

We Replaced a £100k/Year Manual Workflow With a £198/Month Kubernetes Platform — Here's How

Building VidCast: a production-grade, DevSecOps video-to-podcast platform on AWS EKS, from a four-person bootcamp team

by Samuel Onyejekwe, John Babalola, Seun Okegbola, and Chinedu Izuelu

Picture a London podcast company — 12 shows, 28 staff, a handful of very demanding clients — bleeding 36 hours a week of producer time on something that should never have needed a human: turning video recordings into MP3s.

We made this company up. Its name is Wavelength Media, and as far as we know it doesn't exist. But we built it as a deliberately realistic scenario for our capstone project, because “build a video-to-MP3 converter” is a Kubernetes exercise, and “save a real-feeling business £100k a year while keeping their lawyers happy” is an engineering problem with actual stakes attached — even if the business itself is fictional.

Here's the scenario: eight producers, one repetitive ritual — download a recording, find a converter, run a one-off FFmpeg command, wait, re-download, distribute. Multiply that by 60–80 recordings a week, and you get a quietly enormous problem: roughly £100,000 a year of skilled people's time spent on a task a script could do in the background. Worse, the files this produced were scattered across personal drives and Slack DMs with no single source of truth, and the videos themselves — client material from a (also invented) law firm and fintech client — were being uploaded to random third-party converter websites with zero audit trail.

This is the brief we gave ourselves for VidCast, our capstone project on the Cloudboosta DevOps Engineering bootcamp: build the platform that makes this problem go away, and build it to a standard a senior engineer would actually sign off on, not a demo that happens to work in front of an audience. **The client is fictional. Everything below it — the architecture, the security model, the cost, the bugs we hit — is not.**

It's live at **vidcast.online**. It costs about **£198 a month** to run. And it replaced a six-figure annual drag with infrastructure a two-person team can operate.

Here's what we built, and what we learned building it.

The shape of the problem

Three failures compounded each other in the old workflow:

- Labour cost — 36 person-hours a week, $\approx$ £100k/year, spent on a task that's fundamentally automatable.
- Lost files — no catalogue, no naming discipline, no way to answer “where is my file?” without asking around.
- Compliance exposure — client content leaving the company's control and landing on unvetted third-party sites, with no record of who touched what.

None of these are exotic problems. They're the kind of operational debt that accumulates quietly in a lot of small media and content businesses, and they map cleanly onto a pattern the biggest platforms in the world already solved: store, queue, process, notify. It's the same shape YouTube uses to ingest a video or Spotify uses to transcode a podcast — we just had to build it at our own scale, on our own budget.

What we built

VidCast is five Python microservices plus a single-replica “outbox relay,” running on Amazon EKS in eu-west-2 (London, for UK data residency):

- Frontend — a React SPA for upload, history, and downloads
- Gateway — the single API entry point: auth, rate limiting, the upload/download path
- Auth — bcrypt + JWT, backed by PostgreSQL, never exposed to the internet
- Converter — a pure queue consumer that does the actual FFmpeg work
- Notification — emails the producer a download link when the MP3 is ready

[Screenshot/diagram: end-to-end architecture — ALB → five microservices → RabbitMQ → datastores, one public door]

We picked Kubernetes, and specifically EKS, because the problem was never “run one container” — it was five independent services that each needed their own scaling behaviour, their own failure isolation, and a shared policy layer that could enforce security rules on all of them at once. Plain EC2 or a single-container PaaS would have meant rebuilding orchestration, scaling and policy enforcement by hand; EKS gave us that for the cost of learning Kubernetes, plus a mature ecosystem (KEDA for event-driven scaling, Kyverno for policy-as-code) that fit our exact shape of problem.

The core design decision underneath all of it: **a producer should never wait on a slow job**. When someone uploads a video, the gateway writes it to storage, drops a message on a queue, and returns *202 Accepted* in milliseconds. The heavy lifting — FFmpeg conversion — happens asynchronously, behind RabbitMQ, on its own time.

We reached for a real message broker rather than, say, a polling job or a simple background thread, because a queue gives you something polling can’t: a durable buffer that absorbs bursts (twenty files landing at once doesn’t overload the gateway) and survives a converter restarting mid-batch without losing work. RabbitMQ specifically, over a managed alternative, came down to the cost engineering below.

That asynchronicity is also where most of the interesting engineering lives. At-least-once message delivery means a message can arrive twice — so we built three reliability patterns to make that safe:

- Transactional outbox — the gateway writes the file and an “outbox row” in the same database write, so storing a file and queuing its conversion can never silently diverge.
- Idempotent consumers — a Redis SET-NX claim-once guard means a redelivered message is a no-op, not a duplicate MP3 and a duplicate email.
- Bounded retry + dead-letter queues — a transient failure gets retried with backoff; a permanently broken message gets parked for a human to look at, instead of looping forever.

Together, these turn “at-least-once delivery” into something that behaves like “exactly-once” in practice — without paying for a managed message broker to get there.

The security model nobody asked for but everybody needed

Because the scenario we set ourselves involved client media and real-sounding compliance obligations (the kind any law firm or fintech client would actually demand), security wasn't a checkbox — it was half the actual engineering effort. We treated the fictional client's expectations as a hard constraint, not a backdrop, which is what pushed the security work this far.

We built six layers of defence in depth: perimeter (one public door — an ALB terminating TLS, everything else is ClusterIP), network (zero-trust, default-deny NetworkPolicies — every pod-to-pod path is explicitly allow-listed on both ends), identity & secrets (IRSA for keyless AWS access, zero secrets in Git via the External Secrets Operator), container hardening (non-root, read-only filesystems, dropped capabilities, seccomp), supply chain (a Trivy CVE gate that fails the CI build on any fixable critical vulnerability), and policy-as-code (seven Kyverno policies enforcing all of the above at admission time).

Two of those choices are worth a sentence each. Kyverno sits at admission time rather than just scanning after the fact, so a pod that violates a rule never starts running in the first place — it's the difference between a report you read later and a door that simply doesn't open. And IRSA with the External Secrets Operator means no AWS credential and no application secret ever lives in a file, an environment variable, or a Git commit; each pod gets short-lived, scoped credentials at runtime instead.

The philosophy underneath it: **assume a pod will eventually be compromised, and make sure that's a contained incident, not a breach.** A compromised converter pod, for instance, has no network route to the notification service, can't read another service's secrets, can't escalate to root, and can't persist anything beyond a scoped, ephemeral /tmp volume. Each of those barriers comes from a different layer, so no single misconfiguration unravels the whole model.

We didn't just design for that assumption — we instrumented for it. Falco watches at the kernel level for the kind of behaviour a static policy can't catch, like a shell suddenly spawning inside a pod that should only ever be running Gunicorn. When we tested it by exec-ing into the gateway, an alert landed in our Slack channel within about ten seconds. That's the difference between a control you can point to in a slide and one you've actually watched fire.

[Screenshot: Falco/Falcosidekick alert in Slack, fired from a live kubectl exec into the gateway pod]

We also found — and fixed — two genuine authentication bugs while building this: a hard-coded admin claim that gave every logged-in user admin rights, and a sign-up path that minted new accounts as admins by default. Both lived in completely ordinary code. The lesson that stuck: **the scary bugs are rarely in the exotic code paths — they're in the boring ones nobody re-reads.**

The cost engineering

This is the part we're proudest of, honestly. Running this platform costs about **£198 a month** — under 6% of the client's £3,500/month budget cap.

The biggest lever was **refusing to manage what we didn't need to manage**. We costed out fully-managed AWS datastores (RDS, MongoDB Atlas, Amazon MQ, ElastiCache) and found the managed version of RabbitMQ alone — Amazon MQ — would cost roughly \$183/month on its own, more than our entire platform. So we kept MongoDB, PostgreSQL, RabbitMQ, and Redis self-hosted in-cluster, and delivered the reliability guarantees people usually pay for managed brokers in application code instead — the outbox, idempotency, and retry/DLQ patterns above.

The second lever was **KEDA scale-to-zero**. The converter — the heaviest, most CPU-intensive workload — only runs about 40 minutes a week in short bursts. The rest of the time it sits at zero replicas. Compute for the expensive part of the pipeline is effectively paid per conversion, in pence, rather than rented around the clock.

The third — and most aggressive — lever was simply **turning the whole thing off when nobody's using it**. Because 100% of our infrastructure is Terraform and 100% of our application layer is GitOps via Argo CD, we can terraform destroy the entire platform down to ~£0 and rebuild it from scratch in under 20 minutes. For a portfolio platform, that's not a nice-to-have — it's the single biggest cost lever of all.

[Screenshot: Kubecost dashboard showing the actual ~£198/month breakdown by namespace]

How a change actually ships

None of the above means much if getting a change into production is a manual, ad-hoc act, so the pipeline itself was built to the same standard. A pull request triggers GitHub Actions: lint, then a build of each of the five services, then the Trivy CVE gate — any fixable critical vulnerability fails the build right there, before an image ever reaches a registry. A merge to main pushes the new image and lets Argo CD take over: the dev environment auto-syncs within minutes, but production sits behind a manual approval gate, so a human always reviews what's about to change before it does.

That split — fast and automatic in dev, deliberate and gated in prod — is the whole point of GitOps for a small team: nothing reaches production from someone's terminal, and the gate that matters most is a human, not a script.

What broke (and what we learned)

No platform this size reaches production without breaking first, and we think the honest version of this story is more useful than the polished one.

Kubernetes 1.31 quietly removed the in-tree EBS provisioner. Our first PostgreSQL deploy just sat in Pending forever, with no error. The fix — install the EBS CSI driver add-on — was easy once we found it, but the lesson generalised further than the bug: a platform constraint that lives in someone's memory or a wiki page evaporates the moment that cluster gets torn down and rebuilt. The same constraint encoded in Terraform survives every rebuild automatically. We started treating “things that must be true about the cluster” as code to write, not knowledge to remember.

Zero-trust networking traded convenience for silence. The moment we applied a default-deny NetworkPolicy, services started hanging on calls to each other with no error logged — because a denied packet at the network-policy layer just times out, it doesn't fail loudly. We had to make every single required pod-to-pod flow explicit, on both ends, before applying the deny-all rule. It's a real trade-off worth naming honestly: a flat network is maximally convenient and maximally dangerous; default-deny is the opposite, and you pay for that safety with the occasional silent-hang debugging session.

Gunicorn forking workers silently halved our metrics. With the gateway running two worker processes and a naive in-process Prometheus registry, a single /metrics scrape only ever saw one worker's counters. Our dashboards looked plausible and were quietly wrong. Multiprocess-aware metrics collection fixed it — but the bigger lesson was: a dashboard that's wired up is not the same as a dashboard that's correct. Check the numbers, don't trust them.

Where it landed

Every target we set against the platform was validated live, against the running system at vidcast.online — not modelled, not assumed:

- 99.9% availability, measured over a 30-day window
- Sub-500ms p99 upload latency (excluding body transfer — conversion happens asynchronously)
- Sub-2-hour disaster recovery, tested with a dated drill
- Zero vulnerable images shipped, ever, because the Trivy gate makes that a property of the pipeline rather than a periodic check
- Sub-10-second cold start when KEDA scales the converter up from zero

Against the ~£100,000/year of producer time the old workflow consumed, the platform costs roughly £2,400/year to run — a return north of **40x** on running spend, before you even count the compliance liability it removes.

The actual takeaway

The technology here — Kubernetes, RabbitMQ, Kyverno, KEDA — isn't novel. What we think is worth sharing is the discipline underneath it: every architectural decision in this project traces back to a real business problem with a real cost attached, every cost-saving choice was tested against an honestly-costed alternative before being adopted, and every known limitation is documented rather than hidden.

A platform doesn't have to be huge to be production-grade. It has to be honest — about what it does, what it costs, what it doesn't yet do, and what would break first if you pushed it. That's the standard we tried to hold VidCast to, and it's the standard we'd recommend to anyone building something similar.

VidCast was built by Samuel Onyejekwe, John Babalola, Seun Okegbola, and Chinedu Izuelu as a capstone project for the Cloudboosta DevOps Engineering Bootcamp. The platform is live at vidcast.online and the source is open at github.com/johnnybabs/vidcast.