TOKEN="$CP_TOKEN" \
  -e QUEEN_PROXY_SPOOL_DIR=/var/lib/queen/raft/proxy-spool \
  -e QUEEN_PROXY_BOOTSTRAP_TENANT=acme \
  -e QUEEN_PROXY_BOOTSTRAP_API_KEY="$QUEEN_KEY" \
  -e QUEEN_PROXY_DEFAULT_CLUSTER=acme \
  ghcr.io/queen-mq/queen:2.6.0
```

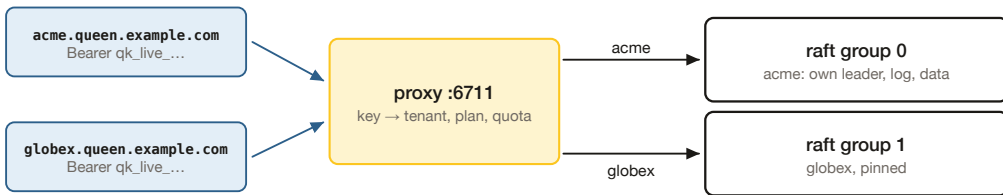
The first start creates the tenant `acme` and its key. Clients now use port 6711, with the key (bearerToken in the JavaScript client); without it they get 401 unauthorized. Port 6632 is the broker itself, with no keys: the nodes talk to each other on it, so it stays inside your network.

Add a tenant with one call to the control plane. You make the key yourself and send only its hash, so the proxy never sees the key:

```
KEY="qk_live_$(openssl rand -base64 32 | tr '+/' '-' | tr -d '=)'"
HASH=$(printf %s "$KEY" | sha256sum | cut -d' ' -f1)
curl -s -X POST localhost:6711/api/cp/provision -H "x-queen-cp-token: $CP_TOKEN" \
  -H 'content-type: application/json' -d '{
    "tenant_slug": "globex", "cluster_slug": "globex", "plan": "pro",
    "user_id": "308d3f3d-2306-41de-95f1-e32b12ec7470", "email": "ops@globex.example",
    "role": "admin", "key_name": "default", "key_hash": "'"$HASH"'",
    "scopes": ["produce", "consume", "read"]}
{"broker_tenant_uuid": "91a5d7bb-879c-4ae2-95e2-295fa2704098", "created": {"cluster": true, ...}}
```

The proxy finds the tenant from the first part of the host name: `globex.queen.example.com` is `globex`. A key works only there: on another tenant's host it gets 403 key/cluster mismatch. Each plan (free, dev, pro, dedicated-s) limits requests a second, queues and stored bytes. The proxy starts by only logging what it would block; `QUEEN_PROXY_ENFORCE=true` makes it block.

With one raft group, all tenants share one leader. `QUEEN_RAFT_GROUPS=3` runs three groups in each node, each with its own leader, log and directory (`groups/g1/, ...`). A tenant lives in one group, chosen by a hash of its id. To choose yourself, pin it by the `broker_tenant_uuid` of the answer above: `QUEEN_TENANT_GROUPS=<uuid>=1` puts it in group 1. Pin a big tenant before it writes, because a tenant cannot move while it has data. With as many groups as nodes, every node leads one group.



A key works only on its own tenant's host: elsewhere, 403 key/cluster mismatch.

Pinning: `QUEEN_TENANT_GROUPS=<broker tenant uuid>=1`. With as many groups as nodes, every node leads one.

Figure 5.6: *Tenants through the proxy, each in one raft group.*

WHY *tenants share machines, not data. One leader per tenant keeps each tenant's transaction a single log entry, and more groups spread the tenants' writes over more leaders. Raft groups were not part of the Jepsen tests, which ran one group.*

Three things to know: every node needs the same secrets; the membership routes are closed on the proxy's port, so call them on 6632; and keep `QUEEN_PROXY_SPOOL_DIR` on the data volume, where usage waits when it cannot be written.

5.3.6 Payloads that must not sit on disk in clear

Give every node the same key, 64 hex characters, then turn encryption on for the queues that need it (`encryptionEnabled`, see Create and configure queues):

```
QUEEN_ENCRYPTION_KEY="$(openssl rand -hex 32)" # the same value on every node
```

Payloads are encrypted with AES-256-GCM. Queue and partition names, KV keys and KV values are not. Check that the boot log says encryption service initialized (AES-256-GCM). Without a key, the node stores those queues in clear and says so only in its config lines; a key of the wrong length logs one warning and does the same.

WHY *some payloads hold personal data, and a disk, or a copy of it, can end up in the wrong hands. The price is the key itself: lose it and the payloads are gone; change it and consumers receive the encrypted envelope as their data, with no error. Keep it safe, apart from the backups.*

5.3.7 Disk that does not fill up

Queen keeps every message until retention removes it, even after every group has acked it (see Create and configure queues). Turn retention on for the queues that grow. As a last guard, a node whose disk is 85% full (QUEEN_RAFT_DISK_HIGH_PCT) refuses the writes that would grow it, with 507 storage_full, and accepts them again below 80% (QUEEN_RAFT_DISK_LOW_PCT). Reads, acks and deletes keep working.

WHY *a broker that keeps history lets new consumer groups and replays read old messages. Retention is where you decide how long that history is worth its disk. The 85% gate leaves room to grow the volume before the node stops.*

5.3.8 Many short-lived partitions

Some workloads open a partition per trace, per session or per order, and never write to it again. By default an empty partition stays for 30 days, and each one costs memory and time in every retention round. To delete them soon after their last message, set three things on the broker:

```
docker run -d --name queen -p 6632:6632 -v /var/lib/queen:/var/lib/queen/raft \
-e PARTITION_CLEANUP_SECONDS=60 \
-e QUEEN_RAFT_TXN_WINDOW_MIN_S=60 \
-e QUEEN_RAFT_RETENTION_SCAN_PER_S=40000 \
ghcr.io/queen-mq/queen:2.6.0
```

and give the queue retention and a short dedup window:

```
curl -s -X POST http://localhost:6632/api/v1/configure -H 'content-type: application/json' \
-d '{"queue":"traces","options":{"retentionEnabled":true,
"completedRetentionSeconds":60,"dedupWindowSeconds":60}}'
```

Retention removes a message once it is a minute old and every group has acked it, and the empty partition is deleted after a minute without a write. We pushed 50 messages to 50 partitions and consumed them: all 50 partitions were gone within 76 seconds.

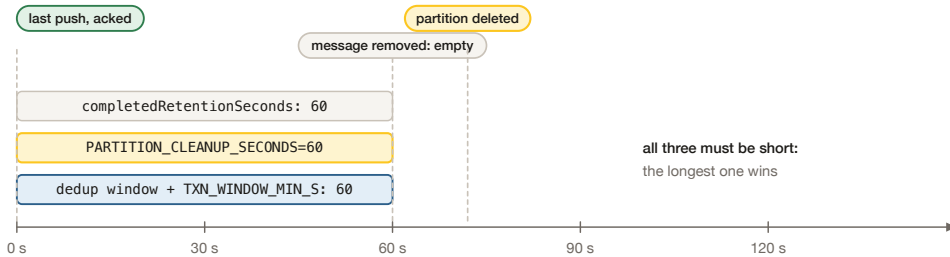


Figure 5.7: A short-lived partition: three settings, and the longest wins.

WHY a partition holds the dedup records of its messages. Deleted too early, a producer that retries would store a message twice. So a partition is never deleted inside its queue’s dedup window, nor before `QUEEN_RAFT_TXN_WINDOW_MIN_S` (15 minutes by default): a short cleanup needs both to be short too.

On 2.6.0 the third setting matters when partitions come and go fast: we saw the problem from 200 new partitions a second. By default the retention scan paces each round from the size of the previous one. When new partitions arrive faster than that, a round never ends, and nothing is deleted: in a test with 1,000 new partitions a second, 594,000 of 595,000 partitions were still there after ten minutes. 40000 turns that pacing off. With it, 1,000 and 2,000 new partitions a second stayed flat, at about 70,000 and 150,000 partitions.

5.3.9 Kafka clients

`QUEEN_KAFKA_EMBEDDED=true` opens a Kafka port (9092) on every node, so existing Kafka producers and consumers work without code changes. On a cluster, each node also needs `QUEEN_KAFKA_ADVERTISED_ADDR`, the address at which clients reach that node’s Kafka port. The Kafka port has no TLS in 2.6, and passwords (`QUEEN_KAFKA_SASL=plain`) cross the network in clear: keep it on a private network. How to use with Kafka clients, in chapter 4, has the details.

5.3.10 Thousands of queues

By default each queue has its own log file on disk, and a write can touch many files. `QUEEN_QLOG_SHARDS=K` puts all queues into K shared logs instead, so a write syncs at

most K files. It is read only when the data directory is created, so set it before the first start.

5.3.11 *To survive the loss of a whole site*

Three nodes in one zone, or in one Kubernetes cluster, all go down together when that goes. A standby is a second cluster, somewhere else, that reads everything the first one commits and stays a moment behind it. When the first is lost, you promote the standby and point the clients at it.

WHY *replicas protect against a lost machine, backups against a mistake. A standby protects against losing everything at once, and loses only the last moment before the failure. It does not protect against mistakes: a deleted queue is replayed on the standby a moment later. Keep the backups too.*

On the site: queenmq.com/manage/configuration. More there: [every setting](#), [security](#), [tenants and API keys](#), [standby cluster](#).

5.4 Upgrading and monitoring

5.4.1 *Upgrade one node at a time*

Stop one node, start the new release on the same data, and wait until its `/health` answers 200 before you touch the next. Healthy means the node knows the leader and has caught up (it is less than 1,000 entries or two seconds behind), so it is back in the majority before the next one stops.

In the Compose cluster, put the new version in `.env` and let `--wait` hold each step until the node is healthy:

```
echo "QUEEN_VERSION=<new-version>" >> .env
for n in 1 2 3; do docker compose up -d --wait queen-$n; done
```

We rolled three nodes from 2.5.0 to 2.6.0 this way in 18 seconds. Every message pushed before the roll was there after it, and pushing one of them again was answered duplicate: the deduplication memory survived too.

On your own machines, do the same by hand, one machine at a time:

```
docker stop -t 60 queen && docker rm queen
# the same docker run as before, with the new version at the end, then:
until curl -sf -o /dev/null http://10.0.0.11:6632/health; do sleep 1; done
```

On Kubernetes, change the image. The StatefulSet replaces one pod at a time, highest number first, and waits for each to be Ready. The operator does the same, and restarts the leader last, so the leadership moves only once:

```
kubectrl -n queen set image statefulset/queen queen=ghcr.io/queen-mq/queen:<new-version>
kubectrl -n queen patch queencluster queen --type merge \
  -p '{"spec":{"version":"<new-version>"}},'
```

(The first line is for the manifest, the second for the operator.) A single node has no one to take over: its upgrade is a stop and a start, a few seconds without service. Give the stop 60 seconds, because pods that are waiting for messages may take up to 30 seconds to finish.

WHY *two nodes of three must be up for anything to commit. One node at a time, each one healthy before the next, keeps two up at every moment, and the clients see at most a slower answer.*

5.4.2 The way back

A release can change how data is written. It does so only when every node runs it: until the last node is upgraded, the cluster keeps writing the old format, and the new feature stays off. The format is the `clusterVersion` in `/health`. While it has not moved, you can go back by starting the old release on the same data. Once it moves, an older release refuses to start on that data.

To keep the way back open while a new release proves itself, set `QUEEN_RAFT_CLUSTER_VERSION_MS=0` on every node: the version then never moves. Remove the setting when you decide to stay, and the version moves within a second. 2.5.0 and 2.6.0 write the same format (version 7).

WHY *a rolling upgrade means old and new nodes run side by side for a while. If a new node wrote something an old node cannot read, the old node would stop. Waiting for the last node is what makes the roll safe in both directions, up to the moment you choose.*

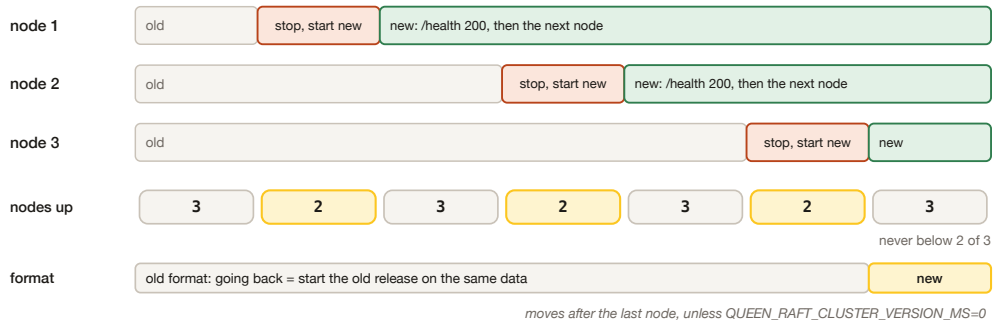


Figure 5.8: A rolling upgrade, and when the data format moves.

5.4.3 What to watch

Every node serves three things on port 6632: the dashboard at /, its health at /health, and its metrics at /metrics/prometheus. Each one describes the node that answered, so probe and scrape every node at its own address, never through a load balancer.

The dashboard. Its System page shows the replicated log and every node, with its role, its term, how far it has applied the log and how much disk it uses. A bar at the top of every page says when a node refuses writes because its disk is full.

/health. It answers 200 healthy when the node knows a leader, is less than two seconds behind, and the leader has heard from a majority in the last five seconds. Otherwise it answers 503 settling.

```
curl -s http://10.0.0.13:6632/health
{"status":"healthy","engine":"raft","version":"2.6.0",
  "raft":{"role":"leader","leader":true,"term":1,"applied":5,"commit":5,"lag":0,
    "quorumAckMs":8,"storageReady":true,"clusterVersion":7,"kinds":7,
    "apply":{"failure":null,"skipped":[]}}}
```

Field	Healthy	What it means
role		leader, follower, learner (joining, not voting yet), candidate or stopped.

Field	Healthy	What it means
applied, commit	equal	How far this node has applied the log, and how far the cluster has committed it.
lag	0	Milliseconds this node is behind the leader.
quorumAckMs	small	Milliseconds since the leader last heard from a majority. It grows when the majority is lost.
apply.failure	null	The entry that stopped this node, if one did (see Recovery).
clusterVersion, kinds		The format the cluster writes, and the newest this build can read.

Use `/health` for readiness. Use a plain TCP check on port 6632 for liveness, never `/health`: a 503 during an election is no reason to restart a node.

A write probe. `/health` reports what a node knows. It writes nothing, so it cannot prove that the cluster commits. Three seconds after we stopped two nodes of three, the third still answered 200, though no write could commit any more. So, beside `/health`, write a small KV key through every node every minute, and alert when it fails or takes more than a few seconds:

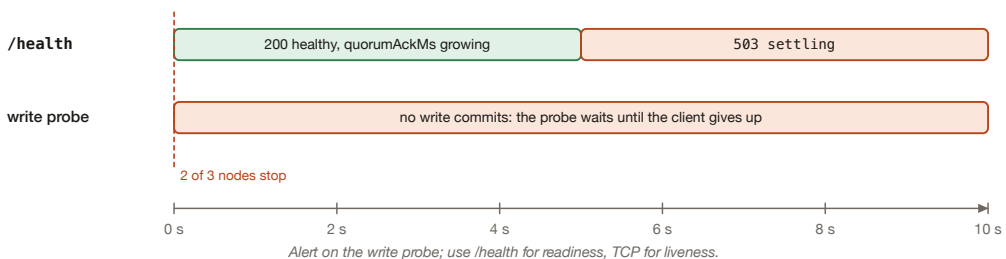


Figure 5.9: After a lost majority: `/health` trails, the write probe does not.

```
curl -s -m 10 -X PUT http://10.0.0.11:6632/api/v1/kv/ops/probe \
  -H 'content-type: application/json' -d '{"value":"probe","ttlSeconds":60}'
{"applied":true,"index":0,"key":"probe","op":"put","value":"probe","version":1}
```

On a healthy cluster it answers in a few tens of milliseconds. The key expires after a minute, so the probe leaves nothing behind.

Metrics. These are the series worth an alert:

Series	Alert when	What it means
queen_raft_storage_full	= 1	The disk is 85% full: writes that grow it answer 507.
queen_raft_index	committed – applied keeps growing	The node applies the log slower than the cluster commits it.
queen_raft_admit_total	refused rises	Writes are refused because the node is overloaded (429).
queen_dlq_depth_by_queue	rises	Messages are piling up in a dead-letter queue.
queen_queue_pop_lag_milliseconds	high	Consumers are behind.
queen_uptime_seconds	< 300	The node restarted.

For memory, compare the process's own memory with its limit (container_memory_rss on Kubernetes). The resident size also counts files the kernel keeps in its cache and drops before it runs out of memory.

Log lines. A few lines deserve an alert of their own:

Line	Level	What it means
FATAL: ...	ERROR	The node will not start. The line says why.
APPLY FAILED at entry N	ERROR	An entry stopped this node (see Recovery).
raft storage pressure changed full=true	WARN	The disk gate closed.
refused this node's QUEEN_RAFT_TOKEN	WARN	The nodes do not share one token.
the process exits to load a received snapshot	ERROR	Exit code 75. Normal once; trouble if it repeats.

WHY *a cluster hides a failure well, which is the point of it, and also the danger: two nodes serve while the third is broken, and nobody notices until a second one goes. The probe and these alerts are how you notice while one failure is still free.*

On the site: queenmq.com/manage/upgrades-monitoring. More there: [every metric](#).

5.5 Recovery

Most failures need nothing from you. A node that crashes catches up from the leader when it starts again. A leader that gets SIGTERM hands its job to a peer before it stops. A node that was away too long gets a snapshot from the leader, exits with code 75, and loads it when it starts again. This section is for the rest: a lost disk, a lost majority, a full disk, a bad day.

5.5.1 Start with a write

The question that decides everything is whether the cluster can still commit. Ask it with a write, through two different nodes, because a node that is itself broken can hang while the cluster is fine:

```
curl -s -m 10 -X PUT http://10.0.0.11:6632/api/v1/kv/ops/probe \
  -H 'content-type: application/json' -d '{"value":"probe","ttlSeconds":60}'
```

An answer within a second means the cluster commits, and you are looking for one sick node. No answer within ten seconds means it does not. Then ask every node what it thinks, and the leader who the members are:

```
for n in 10.0.0.11 10.0.0.12 10.0.0.13; do
  echo "$n $(curl -s -m 2 http://$n:6632/health | jq -c '{status, role: .raft.role,
    term: .raft.term, applied: .raft.applied, commit: .raft.commit}')"
done
curl -s http://10.0.0.11:6632/api/v1/system/raft/membership \
  | jq -c '.membership | {leader, voters, members: [.members[] | {nodeId, live, lag}]}'
```

```
{"leader":2,"voters":[1,2,3],"members":[{"nodeId":1,"live":true,"lag":0},
{"nodeId":2,"live":true,"lag":0}, {"nodeId":3,"live":true,"lag":0}]}
```

On a healthy cluster every node reports the same term, and an applied close to the leader's commit. A node that will not start says why in its last FATAL line, in docker logs or `kubectl logs --previous`.

5.5.2 Rules for a bad day

Each of these comes from a test where breaking it made things worse.

1. **One node at a time.** Change a node, wait until it is healthy, then touch the next.
2. **Remove a node from the members before you wipe its disk.** A node that comes back empty under its old number is never repaired, and requests sent to it hang.
3. **Never put an old copy back on one node of a running cluster.** Its log went backwards, which is the same trap. Replace the node instead.
4. **Never delete files in a data directory to make room.** A missing file is not noticed at start, and the node then serves holes. Grow the disk, or replace the node.
5. **Force a recovery on exactly one node, and only when a majority is gone for good.** On two nodes it makes two clusters.
6. **Keep QUEEN_ENCRYPTION_KEY safe and unchanged.** With another key, consumers of encrypted queues get the encrypted envelope as their data, and nothing reports an error.

5.5.3 Find your case

What you see	What to do
A node restarted, and is healthy again	Nothing
A node exited once with code 75	Nothing: it loaded a snapshot
A node will not start: a FATAL line names a damaged file, or it exits with code 139 and no line	Replace the node
A node answers, but its applied stays at 0 or far behind	Replace the node, starting with its removal
Writes to one node answer 507, or it is stopped with no space left	Grow its disk
A node logs refused this node's QUEEN_RAFT_TOKEN	Give it the same token, restart it
A majority of the nodes is down, their disks intact	Bring them back
A majority of the disks is gone for good	Force a recovery on the survivor
Every node stopped on the same entry (APPLY FAILED)	Skip the entry
Every disk is gone	Promote the standby, or restore a backup

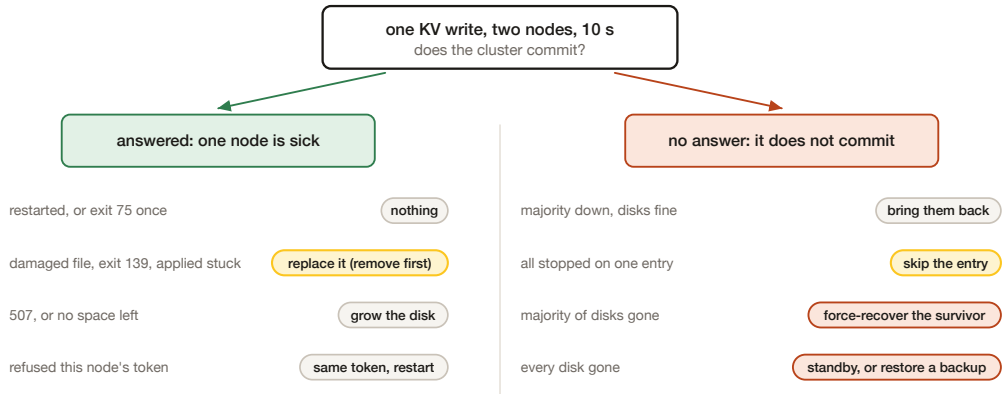


Figure 5.10: Start with a write, then find your case.

5.5.4 Replace a node

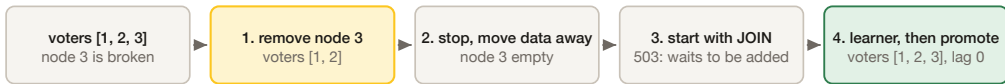
This is the procedure you will use most. The node comes back empty and copies everything from the leader, so nothing is lost while the other two commit. Check first that the probe commits through another node.

```
# 1. Remove node 3 from the members, through a healthy node.
curl -s -X DELETE http://10.0.0.11:6632/api/v1/system/raft/membership/members/3
# 2. On node 3: stop it and move its data away.
docker stop -t 60 queen && docker rm queen
mv /var/lib/queen /var/lib/queen.broken
# 3. Start it on an empty directory: the same docker run, plus -e QUEEN_RAFT_JOIN=true
# 4. Add it back as a learner, then make it a voter.
curl -s -X POST http://10.0.0.11:6632/api/v1/system/raft/membership/learners \
  -H 'content-type: application/json' \
  -d '{"id":3,"raft":"10.0.0.13:7400","http":"10.0.0.13:6632"}'
curl -s -X POST http://10.0.0.11:6632/api/v1/system/raft/membership/promote \
  -H 'content-type: application/json' -d '{"ids":[3]}'
```

Step 1 answers "ok":true with the voters left, [1,2]. A 409 no_quorum means too few nodes are live: stop, and look again. At step 3 the empty node answers 503 and logs this fresh node joins an existing cluster: QUEEN_RAFT_JOIN makes it wait to be added instead of starting a cluster of its own. A learner copies the log but does not vote; if the promotion answers 409 learner_behind, it is still copying, so try again a few seconds later. It is done when the members list three voters with lag 0.

On the operator, one annotation does all four steps (pod queen-2 is node 3): `kubectl -n queen annotate queencluster queen queenmq.com/replace-pod=queen-2`.

WHY *the order keeps the cluster whole: out of the members first, then the disk, then back in as a learner that votes only once it has caught up. A node wiped while still a member would count in the majority with an empty log.*



Remove before you wipe: a node that comes back empty under its old number is never repaired.

A learner copies the log but does not vote: it counts in the majority only once caught up.

Figure 5.11: *Replacing a node: out of the members, wipe, back in as a learner.*

5.5.5 A majority is down, the disks are fine

Two nodes of three are gone for a while: a lost machine, a zone outage, an image that will not pull. Nothing commits, and the survivor answers 503. Bring the missing nodes back, and do nothing else: do not delete a volume, and do not force a recovery. As soon as two nodes run, they elect a leader. When we stopped two nodes of three and started one of them again, the very next write committed.

5.5.6 A disk is full

At 85% a node refuses its own clients' writes, but entries accepted by the other nodes keep arriving. At 100% it logs `No space left on device` and stops; the other nodes keep serving. Grow its volume and restart it: it catches up on its own. Then grow the other volumes too, because they hold the same data, and turn on retention for the biggest queues.

5.5.7 A majority of the disks is gone for good

The majority will never come back, so the cluster can never elect a leader again. `QUEEN_RAFT_FORCE_RECOVER` is the way out. Set on the one survivor, it makes that node the only voter, and it leads alone. The others are rebuilt empty and join it.

1. Restart the survivor with `QUEEN_RAFT_FORCE_RECOVER=1` (its own node number). Its log says `UNSAFE RECOVERY`, and the members list it as the only voter.

2. Restart it once more without the setting.
3. Start the other nodes on empty directories with `QUEEN_RAFT_JOIN=true`, add them as learners and promote them, as in steps 3 and 4 of *Replace a node*.

The cost is the writes the lost nodes had committed without the survivor: usually nothing, at worst the last moment before the failure. If more than one node survived, pick the one with the highest commit, and rebuild the others empty.

5.5.8 *Every node stopped on the same entry*

When every node refuses the same entry, it is a bug in Queen, and restarting never helps: each restart stops on it again. Every node shows the way out in `/health`, under `apply.failure`, and in its `APPLY FAILED` log line: the entry's index and digest. Check that two nodes agree, and that the class is deterministic. Then set `QUEEN_RAFT_APPLY_SKIP=<index>:<digest>` on every node and restart them all. The entry becomes a no-op: none of its effects apply, and nothing else is lost. Keep the log line for the bug report.

5.5.9 *Backups*

Three nodes protect you from losing a machine, not from a mistake: a deleted queue reaches every node as faithfully as a good write. A backup is the copy that lives outside the cluster. A node keeps everything it knows in one directory, and one copy of one node is enough to rebuild a whole cluster.

The copy must be taken at one instant, like the state a power cut leaves behind, which Queen survives. Pause the container, copy, and resume. On a cluster, pause a follower: the other two keep committing without it.

```
docker pause queen
tar -C /var/lib/queen -cf /backup/queen-$(date -u +%Y%m%dT%H%MZ).tar .
docker unpause queen
```

A snapshot of the volume (a cloud disk snapshot, LVM, ZFS) is also taken at one instant, and works while the node runs. A plain `cp` or `rsync` of a running node does not: its files are copied at different moments.

To restore a single node, stop it, unpack the copy into an empty directory, and start it. To restore a whole cluster, unpack the copy as node 1's directory, start node 1 alone with `QUEEN_RAFT_FORCE_RECOVER=1`, restart it without, and let the others join it empty, as above. We did exactly that: every message pushed before the copy was

back, and the one pushed after it was not. If the copy comes from a node that was force-recovered once, move `raft/force_recovered.json` out of the restored directory first, or node 1 refuses a second recovery.

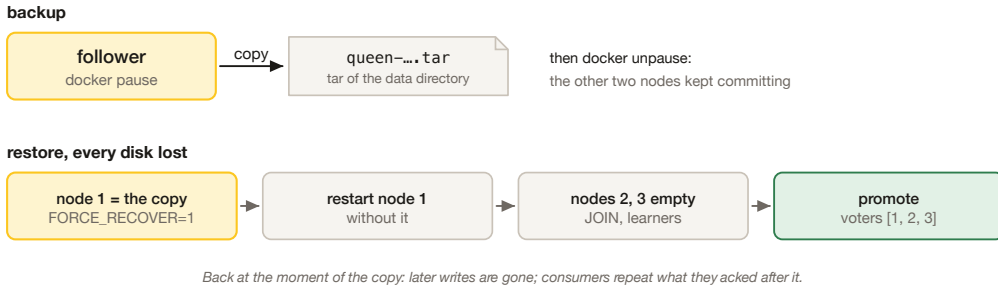


Figure 5.12: One copy of one node rebuilds the whole cluster.

WHY a restore goes back to the moment of the copy, consumers included: what they acked after it is delivered again. Producers can push what was lost again, and deduplication writes only what is missing, within the queue's dedup window. Take copies as often as you can afford to lose, keep several, and restore one now and then on a spare machine: the first restore you run should not be the one during an outage.

Two things are never in a backup: ephemeral queues, which live in memory, and the settings, among them `QUEEN_RAFT_TOKEN` and `QUEEN_ENCRYPTION_KEY`. Keep those apart, with the same care.

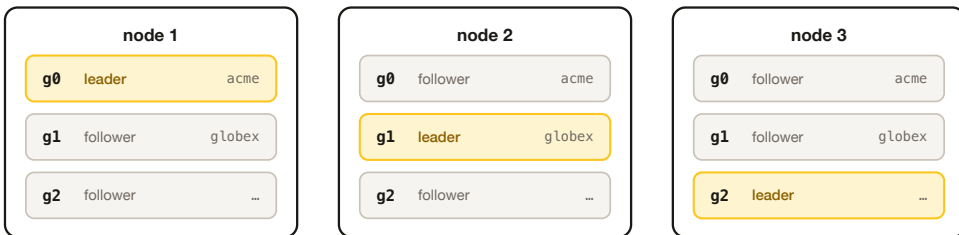
On the site: queenmq.com/manage/recovery. More there: [backups](#) and [restore](#).

6

Architecture

A Queen node is one process. Every write is decided once, on the leader, and then applied on every node. This chapter follows a write through the process, from the request to the disk.

The rule behind the whole design is: **one leader per tenant**. A tenant lives in one raft group, and one leader orders all its writes. So a transaction, even across ten queues, is one log entry: no coordinator, no two-phase commit. The price is that one tenant's writes do not spread over more nodes. To use more nodes for writes, run more raft groups (see Many tenants on one cluster): with as many groups as nodes, every node leads one.



One leader per tenant: an acme transaction across ten queues is one entry in g0's log.
One tenant's writes go through one leader; more groups spread tenants over more leaders.

Figure 6.1: One leader per tenant: three raft groups, each led on a different node.

6.1 Leader and followers

A raft group has one leader. The other nodes are followers, and every node has a full copy of the data.

The leader orders all the writes. A follower is not idle: it reads the requests of its own clients, prepares each one, and sends the leader only the prepared command. When the write is committed, the follower answers its client from its own copy. So the leader spends its time ordering and writing, not parsing JSON.

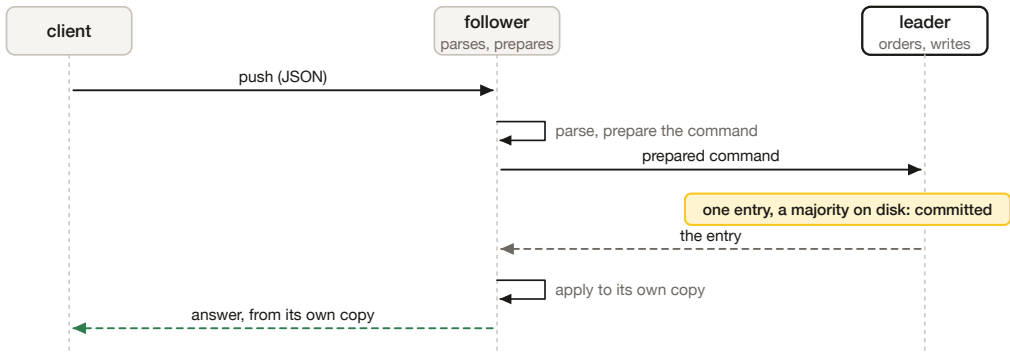


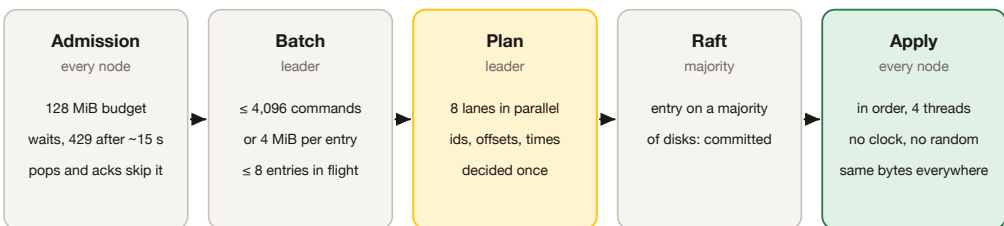
Figure 6.2: *A follower does the work of its own clients.*

A client can talk to any node. A read waits until that node has applied everything that was committed when the read started, so nobody reads behind a write they were already told about.

If the leader stops with SIGTERM, it hands its role to the most up-to-date follower, in a fraction of a second. If it crashes, the others wait one to two seconds and elect a new leader.

6.2 Admission, Batch, Plan, Apply

A write goes through four steps.



A write is answered after Apply on the node that took it.

Figure 6.3: *The four steps of a write.*

Admission. Every node has a memory budget for the writes waiting to be planned (128 MiB by default). A push takes its share before its body is read. When the budget

is full, the push waits, so a fast producer just slows down. After about 15 seconds it gets a 429, and the client tries again. Pops and acks never wait here. We added this after a broker filled 10 GB of memory with waiting requests and was killed.

Batch. The leader collects up to 4,096 commands (or 4 MiB) and plans them together, as one log entry. Up to eight entries are in flight at the same time. A retried request is found by its id and gets the answer it missed, so nothing is planned twice.

Plan. The planner decides what each command means right now, and writes the answer down as facts: “append these messages to partition 17, at offsets 4,210 to 4,309, at this microsecond”. Ids, offsets and times are chosen here, once. To use more cores, eight lanes plan in parallel: each lane owns a share of the partitions, and one last step plans what touches more than one lane.

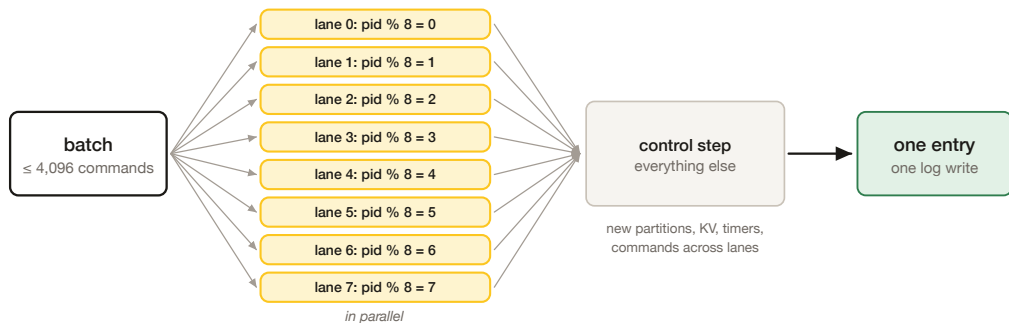


Figure 6.4: *Eight lanes plan in parallel; one step plans what crosses lanes.*

Apply. Every node executes the committed entries, in order. Apply reads no clock and no random number, so every node ends with the same bytes. Four threads apply the work of different partitions at the same time.

6.3 Raft internal protocol

The nodes agree on the log with Raft, through the openraft library. Raft decides who leads and when an entry is committed. An entry is committed when a majority of the nodes has written it and synced it to disk (fsync).

Queen has no separate Raft log: the queue log files are the log (see Storage). The nodes talk on port 7400, over plain HTTP, with a shared token. The leader sends a heartbeat every 100 ms. If the followers hear nothing for one to two seconds, they elect a new leader.

A leader that hears from no majority for 1.5 seconds stops logging new writes and keeps them waiting, in order. After 3 seconds it hands leadership to another node.

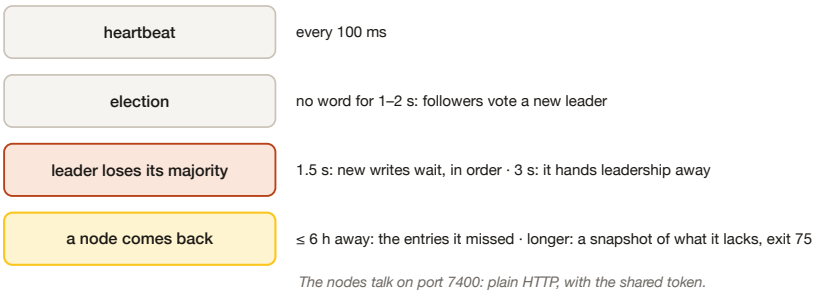


Figure 6.5: *The timings Raft runs on.*

A node that was away gets the entries it missed from the leader, who keeps them for 6 hours while its disk has room. After that, the leader sends a snapshot instead: since 2.6, only the part the node is missing. The node exits with code 75 and loads the snapshot when it starts again.

6.4 Memory state

Each node keeps its whole state in memory: queues, partitions, consumer positions, KV, timers, request ids, and the dedup records of recent pushes. It is loaded once, at boot.

Messages are not in memory. They stay in the queue log files and are read from there. So memory grows with the number of partitions and keys, and with the pushes of the dedup window, not with how many messages a queue keeps.

Message traces are kept apart, in their own file, and are read from disk.

6.5 Consumption engine

Pops, acks and leases do not go through the planner. The leader keeps each consumer group's positions and leases in memory, and gives out the oldest ready partition first.

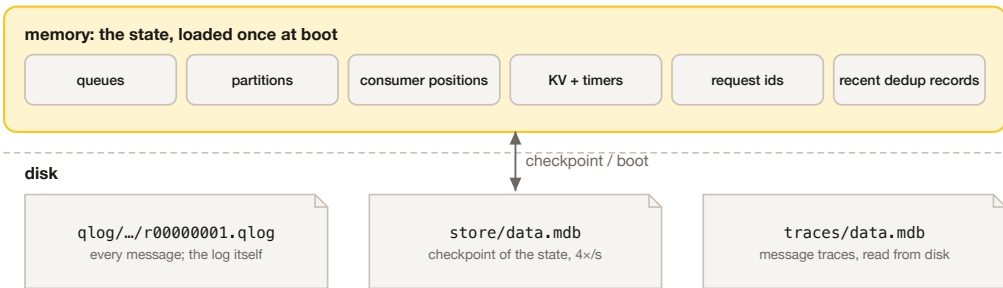
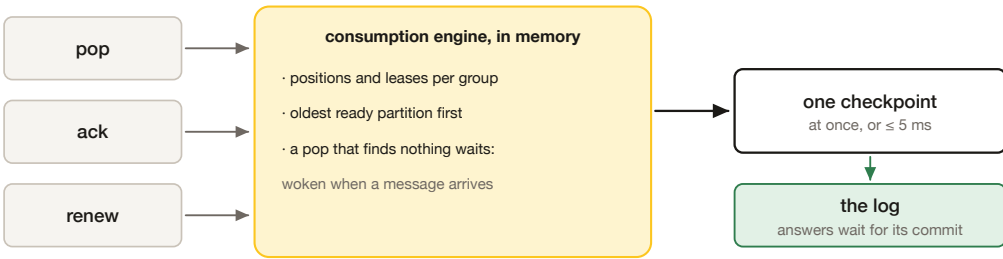


Figure 6.6: *What lives in memory, and what lives on disk.*

The positions that changed go into the log as one small checkpoint, at once when an answer is waiting and at least every 5 ms. A pop gets its answer only after that checkpoint is committed. A pop that finds nothing waits, and is woken when a new message arrives in its partition.



Many acks travel in one checkpoint; a new leader waits ~0.5 s before serving pops.

Figure 6.7: *Pops, acks and leases, served from the leader's memory.*

We moved consumption out of the planner because acks were waiting behind pushes: at 300,000 messages a second, an ack took more than 500 ms. Now many acks travel in one checkpoint.

A partition idle for 10 minutes leaves memory, and is loaded again with its next message. A new leader waits about half a second before it serves pops, so the same messages are never given out twice.

6.6 LMDB

The state in memory is saved to an LMDB file, `store/data.mdb`, four times a second. This is a checkpoint, and no client waits for it: it only decides how much of the log a restart has to replay.

Every value carries a checksum, and the node checks all of them at boot. Every 6 hours a background scan reads the whole store again, to find damage before a restart does.

Traces have their own LMDB file, `traces/data.mdb`.

6.7 Storage

A message is written once, in its queue's log files, and that write is what makes it durable: there is no second copy. We built it this way on purpose. An earlier design wrote every message twice, first in a consensus log and then in a segment file. Now the bytes that make a message durable are the bytes a pop reads back.

6.7.1 *The data directory*

Everything a node knows is in one directory:

```

/var/lib/queen/raft
qlog/                the messages: one log per queue
  LANES              the log layout, fixed at creation (and SHARDS, when set)
q0/                  the system log: entries that touch no queue
q5837314263283300749/ one directory per queue
  r000000001.qlog    a log file, up to 64 MiB, then the next one starts
  r000000001.qidx    the index of a closed file
  r000000001.qhsh     the dedup hashes of a closed file, for a fast restart
store/data.mdb       the state checkpoint (LMDB)
traces/data.mdb      message traces (LMDB)
raft/                cluster only: vote, members, what is kept for an away node
snapshots/           a snapshot from the leader, waiting for the next start
seg/ log/            almost empty, from an earlier design
local.db dash.db     this node's own metrics and dashboard data
groups/gl/           the same tree for each extra raft group

```

A closed file never changes again. Payloads of 512 bytes or more are compressed with `zstd`, when it saves at least a tenth. With thousands of queues, `QUEEN_QLOG_SHARDS` puts them all into a few shared logs, so a write syncs a few files instead of thousands.

6.7.2 What is inside a log file

A log file is a 32-byte header and then records, one after the other. Here is the start of a real file, after one push of `order-8891` with data `{"orderId":8891}`:

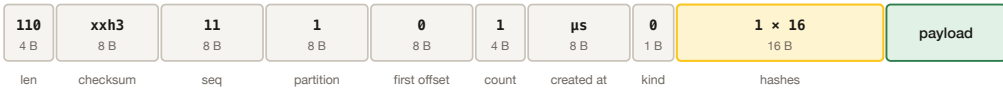
```
xxd qlog/q5837314263283300749/r000000001.qlog | head -10
00000000: 514e 514c 4f47 3100 0100 0000 0000 0000  QNQL0G1.....
00000010: 0b00 0000 0000 0000 0000 0000 0000 0000  ....
00000020: 6e00 0000 da25 b6dc 1ed0 878c 0b00 0000  n...%.....
00000030: 0000 0000 0100 0000 0000 0000 0000 0000  ....
00000040: 0000 0000 0100 0000 75e7 b953 935e 0600  .....u..S.^..
00000050: 00b9 7eb7 7dcc 2db6 dcbc ff49 4354 80b3  ..~.}.-....ICT..
00000060: c82d 0000 0000 01a1 6d65 4e4f 7001 bf27  .-.....meNOp..'
00000070: b56a 9334 d445 0a00 6f72 6465 722d 3838  .j.4.E..order-88
00000080: 3931 7b22 6f72 6465 7249 6422 3a38 3839  91{"orderId":889
00000090: 317d da01 0000 6b08 25f1 17ae 81e2 0b00  1}....k.%.....
```

The record starts at byte 0x20. Every number is little-endian.

Field	Bytes	In this record
len	4	110: the bytes after this field. A reader always knows where the next record starts.
checksum	8	An xxh3 of everything after it. A record torn by a crash fails it, and is cut.
seq	8	11: the log entry that wrote this record.
partition	8	1: the partition id.
first offset	8	0.
count	4	1 message.
created at	8	The time of the push, in microseconds.
kind	1	0: messages. The top bit says the payload is compressed.
hashes	16 a message	The hash of <code>order-8891</code> , for dedup.
payload	the rest	One frame per message: its length (45), flags, its 16-byte id, its transactionId and its data.

The next record (474 bytes, at 0x92) has the same seq, 11: it is the log entry itself, without the payload. An entry that touches several queues is written into each of

message record (offset 0x20 in the file)



payload: one frame per message

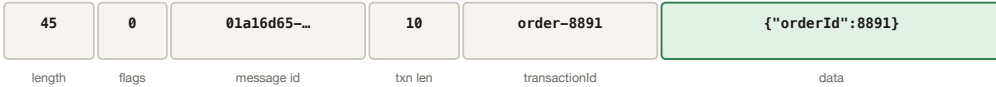


Figure 6.8: The message record of the dump, field by field.

their logs, and after a crash it is replayed only if every copy is there. This is how a transaction stays all or nothing on disk.

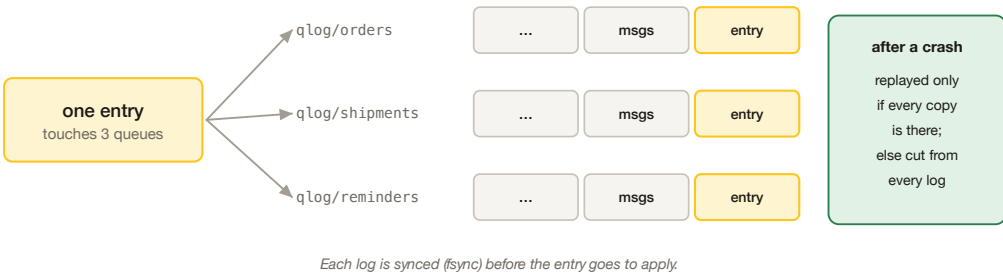


Figure 6.9: An entry in every log it touches; replayed only if every copy is there.

6.7.3 How retention works

Retention is decided once, on the leader, like every write. A scanner walks all the partitions (50,000 a second) and proposes what can go. The planner checks each proposal and logs watermarks: below one the messages are gone, below another their dedup records are gone. A partition that is empty, with no dead letter, no lease and no write for the cleanup age, is deleted.

Moving a watermark frees no disk by itself. Each node then frees its own files, one whole file at a time.

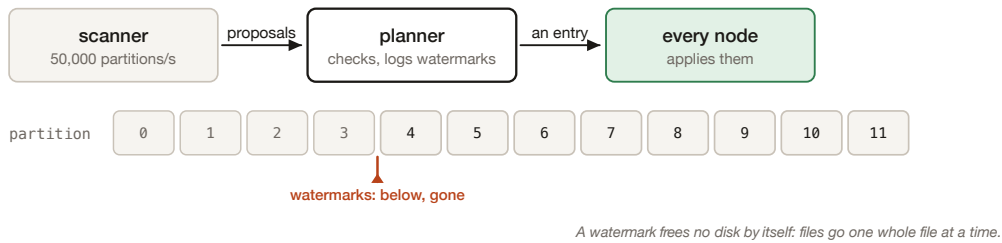


Figure 6.10: Retention decided on the leader, as watermarks.

6.7.4 Rewriting a file

A closed file with no live record is deleted. A closed file that is partly live is rewritten: the node copies only the live records into a new file, without stopping the writes, and swaps it in. A queue's own file is rewritten at its first dead message, a shared file when half of it is dead.

The active file is never touched, and neither is a file that recovery or a node that is behind still needs. One pass looks at up to 256 files, for at most 200 ms, and rewrites at most one file; while work is left, the next pass is four times bigger. An active file with data older than 10 minutes is closed early, so retention can reach it, when it holds a dead message or is an eighth full.

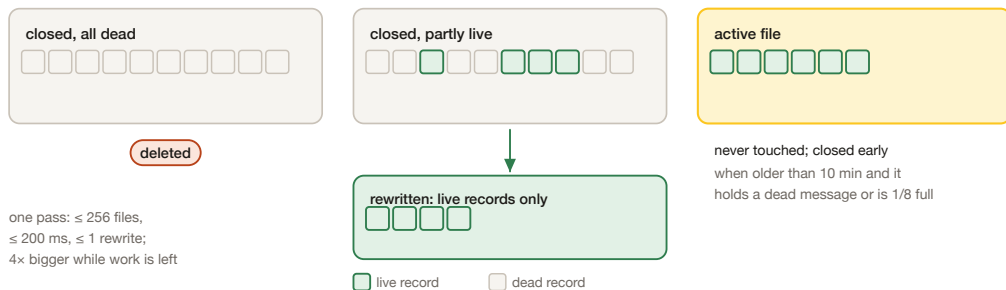


Figure 6.11: Reclaim: dead files deleted, partly live files rewritten.

WHY a file is the unit the disk gives back. Rewriting a file with only its live records gives back the space of the dead ones without waiting for the whole file to die, and the small passes keep the rewrite from slowing down the writes.

Conclusion

We hope this book helped everyone make the most of QueenMQ.

Check the website, queenmq.com, for updated information and for new versions of this book (queenmq.com/book).