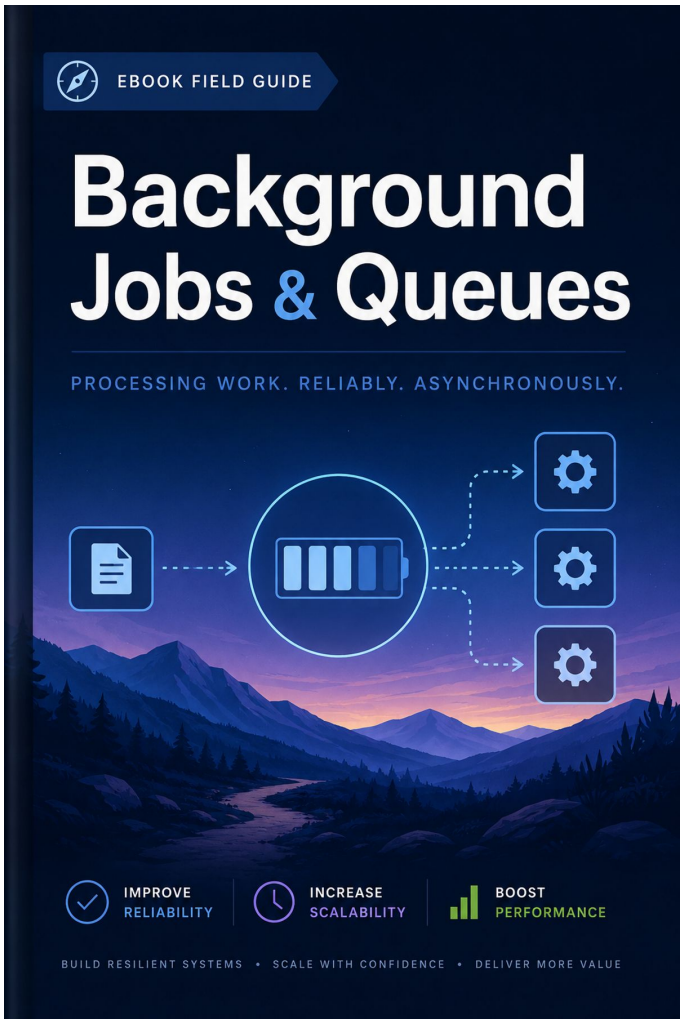

Background Jobs & Queues

The part of every real app nobody teaches well: workers, queues, retries, idempotency, scheduling, and dead-letter handling — built up from first principles into a system you can trust.

Roger Stringer · rogerstringer.com

June 19, 2026

Contents

Why Every Real App Needs a Queue

Here's a pattern you'll recognize: a user clicks "sign up," your code creates the account, sends a welcome email, generates avatar thumbnails, syncs them to your CRM, and *then* returns the page. The request takes four seconds. The email provider has a slow morning and it takes eleven. One of those calls fails and the whole signup errors out — even though the account was created fine. You've tied the user's experience to a pile of work that has no business happening inside their request.

The fix is a queue. The request does the one thing that must happen now — create the account — and *enqueues* the rest as background jobs: send the email, make the thumbnails, sync the CRM. The user gets their page in 200ms. The slow, failable work happens out of band, on its own time, with its own retries. The request got faster, more reliable, and simpler, all at once.

The symptoms that tell you you're missing a queue:

- **Slow requests doing work the user doesn't need to wait for** — emails, image processing, third-party API calls, report generation.
- **Requests that fail because something downstream failed** — the signup that errors because the CRM was down, even though the signup itself worked.
- **Work that should happen later** — a reminder in 24 hours, a nightly cleanup, a subscription that renews next month.
- **Spiky work you want to smooth out** — a thousand people trigger an export at once and you'd rather process them steadily than melt the server.

Every one of these is the same shape: *work that shouldn't block the request and can tolerate happening slightly later.* That's the definition of a background job, and a queue is the machinery that runs them.

This guide builds that machinery from first principles — starting with the simplest queue that works (a database table and a loop, which pairs with the Postgres for App Developers (https://rogerstringer.com/guides/postgres-for-app-developers) guide), then layering on what makes it trustworthy: workers, retries, idempotency, dead-letter handling, scheduling, and observability. By the end you'll understand queues well enough to use any of the real ones wisely, or run your own. It's the piece almost no tutorial teaches well, and it's in nearly every real app. Let's start with the mental model.

The Mental Model

A queue system is three parts, and once you see them the whole field stops being intimidating: a **producer**, a **queue**, and a **worker**.

- The **producer** is your app code deciding there's work to do later. When the signup handler says "send a welcome email," it doesn't send the email — it *enqueues a job*: a little record that says "send-welcome-email to user 42." Then it moves on.
- The **queue** is a durable list of those jobs, waiting to be done. Durable is the key word: the jobs are written down somewhere persistent (a database, Redis, a broker) so they survive a crash. A queue that loses its jobs when the process restarts isn't a queue, it's a wish.
- The **worker** is a separate process whose whole life is: take the next job off the queue, do it, mark it done, repeat. It runs independently of your web app — different process, often a different machine — chewing through work at its own pace.

That's the entire architecture. Your web app produces jobs and returns fast; your workers consume jobs and do the slow stuff; the queue sits between them, holding the work durably so nothing is lost in the handoff.

The mental shift this requires is from **synchronous** to **asynchronous** thinking, and it's worth naming because it trips people up. Synchronous code does a thing and waits for the result before continuing — the natural way you write. Asynchronous-via-queue says "this will happen, but not now, and not here" — you hand off the work and trust it'll get done, without waiting. That trust has to be earned by the machinery: *will* it get done, even if the worker crashes mid-job? What if it gets done twice? What if it fails? Those questions — durability, retries, idempotency — are the rest of this guide, and they're exactly what separates a real queue from a toy.

The payoff of the producer/queue/worker model is that it decouples *deciding to do work* from *doing work*. Your request path stays fast and simple; the heavy lifting moves to workers you can scale, retry, and reason about separately. Let's build the simplest possible version, so the three parts stop being abstract.

Your First Queue

Before reaching for any queue library, build one by hand — it's a couple dozen lines, it teaches you what every queue tool is doing under the hood, and for a lot of apps it's genuinely all you need. We'll use Postgres, because you already have it.

The queue is a table:

```
CREATE TABLE jobs (
  id          bigserial PRIMARY KEY,
  type        text NOT NULL,
  payload     jsonb NOT NULL,
  status      text NOT NULL DEFAULT 'pending',  -- pending | done | dead
  run_at      timestamptz NOT NULL DEFAULT now(),
  attempts    int NOT NULL DEFAULT 0,
  created_at  timestamptz NOT NULL DEFAULT now()
);
```

Producing a job is an insert — this is what your signup handler calls instead of sending the email inline:

```
async function enqueue(type: string, payload: unknown) {
  await db.query(`INSERT INTO jobs (type, payload) VALUES ($1, $2)`,
    [type, JSON.stringify(payload)]);
}

// in the signup handler:
await enqueue("send_welcome_email", { userId: 42 });
```

The worker claims a job, runs it, and marks it done. The one subtle line is how it *claims* a job without two workers grabbing the same one — `FOR UPDATE SKIP LOCKED`, the single most important trick in database-backed queues:

```
async function workOnce() {
  const { rows } = await db.query(`
    UPDATE jobs SET status = 'running', attempts = attempts + 1
    WHERE id = (
      SELECT id FROM jobs
      WHERE status = 'pending' AND run_at <= now()
      ORDER BY id
      FOR UPDATE SKIP LOCKED      -- claim a row no other worker has locked
      LIMIT 1
    )
    RETURNING *;
  `);
  const job = rows[0];
  if (!job) return false;            // nothing to do

  try {
    await handlers[job.type](job.payload);
    await db.query(`UPDATE jobs SET status = 'done' WHERE id = $1`, [job.id]);
  } catch (err) {
    await db.query(
      `UPDATE jobs SET status = 'pending', run_at = now() + interval '1 minute'
       WHERE id = $1`, [job.id]);
  }
  return true;
}

// the worker process: loop forever, sleeping when idle
while (true) {
  const did = await workOnce();
  if (!did) await sleep(1000);
}
```

That's a working queue. It's durable (jobs are rows that survive a crash), safe for multiple workers (`SKIP LOCKED` means each picks a different job), and it already retries on failure (a failed job goes back to pending with a delay). Run that worker as a separate process — on the Boring Stack (https://rogerstringer.com/guides/the-

boring-stack), it's just another app Coolify runs — and your web app's signup handler returns instantly.

This Postgres-backed queue carries a surprising amount of real load, and it has a lovely property: your jobs live in the same database as your data, so enqueuing a job and updating a row can happen in the *same transaction* — more on why that's powerful later. But it's worth knowing when you'd reach for something else, which is the next chapter.

Choosing a Backend

You just built a database-backed queue, and the natural question is whether you should've used Redis, or RabbitMQ, or one of the dozen hosted queue services. The honest answer is *usually not yet* — but knowing the tradeoffs keeps you from both over- and under-engineering.

Database-backed (what we built). Your jobs are rows in Postgres. The wins are real: you already run it, so it's zero new infrastructure; jobs are durable by default; and — the killer feature — you can enqueue a job *in the same transaction* as the data change that triggered it, so they either both happen or neither does (no "created the order but lost the confirmation email"). The cost is throughput: polling a table with `SKIP LOCKED` is fine into the thousands of jobs a minute, but not millions. For the overwhelming majority of apps, a database queue is the right answer, and reaching past it is premature. Libraries like pg-boss (Node) wrap exactly this pattern.

Redis-backed. Redis is fast and purpose-built for this kind of work, and the popular libraries (BullMQ in Node, Sidekiq in Ruby) are Redis-based. You reach for it when database polling becomes a measured bottleneck, or when you want features — delayed jobs, rate limiting, priorities — the libraries hand you out of the box. The cost is another piece of infrastructure, and Redis is less durable than a database by default (it's memory-first), so you configure persistence carefully or accept that a hard crash can lose recently-enqueued jobs. You also lose the transactional-enqueue trick — your job and your data now live in different systems.

Dedicated brokers (RabbitMQ, SQS, Kafka, and friends). Purpose-built messaging systems with serious throughput, routing, and delivery guarantees. You reach for these at real scale, or for cross-service messaging where many services produce and consume — not for "send the welcome email" in a single app. They're powerful and they're a lot of operational weight; bringing one in for a typical web app is a classic case of solving a problem you don't have.

The rule of thumb that keeps you sane: **start with the database queue, move to Redis when you've measured a real throughput or feature need, and only consider a broker at genuine scale or for multi-service messaging.** Most apps live their whole lives happily on the first option — the same "boring wins" instinct from the Boring Stack (https://rogerstringer.com/guides/the-boring-stack) guide. Whatever the backend, the worker that consumes the jobs has the same concerns, so let's look at it properly.

Workers

The worker is where jobs actually get done, and getting it right — concurrency, shutdown, resilience — is most of what separates a queue that works in a demo from one that survives production. Let's go deeper than the loop from chapter three.

A worker is a long-running process, separate from your web app. That separation is the point: it can crash, restart, or be scaled without touching your web tier, and it runs on its own schedule. On the Boring Stack (https://rogerstringer.com/guides/the-boring-stack) it's a second app Coolify runs from the same repo with a different start command (`node worker.js` instead of `node server.js`). Same code, same database, different job.

Concurrency — doing several jobs at once. One worker doing one job at a time is often too slow. Two ways to add concurrency: run *multiple worker processes* (simplest — just start three of them; `SKIP LOCKED` ensures they don't collide), or process *multiple jobs at once within one worker* (claim a batch, `Promise.all` them). Start with multiple processes — it's simpler, uses multiple CPU cores, and one wedged job doesn't block the others. Tune the count to the workload: I/O-bound jobs (API calls, emails) can run lots concurrently; CPU-bound jobs (image processing) shouldn't exceed your core count.

Graceful shutdown — the part everyone forgets, and the cause of mysterious half-done work. When you deploy, the platform sends your worker a `SIGTERM` and then, a few seconds later, kills it. If the worker is mid-job when it's killed, that job is left in limbo — marked running, half-done, maybe with side effects already fired. The fix is to handle the signal: stop claiming *new* jobs, let the current one finish, then exit.

```
let shuttingDown = false;
process.on("SIGTERM", () => { shuttingDown = true; });

while (!shuttingDown) {
  const did = await workOnce();
  if (!did) await sleep(1000);
}
// loop exits only between jobs, so the in-flight job completed cleanly
process.exit(0);
```

Graceful shutdown plus a sane claim mechanism is what makes deploys safe — you can ship a new worker version any time without dropping work on the floor.

Crash resilience. Workers *will* die mid-job sometimes — OOM, a hard kill, a bug. The job was marked `running`; now nothing's working on it, and it'll sit there forever unless you handle it. The standard fix is a timeout: a job stuck in `running` longer than any job should take is presumed dead and reset to `pending` so another worker reclaims it. (This is exactly the crash-and-reclaim the Running the Fleet (https://rogerstringer.com/guides/running-the-fleet) guide builds for agents — same problem, same shape.)

A worker that's concurrent, shuts down gracefully, and reclaims crashed jobs is one you can trust to run unattended. But "do the job" glosses over the hardest question: what happens when the job fails?

Retries & Backoff

Jobs fail. The email provider times out, the third-party API returns a 503, the network blips. The entire reason to run work in a queue rather than inline is that the queue can *retry* — so getting retries right is most of a queue's value. Done well, transient failures become invisible; done badly, they become outages of their own.

Retry, but not forever. A failed job should go back on the queue to try again — but a job that's *always* going to fail (bad data, a permanent 404) will retry endlessly, burning resources and never succeeding. So every job gets a **max attempts** limit. Try a few times; if it still fails, stop and set it aside (that's the dead-letter queue, next chapter). The `attempts` counter from our table is exactly for this.

Back off between retries — don't hammer. Retrying a failed job *immediately*, over and over, is how you turn a brief downstream hiccup into a self-inflicted denial-of-service: the provider is struggling, and you respond by pounding it as fast as you can. **Exponential backoff** is the fix — wait longer after each failure: 1 second, then 2, then 4, then 8. The struggling service gets breathing room, and a transient blip resolves itself without you making it worse.

```
// on failure, schedule the next attempt with exponential backoff
const delaySeconds = Math.min(2 ** job.attempts, 3600);   // cap at 1 hour
await db.query(
  `UPDATE jobs SET status = 'pending', run_at = now() + ($2 || ' seconds')::interval
   WHERE id = $1`,
  [job.id, delaySeconds]
);
```

That `run_at` in the future is why the worker's claim query checks `run_at <= now()` — a backed-off job is simply invisible until its time comes.

A few refinements that matter in practice:

- **Add jitter.** If a thousand jobs all fail at once (the provider went down) and all back off by exactly 4 seconds, they all retry at exactly the same instant — a thundering herd that knocks the recovering provider over again. Add a little randomness to each delay so they spread out.
- **Distinguish retryable from permanent failures.** A 503 is worth retrying; a 400 "invalid email" never will be. If you can tell the difference, fail the permanent ones immediately instead of wasting retries — dead-letter them straight away.
- **Cap the backoff.** Exponential growth gets absurd fast (2^20 seconds is twelve days). Cap it at something sane like an hour so a job retries on a reasonable cadence instead of disappearing into the future.

Retries with backoff turn the messy reality of unreliable dependencies into something your system absorbs quietly. But retries collide with a deeper problem the moment a job has *side effects* — because retrying means running it more than once.

Idempotency

This is the hardest chapter in the guide, and the concept that separates people who've run queues in production from people who've only read about them. Here's the problem in one sentence: **if a job can be retried, it can run more than once — so every job must be safe to run twice.** That property is called idempotency, and ignoring it is how a retry sends a customer three welcome emails or charges their card twice.

Why does a job run more than once? Not just from retries on failure. The nastier case: a worker finishes the job's *work* — the email is sent, the charge is made — and then crashes *before* it can mark the job done. The job is still `running`; your timeout reclaims it; another worker runs it again. The side effect already happened, but the queue doesn't know that. **At-least-once delivery** — the guarantee almost every real queue actually provides — means "this job will run, possibly more than once." (The "exactly-once" some systems advertise is mostly marketing over the same at-least-once machinery plus idempotency; we'll be honest about that two chapters from now.)

So you design jobs to tolerate re-running. The techniques, roughly in order of how often you'll reach for them:

- **Make the operation naturally idempotent.** "Set the user's status to active" is safe to run ten times — the result is identical. "Increment the user's login count" is not. Where you can, write the job so its effect is a *set*, not a *change*. This is the cleanest fix and often just a reframing.
- **Check before you act.** Before sending the welcome email, check whether you've already sent it (a `welcome_email_sent_at` column). Before creating the record, check whether it exists. The job becomes "do this *if it hasn't been done*," which is safe to repeat.
- **Use an idempotency key with the downstream system.** The good third-party APIs — Stripe is the canonical example — accept an *idempotency key* on each request: send the same key twice and they perform the operation once. Pass your job's ID as the key and a double-run charges the card once, because the *provider* dedupes it. When the side effect is external, this is the gold standard.
- **Dedupe with a unique constraint.** If a job inserts a row, a unique constraint on a natural key turns a double-insert into a harmless error you can catch and ignore. The database enforces "only once" for you. (The Postgres for App Developers (https://rogerstringer.com/guides/postgres-for-app-developers) guide goes deep on exactly this.)

The mindset to adopt, and to write into your project's conventions: **assume every job will run at least twice, and design so the second run is harmless.** It feels paranoid until the night a worker crashes at exactly the wrong moment and idempotency is the only reason your customers didn't notice. The jobs that *aren't* safe to repeat — anything that moves money or messages a person — are exactly the ones where getting this wrong is most visible, so spend your care there.

Idempotency handles the jobs that run too many times. The next chapter handles the ones that, after all their retries, still won't run at all.

Dead-Letter Queues

A job has retried its maximum number of times and still fails. Now what? If you just delete it, you've silently lost work — and silently losing work is the worst thing a queue can do, because nobody finds out until a customer asks where their thing went. If you leave it retrying forever, it clogs the queue and wastes resources. The right answer is a **dead-letter queue**: a place where exhausted, failed jobs go to be *seen and dealt with*, rather than lost or looping.

Mechanically it's simple — in our table-based queue, it's a status:

```
if (job.attempts >= MAX_ATTEMPTS) {
  await db.query(
    `UPDATE jobs SET status = 'dead', last_error = $2 WHERE id = $1`,
    [job.id, String(err)]);
} else {
  // back off and retry, as before
}
```

Now a job that's truly broken lands in `status = 'dead'` with its error recorded — out of the retry loop but *not gone*. That distinction — out of the loop, not gone — is the whole point. The dead-letter queue is a visible pile of "things that need a human," and a pile you can actually act on:

- **You can see it.** A count of dead jobs is one of the most important health metrics you have (more in the observability chapter). A growing dead-letter pile means something is systematically broken — a provider changed their API, a bad deploy, a data problem — and it's a signal you want loud, not buried.
- **You can inspect *why*.** Because you recorded `last_error`, you can look at a dead job and understand what failed. Often the dead-letter queue is the first place you learn about a bug — a class of jobs all failing the same way is a stack trace waiting to be read.
- **You can replay.** Once you've fixed the cause — patched the bug, the provider came back — you flip those dead jobs back to `pending` and they run again. The work was never lost; it was *parked*, waiting for you to make it runnable. This is the recovery the Running the Fleet (https://rogerstringer.com/guides/running-the-fleet) guide describes for blocked agent tasks: a failure that's a state you can resolve, not a loss.

The principle underneath: **a failure should never vanish silently.** Either a job succeeds, or it's retrying, or it's sitting in the dead-letter queue where you can see it, understand it, and replay it. There's no fourth state where work just disappears. A queue without a dead-letter queue is a queue that loses things, and losing things is the one job a queue absolutely cannot have. With failures handled, let's add the other big capability: making jobs happen on a schedule.

Scheduling & Cron Jobs

So far jobs run as soon as a worker can get to them. But a huge category of background work is *time-based*: send the reminder in 24 hours, run the cleanup every night, retry the subscription on renewal day. A queue with scheduling handles all of it — and you've already built most of the mechanism.

Delayed jobs — 'do this later' — you already have. Remember `run_at` and the worker's `WHERE run_at <= now()`? That's a scheduler. Enqueue a job with a future `run_at` and it simply waits, invisible to workers, until its time arrives:

```
async function enqueueAt(type: string, payload: unknown, runAt: Date) {
  await db.query(`INSERT INTO jobs (type, payload, run_at) VALUES ($1, $2, $3)`,
    [type, JSON.stringify(payload), runAt]);
}

// send a reminder tomorrow
await enqueueAt("send_reminder", { userId: 42 },
  new Date(Date.now() + 24 * 3600 * 1000));
```

The same `run_at` that powers backoff powers "remind me tomorrow" — one mechanism, two uses. Delayed jobs build reminders, drip campaigns, scheduled publishing, and timeouts ("if they haven't verified their email in 48 hours, nudge them").

Recurring jobs — 'do this every night' — need a scheduler. Something has to *create* the nightly-cleanup job each night. Two common patterns:

- **A cron process that enqueues.** A small scheduler runs on a cron (system cron, or a `setInterval` in a dedicated process) and its only job is to *enqueue* the real jobs: at midnight it enqueues `nightly_cleanup`, every hour it enqueues `refresh_stats`. The scheduler stays dumb — it produces jobs; the workers do them — which keeps the heavy lifting in the queue where retries and observability already live.
- **Self-rescheduling jobs.** A recurring job, as its last step, enqueues its own next run. Simple, but if one run dies without rescheduling, the chain stops — so a supervising cron is the more robust pattern.

Two things to get right with recurring work:

- **Don't double-schedule.** If two scheduler instances both run (you scaled it by accident, or a deploy overlapped), you'll enqueue two of every nightly job. Run exactly one scheduler, or dedupe with a unique key on (job type + time window) so the second insert is harmlessly rejected — the unique-constraint trick from the idempotency chapter, again.
- **Recurring jobs are still jobs.** They retry, they can be idempotent, they can dead-letter — all the machinery applies. A nightly cleanup that fails should retry and, if it keeps failing, land in the dead-letter queue where you'll see it, not silently skip a night.

With scheduling, your queue handles not just "later" but "at a specific time" and "on a repeating cadence" — covering nearly every time-based need an app has. But running jobs in parallel, on a schedule, raises the thorniest questions in the whole field: ordering and concurrency.

Ordering & Concurrency Hazards

This is the chapter where we're honest about what queues *can't* cleanly promise, because the failure modes here are subtle, they only show up under load, and believing the marketing instead of the reality is how people get burned.

Ordering: jobs do not run in the order you enqueued them. With multiple workers running concurrently, job B can finish before job A even if A was enqueued first — A might be slower, or land on a busier worker. If your jobs are independent (two unrelated emails), this is fine and you should *want* the parallelism. If they're not — "apply discount" must happen before "charge card" — you have a problem the queue won't solve for you. The fixes: **don't depend on order** (design jobs to be independent, the best option), **chain them** (job A enqueues job B as its last step, so B can't start until A is done), or **serialize a key** (route all jobs for one user through a single worker or a per-user lock so they can't overlap). What you cannot do is enqueue A then B and assume they run in that order. They won't, reliably.

Concurrency hazards: the same races as any parallel system. Two jobs touching the same row at the same time can clobber each other — the exact vote-counting race the Hypermedia Stack (https://rogerstringer.com/guides/the-hypermedia-stack) and Postgres (https://rogerstringer.com/guides/postgres-for-app-developers) guides describe, now multiplied across many workers. Two `add_loyalty_points` jobs for the same user, running together, both read 100 and both write 110, losing five points. The fixes live in the database: atomic operations (`UPDATE points = points + 10`, which the database serializes), row locks (`SELECT … FOR UPDATE`), or designing the job to be idempotent and order-independent. Background jobs don't escape concurrency control — they make it more important, because now the parallelism is the whole point.

The 'exactly-once' myth. Some queue systems advertise exactly-once delivery. Treat the claim with suspicion. In a distributed system where workers can crash at any instant, you genuinely cannot guarantee a job's side effects happen exactly once — the worker can always die in the gap between *doing the work* and *recording that it did*. What these systems actually provide is at-least-once delivery plus deduplication that makes it *look* exactly-once for well-behaved jobs. The honest engineering position: **assume at-least-once, and make your jobs idempotent** (chapter seven). Then "ran twice" is harmless and you don't need a guarantee nobody can actually keep. Designing for at-least-once is robust; relying on exactly-once is trusting a promise that breaks at the worst moment.

The theme: a queue gives you parallelism, and parallelism gives you ordering and race problems for free. You handle them with independence, idempotency, and the database's concurrency tools — not by wishing the queue would run things one at a time, which would throw away the throughput you built it for. The mature move is to embrace the concurrency and design jobs that don't care about it. Which all gets much easier to manage once you can actually *see* what your queue is doing.

Observability

A queue is, by design, work happening out of sight — which means without observability it's a black box, and a black box full of your users' unfinished work is a terrifying thing to operate. The good news: a few simple metrics tell you almost everything, and you've already got the data, because your jobs are rows you can count.

The numbers that matter, queryable straight from the jobs table:

```
SELECT status, count(*) FROM jobs GROUP BY status;
```

- **Queue depth (pending jobs).** How much work is waiting. A small, stable number is healthy. A number that's *climbing* means jobs are arriving faster than workers can process them — you're falling behind, and you need more workers or faster jobs. This is the single most important queue metric; watch it like you'd watch CPU.
- **Dead-letter count.** How many jobs have failed permanently. This should be near zero. Any sustained growth means something is systematically broken — it's the alarm that catches bad deploys and broken integrations. Alert on it.
- **Job age / latency.** How long jobs wait before running, and how long they take. If the welcome email that should go out in seconds is taking twenty minutes, queue depth is too high or a job type is too slow. Latency creeping up is the early warning before users complain.
- **Throughput and failure rate.** Jobs done per minute, and the fraction that fail before succeeding. A rising failure rate (even if they eventually succeed on retry) means a dependency is getting flaky — worth knowing before it tips into dead-lettering.

What to actually do with these:

- **Put them on a dashboard.** Queue depth and dead-letter count over time, where you'll see them. On the Boring Stack (https://rogerstringer.com/guides/the-boring-stack) this can be a small page that runs those queries; you don't need a heavy observability platform for one app's queue.
- **Alert on the two that signal trouble:** dead-letter count growing (something's broken) and queue depth climbing without recovering (you're falling behind). Those two cover most of the ways a queue silently goes wrong.
- **Keep job history long enough to debug.** Don't delete jobs the instant they're done — keep completed and dead jobs around for a while so that when something's wrong, you can see what ran, what failed, and why. The `last_error` on dead jobs is often your first sight of a bug.

The principle mirrors the Running the Fleet (https://rogerstringer.com/guides/running-the-fleet) guide's flight-recorder idea: the work happens autonomously, so you need a window into it — not to micromanage every job, but to know at a glance whether the system is healthy and to have a trail when it isn't. A queue you can see is a queue you can trust to run unattended. Let's pull all of it together.

Putting It Together

You've built every piece. Let's assemble them into the shape of a real, production job system, so you can see how they fit — and so you have a checklist for any queue you build or adopt.

The full picture, end to end:

- **Your web app produces jobs** instead of doing slow work inline. The signup handler creates the account and enqueues `send_welcome_email`, `generate_thumbnails`, `sync_to_crm` — and because the queue is in Postgres, it enqueues them *in the same transaction* as creating the account, so the account and its follow-up work commit together or not at all. The request returns in milliseconds.
- **The queue holds jobs durably** as rows, each with a type, payload, status, `run_at`, and attempt count — surviving crashes, supporting delayed and scheduled work through `run_at`.
- **Workers consume jobs** — several processes, claiming with `FOR UPDATE SKIP LOCKED` so they never collide, processing concurrently, shutting down gracefully on deploy, and reclaiming jobs from crashed siblings via a timeout.
- **Failures retry with capped exponential backoff and jitter**, distinguishing transient failures (retry) from permanent ones (dead-letter immediately).
- **Every job is idempotent**, because delivery is at-least-once and a worker can always crash between doing the work and recording it — so a second run is harmless.
- **Exhausted jobs land in the dead-letter queue**, visible and replayable, never silently lost.
- **A scheduler enqueues recurring work**, and `run_at` handles delays — reminders, nightly jobs, timeouts.
- **A dashboard shows queue depth and dead-letter count**, alerting when work backs up or starts failing.

Put that way, a job system is just those eight pieces working together, and you can build the whole thing on Postgres and a couple of worker processes — no exotic infrastructure, the Boring Stack (https://rogerstringer.com/guides/the-boring-stack) all the way down. The deploy shape is simple too: the same repo, run once as your web app and once (or a few times) as a worker, both pointed at the same database.

When to graduate to a library or a different backend: when you've measured that database polling is a real bottleneck, or when you want features (priorities, rate limiting, fancy scheduling) that a library like BullMQ or pg-boss hands you for free. But — and this is the payoff of building it yourself — you'll now *understand* what those libraries do, so you'll configure them wisely and debug them when they misbehave, instead of treating them as magic. A queue you understand from first principles is one you can trust, scale, and fix.

That's background jobs, end to end: the thing nearly every real app needs and almost no tutorial teaches whole. Your requests are fast because the slow work moved out of band; your work is reliable because it retries, dedupes, and dead-letters; and you can see the whole thing running. Pair it with Postgres for App Developers (https://rogerstringer.com/guides/postgres-for-app-developers) for the database tricks underneath it, and the Boring Stack (https://rogerstringer.com/guides/the-boring-stack) for where the workers live. Now go move that slow work off the request path, where it never belonged.

About Roger

I'm Roger Stringer — I build things, break them, and write up what I learned so you don't have to learn it the hard way. These Field Guides come straight out of that work.

Working on something bigger? I take on a handful of fractional CTO (https://rogerstringer.com/guides/the-fractional-cto-field-guide) engagements — helping founders and teams set technical direction, build AI-powered workflows, and actually ship the hard parts. If you're wrestling with the kind of problem this guide covers and want someone in your corner who's done it before, that's exactly what I help with. Drop me a line. (mailto:roger.stringer@hey.com)

And if a guide helped, got something wrong, or you just want to compare notes, I'd love to hear from you:

- **Email** — roger.stringer@hey.com (mailto:roger.stringer@hey.com)
- **X** — @freekrai (https://x.com/freekrai)
- **GitHub** — github.com/freekrai (https://github.com/freekrai)
- **LinkedIn** — linkedin.com/in/rogerstringer (https://www.linkedin.com/in/rogerstringer)

New guides go up as I hit problems worth documenting — follow along wherever suits you.