What We Learned After Finding 7 Forgotten Jobs Running for 5 Years

We recently started a small project to clean up how parts of our systems communicate behind the scenes at Buffer.

Some quick context: we use something called SQS (Amazon Simple Queue Service. These queues act like waiting rooms for tasks. One part of our system drops off a message, and another picks it up later. Think of it like leaving a note for a coworker: “Hey, when you get a chance, process this data.” The system that sends the note doesn’t have to wait around for a response.

Our project was to perform routine maintenance: update the tools we use to test queues locally and clean up their configuration.

But while we were mapping out what queues we actually use, we found something we didn’t expect: seven different background processes (or cron jobs, which are scheduled tasks that run automatically) and workers that had been running silently for up to five years. All of them doing absolutely nothing useful.

Here’s why that matters, how we found them, and what we did about it.

Why this matters more than you’d think

Yes, running unnecessary infrastructure costs money. I did a quick calculation and for one of those workers, we would have paid ~$360-600 over 5 years. This is a modest amount in the grand scheme of our finances, but definitely pure waste for a

process that does nothing.

However, after going through this cleanup, I'd argue the financial cost is actually the smallest part of the problem.

Every time a new engineer joins the team and explores our systems, they encounter these mysterious processes. "What does this worker do?" becomes a question that eats up onboarding time and creates uncertainty. We've all been there — staring at a piece of code, afraid to touch it because *maybe* it's doing something important.

Even "forgotten" infrastructure occasionally needs attention. Security updates, dependency bumps, compatibility fixes when something else changes. This led to our team spending maintenance cycles on code paths that served no purpose.

And over time, the institutional knowledge fades. Was this critical? Was it a temporary fix that became permanent? The person who created it left the company years ago, and the context left with them.

How does this even happen?

It's easy to point fingers, but the truth is this happens naturally in any long-lived system.

A feature gets deprecated, but the background job that supported it keeps running. Someone spins up a worker "temporarily" to handle a migration, and it never gets torn down. A scheduled task becomes redundant after an architectural change, but nobody thinks to check.

We used to send birthday celebration emails at Buffer. To do this, we ran a scheduled task that checked the entire database for birthdays matching the current date and sent customers a personalized email. During a refactor in 2020, we switched our transactional email tool but forgot to remove this worker—it kept running for five more years.

None of these are failures of individuals — they're failures of process. Without intentional cleanup built into how we work, entropy wins.

How our architecture helped us find it

Like many companies, Buffer embraced the microservices movement (a popular approach where companies split their code into many small, independent services) years ago.

We split our monolith into separate services, each with its own repository, deployment pipeline, and infrastructure. At the time, it made sense: each service could be deployed on its own, with clear boundaries between teams.

But over the years, we found the overhead of managing dozens of repositories outweighed the benefits for a team our size. So we consolidated into a multi-service single repository. The services still exist as logical boundaries, but they live together in one place.

This turned out to be what made discovery possible.

In the microservices world, each repository is its own island. A forgotten worker in one repo might never be noticed by engineers working in another. There’s no single place to search for queue names, no unified view of what’s running where.

With everything in one repository, we could finally see the full picture. We could trace every queue to its consumers and producers. We could spot queues with producers but no consumers. We could find workers referencing queues that no longer existed.

The consolidation wasn’t designed to help us find zombie infrastructure — but it made that discovery almost inevitable.

What we actually did

Once we identified the orphaned processes, we had to decide what to do with them. Here's how we approached it.

First, we traced each one to its origin. We dug through git history and old documentation to understand why each worker was created in the first place. In most cases, the original purpose was clear: a one-time data migration, a feature that got sunset, a temporary workaround that outlived its usefulness.

Then we confirmed they were truly unused. Before removing anything, we added logging to verify these processes weren't quietly doing something important we'd missed. We monitored for a few days to make sure they were not called at all, and we removed them incrementally. We didn't delete everything at once. We removed processes one by one, watching for any unexpected side effects. (Luckily, there weren't any.)

Finally, we documented what we learned. We added notes to our internal docs about what each process had originally done and why it was removed, so future engineers wouldn't wonder if something important went missing.

What changed after clean up

We're still early in measuring the full impact, but here's what we've seen so far.

Our infrastructure inventory is now accurate. When someone asks, "What workers do we run?" we can actually answer that question with confidence.

Onboarding conversations have gotten simpler, too. New engineers aren't stumbling across mysterious processes and wondering if they're missing context. The codebase reflects what we actually do, not what we did five years ago.

Treat refactors as archaeology and prevention

My biggest takeaway from this project: every significant refactor is an opportunity for archaeology.

When you're deep in a system, really understanding how the pieces connect, you're in the perfect position to question what's still needed. That queue from some old project? The worker someone created for a one-time data migration? The scheduled task that references a feature you've never heard of? They might still be running.

Here's what we're building into our process going forward:

- **During any refactor**, ask: what else touches this system that we haven't looked at in a while?
- **When deprecating a feature**, trace it all the way to its background processes, not just the user-facing code.
- **When someone leaves the team**, document what they were in charge of, especially the stuff that runs in the background.

We still have older parts of our codebase that haven't been migrated to the single repository yet. As we continue consolidating, we're confident we'll find more of these hidden relics. But now we're set up to catch them and prevent new ones from forming.

When all your code lives in one place, orphaned infrastructure has nowhere to hide.

https://buffer.com/resources/infrastructure-refactoring/