Engineer

Queue

work left in order for something else to pick up

What it is

A queue is a holding place between two programs: one side puts messages or jobs in, the other takes them out when it is ready, usually oldest first. Most systems run several queues, one per kind of work. The program that adds work is the producer, the program that takes it is the consumer, and whatever stores the messages in between is the broker, which can be a dedicated server or a table in a database the system already has.

The arrangement lets the two sides stop waiting for each other. A web request that needs an email sent adds a message and answers the user at once; a worker picks the message up moments later, and if a hundred arrive together, it drains them at its own pace. A burst becomes a backlog, and a dependency becomes a delay.

Two ideas describe what a queue promises about delivery. The first is acknowledgement: a consumer tells the broker when it has finished with a message, and only then is the message removed. In the AMQP 0-9-1 model, which RabbitMQ implements, a message whose consumer dies before acknowledging goes to another consumer. The second is durability: a durable queue of persistent messages survives a broker restart, and a queue held only in memory does not.

Three neighbouring terms are worth keeping apart. A message queue carries data between services. A job queue carries units of work for background workers, each with its arguments, a state and a count of attempts. A log, the model behind Apache Kafka, looks like a queue from the outside but keeps events after they have been read, for a retention period chosen per topic, so several readers can replay the same history. Reading from a queue removes the message; reading from a log only moves the reader’s position.

A queue need not be a separate product at all. PostgreSQL can play the part with a table of jobs and a query that claims the next row while stepping over rows another worker has already locked. The manual names queue-like tables as the reason the `SKIP LOCKED` clause exists:

```
WITH next AS (
  SELECT id
    FROM jobs
   WHERE state = 'available'
   ORDER BY id
   LIMIT 1
     FOR UPDATE SKIP LOCKED
)
UPDATE jobs
   SET state = 'running',
       attempt = attempt + 1
  FROM next
 WHERE jobs.id = next.id
RETURNING jobs.id, jobs.args;
```

Use the concept in a decision

Put “What is a queue?” to work in a real decision. We can help relate the concept to the system you are building or buying. Numbers arriving where they are needed, on a schedule, without anyone copying a spreadsheet at the end of the month.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. One infrastructure plan for 14 departments — 14 research departments on one infrastructure plan

One infrastructure plan for 14 departments Get in touch →

Why it matters

A queue decides what a user waits for. Work placed on one happens after the response has gone, so a page that triggers an email, a PDF, a search reindex and a recalculation can answer in the time it takes to write four rows, and the four slow tasks finish behind it. For a buyer, that is the difference between a checkout that responds in a few hundred milliseconds and one that responds whenever the mail server is ready.

It also decides what survives a failure. Work started in a background thread lives in that process’s memory, and a deploy that restarts the process takes the work with it, leaving no trace that it was ever due. Work on a durable queue is a record instead: it can be retried, inspected and counted.

It separates two speeds, too. Producers move at the speed of incoming traffic; consumers move at the speed of the slowest service they call. A queue between them absorbs the difference, and its depth becomes one of the most truthful numbers a system can report. A queue that has grown for an hour says the consumers are losing ground, whatever the processor graph suggests.

The costs arrive with the benefits. Work now sits out of sight between being sent and being done. Delivery is almost always at least once, so a consumer has to tolerate seeing a message twice. Ordering is weaker than it looks. And a dedicated broker is one more server to run, secure, watch and upgrade.

What it looks like in practice

Sending mail, rebuilding a search index, generating documents and recalculating standings were all moved off the request path on one platform. Work handed to a goroutine vanishes when the process restarts in the middle of a deploy, so the work went onto a River job queue kept in PostgreSQL. The database was already the system of record, which makes enqueuing a job part of the same transaction as the data it concerns. What the platform gained is retries and a history someone can open and read; the stated limit is that a queue inside the database will not scale forever.

In a macOS application built over a C library, the queue was there to keep order. The library's blocking event loop emits events in a sequence that matters, because an edit has to arrive after the message it edits. Starting one task per event lets the runtime schedule them in any order, so an edit can overtake the message it refers to. The design that shipped has one producer thread, one consumer and one queue between them, followed by a single hop onto the main actor. Order holds because there is exactly one path through. The price is parallelism: every event passes through that one consumer.

A queue can also do harm. In one macOS application, a burst of core log events was retained in an unbounded main stream and copied to subscribers whose consumers did not use those events. Diagnostic reports recorded 111 GB of compressed pages on a machine with 36 GB; a separate footprint sample recorded growth of about 41 MB per second during a connection retry loop. The event path was given explicit bounds, with subscriptions filtered by event type and an explicit overflow policy. Repetitive diagnostic events may be dropped and

affected state refreshed instead of keeping unlimited events. Architecture checks and application-budget tests defend that design, but the record does not establish a measured post-fix memory figure or a production benchmark.

Where it goes wrong

Queue trouble surfaces late and somewhere else, because the work is out of sight by design.

The backlog grows until memory or disk gives out. A queue with no limit accepts work faster than it is done and keeps the difference. Give every queue a maximum length and decide ahead of time what happens at that length: slow the producer, refuse new work, or drop the oldest. Then watch two numbers, the depth and the age of the oldest message, because a short queue of very old messages is as bad as a long one.

The same message is processed twice. A consumer finishes the work and dies before it acknowledges; the broker, correctly, delivers the message again. RabbitMQ's documentation tells consumers to expect redeliveries and to be written with idempotence in mind. Give each message an identifier and record the identifiers already handled, in the same transaction as the work itself. A note on this site about selling limited stock without overselling builds its writer around exactly that assumption.

Messages arrive in a different order from the one they were sent in. Parallel consumers, prefetching, and retries that put a failed message behind newer ones all reorder work. When order matters, route everything about one account or one order through the same partition or the same consumer. Kafka does this by key, at the cost that one busy key moves only as fast as one consumer.

One bad message blocks or loops forever. A message whose payload crashes the consumer every time is requeued, delivered again, and crashes it again. Cap the attempts, and move a message that has used them all to a separate place where a person can look at it. RabbitMQ calls this dead-lettering and applies it to rejected messages, expired ones and messages that would overflow a length limit.

Work disappears on restart. A channel, a list in memory or a background thread is a queue that keeps nothing across a restart. In AMQP, a durable queue alone is not enough either: the messages must be published as persistent too. Decide per kind of work whether losing it is acceptable.

The database and the queue disagree. A service writes a row, then publishes a message, and crashes between the two; now one exists without the other. Either keep the queue in the same database, so that a job becomes visible to workers only when its transaction commits, which is how River behaves, or write the message into an outbox table in that transaction and publish it from there.

Further reading

- AMQP 0-9-1 Model Explained — exchanges, queues, bindings and acknowledgements, set out by RabbitMQ.
- Consumer Acknowledgements and Publisher Confirms — when a broker considers a message delivered, and why redelivery happens.
- Dead Letter Exchanges — the conditions that move a message aside and where it goes.
- The Locking Clause of SELECT — `FOR UPDATE` and `SKIP LOCKED`, and the queue-like tables they suit.
- River: Getting started — a job queue that stores its jobs in the application's own database.
- Apache Kafka: Introduction — topics, partitions and retention in the log model.

Glossary

Choose for your actual constraints

Bring a representative dataset, the questions it must answer and the current bottleneck. We can agree how to measure quality, throughput and recovery.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a queue?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/queue/