

VidCast

Building a Production-Grade Video-to-MP3 Platform on AWS EKS

A DevSecOps Engineering Report

Live platform: <https://vidcast.online>

Repository: www.github.com/johnnybabs/vidcast

Medium article:

https://drive.google.com/file/d/1VzuuoMyc6jgSNbPSCTFU8zUt5IJFdy80/view?usp=drive_link

Prepared by

Samuel Onyejekwe · John Babalola · Seun Okegbola · Chinedu Izuelu

Cloudboosta DevOps Engineering Bootcamp

June 2026

Declaration / Statement of Authorship

We declare that this report and the VidCast platform it describes are our own work, produced for the Cloudboosta DevOps Engineering Bootcamp. The architecture, infrastructure-as-code, application services, security controls, observability and operational tooling documented here were designed and implemented by the authors, except where explicit reference is made to third-party software, managed AWS services, or prior work. All sources, tools and libraries used are acknowledged in the text. The figures and tables were produced by the authors from the project's own source code, manifests and configuration. No part of this report has been submitted for any other assessment.

Where the platform's behaviour and its accompanying documentation differ, this report follows the platform as configured in the repository and notes the discrepancy. Known limitations, trust boundaries and demonstrative (as opposed to guaranteed) targets are disclosed honestly throughout.

Author	Signature	Date
Samuel Onyejekwe		
John Babalola		
Seun Okegbola		
Chinedu Izuelu		

Abstract

VidCast is a production-grade, event-driven platform that converts uploaded video into podcast-ready MP3 audio on Amazon EKS. It was built for a realistic client — Wavelength Media, a London podcast company producing 60–80 recordings a week — whose manual FFmpeg workflow consumed roughly £100,000 a year of producer time, scattered files across personal drives, and exposed client content on third-party sites. The platform replaces that workflow with five Python microservices plus a single-replica outbox relay, a store-and-queue pipeline (upload returns 202 in milliseconds; conversion happens asynchronously through RabbitMQ), self-hosted datastores (MongoDB/GridFS, PostgreSQL, RabbitMQ, Redis) and nightly encrypted S3 backups. It is delivered the DevSecOps way: Terraform infrastructure-as-code, GitHub Actions and Jenkins CI/CD with a shift-left Trivy gate, Argo CD GitOps with a manual production approval gate, six layers of defence-in-depth (zero-trust networking, IRSA least-privilege, hardened non-root containers, seven Kyverno policies), and full observability with three service-level objectives backed by multi-window burn-rate alerting. The result runs for approximately £198/month — about 6% of the client's budget cap — recovers from catastrophic loss in under two hours, keeps zero secrets in Git, and is operable by a two-person team. This report documents its architecture, implementation, security, operation, economics and the lessons learned in building it.

Executive Summary — A £198/Month Platform That Retires a Six-Figure Drag

VidCast replaces a chaotic weekly ritual of manual FFmpeg commands, lost files and unauditable client content with a centralised, queue-driven, observable platform. A producer uploads a video; the platform converts it; the producer receives an email and a download link; and operations sees every file in real time.

The business case is decisive. The status quo consumed about 36 producer-hours a week — close to a full-time salary, on the order of £100,000 a year — on a task that should be automated, while exposing client content to compliance risk. VidCast runs for roughly £198 a month (about £2,400 a year), a greater-than-40× return on running spend, and sits at only ~6% of the client's £3,500/month cap, leaving ~94% headroom to absorb growth.

The headline outcomes:

Metric	Value	Metric	Value
Microservices	5	Tested RTO	< 2 hours
Security layers	6	Secrets in Git	0
Kyverno policies	7	NodePort exposures	0
Monthly run cost	~£198	Prod approval gate	1
Budget headroom	94%	Live at	vidcast.online

Bottom line: the platform is operable by a two-person team because everything is in Git and nothing is in someone's terminal history; it is recoverable from total failure in under two hours; and it is built and hardened to a standard a senior assessor would recognise as production-grade.

Table of Contents

Abstract.....	3
Executive Summary — A £198/Month Platform That Retires a Six-Figure Drag	4
List of Abbreviations and Acronyms	23
Glossary of Terms	24
Chapter 1 — Introduction.....	26
1.1 Project Background and Context.....	26
1.2 Purpose and Scope of the Report.....	26
1.3 Definition of DevSecOps and Guiding Principles	27
1.4 Report Structure and How to Read It.....	28
1.5 Team, Roles and Responsibilities.....	29
Chapter 2 — Problem Statement	29
2.1 Case Study — Wavelength Media.....	30
2.2 Manual FFmpeg Conversion and 36 Lost Person-Hours per Week.....	31
2.3 Lost and Scattered Files — No Single Source of Truth	32
2.4 Compliance and Data-Residency Exposure (Client Content on Third-Party Sites)	33
Chapter 3 — Project Objectives and Success Criteria.....	34
3.1 Centralise Conversion (Upload → Convert → Deliver)	36
3.2 Guarantee No Lost Files (Durable Storage + Transactional Outbox)	36
3.3 Provide Pipeline Visibility (Dashboards + Correlation IDs).....	37
3.4 Secure Client Content Within the Company AWS Account	37
3.5 Operate Within a Defined Monthly Cost Cap (< £3,500/month).....	38
3.6 Scope, Assumptions and Out-of-Scope Items.....	39
Chapter 4 — Business Case and Value Proposition	40
4.1 Cost of the Status Quo (~£100k/year of Producer Time).....	40
4.2 Value Delivered and Return on Investment (~£2,400/year to Run)	41
4.3 Compliance, Audit Trail and Risk Reduction	42
4.4 How the Solution Was Pitched and Funded (Pain → Outcome → Cost Cap)	42
Chapter 5 — Architecture Overview.....	45
5.1 System Overview and Event-Driven (Store-and-Queue) Design	45

5.2 Design Principles.....	46
5.3 End-to-End Architecture Diagram (ALB as the Only Public Door)	47
5.4 Port Map and Network Topology	48
5.5 Environments — Production (EKS), Staging (Docker Swarm), Local.....	50
Chapter 6 — The Five Microservices	51
6.1 Frontend — React 18 + Vite + Tailwind + nginx (SPA, Upload, History, Downloads) .	52
6.2 Gateway — Flask + Gunicorn (API, Upload/Download, JWT Validation, Rate Limiting, Metrics)	52
6.3 Auth — Flask + bcrypt + PyJWT + psycpg2 (Login, Registration, Role Enforcement)	53
6.4 Converter — Pika + MoviePy + FFmpeg (Queue Consumer, MP4 → MP3)	53
6.5 Notification — Pika + smtplib (Email Delivery, Batch Summary Emails)	53
6.6 Outbox Relay — Single-Replica Publisher (MongoDB → RabbitMQ)	53
6.7 Service Boundaries, Separation of Concerns and Inter-Service Contracts	54
Chapter 7 — Data Layer and Messaging	55
7.1 MongoDB + GridFS — Video/MP3 Blobs, Outbox and job_status Collections	56
7.2 PostgreSQL — User Accounts, bcrypt Hashes, Roles (SCRAM-SHA-256, ACID, EBS PVC)	57
7.3 RabbitMQ — video and mp3 Queues, Retry Queues, Dead-Letter Topology	58
7.4 Redis — Idempotency Keys and Rate-Limit Counters	60
7.5 Amazon S3 — Versioned, Encrypted Backup Bucket	60
7.6 Self-Hosted vs Managed Datastores — Cost-Driven Trade-off Analysis	61
Chapter 8 — Reliability and Resilience Patterns	62
8.1 Single Upload Journey — Every Hop from Browser to Download Link	63
8.2 Transactional Outbox (A1) — Atomic File + Outbox Write, Single Relay with Publisher Confirms	64
8.3 Idempotent Consumers (A2) — Redis SETNX Claim-Once, At-Least-Once → Effectively-Once	65
8.4 Bounded Retry + Dead-Letter Queues (A3) — Backoff, Max-Retries, Poison-Pill Parking	67
8.5 Health Probes and Graceful Degradation (HTTP /healthz, File-Based Heartbeats) ...	68

8.6 Feature Flags (OUTBOX_ENABLED, IDEMPOTENCY_ENABLED, MAX_RETRIES, RETRY_TTL_MS)	69
Chapter 9 — Infrastructure as Code (Terraform)	73
9.1 Module Structure and Reproducibility (Full Rebuild in < 20 Minutes).....	73
9.2 VPC Module — Subnets across AZs, Internet Gateway, Routes, ELB Tags	75
9.3 IAM Module — Cluster Role, Node Role, IRSA Trust Policies.....	75
9.4 EKS Module — Cluster v1.31, Node Group, OIDC Provider, VPC-CNI, EBS CSI Add-on	76
9.5 Security-Groups Module — NodePort and ALB Rules.....	77
9.6 ECR Module — Immutable Tags, Scan-on-Push, Encryption, Lifecycle Policy	77
9.7 Storage Module — Versioned S3 Backup Bucket + Backup IRSA Role	78
9.8 External-Secrets Module — ESO IRSA + SSM Parameter Store Integration	78
9.9 Load Balancer Controller Module — LBC IRSA Role and Policy	79
9.10 GitHub-OIDC Module — Keyless CI/CD Federation (No Stored AWS Keys).....	79
9.11 Environments (dev/prod) and tfvars Configuration	80
Chapter 10 — Containerisation and Image Strategy	81
10.1 Multi-Stage Dockerfiles and Minimal Final Images.....	82
10.2 Non-Root UID 1000, Read-Only Root Filesystem, Dropped Capabilities	83
10.3 Dependency Pinning and CVE Hygiene (requirements.txt / package.json)	85
10.4 SHA-Pinned Image References in Kustomize Overlays.....	86
Chapter 11 — CI/CD Pipelines	87
11.1 GitHub Actions CI — Ruff Lint → Docker Build (matrix) → Trivy Scan → Push (commit-SHA tag).....	88
11.2 Shift-Left Trivy Gate (Fixable CRITICAL/HIGH CVE Fails the Build).....	89
11.3 GitHub Actions CD — OIDC to AWS, kubeconfig, kubectl set image, Rollout Verification.....	90
11.4 Jenkins Pipeline — Swarm Staging Deploy, Smoke Test, Manual Production Approval.....	91
11.5 Docker Hub / ECR Trust Chain and Image Tagging Convention	92
11.6 Dev vs Prod — Two Pipeline Systems Compared	93
Chapter 12 — GitOps with Argo CD	94
12.1 Git as the Single Source of Truth; Continuous Reconciliation	95

12.2 Manifest Strategy — Helm for State, Kustomize (base + overlays) for Apps.....	96
12.3 vidcast-dev — Auto-Sync, Self-Heal, Drift Correction.....	97
12.4 vidcast-prod — Manual Sync as a Human Approval Gate.....	97
12.5 Replica Ownership Carve-Outs (Autoscaler-Owned /spec/replicas).....	98
12.6 Rollback via git revert; The "No kubectl edit to Prod" Rule	99
Chapter 13 — Autoscaling	100
13.1 KEDA on the Converter — Scale 0→N on RabbitMQ Queue Depth, Scale-to-Zero after Idle	101
13.2 HPA on the Gateway — 2→5 Replicas on CPU Utilisation	102
13.3 TriggerAuthentication, Node CPU Budget and maxReplicas Capping	103
13.4 Avoiding KEDA/HPA Conflict (Right Signal per Workload)	104
Chapter 14 — Security Model and Threat Landscape	108
14.1 Threat Model and Security Objectives	108
14.2 The Six-Layer Defence-in-Depth Model.....	109
14.3 Blast-Radius Containment Philosophy.....	111
Chapter 15 — Perimeter and Network Security (Zero-Trust).....	113
15.1 ALB + ACM TLS Only; Every Service ClusterIP; Zero NodePort to the Internet	113
15.2 Route 53 DNS and HTTP→HTTPS Redirect at the ALB.....	114
15.3 NetworkPolicy Default-Deny on Ingress and Egress	115
15.4 Explicit Pod-to-Pod Allow-Lists (Per-Selector, Per-Port, Both Ends).....	115
15.5 VPC-CNI Native Network-Policy Agent (No Calico/Cilium).....	117
15.6 Datastore, Monitoring, Backup and Sigstore Egress Policies	118
Chapter 16 — Identity, Authentication and Secrets.....	119
16.1 bcrypt (cost-12) Password Hashing.....	119
16.2 JWT Issuance/Validation and Role-Based Access Control (RBAC)	120
16.3 Admin Guardrails (Self-Demote 403, Last-Admin 409) and the "Everyone Was an Admin" Fix.....	121
16.4 IRSA — IAM Roles for Service Accounts (Least Privilege, No Long-Lived Keys)	122
16.5 External Secrets Operator + AWS SSM Parameter Store (0 Secrets in Git).....	123
16.6 Secret Rotation, Refresh Intervals and ClusterSecretStore	124
Chapter 17 — Container and Runtime Hardening.....	125

17.1	runAsNonRoot + UID 1000 (No Root in Any Pod)	126
17.2	readOnlyRootFilesystem with Explicit Writable tmp Volumes	127
17.3	Drop ALL Capabilities; allowPrivilegeEscalation: false	127
17.4	seccomp RuntimeDefault (Dangerous Syscalls Blocked)	128
17.5	SHA-Pinned Images and the Hardened Pod Security Context	128
Chapter 18 — Supply-Chain Security		130
18.1	Trivy CVE Gate — Scan Every Image After Build, Before Push	131
18.2	Image SHA-Pinning in Kustomize Overlays	132
18.3	SBOM Generation and SARIF Reporting	132
18.4	Cosign Keyless Signing (Fulcio, Rekor, TUF Trust Anchors)	132
18.5	ECR Hardening — Tag Immutability, Scan-on-Push, Lifecycle, Encryption	133
18.6	Roadmap — Kyverno verify-images from Audit to Enforce	134
Chapter 19 — Policy-as-Code (Kyverno)		136
19.1	Policy Strategy and Audit-Mode Baseline; Exempt Platform Namespaces	136
19.2	require-non-root	138
19.3	require-seccomp-runtime-default	138
19.4	require-requests-limits	139
19.5	disallow-privileged	139
19.6	disallow-latest-tag	139
19.7	require-labels (Attribution & Cost Allocation)	140
19.8	verify-images (Cosign Signature Admission Control)	140
19.9	Security Controls Summary and Compliance Mapping	141
Chapter 20 — Observability – Metrics and Dashboards		143
20.1	kube-prometheus-stack (Prometheus / Grafana / Alertmanager)	143
20.2	Application Metrics via prometheus_client (Multiprocess-Aware)	144
20.3	ServiceMonitors and PodMonitors (Gateway, Converter, Notification, RabbitMQ, Kubecost)	144
20.4	Grafana Dashboards — Operations, SLO/Error-Budget, FinOps	145
20.5	RabbitMQ Queue Depth as the Leading Backlog Indicator	146
Chapter 21 — Structured Logging and Tracing		146
21.1	JSON Logging via Shared jsonlog.py (No print())	147

21.2 Correlation IDs Minted at the Gateway, Carried Every Hop.....	148
21.3 Download Audit Trail.....	149
21.4 "Where Is Our File?" — End-to-End Request Tracing.....	150
Chapter 22 — Service Level Objectives and Alerting.....	151
22.1 Availability SLO — 99.9% Non-5xx Gateway Requests	151
22.2 Conversion-Latency SLO — 95% of Conversions ≤ 5 Minutes	151
22.3 End-to-End Success SLO — 99% of Uploads Produce a Notification.....	152
22.4 Error Budgets and Multi-Window, Multi-Burn-Rate Alerting.....	152
22.5 Operational Alerts (CrashLoopBackOff, Node CPU/Mem, Queue Backlog, Broker Down)	154
22.6 SLO Runbooks (Alert → First Action)	154
Chapter 23 — Abuse Protection and Rate Limiting	155
23.1 Flask-Limiter with Redis Backend (Login 10/min, Upload 20/hr)	155
23.2 X-Forwarded-For Handling and Deployment Caveats	156
Chapter 24 — Disaster Recovery and Business Continuity.....	158
24.1 Backup Strategy — Nightly mongodump (02:00) + pg_dump (02:15) to S3	158
24.2 Backup Bucket Controls — Versioned, Public-Access-Blocked, 30-Day Lifecycle...	159
24.3 Write-Only Backup IRSA vs Separate Restore Role	160
24.4 Two Recovery Paths — App/Config from Git (Argo CD), Data from S3	160
24.5 Tested Runbook and RTO/RPO Targets (< 2-Hour RTO)	161
24.6 Merge Runbooks (bcrypt + RBAC Coordinated Deployment).....	162
Chapter 25 — AWS Well-Architected Review.....	165
25.1 Operational Excellence (GitOps, IaC, Structured Logs, Tested Runbooks)	166
25.2 Security (Six Defence-in-Depth Layers, Zero-Trust, ESO, Kyverno, Trivy Gate).....	167
25.3 Reliability (Outbox, Idempotency, Retry/DLQ, < 2h RTO)	167
25.4 Performance Efficiency (KEDA Event-Driven Scaling, HPA, Right-Sized Nodes).....	168
25.5 Cost Optimization (Scale-to-Zero, Self-Hosted Datastores, Tear-Down)	168
25.6 Sustainability (Resource Right-Sizing, Idle Reduction).....	169
Chapter 26 — Cost Engineering and FinOps.....	170
26.1 Monthly Cost Breakdown, Resource by Resource (~£198/month, eu-west-2).....	170
26.2 KEDA Scale-to-Zero — Compute Paid per Conversion	171

26.3 Self-Hosted Datastores vs Managed (Amazon MQ Floor ~£183/mo)	172
26.4 Right-Sized Nodes and Idle-Utilisation Tuning (93% → 82%)	173
26.5 Off-Unless-Demoing (terraform destroy → ~£0, Rebuild in 20 min)	173
26.6 Kubecost FinOps Dashboard and Cost-per-Conversion	174
26.7 Budget Headroom (94%) and Parameter Store vs Secrets Manager Savings	175
Chapter 27 — Validation and Results	176
27.1 Test Strategy (Lint, Trivy Scan, Smoke Test) and Coverage Gaps	176
27.2 Production Validation Against SLO Targets (Live at vidcast.online)	177
27.3 End-to-End Smoke Test — Login → Upload → Convert → Download	178
27.4 Availability, p99 Latency, Error Rate and DR Recovery Results	178
27.5 Security Scan and Policy-Compliance Results	179
Chapter 28 — Challenges, Root Causes and Solutions	182
28.1 PVCs Stuck Pending (K8s 1.31 In-Tree EBS Provisioner Removed → EBS CSI Add-on)	182
28.2 Silent Cross-Service Hangs (NetworkPolicy Default-Deny → Explicit Allow Rules)	183
28.3 'Everyone Was an Admin' and Sign-Up Privilege Escalation	183
28.4 Metrics, Audit-Log Buffering and Pipeline Pre-Test Issues	184
Metrics half-missing under Unicorn	184
Audit-log and consumer-log buffering	184
Pipeline pre-test issues	184
28.5 Other Documented Incidents and Fixes	185
Chapter 29 — Lessons Learned	186
29.1 Encode Platform Constraints in Code, Not Docs	187
29.2 Zero-Trust Trades Convenience for Blast-Radius Control	188
29.3 Design Decisions and Trade-offs Revisited	189
Chapter 30 — Future Roadmap	190
30.1 S3 + Pre-Signed URLs (Replace GridFS; ~£370/mo Saving at Volume)	191
30.2 Kyverno Enforce Mode (Promote All 7 Policies from Audit)	192
30.3 Cosign Image Signing in the Pipeline	193
30.4 Workspace Multi-Tenancy (3-Role Model in PostgreSQL)	194

30.5 Amazon SES Email (Replace Gmail SMTP)	194
30.6 Log Shipping, Cross-Region Backups and Further Hardening.....	194
Chapter 31 — Conclusion	196
31.1 VidCast by the Numbers	196
31.2 Summary of Outcomes Against Objectives	197
31.3 Closing Remarks	198

List of Figures

Figure 1.1 — The DevSecOps lifecycle as implemented in VidCast: each stage is backed by a specific tool, and security (shift-left) is woven through all of them.	27
Figure 2.1 — The four compounding problems, all stemming from one root: manual, off-platform conversion. Each is examined in this chapter and answered by the objectives in Chapter 3.	30
Figure 2.2 — Wavelength Media at a glance. The volume (60–80 recordings/week) is the multiplier that makes a manual per-file task unsustainable.	30
Figure 2.3 — The manual conversion workflow. Seven human steps per file, none of them tracked, several of them off-platform — the breakages mapped here become the four problems of Figure 2.1.	31
Figure 2.4 — The labour cost. The 36 weekly hours annualise to roughly £100k of producer time — a six-figure drag the platform was built to retire.	32
Figure 2.5 — The data-residency exposure. Client content leaves the company boundary to an unknown third-party site, with no audit trail and no control over where it is stored.	33
Figure 3.1 — Each objective answers a specific Chapter 2 problem. This chapter defines the success criterion for each; the cited later chapters describe how it was built.	35
Figure 3.2 — The target operating model: one platform, three steps, every file tracked end-to-end — replacing the seven manual steps of Figure 2.3.	36
Figure 3.3 — The cost criterion. Actual run cost (~£198/month) sits far under the £3,500/month cap, leaving ~94% headroom to absorb growth.	38
Figure 4.1 — Cost of the status quo: the producer-time calculation to ~£100k/year, plus the real-but-harder-to-price costs of delays, lost files and compliance exposure.	40
Figure 4.2 — Monthly run cost (~£198) broken down against the £3,500/month cap: the platform uses under 6% of the budget, leaving 94% headroom.	41
Figure 4.3 — The five-step pitch: frame the pain in money, propose one outcome, anchor to the cost cap, de-risk with phasing, and prove it is production-grade.	43
Figure 5.1 traces a single upload from the browser to the download link. The boundary between synchronous and asynchronous work is the key design point: only Steps 1 and 2 happen while the user waits — the gateway writes the file and an outbox row in one Mongo write and returns 202 Accepted. Everything from Step 3 onward runs on its own time.	45
Figure 5.1 — The single-upload journey. Steps 1–2 are synchronous (the 202 returns in milliseconds); Steps 3–6 are asynchronous, event-driven hops.	46
Figure 5.2 shows every component of the platform on one diagram. The defining property of the topology is that there is exactly one public door: the internet-facing AWS Application Load Balancer. It is the only resource that accepts traffic from the internet, and it is where TLS terminates.	47
Figure 5.2 — End-to-end architecture. The internet-facing ALB is the only public entry point; every microservice is ClusterIP, reachable only inside the cluster.	47

Figure 5.3 — Zero-trust network topology. The gateway has no route to notification (shown blocked); a compromised pod cannot pivot to a service it was never allowed to reach.	50
Figure 6.1 — Service boundaries and the contract on each: HTTP/REST for requests, AMQP messages for the pipeline, SMTP at the edge; GridFS is the blob medium for the gateway.	54
Figure 6.2 — The single async message contract, enriched at each hop (the converter adds mp3_fid) and never reshaped; idempotency is keyed on video_fid.	55
Figure 7.1 — Persistence and durability of the stateful tier: each datastore's backing volume, and the nightly logical dump to S3 that is the durability backstop.	56
Figure 7.2 — RabbitMQ retry and dead-letter topology for the video pipeline, with the Redis claim-once guard. The mp3 pipeline mirrors it exactly.	59
Figure 7.3 — Standing monthly cost (eu-west-2, 24/7): the all-managed datastore stack ≈ \$267/mo versus ≈ \$0 standing for the in-cluster Helm charts.	61
Figure 8.1 — The nine hops of a single upload. Solid = synchronous; dashed (purple) = the outbox-relay publish path; dotted = datastore reads/writes. The gateway returns 202 at step ②; everything after is asynchronous.	63
Figure 8.2 — The outbox pattern: the gateway writes the GridFS object and the outbox row through the same MongoDB client; a single-replica relay later publishes unpublished rows and stamps them.	64
Figure 8.3 — The claim-once decision. The first delivery wins the atomic Redis claim and processes; a duplicate loses the claim, ACKs, and skips. A Redis outage fails open.	66
Figure 8.4 — With the defaults (MAX_RETRIES=3, RETRY_TTL_MS=30000), a persistently failing message is attempted at t=0, 30, 60, 90 s, then parked in the terminal DLQ — total dwell ≈ 90 s before a human inspects it.	68
Figure 9.1 — Module dependency graph. The foundation (vpc, iam) feeds eks; eks's OIDC provider feeds every IRSA-consuming module; security-groups, ecr and github-oidc are independent. Terraform derives apply order from these edges.	74
Figure 9.2 — The keyless IRSA flow shared by ESO, the LBC, and the backup jobs: a ServiceAccount's projected OIDC token is exchanged for short-lived role credentials. No long-lived AWS keys exist anywhere.	79
Figure 10.1 — Build strategies. The frontend discards its Node toolchain in a second nginx stage; the Python images stay single-stage but are kept lean and consistently hardened.	82
Figure 10.2 — The layered runtime hardening applied to every VidCast pod. A compromised process is non-root, on an immutable filesystem, with no Linux capabilities and no path to escalate.	84
Figure 11.1 — The CI pipeline. Lint gates the matrix; each of the five services builds, is Trivy-scanned, and only then (on a main-branch push) is pushed with its commit-SHA tag.	89
Figure 11.2 — Why the gate's placement matters. The cost and blast radius of a vulnerable image rise sharply the later it is caught; VidCast fails at the CI gate, before the registry or runtime.	90
Figure 11.3 — The CD deploy flow. A per-run OIDC token is exchanged for temporary credentials on the scoped deploy role, which can describe the cluster; an EKS access entry grants the kubectl-level rights.	91
Figure 11.4 — The Jenkins pipeline: build and scan, deploy to a Swarm staging environment, smoke-test it, pause for a human approval, then deploy to EKS — with automatic rollback if any stage fails.	92

Figure 12.1 — The reconciliation loop. Argo polls Git (desired state), compares it to the live cluster, and applies the diff. Out-of-band drift is detected and, in dev, reverted by selfHeal.....	95
Figure 12.2 — Promotion contrast. dev (top) flows hands-off to live; prod (bottom) requires a reviewed PR merge as approval and an explicit manual sync as the deploy.....	98
Figure 13.1 — KEDA scale-to-zero in action. The converter sits at 0 replicas while idle, scales up when a burst lands on the video queue, and — 60 s after the queue drains (cooldownPeriod) — scales back to 0.	102
Figure 13.2 — The gateway HPA. As a load surge pushes CPU above the 70% target, the HPA adds replicas until average utilisation falls back to target; it never scales below 1.	103
Figure 13.3 — Separation by construction. KEDA scales the converter on queue depth; the HPA scales the gateway on CPU. Distinct targets mean the controllers' writes never collide.	105
Figure 14.1 — The attacker kill-chain (top) and the VidCast control that blocks each step (bottom). Reaching the data requires defeating every layer in sequence.	109
Figure 14.2 — The six-layer model. From the public edge inward: Perimeter, Network, Identity & Secrets, Container, Supply Chain, and Policy-as-Code, wrapping the five services and their data.	110
Figure 14.3 — Containment of a single compromised pod. Each control independently denies a path an attacker would otherwise use to widen the breach.	112
Figure 15.1 — The perimeter. The ALB terminates TLS and is the only internet-facing entry; the frontend, gateway, and all datastores are ClusterIP, with no public path to the stateful tier.	113
Figure 15.2 — Both-ends rule. The gateway→auth flow requires an egress rule on the gateway AND a matching ingress rule on auth; either one missing means the packet is denied.....	116
Figure 15.3 — The complete allowed-traffic matrix. Each ✓ is an explicitly declared flow; every blank cell is denied by default-deny. A compromised converter, for instance, can reach only RabbitMQ, Mongo and Redis.	117
Figure 16.1 — Login → JWT → RBAC. The auth service verifies the bcrypt hash and mints a signed JWT carrying the role; the gateway is where admin authorization is enforced.....	120
Figure 16.2 — The two admin guardrails. Self-demotion is blocked (403) and demoting the last remaining admin is blocked (409); every role change is audit-logged.	122
Figure 16.3 — ESO pulls each parameter from SSM Parameter Store (via the vidcast-eso IRSA role) and writes the Kubernetes Secret the pod consumes. Only the path references are committed to Git.	123
Figure 17.1 — Anatomy of a hardened pod: pod-level identity/seccomp wrapping container-level filesystem/capability/escalation locks, with one writable /tmp. Each control denies a distinct escalation path.	126
Figure 17.2 — Hardening-control coverage. First-party services are fully hardened; MongoDB/PostgreSQL are documented root exceptions (seccomp + no-escalation only) and the frontend needs a writable rootfs.....	129
Figure 18.1 — The verifiable supply chain (docs/SUPPLY_CHAIN.md). Each stage contributes one trust property; together they let the cluster admit only images built and signed by VidCast's own pipeline.	131
Figure 18.2 — The keyless signing trust anchors. The GitHub OIDC identity is bound by a short-lived Fulcio certificate; the signature is recorded in Rekor; the TUF root bootstraps trust in the Fulcio/Rekor keys.	133

Figure 18.3 — Honest supply-chain status. The Trivy gate, SHA-pinning, ECR hardening and the written verify-images policy are live; SBOM/SARIF, Cosign signing, and the Audit→Enforce promotion are the roadmap. .	135
Figure 19.1 — Kyverno sits in the admission path. Today every policy is Audit (report-only); after promotion, a violating pod is rejected before it can run.....	136
Figure 19.2 — The seven ClusterPolicies grouped by category: Pod Security Standards, Best Practices, and Supply Chain Security.....	138
Figure 20.1 — The kube-prometheus-stack scrape topology: Service/PodMonitors drive Prometheus, which feeds the three Grafana dashboards and Alertmanager.	143
Figure 20.2 — Queue depth as the leading backlog indicator: it spikes on an upload burst ahead of latency, crosses the alert threshold, and is the signal KEDA scales the converter on.	146
Figure 21.1 — Anatomy of a structured log line: a fixed common set of fields plus arbitrary per-call context, emitted as one JSON object per line.....	147
Figure 21.2 — The correlation_id is minted once at the gateway and carried through the queues, relay, converter and notification — so one id appears on every log line from upload to email.....	149
Figure 22.1 — Error-budget burn-down for the availability SLO: a 1× burn just exhausts the 43.2-minute budget at the 30-day edge; a 14× burn exhausts it far sooner and pages.	153
Figure 22.2 — Multi-window, multi-burn-rate logic: the long window sets severity; the short window must also be burning (logical AND), so the alert self-clears after the incident.	153
Figure 23.1 — Flask-Limiter with a shared Redis counter: all workers/pods increment one per-IP bucket; under the limit the handler runs, over it the request gets a 429.	156
Figure 23.2 — The X-Forwarded-For chain through ALB and nginx, the spoofing caveat of keying on the first hop, and the robust handling that trusts only known proxy hops.	157
Figure 24.1 — Backup architecture: staggered nightly CronJobs dump and gzip, then upload to a hardened S3 bucket as the write-only vidcast-backup ServiceAccount.	159
Figure 24.2 — The two recovery paths converging on an end-to-end smoke test: app/config reconciled from Git by Argo CD, data restored from the S3 dumps.	161
Figure 24.3 — Recovery timeline: the runbook's steps sum to roughly 45 minutes, comfortably inside the < 2-hour RTO target, against an RPO of ≤ 24 hours.	162
Figure 25.1 — VidCast mapped to the six Well-Architected pillars at a glance. Each pillar is examined in detail in the sections that follow (and built out in the cited chapters).	165
Figure 25.2 — Self-assessed coverage per pillar. Strong across the board for a demo platform; the deliberate gaps are scope decisions, not oversights.....	166
Figure 26.1 — Where the money goes. Compute (the two nodes) is 58% of spend and the EKS control plane 30%; the ALB, DNS and S3 together are the remaining ~12%.	171
Figure 26.2 — The converter's weekly duty cycle. It runs only ~40 minutes a week in short bursts and sits at 0 replicas for 100+ idle hours — so compute is paid per conversion, in pence.	171
Figure 26.3 — Managed-datastore standing cost vs the self-hosted (£0) choice. Amazon MQ for RabbitMQ alone floors at ~\$183/month — more than the entire VidCast bill.....	172

Figure 26.4 — The off-unless-demoing lifecycle. Standing cost is ~£0; the cluster is applied for a demo window and destroyed afterwards, with EBS snapshots persisting cheaply in between.....	174
Figure 27.1 — The validation strategy: the automated and manual gates that run (left) and the honestly-acknowledged coverage gap (right), with the mitigations that compensate for it.	176
Figure 27.2 — Production validation against the SLO targets. All four headline targets — availability, upload latency, error rate, and DR recovery — are met.	177
Figure 27.3 — The end-to-end smoke test path, all stages passing — the complete journey from authentication to a downloadable MP3.	178
Figure 27.4 — Measured KEDA cold start. From an idle (zero-replica) converter, a queued message brings a Ready pod in under 10 seconds.....	179
Figure 28.1 — The two authentication holes (hard-coded admin claim and admin-minting sign-up) and the fixes that closed them.....	183
Figure 28.2 — The poison-message fix: the original NACK-requeue loop (left) versus the bounded retry-plus-dead-letter topology that replaced it (right).	185
Figure 29.1 — The EBS-provisioner failure and its fix, captured in Terraform so the next cluster apply cannot reproduce the symptom.	187
Figure 29.2 — Convenience vs blast-radius control: in a flat network a compromised gateway reaches everything; under default-deny it is contained to its allow-list (auth, MongoDB, RabbitMQ).	188
Figure 30.1 — Roadmap prioritisation. Bubble colour denotes the dominant value (cost+security, security, product, reliability, ops); all items are designed but not yet shipped.	191
Figure 30.2 — Today's GridFS blob store vs the roadmap S3 + pre-signed-URL model. The pre-signed URL also fixes the "email leaks the file id" weakness.....	192
Figure 30.3 — Closing the supply-chain loop. CI build and the Trivy gate exist today; Cosign signing, the Rekor log entry, and flipping Kyverno verify-images to Enforce are the roadmap.....	193
Figure 31.1 — VidCast by the numbers: the nine headline metrics that summarise the build.....	196
Figure 31.2 — Annual economics: ~£2,400/yr to run versus ~£100,000/yr of producer time reclaimed — a greater-than-40× return on running spend.	198

List of Tables

Table 1.1 — VidCast project at a glance.	26
Table 1.2 — The eight guiding principles and their concrete expression in the platform.....	28
Table 1.3 — The six parts of the report, their coverage, and the intended reader for each.	28
Table 1.4 — The team and their lead responsibilities across the DevSecOps workstreams.	29
Table 2.1 — The Wavelength Media operating profile (source: project case study / pitch deck).	31
Table 2.2 — The cost of manual conversion (source: business case, pitch deck slide 6).	32
Table 2.3 — The file-management failures of the status quo.	33
Table 3.1 — The objectives, the problems they answer, and where each is delivered in this report.....	35
Table 3.2 — Success criteria for "no lost files" (targets from docs/SLO.md).	37
Table 3.3 — Visibility success criteria, anchored to the formal SLOs (docs/SLO.md).	37
Table 3.4 — Cost success criteria (targets from the pitch; actuals from Chapter 26 cost analysis).	38
Table 3.5 — Out-of-scope items and the documented reason for each (PHASE_UP_PLAN.md §2.2).	39
Table 4.1 — The cost of the status quo: one quantified driver (~£100k/yr) and four real-but-unpriced costs.	41
Table 4.2 — Return on investment: ~£2,400/year to run reclaims ~£100,000/year of producer time.	41
Table 4.3 — Compliance and risk reduction: each status-quo risk is matched by a concrete platform control.	42
Table 5.1 — The four design principles and the client constraints they answer.	47
Table 5.2 lists the port each component listens on and how it is reached. In the hardened production posture the ALB on 443 is the only port open to the internet and administrative UIs are reached with kubectl port-forward. The NodePort range (30002–30008) is provisioned in the security-group module for non-ALB and admin access in lower environments; it is not the production ingress path.....	48
Table 5.2 — Port map. Production exposes only the ALB (443); the NodePort range exists for admin and lower-environment access.....	49
Table 5.3 — The complete allow-list of pod-to-pod (and egress) flows. Everything else is denied.	49
Table 5.4 — The three environments. Same images and code; platform and cost scale to purpose.	51
Table 6.1 — Service inventory: stack, exposure, replica count and single responsibility.	52
Table 6.2 — The synchronous HTTP API: the gateway's public-facing routes and Auth's internal routes.	53
Table 6.3 — The inter-service contracts: who talks to whom, over what transport, with what payload.	55
Table 7.1 — Datastore inventory as configured in the Helm charts and Terraform.	56
Table 7.2 — The two GridFS databases and the two operational collections, all in MongoDB.	57
Table 7.3 — The auth_user table (Helm_charts/Postgres/init.sql).	58
Table 7.4 — RabbitMQ queue and exchange topology (declared idempotently on consumer start).	59

Table 7.5 — Redis usage: idempotency claims (consumers) and rate-limit counters (gateway).	60
Table 7.6 — Self-hosted vs managed datastore trade-off (from the A5 decision record, docs/MANAGED_SERVICES.md).	62
Table 8.1 — The four reliability patterns, the failure each prevents, and their implementation.	63
Table 8.2 — How the idempotency claim is handled per processing outcome (converter & notification consumers).	67
Table 8.3 — A3 retry/DLQ objects declared by rabbitmq_retry.declare_topology().	67
Table 8.4 — Probe configuration per workload (k8s/base/*/deployment.yaml).	69
Table 8.5 — Graceful-degradation responses to dependency failure.	69
Table 8.6 — Reliability feature flags, their defaults, and effects (k8s/base/*/configmap.yaml).	70
Table 9.1 — The nine Terraform modules and what each provisions.	73
Table 9.2 — Subnet tags applied by the vpc module (modules/vpc/main.tf).	75
Table 9.3 — The two EKS service roles (modules/iam/main.tf).	75
Table 9.4 — NodePort ports opened by the security group (root sets the list; ports map to k8s services).	77
Table 9.5 — ECR hardening controls (modules/ecr/main.tf; defaults from variables.tf).	77
Table 9.6 — Selected root variables (environments/dev/variables.tf, terraform.tfvars.example).	80
Table 10.1 — The six VidCast images, their bases, build styles and runtime entrypoints.	82
Table 10.2 — Per-image toolchain footprint. Smaller is better; each inclusion is justified.	83
Table 10.3 — The hardened security context (k8s/base/*/deployment.yaml + Dockerfile USER).	84
Table 10.4 — Selected dependency floors with the CVE/compatibility reason recorded in the manifests.	85
Table 11.1 — The two pipeline systems at a glance (.github/workflows/, Jenkinsfile).	87
Table 11.2 — The Trivy gate configuration (ci.yml). The combination is a true shift-left control.	89
Table 11.3 — The nine Jenkins stages (Jenkinsfile).	92
Table 11.4 — The image trust chain. The deploy path is keyless; registry push uses scoped tokens; ECR is hardened with immutable tags + scan-on-push.	93
Table 11.5 — Image tagging conventions. Both are immutable and map a running pod back to its source.	93
Table 11.6 — The two systems compared across the dimensions that matter for delivery.	94
Table 12.1 — The manage-vs-manual boundary (docs/GITOPS.md §4).	96
Table 12.2 — The manifest split. Helm for state and platform; Kustomize for the apps Argo manages.	96
Table 12.3 — vidcast-dev sync options (k8s/argocd/application-dev.yaml).	97
Table 12.4 — dev vs prod sync behaviour (docs/GITOPS.md §5).	98
Table 12.5 — Replica field ownership. Only the two autoscaled Deployments are carved out.	99
Table 13.1 — The two autoscalers, their signals, and their committed replica ranges (k8s/keda/).	101
Table 13.2 — ScaledObject parameters (k8s/keda/scaledobject-converter.yaml).	101

Table 13.3 — Approximate CPU-request budget on the single node. The converter’s 250m request is why a third replica will not schedule.	104
Table 13.4 — The two autoscalers never share a target — the root of the no-conflict guarantee.	105
Table 14.1 — Threat actors VidCast’s security model is designed to resist.	108
Table 14.2 — Security objectives and where each is delivered in Part IV.	109
Table 14.3 — The six layers, their controls, and where each is detailed (slide “Six Security Layers”).	110
Table 14.4 — Blast-radius containment: how each layer limits what a single compromised pod can do.	112
Table 15.1 — Route 53 + ACM + ALB edge configuration (vidcast-ingress.yaml).....	115
Table 15.2 — Per-app allow-lists (app-policies.yaml). DNS egress is universal; datastore ingress is in Table 15.3.	116
Table 15.3 — Datastore ingress allow-lists (datastore-policies.yaml). Datastores need no egress policy — they never initiate connections.	118
Table 15.4 — The CIDR-based egress exceptions and why each is necessary.....	119
Table 16.1 — Admin role-change guardrails (gateway-service/server.py).	122
Table 16.2 — Every AWS-touching identity is keyless IRSA, scoped least-privilege. (The CD role uses the same OIDC mechanism via GitHub.).....	123
Table 16.3 — Rotation and refresh model (k8s/external-secrets/).	124
Table 17.1 — The single writable /tmp emptyDir per service and what writes to it (k8s/base/*/deployment.yaml).	127
Table 17.2 — Capability and escalation locks (all containers).	128
Table 17.3 — Per-workload hardening coverage. Exceptions are documented, not accidental (Chapter 7, Chapter 10).	130
Table 18.1 — Build-time supply-chain artefacts (docs/SUPPLY_CHAIN.md CI diff).	132
Table 18.2 — Sigstore trust anchors for keyless signing (docs/SUPPLY_CHAIN.md).	133
Table 18.3 — ECR hardening controls (terraform/modules/ecr).	134
Table 18.4 — Honest implementation status (docs/SUPPLY_CHAIN.md).	135
Table 19.1 — The three governing strategy decisions (k8s/kyverno/).	137
Table 19.2 — The seven Kyverno policies, their framework category, and what each enforces.	141
Table 20.1 — Application metrics exposed by the services (the SLI sources for Chapter 22).	144
Table 20.2 — The scrape matrix: which targets are scraped, how, and why pod- versus service-level.	145
Table 20.3 — The three provisioned Grafana dashboards and their panels.	145
Table 21.1 — Fields on every log line (src/*/jsonlog.py, docs/OBSERVABILITY.md §1).	147
Table 21.2 — How each service obtains and logs the correlation_id (src/*/*.py).	149
Table 21.3 — The audit events emitted by the gateway (docs/OBSERVABILITY.md §3).	149

Table 21.4 — The trace of one upload, reconstructed from logs by correlation_id (docs/OBSERVABILITY.md §2).	150
Table 22.1 — The three VidCast SLOs, their targets, windows, and the SLI each is computed from.	151
Table 22.2 — The six burn-rate alerts: each SLO has a fast-burn (page) and slow-burn (ticket) pair, both requiring long AND short windows to be burning.	154
Table 22.3 — Operational alerts: direct signals on pod health, node resources, and the broker.	154
Table 22.4 — SLO runbooks: each burn-rate alert maps to a short, dependency-ordered first response.	155
Table 23.1 — The rate-limit policy: only /login and /upload are throttled; /upload's cost scales with batch size.	156
Table 23.2 — Deployment caveats of the rate-limiting design and their mitigation or status.	158
Table 24.1 — What is backed up, with what tool, when, and where (timestamped object keys).	159
Table 24.2 — Asymmetric IRSA: the cluster can write backups but not read or delete them; restore uses a separate identity.	160
Table 24.3 — RTO/RPO objectives and the most recent drill results (from the DR runbook).	162
Table 25.1 — Operational Excellence: principle → implementation.	166
Table 25.2 — Security: principle → implementation.	167
Table 25.3 — Reliability: principle → implementation.	167
Table 25.4 — Performance Efficiency: principle → implementation.	168
Table 25.5 — Cost Optimization: principle → implementation.	169
Table 25.6 — Sustainability: principle → implementation.	169
Table 26.1 — Monthly cost breakdown (pitch deck slide 25; eu-west-2 list pricing).	170
Table 26.2 — Managed-datastore standing costs vs the self-hosted choice (docs/MANAGED_SERVICES.md).	172
Table 26.3 — Right-sizing and utilisation-tuning levers (pitch deck slide 26).	173
Table 26.4 — FinOps dashboard panels (monitoring/dashboards/vidcast-finops.json).	174
Table 26.5 — Budget position against the cap.	175
Table 26.6 — Parameter Store vs Secrets Manager: the same capability at \$0 (Chapter 9 §9.8).	175
Table 27.1 — The validation gates and what each covers.	177
Table 27.2 — Production validation results (pitch deck slide 27, "Verified Against Targets").	178
Table 27.3 — Security validation results across build, registry, admission and runtime.	180
Table 28.1 — Overview of the four detailed incidents covered in 28.1–28.4.	182
Table 28.2 — The documented incident register: medium/low-severity issues and their resolution or status.	186
Table 29.1 — The five incidents that seeded the chapter's lessons (challenge → root cause → fix → lesson).	187

Table 29.2 — Design decisions revisited: each choice, its rejected alternative, and the trade-off it accepts.	189
Table 30.1 — The roadmap at a glance (pitch deck "Future Roadmap" + docs next-steps).	191
Table 30.2 — The Audit→Enforce promotion. Six policies can flip immediately; verify-images is gated on signing.	193
Table 30.3 — Single-tenant today vs the workspace multi-tenancy roadmap.	194
Table 30.4 — Operational and resilience next-steps, each already documented in the repo.	195
Table 31.1 — The nine headline metrics and the decision each one encodes.	196
Table 31.2 — The six funded objectives and the mechanism that delivers each one.	197
Table 31.3 — Production validation results: every target verified against the live platform.	197

List of Abbreviations and Acronyms

The abbreviations and acronyms used throughout this report.

Abbr.	Meaning	Abbr.	Meaning
ACM	AWS Certificate Manager	IRSA	IAM Roles for Service Accounts
ALB	Application Load Balancer	JWT	JSON Web Token
AMQP	Advanced Message Queuing Protocol	KEDA	Kubernetes Event-Driven Autoscaling
AZ	Availability Zone	mTLS	mutual Transport Layer Security
CI/CD	Continuous Integration / Continuous Delivery	OIDC	OpenID Connect
CORS	Cross-Origin Resource Sharing	PVC	PersistentVolumeClaim
CRD	Custom Resource Definition	RBAC	Role-Based Access Control
CSI	Container Storage Interface	RPO	Recovery Point Objective
CVE	Common Vulnerabilities and Exposures	RTO	Recovery Time Objective
DLQ	Dead-Letter Queue	SBOM	Software Bill of Materials
DNS	Domain Name System	SCP	Service Control Policy
EBS	Elastic Block Store	SCRAM	Salted Challenge Response Auth Mechanism
ECR	Elastic Container Registry	SES	Simple Email Service
EKS	Elastic Kubernetes Service	SLI/SLO	Service Level Indicator / Objective
ESO	External Secrets Operator	SMTP	Simple Mail Transfer Protocol
FinOps	Cloud Financial Operations	SPA	Single-Page Application
GridFS	MongoDB binary chunked file store	SSM	AWS Systems Manager (Parameter Store)
HPA	Horizontal Pod Autoscaler	TLS	Transport Layer Security
HTTP(S)	Hypertext Transfer Protocol (Secure)	TTL	Time To Live
IaC	Infrastructure as Code	VPC	Virtual Private Cloud
IAM	Identity and Access Management	WSGI	Web Server Gateway Interface

Glossary of Terms

Definitions of the key technical terms and patterns used in this report.

Term	Definition
Store-and-queue	Storing an upload immediately and enqueueing the work, returning to the user at once instead of blocking on processing.
Transactional outbox	Writing an event row in the same store as the data, then publishing it via a single relay, so an accepted upload is never silently dropped.
Idempotency	Guaranteeing that processing the same message more than once has the same effect as once (here, via a Redis claim-once on the job ID).
Dead-letter queue	A terminal queue where a message is parked after exhausting its bounded retries, for an operator to inspect.
Error budget	The allowed amount of failure under an SLO (e.g. 0.1% for 99.9% availability) that can be 'spent' before the objective is breached.
Burn rate	How fast the error budget is being consumed, normalised so 1× exactly exhausts it over the SLO window and 14× is fourteen times too fast.
Multi-window multi-burn-rate	Alerting that requires both a long and a short window to be burning, so alerts page on real incidents and clear quickly afterwards.
GitOps	Operating the cluster by reconciling it to a Git repository (via Argo CD), so Git is the single source of truth and rollback is git revert.
Zero-trust networking	A default-deny posture where every pod-to-pod path must be explicitly allow-listed on both ends, containing blast radius.
Blast radius	The extent of damage a single compromised component can cause; minimised by network and identity isolation.
Shift-left security	Catching vulnerabilities and policy violations early — at build time (Trivy) and admission time (Kyverno) — not in production.
Scale-to-zero	Running zero replicas of a workload when idle (KEDA on the converter), so heavy compute costs nothing between jobs.
Reconciliation	Argo CD continuously bringing the live cluster back into line with the committed manifests, correcting drift.
Correlation ID	A unique identifier minted at the gateway and carried through every hop and log line, enabling end-to-end request tracing.
IRSA	IAM Roles for Service Accounts — granting each pod only its own scoped AWS permissions with no long-lived credentials.
Well-Architected	AWS's six-pillar framework (operational excellence, security, reliability, performance, cost, sustainability) used to review the platform.

Part I — Introduction & Business Context

Chapter 1 — Introduction

VidCast is a production-grade, video-to-MP3 conversion platform built on AWS EKS. A user uploads a video, the platform converts it to podcast-ready audio asynchronously, and emails a download link when it is ready. The platform is live at vidcast.online and the source is open at github.com/johnnybabs/vidcast. This report is the engineering record behind that platform: how it was designed, secured, operated and costed to a standard a senior assessor would recognise as production-grade. This opening chapter sets the context (1.1), states the report's purpose and scope (1.2), defines the DevSecOps approach and the principles that governed every decision (1.3), explains how the report is organised and how to read it (1.4), and introduces the team behind it (1.5).

1.1 Project Background and Context

VidCast was built as the capstone project for the Cloudboosta DevOps Engineering bootcamp (June 2026) by a four-person team. Rather than a toy demo, it was framed around a realistic client and a real operational pain, so that every technical decision could be tied to a business need.

The client scenario is Wavelength Media, a London podcast company — 12 shows, 28 staff and 11 freelancers producing 60–80 recordings a week — that had outgrown manual conversion. Producers were turning video recordings (Zoom calls, webinars, conference talks) into podcast audio by hand with one-off FFmpeg commands. That approach cost the business on three fronts: roughly 36 person-hours a week lost to manual conversion (close to a full-time salary); files lost and scattered across drives, laptops and Slack DMs with no answer to "where is my file?"; and a compliance exposure from uploading client content to unknown third-party converter sites. VidCast replaces that ritual with a single, centralised, observable platform.

Architecturally, VidCast implements the store-and-queue pattern used by every media platform at scale — upload, queue, process, store, notify — the same shape as YouTube ingesting a video or Spotify transcoding a podcast, differing only in scale. The headline outcome is that a platform costing roughly £198/month retires a six-figure annual operational drag. Table 1.1 summarises the project at a glance.

Attribute	Value
Product	VidCast — turn video recordings into podcast-ready audio
Client scenario	Wavelength Media (London podcast company); 60–80 recordings/week
Platform	AWS EKS (Kubernetes 1.31), eu-west-2 (London)
Application	5 Python microservices + a single-replica outbox relay
Live URL / repository	https://vidcast.online · github.com/johnnybabs/vidcast
Running cost	~£198/month (≈ 94% under the £3,500/month cap); ~£0 when torn down
Context	Cloudboosta DevOps Engineering bootcamp capstone, June 2026

Table 1.1 — VidCast project at a glance.

1.2 Purpose and Scope of the Report

The purpose of this report is to be the authoritative engineering record for VidCast: to document, in enough depth to be reproduced and assessed, how the platform is architected, how it is secured end to

end, how it is operated and observed, how it recovers from failure, and what it costs. It is written so that an assessor can verify the claims, an engineer can rebuild the system, and the client's technical stakeholders can understand what they are paying for and why.

The report is scoped deliberately:

- **In scope** — the system architecture and event-driven design; the data and messaging layer; infrastructure as code; CI/CD and GitOps delivery; the six-layer security model (perimeter, network, identity & secrets, container, supply chain, policy-as-code); autoscaling; observability and the SLO/alerting framework; abuse protection; disaster recovery and business continuity; cost engineering; production validation; and the challenges, lessons and roadmap.
- **Out of scope** — exhaustive source-code listings (the repository is the source of truth and is referenced throughout); internal study aids kept out of version control; and features that are designed and costed but not yet shipped (for example S3 pre-signed URLs, Kyverno Enforce mode, Cosign image signing, Amazon SES and log shipping to Loki), which appear only as a clearly-labelled roadmap.

Where the platform's behaviour and its documentation or slides differ, this report follows the platform as it is actually configured in the repository, and notes the discrepancy rather than papering over it. Honesty about gaps — documented trust boundaries, known low-severity issues, demonstrative versus guaranteed targets — is treated as part of the engineering record, not an embarrassment to hide.

1.3 Definition of DevSecOps and Guiding Principles

DevSecOps is the practice of building security into every stage of the software delivery lifecycle — planning, coding, building, testing, releasing, deploying, operating and monitoring — rather than bolting it on at the end. It extends the DevOps idea of shared ownership between development and operations to include security as a shared, automated responsibility, and it "shifts security left" so that vulnerabilities are caught at build time and policy is enforced at admission time, not discovered in production. Figure 1.1 shows how VidCast realises this: a continuous loop of lifecycle stages, each backed by a concrete tool, with security woven through every stage rather than confined to one.

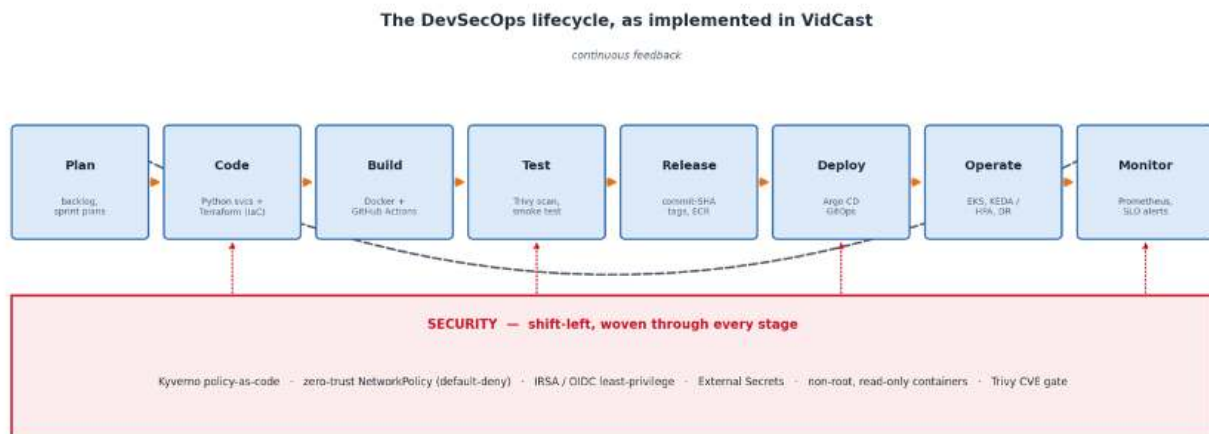


Figure 1.1 — The DevSecOps lifecycle as implemented in VidCast: each stage is backed by a specific tool, and security (shift-left) is woven through all of them.

Eight principles guided the build and recur throughout the report. They are not abstract ideals — each is a direct response to a constraint of the client (a tight budget, UK-resident client content, a small operations team) and each is verifiable in the codebase.

Principle	What it means in VidCast
Git is the source of truth (GitOps)	Argo CD reconciles the cluster to the repository; rollback is git revert; prod sync is a manual human gate
Everything as code	Terraform for all AWS resources; Kustomize/Helm for Kubernetes; Kyverno for policy — reproducible, diffable, reviewed
Shift-left security	Trivy fails the build on fixable CRITICAL/HIGH CVEs; Kyverno enforces the baseline at admission
Zero-trust networking	Default-deny NetworkPolicies; every pod-to-pod path is explicitly allow-listed on both ends
Least privilege	IRSA scopes each pod to its own IAM role; OIDC gives keyless CI; backups use a write-only role
Async by design	The gateway returns 202 immediately; conversion flows through RabbitMQ with outbox, idempotency and DLQs
Observability with SLOs	Metrics, structured logs and correlation IDs feed three SLOs with multi-window burn-rate alerts
FinOps & reproducibility	Cost-capped, scale-to-zero, off-when-idle; rebuilds in < 20 min and recovers in < 2 h

Table 1.2 — The eight guiding principles and their concrete expression in the platform.

1.4 Report Structure and How to Read It

The report is organised into six thematic parts that mirror the lifecycle and the Well-Architected pillars. Each part contains several numbered chapters that go to implementation depth; the parts and their coverage are summarised in Table 1.3, with a guide to which reader each is most relevant to.

Part	Covers	Read this if you are...
I — Why & What	Introduction, the problem, objectives, business case and pitch	An executive or assessor wanting the context and the case
II — Architecture	System design, the five microservices, data & messaging layer, reliability patterns	An architect or engineer learning how it is built
III — Dev: Build & Deliver	Infrastructure as code, CI/CD, GitOps, autoscaling, disaster recovery	A platform/DevOps engineer rebuilding or operating it
IV — Sec: Defence in Depth	The six security layers, zero-trust, supply chain, policy-as-code, abuse protection	A security reviewer assessing the posture
V — Ops & Well-Architected	Observability, SLOs and alerting, the six AWS Well-Architected pillars	An SRE or reliability reviewer
VI — Cost, Validation & Lessons	Cost analysis and savings, production validation, challenges, lessons, roadmap, conclusion	Anyone judging outcomes, value and maturity

Table 1.3 — The six parts of the report, their coverage, and the intended reader for each.

How to read it: a reader short on time can take this introduction, the executive summary in Part I, and the conclusion to get the whole story; the detailed chapters are self-contained and cross-referenced, so a reader can go straight to the part that matters to them — architecture, security, operations or cost — without reading front to back. Figures and tables are numbered by chapter, and every technical claim is traceable to a file in the repository so it can be checked rather than taken on trust.

1.5 Team, Roles and Responsibilities

VidCast was delivered by a four-person DevOps Engineering team on the Cloudboosta bootcamp. A deliberate goal of the project was that the resulting platform be operable by a two-person team — nothing tribal, everything in Git — so the team worked with shared ownership and deliberately documented runbooks rather than siloing knowledge. The indicative lead responsibilities, organised around the DevSecOps workstreams, were as follows.

Team member	Lead responsibility	Primary workstream
Samuel Onyejekwe	Architecture & project lead	System design, infrastructure as code, documentation and this report
John Babalola	Platform & GitOps engineer	Repository owner; EKS, Terraform, CI/CD and Argo CD delivery
Seun Okegbola	Security & compliance engineer	Zero-trust networking, Kyverno policy-as-code, supply chain and secrets
Chinedu Izuelu	Observability & reliability engineer	Metrics/SLOs, alerting, autoscaling, and disaster recovery

Table 1.4 — The team and their lead responsibilities across the DevSecOps workstreams.

These are lead areas, not exclusive ones: in keeping with the DevSecOps principle of shared responsibility, security and operability were owned by everyone, code review crossed workstreams, and the production approval gate meant a second pair of eyes reviewed every change before it reached vidcast.online. That same discipline — auditable, automated, GitOps-driven delivery operable by a small team — is the thread that runs through the rest of this report.

Chapter 2 — Problem Statement

Every engineering decision in this report traces back to a single business problem. VidCast was not built to showcase Kubernetes — it was built because a real-world media company had outgrown a manual, error-prone, off-platform way of turning video recordings into podcast audio. This chapter states that problem precisely: the company and the scale it operates at (§2.1), the labour cost of doing conversion by hand (§2.2), the loss and disorganisation of the resulting files (§2.3), and — most seriously — the compliance and data-residency exposure of pushing client content through unknown third-party sites (§2.4).

Stated plainly, the status quo was *a chaotic weekly ritual of manual FFmpeg commands, lost files and unauditable client content*. Those three failures are not independent — they compound. Manual conversion creates labour cost; the manual hand-off scatters files and erases visibility; and the off-platform tools used to

do the conversion are exactly what create the compliance exposure. The four problems and their shared root are summarised below.

Figure 2.1 — Four Compounding Problems of Manual Conversion

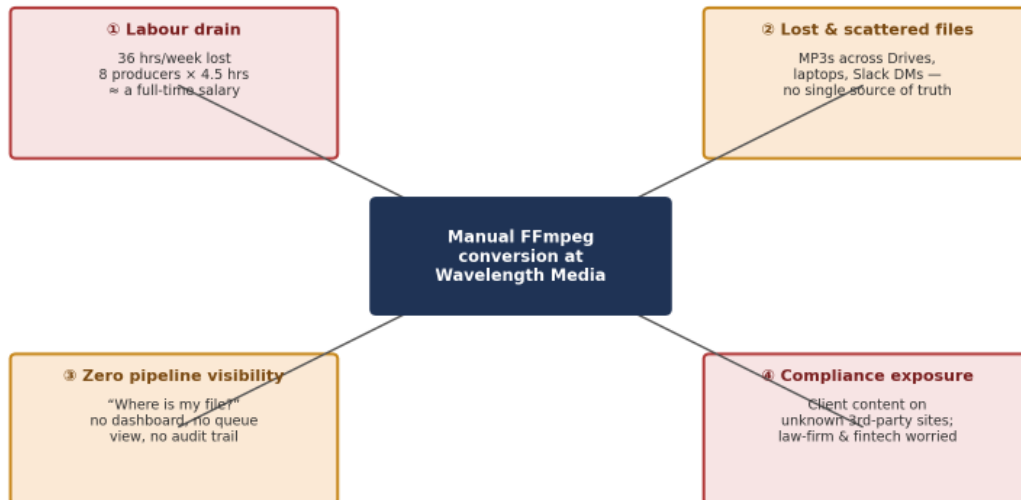


Figure 2.1 — The four compounding problems, all stemming from one root: manual, off-platform conversion. Each is examined in this chapter and answered by the objectives in Chapter 3.

2.1 Case Study — Wavelength Media

The client is **Wavelength Media**, a London-based podcast production company. It produces a substantial and *growing* volume of content: creators record videos — Zoom calls, webinars, conference talks — and need the audio extracted as a standalone podcast episode. The scale of the operation is what turns an occasional manual task into a structural problem.

Figure 2.2 — Wavelength Media at a Glance (a London podcast company)



A growing volume of recordings (Zoom calls, webinars, conference talks) that all need the audio extracted as a podcast.

Figure 2.2 — Wavelength Media at a glance. The volume (60–80 recordings/week) is the multiplier that makes a manual per-file task unsustainable.

Dimension	Figure	Why it matters
Shows	12	twelve concurrent content streams, each producing episodes on a cadence
Staff	28	a mid-sized operation — large enough to feel the cost, too small to absorb it
Freelancers	11	external contributors handing content in and out, widening the file-handling surface
Recordings / week	60–80	the core driver — every one needs the audio extracted
Producers doing conversion	8	the people whose time the manual task consumes (\$2.2)

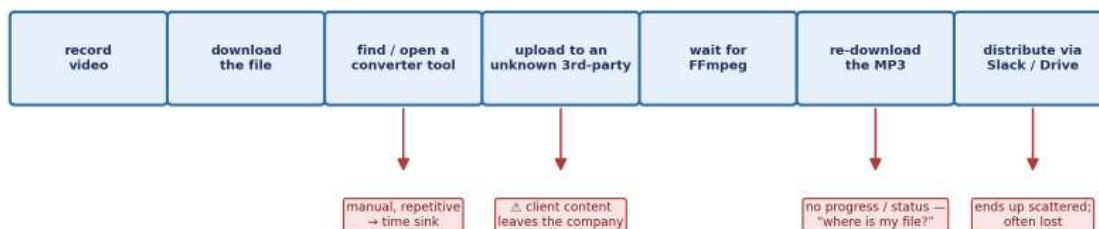
Table 2.1 — The Wavelength Media operating profile (source: project case study / pitch deck).

Two characteristics of this profile are decisive. First, the **volume** (60–80 recordings per week) means conversion is not a now-and-then chore but a continuous, every-week load. Second, the **mix of staff and freelancers** (28 + 11) means content is constantly changing hands between people and devices — which, as §2.3 shows, is precisely where files get lost. A company at this scale needs a **system**, not a habit.

2.2 Manual FFmpeg Conversion and 36 Lost Person-Hours per Week

The conversion itself was done by hand. A producer would download the recording, find or open a converter tool, run a one-off **FFmpeg** command (or upload to a web converter), wait, re-download the resulting MP3, and then distribute it. Every step is human-driven and repeated for every single recording.

Figure 2.3 — The Manual Conversion Workflow (and where it breaks)



Every step is human-driven, untracked, and off-platform — the root of all four problems in Figure 2.1.

Figure 2.3 — The manual conversion workflow. Seven human steps per file, none of them tracked, several of them off-platform — the breakages mapped here become the four problems of Figure 2.1.

The labour cost

Across the team this added up to a startling figure: **36 person-hours every week lost to manual conversion** — calculated as **8 producers × 4.5 hours each**. That is, by the company's own reckoning, **nearly a full-time salary's worth of time** spent every week on a task that should be automated.

Figure 2.4 — The Labour Cost of Manual Conversion

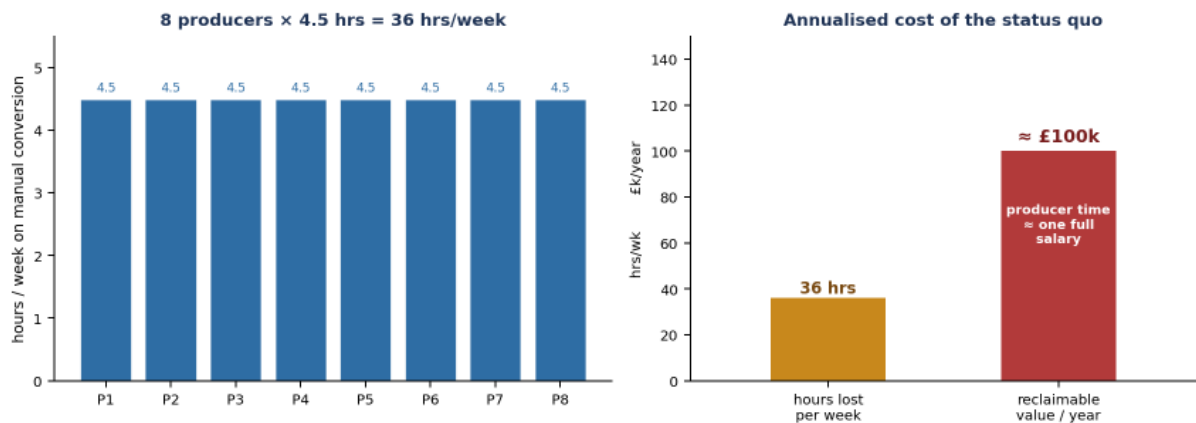


Figure 2.4 — The labour cost. The 36 weekly hours annualise to roughly £100k of producer time — a six-figure drag the platform was built to retire.

Measure	Value	Note
Producers affected	8	the staff performing conversion
Time each, per week	4.5 hrs	manual download → convert → re-download → distribute
Total time lost / week	36 person-hours	≈ a full-time-equivalent of effort
Annualised value	≈ £100,000 / year	producer time consumed by an automatable task
Nature of the work	repetitive, low-value	time that should be billable, creative work

Table 2.2 — The cost of manual conversion (source: business case, pitch deck slide 6).

Why this number drives everything. The ~£100k/year of lost producer time is the financial anchor of the whole project — the pitch "led with the business problem, not the technology," framing the pain in money first. It is also the benchmark the solution is measured against: a platform costing roughly £2,400/year to run against ~£100k/year of reclaimed time is a >40× return (developed fully in Chapter 4). The cost of the problem is what justifies building the solution.

Beyond the raw hours, the manual process had a quality cost: **client deliverables were delayed, and episodes slipped over codec issues** — a one-off FFmpeg command that worked last week could fail on a differently-encoded file this week, with no standard pipeline to fall back on. The task was not only expensive but unreliable.

2.3 Lost and Scattered Files — No Single Source of Truth

Because conversion happened on individual laptops and through ad-hoc tools, the *outputs* had nowhere central to live. The resulting MP3s ended up **scattered across personal Drives, laptops, and Slack DMs** — with no catalogue, no naming discipline, and no authoritative copy. The company’s own description captures it: *"not sure we still have it."*

This produced two distinct failures that compound each other:

- **No single source of truth.** The same file might exist in three places, or none. There was no canonical store a producer could point to and say "this is the master." With 60–80 new files a week flowing between 28 staff and 11 freelancers, the scatter grew continuously.
- **Zero pipeline visibility.** The question *"where is my file?"* simply had no answer — there was **no dashboard, no queue view, and no audit trail**. A producer could not tell whether a conversion was queued, in progress, finished, or lost; tracking a file down meant asking around.

Symptom	Manifestation	Consequence
Scattered storage	MP3s on Drives, laptops, Slack DMs	no master copy; duplicated and inconsistent files
No catalogue	no central index of what exists	"not sure we still have it" — files genuinely lost
No status tracking	no queue/dashboard view	"where is my file?" is unanswerable
No audit trail	no record of who did what, when	cannot reconstruct a file's history

Table 2.3 — The file-management failures of the status quo.

These are not merely inconveniences — for a business whose product *is* the delivered audio, losing a deliverable or being unable to locate it is a direct hit to client trust and revenue. The absence of a system of record is the operational heart of the problem, and it is what the platform's durable storage, central catalogue, and correlation-ID tracing (Chapter 3) were designed to end.

2.4 Compliance and Data-Residency Exposure (Client Content on Third-Party Sites)

The most serious problem is not cost or convenience — it is exposure. To convert files, producers were **uploading client content to unknown third-party converter websites**. That single habit moves confidential client material *outside the company's control* and creates a compliance liability that the company's own clients had begun to question.

Figure 2.5 — Compliance & Data-Residency Exposure (the status quo)

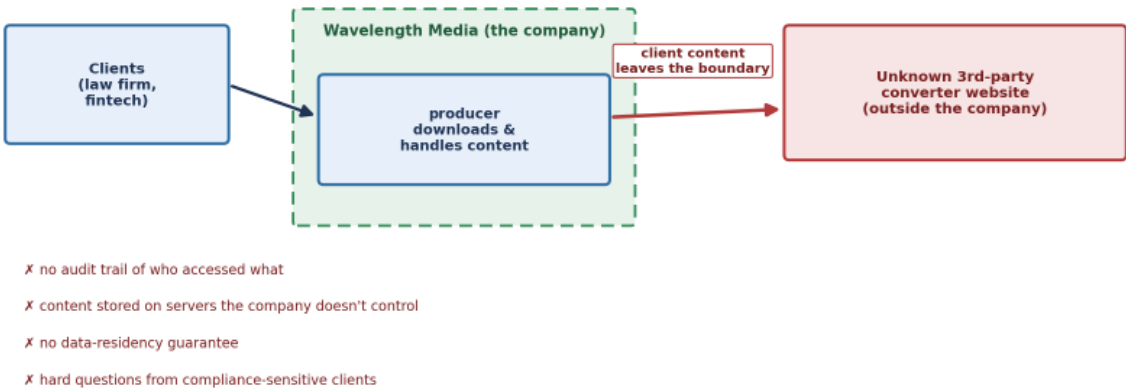


Figure 2.5 — The data-residency exposure. Client content leaves the company boundary to an unknown third-party site, with no audit trail and no control over where it is stored.

Wavelength Media's clients include a **law firm and a fintech** — precisely the kind of organisations with strict obligations over where their data goes and who can touch it. They were "asking hard questions," and the honest answers were uncomfortable:

- **Content left the company boundary.** Files were uploaded to servers the company does not own or control, with no contractual or technical guarantee of how the content would be handled or retained.
- **No data-residency guarantee.** There was no assurance the content stayed in a particular jurisdiction — a serious gap for UK clients with regulatory and contractual data-locality requirements.
- **No audit trail.** The company could not say who accessed a file, when, or where it was processed — so it could not answer a client's or regulator's questions even after the fact.
- **Unvetted third parties.** The "converter site" was an unknown quantity: its security posture, retention policy, and trustworthiness were all unverified.

From liability to asset. This exposure is the problem with the highest stakes, and it shapes a core design objective: keep all content **inside the company's own AWS account** with a full **download audit trail** (Chapter 3). The platform turns a compliance **liability** — content on unknown third-party servers with no record — into a compliance **asset**: content that never leaves the company boundary, with an auditable record of every access.

In summary, Wavelength Media — a 28-person London podcast company producing 60–80 recordings a week across 12 shows — was converting audio by hand. The manual FFmpeg workflow consumed **36 person-hours a week (≈ £100k/year)** of producer time on a task that should be automated; it scattered the resulting files across Drives, laptops and Slack with **no single source of truth and no visibility** into where any file was; and, most seriously, it pushed **confidential client content through unknown third-party sites**, creating a data-residency and compliance exposure that the company's own law-firm and fintech clients had begun to challenge. These four compounding problems — labour drain, lost files, zero visibility, and compliance exposure — define exactly what the platform had to solve, and they map directly onto the project objectives set out in Chapter 3 and the business case quantified in Chapter 4.

Chapter 3 — Project Objectives and Success Criteria

Chapter 2 set out four compounding problems. This chapter turns each into a concrete objective with a measurable success criterion, so the platform's value can be judged against evidence rather than assertion. The objectives are deliberately framed as outcomes for the business — centralise, never lose a file, make the pipeline visible, keep content in-house, and do it within a hard budget — not as a feature list. Each maps one-to-one onto a Chapter 2 problem, and each is delivered by a mechanism that the later chapters of this report detail.

Figure 3.1 — From Problem (Ch.2) to Objective (Ch.3)

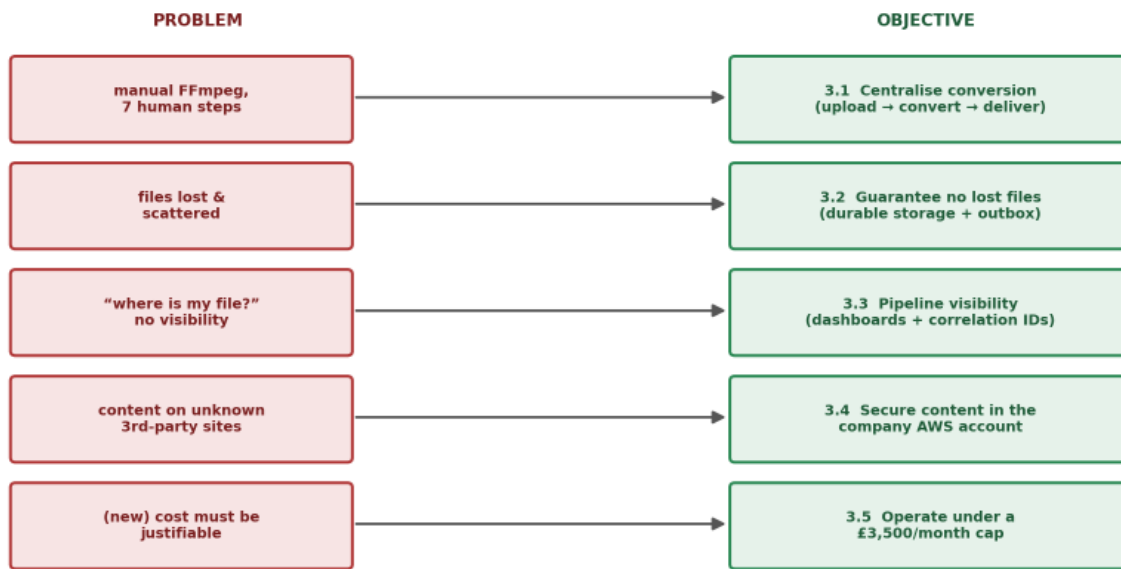


Figure 3.1 — Each objective answers a specific Chapter 2 problem. This chapter defines the success criterion for each; the cited later chapters describe how it was built.

Two principles govern the whole set. First, **success must be measurable** — wherever possible an objective has a numeric target taken from the project's own Service Level Objectives (`docs/SLO.md`) or production-validation results, not a vague aspiration. Second, an explicit **scope boundary** (§3.6) records what was deliberately *not* attempted, so the objectives are honest about the limits of a single-node, cost-capped demonstration platform.

#	Objective	Answers (Ch.2)	Headline success criterion	Built in
3.1	Centralise conversion	§2.2 manual FFmpeg	one platform: upload→convert→deliver	Ch.5–6
3.2	Guarantee no lost files	§2.3 lost files	durable storage + transactional outbox; 99% E2E	Ch.7–8
3.3	Pipeline visibility	§2.3 no visibility	real-time dashboards + correlation IDs	Ch.20–22
3.4	Secure client content	§2.4 compliance	content stays in the company AWS account + audit	Ch.16
3.5	Operate within budget	(new) cost discipline	run under £3,500/month	Ch.26
3.6	(Operable by two engineers)	cross-cutting	GitOps, no tribal knowledge	Ch.9,12

Table 3.1 — The objectives, the problems they answer, and where each is delivered in this report.

3.1 Centralise Conversion (Upload → Convert → Deliver)

Objective. Replace the ad-hoc, per-laptop FFmpeg ritual with a single platform that handles the whole lifecycle: a producer uploads one video, the platform converts it, and the producer receives the result — no tool-hunting, no manual commands, no off-platform steps.

Figure 3.2 — The Target: one platform, three steps, fully tracked

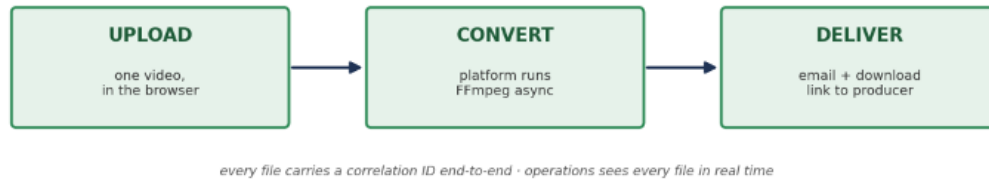


Figure 3.2 — The target operating model: one platform, three steps, every file tracked end-to-end — replacing the seven manual steps of Figure 2.3.

Concretely, the target user journey is: **upload a video → the platform converts it → the producer gets an email and a download link → operations sees every file in real time**. The conversion runs **asynchronously** (the upload returns immediately and FFmpeg runs behind a queue), so a producer is never blocked waiting at their machine.

Success criteria

- A single web interface performs upload, conversion, and delivery — the manual download/convert/re-upload steps are eliminated entirely.
- Conversion is automated and standardised (one pipeline, not one-off commands), removing the codec-issue failures of the manual process.
- The producer is notified on completion (email + download link) rather than polling or asking around.

This objective is realised by the five-microservice, event-driven architecture (Chapter 5) and the gateway/converter/notification services (Chapter 6); the conversion latency itself is held to a target in §3.2/§3.3.

3.2 Guarantee No Lost Files (Durable Storage + Transactional Outbox)

Objective. Make it structurally impossible for an accepted upload to be silently dropped. Where the manual process scattered and lost files (§2.3), the platform must provide durable storage and a delivery guarantee.

The mechanism is twofold and specific: **durable storage** (the file is persisted before anything else happens) plus a **transactional outbox** so that the act of storing a file and the act of queuing it for conversion cannot diverge — an upload is never accepted-but-forgotten. The reliability patterns behind this (transactional outbox A1, idempotent consumers A2, bounded retry/DLQ A3) are detailed in Chapter 8.

Success criteria

Criterion	Target	How measured / source
End-to-end success	≥ 99% of uploads produce a notification	SLO 3: <code>notifications_total{success} ÷ uploads_total</code> (30-day window)
No silent drops	an accepted upload is always queued or	transactional outbox + compensating delete

	recoverable	(A1)
Single source of truth	one durable, owner-tagged copy per file	GridFS with <code>owner_email</code> + <code>job_status</code> record
Survives restarts	data persists across pod reschedules	EBS-backed PVCs + nightly S3 backups

Table 3.2 — Success criteria for "no lost files" (targets from docs/SLO.md).

The headline target — **99% of uploads result in a delivered notification** — is the project's third formal SLO, and it directly answers the "not sure we still have it" failure of the status quo.

3.3 Provide Pipeline Visibility (Dashboards + Correlation IDs)

Objective. Make the question *"where is my file?"* answerable in minutes rather than never. The status quo had no dashboard, no queue view, and no audit trail (§2.3); the platform must give operations real-time insight into every file and every stage.

Two mechanisms deliver this. **Real-time dashboards** (Grafana over Prometheus) show pod health, queue depth, request rates, and SLO burn-down. **Correlation IDs** — a unique ID minted at the gateway for every upload and carried through every hop (gateway → queue → converter → notification) — mean a single file can be traced across the whole pipeline from the logs. Together they turn an opaque process into an observable one (detailed in Chapters 20–22).

Success criteria

Criterion	Target	Source
Availability	≥ 99.9% of gateway requests non-5xx	SLO 1 (30-day window)
Conversion latency	95% of conversions finish ≤ 5 min	SLO 2 (1e="300" bucket)
Traceability	any file locatable end-to-end via one ID	correlation ID minted at gateway, carried every hop
Operational view	real-time pod/queue/SLO dashboards	kube-prometheus-stack + Grafana

Table 3.3 — Visibility success criteria, anchored to the formal SLOs (docs/SLO.md).

The three SLOs (availability 99.9%, conversion latency 95% ≤ 5 min, end-to-end success 99%) are the measurable backbone of this objective — they make "visibility" concrete, with burn-rate alerting (Chapter 22) firing before a target is breached.

3.4 Secure Client Content Within the Company AWS Account

Objective. Eliminate the data-residency and compliance exposure of §2.4 by keeping all client content **inside the company's own AWS account**, with a **full download audit trail** — turning the biggest liability of the status quo into a defensible, auditable asset.

This is the highest-stakes objective because it addresses the law-firm and fintech clients' "hard questions." The success condition is categorical: content never traverses an unknown third-party site again. Supporting it are the in-account storage (GridFS on EBS within the cluster's VPC), the zero-trust perimeter that keeps the datastores private (Chapter 15), the secrets discipline that keeps credentials out of Git (Chapter 16), and a download audit trail.

Success criteria

- **Content residency:** all uploaded and converted media remains inside the company AWS account (eu-west-2, London) — no third-party processing services.
- **Audit trail:** every download is recorded, so the company can answer "who accessed what, when."
- **Access control:** authentication (JWT) and role-based access (RBAC) gate who can upload and download (Chapter 16).
- **Defence in depth:** the six-layer security model (Chapter 14) ensures content is protected at the perimeter, network, identity, container, supply-chain and policy layers.

UK data residency by design. The single EKS cluster runs in eu-west-2 (London) specifically for UK data residency — a deliberate choice tied to this objective. Keeping content in-account is what converts the compliance *liability* of \$2.4 into a compliance *asset*: a contained boundary with an auditable record.

3.5 Operate Within a Defined Monthly Cost Cap (< £3,500/month)

Objective. The platform must be demonstrably cheaper than the problem it solves. The agreed ceiling is **£3,500/month — roughly the cost of one extra headcount** — so the spend is always justifiable against the ~£100k/year of producer time it reclaims (Chapter 4).

Figure 3.3 — Success criterion 3.5: run far under the £3,500/month cap

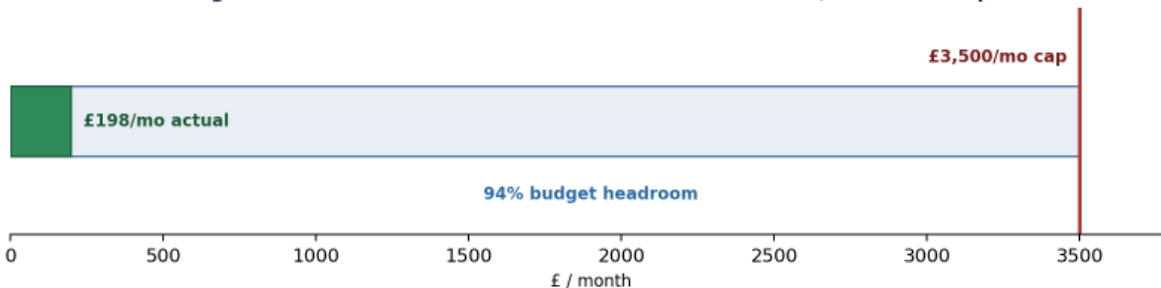


Figure 3.3 — The cost criterion. Actual run cost (~£198/month) sits far under the £3,500/month cap, leaving ~94% headroom to absorb growth.

The success criterion is not merely "under cap" but "**far** under cap, with room to grow." The platform runs at roughly **£198/month** — about 5–6% of the ceiling — leaving **~94% budget headroom**. This is achieved through deliberate FinOps choices (scale-to-zero on the converter, self-hosted datastores, right-sized nodes, tear-down when idle) detailed in Chapter 26.

Success criteria

Criterion	Target	Actual
Monthly run cost	< £3,500/month (cap)	≈ £198/month
Budget headroom	positive, room for growth	≈ 94% under cap
Cost vs value	clearly < reclaimed time	≈ £2,400/yr vs ≈ £100k/yr (>40×)
Idle cost	minimise when not in use	≈ £0 when torn down (terraform destroy)

Table 3.4 — Cost success criteria (targets from the pitch; actuals from Chapter 26 cost analysis).

3.6 Scope, Assumptions and Out-of-Scope Items

Clear objectives need clear boundaries. This section records what the project assumed, and — importantly — what it deliberately did **not** attempt. The out-of-scope list is taken verbatim from the project's own planning record ([PHASE_UP_PLAN.md §2.2](#)), where each omission is documented as a conscious trade-off rather than a gap.

Operating assumptions

- **Single-node, single-region demonstration platform.** One EKS node group in eu-west-2; the node is the availability boundary. This is accepted for a portfolio/demo cluster and noted as a single-point-of-failure trade-off.
- **Operable by a two-person team.** A cross-cutting objective: everything lives in Git and is GitOps-driven, with no tribal knowledge or manual terminal steps — so the platform is auditable and recoverable by a small team (Chapters 9 and 12).
- **Cost-first posture.** The cluster is deliberately torn down when idle to drive standing cost toward zero, on the assumption that a ~20-minute rebuild is acceptable between demos.
- **Incremental delivery.** Built in five merged-to-main sprints, delivering value early and keeping risk low.

Explicitly out of scope (deliberate omissions)

Item	Decision	Rationale (documented)
Service mesh (Istio/Linkerd)	parked	NetworkPolicy + Kyverno cover ~80% of the value without the operational weight
Multi-region HA	out of scope	single-region SPOF accepted for a demo; HA would need cross-region RDS replica + Route 53 failover + DocumentDB global — cost-prohibitive
Switching IaC tool	no	Terraform stays
Single-CI consolidation	no	both GitHub Actions and Jenkins kept; the Jenkins manual gate is a strength, relocated (not removed) by the Argo CD migration
Managed-datastore cutover	deferred	documented and costed, not adopted — keeps standing cost at £0 (Ch.7 §7.6)

Table 3.5 — Out-of-scope items and the documented reason for each ([PHASE_UP_PLAN.md §2.2](#)).

Recording these omissions explicitly is itself a quality marker: each is a reasoned engineering trade-off appropriate to a cost-capped, single-node platform, not an oversight. Where a boundary has a clear production upgrade path (multi-region HA, managed datastores, a service mesh), that path is documented so the platform can grow into it without redesign.

In summary, VidCast's objectives translate the four problems of Chapter 2 into five measurable goals plus a cross-cutting operability goal: centralise conversion into one upload→convert→deliver platform; guarantee no lost files via durable storage and a transactional outbox (99% end-to-end success); make the pipeline visible through dashboards and correlation IDs (99.9% availability, 95% of conversions ≤ 5 min); keep all client content inside the company AWS account with a full audit trail; and run far under a £3,500/month cap (≈ £198/month, ~94% headroom). Each objective carries a numeric success criterion drawn from the project's own SLOs and cost figures, and the scope section records — honestly and verbatim from the planning record

— what was deliberately left out and why. These objectives are the yardstick against which the rest of this report’s architecture, security, operations and cost work should be judged.

Chapter 4 — Business Case and Value Proposition

VidCast was justified in business terms before a line of infrastructure was written. The case is simple and large: a task that consumes roughly £100,000 a year of producer time is replaced by a platform that costs roughly £2,400 a year to run — a return of more than forty times its running spend, before counting the compliance liability it removes. This chapter sets out the cost of the status quo (4.1), the value the platform delivers and its return on investment (4.2), the compliance and risk reduction that sits alongside the raw economics (4.3), and how the solution was pitched and funded against a hard cost cap (4.4).

4.1 Cost of the Status Quo (~£100k/year of Producer Time)

Before VidCast, Wavelength Media converted video to podcast audio by hand. Eight producers each spent about 4.5 hours a week running one-off FFmpeg commands — roughly 36 person-hours every week, close to a full-time role's worth of effort, spent on a task that should be automated. Annualised, that is on the order of 1,870 hours a year and approximately £100,000 of producer time (Figure 4.1). It is the single largest order line in the business case and the one the platform attacks directly.

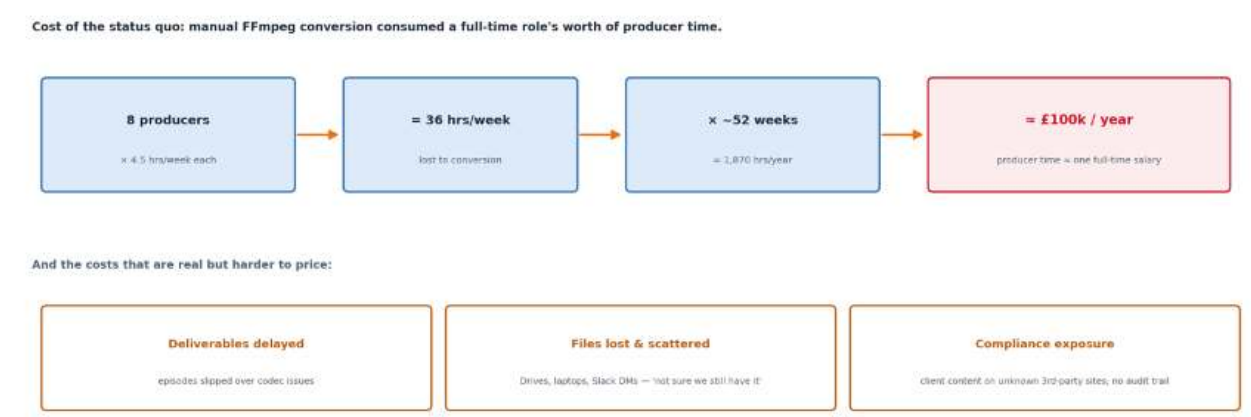


Figure 4.1 — Cost of the status quo: the producer-time calculation to ~£100k/year, plus the real-but-harder-to-price costs of delays, lost files and compliance exposure.

The producer-time figure is not the whole cost. Three further costs were real but harder to price, and each maps to a project objective the platform was funded to meet.

Cost driver	Quantification	Business impact
Producer time on manual conversion	8 producers × 4.5 h/wk = 36 h/wk ≈ £100k/yr	A near-full-time role consumed by an automatable task
Delayed deliverables	Not directly priced	Client episodes slipped, often over codec issues
Lost & scattered files	Not directly priced	MP3s spread across drives, laptops, Slack DMs — no single source

Zero pipeline visibility	Not directly priced	"Where is my file?" had no answer — no dashboard, queue view or audit trail
Compliance exposure	Not directly priced (liability)	Client content uploaded to unknown third-party converter sites

Table 4.1 — The cost of the status quo: one quantified driver (~£100k/yr) and four real-but-unpriced costs.

4.2 Value Delivered and Return on Investment (~£2,400/year to Run)

Against that ~£100,000/year of consumed producer time, VidCast costs approximately £198 a month — about £2,400 a year — to run, in eu-west-2. As Figure 4.2 shows, that is only 5.7% of the client's £3,500/month budget cap, leaving roughly 94% headroom to absorb growth. The dominant line items are the two m7i-flex.large worker nodes (~£114) and the EKS control plane (~£60); the ALB, Route 53 and S3 backups together add only ~£24.

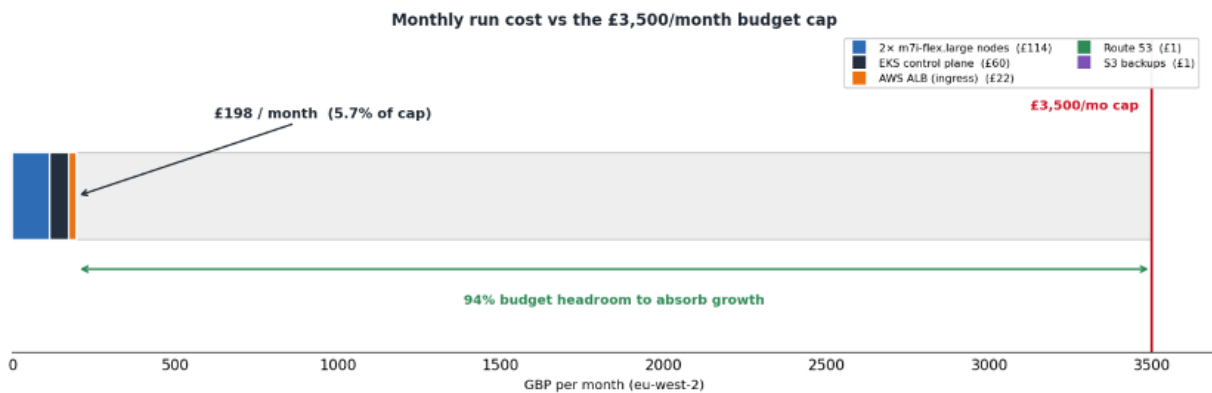


Figure 4.2 — Monthly run cost (~£198) broken down against the £3,500/month cap: the platform uses under 6% of the budget, leaving 94% headroom.

The return is therefore stark. Reclaiming ~£100,000/year of producer time for an outlay of ~£2,400/year is a greater-than-forty-times return on running spend, and a net annual value of roughly £97,600 — and that is before the platform can be torn down to ~£0 when idle and rebuilt in under twenty minutes, which trims even the £2,400 in practice. Each cost of the status quo is matched by a concrete capability that removes it.

Measure	Value
Cost of the status quo (producer time)	≈ £100,000 / year
Cost to run VidCast	≈ £2,400 / year (~£198 / month)
Net annual value	≈ £97,600 / year
Return on running spend	> 40×
Budget utilisation / headroom	5.7% of the £3,500/month cap · ~94% headroom
Cost when torn down (idle)	≈ £0 (rebuilds in < 20 minutes)

Table 4.2 — Return on investment: ~£2,400/year to run reclaims ~£100,000/year of producer time.

Beyond the headline return, the platform converts the unpriced costs of 4.1 into delivered value: producer hours return to billable, creative work; a central catalogue with per-request correlation IDs ends the "lost file" problem and answers "where is my file?" in minutes; and real-time dashboards give the pipeline visibility that did not exist before.

4.3 Compliance, Audit Trail and Risk Reduction

For Wavelength Media the compliance dimension is not a side benefit — it is a requirement driven by their own clients, who include a law firm and a fintech. Those clients ask hard questions about where their content goes, and the status quo had the worst possible answer: recordings were uploaded to unknown third-party converter sites with no record of who accessed what. VidCast turns that liability into an asset by keeping client content inside the company's own AWS account and instrumenting access.

Risk under the status quo	Control in VidCast	Outcome
Client content on unknown 3rd-party sites	Content stays in the company's UK (eu-west-2) AWS account	Data residency and custody are provable
No record of who downloaded what	Structured download-audit log line per request	A queryable audit trail — a compliance asset
"Where is my file?" unanswerable	Correlation ID minted at the gateway, carried every hop	Any file traceable end-to-end in minutes
Secrets and access ungoverned	Zero secrets in Git; default-deny networking; least-privilege IRSA	Access is scoped, auditable and reviewable
Unrecoverable data loss	Nightly encrypted S3 backups; tested < 2 h recovery	Business continuity is demonstrable, not hoped-for

Table 4.3 — Compliance and risk reduction: each status-quo risk is matched by a concrete platform control.

The shift is qualitative as well as quantitative: the same content that was a compliance liability — sitting on servers the company did not control, with no audit trail — becomes an auditable, in-account asset. That is precisely the kind of control the law-firm and fintech clients require, and it materially de-risks the client relationships that drive Wavelength Media's revenue.

4.4 How the Solution Was Pitched and Funded (Pain → Outcome → Cost Cap)

The pitch deliberately led with the business problem, not the technology, and tied every spend to a pain point. It followed five steps (Figure 4.3), moving from the money at stake to a proof that the solution was production-grade rather than a prototype.



Figure 4.3 — The five-step pitch: frame the pain in money, propose one outcome, anchor to the cost cap, de-risk with phasing, and prove it is production-grade.

- **Framed the pain in money** — "You are losing ~£100k/year of producer time and exposing client content to unknown sites."
- **Proposed one outcome** — a single platform: upload → convert → deliver, with full pipeline visibility and an audit trail.
- **Anchored to the cap** — designed to run under £3,500/month and to be operable by the existing two-person team.
- **De-risked with phasing** — delivered in five incremental sprints, each merged to main — value early, risk low.
- **Proved it production-grade** — a live URL, tested disaster recovery, security hardening and dashboards — not a slide-ware demo.

The cost cap was the anchor that made the case fundable. At ~£198/month the platform sits at under 6% of the £3,500/month ceiling — roughly the cost of one extra headcount, against the near-full headcount of producer time it returns — so the decision was never "can we afford it?" but "why would we not?". Phasing the build into five sprints, each independently merged and demonstrable, meant the client saw value accrue incrementally rather than betting on a big-bang delivery, and the final proof points — a live service at vidcast.online, a tested sub-two-hour recovery, and real dashboards — answered the only remaining question an assessor or sponsor asks of any platform: is it actually real? It is.

Part II — Solution Architecture & Design

Chapter 5 — Architecture Overview

This chapter describes how VidCast is put together as a running system. VidCast is an event-driven microservices platform that converts uploaded video into MP3 audio for Wavelength Media, a London podcast company. The chapter opens with the system model and the store-and-queue pattern that underpins it (5.1), states the design principles that governed every structural choice (5.2), presents the end-to-end architecture with the Application Load Balancer as the single public entry point (5.3), documents the port map and the zero-trust network topology that connects the components (5.4), and closes with the three environments the platform runs in — production on EKS, staging on Docker Swarm, and local Docker Compose (5.5). Every figure and table is drawn from the project's own Terraform, Kubernetes manifests, and service code rather than a generic reference design.

5.1 System Overview and Event-Driven (Store-and-Queue) Design

VidCast is built around a single idea: a user should never wait for a slow job to finish. When a producer uploads a recording, the platform stores the file immediately, places a message on a queue, and returns control to the user in milliseconds. Worker pods then pick the message up asynchronously, run the FFmpeg conversion, and trigger an email when the MP3 is ready. This is the store-and-queue pattern — store the work and enqueue it, rather than store-and-block while the conversion runs — and it is the same model used by YouTube, TikTok, and Spotify for media processing at scale.

The platform is decomposed into five application microservices plus a single-replica outbox relay, each with a single responsibility and its own deployment lifecycle:

- **Frontend** — React 18 + Vite SPA served by nginx; the producer's web interface for upload, history, and downloads. nginx also proxies /api/ to the gateway.
- **Gateway** — Flask + Gunicorn; the API surface. Handles upload, download, rate limiting, and the structured download-audit log. The single ingress point for application traffic.
- **Auth** — Flask + bcrypt (cost-12) + JWT; login, registration and role enforcement, backed by PostgreSQL. Reachable only from the gateway.
- **Converter** — Pika + MoviePy + FFmpeg; a pure queue consumer (no HTTP interface) that reads the video queue, extracts audio, and writes the MP3 back to GridFS. Scaled by KEDA on queue depth, scale-to-zero when idle.
- **Notification** — Pika + smtplib; a queue consumer that reads the mp3 queue and emails the download link via SMTP.
- **Outbox Relay** — a single-replica poller that reads the outbox table and publishes to RabbitMQ with publisher confirms. This is the reliability insurance that guarantees an accepted upload is never silently dropped.

Figure 5.1 traces a single upload from the browser to the download link. The boundary between synchronous and asynchronous work is the key design point: only Steps 1 and 2 happen while the user waits — the gateway writes the file and an outbox row in one Mongo write and returns 202 Accepted. Everything from Step 3 onward runs on its own time.

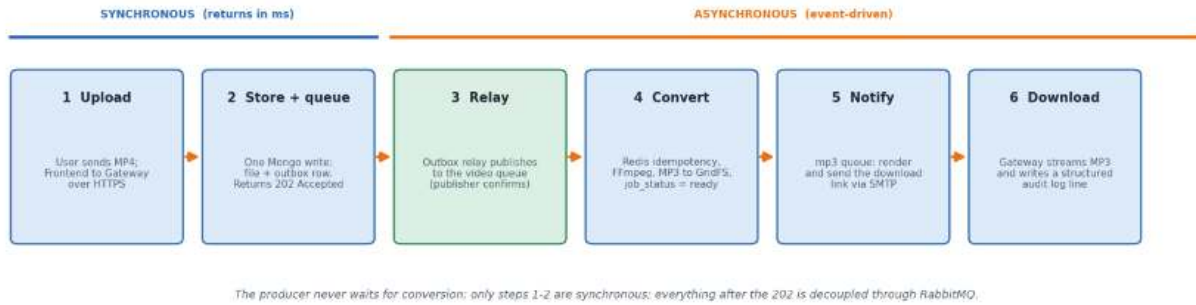


Figure 5.1 — The single-upload journey. Steps 1–2 are synchronous (the 202 returns in milliseconds); Steps 3–6 are asynchronous, event-driven hops.

Because at-least-once messaging can deliver the same message twice, three reliability patterns turn an unreliable queue into a pipeline that never loses an upload and never double-charges compute:

- **Transactional outbox** — the file and an outbox row are written in one Mongo write, and a single relay publishes them with confirms; a crash mid-upload leaves nothing orphaned.
- **Idempotent consumers** — the converter claims each message once with a Redis SETNX key (24-hour TTL), so a redelivery never produces a duplicate MP3.
- **Bounded retry + dead-letter** — each primary queue is paired with a retry and a dead-letter queue; a message is retried five times with backoff and then parked, so a corrupt file is surfaced to operations instead of looping forever.

5.2 Design Principles

Four principles governed the structure of the platform. Each was a deliberate response to a constraint of the client (a two-person operations team, UK-resident client content, a tight budget) rather than a default.

Principle	What it means in VidCast	Why it was chosen
Single EKS cluster	One EKS cluster in eu-west-2; Helm manages the stateful workloads (MongoDB, PostgreSQL, RabbitMQ), Kustomize manages the five app services with dev/prod overlays.	One cluster is operable by two engineers; a second EKS cluster for staging would roughly double the bill for no proportional benefit (staging runs on Docker Swarm instead).
UK data residency	Every resource is provisioned in eu-west-2 (London); client video never leaves the company's own AWS account in the UK.	The law-firm and fintech clients require client content to stay UK-resident with a full audit trail — compliance is a hard requirement, not a preference.
Async by design	The gateway returns 202 Accepted the moment an upload is durably stored; all conversion and notification work flows through RabbitMQ.	Producers must not wait on FFmpeg. Decoupling via a queue also lets the heavy converter scale independently (KEDA) and absorb bursts.
Git is the source of truth	Every AWS resource is Terraform; every Kubernetes object is in Git and reconciled by Argo CD. Image tags are the commit SHA and SHA-pinned in the overlay.	git log answers 'what changed in prod yesterday?' and git blame answers 'who deployed this?'. The cluster is reproducible, not precious — it rebuilds in under 20

minutes.

Table 5.1 — The four design principles and the client constraints they answer.

Two further conventions fall out of these principles and apply to every service. First, each service is self-describing: it owns its container image, a CPU/memory budget, a NetworkPolicy, and — where relevant — its own scaling rule. Second, there are no templating surprises in the app layer: Kustomize overlays are pure YAML that diff cleanly in a pull request, while Helm is reserved for the stateful workloads where lifecycle hooks and templated values genuinely earn their keep.

5.3 End-to-End Architecture Diagram (ALB as the Only Public Door)

Figure 5.2 shows every component of the platform on one diagram. The defining property of the topology is that there is exactly one public door: the internet-facing AWS Application Load Balancer. It is the only resource that accepts traffic from the internet, and it is where TLS terminates.

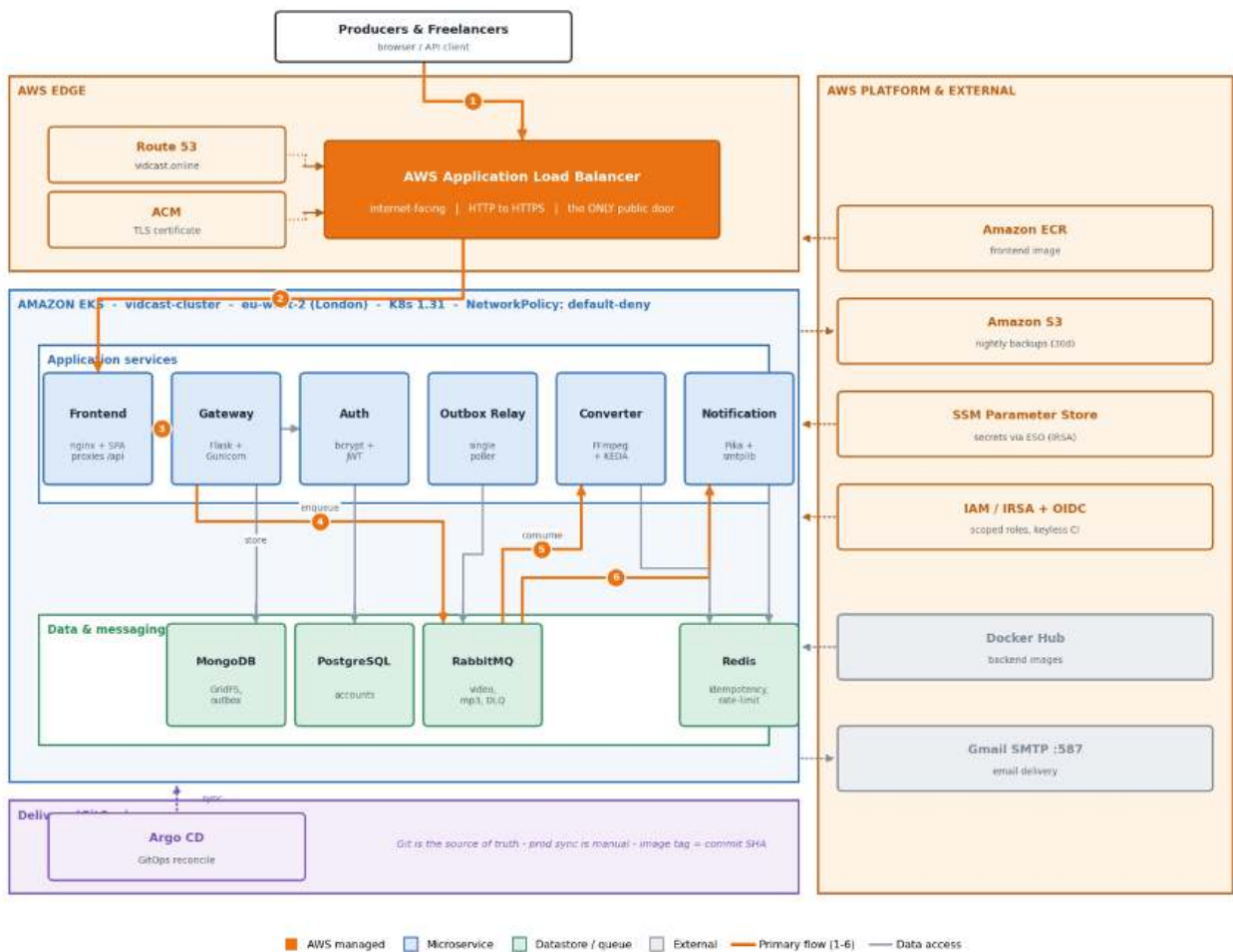


Figure 5.2 — End-to-end architecture. The internet-facing ALB is the only public entry point; every microservice is ClusterIP, reachable only inside the cluster.

The request path from the public internet is:

- A producer resolves vidcast.online through Route 53, which points at the ALB.
- The ALB terminates HTTPS using an ACM certificate and redirects any plain HTTP to 443.
- The ALB (provisioned by the AWS Load Balancer Controller from a single Ingress, target-type ip) forwards to the frontend service on port 8080.
- The frontend's nginx serves the React SPA and proxies /api/ to the gateway, stripping the /api prefix.

Routing all paths through the frontend is a deliberate decision recorded in the Ingress manifest. An ALB cannot rewrite or strip a path prefix, so routing /api straight to the gateway would deliver /api/login to a gateway that only knows /login and return a 404. Going through the frontend's nginx preserves the working request path for both browsers and API clients and keeps the gateway internal (ClusterIP) — a smaller attack surface. Every microservice behind the frontend is ClusterIP; none is published to the internet.

Around the cluster sit the AWS-managed and external services the platform depends on:

- **Amazon ECR** — hosts the frontend image (the backend images currently live on Docker Hub).
- **S3** — receives the nightly mongodump and pg_dump backups under a 30-day lifecycle, written by CronJobs using a write-only IRSA role.
- **SSM Parameter Store** — holds application secrets, pulled at runtime by the External Secrets Operator through an IRSA role (zero secrets in Git).
- **Route 53 + ACM** — DNS for vidcast.online and the TLS certificate the ALB serves.
- **IAM / IRSA + OIDC** — each pod assumes only its own scoped IAM role; CI federates into AWS with short-lived OIDC sessions, no long-lived keys.
- **Argo CD** — continuously reconciles the cluster to match Git; the repository is authoritative, not the cluster.
- **Gmail SMTP** — the external email path used by the notification service (a roadmap item replaces this with Amazon SES).

5.4 Port Map and Network Topology

Table 5.2 lists the port each component listens on and how it is reached. In the hardened production posture the ALB on 443 is the only port open to the internet and administrative UIs are reached with kubectl port-forward. The NodePort range (30002–30008) is provisioned in the security-group module for non-ALB and admin access in lower environments; it is not the production ingress path.

Component	In-cluster port	How it is reached	NodePort (non-prod / admin)
AWS ALB	80 → 443	Public — the only internet-facing port	—
Frontend (nginx)	8080	ALB target in prod (target-type ip)	30006
Gateway (API)	8080	ClusterIP; via the frontend /api proxy	30002
Auth	5000	ClusterIP — from the gateway only	— (never external)
Converter	none	Queue consumer (no HTTP interface)	—
Notification	none	Queue consumer (no HTTP interface)	—
Outbox relay	none	Poller (no HTTP interface)	—
MongoDB + GridFS	27017	ClusterIP	30005 (admin)
PostgreSQL	5432	ClusterIP	30003 (admin)

RabbitMQ	5672 (AMQP) / 15672 (UI)	ClusterIP; mgmt UI via port-forward only	30004
Redis	6379	ClusterIP	—
Grafana / Alertmanager	internal	Port-forward / admin	30007 / 30008

Table 5.2 — Port map. Production exposes only the ALB (443); the NodePort range exists for admin and lower-environment access.

At the network layer the cluster runs zero-trust. The VPC-CNI network-policy agent (native to EKS — no Calico or Cilium) enforces a default-deny baseline on both ingress and egress, so a pod can send and receive nothing until an explicit rule allows it. Every A→B flow therefore needs two rules: an egress rule on A and an ingress rule on B. Table 5.3 enumerates the complete set of allowed paths, and Figure 5.3 draws them; any path not shown is dropped at the kernel.

Source	Destination	Port	Purpose
Internet (via ALB)	Frontend	8080	Serve the SPA
Internet / Frontend	Gateway	8080	API + /api proxy
Gateway	Auth	5000	Validate credentials / JWT
Gateway	MongoDB	27017	GridFS + outbox write
Gateway	RabbitMQ	5672	Publish (direct / outbox path)
Auth	PostgreSQL	5432	User accounts (SCRAM-SHA-256)
Outbox relay	MongoDB	27017	Read outbox rows
Outbox relay	RabbitMQ	5672	Publish with confirms
Converter	RabbitMQ	5672	Consume video / publish mp3 / retry / DLX
Converter	MongoDB	27017	Read video, write MP3
Converter	Redis	6379	Idempotency claim
Notification	RabbitMQ	5672	Consume mp3 queue
Notification	Redis	6379	Idempotency / state
Notification	Gmail SMTP (0.0.0.0/0)	587	Send the download-link email

Table 5.3 — The complete allow-list of pod-to-pod (and egress) flows. Everything else is denied.

Default-deny on ingress AND egress. Every arrow is an explicit allow rule (egress on the source + ingress on the target). Anything not drawn is dropped.

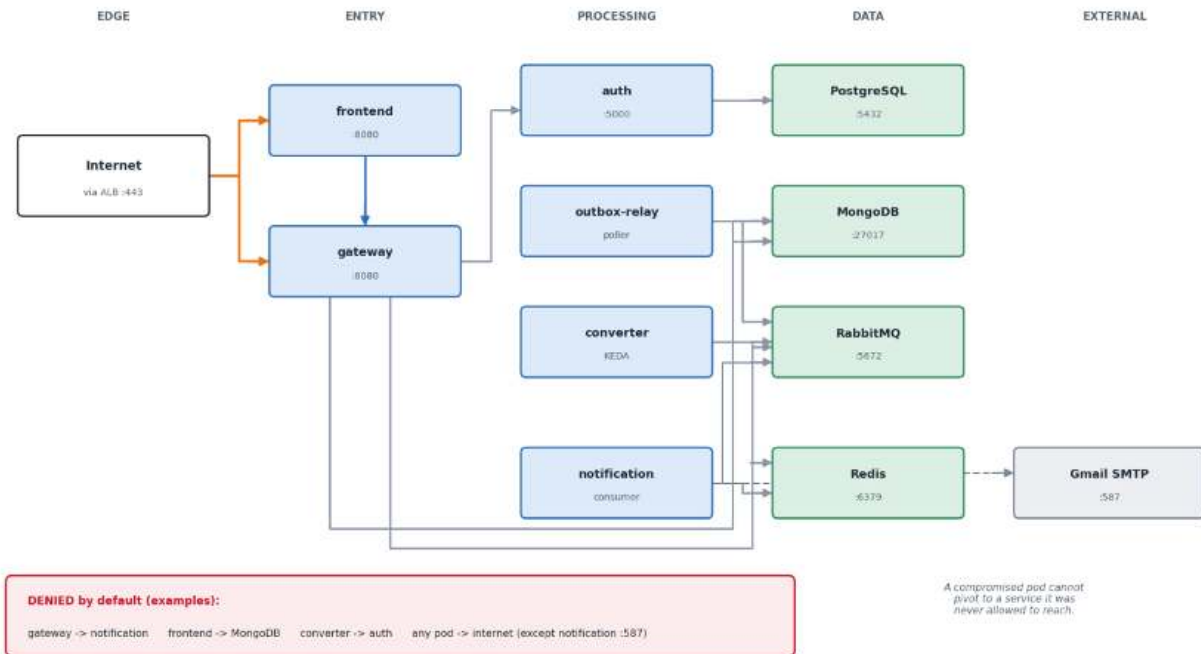


Figure 5.3 — Zero-trust network topology. The gateway has no route to notification (shown blocked); a compromised pod cannot pivot to a service it was never allowed to reach.

The concrete payoff is blast-radius control: the gateway, for example, has no network route to the notification service, so a compromised gateway pod cannot pivot to a service it was never meant to reach. The same model lets the notification service make one tightly-scoped exception — egress to 0.0.0.0/0 on TCP 587 — because Gmail's SMTP IPs are dynamic; nothing inside the cluster listens on 587, so this opens no internal path.

Underneath the pod network sits the VPC. Terraform provisions a 10.0.0.0/16 VPC with two public subnets carved across eu-west-2a and eu-west-2b (one /24 per AZ), an internet gateway, and a public route table. The EKS worker nodes run in those public subnets and the ALB is internet-facing; the subnets are tagged `kubernetes.io/role/elb` so the Load Balancer Controller can place the ALB correctly. The `vpc-cni` addon is installed with `enableNetworkPolicy=true`, which is what makes the default-deny policies above effective at the data-plane level.

5.5 Environments — Production (EKS), Staging (Docker Swarm), Local

VidCast runs in three environments. They share the same container images and the same application code but differ deliberately in platform and cost — the most expensive infrastructure exists only where production traffic justifies it.

	Production	Staging	Local
Platform	AWS EKS (K8s 1.31)	Docker Swarm on EC2	Docker Compose
Region / host	eu-west-2 (London)	Single EC2 instance	Developer laptop
Compute	2 × m7i-flex.large nodes	One small EC2 host	Local Docker engine

Ingress	Internet-facing ALB + ACM TLS	Published Swarm ports	localhost port maps
Delivery	Argo CD GitOps (prod = manual sync)	Jenkins + manual approval	docker compose up
Secrets	External Secrets Operator + SSM	Swarm env / secrets	Local env values
Purpose	Live client traffic	Pre-production integration testing	Developer testing
Approx. cost	~£198 / month	~£10 / month	Free

Table 5.4 — The three environments. Same images and code; platform and cost scale to purpose.

Production is the EKS cluster described throughout this chapter: vidcast-cluster in eu-west-2, two m7i-flex.large nodes (a burstable M-series type chosen because the account's service-control policy blocks the cheaper T-series), an internet-facing ALB, Argo CD reconciling Git, and an all-in running cost of roughly £198/month — about 94% under the client's £3,500 cap. Because everything is in Git or S3, the whole environment can be torn down with terraform destroy to ~£0 and rebuilt in under 20 minutes.

Staging deliberately avoids a second EKS cluster. It is a Docker Swarm stack defined in docker-compose.swarm.yml, deployed by Jenkins with a manual approval gate, on a single small EC2 instance for roughly £10/month — about a 97% saving over a duplicate EKS cluster for equivalent integration-test functionality. The stack mirrors production's shape: the gateway runs two replicas and the converter four, all services share a vidcast-net overlay network, MongoDB and PostgreSQL use named volumes, and each service carries a rollback-on-failure update policy. Swarm's built-in mTLS on the overlay network gives staging service-to-service encryption without a service mesh.

Local is plain Docker Compose on a developer's machine: the full stack on localhost, no cloud cost, for fast iteration before a change is pushed and promoted through staging to production. The progression — Compose, then Swarm, then EKS — keeps the developer feedback loop free and fast while reserving managed AWS infrastructure for the environment that actually serves clients.

Chapter 6 — The Five Microservices

VidCast is decomposed into five application microservices — Frontend, Gateway, Auth, Converter and Notification — plus a single-replica Outbox Relay that is the platform's reliability insurance. Each service has one job, its own container image and deployment lifecycle, and a defined contract with its neighbours; none reaches into another's database or shares its process. This chapter describes each service in turn (6.1–6.6) and then steps back to the boundaries, separation of concerns and inter-service contracts that hold them together (6.7). Table 6.1 is the inventory the rest of the chapter expands on.

Service	Stack	Port / exposure	Replicas	Single responsibility
Frontend	React 18 + Vite + Tailwind / nginx	8080 (NodePort 30006); ALB target in prod	1	Web UI: login, upload, history, downloads
Gateway	Flask + Gunicorn (2 workers)	8080 ClusterIP (via /api)	2	API: upload/download, JWT validation, rate-limit, metrics
Auth	Flask + bcrypt + PyJWT +	5000 ClusterIP (internal)	2	Login, registration, role enforcement,

	psycopg2	only)		JWT issue/validate
Converter	Pika + MoviePy + FFmpeg	none (metrics :9000)	2 (KEDA 0→2)	Consume video queue; MP4 → MP3 into GridFS
Notification	Pika + smtplib	none (metrics :9000)	2	Consume mp3 queue; email (individual or batch summary)
Outbox Relay	Python poller (Pika + PyMongo)	none	1 (single by design)	Publish outbox rows → RabbitMQ with confirms

Table 6.1 — Service inventory: stack, exposure, replica count and single responsibility.

All services run with the same hardened container context — `runAsNonRoot` (uid 1000, or 1001 for the frontend's `nginx`), all Linux capabilities dropped, no privilege escalation, and `seccomp RuntimeDefault`. Every service except the frontend also runs with a read-only root filesystem (the frontend's `nginx` needs a writable filesystem, a documented exception), with explicit writable `/tmp` volumes where a process genuinely needs scratch space.

6.1 Frontend — React 18 + Vite + Tailwind + nginx (SPA, Upload, History, Downloads)

The frontend is the only thing producers see: a single-page application built with React 18, `react-router-dom` and `axios`, bundled by Vite and styled with Tailwind, then served by `nginx` from a multi-stage image (Node build stage → `nginx` runtime). It provides login and registration, multi-file upload, a download/history view, a "your file is ready" notification badge, and — for admins — a user-management screen. The SPA talks only to a relative `/api` base (`axios`), and `nginx` both serves the static bundle and reverse-proxies `/api/` to the gateway, stripping the prefix; an ALB cannot rewrite a path, so routing everything through `nginx` keeps the request path working for browsers and API clients alike while keeping the gateway internal. The container runs as non-root uid 1001 with capabilities dropped and sets the `X-Frame-Options`, `X-Content-Type-Options` and `X-XSS-Protection` headers.

6.2 Gateway — Flask + Gunicorn (API, Upload/Download, JWT Validation, Rate Limiting, Metrics)

The gateway is the single application entry point and the busiest service (two replicas, two Gunicorn workers each). It is a Flask app run under Gunicorn for production concurrency, and it owns five concerns at once: it validates JWTs (delegating to Auth), stores and streams media via GridFS, publishes work to RabbitMQ through the outbox, enforces rate limits backed by Redis, and exposes Prometheus metrics. The upload path returns 202 Accepted in milliseconds — it writes the file and an outbox row, never blocking on conversion. Table 6.2 lists the HTTP API it and Auth expose.

Service	Route	Purpose / contract
Gateway	POST <code>/login</code>	Delegates to Auth; returns a JWT (rate-limited 10/min)
Gateway	POST <code>/register</code>	Creates a user-role account; returns a JWT
Gateway	POST <code>/upload</code>	JWT-gated; GridFS + outbox write; 202 Accepted (20/hr, batch-weighted)
Gateway	GET <code>/download?fid=</code>	JWT-gated; streams the MP3 from GridFS; writes a download-audit log
Gateway	GET <code>/notifications/unseen-count</code>	JWT-gated; powers the "file ready" badge (5s polling)
Gateway	GET <code>/admin/users</code> · PATCH	Admin-only; list users / change role (403 self, 409 last-admin)

	/admin/users/<email>	guards)
Gateway	GET /healthz · GET /metrics	Dependency health (200/503) · Prometheus exposition
Auth	POST /login · POST /register · POST /validate	Issues / validates JWTs against PostgreSQL (internal)
Auth	GET /users · PATCH /users/<email>	Internal user admin — no role check; trusts in-cluster callers

Table 6.2 — The synchronous HTTP API: the gateway's public-facing routes and Auth's internal routes.

6.3 Auth — Flask + bcrypt + PyJWT + pycopp2 (Login, Registration, Role Enforcement)

Auth is the identity service, reachable only inside the cluster on ClusterIP 5000 — never exposed to the internet. It verifies credentials against the PostgreSQL `auth_user` table with `bcrypt.checkpw` (constant-time, against cost-12 hashes), issues HS256 JWTs carrying the user's real role from the database (`admin = role == 'admin'`, plus a forward-compatible role string), and validates tokens on demand. Registration always creates a role 'user' account, so self-registration can never mint an administrator. A deliberate, documented trust boundary applies: Auth's internal `/users` endpoints carry no role check of their own and trust in-cluster callers, with the gateway enforcing "admin only" — the privilege boundary therefore sits at the network layer (ClusterIP + default-deny), with mTLS or a mesh recorded as the fix at scale.

6.4 Converter — Pika + MoviePy + FFmpeg (Queue Consumer, MP4 → MP3)

The converter is a pure queue consumer with no HTTP interface — it only exposes a metrics endpoint on :9000 for a PodMonitor. It consumes the video queue, and for each message it claims the job once in Redis (idempotency), fetches the MP4 from GridFS by `video_fid`, extracts the audio with `MoviePy/FFmpeg` into a temporary file, writes the resulting MP3 back to GridFS (carrying `owner_email` in the metadata so downloads can be scoped), publishes the enriched message to the `mp3` queue, and acknowledges. It is the heaviest workload on the cluster, so it is the one service scaled by KEDA on the video queue's depth — from zero when idle up to the node-budget cap of two replicas — and it mounts an `emptyDir` at `/tmp` for the conversion scratch files while keeping the rest of its root filesystem read-only.

6.5 Notification — Pika + smtpplib (Email Delivery, Batch Summary Emails)

The notification service is the other end of the pipeline: a queue consumer (two replicas, metrics on :9000) that reads the `mp3` queue and emails the producer a download link via SMTP (Gmail in the current deployment; Amazon SES on the roadmap). Its non-obvious feature is batch awareness. A single upload triggers one email, but a multi-file batch would otherwise produce a flood of them; instead the service uses the `job_status` collection to detect when every file in a `batch_id` is terminal and then sends one summary email, claiming a `batchsummary` marker so that exactly one summary is sent even when two workers race. If MongoDB is unavailable it falls back to per-file emails rather than dropping the notification — degrade, don't fail.

6.6 Outbox Relay — Single-Replica Publisher (MongoDB → RabbitMQ)

The outbox relay is the sixth component and the one that makes the pipeline reliable. It is a separate, deliberately single-replica Deployment — not a thread inside the gateway — because the gateway runs multiple Gunicorn workers, and an in-process relay would run once per worker and re-introduce the duplicate-publish bug the outbox exists to kill. One replica means one publisher by construction. On a

fixed interval it scans MongoDB for outbox rows with no published_at, publishes each to RabbitMQ as a persistent message with publisher confirms (oldest first), and stamps it published. A relay restart mid-publish simply re-picks the unstamped rows next cycle, and the idempotent converter makes any rare duplicate a no-op. Its liveness probe touches /tmp/healthy every loop, so a transient broker or Mongo outage is retried through rather than killing the pod.

6.7 Service Boundaries, Separation of Concerns and Inter-Service Contracts

What turns six processes into a system is the discipline of the boundaries between them. Each service owns exactly one concern and reaches its neighbours only across a defined contract — never by sharing a database table or a process. Figure 6.1 shows those boundaries and the transport on each: synchronous HTTP/REST for request/response, AMQP messages for the asynchronous pipeline, and SMTP at the edge.

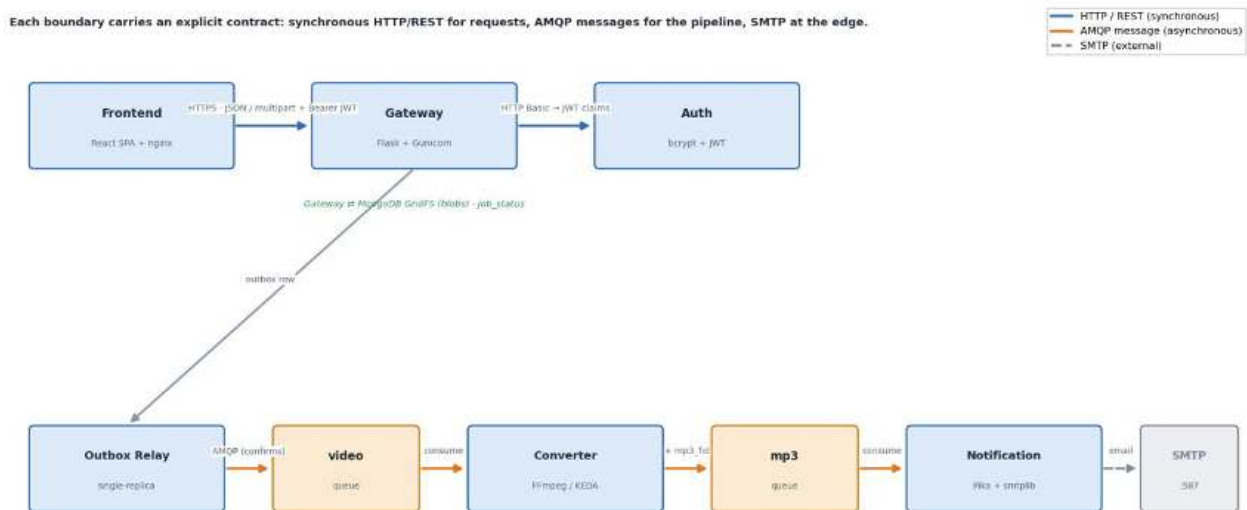


Figure 6.1 — Service boundaries and the contract on each: HTTP/REST for requests, AMQP messages for the pipeline, SMTP at the edge; GridFS is the blob medium for the gateway.

The contracts fall into three kinds — synchronous HTTP, asynchronous AMQP messages, and datastore access — each with a defined shape, summarised in Table 6.3.

From	To	Transport	Contract
Frontend	Gateway	HTTPS (ALB → nginx /api)	JSON / multipart form; Bearer JWT
Gateway	Auth	HTTP (ClusterIP :5000)	HTTP Basic (login) / Bearer (validate) → JWT claims
Gateway	MongoDB	TCP :27017	GridFS put/get; outbox + job_status documents
Gateway / Outbox Relay	RabbitMQ	AMQP :5672	video-queue message {video_fid, username, correlation_id, batch_id?, batch_size}
Converter	RabbitMQ	AMQP :5672	mp3-queue message (the same envelope + mp3_fid)
Converter	Redis	TCP :6379	Idempotency claim (SET NX on video_fid)
Notification	SMTP	TCP :587	Email — individual or one batch summary
Auth	PostgreSQL	TCP :5432	SQL against auth_user (SCRAM-SHA-256)

Table 6.3 — The inter-service contracts: who talks to whom, over what transport, with what payload.

The asynchronous side deserves a closer look because it is where separation of concerns pays off. There is exactly one message contract, and it is enriched at each hop rather than reshaped (Figure 6.2): the gateway/relay publishes {video_fid, username, correlation_id, and batch fields}; the converter adds mp3_fid and republishes the same envelope to the mp3 queue; the notification service reads it to build the download link and decide on batching. Because the contract is stable and idempotency is keyed on video_fid, any service can be scaled, restarted or redeployed independently without coordinating with the others.

The one message contract that flows the pipeline — enriched at each hop, never reshaped.



Idempotency keyed on video_fid (Redis SETNX) so a redelivery never adds a second mp3.

Figure 6.2 — The single async message contract, enriched at each hop (the converter adds mp3_fid) and never reshaped; idempotency is keyed on video_fid.

This is the heart of the design's value: the boundaries are narrow and explicit, so the system is loosely coupled in fact and not just on a diagram. The gateway knows nothing of FFmpeg; the converter knows nothing of SMTP; Auth is unreachable from the internet; and the outbox relay is the only publisher of record. Each service can fail, scale or be replaced behind its contract — which is precisely what lets a two-person team operate the whole platform with confidence.

Chapter 7 — Data Layer and Messaging

VidCast keeps four datastores and one object store, each chosen for a single job rather than as a general-purpose database. MongoDB with GridFS holds the binary media and the event/state collections; PostgreSQL holds the user accounts; RabbitMQ is the event bus that decouples upload from conversion; Redis provides the fast, disposable state — idempotency claims and rate-limit counters; and Amazon S3 is the off-cluster backup target that makes the whole stateful tier recoverable. All four datastores are self-hosted in the cluster as Helm releases rather than bought as AWS-managed services — a deliberate, cost-driven decision examined in 7.6. This chapter documents each store as it is actually configured in the repository (Helm charts, service code, and Terraform), not as a generic reference design.

Persistence & durability: each datastore's backing volume, plus the nightly dump to S3 that is the durability backstop for the stateful tier.

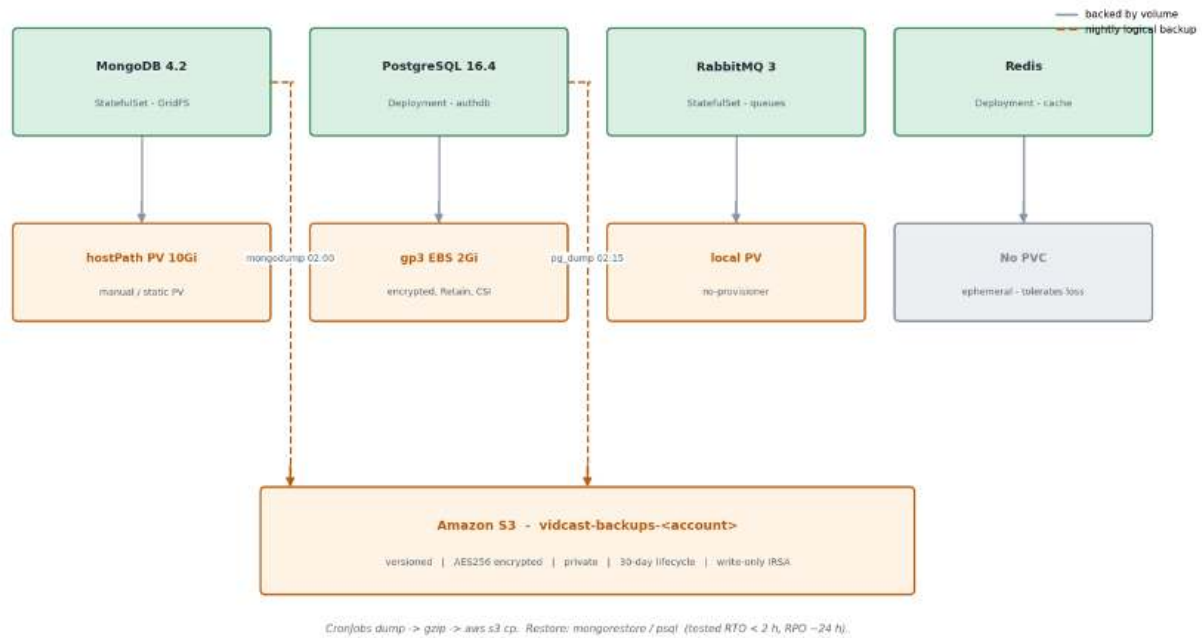


Figure 7.1 — Persistence and durability of the stateful tier: each datastore's backing volume, and the nightly logical dump to S3 that is the durability backstop.

Store	Image	Workload	In-cluster port	Backing storage	Backup
MongoDB + GridFS	mongo:4.2	StatefulSet (1)	27017	hostPath PV 10Gi (static)	Nightly mongodump → S3
PostgreSQL	postgres:16.4-alpine	Deployment (1)	5432	gp3 EBS 2Gi (encrypted, Retain)	Nightly pg_dump → S3
RabbitMQ	rabbitmq:3-management	StatefulSet (1)	5672 / 15672 / 15692	local PV (no-provisioner)	Durable queues; no dump
Redis	redis	Deployment (1)	6379	None — ephemeral	None (tolerates loss)
Amazon S3	—	AWS managed	—	vidcast-backups-<account>	Is the backup target

Table 7.1 — Datastore inventory as configured in the Helm charts and Terraform.

7.1 MongoDB + GridFS — Video/MP3 Blobs, Outbox and job_status Collections

MongoDB is the heaviest-loaded store because it holds the media itself. It runs as a single-replica StatefulSet on mongo:4.2 — a version pinned deliberately: mongo:4.0.8 (wire protocol v7) was rejected at runtime because the services' CVE-bumped PyMongo requires at least MongoDB 4.2 (wire v8), which had been breaking the gateway and converter. Binary files are stored through GridFS, which transparently chunks each file into 255 KB documents so objects larger than MongoDB's 16 MB document limit (i.e. every video) can be stored and streamed.

The data is split across two databases and two operational collections:

Database / collection	Holds	Written by	Read by
videos (GridFS fs_videos)	Raw uploaded MP4 blobs, chunked	Gateway (fs.put)	Converter
mp3s (GridFS fs_mp3s)	Converted MP3 blobs, chunked	Converter	Gateway (download)
videos.outbox	One event row per accepted upload (transactional outbox)	Gateway	Outbox relay
videos.job_status	Per-job state: queued → ready, owner_email, correlation_id, batch_id	Gateway (queued), Converter (ready)	Frontend / API

Table 7.2 — The two GridFS databases and the two operational collections, all in MongoDB.

Co-locating the outbox and job_status collections in the same videos database as the GridFS file is what makes the reliability story work. On upload the gateway performs the GridFS put, inserts a job_status row marked "queued", and (when the outbox is enabled) inserts the event into videos.outbox instead of publishing to RabbitMQ directly. Because mongod runs standalone, there is no multi-document transaction available, so the GridFS write and the outbox insert are not strictly atomic; the code bounds that window by ordering the writes and deleting the orphaned status/file if the outbox insert then fails. The converter later advances the same job_status row to "ready" once the MP3 is written.

On storage, MongoDB is backed by a static PersistentVolume rather than dynamic EBS: the chart declares a 1 Gi mongodb-pvc bound to a 10 Gi hostPath PersistentVolume at /mnt/data via the manual StorageClass (no-provisioner). Data therefore survives a pod restart on the same node, and the durability backstop for a node loss is the nightly mongodump to S3 (7.5), not block-level replication. (Note: the presentation's reference to "gp3 EBS" for the media store describes the production-intended backing; the committed chart uses the hostPath PV above, while PostgreSQL is the store actually wired to gp3 EBS — see 7.2.) The pod runs with seccomp RuntimeDefault and no privilege escalation; running fully non-root is a documented Kyverno Audit exception because the official mongo image's entrypoint must start as root to chown /data/db before initdb.

7.2 PostgreSQL — User Accounts, bcrypt Hashes, Roles (SCRAM-SHA-256, ACID, EBS PVC)

PostgreSQL is the system of record for identity, and it is the one store held to a true durability standard because losing a user account is not acceptable. It runs as a single-replica Deployment on postgres:16.4-alpine (pinned off :latest for a small, low-CVE base) using the Recreate update strategy — correct for a single ReadWriteOnce EBS volume, since the old pod must release the volume before the new one attaches. The database is authdb and the schema is a single table:

Column	Type	Notes
id	integer, GENERATED ALWAYS AS IDENTITY	Primary key
email	VARCHAR(255) NOT NULL UNIQUE	Login identity
password	VARCHAR(255) NOT NULL	bcrypt hash — never plaintext
role	VARCHAR(32) NOT NULL DEFAULT 'user'	RBAC: 'admin' or 'user'
created_at	TIMESTAMP DEFAULT CURRENT_TIMESTAMP	Audit

Table 7.3 — The `auth_user` table (*Helm_charts/Postgres/init.sql*).

Three properties matter here. First, authentication: the container sets `POSTGRES_HOST_AUTH_METHOD=scram-sha-256`, replacing the original 'trust' method that accepted any password — credentials are now genuinely enforced (SCRAM-SHA-256 challenge–response), with `deploy.sh` guaranteeing a non-empty password. Second, the password column stores a `bcrypt (cost-12)` hash verified with `constant-time bcrypt.checkpw`; `init.sql` contains no admin row and no hash, so nothing secret lives in the public repo — the admin is seeded at deploy time via `pgcrypto's crypt()/gen_salt('bf')`, whose `$2a$` output is compatible with the auth service. Third, ACID: register, password-change and role-demote operations are single-row, atomic, durable writes against this table.

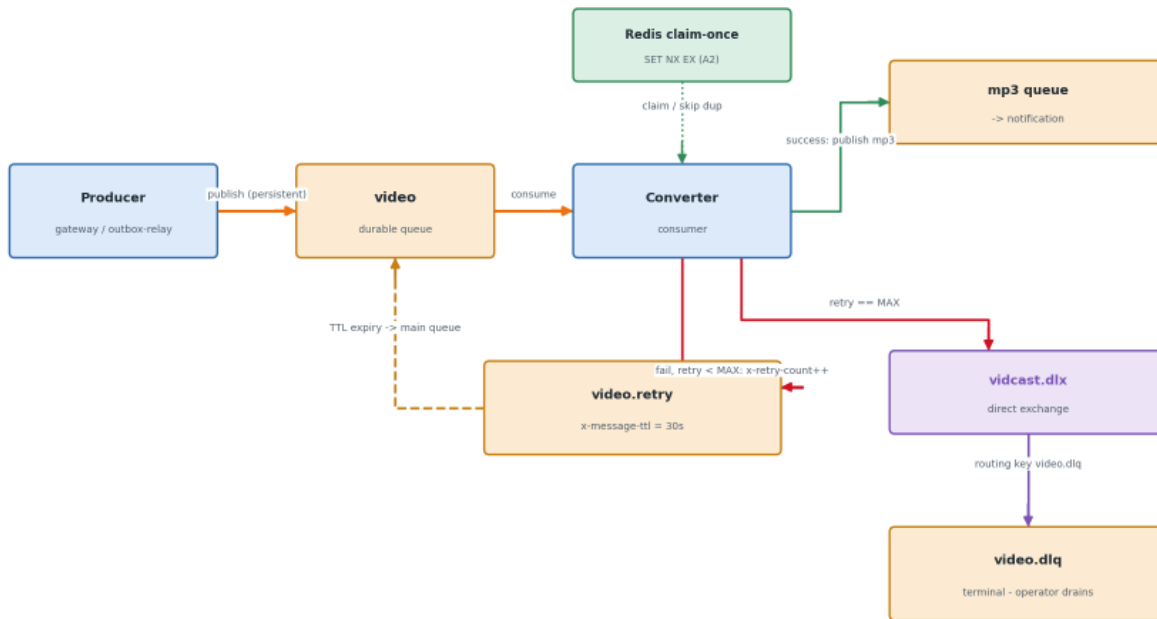
Durability is provided by a dynamic `gp3` EBS volume (the A11 fix). The chart creates a `StorageClass` `vidcast-ebs-gp3` backed by the `aws-ebs-csi-driver` addon with type `gp3`, `encrypted="true"`, `volumeBindingMode WaitForFirstConsumer` (pins the volume to the AZ the pod lands in), `allowVolumeExpansion`, and — critically — `reclaimPolicy: Retain`, so deleting the PVC or a stray helm uninstall cannot take the user accounts with it. A 2 Gi `ReadWriteOnce` `postgres-pvc` binds to it, and `PGDATA` is pointed at a `/pgdata` subdirectory because `initdb` refuses to initialise in a volume root that already contains `lost+found`. Without this PVC, every registered user except the seed admin would vanish on the first pod restart.

7.3 RabbitMQ — video and mp3 Queues, Retry Queues, Dead-Letter Topology

RabbitMQ is the asynchronous spine of the platform: the gateway/outbox-relay publish work to it and the converter and notification services consume from it, so an upload returns in milliseconds while conversion happens later. It runs as a single-replica `StatefulSet` on `rabbitmq:3-management`. Unlike the database pods, RabbitMQ fully satisfies the non-root baseline — it runs as `uid/gid 999` with an `fsGroup` making its data directory writable, drops ALL capabilities, and applies `seccomp RuntimeDefault`. It exposes AMQP on 5672, the management UI on 15672 (reachable by port-forward only — never a `NodePort`), and a Prometheus metrics endpoint on 15692 scraped by a `ServiceMonitor`.

Two durable primary queues carry the pipeline — video (gateway → converter) and mp3 (converter → notification) — but the value is in what wraps them. Because delivery is at-least-once (a relay double-publish, a broker redelivery after a crash between publish and ack, or a retry), each primary queue is paired with a delayed-retry queue and a terminal dead-letter queue, declared from code in `rabbitmq_retry.py`. Figure 7.2 shows the topology for the video pipeline; the mp3 pipeline is identical.

At-least-once delivery made safe: idempotent claim-once, bounded delayed retry, terminal dead-letter. Shown for the video pipeline; the mp3 pipeline is identical.



Retry count tracked in an explicit x-retry-count header (not the broker x-death). The whole topology is declared idempotently from code on consumer start - reproducible, no manual UI.

Figure 7.2 — RabbitMQ retry and dead-letter topology for the video pipeline, with the Redis claim-once guard. The mp3 pipeline mirrors it exactly.

Queue / exchange	Type	Role
video, mp3	Durable primary queues	Carry work; consumed by converter / notification
video.retry, mp3.retry	Durable, x-message-ttl = 30 s	Hold a failed message for the TTL, then dead-letter (default exchange) back to the main queue
vidcast.dlx	Durable direct exchange	Routes terminally-failed messages to the DLQs
video.dlq, mp3.dlq	Durable terminal queues	Park messages after MAX_RETRIES; an operator drains/inspects them

Table 7.4 — RabbitMQ queue and exchange topology (declared idempotently on consumer start).

On a transient failure the consumer increments an explicit x-retry-count header and republishes the message to <queue>.retry; after the TTL expires with no consumer, RabbitMQ dead-letters it via the default exchange straight back onto the main queue for another attempt. Once x-retry-count reaches MAX_RETRIES (default 3, configurable; the deck cites the tuned value of 5) the message is published to vidcast.dlx with routing key <queue>.dlq and parked terminally. Two design choices make this robust: the retry count is tracked in an explicit header rather than the broker's version-dependent x-death, and after re-publishing the consumer acks the original delivery — which is precisely what breaks the old infinite NACK-requeue poison loop. The entire topology is declared idempotently every time a consumer starts, so it is reproducible from scratch with no manual queue creation in the management UI.

7.4 Redis — Idempotency Keys and Rate-Limit Counters

Redis holds the platform's fast, disposable state — data that is cheap to rebuild and explicitly tolerant of loss. It serves two purposes, both of which depend on a shared store that all pods and all gunicorn workers can see.

Use	Mechanism	Key / scope	Failure behaviour
Idempotency (claim-once)	SET key NX EX(ttl) — atomic set-if-absent	Per-service key on the job's video_fid (TTL default 300 s)	Fails OPEN: on a Redis error, process anyway
Rate limiting	flask-limiter, fixed-window, storage in Redis	/login 10/min (brute-force); /upload 20/hour, cost = files in batch	Shared across gunicorn workers via Redis

Table 7.5 — Redis usage: idempotency claims (consumers) and rate-limit counters (gateway).

The idempotency guard is what makes at-least-once delivery safe to combine with the retry topology in 7.3. Before processing, a consumer issues SET key NX EX(ttl): because set-if-absent is atomic, concurrent deliveries race safely and exactly one wins the claim — a redelivered or duplicated message is skipped rather than producing a second MP3 and a second email. The TTL bounds the dedup window so a crash can never wedge a claim forever, and the claim is keyed per-service so the converter and notification pipelines don't collide. Critically, the guard fails open: if Redis is unreachable, claim_once returns true and the message is processed anyway — a Redis outage degrades the system to "occasional duplicate" (which the consumers absorb) rather than "stuck pipeline". A claim is released only on a retryable failure (so the next attempt can re-claim), never on success and never on a terminal DLQ failure.

The gateway's rate limiting reuses the same Redis. flask-limiter stores its fixed-window counters there so that the limits are enforced across all gunicorn worker processes rather than per-process: /login is capped at 10 per minute as brute-force protection, and /upload at 20 per hour with the cost weighted by the number of files in a batch upload. Redis has no PersistentVolume — both uses tolerate loss by design, which is why it can run as a lightweight, restartable pod.

7.5 Amazon S3 — Versioned, Encrypted Backup Bucket

S3 is the one managed AWS service in the data layer, and it exists precisely because the in-cluster stores are not block-replicated: it is the durability backstop. While the application layer is recoverable from Git via Argo CD, the databases needed an off-cluster copy, and a single private bucket — vidcast-backups-`<account-id>`, named deterministically so the CronJobs can hardcode it — provides it. The Terraform storage module hardens the bucket:

- **Versioning enabled** — keeps a short history so a bad nightly dump does not immediately overwrite the last good one.
- **Public access fully blocked** — all four block-public settings are true; the dumps contain GridFS files and auth rows and must never be public.
- **Server-side encryption (AES256)** — the free AWS-managed S3 key, consistent with the project's no-CMK cost decision.
- **Lifecycle policy** — current and non-current objects expire after 30 days, and abandoned multipart uploads abort after 7, so the bucket cannot grow unbounded.

- **Least-privilege IRSA** — the backup CronJobs assume a vidcast-backup role scoped to s3:PutObject and s3:ListBucket on this bucket only — they can write dumps and nothing else.

Two Kubernetes CronJobs feed the bucket. mongo-backup runs nightly at 02:00 UTC and uses mongodump --gzip --archive to dump the videos and mp3s databases (every GridFS file plus the outbox) into separate archives; postgres-backup runs at 02:15, staggered 15 minutes later to avoid contending for node CPU and network, and pipes pg_dump of authdb through gzip. Each job dumps in an init container, then an amazon/aws-cli container ships the artefact to S3; both run non-root (uid 1000) with concurrencyPolicy: Forbid so a slow dump never overlaps the next night. The recovery path is mongorestore / psql from these dumps, giving a tested RTO under two hours and an RPO of roughly 24 hours (documented in the disaster-recovery runbook).

7.6 Self-Hosted vs Managed Datastores — Cost-Driven Trade-off Analysis

An explicit option was to replace all four in-cluster datastores with AWS-managed equivalents — RDS for PostgreSQL, MongoDB Atlas for MongoDB/GridFS, Amazon MQ for RabbitMQ, and ElastiCache for Redis — and a sprint was originally scoped to cut over to them in production. After honest costing, that cutover was cancelled and recorded as a documented-and-deferred migration path rather than built. The reason is in Figure 7.3: the managed alternative carries a large standing bill while the in-cluster charts cost essentially nothing when the cluster is off.

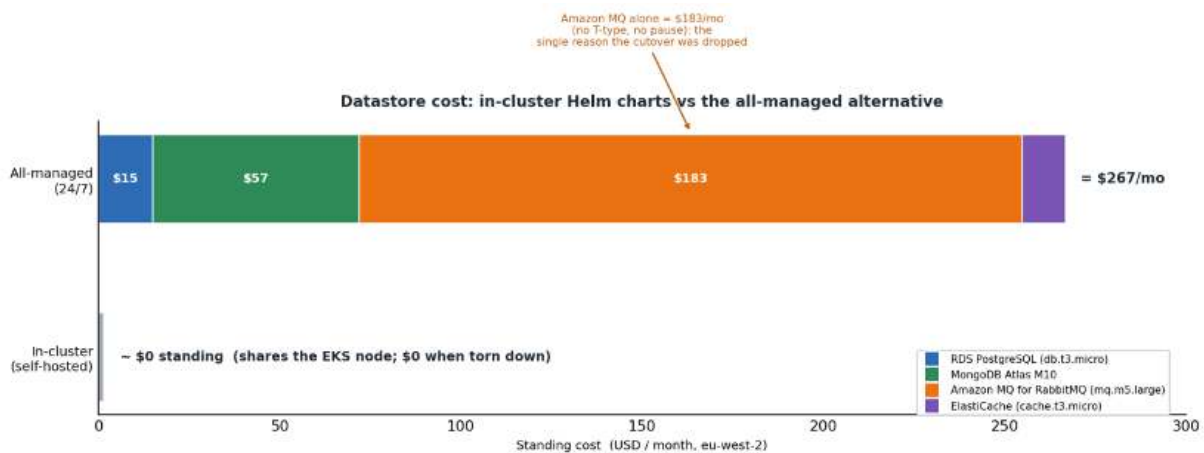


Figure 7.3 — Standing monthly cost (eu-west-2, 24/7): the all-managed datastore stack ≈ \$267/mo versus ≈ \$0 standing for the in-cluster Helm charts.

Store	Today (kept)	Managed candidate	Standing cost (24/7)	Adopt when
PostgreSQL	postgres Deployment + gp3 EBS PVC	RDS PostgreSQL db.t3.micro	~\$15 Single-AZ / ~\$31 Multi-AZ	PITR / automated backups needed beyond the nightly dump
MongoDB + GridFS	mongo:4.2 StatefulSet	MongoDB Atlas (M0 free / M10)	\$0 (M0) / ~\$57 (M10, ~\$1–2 paused)	Off-cluster blob durability + managed backups matter
RabbitMQ	rabbitmq:3 StatefulSet	Amazon MQ for RabbitMQ mq.m5.large	~\$183 (no T-type, no pause)	The broker must outlive the node (i.e. multi-node)

Redis	in-cluster Redis pod	ElastiCache cache.t3.micro	~\$12	The lock store must be HA / managed
All-managed, left running	—	RDS + Atlas M10 + Amazon MQ + ElastiCache	~\$262–273 / month	—

Table 7.6 — Self-hosted vs managed datastore trade-off (from the A5 decision record, docs/MANAGED_SERVICES.md).

Three findings drove the decision. The dominant one is Amazon MQ for RabbitMQ: the smallest broker instance that exists for it is mq.m5.large at roughly \$183/month — there is no T-type and no pause — making the managed broker, on its own, larger than the EKS control plane and about three times the rest of the managed stack combined. (Amazon MSK / Kafka was rejected outright: it would mean rewriting every Pika producer and consumer, which is scope creep, not reliability work.) Second, for MongoDB the chosen managed path is Atlas, not Amazon DocumentDB: DocumentDB only emulates the Mongo API and has historically had gaps around the exact fs.chunks/GridFS operations VidCast depends on, and it is also the priciest minimum — Atlas is genuine MongoDB, so GridFS works unchanged, and its M0 tier is free for dev. Third, the all-managed stack left running costs roughly \$262–273/month — a 15–40× jump over the ~\$10/month the cluster was torn down to save, on a project whose explicit goal is near-zero cost when idle.

The decisive argument, though, is not price alone but that managed datastores buy little here. Teams usually adopt them for the reliability guarantees — no lost events, idempotent retries, dead-lettering, durable accounts — and VidCast already delivers those in code against the in-cluster brokers: the transactional outbox (7.1), the idempotent consumers (7.4), the bounded retry/DLQ topology (7.3), and the gp3-EBS-backed PostgreSQL plus nightly S3 dumps (7.2, 7.5). On a single-node cluster the node is the availability boundary for everything anyway, so paying \$183/month for a broker that survives a node loss is moot. The conclusion is to keep all four datastores in-cluster for now, with each managed service documented as a specific, costed migration trigger for when real production traffic justifies it.

Chapter 8 — Reliability and Resilience Patterns

An asynchronous, queue-driven platform earns its keep only if work is never silently lost, never processed twice, and never able to wedge the pipeline. VidCast meets that bar not by paying for managed infrastructure (see §7.6) but with **four reliability patterns implemented in application code** against the in-cluster brokers, backed by health probes and graceful degradation. This chapter documents each pattern as it actually exists in the repository — the transactional outbox (A1), idempotent consumers (A2), bounded retry with dead-letter queues (A3), and the health/heartbeat model — and the feature flags that govern them.

The patterns compose into a single guarantee: **at-least-once delivery made effectively-once**. A1 guarantees the event is never dropped between the upload and the broker; A3 guarantees a transient failure is retried and a permanent failure is parked rather than looped; and because both can cause a message to be delivered more than once, A2 guarantees the duplicate is a no-op. The table below maps each pattern to the failure it defends against and the code that implements it.

Pattern	Failure it prevents	Where it lives in the repo	Flag
---------	---------------------	----------------------------	------

A1 Transactional Outbox	Upload accepted but event lost on a broker outage	<code>gateway/storage/util.py</code> , <code>outbox-relay/relay.py</code>	OUTBOX_ENABLED
A2 Idempotent Consumers	Duplicate conversion / duplicate email on redelivery	<code>*/idempotency.py</code> (converter + notification)	IDEMPOTENCY_ENABLED
A3 Retry + DLQ	Transient failure lost; poison message queued forever	<code>*/rabbitmq_retry.py</code>	MAX_RETRIES, RETRY_TTL_MS
Health & degradation	Healthy pod killed; outage cascades into crash-loop	<code>/healthz</code> + <code>/tmp/healthy</code> probes	—

Table 8.1 — The four reliability patterns, the failure each prevents, and their implementation.

8.1 Single Upload Journey — Every Hop from Browser to Download Link

Before examining each pattern in isolation, it helps to follow one file through the whole system. The gateway is **asynchronous by contract**: it returns `202 Accepted` the moment the upload is durably stored and enqueued, and the heavy work happens behind the queues. Each uploaded file is an independent job with its own `correlation_id` (a fresh `uuid4` minted per file), so the trace can be followed across every hop.

Figure 8.1 — Single Upload Journey: Browser → Download Link

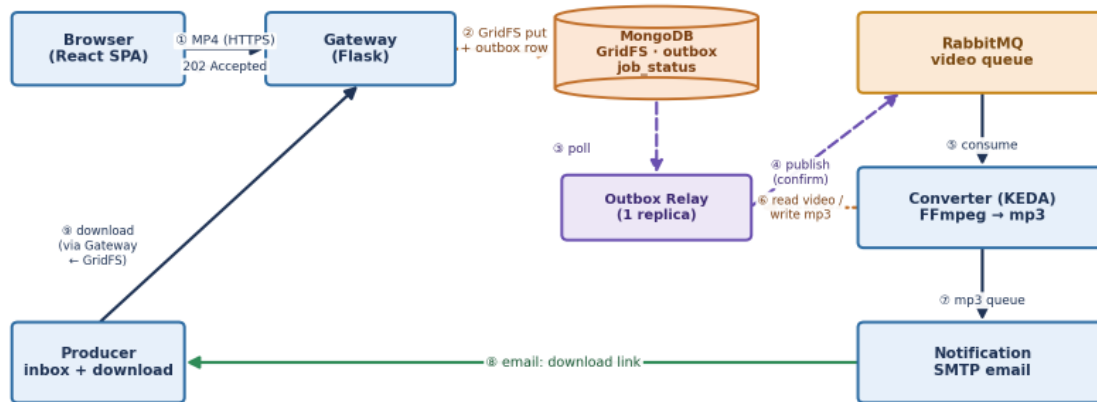


Figure 8.1 — The nine hops of a single upload. Solid = synchronous; dashed (purple) = the outbox-relay publish path; dotted = datastore reads/writes. The gateway returns `202` at step ②; everything after is asynchronous.

The hops, traced through the code:

1. **Upload.** The React SPA POSTs the MP4 to the gateway over HTTPS (`POST /upload`, rate-limited to 20/hour with a per-file cost).
2. **Store + enqueue, then 202.** `storage/util.upload()` writes the video to MongoDB GridFS (`fs.put`, tagging `metadata.owner_email` from the JWT), inserts the `queued job_status` document, and either writes an `outbox` row (A1) or publishes directly to the `video` queue. The gateway returns `202 Accepted` and increments `vidcast_uploads_total`.

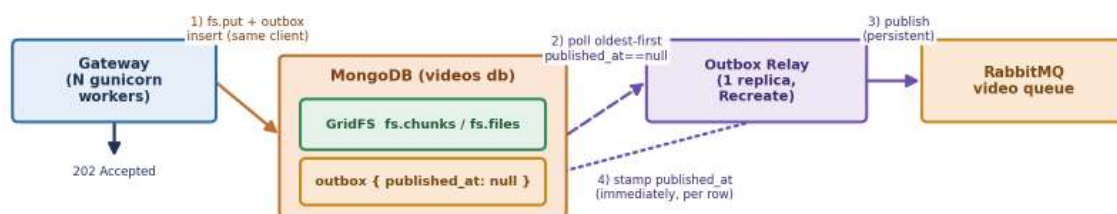
3. **Relay polls.** When `OUTBOX_ENABLED=true`, the single-replica outbox-relay polls the `outbox` collection for rows where `published_at` is null.
4. **Relay publishes.** It publishes each event to the `video` queue (persistent delivery, stamping the AMQP `timestamp` for the latency SLO) and immediately stamps `published_at`.
5. **Converter consumes.** A KEDA-scaled converter pod consumes from `video`, claims the job in Redis (A2), and marks `job_status processing`.
6. **Convert.** `convert/to_mp3.py` streams the video out of GridFS to a temp file, runs MoviePy/FFmpeg, writes the MP3 back to the `mp3s` GridFS (copying the owner tag), and records `vidcast_conversion_duration_seconds`.
7. **mp3 queue.** The converter publishes the result to the `mp3` queue and marks `job_status ready` with the `mp3_fid` and size.
8. **Notify.** A notification pod consumes from `mp3`, claims it in Redis (A2), and emails the producer a download link; it increments `vidcast_notifications_total{status="success"}` — the end-to-end SLO numerator.
9. **Download.** The producer downloads the MP3 through the gateway, which streams it back out of GridFS.

Why 202, not 200. Returning `202 Accepted` before conversion is what makes the platform resilient under load: the slow, CPU-heavy FFmpeg step is decoupled from the request, absorbed by the queue, and scaled independently by KEDA. The reliability patterns in §8.2–§8.4 exist to make that decoupling safe.

8.2 Transactional Outbox (A1) — Atomic File + Outbox Write, Single Relay with Publisher Confirms

The hazard A1 removes is the **dual-write problem**: the gateway must both store the file and tell RabbitMQ about it. If it stored the file and then the broker was momentarily unreachable, the upload would be accepted but the conversion event lost — a silently dropped file, exactly the failure VidCast was built to eliminate.

Figure 8.2 — Transactional Outbox (A1): one writer, one publisher



Crash before step 4 → row stays unpublished → re-published next cycle → A2 makes the duplicate a no-op.

Figure 8.2 — The outbox pattern: the gateway writes the GridFS object and the outbox row through the same MongoDB client; a single-replica relay later publishes unpublished rows and stamps them.

The write side (gateway)

In `storage/util.upload()`, when `OUTBOX_ENABLED=true`, instead of publishing to RabbitMQ the gateway inserts a row into the `outbox` collection that lives in the **same** ``videos`` database as the GridFS video — so the file write and the event write go through the same MongoDB client:

```
outbox.insert_one({
  "event_type": "video.uploaded",
  "routing_key": "video",
  "payload": message,           # video_fid, username, correlation_id, batch_id ...
  "created_at": utcnow(),
  "published_at": None,         # <- "not yet published" marker
})
```

The platform runs on a **standalone mongod**, so there is no multi-document ACID transaction spanning the GridFS chunks and the outbox row. A1 is therefore implemented honestly as **ordering plus a compensating action**: the GridFS `fs.put` happens first, then the outbox insert; if the insert fails, the gateway deletes the orphaned GridFS object (`fs.delete`) and the queued `job_status`, and returns an error. The ordering and the compensating delete bound the inconsistency window to a single failed call.

The publish side (the relay)

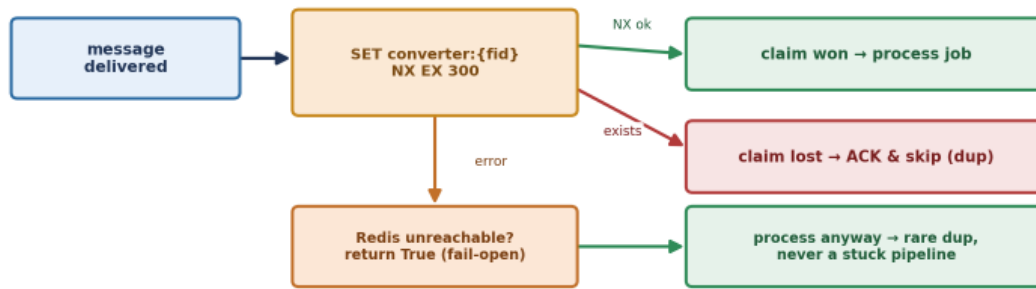
Publishing is delegated to a **separate, single-replica `outbox-relay` Deployment** — deliberately not an in-process thread in the gateway. The gateway runs under Gunicorn with multiple worker processes (A4); an in-process relay would run once per worker, giving N concurrent publishers and re-introducing the very duplicate-publish bug the outbox exists to kill. **One replica = one publisher = no double-send by construction**, which is why the Deployment is pinned to `replicas: 1` with `strategy: Recreate` (never RollingUpdate, which would briefly run two relays). The relay loop:

- Every `OUTBOX_POLL_INTERVAL` seconds (default 30), find `outbox` rows with `published_at == null`, **oldest first** (`sort("created_at", 1)`) so events publish in production order.
- Publish each to RabbitMQ with `PERSISTENT_DELIVERY_MODE` and the publish `timestamp` (the same SLO clock-start the gateway's direct path uses, keeping latency measurement consistent across both paths).
- **Stamp `published\_at` immediately after each publish** — per row, not batched at the end — so a crash mid-loop only ever leaves **unpublished** rows to retry; it can never lose the record that a publish already happened.
- On any error (Mongo or RabbitMQ unreachable, a publish failure) it logs, skips the cycle, and retries on the next poll. **The pod never crashes** on a dependency outage.

The exactly-once illusion. A relay restart between publishing a row and stamping it will re-publish that row next cycle. A1 does not prevent this — it deliberately leans on A2: the rare duplicate is claimed-once in Redis and becomes a no-op. A1 (never lose) + A2 (never duplicate) together yield effectively-once delivery without distributed transactions.

8.3 Idempotent Consumers (A2) — Redis SETNX Claim-Once, At-Least-Once → Effectively-Once

Because A3 retries and A1 relay-restarts both make delivery **at-least-once**, a message can legitimately arrive more than once. Without a guard, a redelivery means a duplicate conversion → duplicate MP3 → duplicate email. A2 makes processing idempotent with a **claim-once** guard shared (by copy) between the converter and notification services (`idempotency.py`).

Figure 8.3 — Idempotent Consumer (A2): Redis SET NX EX claim-once

release(key) is called ONLY on a retryable failure — kept on success (suppress dup) and on terminal DLQ (never reprocess).

Figure 8.3 — The claim-once decision. The first delivery wins the atomic Redis claim and processes; a duplicate loses the claim, ACKs, and skips. A Redis outage fails open.

Mechanism

The guard is a single atomic Redis operation — `SET key value NX EX ttl` — which sets the key only if absent. Concurrent deliveries race safely: exactly one wins the claim. The TTL (`IDEMPOTENCY_TTL_SECONDS`, default 300 s) bounds the dedup window so a crash can never wedge a claim forever.

```
def claim_once(key):
    try:
        return bool(_redis().set(name=key, value="1", nx=True,
                                ex=CLAIM_TTL_SECONDS))
    except Exception:
        return True          # FAIL OPEN: process anyway if Redis is down
```

Keys are **namespaced per service** so the two pipelines never collide: the converter claims `converter:{video_fid}`; the notification service claims `notification:{mp3_fid}`. If the claim is lost (already set), the consumer ACKs and returns immediately — the duplicate is dropped without side effects.

Three rules that make it correct

- **Fail open.** If Redis is unreachable, `claim_once` returns `True` and the message is processed. A Redis outage degrades the system to “possible occasional duplicate,” never a stuck pipeline — a rare double-email beats halting all conversions because the dedup store is down.
- **Release only on retry.** `release(key)` is called *only* when A3 routes a message to a retry, so the next attempt can re-claim. The claim is **kept on success** (to suppress a later duplicate delivery) and **kept on terminal DLQ** (so an unfixable job is never reprocessed). This three-way handling is visible in both consumers’ callbacks.
- **ACK after deciding.** The original delivery is always ACKed once the consumer has either processed it, skipped it as a duplicate, or re-published it via A3 — so a message is never left unacked to be redelivered by the broker on top of the application-level retry.

Outcome	Claim action	Why
Success	keep claim	suppress any later duplicate of the same job

Retryable failure	release claim	let the next A3 attempt re-claim and reprocess
Terminal (DLQ) failure	keep claim	permanent failure — never reprocess
Redis down	treat as claim won	fail open — never stall the pipeline

Table 8.2 — How the idempotency claim is handled per processing outcome (converter & notification consumers).

8.4 Bounded Retry + Dead-Letter Queues (A3) — Backoff, Max-Retries, Poison-Pill Parking

A3 turns failures into a bounded, observable process instead of two bad extremes: dropping the message, or NACK-requeuing it forever (the “poison-pill” loop, fixed as defect L-4). It is implemented in `rabbitmq_retry.py` and declared from code on every consumer startup, so the topology is reproducible from scratch with no manual queue creation.

Topology

For a main queue (e.g. `video`) the helper declares two extra queues and one exchange — all additive, leaving the main queue and message payload unchanged:

Object	Type	Behaviour
<code><main>.retry</code>	durable queue	<code>x-message-ttl = RETRY_TTL_MS</code> ; on TTL expiry dead-letters via the default exchange back to <code><main></code> for another attempt
<code><main>.dlq</code>	durable queue	bound to <code>vidcast.dlx</code> ; terminal — messages land here after <code>MAX_RETRIES</code> and are not auto-retried
<code>vidcast.dlx</code>	direct exchange	routes terminal failures to the matching <code><main>.dlq</code>

Table 8.3 — A3 retry/DLQ objects declared by `rabbitmq_retry.declare_topology()`.

Critically, consumers do **not** consume `<main>.retry`: the TTL-expiry dead-letter performs the delay-and-reinject automatically. This is a delayed-retry queue built from native RabbitMQ primitives — no plugins required.

The retry counter and the timeline

The retry count is tracked in an **explicit** `x-retry-count` header that the consumer increments, rather than the broker’s `x-death` header (which varies by RabbitMQ version and counts per-queue). On a failure, `handle_failure()` reads the header: while `retry_count < MAX_RETRIES` it re-publishes to `<main>.retry` (persistent) with an incremented count and returns `"retry"`; once the limit is reached it publishes to `vidcast.dlx` → `<main>.dlq` and returns `"dlq"`. The caller then ACKs the original delivery, so the message leaves the main queue — **breaking the infinite requeue loop**.

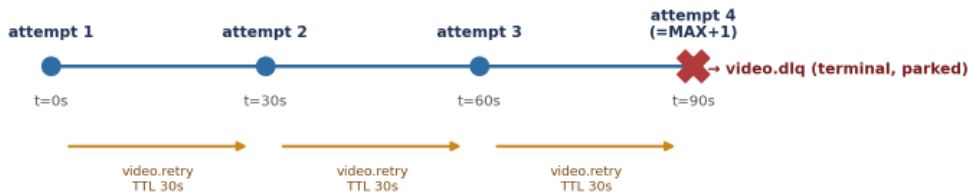
Figure 8.4 — Bounded Retry Timeline (MAX_RETRIES=3, RETRY_TTL_MS=30000)

Figure 8.4 — With the defaults (MAX_RETRIES=3, RETRY_TTL_MS=30000), a persistently failing message is attempted at t=0, 30, 60, 90 s, then parked in the terminal DLQ — total dwell ≈ 90 s before a human inspects it.

How consumers use it

Both consumers wrap processing in a try/except and route failures uniformly. The converter additionally distinguishes status: a terminal DLQ outcome marks `job_status` `failed`, while a pending retry stays `processing`. The return value also drives A2 — the claim is released only on `"retry"`:

```
outcome = rabbitmq_retry.handle_failure(ch, properties, body, video_queue)
if outcome != "retry":
    _set_status(video_fid, status="failed") # terminal
if job_id and outcome == "retry":
    idempotency.release(job_id) # allow re-claim
ch.basic_ack(delivery_tag=method.delivery_tag) # leave the main queue
```

The conversion path also hardens the worst offender directly: MoviePy/FFmpeg on a corrupt video used to raise out of `to_mp3.start` and crash the consumer; that call is now wrapped so the failure is caught and routed through retry/DLQ like any other, keeping the pod alive. Every attempt also increments `vidcast_conversions_total{status="success"|"failure"}` (and notifications likewise), so the DLQ rate is observable in Grafana.

8.5 Health Probes and Graceful Degradation (HTTP /healthz, File-Based Heartbeats)

Reliability also means Kubernetes restarts the **right** pods at the **right** time — and, just as importantly, does not kill a healthy pod because a dependency blipped. VidCast uses two probe styles matched to the two kinds of workload.

HTTP probes for request servers (gateway, auth)

The gateway and auth services expose `GET /healthz`. The gateway's endpoint is a genuine **dependency check**: it pings MongoDB and opens a RabbitMQ connection, returning `200 {"status":"ok"}` only if both succeed and `503 {"status":"degraded", "checks":{"...}}` otherwise. Both liveness and readiness probes target it:

Service	Probe	Path / command	initialDelay	period	failureThreshold
Gateway	liveness	<code>GET /healthz:8080</code>	15s	10s	3

Gateway	readiness	<code>GET /healthz:8080</code>	5s	5s	3
Auth	liveness/readiness	<code>GET /healthz</code>	15s / 5s	10s / 5s	3
Converter	liveness	<code>test -f /tmp/healthy</code>	15s	10s	3
Notification	liveness	<code>test -f /tmp/healthy</code>	15s	10s	3
Outbox Relay	liveness	<code>test -f /tmp/healthy</code>	15s	10s	3

Table 8.4 — Probe configuration per workload (`k8s/base/*/deployment.yaml`).

File-based heartbeats for queue consumers

The converter, notification, and outbox-relay have no HTTP server, so liveness is a file check: `test -f /tmp/healthy`. The semantics of `*when*` the file is touched encode the resilience policy and differ subtly by service:

- **Touched on startup.** Every consumer touches `/tmp/healthy` as soon as it is connected and ready to consume — before any message arrives. Without this, an idle consumer (no work yet) would never create the file and would crash-loop on the probe. For the notification service this is doubly important: if email delivery is misconfigured, the per-message touch never runs, so the **startup touch is the only thing keeping the pod alive** while the operator fixes credentials.
- **Refreshed on successful work.** The converter and notification re-touch the file after each successfully processed message, so liveness reflects real progress.
- **Touched every loop — even during outages (relay).** The outbox-relay touches `/tmp/healthy` on `*every*` cycle, including cycles where Mongo or RabbitMQ is down. This is deliberate: the relay **process** is healthy as long as its loop turns; a broker outage is a transient condition it retries through, not a reason for Kubernetes to kill and reschedule the pod.

Graceful degradation

The same philosophy runs through the data paths: every external dependency is treated as something that may be briefly absent, and the response is to degrade rather than fail hard.

Dependency down	VidCast's response	Net effect
Redis (idempotency)	<code>claim_once</code> fails open → process anyway	rare duplicate, never a stall
Redis (rate-limit)	flask-limiter degrades to in-memory counters	request still served
RabbitMQ (relay publish)	log, skip cycle, retry next poll; pod stays up	events drain when broker returns
MongoDB / broker (gateway)	<code>/healthz</code> returns 503 → out of rotation	traffic shifts to healthy replicas
<code>job_status</code> write	best-effort, wrapped, never raises	status is a UX nicety, never blocks a job

Table 8.5 — Graceful-degradation responses to dependency failure.

8.6 Feature Flags (`OUTBOX_ENABLED`, `IDEMPOTENCY_ENABLED`, `MAX_RETRIES`, `RETRY_TTL_MS`)

The reliability patterns are governed by environment-variable feature flags set in the services' ConfigMaps (`k8s/base/*/configmap.yaml`). This keeps every pattern independently toggleable and tunable per

environment without a code change or image rebuild — and lets behaviour fall back to the exact legacy path when a flag is off, which is how the patterns were rolled out incrementally and de-risked.

Flag	Default	Scope	Effect
<code>OUTBOX_ENABLED</code>	<code>false</code>	gateway	<code>true</code> routes uploads through the outbox + relay (A1); <code>false</code> publishes directly to RabbitMQ (legacy path, byte-for-byte unchanged)
<code>IDEMPOTENCY_ENABLED</code>	<code>false</code>	converter, notification	<code>true</code> enables the Redis claim-once guard (A2); <code>false</code> processes every delivery
<code>IDEMPOTENCY_TTL_SECONDS</code>	<code>300</code>	converter, notification	dedup window / claim TTL in seconds
<code>MAX_RETRIES</code>	<code>3</code>	converter, notification	retry attempts before terminal DLQ (A3)
<code>RETRY_TTL_MS</code>	<code>30000</code>	converter, notification	delay (ms) a message waits in <code><main>.retry</code> between attempts
<code>OUTBOX_POLL_INTERVAL</code>	<code>30</code>	outbox-relay	seconds between relay poll cycles
<code>REDIS_HOST</code>	<code>redis</code>	gateway, converter, notification	in-cluster Redis service for A2 + rate limiting

Table 8.6 — Reliability feature flags, their defaults, and effects (`k8s/base/*/configmap.yaml`).

Why the durability flags default to off

`OUTBOX_ENABLED` and `IDEMPOTENCY_ENABLED` both ship **disabled by default**, which is a deliberate, safe-rollout decision rather than an oversight. Each pattern was added behind a flag whose `false` branch preserves the previous behaviour exactly — the gateway’s direct-publish path is kept verbatim — so the feature can be enabled, observed, and rolled back per environment with no code change. The two flags are also **designed to be enabled together**: A1’s relay can re-publish a row after a restart, so turning on the outbox without idempotency would expose occasional duplicates. Enabling A2 alongside A1 closes that window, which is exactly the effectively-once posture described in §8.2–§8.3.

Tuning guidance

- ``RETRY_TTL_MS` × `MAX_RETRIES`` sets the worst-case dwell before a poison message is parked (≈ 90 s at the defaults). Raise the TTL for failures that need longer to clear (e.g. a flaky downstream); raise `MAX_RETRIES` only if transient failures genuinely succeed on later attempts — otherwise it just delays the inevitable DLQ.
- ``IDEMPOTENCY_TTL_SECONDS`` must comfortably exceed the longest realistic retry window (`MAX_RETRIES × RETRY_TTL_MS`) so a claim does not expire mid-retry; 300 s against a 90 s retry window leaves ample headroom.
- ``OUTBOX_POLL_INTERVAL`` trades publish latency against Mongo load. The default 30 s adds bounded latency to the outbox path; lower it for snappier delivery at the cost of more frequent polls.

In summary, VidCast’s reliability is layered and self-reinforcing: the outbox guarantees an accepted upload is never lost (A1); the single-replica relay guarantees it is published exactly by one writer; idempotent consumers guarantee an inevitable duplicate is harmless (A2); bounded retry with dead-lettering guarantees transient failures recover and permanent ones are parked, not looped (A3); and probe semantics plus fail-

open degradation guarantee Kubernetes restarts only genuinely unhealthy pods. Every one of these is delivered in code against the in-cluster brokers and governed by feature flags — which is why §7.6 could justify keeping the datastores self-hosted at \$0 standing cost while still claiming a real, demonstrable reliability story.

Part III — Development: Build & Delivery (Dev)

Chapter 9 — Infrastructure as Code (Terraform)

Every AWS resource VidCast runs on is declared in Terraform — nothing is built by clicking in the console. The infrastructure is a set of Git commits that provisions a complete, hardened EKS platform: networking, the cluster and its node group, container registry, secrets federation, load-balancer and backup IAM, and keyless CI/CD federation. This is what makes the platform’s defining cost decision possible: the whole stack can be `terraform destroy`-ed to roughly £0 between demos and rebuilt from scratch in under 20 minutes (§9.1). This chapter walks the code as it exists under `terraform/`, module by module.

The layout separates **reusable modules** from a thin **per-environment root** that wires them together:

```
terraform/
├── environments/
│   └── dev/          # root module: main.tf, variables.tf, backend.tf,
│                   # outputs.tf, terraform.tfvars(.example)
├── modules/
│   ├── vpc/ iam/ eks/ security-groups/ ecr/
│   └── external-secrets/ lbc/ storage/ github-oidc/
```

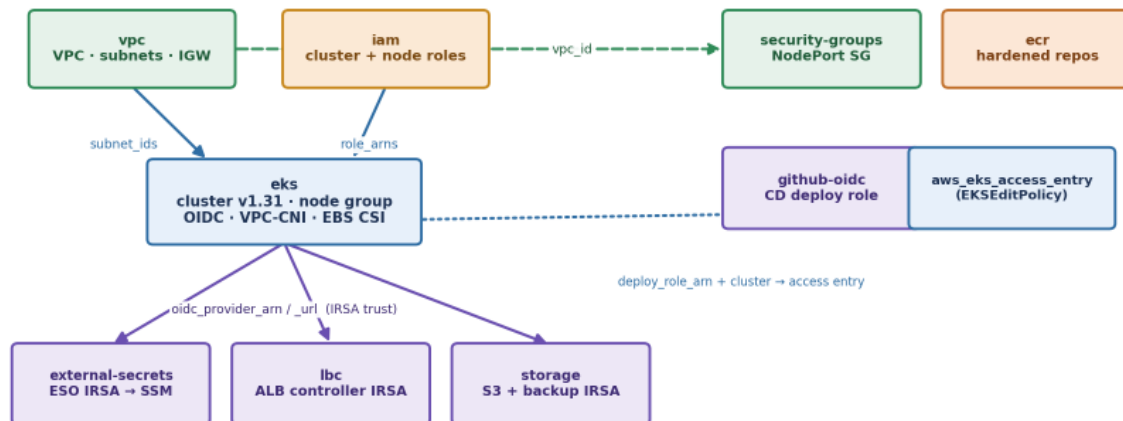
#	Module	Provisions	§
1	vpc	VPC, public subnets across 2 AZs, IGW, route table, ELB tags	9.2
2	iam	EKS cluster role + node role (managed-policy attachments)	9.3
3	eks	Cluster v1.31, node group, OIDC provider, VPC-CNI + EBS-CSI add-ons	9.4
4	security-groups	NodePort ingress SG (30002–30008)	9.5
5	ecr	Hardened image repos: immutable tags, scan-on-push, lifecycle	9.6
6	storage	Versioned encrypted S3 backup bucket + backup IRSA role	9.7
7	external-secrets	ESO IRSA role → SSM Parameter Store (least-privilege)	9.8
8	lbc	AWS Load Balancer Controller IRSA role + policy	9.9
9	github-oidc	GitHub OIDC provider + CD deploy role	9.10

Table 9.1 — The nine Terraform modules and what each provisions.

9.1 Module Structure and Reproducibility (Full Rebuild in < 20 Minutes)

The design principle is **small, single-purpose modules composed by a per-environment root**. Each module owns one concern and exposes only the outputs other modules need; the `environments/dev` root passes those outputs between modules as inputs. Terraform reads these references to build a dependency graph and resolve apply order automatically — there is no hand-written sequencing.

Figure 9.1 — Terraform Module Dependency Graph (environments/dev)



Foundation → eks (OIDC) → IRSA-consuming modules. security-groups, ecr, github-oidc are independent.

Apply order is resolved automatically by Terraform from these references — no manual sequencing.

Figure 9.1 — Module dependency graph. The foundation (vpc, iam) feeds eks; eks's OIDC provider feeds every IRSA-consuming module; security-groups, ecr and github-oidc are independent. Terraform derives apply order from these edges.

Composition in the root module

`environments/dev/main.tf` instantiates all nine modules and threads outputs between them — e.g. the EKS module receives the VPC's subnets and IAM's role ARNs, and the IRSA modules receive the EKS OIDC provider:

```
module "eks" {
  cluster_role_arn = module.iam.cluster_role_arn
  node_role_arn    = module.iam.node_role_arn
  subnet_ids       = module.vpc.public_subnet_ids # ...
}
module "external_secrets" {
  oidc_provider_arn = module.eks.oidc_provider_arn
  oidc_provider_url = module.eks.oidc_provider_url
}
```

State, locking and providers

- Remote state** is stored in S3 with a `vidcast/dev/terraform.tfstate` key; **state locking** uses a DynamoDB table (`vidcast-terraform-locks`). Both are supplied at `terraform init` time via `-backend-config` (see `backend.tf`), so no account-specific values are committed.
- Providers** are pinned in `backend.tf`: `required_version >= 1.5`, `hashicorp/aws ~> 5.0`, and `hashicorp/tls ~> 4.0` (the TLS provider derives OIDC thumbprints dynamically). The AWS provider region comes from `var.aws_region` (default `eu-west-2`).

Reproducibility and the cost story

Because 100% of the platform is declarative, the cluster is treated as **cattle, not a pet**: it is deliberately torn down to drive the standing bill toward zero and rebuilt on demand. The README and decision records state the rebuild target plainly — a full `terraform apply` stands the platform back up in **under 20 minutes**. The one manual step the code documents is a single import of the pre-existing `vidcast-frontend` ECR repository before the first apply (§9.6); everything else is greenfield each time. The thirteen `outputs.tf` values (cluster endpoint, kubeconfig command, OIDC ARN, deploy-role ARN, IRSA role ARNs, ECR URLs, backup bucket) feed straight into the post-Terraform deploy steps, so the handoff from IaC to the application layer is itself scripted.

9.2 VPC Module — Subnets across AZs, Internet Gateway, Routes, ELB Tags

The `vpc` module builds the network the cluster lives in: a VPC (`10.0.0.0/16`) with DNS support and hostnames enabled, an Internet Gateway, and a **public subnet in each availability zone** (`eu-west-2a`, `eu-west-2b`). Subnet CIDRs are carved deterministically with `cidrsubnet(var.vpc_cidr, 8, count.index + 1)`, and a single public route table sends `0.0.0.0/0` to the IGW, associated to every subnet.

Kubernetes-aware subnet tags

The subnets carry tags that are **load-bearing for EKS and the ALB controller**, not cosmetic:

Tag	Value	Why it matters
<code>kubernetes.io/role/elb</code>	<code>1</code>	marks the subnet as eligible for public (internet-facing) load balancers — the LBC discovers ALB subnets by this tag
<code>kubernetes.io/cluster/<name></code>	<code>shared</code>	associates the subnet with the cluster so EKS and the CNI recognise it
<code>Name</code>	<code><cluster>-public-N</code>	human-readable identification

Table 9.2 — Subnet tags applied by the `vpc` module (`modules/vpc/main.tf`).

Design note. The architecture is intentionally **public-subnet only** — there are no private subnets or NAT gateways. This is a cost decision: NAT gateways carry an hourly + per-GB charge, and on a single-node portfolio cluster the security boundary is provided by NetworkPolicies, security groups, and the ALB-only ingress rather than subnet privacy. `map_public_ip_on_launch = true` lets nodes reach the EKS API and pull images directly.

9.3 IAM Module — Cluster Role, Node Role, IRSA Trust Policies

The `iam` module creates the two foundational service roles EKS requires, each with a tightly-scoped assume-role trust policy and AWS-managed permission policies — no inline custom permissions:

Role	Trusted principal	Attached managed policies
<code><cluster>-cluster-role</code>	<code>eks.amazonaws.com</code>	<code>AmazonEKSClusterPolicy</code>
<code><cluster>-node-role</code>	<code>ec2.amazonaws.com</code>	<code>AmazonEKSWorkerNodePolicy</code> , <code>AmazonEKS_CNI_Policy</code> , <code>AmazonEC2ContainerRegistryReadOnly</code>

Table 9.3 — The two EKS service roles (`modules/iam/main.tf`).

The node role's `AmazonEC2ContainerRegistryReadOnly` attachment is what lets worker nodes pull images from ECR without any static credentials. Critically, this module **does not** create IRSA (workload) roles. Those live in separate modules (`external-secrets`, `lbc`, `storage`, and the EBS-CSI role inside `eks`) because an IRSA role's trust policy depends on the EKS OIDC provider, and the EKS module already depends on this IAM module — putting an OIDC-consuming role here would form an `iam` → `eks` → `iam` dependency cycle. Splitting them keeps the graph acyclic, a deliberate decision documented in the module comments.

9.4 EKS Module — Cluster v1.31, Node Group, OIDC Provider, VPC-CNI, EBS CSI Add-on

The `eks` module is the heart of the platform. It provisions five tightly-related resources:

1) The cluster

`aws_eks_cluster` runs Kubernetes **v1.31** with `authentication_mode = API_AND_CONFIG_MAP`, which enables EKS **access entries** (used in §9.10 to grant the CD role `kubectl` rights) while keeping `aws-auth` working, and `bootstrap_cluster_creator_admin_permissions = true` so the applying principal is cluster admin. The API endpoint is public (`endpoint_public_access = true`), private access off.

2) The node group

A managed node group of **m7i-flex.large** instances (`AL2_x86_64` AMI), sized `min 1 / desired 1 / max 2`. The instance family is constrained for a reason documented in the variables: `"Must be M/C/R-series — T-type is blocked by SCP."` m7i-flex gives 2 vCPU / 8 GB at a small flex discount — the right-sized compute behind every resource-limit and `maxReplicas` decision elsewhere in the project.

3) The OIDC provider

`aws_iam_openid_connect_provider` is created from the cluster's issuer URL, with its thumbprint derived dynamically from the issuer's TLS certificate (so it survives a CA rotation). **This single resource is the root of all IRSA** — every keyless workload role (ESO, LBC, backup, EBS-CSI) trusts this provider.

4) VPC-CNI add-on with NetworkPolicy enforcement

The `vpc-cni` add-on is installed with `enableNetworkPolicy = "true"`:

```
resource "aws_eks_addon" "vpc_cni" {
  addon_name = "vpc-cni"
  configuration_values = jsonencode({ enableNetworkPolicy = "true" })
}
```

Why this matters. Without `enableNetworkPolicy`, `NetworkPolicy` objects are accepted by the API server but **never enforced** — the zero-trust default-deny (Chapter on Security) becomes decorative YAML that silently does nothing. This flag flips on the eBPF agent in the `aws-node` DaemonSet. It is set here so it is configured while the cluster is applied from scratch, because toggling it on a live cluster recycles `aws-node`.

5) EBS CSI driver add-on (with its own IRSA)

Kubernetes 1.31 removed the in-tree `kubernetes.io/aws-ebs` provisioner, so dynamically-provisioned EBS PVCs would stay **Pending** forever. The module installs the managed `aws-ebs-csi-driver` add-on and grants its controller permissions via a dedicated IRSA role (`<cluster>-ebs-csi-irsa`, trusting only `kube-system:ebs-csi-controller-sa`) with the AWS-managed `AmazonEBSCSIDriverPolicy` — no node-wide EBS

permissions, no static keys. This is what lets the PostgreSQL gp3 PVC (Chapter 7) actually bind. `depends_on` wiring ensures the node group exists (to schedule the controller) and the role is attached before the SA is annotated.

9.5 Security-Groups Module — NodePort and ALB Rules

The `security-groups` module creates a single security group, `<cluster>-nodeport-sg`, that opens the cluster's NodePort range to inbound TCP. The ingress rules are generated with a `dynamic "ingress"` block over the `nodeport_ports` variable, which the root module sets to **30002–30008**, with all-outbound egress:

Port	Service exposed
30002	Gateway API
30003	PostgreSQL (admin/debug)
30004	RabbitMQ
30005	Redis
30006	Frontend
30007	Grafana
30008	Alertmanager

Table 9.4 — NodePort ports opened by the security group (root sets the list; ports map to k8s services).

Each port becomes a discrete ingress rule from `0.0.0.0/0` so a NodePort service can be reached directly during development and demos. In the production posture the public door is the ALB (provisioned by the LBC, §9.9) terminating HTTPS; the NodePort SG remains as the development/diagnostic path and is the reason the deck can state "ALB is the only public door" for prod traffic while still allowing direct node access when needed.

9.6 ECR Module — Immutable Tags, Scan-on-Push, Encryption, Lifecycle Policy

The `ecr` module (part of the A8 supply-chain work) creates hardened image repositories from the `repository_names` list — currently just `vidcast-frontend` (the Python backends live on Docker Hub today). It applies four controls, all free within ECR limits:

Control	Setting	Purpose
Tag immutability	<code>image_tag_mutability = "IMMUTABLE"</code>	a pushed tag can never be overwritten, so a digest verified once (cosign) can't be swapped under the same tag
Scan-on-push	<code>scan_on_push = true</code>	a basic CVE scan on every push — defence in depth behind the CI Trivy gate
Encryption	<code>AES256</code> (AWS-managed key)	encrypted at rest; a customer-managed KMS key is deliberately skipped (~\$1/mo for marginal benefit)
Lifecycle	expire untagged > 7 days; keep last 10 images	repo can't grow unbounded and bill over time

Table 9.5 — ECR hardening controls (modules/ecr/main.tf; defaults from variables.tf).

The lifecycle policy is built as two ordered JSON rules: rule 1 expires untagged images older than `untagged_expire_days` (default 7), rule 2 keeps only the `keep_last_images` (default 10) most recent. Because

the `vidcast-frontend` repo predates this module, the root `main.tf` documents the **one-time import** required before the first apply, otherwise apply fails with "repository already exists":

```
terraform import \
  'module.ecr.aws_ecr_repository.this["vidcast-frontend"]' \
  vidcast-frontend
```

9.7 Storage Module — Versioned S3 Backup Bucket + Backup IRSA Role

The `storage` module provisions the durability backstop for the stateful tier (detailed operationally in Chapter 7 §7.5): a private S3 bucket the nightly `mongodump/pg_dump` CronJobs write to, plus the IRSA role they assume. The bucket name is deterministic — `vidcast-backups-<account_id>` — so the CronJobs can hardcode it without a Terraform→Kustomize value handoff.

Bucket controls

- **Versioning enabled** — a bad dump can't immediately overwrite the last good one.
- **All public access blocked** — backups can contain user data, so never public.
- **AES256 server-side encryption** (AWS-managed key; no CMK, consistent with ECR/ESO cost decisions).
- **Lifecycle**: expire objects and noncurrent versions after `retention_days` (30), and abort incomplete multipart uploads after 7 days.

Least-privilege backup role

The IRSA role `<cluster>-backup-irsa` trusts only `default:vidcast-backup` on this cluster's OIDC provider, and its inline policy grants exactly two actions, scoped to **this bucket only**: `s3:ListBucket` on the bucket ARN and `s3:PutObject` on its objects. The CronJobs can write dumps and nothing else — they cannot read or delete existing backups, so a compromised backup job can't exfiltrate or destroy history (restore uses a separate role).

9.8 External-Secrets Module — ESO IRSA + SSM Parameter Store Integration

The `external-secrets` module (the A9 work) provisions the IRSA role the External Secrets Operator assumes to pull VidCast's secrets from AWS **Systems Manager Parameter Store** — chosen over Secrets Manager precisely to avoid the \$0.40/secret/month charge (standard-tier SSM parameters are free, and SecureString uses the free AWS-managed `alias/aws/ssm` key).

Figure 9.2 — Keyless Access via OIDC + IRSA (no static AWS keys)



Trust condition: `sub = system:serviceaccount:<ns>:<name> AND aud = sts.amazonaws.com`

Used by: `external-secrets`, `lbc`, `storage (backup)`, `eks/ebs-csi` · GitHub Actions uses the same flow via `github-oidc`

Figure 9.2 — The keyless IRSA flow shared by ESO, the LBC, and the backup jobs: a ServiceAccount's projected OIDC token is exchanged for short-lived role credentials. No long-lived AWS keys exist anywhere.

Scoped trust and least-privilege read

The role `<cluster>-external-secrets-irsa` trusts only `default:vidcast-eso` (sub condition) with `aud = sts.amazonaws.com`. Its permission policy grants read on **only** `/vidcast/*` parameters and a `kms:Decrypt` that is constrained to Parameter Store via a condition — so this role cannot decrypt arbitrary KMS data elsewhere:

```
# read only /vidcast/* parameters
actions    = ["ssm:GetParameter", "ssm:GetParameters", "ssm:GetParametersByPath"]
resources  = ["arn:aws:ssm:<region>:<account>:parameter/vidcast/*"]

# decrypt ONLY via the SSM service (not arbitrary KMS use)
actions    = ["kms:Decrypt"]
condition { kms:ViaService = "ssm.<region>.amazonaws.com" }
```

The in-cluster `ClusterSecretStore` points at the `vidcast-eso` ServiceAccount annotated with this role's ARN (surfaced as the `external_secrets_irsa_role_arn` output), completing the chain that keeps **zero secrets in Git**. Standing cost of the whole A9 design: **\$0**.

9.9 Load Balancer Controller Module — LBC IRSA Role and Policy

The `lbc` module provisions the IAM the AWS Load Balancer Controller needs to create and manage ALBs from `Ingress` resources. The controller runs in `kube-system` and assumes the role `<cluster>-lbc-irsa` through its `aws-load-balancer-controller` ServiceAccount (scoped trust on the cluster OIDC provider).

Rather than hand-craft permissions, the module attaches the **official AWS LBC IAM policy**, loaded from a vendored JSON file pinned to the controller version installed (**v2.8.1**):

```
resource "aws_iam_policy" "lbc" {
  name     = "${var.cluster_name}-lbc-policy"
  policy   = file("${path.module}/lbc-iam-policy.json") # pinned v2.8.1
}
```

The module comment records the exact `curl` used to fetch that JSON, so it can be re-synced when the chart/controller version changes. The policy covers `elasticloadbalancing`, `ec2`, `acm`, `wafv2`, and `shield` calls. Like the other IRSA modules, this is intentionally separate from `iam` to avoid the OIDC dependency cycle; its role ARN is exported as `lbc_irsa_role_arn` for the SA annotation. This is the IAM that makes "ALB is the only public door" real — the controller terminates HTTPS via an ACM certificate and routes to the frontend.

9.10 GitHub-OIDC Module — Keyless CI/CD Federation (No Stored AWS Keys)

The `github-oidc` module lets the CD workflow authenticate to AWS with a **short-lived, federated role** instead of long-lived access keys stored as GitHub secrets. It registers GitHub's OIDC issuer as an identity provider and creates a deploy role scoped to this repository.

Provider, trust, and least-privilege

- **OIDC provider** for `https://token.actions.githubusercontent.com`, thumbprint derived dynamically from the issuer's certificate (survives GitHub CA rotation).

- **Trust policy:** only the GitHub provider may assume `<cluster>-github-deploy`, and only for `repo:<org>/<repo>:*` (any branch/ref; the comment notes how to tighten to `:ref:refs/heads/main`). `aud` is pinned to `sts.amazonaws.com`.
- **Permission policy:** the only AWS API the workflow calls is `eks:DescribeCluster` (for `aws eks update-kubeconfig`), scoped to this one cluster ARN.

Two-layer authorization for CD

An IAM role alone only yields a `kubeconfig` — it does **not** grant Kubernetes-level rights. The root module closes that gap with an EKS **access entry**: it maps the deploy role to the cluster and associates the `AmazonEKSEditPolicy`, which is what lets `kubectl set image` actually patch deployments and read pods during CD.

```
resource "aws_eks_access_entry" "github_deploy" {
  principal_arn = module.github_oidc.deploy_role_arn
}
resource "aws_eks_access_policy_association" "github_deploy" {
  policy_arn = "arn:aws:eks::aws:cluster-access-policy/AmazonEKSEditPolicy"
  access_scope { type = "cluster" }
}
```

The deploy-role ARN is exported as `github_actions_role_arn`, to be set as the `AWS_DEPLOY_ROLE_ARN` GitHub secret. Net result: the CD pipeline holds **no AWS credentials** — it exchanges a per-run OIDC token for temporary, scoped access.

9.11 Environments (dev/prod) and tfvars Configuration

The `environments/dev` root is the only place modules are instantiated and the only place values are configured. Configuration is layered: typed `variables.tf` with sensible defaults, overridable by a gitignored `terraform.tfvars` (an `.example` is committed), with backend values injected at `init` time.

Key variables and defaults

Variable	Default	Notes
<code>aws_region</code>	<code>eu-west-2</code>	London — UK data residency
<code>cluster_name</code>	<code>vidcast-cluster</code>	prefix for nearly every resource name
<code>vpc_cidr</code>	<code>10.0.0.0/16</code>	subnets carved with <code>cidrsubnet(...,8,i+1)</code>
<code>availability_zones</code>	<code>[eu-west-2a, eu-west-2b]</code>	one public subnet each
<code>kubernetes_version</code>	<code>1.31</code>	drives the EBS-CSI requirement (\$9.4)
<code>node_instance_type</code>	<code>m7i-flex.large</code>	M/C/R-series only — T-type blocked by SCP
<code>node_min/desired/max_count</code>	<code>1 / 1 / 2</code>	single-node baseline, room for one more
<code>github_org / github_repo</code>	<code>johnnybabs / vidcast</code>	OIDC deploy-role trust scope
<code>state_bucket / state_lock_table</code>	<code>— / vidcast-terraform-locks</code>	remote state + DynamoDB lock

Table 9.6 — Selected root variables (`environments/dev/variables.tf`, `terraform.tfvars.example`).

Common tags and outputs

Every module receives a `common_tags` local — `Project=vidcast, ManagedBy=terraform, Environment=dev, Region=<region>` — so all resources are uniformly attributable (and cost-allocatable). The root publishes thirteen outputs that drive the downstream deploy: cluster endpoint and name, kubeconfig command, VPC and subnet IDs, the OIDC provider ARN, and the four "annotate this SA / set this secret" values (`github_actions_role_arn`, `external_secrets_irsa_role_arn`, `lbc_irsa_role_arn`, `backup_irsa_role_arn`) plus the ECR URLs and backup bucket name.

On the dev/prod split

The repository ships a single `dev` root, and the multi-environment story is handled **above Terraform** rather than by a second copied root: the GitOps layer keeps `k8s/overlays/dev` and `k8s/overlays/prod` (Argo CD auto-syncs dev, manual-syncs prod), and the secrets layer keeps `/vidcast/dev/*` and `/vidcast/prod/*` Parameter Store paths. The Terraform module set is environment-agnostic by construction — every module is parameterised by `cluster_name`, `aws_region`, and tags — so a `prod` root is a thin copy of `environments/dev` with its own `terraform.tfvars` and a distinct state key (`vidcast/prod/terraform.tfstate`). This keeps the IaC DRY: one set of audited modules, many environments.

In summary, VidCast's infrastructure is fully codified across nine focused Terraform modules wired by a thin per-environment root. The graph is deliberately acyclic — foundation (`vpc`, `iam`) → `eks` → OIDC-fed IRSA modules — and security is built in at the IaC layer: keyless IRSA everywhere (ESO, LBC, backup, EBS-CSI, and GitHub CD), least-privilege scoped policies, immutable scanned images, encrypted versioned backups, and enforced NetworkPolicy via the CNI flag. Because all of it is declarative with remote, locked state, the platform is genuinely reproducible — destroyable to ~£0 and rebuildable in under 20 minutes — which is the foundation the project's entire cost and reliability posture is built on.

Chapter 10 — Containerisation and Image Strategy

VidCast ships as six container images — five Python services and a React frontend — and the way those images are built and run is a deliberate security posture, not an afterthought. The strategy has four pillars: build minimal images (multi-stage where it pays, lean single-stage otherwise); run every container with a hardened, non-root, read-only security context; pin dependencies to CVE-free floors with documented rationale; and reference images by immutable, commit-pinned tags through GitOps. This chapter walks each as it exists in the repository (`src/*/Dockerfile`, `src/*/requirements.txt`, `src/frontend/package.json`, and `k8s/overlays/*`).

Image	Base	Build style	Runtime
auth-service	python:3.10-slim-bookworm	single-stage (+ build tools)	gunicorn, 2 workers, :5000
gateway-service	python:3.10-slim-bookworm	single-stage (+ build tools)	gunicorn, 2 workers, --timeout 120, :8080
converter-service	python:3.10-slim-bookworm	single-stage (+ ffmpeg)	python consumer.py
notification-service	python:3.10-slim-bookworm	single-stage (lean, no ffmpeg)	python consumer.py

outbox-relay	python:3.10-slim-bookworm	single-stage (lean, no build tools)	python relay.py
frontend	node:18-alpine → nginx:1.25-alpine	multi-stage	nginx, :8080

Table 10.1 — The six VidCast images, their bases, build styles and runtime entrypoints.

10.1 Multi-Stage Dockerfiles and Minimal Final Images

The goal is the smallest sensible final image — less to ship, less to scan, smaller attack surface. VidCast uses two patterns chosen per workload: a true multi-stage build for the frontend (where the build toolchain is large and entirely unneeded at runtime), and a lean single-stage build for the Python services (where the runtime *is* the interpreter, so a second stage would add complexity for little gain).

Figure 10.1 — Image Build Strategies: multi-stage (frontend) and lean single-stage (Python)

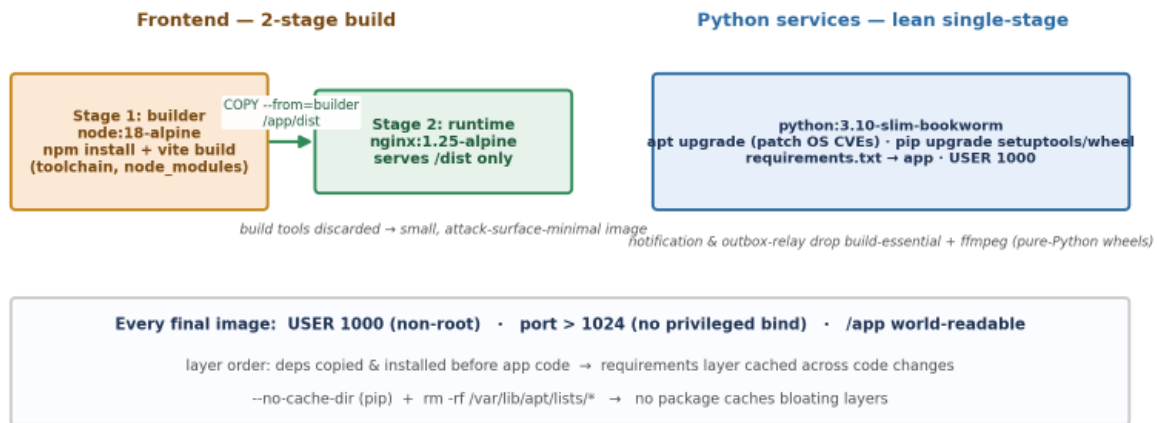


Figure 10.1 — Build strategies. The frontend discards its Node toolchain in a second nginx stage; the Python images stay single-stage but are kept lean and consistently hardened.

Frontend — genuine multi-stage

The frontend Dockerfile has two stages. **Stage 1** (`node:18-alpine`) runs `npm install` and `vite build`, producing static assets in `/app/dist`. **Stage 2** (`nginx:1.25-alpine`) copies only `/app/dist` and the nginx config — the entire Node toolchain and `node_modules` are left behind:

```
FROM node:18-alpine AS builder
RUN npm install && npm run build          # → /app/dist

FROM nginx:1.25-alpine
COPY --from=builder /app/dist /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
```

Build-time configuration is injected with a build arg — `ARG VITE_GRAFANA_URL` is baked into the bundle so the Dashboard page’s links point at the live Grafana NodePort — kept out of Git so it can track the current node IP, with an in-code fallback if unset.

Python services — lean single-stage

All five Python images share one disciplined recipe on `python:3.10-slim-bookworm`: patch the OS, upgrade the toolchain, install pinned requirements, copy app code, drop to a non-root UID. Two techniques keep them small and cacheable:

- **Layer ordering for cache reuse.** `requirements.txt` is copied and `pip install`-ed **before** the application code is copied (`COPY . /app`). Because the dependency layer only changes when dependencies change, day-to-day code edits reuse the cached install layer and rebuild fast.
- **No leftover caches.** `pip install --no-cache-dir` and `rm -rf /var/lib/apt/lists/*` ensure neither pip’s wheel cache nor apt’s package lists bloat the final layers.

The lean discipline is taken further per service. The **notification** and **outbox-relay** images deliberately omit the `build-essential/libpq-dev/python3-dev` toolchain entirely, because their only dependencies (pika, pymongo, certifi, urllib3, redis) ship as pure-Python or manylinux wheels needing no compilation — and notification additionally drops the ~100 MB `ffmpeg`/MoviePy media stack that was once inherited by copy-paste from the converter. Only the **converter** keeps `ffmpeg` (it does the actual conversion), and auth/gateway keep `build-essential + libpq-dev` (needed to build `psycopg2`).

Image	build-essential / libpq / py-dev	ffmpeg	Why
auth, gateway	yes	no	compile psycopg2-binary deps
converter	yes	yes	FFmpeg is the workload
notification	no	no	pure-Python wheels only; no media work
outbox-relay	no	no	pika + pymongo wheels only

Table 10.2 — Per-image toolchain footprint. Smaller is better; each inclusion is justified.

10.2 Non-Root UID 1000, Read-Only Root Filesystem, Dropped Capabilities

Every container runs under a hardened security context applied in two places that reinforce each other: the **Dockerfile** sets `USER 1000` so the image **cannot** start as root even if run outside Kubernetes, and the **Kubernetes deployment** asserts the same intent declaratively and adds the controls only the kubelet can enforce. The result is defence in depth — five independent controls between an exploited process and the node.

Figure 10.2 — Container Runtime Hardening (defence in depth)

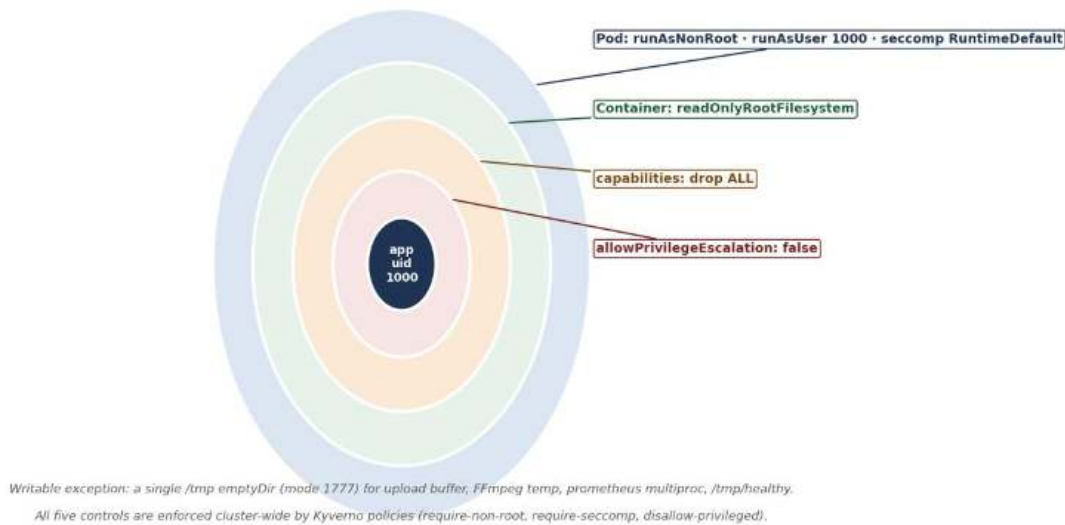


Figure 10.2 — The layered runtime hardening applied to every VidCast pod. A compromised process is non-root, on an immutable filesystem, with no Linux capabilities and no path to escalate.

Control	Where set	Effect if a process is compromised
<code>runAsNonRoot: true / USER 1000</code>	pod spec + Dockerfile	no root in the container; a root escape can't become node root
<code>runAsUser: 1000</code>	pod spec	unprivileged uid; ports are >1024 so no privileged bind needed
<code>seccompProfile: RuntimeDefault</code>	pod-level	blocks ~44 dangerous syscalls (PSS Restricted)
<code>readOnlyRootFilesystem: true</code>	container	binaries/config immutable — no dropping a payload on disk
<code>capabilities: drop: ["ALL"]</code>	container	no raw sockets, no ptrace, no privileged ops
<code>allowPrivilegeEscalation: false</code>	container	setuid/setgid can't gain new privileges

Table 10.3 — The hardened security context (k8s/base/*/deployment.yaml + Dockerfile USER).

The example below is the gateway's context — pod-level identity/seccomp plus container-level filesystem/capability lockdown — and is representative of all six services:

```
# pod-level
securityContext:
  runAsNonRoot: true
  runAsUser: 1000
  seccompProfile: { type: RuntimeDefault }
# container-level
securityContext:
  readOnlyRootFilesystem: true
  allowPrivilegeEscalation: false
  capabilities: { drop: ["ALL"] }
```

The one writable exception

A read-only root filesystem would break any process that needs scratch space, so each pod mounts exactly one writable `emptyDir` at `/tmp` and nothing else. This single exception serves every legitimate write the services make:

- **Gateway** — Werkzeug buffers multipart `/upload` bodies to a temp dir (without it, `/upload` fails with "No usable temporary directory found"), and the 2 Gunicorn workers write Prometheus multiprocess sample files to `/tmp/prometheus`.
- **Converter** — FFmpeg/MoviePy stage temp video/MP3 files in `/tmp`.
- **All consumers** — the `/tmp/healthy` liveness heartbeat (Chapter 8 §8.5) is touched here.

Belt and braces. Setting `USER 1000` in the Dockerfile *and* `runAsNonRoot` in the pod spec is intentional redundancy. The Dockerfile makes the image safe by default anywhere it runs; the pod spec makes Kubernetes refuse to start it as root regardless of the image. A subtle knock-on is visible in the build: Gunicorn 26.x's control socket defaults to `$HOME/.gunicorn/gunicorn.ctl`, which cannot be created under a non-root UID on a read-only filesystem — so the CMD passes `--no-control-socket`. The hardening shaped the runtime flags.

Critically, these are not just conventions — they are **enforced cluster-wide by Kyverno** policies (`require-non-root`, `require-seccomp-runtime-default`, `disallow-privileged`, `require-requests-limits`, `disallow-latest-tag`), so a pod that omitted any of them would be flagged at admission.

10.3 Dependency Pinning and CVE Hygiene (`requirements.txt` / `package.json`)

VidCast pins dependencies to **minimum CVE-free floors** rather than freezing an exact lockfile, and — distinctively — every non-obvious pin carries an inline comment explaining the CVE or compatibility reason. This makes the security intent auditable directly from the manifest. The strategy lets pip resolve patched transitive dependencies (Jinja2, MarkupSafe, click, idna, ...) up from the declared floors, while the CI Trivy gate (Chapter 11) is the backstop that fails the build on any remaining fixable CRITICAL/HIGH.

Representative pins and their reasons

Package	Floor	Documented reason
werkzeug	>=3.0.3	CVE-2024-34069 (debugger RCE, HIGH) fixed only in 3.0.3; forces Flask 3
flask	>=3.0.3	required by Werkzeug 3; only basic APIs used (stable 2.x→3.x)
urllib3	>=2.6.0	1.26.x still carries 4 fixable HIGH CVEs (e.g. CVE-2025-66418); only the 2.x line is patched
gunicorn	>=26.0.0	clears CVE-2024-1135 / CVE-2024-6827 (smuggling); 26.x adds <code>--no-control-socket</code>
bcrypt	>=4.1.2	constant-time <code>checkpw/hashpw</code> (replaces plaintext compare)
Pillow	>=10.3.0	CVE-2023-44271 / CVE-2023-50447 (CRITICAL)
numpy	>=1.26.0,<2.0	moviepy 1.0.3 is incompatible with the numpy 2.x API
setuptools/wheel	pip --upgrade	CVE-2026-24049 (wheel), CVE-2026-23949 (jaraco.context in setuptools)

Table 10.4 — Selected dependency floors with the CVE/compatibility reason recorded in the manifests.

Minimising the dependency surface

Pinning is paired with **removal**. The old frozen requirement lists carried dev-only tooling never imported at runtime (`pylint`, `astroid`, `jedi`, `isort`, `dill`, ...) plus genuinely unused declarations — the gateway had even declared `prometheus-client` while never importing it, and `cryptography` was dropped from auth because JWTs are signed with HS256 (stdlib `hmac`), not RS256. Every dependency that remains is one the service actually imports, which shrinks both the image and the CVE surface. The OS layer is patched too: each Dockerfile runs `apt-get upgrade` to pull fixed `libgnutls30/libkrb5*` packages the base image predates, and `pip install --upgrade setuptools wheel` to clear toolchain CVEs.

Frontend dependencies

`package.json` keeps the runtime dependency set small — `react`, `react-dom`, `react-router-dom`, `axios` — with the build tooling (`vite`, `tailwindcss`, `postcss`, `autoprefixer`, `@vitejs/plugin-react`) confined to `devDependencies`. Because the multi-stage build discards stage 1, **none of those build/dev packages reach the runtime image** — the shipped nginx container contains only the compiled static bundle, so the frontend's runtime CVE surface is essentially the nginx base image itself.

10.4 SHA-Pinned Image References in Kustomize Overlays

How images are *referenced* matters as much as how they are built. VidCast never deploys a floating `:latest` tag; every image is referenced by an **immutable, commit-SHA-pinned tag** set in the Kustomize overlay, and those overlay entries are the GitOps source of truth that Argo CD reconciles (Chapter 12). The CI pipeline tags each build with the 7-character git commit SHA (e.g. `65f2f57`), and the overlay's `images:` block pins exactly that tag.

```
# k8s/overlays/dev/kustomization.yaml
images:
- name: johnbaabalola/auth-service
  newTag: 65f2f57
- name: johnbaabalola/gateway-service
  newTag: 65f2f57
# ... converter, notification, outbox-relay (all same CI SHA)
- name: vidcast-frontend
  newName: 501562869470.dkr.ecr.eu-west-2.amazonaws.com/vidcast-frontend
  newTag: 6eb217c
```

Two properties make a commit-SHA tag effectively a content pin:

- **Immutability at the registry.** The ECR repository is created with `image_tag_mutability = "IMMUTABLE"` (Chapter 9 §9.6), so a pushed tag can never be overwritten — a tag that resolved to one digest yesterday resolves to the same digest forever. The Docker Hub backends rely on the convention that a commit-SHA tag is built once and never re-pushed.
- **Traceability.** The tag *is* the git commit, so any running pod maps directly back to the exact source that built it — invaluable for incident forensics and rollback.

The overlays also show how environments diverge cleanly: the frontend is rewritten (`newName`) to this account's **ECR** registry (CI builds the frontend image to ECR), while the Python backends resolve from **Docker Hub**; and the dev and prod overlays can carry *different* frontend tags (dev `6eb217c`, prod `d9e4282` at the time of writing) so an image can be validated in dev before prod is bumped.

Why tags, not yet digests — and the roadmap

The pins here are immutable `*tags*`, not `@sha256:` digest references. Combined with ECR tag-immutability and the `disallow-latest-tag` Kyverno policy, this already gives reproducible, non-floating deployments. The project's documented next step closes the remaining gap with cryptographic identity rather than naming convention: CI **Cosign** keyless signing after the Trivy gate, and the Kyverno `verify-images` policy (currently in Audit) flipped to **Enforce** so only CI-signed images can run. That moves the guarantee from "this tag is immutable" to "this digest was signed by our pipeline."

GitOps tie-in. Because the tag lives in Git, a deploy is a commit and a rollback is a `git revert` — there is no `kubectl set image` against prod. The CD pipeline bumps `newTag` via a pull request and Argo CD reconciles the cluster to match (Chapter 12). Image referencing, supply-chain hardening, and GitOps are one continuous chain.

In summary, VidCast's container strategy is security-by-construction at every layer: minimal images (multi-stage for the frontend, lean single-stage for the services, with per-image toolchain trimming); a five-control hardened runtime (non-root UID 1000, read-only rootfs, dropped capabilities, no privilege escalation, seccomp) set in both the Dockerfile and the pod spec and enforced by Kyverno; CVE-free dependency floors with the reasoning recorded inline and unused packages removed; and immutable commit-SHA image references driven through GitOps, with Cosign digest-signing as the documented next step. The images are small, attributable, hard to exploit, and reproducible — the unit of deployment matching the rigour of the platform that runs them.

Chapter 11 — CI/CD Pipelines

VidCast's path from a git push to a running pod is automated, security-gated, and credential-free. The repository ships **two complementary pipeline systems**: a primary, cloud-native **GitHub Actions** pair (CI then CD) that lints, builds, scans, pushes, and deploys via keyless OIDC; and an alternative **Jenkins** declarative pipeline that mirrors those stages while adding a Docker Swarm staging deploy, a smoke test, a manual production approval, and automatic rollback — the pattern an enterprise team would recognise. This chapter walks both as they exist in `.github/workflows/{ci,cd}.yaml` and `Jenkinsfile`, the shift-left Trivy gate they share, the registry trust chain and tagging convention, and how the two systems compare.

Concern	GitHub Actions (primary)	Jenkins (alternative)
Lint	<code>ruff</code> (lint job)	<code>ruff</code> (Lint stage)
Build	matrix ×5 services	parallel ×4 services
Scan	Trivy CRITICAL/HIGH gate	Trivy CRITICAL/HIGH gate
Registry	Docker Hub (+ ECR frontend)	Docker Hub
Deploy	OIDC → <code>kubectl set image</code> (EKS)	Swarm staging → manual gate → EKS
Approval	none (auto on CI success)	<code>input</code> manual gate
Rollback	—	<code>post.failure</code> → <code>rollout undo</code>

Table 11.1 — The two pipeline systems at a glance (`.github/workflows/`, `Jenkinsfile`).

11.1 GitHub Actions CI — Ruff Lint → Docker Build (matrix) → Trivy Scan → Push (commit-SHA tag)

The CI workflow (`ci.yml`, "VidCast CI — Lint, Scan, Build, Push") runs on every `push` and `pull_request` to `main` that touches `src/**`. The path filter means a docs-only change never spins up a build. It has two jobs with an explicit dependency:

Job 1 — lint

A single job on `ubuntu-latest`: checkout, set up Python 3.10, `pip install ruff`, then `ruff check src/ --exclude src/frontend` (the frontend is JavaScript, so it is excluded from the Python linter). A lint failure stops the pipeline before any image is built — the cheapest possible failure.

Job 2 — build-and-scan (matrix × 5)

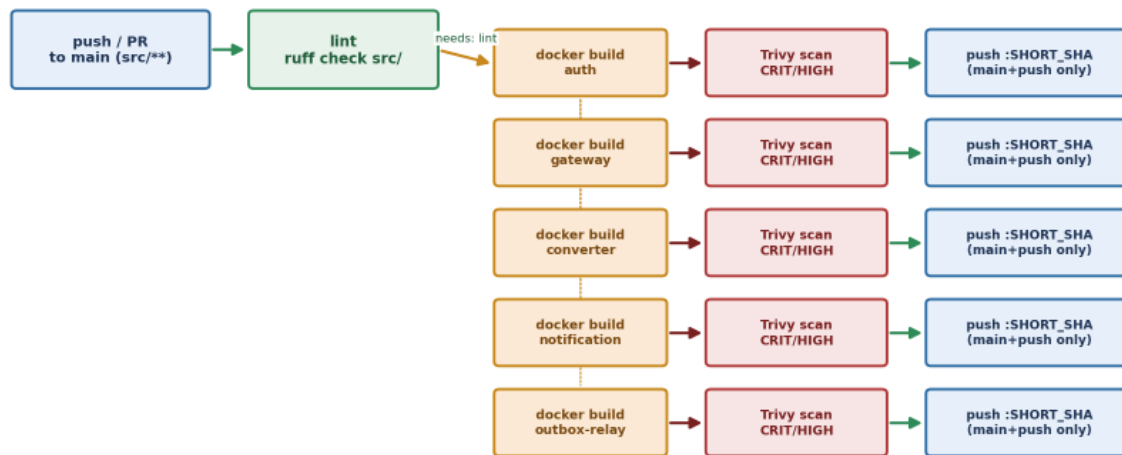
Gated on `needs: lint`, this job fans out across a **matrix of all five Python services** with `fail-fast: false` so one service's failure does not cancel the others' feedback:

```
strategy:
  fail-fast: false
  matrix:
    service: [auth-service, gateway-service, converter-service,
              notification-service, outbox-relay]
```

Each matrix leg runs the same sequence: checkout → compute `SHORT_SHA=${GITHUB_SHA:0:7}` → `docker build` the service into `DOCKERHUB_USERNAME/<service>:<SHORT_SHA>` → **Trivy scan** (§11.2) → conditional login and push. The login and push steps are guarded so images are only published from the real branch on a real merge, never from a pull request:

```
if: github.ref == 'refs/heads/main' && github.event_name == 'push'
```

Figure 11.1 — GitHub Actions CI: lint → build×5 → Trivy gate → push
build-and-scan (matrix × 5, fail-fast: false)



Trivy exit-code 1 on a fixable CRITICAL/HIGH → job fails → image is NEVER pushed

tag = 7-char commit SHA · frontend built separately to ECR (not in this matrix)

Figure 11.1 — The CI pipeline. Lint gates the matrix; each of the five services builds, is Trivy-scanned, and only then (on a main-branch push) is pushed with its commit-SHA tag.

On a pull request the workflow still builds and scans every image — giving the reviewer a full security signal — but stops short of publishing, so unreviewed code can never reach the registry. The frontend image is deliberately **not** in this matrix; it is built separately to ECR (Chapter 9 §9.6), keeping the Python CI lean.

11.2 Shift-Left Trivy Gate (Fixable CRITICAL/HIGH CVE Fails the Build)

The security keystone of CI is the Trivy scan that runs **after build but before push** on every image:

```
- name: Trivy vulnerability scan
  uses: aquasecurity/trivy-action@master
  with:
    image-ref: ${ secrets.DOCKERHUB_USERNAME }/${ matrix.service }:${ env.SHORT_SHA }
    severity: CRITICAL,HIGH
    exit-code: '1'      # a finding fails the job
    ignore-unfixed: true # only gate on CVEs that have a fix
    format: table
```

Setting	Value	Effect
severity	CRITICAL,HIGH	only the two most serious classes gate the build
exit-code	1	any matching finding makes the step (and job) fail
ignore-unfixed	true	CVEs with no available patch don't block (no un-actionable failures)
placement	after build, before push	the failure happens before the image can ship

Table 11.2 — The Trivy gate configuration (ci.yml). The combination is a true shift-left control.

This is "shift-left" in the literal sense: the vulnerability check sits at the earliest practical point in the lifecycle, so a fixable CRITICAL/HIGH CVE **fails the build and the image is never pushed** — it cannot reach the registry, let alone the cluster. `ignore-unfixed: true` keeps the gate actionable: it only fails on vulnerabilities a developer can actually do something about (a patched version exists), avoiding the alert fatigue of un-fixable noise that teams learn to ignore.

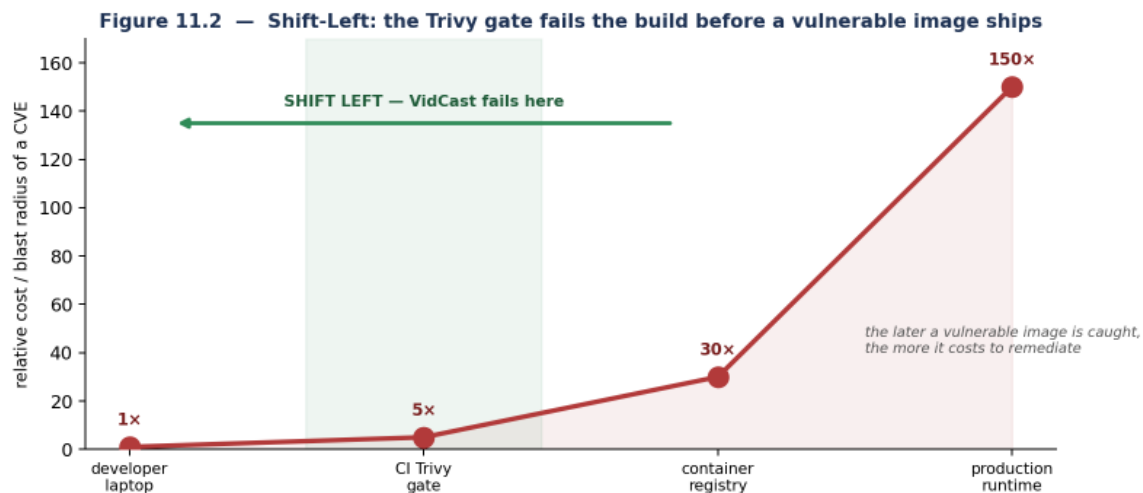


Figure 11.2 — Why the gate's placement matters. The cost and blast radius of a vulnerable image rise sharply the later it is caught; VidCast fails at the CI gate, before the registry or runtime.

Defence in depth. The CI Trivy gate is the primary control, but it is not the only one. ECR repositories are configured with `scan_on_push = true` (Chapter 9), so images that do reach a registry are scanned again — and the Jenkins pipeline (§11.4) runs the identical Trivy gate. The same `--severity CRITICAL,HIGH --exit-code 1 --ignore-unfixed` policy is applied consistently across both CI systems, so the security bar does not depend on which pipeline ran.

11.3 GitHub Actions CD — OIDC to AWS, kubeconfig, kubectl set image, Rollout Verification

The CD workflow (`cd.yml`, "VidCast CD — Deploy to EKS") is chained to CI via a `workflow_run` trigger — it fires when the CI workflow **completes on `main`**, and runs only if CI succeeded:

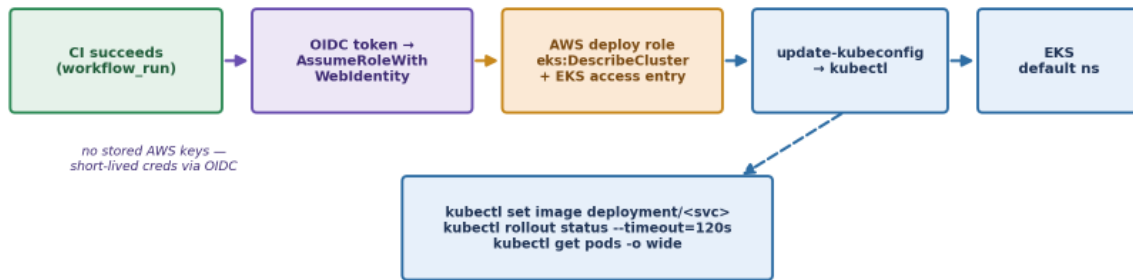
```
on:
  workflow_run:
    workflows: ["VidCast CI - Lint, Scan, Build, Push"]
    types: [completed]
    branches: [main]
jobs:
  deploy:
    if: ${ github.event.workflow_run.conclusion == 'success' }
```

Keyless authentication via OIDC

The decisive security property of CD is that **it stores no AWS credentials**. The workflow requests an OIDC token (`permissions: id-token: write`) and exchanges it for short-lived AWS credentials with `aws-actions/configure-aws-credentials@v4`, assuming the `AWS_DEPLOY_ROLE_ARN` deploy role created by the `github-oidc` Terraform module (Chapter 9 §9.10):

```
- name: Configure AWS credentials (OIDC)
  uses: aws-actions/configure-aws-credentials@v4
  with:
    role-to-assume: ${ secrets.AWS_DEPLOY_ROLE_ARN }
    aws-region: ${ secrets.AWS_REGION }
```

Figure 11.3 — GitHub Actions CD: keyless OIDC deploy to EKS (push model)



Deploys auth · gateway · converter · notification. Documented migration → Argo CD pull model (Ch. 12).

Figure 11.3 — The CD deploy flow. A per-run OIDC token is exchanged for temporary credentials on the scoped deploy role, which can describe the cluster; an EKS access entry grants the kubectl-level rights.

Deploy and verify

With a kubeconfig in hand (`aws eks update-kubeconfig`), CD derives the same `SHORT_SHA` from the triggering run's `head_sha` and rolls each backend forward, then verifies:

```
for svc in auth-service gateway-service converter-service notification-service; do
  deploy_name="${svc%-service}"
  kubectl set image deployment/${deploy_name} \
    ${deploy_name}=${DOCKERHUB_USERNAME}/${svc}:${SHORT_SHA} || true
  kubectl rollout status deployment/${deploy_name} --timeout=120s || true
done
kubectl get pods -o wide          # verify
```

Recall from Chapter 9 that an IAM role alone only yields a kubeconfig — the deploy role's policy is just `eks:DescribeCluster`; the actual `kubectl set image` rights come from a separate **EKS access entry** mapping the role to `AmazonEKSEditPolicy`. This two-layer authorization is what makes the deploy least-privilege.

Honest status — push model, migrating to pull. This committed `cd.yml` is the **push model**: it imperatively runs `kubectl set image` against the live cluster (deploying auth, gateway, converter, notification). It has two acknowledged rough edges — the `|| true` after each command swallows a failed rollout (so a broken deploy can pass silently), and the deploy loop omits `outbox-relay` even though CI now builds it. As documented in `GITOPS.md` §6 and Chapter 12, the designed end-state replaces this with the **Argo CD pull model**: CD bumps the image tag in the overlay and Argo reconciles, removing the AWS credentials from CD entirely and making every deploy a reviewable git diff.

11.4 Jenkins Pipeline — Swarm Staging Deploy, Smoke Test, Manual Production Approval

The `Jenkinsfile` is a declarative pipeline that mirrors CI/CD for teams that run Jenkins, and adds the pre-production verification and human gate an enterprise change process expects. It defines its config in an

`environment` block — Docker Hub and AWS credentials, the cluster (`vidcast-cluster`) and region (`eu-west-2`), the staging IP, and a build tag — and runs nine stages plus a `post` block.

Figure 11.4 — Jenkins pipeline: build → scan → Swarm staging → manual gate → prod

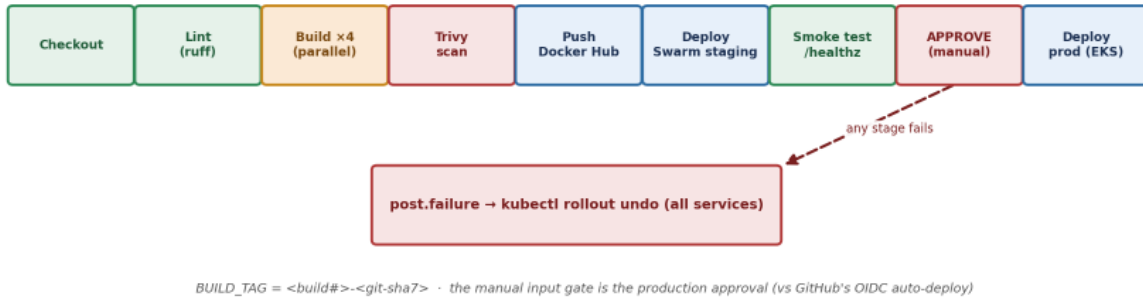


Figure 11.4 — The Jenkins pipeline: build and scan, deploy to a Swarm staging environment, smoke-test it, pause for a human approval, then deploy to EKS — with automatic rollback if any stage fails.

Stage	What it does
Checkout	<code>git clone of main</code>
Lint	<code>ruff check src/ --exclude src/frontend</code>
Build Images	<code>**parallel** build of auth, gateway, converter, notification → vidcast/<svc>:<BUILD_TAG></code>
Security Scan	<code>trivy image --severity CRITICAL,HIGH --exit-code 1 --ignore-unfixed per service</code>
Push Images	<code>docker login</code> then push the four images to Docker Hub
Deploy Staging (Swarm)	<code>ssh to the staging host → docker stack deploy -c docker-compose.swarm.yml</code>
Smoke Test Staging	<code>curl -f http://<staging>:8080/healthz</code> (fails the build on non-200)
Approve Production	<code>input</code> — a human clicks "Deploy to Production"
Deploy Production (EKS)	<code>kubectl set image + rollout status</code> for the four services

Table 11.3 — The nine Jenkins stages (Jenkinsfile).

Two things distinguish Jenkins from the GitHub path. First, it deploys to a **Docker Swarm staging environment on a cheap EC2 instance** and runs a real HTTP smoke test there before prod is ever offered — genuine pre-production verification. Second, the `Approve Production` stage is an explicit **manual `input` gate**: prod only happens when a human approves after seeing staging pass. Finally, the `post.failure` block provides **automatic rollback** — on any failure it runs `kubectl rollout undo` for all four services and logs the rollback, so a bad deploy self-heals to the previous revision.

11.5 Docker Hub / ECR Trust Chain and Image Tagging Convention

The registry trust chain

Images flow through two registries with two distinct auth mechanisms, and pods pull them with a third — none involving a long-lived key on the deploy path:

Hop	Mechanism	Credential
CI/Jenkins → Docker Hub (push)	<code>docker/login-action</code> / <code>docker login</code>	<code>DOCKERHUB_USERNAME</code> + <code>DOCKERHUB_TOKEN</code> (scoped token, not a password)
CI → ECR (frontend)	registry login during build	GitHub OIDC → AWS (no static key)
CD → EKS (deploy)	OIDC federation	<code>AssumeRoleWithWebIdentity</code> → short-lived creds
Node → registry (pull)	node IAM / public pull	<code>AmazonEC2ContainerRegistryReadOnly</code> on the node role (ECR)

Table 11.4 — The image trust chain. The deploy path is keyless; registry push uses scoped tokens; ECR is hardened with immutable tags + scan-on-push.

ECR repositories are created **immutable** with scan-on-push (Chapter 9 §9.6), so a verified digest can never be silently overwritten under the same tag — the registry itself enforces a property the tagging convention relies on.

Image tagging convention

Neither pipeline ever uses a floating `:latest` tag (also forbidden by the Kyverno `disallow-latest-tag` policy). Each system pins to an immutable, traceable tag:

Pipeline	Tag format	Example	Traces to
GitHub Actions	7-char commit SHA	<code>auth-service:65f2f57</code>	exact git commit
Jenkins	<code><build#>-<git-sha7></code>	<code>vidcast/auth:142-65f2f57</code>	Jenkins build + git commit

Table 11.5 — Image tagging conventions. Both are immutable and map a running pod back to its source.

The GitHub convention (`SHORT_SHA = ${GITHUB_SHA: :7}`) is the same tag the Kustomize overlays pin in their `images:` block (Chapter 10 §10.4), so CI output, GitOps source of truth, and the running pod all reference the same commit. The Jenkins convention additionally embeds the build number, which is useful for correlating a deployment with a specific Jenkins run. In both cases the tag *is* an audit record: given a running pod, you can find the exact commit that produced its image.

11.6 Dev vs Prod — Two Pipeline Systems Compared

VidCast deliberately carries two CI/CD systems that reach the same EKS cluster by different routes. They are not redundant by accident — they represent two valid deployment philosophies, and the project documents a clear recommendation between them.

Dimension	GitHub Actions	Jenkins
Role	primary / cloud-native	alternative / enterprise pattern
Hosting	managed runners (no infra)	self-hosted controller (infra to run)
AWS auth	keyless OIDC (no stored keys)	stored <code>aws-credentials</code>
Build fan-out	matrix ×5 (incl. outbox-relay)	parallel ×4

Pre-prod stage	none	Swarm staging + smoke test
Prod approval	automatic on CI success	manual input gate
Rollback	(via Argo / git in target model)	post.failure → rollout undo
Best for	solo / fast iteration	teams needing a staged gate + audit

Table 11.6 — The two systems compared across the dimensions that matter for delivery.

The trade-off and the recommendation

The Jenkins pipeline's Swarm smoke-test and manual gate are genuinely valuable — they catch problems in a real environment before production and give a deliberate human checkpoint. But for a **solo portfolio project**, **GITOPS.md** concludes that maintaining two systems is not worth the marginal safety: the recommendation (Option A) is to standardise on GitHub Actions and let **Argo CD own deployment and rollback** (Chapter 12), with the PR diff serving as the approval and audit trail. If Jenkins were kept, it would be demoted to "smoke-test, then open the deploy PR" (Option B) — its **kubect1**/rollback stages removed, because Argo would own deploy and rollback.

How dev and prod actually differ in the target model. In the GitOps end-state the **pipeline** is the same for both environments — the difference is in **how the deploy is applied**, not which CI ran. CI builds and scans once; the dev overlay tag is bumped and Argo auto-syncs (fast loop), while the prod overlay tag bump goes through a reviewed PR and a manual Argo sync (the approval gate). The "two pipeline systems" of this chapter converge on one CI feeding a GitOps deploy with per-environment sync policy (Chapter 12).

In summary, VidCast automates delivery with a security gate at the front and keyless credentials throughout. GitHub Actions CI lints, then builds all five services in a matrix, Trivy-scans each, and pushes commit-SHA-tagged images only from a real merge — with a fixable CRITICAL/HIGH CVE failing the build before anything ships. CD deploys to EKS via OIDC federation with no stored AWS keys, behind a two-layer IAM + EKS-access-entry authorization. A parallel Jenkins pipeline mirrors the build-and-scan stages and adds Swarm staging, a smoke test, a manual approval, and automatic rollback for teams that want them. The registry trust chain is token- and OIDC-based with immutable, commit-traceable tags, and the project's documented direction consolidates on GitHub Actions feeding the Argo CD pull model — making every deploy a scanned, signed-off, reversible git change.

Chapter 12 — GitOps with Argo CD

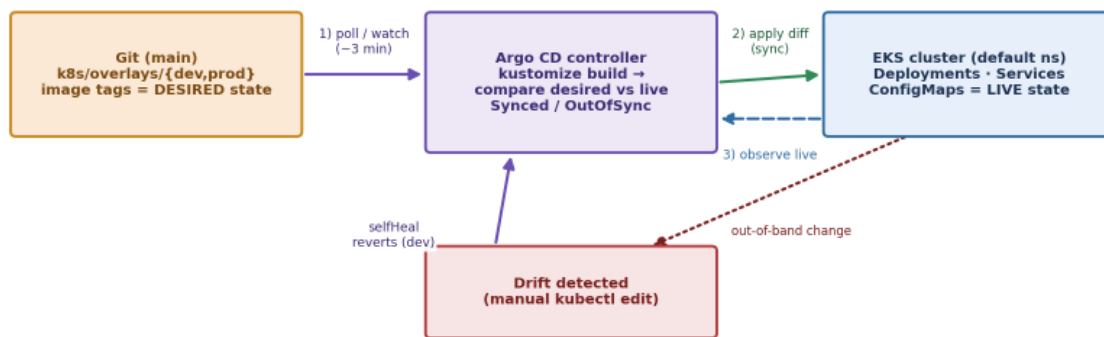
Chapters 9–11 established that VidCast's infrastructure, manifests and images are all declarative. GitOps is the discipline that closes the loop: an in-cluster controller — **Argo CD** — continuously makes the running cluster match what is in Git, so a deployment becomes a commit and the repository, not the cluster, is the source of truth. This chapter documents the GitOps model exactly as the repository implements it: the two **Application** objects under **k8s/argocd/**, the Helm/Kustomize manifest split they reconcile, the deliberately different sync behaviour of dev and prod, the autoscaler field carve-outs, and the rollback and "don't touch the cluster" rules that make the model safe.

Implementation status. B1 ships the GitOps machinery — the two **Application** CRDs, the tuned Argo Helm values, and the **GITOPS.md** contract. The CD tag-bump flow (§12.6) is documented as the operator’s next step; the current **cd.yml** still uses the push model. This chapter describes the designed pull-model target and flags where the repo is mid-migration.

12.1 Git as the Single Source of Truth; Continuous Reconciliation

Argo CD runs in-cluster and continuously reconciles the **default** namespace to the Kustomize manifests in this repo under **k8s/overlays/{dev,prod}**. The model in one sentence (from **GITOPS.md**): **nobody runs **kubectl apply** or **kubectl set image** against the app anymore — you change Git, and Argo makes the cluster match.** The cluster is **not authoritative; the repository is**.

Figure 12.1 — Continuous Reconciliation: Git is the single source of truth



The cluster is never authoritative — the repository is. To change the cluster, change Git.

Argo manages app workloads only; platform (Argo itself, KEDA, ESO, Kyverno, NetworkPolicies) is operator-applied.

Figure 12.1 — The reconciliation loop. Argo polls Git (desired state), compares it to the live cluster, and applies the diff. Out-of-band drift is detected and, in dev, reverted by selfHeal.

How a reconcile works

10. **Observe desired state.** Argo runs **kustomize build** on the overlay path (**k8s/overlays/dev** or **/prod**) — the same command validated locally; there is no Argo-specific manifest format.
11. **Observe live state.** It reads the actual Deployments, Services, ConfigMaps (and ESO-created Secrets) in the cluster.
12. **Diff and report.** It compares the two and marks the Application **Synced** or **OutOfSync**, with per-resource health.
13. **Reconcile.** Depending on the sync policy (§12.3–12.4), it applies the diff — automatically for dev, on a human action for prod. **ApplyOutOfSyncOnly=true** means only changed resources are re-applied, not the whole tree.

What Argo manages — and what it does not

A core decision is the boundary between the **application layer** (Argo-managed) and the **platform layer** (operator-managed). Argo manages only the app workloads in the overlays; the control planes that make the

cluster what it is — including Argo's own install — are applied manually by the platform engineer. This avoids a chicken-and-egg (Argo managing its own installation) and keeps infrequent, privileged platform changes out of the app delivery loop.

Layer	Owner	How it is applied
App workloads (Deployments, Services, ConfigMaps, ESO Secrets in <code>overlays/*</code>)	Argo CD	synced from Git
Argo CD itself	platform	<code>helm install argocd</code>
ESO (<code>ClusterSecretStore</code> , <code>ExternalSecrets</code>)	platform	<code>kubectl apply -f k8s/external-secrets</code>
KEDA (<code>ScaledObject</code> , <code>TriggerAuthentication</code>)	platform	<code>kubectl apply -k k8s/keda</code>
NetworkPolicies	platform	<code>kubectl apply -k k8s/network-policies</code>
Kyverno + ClusterPolicies	platform	<code>kubectl apply -k k8s/kyverno</code>

Table 12.1 — The manage-vs-manual boundary (`docs/GITOPS.md §4`).

12.2 Manifest Strategy — Helm for State, Kustomize (base + overlays) for Apps

VidCast uses two manifest tools for two different jobs, a deliberate division rather than a single-tool dogma:

Tool	Used for	Why
Helm	Stateful third-party services: MongoDB, PostgreSQL, RabbitMQ (Chapter 7); plus Argo CD, KEDA, ESO, Kyverno, kube-prometheus-stack	upstream charts, lifecycle hooks, value templating for off-the-shelf software the project does not author
Kustomize	First-party app workloads: auth, gateway, converter, notification, outbox-relay, frontend, redis, backup	plain YAML bases + thin per-environment overlays; no templating language; what Argo reconciles

Table 12.2 — The manifest split. Helm for state and platform; Kustomize for the apps Argo manages.

The base + overlay layout Argo reads

The app manifests follow a Kustomize base/overlay structure. Each workload has one base under `k8s/base/<svc>/`; each environment has a thin overlay that references those bases and patches the few things that differ (replica counts, image tags, governance labels):

```
k8s/
  base/<svc>/          # one base per workload (Deployment, Service, ConfigMap)
  overlays/
    dev/    → Application vidcast-dev (auto-sync)  1 replica each
    prod/   → Application vidcast-prod (manual-sync) live footprint
```

The Argo `Applications` point at these overlay paths in **this same repo** — a documented "Q3" decision to use in-repo manifests rather than a separate deployment repository:

- **A separate manifest repo** is the textbook multi-team pattern — it decouples "who can change code" from "who can change what's deployed" and lets many app repos feed one deploy repo.

- **A single repo** is the right call for a solo project: one PR captures both the code change *and* the manifest/image-tag change, with one review and one audit trail. A second repo would add ceremony with no separation-of-duties benefit when one person owns everything.

12.3 vidcast-dev — Auto-Sync, Self-Heal, Drift Correction

The **vidcast-dev** Application is the fast validation loop. It points at `k8s/overlays/dev` on `main` and carries a full **automated** sync policy, so every change to the dev overlay (or an image-tag bump) is reconciled automatically within the controller's poll interval (~3 minutes):

```
# k8s/argocd/application-dev.yaml
spec:
  source:
    repoURL: https://github.com/johnnybabs/vidcast.git
    targetRevision: main
    path: k8s/overlays/dev
  syncPolicy:
    automated:
      prune: true      # delete resources removed from Git
      selfHeal: true   # revert manual kubectl drift back to Git
    syncOptions:
      - CreateNamespace=false
      - ApplyOutOfSyncOnly=true
```

Option	Effect
<code>automated</code>	reconcile without a human; the dev loop is hands-off
<code>prune: true</code>	a resource deleted from Git is deleted from the cluster (no orphans)
<code>selfHeal: true</code>	out-of-band <code>kubectl edit</code> drift is automatically reverted to match Git
<code>ApplyOutOfSyncOnly=true</code>	only changed resources are re-applied each cycle
<code>finalizers: resources-finalizer.argocd.argoproj.io</code>	deleting the Application cascades to prune its synced resources

Table 12.3 — vidcast-dev sync options (`k8s/argocd/application-dev.yaml`).

The dev workflow is therefore: **merge to main → CI builds the image → the dev overlay tag is bumped → Argo auto-syncs within ~3 min.** `selfHeal` is what makes the cluster genuinely converge on Git — a developer who hand-edits a Deployment to debug will see Argo quietly put it back, reinforcing that Git is the only durable way to change anything.

12.4 vidcast-prod — Manual Sync as a Human Approval Gate

The **vidcast-prod** Application is identical in structure but **deliberately omits the `automated` block**. That absence *is* the approval gate — a production deploy happens only when a human acts:

```
# k8s/argocd/application-prod.yaml
source:
  path: k8s/overlays/prod
syncPolicy:
  # PROD = MANUAL. There is deliberately NO automated: block.
  syncOptions:
    - CreateNamespace=false
```

```
- ApplyOutOfSyncOnly=true
```

Because there is no `automated/selfHeal`, a merged change to the prod overlay leaves the Application showing **OutOfSync** until a human runs `argocd app sync vidcast-prod` (or clicks Sync in the UI). The two-step gives a clean, auditable two-person-rule-friendly flow: the **PR merge is the approval**, and the **manual sync is the deploy action**.

Figure 12.2 — Promotion: dev auto-syncs · prod is a manual approval gate

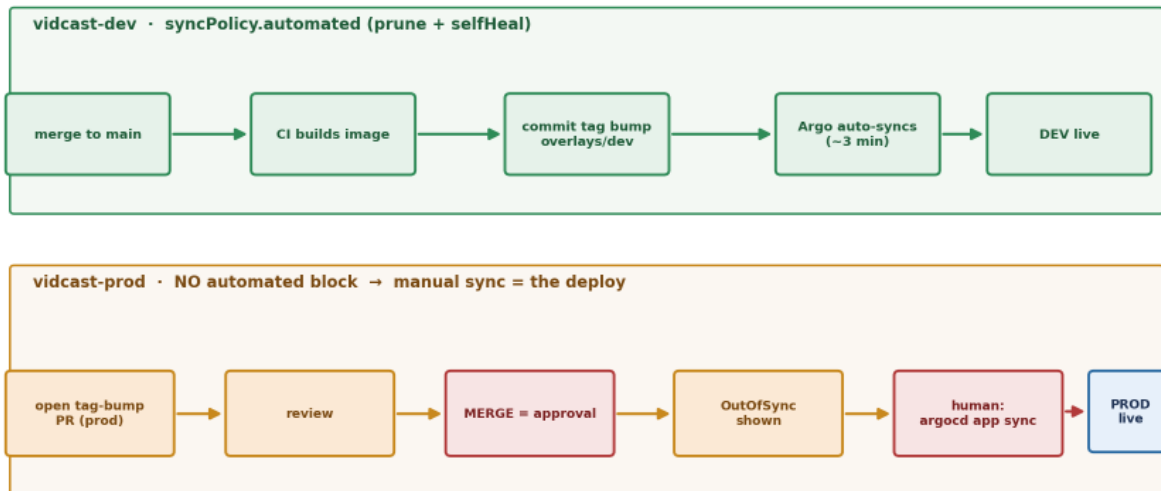


Figure 12.2 — Promotion contrast. dev (top) flows hands-off to live; prod (bottom) requires a reviewed PR merge as approval and an explicit manual sync as the deploy.

Aspect	vidcast-dev	vidcast-prod
<code>syncPolicy.automated</code>	present (<code>prune</code> , <code>selfHeal</code>)	absent (manual only)
Trigger	every change to <code>overlays/dev</code> , auto	a human runs <code>argocd app sync</code>
Drift (<code>kubectl edit</code>)	auto-reverted (<code>selfHeal</code>)	shown as <code>OutOfSync</code> until a human acts
Purpose	fast validation loop	the production approval gate

Table 12.4 — dev vs prod sync behaviour (`docs/GITOPS.md §5`).

Single-cluster caveat. On this demo cluster both Applications target the **same** `default` namespace, so they manage the same-named resources and must not be synced simultaneously — you run one at a time (dev for validation, prod for the live footprint). In a real deployment the two Applications point at different clusters via `destination.server`; this is the one place the demo topology diverges from production GitOps, and it is documented explicitly in the manifest and `GITOPS.md`.

12.5 Replica Ownership Carve-Outs (Autoscaler-Owned /spec/replicas)

GitOps and autoscaling appear to conflict: Argo wants the cluster to match the overlay (`replicas: 1`), while KEDA scales the converter (0→2, scale-to-zero) and the HPA scales the gateway at runtime. Without a carve-

out, Argo would forever see live replicas $\neq$ overlay replicas, report `OutOfSync`, and — worse, with `selfHeal` — actively fight the autoscaler, scaling pods back down moments after they scaled up.

The fix is `ignoreDifferences`: the dev Application tells Argo to **ignore the `/spec/replicas` field** on the two autoscaled Deployments, so the autoscaler owns that one field and Argo owns everything else:

```
# k8s/argocd/application-dev.yaml
ignoreDifferences:
  - group: apps
    kind: Deployment
    name: converter      # KEDA owns its replica count (0..2)
    jsonPointers: [ /spec/replicas ]
  - group: apps
    kind: Deployment
    name: gateway        # HPA owns its replica count
    jsonPointers: [ /spec/replicas ]
```

This is field-level ownership: the overlay's `replicas` value becomes merely the **bootstrap count** applied at first creation, after which KEDA/HPA take over and Argo no longer reconciles it. Every other field of those Deployments (image, env, probes, security context, resources) remains fully Git-owned. The `GITOPS.md` rule states it plainly: the converter's replica count is owned by KEDA at runtime, so the overlay value is just the bootstrap, and Argo won't fight KEDA *as long as nobody also sets it by hand*.

Deployment	Replica owner	Argo behaviour on <code>/spec/replicas</code>
converter	KEDA (<code>ScaledObject</code> , 0→2, scale-to-zero)	ignored (<code>ignoreDifferences</code>)
gateway	HPA (CPU-based)	ignored (<code>ignoreDifferences</code>)
auth, notification, frontend, ...	Git (overlay <code>replicas</code> .)	reconciled normally

Table 12.5 — Replica field ownership. Only the two autoscaled Deployments are carved out.

12.6 Rollback via git revert; The "No kubectl edit to Prod" Rule

Because the deployed state is fully described in Git, both rollback and change control reduce to Git operations.

Rollback is a git revert

To undo a bad deploy, you `git revert` the offending commit — typically the image-tag bump — and Argo reconciles back to the previous state: automatically for dev, on the next manual sync for prod.

```
git revert <bad-commit>      # the image-tag bump (or any manifest change)
# dev:  Argo auto-syncs back to the prior state
# prod: argocd app sync vidcast-prod
```

The key property is that rollback is now a **versioned, auditable Git operation**, not an invisible `kubectl rollout undo`. `git log` shows exactly what was rolled back, by whom, and when — permanently. This is the same audit trail the deploy itself carries: under the pull model, a deploy is a PR diff with a reviewer, so you can see precisely which image SHA went to prod, who approved it, and when, forever (§12.6 pull-model migration in `GITOPS.md` §6).

The one rule: don't kubectl edit synced resources

Once Argo owns a resource, **Git is the only legitimate way to change it**. A manual `kubectl edit/kubectl apply` on a synced workload is treated as drift:

- In **dev**, `selfHeal` silently reverts the change back to Git within a poll interval — the manual edit simply vanishes.
- In **prod**, the change shows as **OutOfSync** drift and stays visible until a human either reconciles to Git or codifies the change properly in a commit.

This rule extends to the autoscaled Deployments: KEDA/HPA own `/spec/replicas` (\$12.5), so you must not set replicas by hand either — to change anything, change Git. The practical effect is that the cluster has no "pet" state: every running configuration is traceable to a commit, drift cannot accumulate unnoticed, and the temptation to "just quickly fix it in the cluster" is structurally removed.

The push → pull migration (designed)

The GitOps end-state replaces the legacy push model — where `cd.yml` ran `kubectl set image` straight against EKS, leaving the record of what's deployed only in the cluster — with a pull model where CI builds and pushes the image, a CD job bumps the overlay tag (committing to main for dev, opening a PR for prod), and Argo deploys. The recommended Option A keeps everything in GitHub: the **PR diff is the audit and the approval**, matching the in-repo single-repo decision, and the CD job no longer needs AWS/EKS credentials at all because Argo — not the workflow — performs the deploy.

In summary, VidCast's GitOps layer makes the repository the single source of truth and Argo CD the engine that continuously reconciles the cluster to it. The manifest split keeps Helm for stateful/platform software and Kustomize bases+overlays for the app workloads Argo manages; `vidcast-dev` auto-syncs and self-heals for a fast loop, while `vidcast-prod` deliberately omits automation so a human merge-then-sync forms the approval gate; `ignoreDifferences` carve-outs let KEDA and the HPA own replica counts without Argo fighting them; and rollback becomes a `git revert` while the "don't `kubectl edit`" rule guarantees the cluster never drifts from Git. The result is deployments that are auditable, reversible, and reproducible — the natural completion of the all-declarative platform built in the preceding chapters.

Chapter 13 — Autoscaling

VidCast's two compute tiers have opposite load profiles, so they are scaled by two different controllers on two different signals. The **converter** is an asynchronous, CPU-heavy, bursty queue consumer that is idle most of the time — it is scaled by **KEDA** on RabbitMQ queue depth, all the way down to zero replicas when there is no work. The **gateway** is the synchronous, always-on, user-facing request tier — it is scaled by a built-in **Horizontal Pod Autoscaler (HPA)** on CPU utilisation. This chapter documents both as they exist in `k8s/keda/`, the authentication and node-budget constraints that bound them, and why the two controllers can never fight.

Tier	Controller	Signal	Range (committed)	Why this signal
Converter	KEDA <code>ScaledObject</code>	RabbitMQ <code>video</code> queue depth	0 → 2	async, bursty, idle-most-of-the-time → scale to zero
gateway	HPA (<code>autoscaling/v2</code>)	CPU utilisation (70% of request)	1 → 3	synchronous user tier → must

Table 13.1 — The two autoscalers, their signals, and their committed replica ranges (k8s/keda/).

A note on the numbers. This chapter’s heading frames the ranges illustratively (“0→N”, “2→5”). The values committed in the repository are tuned down for the single 2-vCPU demo node: the converter `ScaledObject` caps at `maxReplicaCount: 2` and the gateway HPA at `minReplicas: 1`, `maxReplicas: 3`. The figures and tables below use the **committed** values; §13.3 explains the node-budget reason the caps are what they are, and how they would rise on a multi-node cluster.

13.1 KEDA on the Converter — Scale 0→N on RabbitMQ Queue Depth, Scale-to-Zero after Idle

The converter does the heavy lifting — pulling a video out of GridFS and running FFmpeg — but only when there is something in the queue. Provisioning it for peak and leaving it running would burn CPU (and, on a metered node, money) doing nothing for the ~100+ idle hours a week the deck cites. KEDA solves this by scaling the converter Deployment on the depth of the RabbitMQ `video` queue, **including down to zero**:

```
# k8s/keda/scaledobject-converter.yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
spec:
  scaleTargetRef:
    name: converter          # the converter Deployment ONLY
  pollingInterval: 15        # check the queue every 15s
  cooldownPeriod: 60         # wait 60s after drain before scaling down
  minReplicaCount: 0         # scale to ZERO when there is no work
  maxReplicaCount: 2         # single-node cap (see §13.3)
  triggers:
  - type: rabbitmq
    metadata:
      protocol: amqp
      queueName: video        # MAIN queue depth (not retry/dlq)
      mode: QueueLength
      value: "5"              # ~5 queued messages per replica
    authenticationRef:
      name: keda-rabbitmq-auth
```

Field	Value	Meaning
<code>minReplicaCount</code>	0	no work → zero pods → zero idle CPU burn (the whole point)
<code>maxReplicaCount</code>	2	upper bound on the 2-vCPU node (§13.3)
<code>pollingInterval</code>	15 s	how often KEDA samples the queue
<code>cooldownPeriod</code>	60 s	grace period after the queue drains before scaling down
<code>queueName / mode</code>	<code>video / QueueLength</code>	scale on the main queue’s length — not the retry/DLQ queues
<code>Value</code>	5	target of ~5 queued messages per converter replica

Table 13.2 — `ScaledObject` parameters (k8s/keda/scaledobject-converter.yaml).

The desired replica count is essentially `ceil(queueLength / value)`, clamped to `[0, 2]`. With `value: "5"`, an empty queue yields 0 replicas; a burst of ~10+ messages drives it to the cap of 2. Scaling on the **main `video` queue only** (not `video.retry/video.dlg`) is deliberate — retrying or dead-lettered messages are not pending work that more converters would clear.

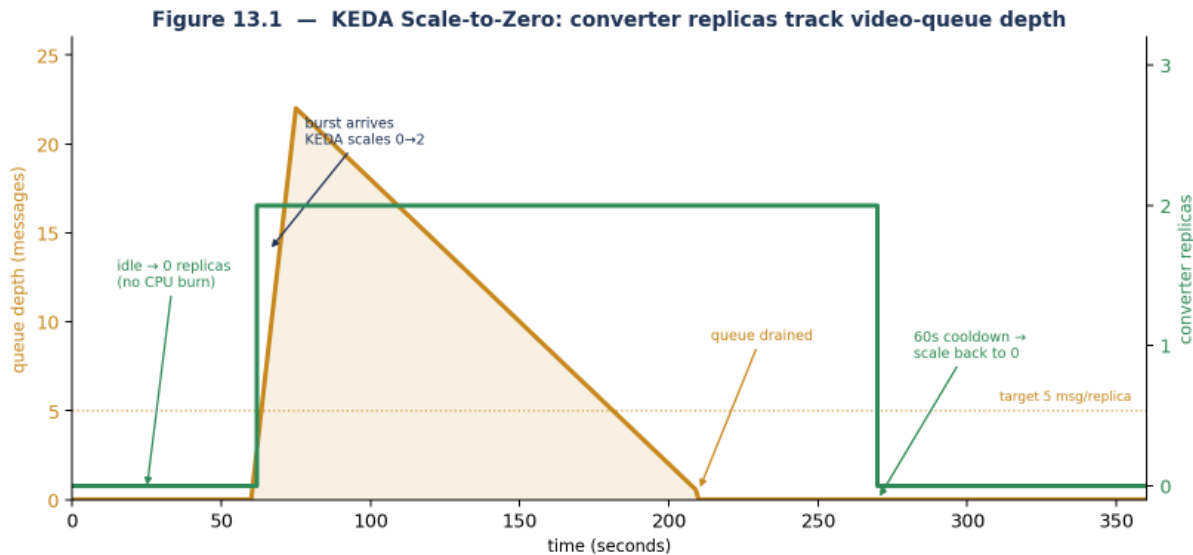


Figure 13.1 — KEDA scale-to-zero in action. The converter sits at 0 replicas while idle, scales up when a burst lands on the video queue, and — 60 s after the queue drains (`cooldownPeriod`) — scales back to 0.

From 0→1 KEDA uses its built-in activation path (the queue becoming non-empty "activates" the `ScaledObject`); from 1→N it hands off to a managed HPA it creates under the hood. The `cooldownPeriod` of 60 s prevents flapping: a brief lull does not immediately tear pods down, but a genuinely empty queue scales the converter to zero a minute later, reclaiming the node's most expensive request.

13.2 HPA on the Gateway — 2→5 Replicas on CPU Utilisation

The gateway is the opposite kind of workload: it terminates user requests synchronously, must always have at least one replica ready, and its load correlates with CPU, not a queue. It is therefore scaled by a plain built-in `HorizontalPodAutoscaler` — no KEDA dependency — on CPU utilisation as a percentage of the gateway's CPU request (100m):

```
# k8s/keda/hpa-gateway.yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: gateway          # the gateway Deployment ONLY
  minReplicas: 1
  maxReplicas: 3
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
```

```
type: Utilization
averageUtilization: 70
```

The HPA keeps average CPU across gateway pods at ~70% of the 100m request, scaling between 1 and 3 replicas. The committed range is `minReplicas: 1` (always-on; the user-facing tier never goes to zero) and `maxReplicas: 3` (the single-node ceiling). Conceptually this is the "2→5 on CPU" pattern the section title describes; the committed numbers are the node-sized version of it.

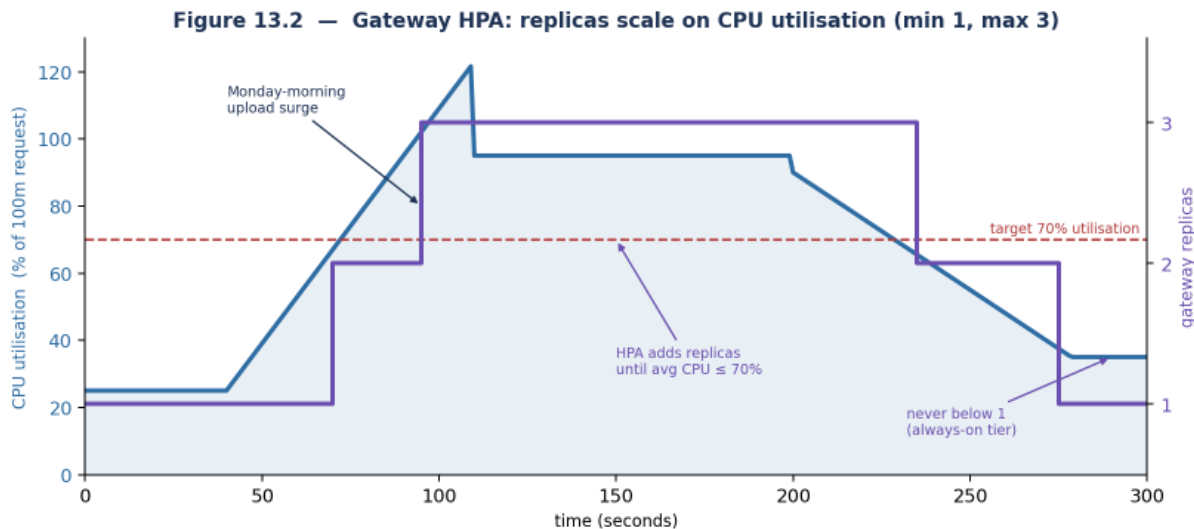


Figure 13.2 — The gateway HPA. As a load surge pushes CPU above the 70% target, the HPA adds replicas until average utilisation falls back to target; it never scales below 1.

Prerequisites

CPU-utilisation targeting has two hard requirements, both documented in the KEDA README and satisfied in the manifests:

- **metrics-server must be installed.** EKS does not bundle it; without it the HPA reports `<unknown>` CPU and will not scale.
- **The gateway must declare a CPU request.** Utilisation is a percentage *of the request* — the gateway requests `100m` (limit `300m`), so 70% utilisation is ~70m of sustained CPU per pod before a scale-up is triggered.

13.3 TriggerAuthentication, Node CPU Budget and maxReplicas Capping

TriggerAuthentication — how KEDA reaches the broker

KEDA's RabbitMQ scaler needs a full AMQP connection string (it embeds credentials), so the credential cannot live in plain `ScaledObject` metadata. A dedicated `TriggerAuthentication` maps the scaler's `host` parameter to the `host` key of a `keda-rabbitmq-secret`:

```
# k8s/keda/triggerauthentication.yaml
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata: { name: keda-rabbitmq-auth }
spec:
  secretTargetRef:
```

```
- parameter: host
  name: keda-rabbitmq-secret
  key: host          # amqp://<user>:<pass>@rabbitmq:5672/
```

The stock `rabbitmq-secret` only holds `RABBITMQ_DEFAULT_USER/PASS`, not a combined URI, which is why this dedicated secret exists. The secret itself is **gitignored** (`secret.yaml.example` is the template) and, under the External Secrets Operator (A9), becomes an `ExternalSecret` pulling `/vidcast/<env>/keda/rabbitmq-host` from SSM Parameter Store — so even the broker URI stays out of Git, consistent with the platform’s zero-secrets-in-Git posture.

Node CPU budget — why the caps are 2 and 3

The replica caps are not arbitrary; they are dictated by what a single **m7i-flex.large (2 vCPU / ~2000m allocatable)** node can actually schedule once every pod’s resource `*requests*` are scheduler-visible (a Sprint-4 gap-fix that made datastore requests count). The node already carries the app tier, the self-hosted datastores, Redis, the relay, and the platform add-ons (KEDA, Argo, Kyverno, monitoring). The converter is the single most expensive workload at **250m CPU request** each:

Workload (CPU request)	×	≈ CPU req
Converter	1 (idle) → 2 (peak)	250m → 500m
Gateway	1 → 3	100m → 300m
auth / notification / frontend / relay / redis	1 each	~300m total
datastores (Mongo + Postgres + RabbitMQ)	1 each	~300m total
platform (KEDA ~125m, Argo ~250m, monitoring, Kyverno)	—	~600m+
m7i-flex.large allocatable	—	~2000m (2 vCPU)

Table 13.3 — Approximate CPU-request budget on the single node. The converter’s 250m request is why a third replica will not schedule.

The `ScaledObject` comment states it plainly: the 2-vCPU node cannot schedule three converter replicas at any resource level once datastore requests are scheduler-visible — so `maxReplicaCount: 2`. Likewise the gateway HPA caps at 3. These caps are a **node constraint, not a design ceiling**: horizontal node scaling (a Cluster Autoscaler or a larger/multi-node group) in production would raise both. KEDA itself is installed with deliberately conservative requests (~125m / 160Mi across its operator, metrics server and webhook) for the same reason — it must not crowd out the workloads it scales.

13.4 Avoiding KEDA/HPA Conflict (Right Signal per Workload)

Running KEDA and an HPA in the same cluster is a classic footgun: if two controllers target the **same** Deployment, they continuously overwrite each other’s desired replica count and the pod count oscillates. VidCast avoids this structurally — the two controllers target **different Deployments on different signals**, so there is no shared `scaleTargetRef` to fight over.

Figure 13.3 — One signal, one controller, one target — no conflict

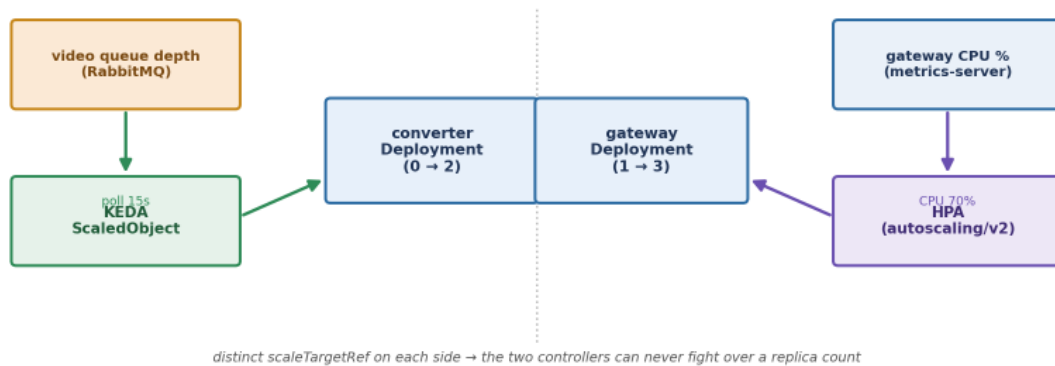


Figure 13.3 — Separation by construction. KEDA scales the converter on queue depth; the HPA scales the gateway on CPU. Distinct targets mean the controllers' writes never collide.

	Converter	Gateway
Controller	KEDA <code>ScaledObject</code>	HPA <code>autoscaling/v2</code>
<code>scaleTargetRef</code>	<code>converter</code>	<code>gateway</code>
Signal	<code>video</code> queue length	CPU utilisation 70%
Min replicas	0 (scale-to-zero)	1 (always-on)
Max replicas	2	3
Rationale	async, bursty, idle-heavy	sync, user-facing, load-tracking

Table 13.4 — The two autoscalers never share a target — the root of the no-conflict guarantee.

Choosing the **right signal per workload** is the deeper point. Scaling the async converter on CPU would be wrong — a converter waiting on FFmpeg or idle between bursts shows low CPU even when a backlog is building, so CPU is a lagging, misleading signal for queue work; queue depth is the leading indicator that directly measures pending work. Conversely, scaling the synchronous gateway on queue depth is meaningless — it has no queue; its pressure *is* CPU under concurrent requests. Matching signal to workload is what lets each tier scale correctly and independently.

Interaction with GitOps

There is one more place a conflict could arise: Argo CD also wants to reconcile replica counts to the Git overlay (`replicas: 1`). As covered in Chapter 12 §12.5, the Argo `Application` carves out `/spec/replicas` for both the converter and gateway via `ignoreDifferences`, so the autoscalers own that field at runtime while Git owns everything else. The overlay's `replicas:` is therefore only the **pre-autoscaler bootstrap value** — re-applying the overlay may briefly reset it until KEDA/HPA reconcile, after which the controllers take over. Autoscaling and GitOps coexist cleanly because field ownership is explicit on all three controllers.

In summary, VidCast scales each tier with the controller and signal that fit its load shape: KEDA drives the bursty converter on RabbitMQ `video`-queue depth with true scale-to-zero (0→2, 60 s cooldown), reclaiming the node's most expensive request whenever the pipeline is idle; a built-in HPA keeps the always-on gateway

between 1 and 3 replicas at 70% CPU. A `TriggerAuthentication` keeps KEDA's broker URI out of Git, the replica caps are honestly bounded by the single 2-vCPU node's budget (and would rise with node scaling), and — because the two controllers target distinct Deployments on distinct signals, with Argo carving out `/spec/replicas` — the autoscalers, the HPA, and GitOps coexist without ever fighting over a replica count.

Part IV — Security: Defence in Depth (Sec)

Chapter 14 — Security Model and Threat Landscape

Security is the “Sec” in DevSecOps, and in VidCast it is not a single feature but a layered model spanning the whole platform — from the public edge down to the kernel syscalls a pod may make. This chapter frames that model: it states what the platform is protecting and against whom (§14.1), presents the six-layer defence-in-depth architecture that the rest of Part IV details layer by layer (§14.2), and articulates the blast-radius containment philosophy that ties the layers together (§14.3). The controls referenced here are documented in full in Chapters 15–19; this chapter is the map.

A useful anchor is that VidCast’s security posture is **evidence-based, not aspirational**. An independent baseline analysis of the project ([docs/TECHNICAL_ANALYSIS.md](#)) recorded a concrete set of findings — public-facing datastores, password-less PostgreSQL, secrets in the tree, unauthenticated service-to-service calls — and the security work documented across Part IV is the systematic closure of those findings into the layered model below. Where a layer’s control maps to a specific finding it closed, this chapter names it.

14.1 Threat Model and Security Objectives

What is being protected (assets)

VidCast handles client media on behalf of a media company, so the assets are concrete and the stakes are real — the original business case (Chapter 2) was partly about **removing** the compliance exposure of client content sitting on unknown third-party sites:

- **Client content** — uploaded videos and converted MP3s in MongoDB GridFS (confidential client deliverables).
- **User identities and credentials** — accounts, bcrypt password hashes and roles in PostgreSQL.
- **Secrets** — database URIs, the JWT signing key, SMTP credentials, broker credentials.
- **The platform itself** — the EKS cluster, the CI/CD deploy path, and the AWS account behind it.

Who and what we defend against (threat actors)

Actor	Capability / motive	Primary concern
External attacker	Internet-facing scanning, exploitation of any exposed endpoint	reaching a service or datastore from outside
Malicious uploader / authed user	A valid account abused for escalation or abuse	privilege escalation, rate-abuse, accessing others’ files
Compromised workload	A pod exploited via app/dependency CVE	lateral movement, credential theft, node escape
Supply-chain threat	A vulnerable or tampered image/dependency	shipping a known-bad or unsigned artefact
Accidental insider	Operator error / leaked credential	secrets in Git, over-broad permissions

Table 14.1 — Threat actors VidCast’s security model is designed to resist.

Security objectives

The objectives map cleanly onto the classic confidentiality–integrity–availability triad, plus the operational property of auditability:

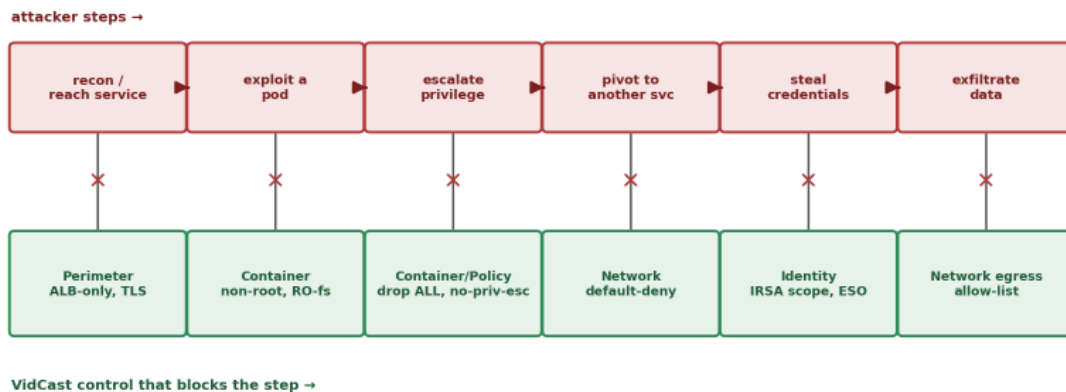
Objective	What it means for VidCast	Where met (chapter)
Confidentiality	client media and credentials never leave the company AWS account; TLS in transit; encryption at rest	Perimeter (15), Identity (16)
Integrity	only reviewed, scanned, correctly-tagged images run; no privilege escalation	Supply chain (18), Policy (19), Identity (16)
Availability	abuse (brute-force, upload flooding) is rate-limited; a compromise is contained, not total	Network (15), Container (17)
Least privilege	every identity (human, workload, pipeline) has only the access it needs	Identity/IRSA (16)
Auditability	role changes, downloads, deploys are recorded and traceable	Identity/RBAC (16), GitOps (12)

Table 14.2 — Security objectives and where each is delivered in Part IV.

Threats modelled as an attacker kill-chain

Rather than a control checklist, the model is framed as the sequence of steps an attacker would have to complete — and the independent layer that blocks each step. No single control is assumed perfect; the design assumes any one may fail and ensures another stands behind it.

Figure 14.2 — Attacker Kill-Chain vs the Defence Layers



No single control is assumed perfect — each step has an independent layer that must also be defeated.

Figure 14.1 — The attacker kill-chain (top) and the VidCast control that blocks each step (bottom). Reaching the data requires defeating every layer in sequence.

14.2 The Six-Layer Defence-in-Depth Model

VidCast's security is organised into **six concentric layers** an attacker must defeat in turn to reach the application and its data. The model is deliberately defence-in-depth: each layer is independent, so a weakness or bypass in one does not expose the core — it merely advances the attacker to the next barrier. The deck summarises this as the platform's "six security layers"; this report devotes a chapter to each.

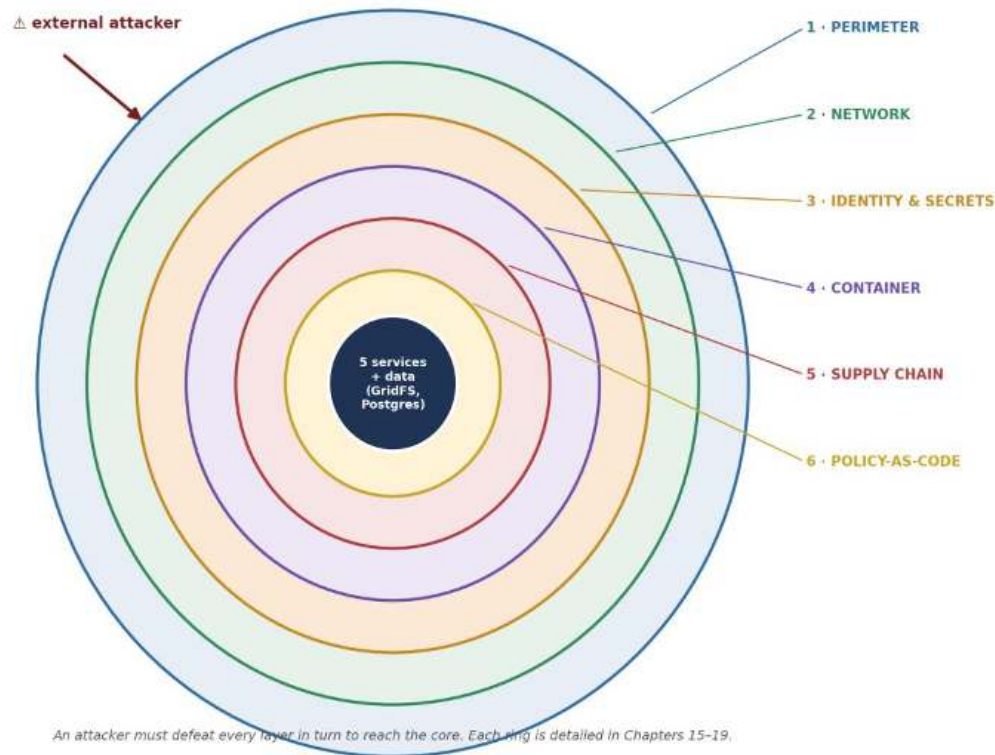
Figure 14.1 — The Six-Layer Defence-in-Depth Model

Figure 14.2 — The six-layer model. From the public edge inward: Perimeter, Network, Identity & Secrets, Container, Supply Chain, and Policy-as-Code, wrapping the five services and their data.

#	Layer	Core controls	Detailed in
1	Perimeter	ALB + ACM TLS only; every service ClusterIP ; no NodePort exposed to the internet in the prod posture	Ch. 15
2	Network	NetworkPolicy default-deny on ingress + egress; explicit pod-to-pod allow-lists; VPC-CNI enforcement agent	Ch. 15
3	Identity & Secrets	bcrypt cost-12, JWT/RBAC + guardrails, IRSA (keyless), External Secrets Operator → 0 secrets in Git	Ch. 16
4	Container	non-root UID 1000, read-only rootfs, drop ALL capabilities, no-priv-escalation, seccomp RuntimeDefault	Ch. 17
5	Supply Chain	Trivy CVE gate (shift-left), SHA/commit-pinned images, ECR immutable + scan-on-push, Cosign (roadmap)	Ch. 18
6	Policy-as-Code	7 Kyverno ClusterPolicies enforcing the above at admission	Ch. 19

Table 14.3 — The six layers, their controls, and where each is detailed (slide “Six Security Layers”).

Why the layers complement each other

The layers are not redundant copies of one control; each addresses a **different** failure mode, which is what makes the depth real:

- **Perimeter** keeps the front door singular and encrypted, so there is one auditable way in (the ALB) rather than seven open NodePorts — directly closing finding **H-1** (databases exposed to the public internet).
- **Network** assumes the perimeter can be breached and limits what a foothold can reach: default-deny means a compromised pod has **no** network path it was not explicitly granted — closing finding **M-5** (unauthenticated, unrestricted service-to-service calls).
- **Identity & Secrets** assumes a pod or pipeline can be reached and minimises what its credentials are worth: scoped IRSA roles, RBAC with escalation guardrails, and secrets pulled from Parameter Store rather than living in Git — closing **H-2/H-3/H-4** (trust-auth Postgres, a leaked Gmail password, no secret management).
- **Container** assumes the application code itself can be exploited and constrains the process: non-root, read-only, capability-stripped, seccomp-filtered.
- **Supply Chain** assumes a dependency or base image can carry a vulnerability and stops it **before** it ships: the Trivy gate fails the build on a fixable CRITICAL/HIGH.
- **Policy-as-Code** assumes any of the above could be forgotten in a new manifest and enforces them centrally: Kyverno rejects (or, today, audits) a pod that violates the baseline.

Defence in depth, stated plainly. Each layer is built on the assumption that the layer in front of it may fail. The perimeter may be misconfigured; an app may have a CVE; a manifest may forget a setting. The model's value is that none of those single failures reaches the assets on its own — an attacker must chain a defeat of every layer.

14.3 Blast-Radius Containment Philosophy

Defence-in-depth answers "how do we keep attackers out?"; blast-radius containment answers the more honest question "**when** something is compromised, how much can it touch?" VidCast is designed around the assumption that a single pod **will** eventually be compromised — via an application bug, a dependency CVE, or a misconfiguration — and that the job of the architecture is to make that event a contained incident rather than a cluster-wide breach.

Figure 14.3 — Blast-Radius Containment: one compromised pod

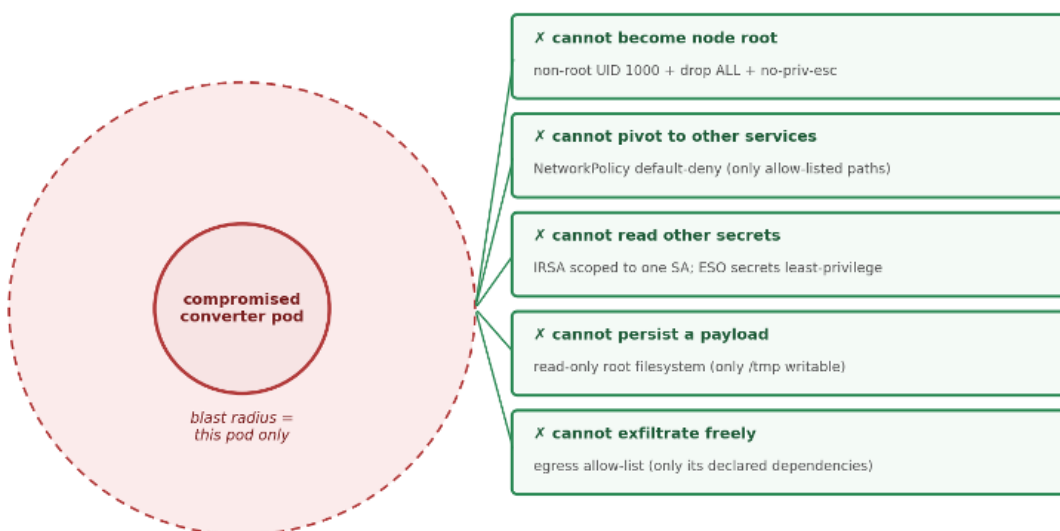


Figure 14.3 — Containment of a single compromised pod. Each control independently denies a path an attacker would otherwise use to widen the breach.

Concretely, a compromised converter pod is hemmed in on every side, and crucially **each barrier comes from a different layer** — so the containment does not depend on any one control:

An attacker in a compromised pod...	...is blocked by	Layer
cannot become root on the node	non-root UID 1000 + drop ALL capabilities + no-priv-escalation + seccomp	Container
cannot pivot to another service	NetworkPolicy default-deny — only its declared paths are reachable	Network
cannot read other services' secrets	IRSA scoped to one ServiceAccount; ESO secrets are least-privilege per service	Identity
cannot drop a persistent payload	read-only root filesystem (only a scoped <code>/tmp</code> emptyDir is writable)	Container
cannot exfiltrate to arbitrary hosts	egress allow-list — outbound is restricted to its declared dependencies	Network
cannot reach datastores from outside	they are <code>ClusterIP</code> , behind default-deny, with no public NodePort	Perimeter/Network

Table 14.4 — Blast-radius containment: how each layer limits what a single compromised pod can do.

Design principles that flow from this philosophy

- **Least privilege by default.** Every identity — a workload's IRSA role, the CD pipeline's deploy role, a user's RBAC role — is scoped to the minimum it needs. A stolen credential is worth only what that one identity could do.
- **Assume breach, segment accordingly.** Default-deny networking and per-pod hardening mean the unit of compromise is a single pod, not the namespace or the node. The zero-trust networking lesson recorded in the project — that it "trades convenience for blast-radius control" — is exactly this principle in practice.
- **No standing, broad credentials.** Keyless OIDC/IRSA and a zero-secrets-in-Git posture mean there is no long-lived master key to steal: credentials are short-lived, scoped, and absent from the source tree.
- **Immutability limits persistence.** Read-only filesystems and immutable image tags mean an attacker cannot easily establish persistence or tamper with what runs — a foothold is transient by construction.

The single-node demo topology is honest about its own limit here: with one node, the node *is* the availability boundary, so node-level isolation between tenants is out of scope (there is one tenant). The containment model is therefore pod-level and credential-level, which is the right granularity for this architecture and scales naturally to per-node isolation on a multi-node cluster.

In summary, VidCast treats security as a layered system rather than a checklist. The threat model names concrete assets (client media, identities, secrets, the platform) and actors (external attackers, abusive users, compromised workloads, supply-chain and insider risks), and sets confidentiality, integrity, availability, least-privilege and auditability as objectives. Those objectives are delivered through six independent, concentric layers — Perimeter, Network, Identity & Secrets, Container, Supply Chain, and Policy-as-Code — each of which closes specific findings from the project's own baseline audit and each of which assumes the others

may fail. Underpinning all of it is a blast-radius philosophy: assume a pod will be compromised, and ensure that when it is, every escalation path it would reach for has already been independently denied. Chapters 15–19 examine each layer in turn.

Chapter 15 — Perimeter and Network Security (Zero-Trust)

This chapter details the first two layers of the defence-in-depth model (Chapter 14): the **perimeter** that controls what the outside world can reach, and the **network** layer inside the cluster that controls what pods can reach each other. Together they implement zero-trust networking — the principle that no traffic is trusted by default, neither from the internet nor between pods, and every permitted flow must be explicitly declared. The controls live in `k8s/ingress/` (the public edge) and `k8s/network-policies/` (the in-cluster policy set), and they are the layers that directly close the project's most serious baseline finding, **H-1** (datastores exposed to the public internet).

15.1 ALB + ACM TLS Only; Every Service ClusterIP; Zero NodePort to the Internet

The perimeter principle is "**one public door.**" External traffic enters exclusively through an AWS Application Load Balancer, provisioned by the AWS Load Balancer Controller from the `vidcast-ingress` Ingress resource. Every application Service is `ClusterIP` — reachable only from inside the cluster — so there is exactly one internet-facing component to secure, monitor, and reason about.

Figure 15.1 — The ALB is the only public door; everything else is ClusterIP

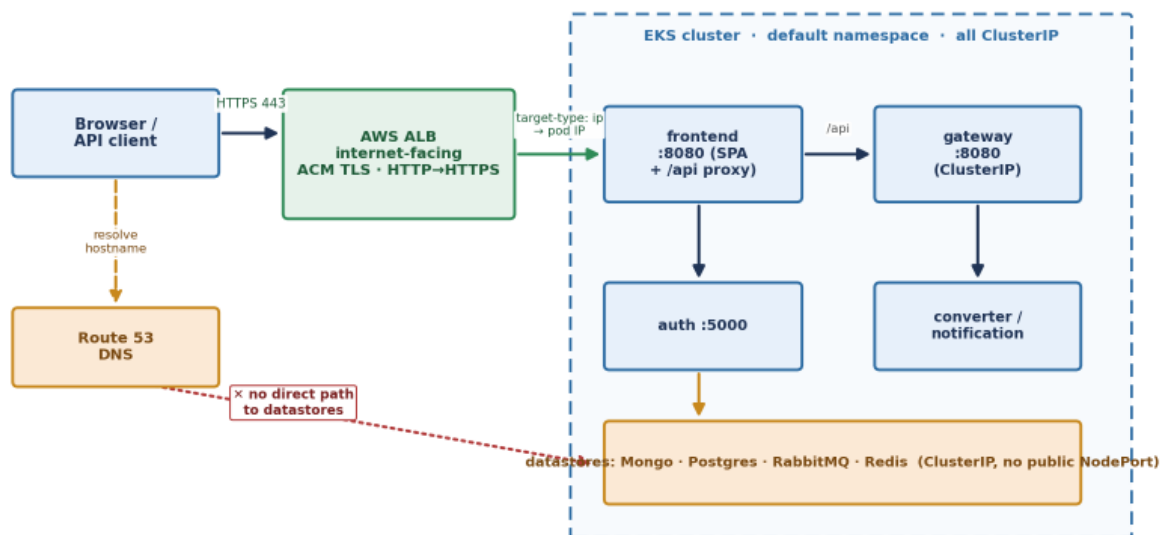


Figure 15.1 — The perimeter. The ALB terminates TLS and is the only internet-facing entry; the frontend, gateway, and all datastores are ClusterIP, with no public path to the stateful tier.

The Ingress and the routing decision

A single Ingress rule sends **all paths to the `frontend` service** (port 8080), not split between frontend and gateway. This is deliberate and documented in the manifest: the frontend's nginx already serves the React SPA *and* proxies `/api/` to `gateway:8080` while stripping the `/api` prefix. An ALB cannot rewrite or strip path prefixes, so routing `/api` straight to the gateway would deliver `/api/login` to a gateway that only knows `/login` (a 404). Routing through the frontend preserves the working path for both browsers and API clients — and keeps the **gateway internal (ClusterIP)**, reachable only via the frontend, shrinking the attack surface.

```
# k8s/ingress/vidcast-ingress.yaml
annotations:
  alb.ingress.kubernetes.io/scheme: internet-facing
  alb.ingress.kubernetes.io/target-type: ip          # pod IPs (works w/ ClusterIP)
  alb.ingress.kubernetes.io/listen-ports: '[{"HTTP":80,"HTTPS":443}]'
  alb.ingress.kubernetes.io/ssl-redirect: "443"
  alb.ingress.kubernetes.io/certificate-arn: "${ACM_CERTIFICATE_ARN}"
spec:
  ingressClassName: alb
  rules:
    - host: "${VIDCAST_HOSTNAME}"
      http: { paths: [ { path: /, backend: { service: frontend:8080 } } ] }
```

`target-type: ip` registers pod IPs directly with the ALB target group, which is what lets a **ClusterIP** Service sit behind an internet-facing load balancer with no NodePort. The Ingress also uses a named ALB **group** (`group.name: vidcast`) so a future host (e.g. Grafana) can share the one ALB rather than each provisioning its own. The ALB controller itself runs in `kube-system` with a single replica and assumes its `lbc-irsa` role (Chapter 9 §9.9) — keyless, no static credentials.

No NodePort to the internet — closing H-1

The Terraform NodePort security group does open ports 30002–30008 (Chapter 9 §9.5) for development/diagnostic access, but in the production posture the stateful tier is **not** internet-reachable: the datastores are **ClusterIP**, and — critically — the network layer (§15.3) does not grant any allow-from-anywhere rule on the DB/broker ports. Once default-deny is in place, the DB/broker NodePorts (30003/30004/30005) are no longer reachable from outside the cluster, and admin access is via `kubectl port-forward` instead. This is the direct remediation of finding **H-1** ("databases exposed to the public internet via NodePort + 0.0.0.0/0").

15.2 Route 53 DNS and HTTP→HTTPS Redirect at the ALB

The public hostname (`${VIDCAST_HOSTNAME}`, e.g. `vidcast.online`) is served by **Route 53**, whose record points at the ALB. TLS is terminated **at the ALB using an ACM certificate** (`certificate-arn`), so encryption is handled at the edge by AWS-managed infrastructure — no certificate material is mounted into a pod.

Concern	Setting / mechanism	Effect
DNS	Route 53 hosted zone → ALB	public hostname resolves to the load balancer
TLS termination	ACM certificate on the ALB	HTTPS at the edge; certs auto-renewed by ACM
Listen ports	<code>[{"HTTP":80,"HTTPS":443}]</code>	ALB listens on both 80 and 443
HTTP→HTTPS	<code>ssl-redirect: "443"</code>	every plaintext request is 301-redirected to HTTPS

Health check	<code>healthcheck-path: /</code>	ALB checks the SPA root for target health
--------------	----------------------------------	---

Table 15.1 — Route 53 + ACM + ALB edge configuration (vidcast-ingress.yaml).

The `ssl-redirect: "443"` annotation makes the ALB redirect all HTTP traffic to HTTPS, so there is no unencrypted path to the application even if a client tries `http://`. On TLS, the project made a clean trade-off: the ALB terminates with ACM directly, so a `cert-manager` `ClusterIssuer` ships only as an alternative path (for in-cluster ingress or DNS-01 issuance), not as the production mechanism — avoiding the complexity of in-cluster certificate secrets when ACM already does the job natively.

15.3 NetworkPolicy Default-Deny on Ingress and Egress

Inside the cluster, the network layer starts from zero trust: a single `default-deny-all` NetworkPolicy selects **every pod** in the `default` namespace and denies **all ingress and all egress**. From that baseline, traffic only flows where an explicit allow policy permits it.

```
# k8s/network-policies/default-deny.yaml
spec:
  podSelector: {}          # every pod in the namespace
  policyTypes: [ Ingress, Egress ]
  # no ingress/egress rules => deny everything not allowed elsewhere
```

Two properties make this safe and operable, both documented in the manifests:

- **NetworkPolicies are additive.** A packet is permitted if *any* policy allows it, so the model is "default-deny + a set of allow exceptions." The allow-*** policies are applied *first*, and `default-deny` is applied **last**, so nothing is cut off before its exception exists.
- **DNS must be allowed before deny.** A separate `allow-dns-egress` policy lets every pod reach CoreDNS (`k8s-app: kube-dns` in `kube-system`) on UDP/TCP 53. Without it, default-deny would break all name resolution — and since services are addressed by DNS name (`rabbitmq`, `mongodb`, `auth`, ...), the whole app would fail. This is the one universal egress exception.

Fast, reversible rollback. Because default-deny is a single object, the most powerful safety control is also the simplest to undo: `kubectl delete networkpolicy default-deny-all -n default` instantly restores open networking. The project documents this as the fastest rollback in the entire plan — a deliberate operability choice for a control that, if misconfigured, could black-hole the app.

15.4 Explicit Pod-to-Pod Allow-Lists (Per-Selector, Per-Port, Both Ends)

Every legitimate flow is declared as a minimal allow rule. The governing principle, stated in `app-policies.yaml`, is that **every A→B flow needs BOTH an egress rule on A and an ingress rule on B** — a packet must be permitted to leave the source *and* permitted to enter the destination, or default-deny drops it.

Figure 15.2 — Default-deny + an explicit allow needs BOTH ends



If either rule is missing, default-deny drops the packet. Every A→B flow = an egress rule on A AND an ingress rule on B.

Figure 15.2 — Both-ends rule. The gateway→auth flow requires an egress rule on the gateway AND a matching ingress rule on auth; either one missing means the packet is denied.

Rules are scoped tightly by **pod label selector and port** — never "allow from the namespace" or "allow all ports." For example, the gateway policy accepts ingress only on 8080 and allows egress only to auth (5000), MongoDB (27017), and RabbitMQ (5672); the queue consumers have **no ingress rule at all** (they only make outbound connections):

Pod	Ingress (accepts)	Egress (may reach)
Gateway	TCP 8080 (any source: NodePort + frontend)	auth:5000, mongo:27017, rabbitmq:5672
Auth	TCP 5000 from gateway	postgres:5432
Frontend	TCP 8080 (any source: NodePort)	gateway:8080
Converter	— (none)	rabbitmq:5672, mongo:27017, redis:6379
Notification	— (none)	rabbitmq:5672, redis:6379, SMTP 0.0.0.0/0:587
outbox-relay	— (none)	mongo:27017, rabbitmq:5672

Table 15.2 — Per-app allow-lists (app-policies.yaml). DNS egress is universal; datastore ingress is in Table 15.3.

The whole permitted surface can be read as a single traffic matrix — and everything *not* in it is denied. This is what gives the blast-radius guarantee from Chapter 14: a compromised pod has no network path it was not explicitly granted.

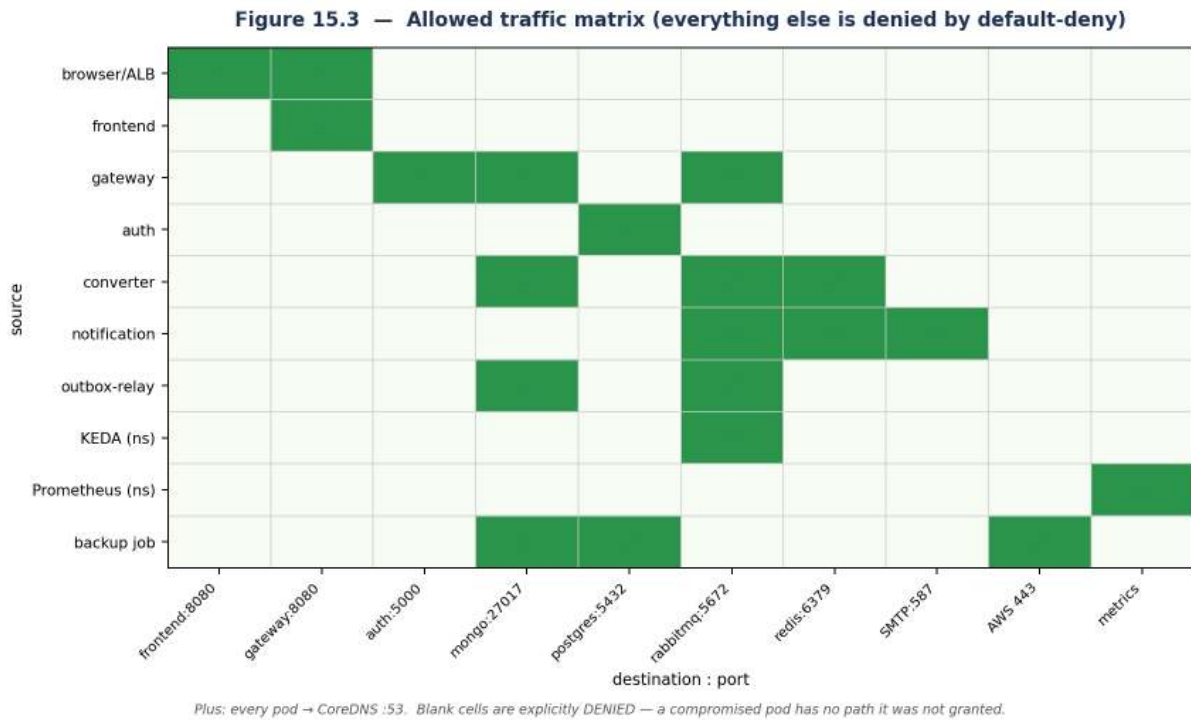


Figure 15.3 — The complete allowed-traffic matrix. Each ✓ is an explicitly declared flow; every blank cell is denied by default-deny. A compromised converter, for instance, can reach only RabbitMQ, Mongo and Redis.

Two egress rules use a CIDR rather than a pod selector, by necessity. The notification service's SMTP egress targets `0.0.0.0/0:587` because Gmail's IPs are dynamic — but, as the comment notes, nothing internal listens on 587, so this opens no in-cluster path. (The same pattern recurs for AWS and Sigstore egress in §15.6.)

15.5 VPC-CNI Native Network-Policy Agent (No Calico/Cilium)

A NetworkPolicy is only as good as the agent that enforces it. VidCast relies on the **VPC CNI's native network-policy agent** — the eBPF enforcer built into the EKS `aws-node` DaemonSet — rather than adding a third-party CNI like Calico or Cilium. This keeps the networking stack to the AWS-supported default with no extra control plane to run on the 2-vCPU node.

The hard prerequisite. The agent is **off by default**, and this is the single most important configuration in the whole layer. Without it, NetworkPolicy objects are accepted by the API server but **never enforced** — they become decorative YAML, and default-deny silently does nothing. It is enabled in Terraform (Chapter 9 §9.4): the `vpc-cni` add-on is installed with `configuration_values = { enableNetworkPolicy = "true" }`, which flips on the eBPF agent. The README mandates confirming it on a live cluster with `kubectl get ds aws-node -n kube-system` before declaring the zero-trust work done.

The honest limitation — no hostname egress pinning

The project is candid about what vanilla Kubernetes NetworkPolicy cannot do: it matches egress by **IP/CIDR**, **not hostname**. So it cannot pin egress to, say, `*.sigstore.dev` — external services on rotating CDN IPs can only be expressed as "TCP 443 to the public internet." True FQDN-level egress pinning would require a DNS-aware CNI (Cilium) or an egress proxy, which is explicitly out of scope and documented as such. This is

recorded as a known limitation rather than hidden — the same intellectual honesty seen throughout the project.

15.6 Datastore, Monitoring, Backup and Sigstore Egress Policies

Beyond the core app flows, four additional policy groups extend the allow-list to the platform’s supporting traffic — each added precisely because default-deny would otherwise break a legitimate function.

Datastore ingress

Each datastore accepts connections **only from the specific clients that need it**, and deliberately has **no** allow-from-anywhere rule on its port — which is what makes the stateful NodePorts unreachable externally:

Datastore (label)	Accepts ingress from	Port
MongoDB (app: database)	gateway, converter, outbox-relay	27017
PostgreSQL (app: auth-app)	auth only	5432
RabbitMQ (app: rabbitmq)	gateway, outbox-relay, converter, notification; KEDA ns; monitoring ns (:15692)	5672 / 15692
Redis (app: redis)	converter, notification	6379

Table 15.3 — Datastore ingress allow-lists (datastore-policies.yaml). Datastores need no egress policy — they never initiate connections.

Monitoring scrape

Prometheus (in the `monitoring` namespace) must reach the app pods’ metrics endpoints, or default-deny would silently break observability (targets go DOWN, and alerts that depend on them can never fire). Two policies grant it: `allow-monitoring-scrape` opens gateway:8080 and auth:5000, and `allow-monitoring-scrape-consumers` opens the converter/notification :9000 metrics ports (added for the B4 SLO work). RabbitMQ’s ingress also admits the monitoring namespace on :15692 for its Prometheus plugin.

Backup egress

The nightly backup CronJobs (selected by `app.kubernetes.io/component: backup`) need both ends opened: egress to Mongo (27017), Postgres (5432), and **AWS over HTTPS** (0.0.0.0/0:443 for STS AssumeRoleWithWebIdentity + S3 PUT), plus matching **ingress** on Mongo and Postgres from the backup pods (additive to the datastore policies). Without these, `mongodump/pg_dump` hang and the `aws-cli` upload CrashLoops with no path to STS/S3.

Sigstore egress (Kyverno namespace)

When the Kyverno `verify-images` policy (Chapter 18) evaluates a pod, the admission controller must reach the OCI registries (Docker Hub, ECR) plus Fulcio, Rekor and the TUF CDN to verify a Cosign signature. The `allow-kyverno-sigstore-egress` policy — uniquely, in the `kyverno` namespace, not `default` — grants DNS, 0.0.0.0/0:443 (the registries + Sigstore, which cannot be hostname-pinned), and IMDS on 169.254.169.254:80 (scoped to the link-local address) so the AWS SDK can fetch node-role credentials to authenticate to the **private** ECR repo. Without the IMDS rule, the ECR pull hangs past the webhook deadline and `failurePolicy: Fail` would reject every ECR-image workload.

Egress policy	Namespace	Allows
app SMTP (notification)	default	0.0.0.0/0:587 (Gmail; nothing internal listens on

		587)
backup-egress	default	mongo:27017, postgres:5432, 0.0.0.0/0:443 (AWS STS+S3)
allow-kyverno-sigstore-egress	kyverno	DNS:53, 0.0.0.0/0:443 (registries+Sigstore), IMDS 169.254.169.254:80

Table 15.4 — The CIDR-based egress exceptions and why each is necessary.

In summary, VidCast’s perimeter and network layers implement zero-trust in both directions. At the edge, a single ACM-terminated ALB — fronted by Route 53 with HTTP→HTTPS redirect — is the only public door; every service is ClusterIP and the stateful tier has no internet path, closing finding H-1. Inside the cluster, a default-deny NetworkPolicy denies all ingress and egress, and a tightly-scoped set of allow-lists re-opens only the declared flows, each requiring a rule on both ends, enforced by the VPC-CNI’s native eBPF agent (the prerequisite without which the policies are inert). Supporting traffic — datastore clients, Prometheus scrapes, nightly backups to AWS, and Kyverno’s Sigstore verification — is admitted by precise, documented exceptions, with the one honest limitation (no hostname-level egress pinning) recorded openly. The net effect is that a compromised pod has exactly the network reach it was granted and not one connection more.

Chapter 16 — Identity, Authentication and Secrets

This chapter covers the third layer of the defence-in-depth model (Chapter 14): identity. It is the layer that answers **who is this, what may they do, and where do the credentials live?** VidCast addresses all three with concrete, code-level controls — bcrypt password hashing (§16.1), JWT-based authentication with role-based access control (§16.2), admin guardrails that prevent privilege-escalation and self-logout (§16.3) — and extends the same least-privilege, no-long-lived-keys philosophy to **machine** identity and secrets: IRSA for workloads (§16.4), the External Secrets Operator pulling from SSM Parameter Store so there are zero secrets in Git (§16.5), and the rotation/refresh model that keeps them current (§16.6).

Two of these directly close findings from the project’s baseline audit (Chapter 14): **H-2** (PostgreSQL ran with **trust** auth — fixed by SCRAM + real bcrypt verification), **H-3/H-4** (a leaked Gmail password and no secret management — fixed by the External Secrets Operator), and the application-level **"everyone was an admin"** privilege-escalation hole (§16.3).

16.1 bcrypt (cost-12) Password Hashing

Passwords are never stored or compared in plaintext. The auth service hashes every new password with **bcrypt at cost factor 12** and verifies logins with bcrypt’s **constant-time** comparison, against the hash stored in the PostgreSQL **auth_user** table (Chapter 7 §7.2).

```
# register: hash a new password (auth-service/server.py)
hashed = bcrypt.hashpw(password.encode("utf-8"),
                        bcrypt.gensalt(rounds=12)).decode("utf-8")

# login: constant-time verification against the stored hash
```

```
password_ok = bcrypt.checkpw(auth.password.encode("utf-8"),
                             password_hash.encode("utf-8"))
```

- **Cost 12** sets the work factor — each hash runs 2^{12} key-expansion rounds, deliberately slow so brute-forcing a stolen hash is computationally expensive, while still fast enough for interactive login.
- **Constant-time** `checkpw` prevents timing side-channels that could leak whether a guess was partially correct.
- **Defensive handling of legacy rows.** `checkpw` raises if the stored value is not a valid bcrypt hash (e.g. a pre-migration plaintext row); the service catches that and treats it as an **auth failure, never a 500** — so the bcrypt migration is forward-only and a stale row can never bypass authentication.

The admin account itself is seeded at deploy time with a bcrypt hash generated *inside* PostgreSQL via `pgcrypto` (Chapter 7), so no plaintext password is ever written to the repo or a config file. This — combined with PostgreSQL's SCRAM-SHA-256 auth (replacing the old `trust` method, finding H-2) — means the credential is genuinely enforced end to end.

16.2 JWT Issuance/Validation and Role-Based Access Control (RBAC)

After a successful login the auth service issues a **JSON Web Token (JWT)** that the client presents on subsequent requests. The token is signed with **HS256** using the `JWT_SECRET` (itself pulled from Parameter Store, §16.5) and expires after **1 day**.

Figure 16.1 — Login → JWT → RBAC enforcement at the gateway

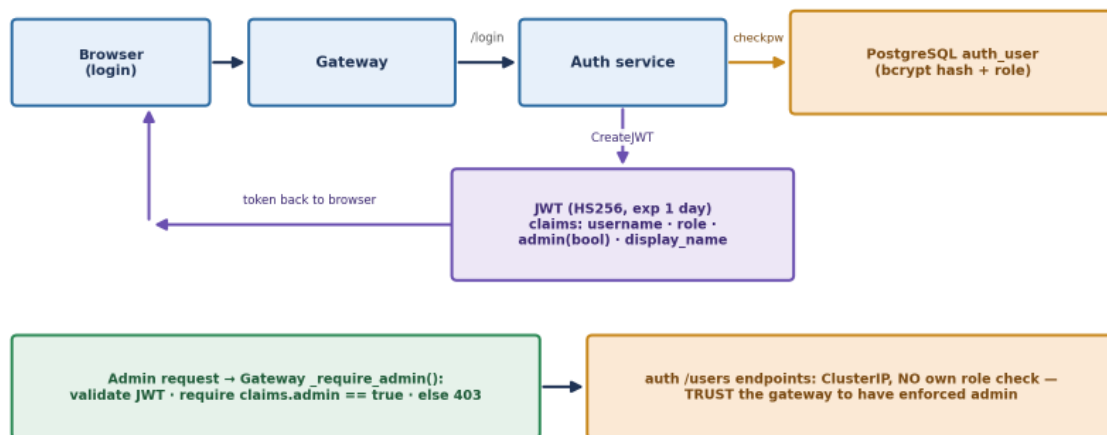


Figure 16.1 — Login → JWT → RBAC. The auth service verifies the bcrypt hash and mints a signed JWT carrying the role; the gateway is where admin authorization is enforced.

The token and its claims

```
# CreateJWT (auth-service/server.py)
jwt.encode({
    "username": username,
    "exp": now + timedelta(days=1),    "iat": now,
    "admin": role == "admin",           # bool — backward-compat
    "role": role,                       # string — forward-compat
    "display_name": username.split("@")[0],
}, secret, algorithm="HS256")
```

The token carries both an `admin` boolean (read by the existing gateway/frontend) and a forward-compatible `role string` — a deliberate design choice so more roles (auditor, support, ...) can be added later without breaking consumers. The `/validate` endpoint decodes and verifies the signature, returning `401 Unauthorized` on any failure.

Where authorization is enforced

A crucial architectural point: the auth service's user-administration endpoints (`/users`, `PATCH /users/<email>`) are **ClusterIP-only and carry no role check of their own** — they trust in-cluster callers, exactly like `/login` and `/validate`. **The gateway is the component that enforces "admin only"** before calling them, via a `_require_admin()` helper that validates the JWT and rejects a non-admin with `403`:

```
# gateway-service/server.py
def _require_admin(request):
    claims, err = validate(request)      # verify JWT via auth /validate
    if not claims or not claims.get("admin"):
        return None, ("admin only", 403)
    return claims, None
```

A documented trust gap. The project is candid that this places the entire admin-authorization burden on the gateway: the auth endpoints would admit any in-cluster caller. This is recorded honestly in `ADMIN_USERS_EXPLAINED.md`, and it is **contained** by the network layer (Chapter 15) — those endpoints are reachable only from the gateway under default-deny, so there is no path for an arbitrary pod to call them. The documented "real fix" is to add a role check at the auth service too (defence in depth).

16.3 Admin Guardrails (Self-Demote 403, Last-Admin 409) and the "Everyone Was an Admin" Fix

Role management is where authorization bugs do real damage, so VidCast adds explicit guardrails — and the chapter's headline is a fixed privilege-escalation hole.

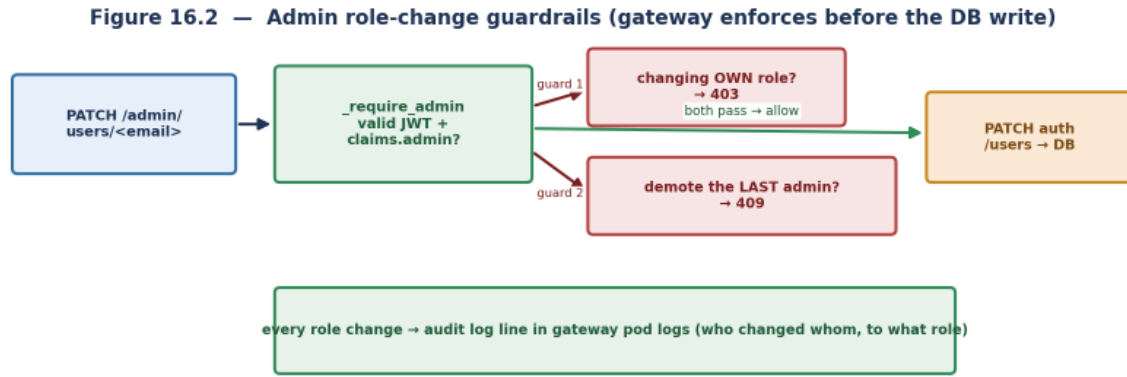
The root fix — self-registration cannot mint an admin

The original code returned an `admin` JWT from the registration endpoint — a straightforward privilege-escalation hole where anyone who signed up became an admin ("everyone was an admin"). The fix is categorical: self-registration is **hard-pinned to `role="user"`** in both the database insert and the auto-login token, so it is structurally impossible for a sign-up to create an admin account.

```
# register() - auth-service/server.py
INSERT INTO auth_user (email, password, role) VALUES (%s, %s, 'user')
...
return CreateJWT(email, JWT_SECRET, "user"), 201  # always a regular user
```

Two guardrails on role changes

When an admin **does** change another user's role (via `PATCH /admin/users/<email>`), the gateway applies two guardrails **before** the database write, to prevent foot-guns that would lock the platform out of administration:



Root fix of 'everyone was an admin': self-registration is hard-pinned to role='user' — it can never mint an admin JWT.

Figure 16.2 — The two admin guardrails. Self-demotion is blocked (403) and demoting the last remaining admin is blocked (409); every role change is audit-logged.

Guardrail	Condition	Response	Why
Self-demote	admin changes their *own* role	403	prevents accidental self-lockout
Last-admin	demotion would leave *zero* admins	409	prevents cluster-wide admin lockout
(both pass)	valid admin, not self, not last	allow → DB write	the only path that mutates a role

Table 16.1 — Admin role-change guardrails (gateway-service/server.py).

Both guards rely on PostgreSQL’s ACID guarantees (Chapter 7) so the check-then-write is consistent, and every role change writes an **audit-trail log line** in the gateway pod logs recording who changed whom to what role. These are operational-maturity touches — not just feature code — and they are recorded in `DECISIONS_MADE.md` as deliberate design choices.

16.4 IRSA — IAM Roles for Service Accounts (Least Privilege, No Long-Lived Keys)

Human identity (above) is matched by rigorous **machine identity**. Every workload that needs AWS access does so through **IRSA — IAM Roles for Service Accounts** — never a long-lived access key. A Kubernetes ServiceAccount is annotated with an IAM role ARN; the pod’s projected OIDC token is exchanged (`AssumeRoleWithWebIdentity`) for short-lived credentials scoped to exactly that role (the keyless flow introduced in Chapter 9 §9.8 and Figure 9.2).

VidCast uses IRSA for four distinct identities, each scoped to the minimum it needs:

Workload	ServiceAccount	IAM role grants	Chapter
External Secrets Operator	<code>vidcast-eso</code>	read <code>/vidcast/*</code> SSM params; decrypt via SSM only	16.5, 9.8
Backup CronJobs	<code>vidcast-backup</code>	<code>s3:PutObject/ListBucket</code> on the backup bucket only	24, 9.7
EBS CSI driver	<code>ebs-csi-controller-sa</code>	<code>AmazonEBSCSIDriverPolicy</code> (volume lifecycle)	9.4

Load Balancer Controller	<code>aws-load-balancer-controller</code>	official LBC policy (ELB/EC2/ACM)	9.9
GitHub Actions CD	(federated role)	<code>eks:DescribeCluster</code> + EKS access entry	11.3, 9.10

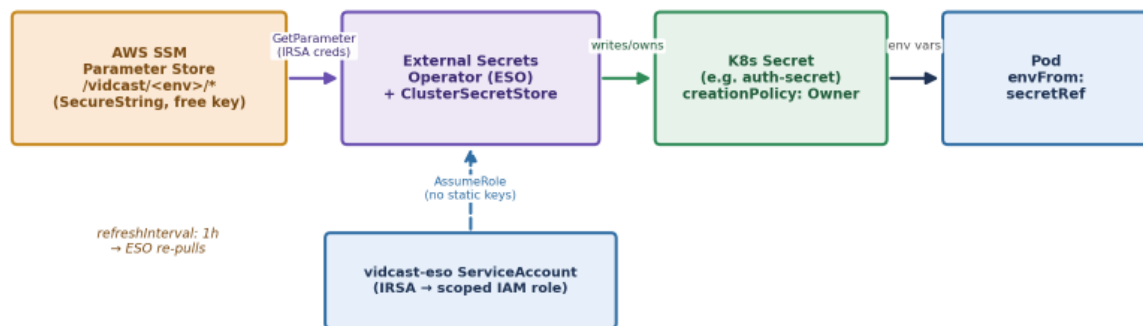
Table 16.2 — Every AWS-touching identity is keyless IRSA, scoped least-privilege. (The CD role uses the same OIDC mechanism via GitHub.)

The security pay-off is twofold: there are **no static AWS keys anywhere** to leak from a pod, an image, or Git; and each credential is **scoped so narrowly** that a compromised workload's stolen credentials are worth almost nothing — the backup job can only PUT to one bucket, ESO can only read `/vidcast/*` parameters. This is the least-privilege principle applied to machines, and it is central to the blast-radius containment of Chapter 14.

16.5 External Secrets Operator + AWS SSM Parameter Store (0 Secrets in Git)

VidCast's application secrets — database URIs, the JWT signing key, SMTP credentials — live in **AWS SSM Parameter Store** and are pulled into the cluster by the **External Secrets Operator (ESO)**. This replaced the old approach of manual, gitignored `secret.yaml` files and is what delivers the "**zero secrets in Git**" guarantee, closing findings H-3 (a leaked Gmail password in the tree) and H-4 (no secret management).

Figure 16.3 — External Secrets Operator: SSM Parameter Store → K8s Secret (0 secrets in Git)



Nothing secret ever lives in Git — only the path references (`/vidcast/dev/auth/jwt-secret, ...`) are committed.

Figure 16.3 — ESO pulls each parameter from SSM Parameter Store (via the vidcast-eso IRSA role) and writes the Kubernetes Secret the pod consumes. Only the path references are committed to Git.

How it works

14. A `ClusterSecretStore` (`vidcast-parameter-store`) points ESO at **SSM Parameter Store** in eu-west-2, authenticating via the `vidcast-eso` ServiceAccount's IRSA role (§16.4) — no static keys.
15. One `ExternalSecret` per service declares which parameters to fetch and what Kubernetes Secret to produce. For example, `auth-secret` maps the SSM path `/vidcast/dev/auth/jwt-secret` to the `JWT_SECRET` key and `/vidcast/dev/auth/psql-password` to `DATABASE_PASSWORD`.
16. ESO creates and **owns** the resulting Secret (`creationPolicy: Owner`), which the Deployment consumes via `envFrom.secretRef` — so the app reads ordinary environment variables, unaware of the source.

```
# k8s/external-secrets/dev/externalsecret-auth.yaml
spec:
  refreshInterval: 1h
  secretStoreRef: { name: vidcast-parameter-store, kind: ClusterSecretStore }
  target: { name: auth-secret, creationPolicy: Owner }
  data:
    - secretKey: DATABASE_PASSWORD
      remoteRef: { key: /vidcast/dev/auth/psql-password }
    - secretKey: JWT_SECRET
      remoteRef: { key: /vidcast/dev/auth/jwt-secret }
```

Why Parameter Store, not Secrets Manager. A deliberate cost decision (Chapter 26): Secrets Manager bills ~\$0.40/secret/month (~\$3/mo for the project's ~7 values, and it persists even when the cluster is destroyed). **Standard-tier SSM parameters are free**, and **SecureString** uses the **free** AWS-managed **alias/aws/ssm** key. ESO supports both backends — the only difference is **service: ParameterStore** in the store — so the project gets managed-secret hygiene at **\$0 standing cost**, preserving the "£0 when the cluster is off" target.

Critically, what **is** committed to Git is only the **path references** (`/vidcast/dev/auth/jwt-secret`, ...) and the key mappings — never a secret value. Parameters are seeded out-of-band from environment variables (`aws ssm put-parameter --type SecureString`), so no secret is ever written to a tracked file.

16.6 Secret Rotation, Refresh Intervals and ClusterSecretStore

The final piece is keeping secrets current. The model is pull-based and continuous rather than a one-time injection.

Refresh interval

Each **ExternalSecret** carries a `refreshInterval: 1h`, so ESO re-reads its parameters from SSM every hour and updates the Kubernetes Secret if the source changed. This means **rotation is a Parameter Store operation, not a redeploy**: to rotate the JWT secret or a database password, you `aws ssm put-parameter` the new value, and within the refresh window ESO propagates it into the cluster Secret — no manifest change, no image rebuild, no manual `kubectl`.

The ClusterSecretStore as a shared, cluster-scoped anchor

A single `ClusterSecretStore` (cluster-scoped, not namespace-scoped) is the shared trust anchor for all four service secrets. Centralising it means there is one place that defines the provider (Parameter Store), the region, and the IRSA identity — and one IAM role whose policy (`/vidcast/* read` + SSM-scoped decrypt) bounds what **any** secret pull can access. The per-environment split is handled by the path prefix: **dev/** ExternalSecrets read `/vidcast/dev/*` and **prod/** read `/vidcast/prod/*`, both through the same store.

Concern	Mechanism	Effect
Rotation	<code>aws ssm put-parameter</code> new value	propagates within the refresh window — no redeploy
Refresh cadence	<code>refreshInterval: 1h</code> per ExternalSecret	hourly re-pull and reconcile
Ownership	<code>creationPolicy: Owner</code>	ESO creates/updates/deletes the Secret; no manual drift
Trust anchor	one cluster-scoped ClusterSecretStore	single provider/region/IRSA definition
Env separation	<code>/vidcast/dev/*</code> vs <code>/vidcast/prod/*</code>	same store, different path prefix

Table 16.3 — Rotation and refresh model (k8s/external-secrets/).

The result is a secrets lifecycle that is auditable (every value has a known SSM path and version history), least-privilege (one scoped IRSA role gates all reads), rotatable without downtime, and — by design — entirely absent from the source repository.

In summary, VidCast's identity layer applies the same principles to people and machines: verify strongly, grant the least privilege, and never keep a long-lived credential where it can leak. Human authentication uses bcrypt cost-12 hashing with constant-time verification and SCRAM-backed PostgreSQL (closing H-2), JWT/HS256 tokens carrying a forward-compatible role claim, and gateway-enforced RBAC with explicit guardrails — self-demote 403, last-admin 409, and a hard-pinned **user** role on registration that eliminates the "everyone was an admin" escalation. Machine identity is keyless IRSA throughout — ESO, backups, EBS-CSI, the LBC, and CI — each role scoped to the minimum it needs. And application secrets live in SSM Parameter Store, pulled by the External Secrets Operator into owner-managed Kubernetes Secrets on an hourly refresh, rotatable without a redeploy and never present in Git at \$0 standing cost. Together these make identity a genuine defence-in-depth layer rather than a single gate.

Chapter 17 — Container and Runtime Hardening

This chapter covers the fourth defence-in-depth layer (Chapter 14): the runtime constraints that hem in the application **process** itself. Where the network layer (Chapter 15) limits what a pod can reach and the identity layer (Chapter 16) limits what its credentials are worth, this layer assumes the application code can be exploited and constrains what the resulting process may do — run as root, write to disk, hold Linux capabilities, escalate privilege, or make dangerous syscalls. Each is denied by an explicit setting in the Kubernetes **pod security context**, applied to every first-party workload and reinforced by the image-level **USER** directive (Chapter 10 §10.2) and by Kyverno admission policies (Chapter 19).

The defining property is **redundancy and breadth**: the controls are set in two places that reinforce each other (the Dockerfile and the pod spec), and they apply to every VidCast service, so a compromised process is non-root, on an immutable filesystem, with no capabilities and no path to escalate. The figure below shows how a single pod composes the pod-level and container-level controls.

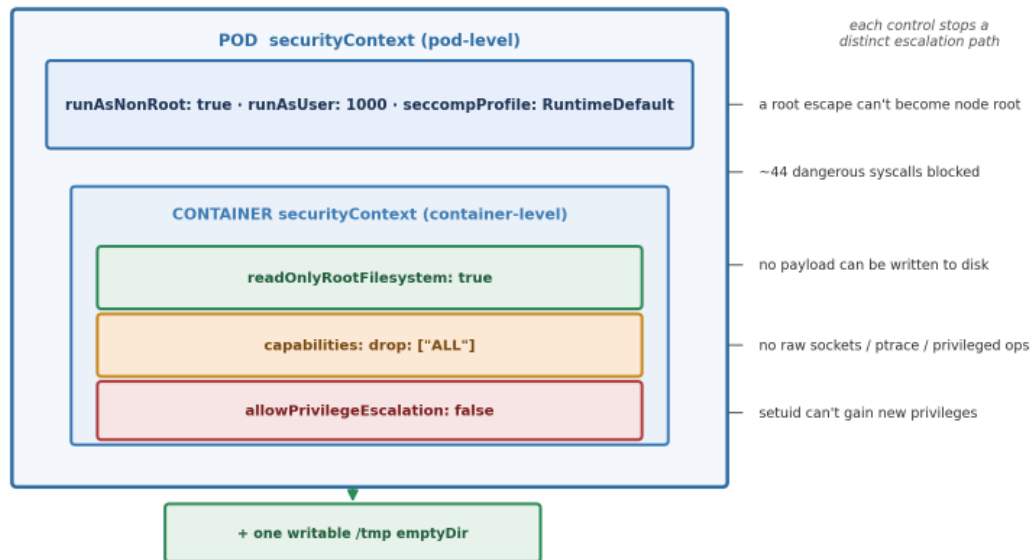
Figure 17.1 — Anatomy of a hardened VidCast pod

Figure 17.1 — Anatomy of a hardened pod: pod-level identity/seccomp wrapping container-level filesystem/capability/escalation locks, with one writable /tmp. Each control denies a distinct escalation path.

17.1 runAsNonRoot + UID 1000 (No Root in Any Pod)

The foundational control is that **no application process runs as root**. Every first-party workload sets, at the pod level, `runAsNonRoot: true` and `runAsUser: 1000` (the frontend uses `1001`):

```
# pod-level securityContext (all first-party services)
securityContext:
  runAsNonRoot: true
  runAsUser: 1000      # frontend uses 1001
```

- `runAsNonRoot: true` makes the kubelet *refuse to start the pod* if the image would run as UID 0 — a guarantee enforced by Kubernetes regardless of what the image tries to do.
- `runAsUser: 1000` pins the process to an unprivileged user. Because every service listens on a port above 1024 (5000, 8080, 9000), no privileged port-bind capability is needed (Chapter 10).

The security significance is containment: if an attacker exploits the application and gains code execution, they are an **unprivileged user inside the container** — so a container-breakout that would otherwise yield root on the node yields only an unprivileged process. This is the single most important control for blast-radius (Chapter 14 §14.3).

The honest exception — datastore images

"No root in any pod" is true for the *application* tier, but the project is candid that two third-party images cannot meet it. The official **MongoDB** and **PostgreSQL** entypoints must start as root to `chown` their data directories and run `initdb` (PostgreSQL then drops to the `postgres` user via `gosu`). These therefore run as root with only the *safe subset* of hardening applied (seccomp + no-privilege-escalation), recorded as a

documented Kyverno `require-non-root` Audit exception (Chapter 7). By contrast, RabbitMQ and Redis run fully non-root (uid 999). The coverage is mapped explicitly in §17.5.

17.2 readOnlyRootFilesystem with Explicit Writable tmp Volumes

Every backend container runs with `readOnlyRootFilesystem: true` — the container's entire filesystem is immutable at runtime. An attacker cannot drop a binary, modify application code, or persist a payload to disk, because there is nowhere writable to put it.

A read-only filesystem would break any process that legitimately needs scratch space, so each pod mounts **exactly one writable `emptyDir` at `/tmp`** and nothing else. This single, tightly-scoped exception serves every legitimate write each service makes:

Service	Why /tmp must be writable
Gateway	Werkzeug buffers multipart <code>/upload</code> bodies to a temp dir; Gunicorn workers write Prometheus multiprocess files to <code>/tmp/prometheus</code>
Auth	Gunicorn needs a scratch directory
Converter	FFmpeg/MoviePy stage temp video/MP3 files
Notification	the <code>/tmp/healthy</code> liveness heartbeat
outbox-relay	the <code>/tmp/healthy</code> heartbeat only

Table 17.1 — The single writable `/tmp emptyDir` per service and what writes to it (`k8s/base/*/deployment.yaml`).

The discipline here matters: the writable surface is one named volume, scoped to the one path each process needs, rather than a blanket writable filesystem. The comment trail in the manifests shows this was reasoned (e.g. the gateway's "Werkzeug buffers multipart uploads to a temp directory; without this `/upload` fails") rather than cargo-culted.

Frontend nuance. The frontend is the one first-party service with `readOnlyRootFilesystem: false`, because the nginx runtime writes to its cache, log, and PID directories. It is still hardened in every other way — non-root uid 1001, drop ALL capabilities, no privilege escalation, seccomp — and its Dockerfile pre-`chowns` those nginx directories to the non-root user (Chapter 10). So the writable surface is a controlled property of the nginx image, not an unhardened pod.

17.3 Drop ALL Capabilities; allowPrivilegeEscalation: false

Linux capabilities split root's power into discrete privileges (raw networking, mounting, ptrace, etc.). VidCast's containers need **none** of them, so every container drops them all and forbids regaining any:

```
# container-level securityContext
securityContext:
  capabilities:
    drop: ["ALL"]
  allowPrivilegeEscalation: false
```

Control	What it removes	Attack it blocks
<code>capabilities: drop: ["ALL"]</code>	every Linux capability (NET_RAW, SYS_ADMIN, SYS_PTRACE, ...)	raw-socket attacks, privileged syscalls, debugging other processes

<code>allowPrivilegeEscalation: false</code>	the ability to gain <i>*more*</i> privileges than the parent	a <code>setuid/setgid</code> binary cannot elevate the process
--	--	--

Table 17.2 — Capability and escalation locks (all containers).

`drop: ["ALL"]` is the least-privilege default — start from zero capabilities and add back only what is genuinely required, which for these pure-application processes is nothing. `allowPrivilegeEscalation: false` is the complementary guarantee: even if a `setuid` binary were present, the kernel will not let the process gain privileges beyond what it started with. Together they mean a compromised process has no Linux-level lever to widen its reach — no raw sockets to spoof traffic, no `ptrace` to inspect other processes, no privileged operations at all.

17.4 seccomp RuntimeDefault (Dangerous Syscalls Blocked)

Below capabilities sits the system-call interface — the kernel’s actual attack surface. Every pod applies a `seccompProfile` of `RuntimeDefault` at the pod level, which installs the container runtime’s default seccomp filter:

```
# pod-level securityContext
seccompProfile:
  type: RuntimeDefault
```

As the manifests note, `RuntimeDefault` **blocks roughly 44 dangerous syscalls** (the set the Pod Security Standards "Restricted" profile expects to be filtered) while permitting the syscalls a normal application needs. Blocked calls include the kind used in kernel-exploit and container-escape chains — so even if an attacker has code execution, the syscalls they would reach for to attack the kernel or break out of the container are simply unavailable.

- It is set at **pod level** so it also covers any future init or sidecar container, not just the main one.
- It satisfies the Kyverno `require-seccomp-runtime-default` policy (Chapter 19), so a pod that omitted it would be flagged at admission.
- It is the one hardening control that **is** applied even to the root-running MongoDB/PostgreSQL images (alongside `no-privilege-escalation`), so even those exceptions get syscall filtering.

Seccomp is the deepest of the runtime controls: capabilities limit what root-like operations are possible, but seccomp limits the kernel calls themselves — the last line of defence between a compromised process and the host kernel.

17.5 SHA-Pinned Images and the Hardened Pod Security Context

The final element is ensuring the **thing** that runs inside this hardened context is itself trustworthy and fixed. VidCast never deploys a floating `:latest` tag; every workload is pinned to an **immutable commit-SHA tag** set in the Kustomize overlay (Chapter 10 §10.4), so the running pod maps deterministically back to the exact source commit that built its image.

- The Kustomize overlays pin each image to its CI commit-SHA tag (e.g. `...-service:65f2f57`); base manifests carry a tag and the overlay rewrites it, so what is **admitted** always has a fixed tag, never `:latest`.

- ECR repositories are created **immutable** (Chapter 9 §9.6), so a SHA tag can never be silently overwritten — the tag is effectively a content pin.
- Kyverno **disallow-latest-tag** enforces this at admission, rejecting any pod that tries to run a **:latest** or untagged image; the documented next step is **Cosign digest signing + `verify-images` Enforce** (Chapter 30), moving from "this tag is immutable" to "this digest was signed by our CI."

The complete hardened security context

Pulling the layer together, every first-party pod runs under the same composite context — pod-level identity and seccomp, container-level filesystem, capabilities and escalation locks:

```
securityContext:          # POD level
  runAsNonRoot: true
  runAsUser: 1000
  seccompProfile: { type: RuntimeDefault }
# --- per container ---
securityContext:          # CONTAINER level
  readOnlyRootFilesystem: true
  allowPrivilegeEscalation: false
  capabilities: { drop: ["ALL"] }
```

The coverage of these controls across every workload — and the honest exceptions for the official datastore images — is summarised in the matrix below.

Figure 17.2 — Hardening-control coverage: app pods full; datastore images are documented exceptions

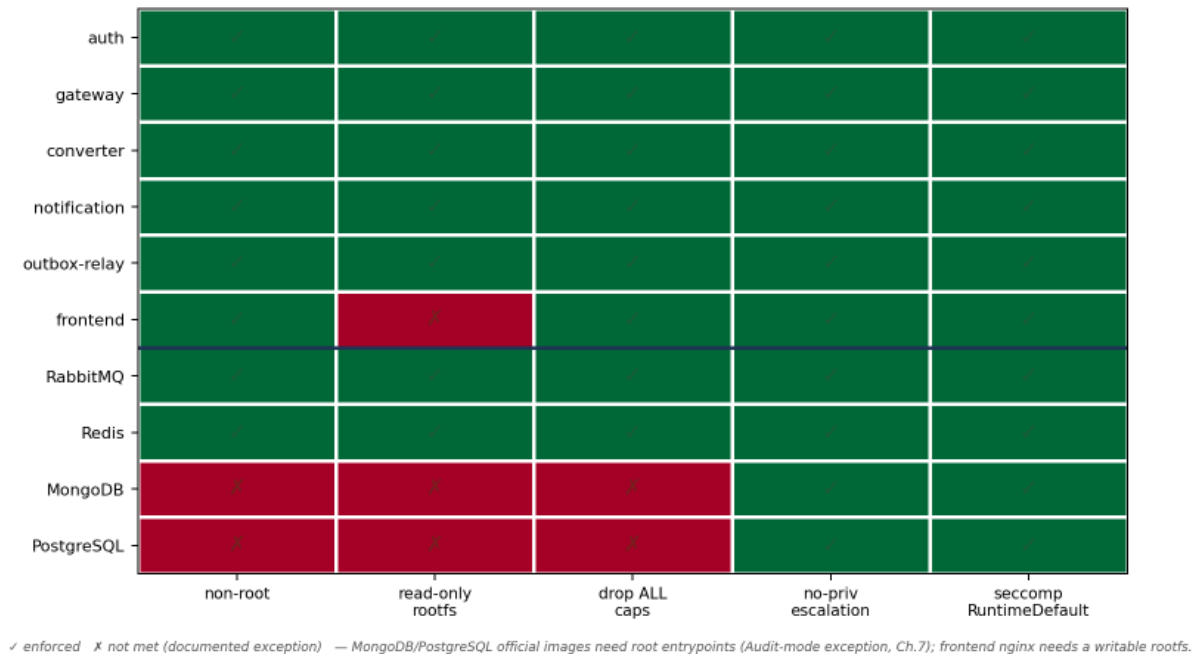


Figure 17.2 — Hardening-control coverage. First-party services are fully hardened; MongoDB/PostgreSQL are documented root exceptions (seccomp + no-escalation only) and the frontend needs a writable rootfs.

Workload	non-root	read-only fs	drop ALL	no-priv-esc	seccomp
auth, gateway, converter, notification, outbox-relay	✓ (uid 1000)	✓	✓	✓	✓
Frontend	✓ (uid 1001)	✗ (nginx needs	✓	✓	✓

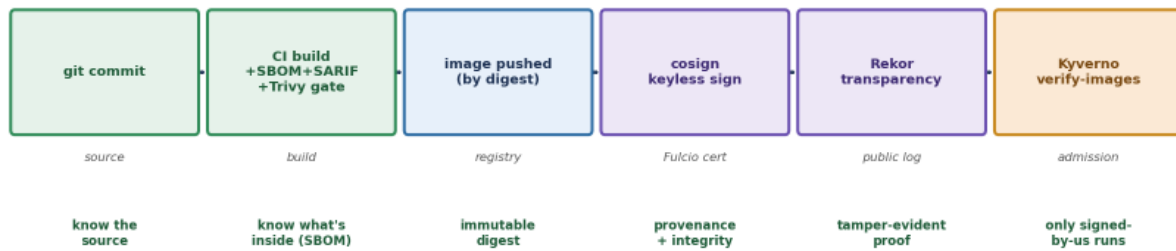
		writes)			
RabbitMQ, Redis	✓ (uid 999)	—	✓	✓	✓
MongoDB, PostgreSQL	X (root entrypoint)	X	X	✓	✓

Table 17.3 — Per-workload hardening coverage. Exceptions are documented, not accidental (Chapter 7, Chapter 10).

In summary, VidCast’s container and runtime hardening constrains the application process on five independent axes, each set explicitly in the pod security context and reinforced by the image and by Kyverno admission policies: it runs **non-root** (UID 1000, the kubelet refuses root), on a **read-only root filesystem** with a single tightly-scoped writable `/tmp`, with **all Linux capabilities dropped** and **privilege escalation forbidden**, under a **seccomp RuntimeDefault** filter that blocks ~44 dangerous syscalls — and the image it runs is pinned to an immutable commit-SHA tag. Every first-party service meets all five; the only gaps — the root-requiring MongoDB/PostgreSQL official images and the frontend’s writable nginx filesystem — are documented exceptions with the safe subset still applied. The net effect, with the other defence-in-depth layers, is that a process compromise is contained to a single unprivileged, immutable, capability-stripped, syscall-filtered container with nowhere to escalate.

Chapter 18 — Supply-Chain Security

This chapter covers the fifth defence-in-depth layer (Chapter 14): the supply chain. Its question is **can we trust the artefact we are about to run?** — independent of how hardened the runtime around it is. VidCast’s answer is a chain that makes every container image **verifiable** from the git commit that produced it, through the build pipeline, to the moment a pod is admitted to the cluster. Each link adds a distinct property: know the source, know what is inside, block known vulnerabilities, prove provenance, record it tamper-evidently, and finally admit only images that pass all of the above.

Figure 18.1 — The verifiable supply chain: git commit → signed image → verified at admission

Each link adds a property. Sections 18.1–18.6 detail each step; today CI build + Trivy gate + ECR hardening are live, signing/verify are roadmap (Ch.30).

Figure 18.1 — The verifiable supply chain (docs/SUPPLY_CHAIN.md). Each stage contributes one trust property; together they let the cluster admit only images built and signed by VidCast's own pipeline.

VidCast is candid about the state of this chain: the **build-time gates and registry hardening are live**, while the cryptographic signing and admission-time enforcement are **designed, written, and documented but not yet producing live signatures** (the roadmap of Chapter 30). This chapter describes each link and is explicit about which are implemented today.

18.1 Trivy CVE Gate — Scan Every Image After Build, Before Push

The first and most important **live** control is the **Trivy vulnerability gate** in CI. Every image is scanned immediately after it is built and **before it is pushed**, so a vulnerable image cannot reach the registry, let alone the cluster (introduced in Chapter 11 §11.2):

```
# .github/workflows/ci.yml – runs after build, before push
- name: Trivy vulnerability scan
  uses: aquasecurity/trivy-action@master
  with:
    image-ref: ${DOCKERHUB_USERNAME}/${service}:${SHORT_SHA}
    severity: CRITICAL,HIGH
    exit-code: "1"          # a finding FAILS the build
    ignore-unfixed: true    # only gate on fixable CVEs
```

- **`severity: CRITICAL,HIGH`** — only the two most serious classes gate the build.
- **`exit-code: 1`** — any matching finding fails the job, so the push step never runs. The result is a **property of the pipeline**: zero fixable CRITICAL/HIGH images ship, by construction.
- **`ignore-unfixed: true`** — keeps the gate actionable; it only fails on vulnerabilities a developer can actually patch.

This is "shift-left" in the literal sense — the check sits at the earliest point where it can stop a bad artefact, and it is reinforced downstream by ECR scan-on-push (§18.5). The same gate runs identically in the Jenkins pipeline (Chapter 11 §11.4), so the security bar does not depend on which CI system ran.

18.2 Image SHA-Pinning in Kustomize Overlays

A verified image is only useful if what runs is *exactly* that image. VidCast never deploys a floating `:latest` tag; every workload is referenced by an **immutable commit-SHA tag** set in the Kustomize overlay's `images:` block (detailed in Chapter 10 §10.4):

```
# k8s/overlays/dev/kustomization.yaml
images:
- name: johnbaabalola/gateway-service
  newTag: 65f2f57          # the CI commit SHA
- name: vidcast-frontend
  newName: 501562869470.dkr.ecr.eu-west-2.amazonaws.com/vidcast-frontend
  newTag: 6eb217c
```

The commit-SHA tag ties a running pod to the exact source commit that built its image — invaluable for forensics and rollback — and, combined with ECR tag-immutability (§18.5), behaves as a content pin: the tag can never be silently repointed to a different image. The Kyverno `disallow-latest-tag` policy (Chapter 19) enforces "no `:latest`" at admission. The documented next step (§18.4, Chapter 30) is to move from immutable *tags* to cryptographic *digest* references via Cosign, signing strictly **by digest, never by mutable tag**.

18.3 SBOM Generation and SARIF Reporting

Two further build-time artefacts make the contents and the scan results of each image transparent. Both are specified in the CI diff in `docs/SUPPLY_CHAIN.md` (the operator's to merge).

SBOM — know what is inside

A **Software Bill of Materials** in **CycloneDX JSON** is generated for every image with `anchore/sbom-action` (syft) — a complete component inventory of what the image contains. It is uploaded as a build artifact and, after push, attached to the image as a **signed Cosign attestation** (`cosign attest --type cyclonedx`, §18.4), so the inventory travels with the image and is itself tamper-evident.

SARIF — vulnerabilities visible in GitHub Security

A **second, non-gating** Trivy run emits **SARIF** and uploads it to GitHub code-scanning, so findings appear in the repository's Security tab. This is deliberately separate from the gate of §18.1: the SARIF run uses `exit-code: 0` so it never fails the build — the gate already did that — its job is *visibility* over time, not blocking.

Artefact	Tool / format	Purpose	Gating?
Trivy gate	aquasecurity/trivy-action, table	block fixable CRITICAL/HIGH	YES — fails build
SBOM	syft / CycloneDX JSON	component inventory ("what's inside")	no
SARIF report	Trivy / SARIF → GitHub Security	vulnerabilities visible over time	no (exit-code 0)

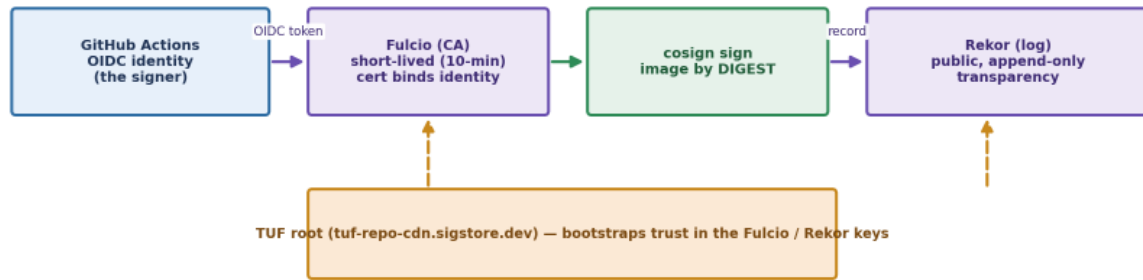
Table 18.1 — Build-time supply-chain artefacts (`docs/SUPPLY_CHAIN.md` CI diff).

18.4 Cosign Keyless Signing (Fulcio, Rekor, TUF Trust Anchors)

The cryptographic heart of the chain is **Cosign keyless signing**. After the Trivy gate and push, CI signs each image **by digest** using its GitHub Actions OIDC identity — there is **no private key to store or leak**. The signer's

identity *is* the OIDC token; Sigstore’s Fulcio issues a throwaway ~10-minute certificate binding that identity to the signature, and the binding is recorded in the Rekor transparency log forever.

Figure 18.2 — Cosign keyless signing trust anchors (no private key to leak)



```
certificate-identity = github.com/<org>/vidcast/.github/workflows/ci.yml@refs/heads/main · issuer = token.actions.githubusercontent.com
```

Figure 18.2 — The keyless signing trust anchors. The GitHub OIDC identity is bound by a short-lived Fulcio certificate; the signature is recorded in Rekor; the TUF root bootstraps trust in the Fulcio/Rekor keys.

Trust anchor	Endpoint	Role
OIDC issuer	token.actions.githubusercontent.com	GitHub vouches for the workflow’s identity
Fulcio (CA)	fulcio.sigstore.dev	issues the short-lived cert binding identity → signature
Rekor (log)	rekor.sigstore.dev	public, append-only transparency log of every signature
TUF root	tuf-repo-cdn.sigstore.dev	bootstraps trust in the Fulcio/Rekor keys

Table 18.2 — Sigstore trust anchors for keyless signing (docs/SUPPLY_CHAIN.md).

The signing identity is exact and must match the admission policy character-for-character — it is the GitHub Actions OIDC subject for the CI workflow on **main**:

```
certificate-identity:
https://github.com/<org>/vidcast/.github/workflows/ci.yml@refs/heads/main
certificate-oidc-issuer: https://token.actions.githubusercontent.com
```

A passing **cosign verify** against that identity proves: **this exact image digest was signed by our CI workflow on `main`, and the signature is in Rekor** — so it cannot have been forged or back-dated. The same mechanism attaches the SBOM as a verifiable attestation. CI signs the backend images (Docker Hub) and the frontend (ECR), so both registries are covered by one signer.

18.5 ECR Hardening — Tag Immutability, Scan-on-Push, Lifecycle, Encryption

The registry that holds the frontend image is itself hardened, in Terraform (**terraform/modules/ecr**, Chapter 9 §9.6) — and this part is **implemented today** (**terraform validate** passes). Four controls, all free within ECR limits:

Control	Setting	Why it matters
---------	---------	----------------

Tag immutability	IMMUTABLE	a verified digest cannot be swapped under the same tag — the basis of §18.2's content-pin
Scan on push	scan_on_push = true	a second CVE scan on every push — defence in depth behind the CI Trivy gate
Lifecycle	untagged expire after 7d; keep last 10 images	bounded storage and cost
Encryption	AES256 (AWS-managed key)	encrypted at rest; a CMK is deliberately skipped (~\$1/mo for marginal benefit)

Table 18.3 — ECR hardening controls (terraform/modules/ecr).

Tag immutability is the keystone that links registry hardening to the rest of the chain: it is what makes a commit-SHA tag (§18.2) a stable reference and prevents an attacker from re-pushing a malicious image under a tag a verified digest already occupies. As with the rest of A8, every control here adds **\$0 standing cost** — the deliberate "no CMK" decision keeps the supply-chain work free.

18.6 Roadmap — Kyverno verify-images from Audit to Enforce

The final link closes the loop at **admission**: the Kyverno **verify-images** ClusterPolicy (k8s/kyverno/verify-images.yaml) checks each pod's image signature *before the pod is allowed to run*. The policy is written, activated, and pointed at both registries with the exact keyless identity — but it deliberately runs in **Audit** mode today.

```
# k8s/kyverno/verify-images.yaml (excerpt)
spec:
  validationFailureAction: Audit          # observe only – not Enforce yet
  rules:
    - name: verify-cosign-keyless
      verifyImages:
        - imageReferences:
            - "docker.io/johnbaabalola/*"
            - "501562869470.dkr.ecr.eu-west-2.amazonaws.com/vidcast-frontend*"
          mutateDigest: false              # flip to true at Enforce
          attestors:
            - entries:
                - keyless:
                    subject:
                      "https://github.com/johnnybabs/vidcast/.github/workflows/ci.yml@refs/heads/main"
                      issuer: "https://token.actions.githubusercontent.com"
                      rekor: { url: "https://rekor.sigstore.dev" }
```

Why it stays Audit — the honest state. The policy must not go Enforce until (1) CI's Cosign-sign job is merged and producing signatures, and (2) at least one signed image has verified PASS on a live cluster. Until CI signs, the Audit report correctly shows the images as FAIL ("no signature") — the **expected, honest "not yet signed" state, not a bug**. Promoting prematurely would reject every workload.

The promotion is the Chapter 30 roadmap item: once signatures exist and verify live, **validationFailureAction** flips to **Enforce** and **mutateDigest: true** is set so admitted pods are pinned to the verified digest. At that point the guarantee becomes categorical — **only images signed by VidCast's own CI can run in the cluster**. The policy also **excludes** the platform namespaces (kube-system, kyverno, argocd,

kedo, external-secrets, monitoring) because those images are signed by other identities, so verification targets only first-party workloads.

Network prerequisite

For the admission controller to verify a signature it must reach Fulcio, Rekor, the TUF CDN and the registries. The `allow-kyverno-sigstore-egress` NetworkPolicy (Chapter 15 §15.6) is the egress carve-out — with the documented honest caveat that vanilla NetworkPolicy cannot pin to the Sigstore `*hostnames*` (IP/CIDR only), so it is a TCP-443-to-internet allow; true FQDN pinning would need a DNS-aware CNI or egress proxy.

Status — implemented vs roadmap

Figure 18.3 — Supply-chain status: what is live vs what is designed



Figure 18.3 — Honest supply-chain status. The Trivy gate, SHA-pinning, ECR hardening and the written verify-images policy are live; SBOM/SARIF, Cosign signing, and the Audit→Enforce promotion are the roadmap.

Item	State
ECR Terraform (immutability/scan/lifecycle/AES256)	live — <code>terraform validate</code> passes; import+apply at re-apply
Trivy CVE gate (CI + Jenkins)	live — blocks the build on fixable CRITICAL/HIGH
Commit-SHA image pinning (overlays)	live
Kyverno verify-images policy	written — Audit mode, both registries, real identity
SBOM / SARIF / Cosign sign (CI diffs)	documented for the operator; not yet merged
Live signatures in Rekor + a verify PASS	deferred — needs the CI signing job merged + a real run (Ch.30)

Table 18.4 — Honest implementation status (`docs/SUPPLY_CHAIN.md`).

In summary, VidCast’s supply-chain layer makes every image verifiable end to end. The live controls are substantial: a blocking Trivy CVE gate that stops fixable CRITICAL/HIGH images before they ship, immutable commit-SHA pinning so a running pod maps to its exact source, and a hardened ECR (immutable tags, scan-on-push, lifecycle, AES256) — all at \$0 standing cost. The cryptographic links — SBOM and SARIF for transparency, Cosign keyless signing anchored in Fulcio/Rekor/TUF for provenance, and the Kyverno `verify-images` policy for admission enforcement — are fully designed, written, and documented, with the policy

held honestly in Audit until CI is actually producing signatures. The chapter's integrity is in that honesty: it claims only what runs today and names the Audit→Enforce promotion as the roadmap step (Chapter 30) that will make "only signed-by-us images run" a categorical guarantee.

Chapter 19 — Policy-as-Code (Kyverno)

This chapter covers the sixth and outermost defence-in-depth layer (Chapter 14): policy-as-code. The previous five layers each rely on a setting being present — a pod *should* run non-root, *should* drop capabilities, *should* pin an image tag. Policy-as-code is the layer that **guarantees** those settings are actually there, by checking every workload against a codified baseline at admission and flagging (or, once promoted, rejecting) anything that violates it. VidCast implements this with **seven Kyverno `ClusterPolicy` resources** ([k8s/kyverno/](#)) that turn the security intentions of Chapters 15–18 into automatically-enforced rules.

The value is that the baseline is no longer a convention a developer might forget — it is admission-controlled. A new manifest that omits `runAsNonRoot`, sets `:latest`, or forgets resource limits is caught the moment it is applied, not discovered later in production.

Figure 19.1 — Kyverno admission control: today (Audit) vs after promotion (Enforce)

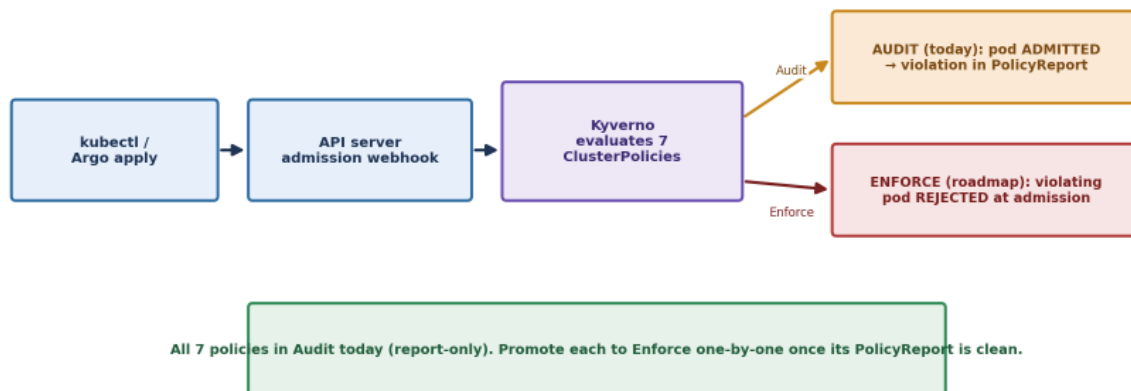


Figure 19.1 — Kyverno sits in the admission path. Today every policy is Audit (report-only); after promotion, a violating pod is rejected before it can run.

19.1 Policy Strategy and Audit-Mode Baseline; Exempt Platform Namespaces

Three deliberate strategy decisions govern the whole policy set.

Audit-mode baseline

Every policy ships in `Audit` mode (`validationFailureAction: Audit`) — violations are reported in a `PolicyReport`, but nothing is blocked. This is a safe-rollout choice: it surfaces the real state of the cluster (which workloads comply, which do not) *without* the risk of a too-strict rule blocking a legitimate deploy. Promotion to `Enforce` is a separate, deliberate follow-up done **one policy at a time, only once that policy's**

report is clean (the roadmap of Chapter 30). The README is explicit: "Never bulk-flip all policies to Enforce, or you'll block legitimate deploys."

Exempt platform namespaces

Every policy **excludes** the platform/system namespaces so the Audit report stays focused on the VidCast application in **default**. Third-party platform components (which VidCast does not author and which are signed/structured by other parties) would otherwise flood the report with violations the project cannot fix:

```
exclude:
  any:
    - resources:
        namespaces:
          - kube-system   - kube-public   - kube-node-lease
          - kyverno       - argocd       - keda       - external-secrets   - monitoring
```

Background scanning

Six of the seven policies set **background: true**, so Kyverno also evaluates **already-running** pods on a schedule — not just new admissions — giving a continuous compliance view. The exception is **verify-images**, which sets **background: false** because image-signature verification cannot run as a background scan (it needs the admission-time image reference).

Strategy decision	Setting	Rationale
Audit-mode baseline	validationFailureAction: Audit	report, don't block — safe rollout; promote per-policy when clean
Exempt platform ns	exclude 8 namespaces	keep the report focused on first-party app workloads
Background scan	background: true (6 of 7)	evaluate running pods continuously, not just at admission

Table 19.1 — The three governing strategy decisions (k8s/kyverno/).

Figure 19.2 — The seven Kyverno ClusterPolicies, by category



All exclude the platform namespaces (kube-system, kyverno, argocd, keda, external-secrets, monitoring) so the Audit report stays focused on the app.

Figure 19.2 — The seven ClusterPolicies grouped by category: Pod Security Standards, Best Practices, and Supply Chain Security.

19.2 require-non-root

Category: Pod Security Standards (Restricted). Rejects any pod that does not set `runAsNonRoot: true` — accepting it either at the pod level *or* on every container's `securityContext (anyPattern)`. As the policy comment puts it, running as non-root is "the single highest-leverage container hardening step — it turns many 'root on the node' escapes into nothing": a container that escapes the runtime as root owns the node, whereas a non-root escape is far less dangerous.

```
validate:
  message: "runAsNonRoot must be true (pod or every container securityContext)."
```

`anyPattern:`

- spec: { securityContext: { runAsNonRoot: true } }
- spec: { containers: [{ securityContext: { runAsNonRoot: true } }] }

This policy is the admission-time enforcement of the runtime control in Chapter 17 §17.1. Its known Audit violations are the official MongoDB/PostgreSQL images, whose entrypoints require root (Chapter 7) — the documented exception that must be carved out before this policy can go Enforce.

19.3 require-seccomp-runtime-default

Category: Pod Security Standards (Restricted). Rejects any pod that does not set a `seccompProfile` of type `RuntimeDefault` (pod or container level). `seccomp` filters which Linux syscalls a container may make; `RuntimeDefault` blocks roughly 44 dangerous or obscure syscalls the application never needs, shrinking the kernel attack surface available to a compromised container.

This is the admission counterpart to Chapter 17 §17.4. Notably, it is the one Restricted-profile control that *every* VidCast workload satisfies — including the MongoDB/PostgreSQL exceptions, which apply `seccomp` even though they cannot run non-root — so its Audit report is the closest to clean of the security policies.

19.4 require-requests-limits

Category: Best Practices. Rejects any pod whose containers do not set **both** CPU+memory **requests** AND CPU+memory **limits**. The rationale is operational and acute on a single 2-vCPU node: a container with no requests is invisible to the scheduler (it can be placed on an already-full node), and a container with no limits can consume the whole node and starve everything else.

```
validate:
  message: "CPU and memory requests AND limits are required on every container."
  pattern:
    spec:
      containers:
        - resources:
            requests: { cpu: "?*", memory: "?*" }
            limits: { cpu: "?*", memory: "?*" }
```

This is the policy "backing all the node-budget tracking" of the project — it is what makes the right-sizing and FinOps work of Chapter 26 enforceable rather than aspirational. The *?** pattern simply requires each field to be present and non-empty.

19.5 disallow-privileged

Category: Pod Security Standards (Baseline). This policy has **two** rules. The first rejects **privileged containers** (*privileged: false* required on init and main containers) — a privileged container effectively disables all container isolation, accessing host devices and the kernel directly and making escape trivial. The second uses a *foreach* to **deny dangerous added capabilities** — *SYS_ADMIN*, *NET_ADMIN*, or *ALL* — which grant near-root kernel powers:

```
# rule 2: disallow-dangerous-capabilities
foreach:
  - list: "request.object.spec.containers"
    deny:
      conditions:
        any:
          - key: "{{ element.securityContext.capabilities.add[] || `[]` }}"
            operator: AnyIn
            value: [ SYS_ADMIN, NET_ADMIN, ALL ]
```

No VidCast workload needs any of these — the services already *drop: ["ALL"]* (Chapter 17 §17.3) — so this policy enforces the floor below which container isolation would break down. It complements *require-non-root*: even a non-root container with *SYS_ADMIN* would be dangerous, and this rule forbids that combination.

19.6 disallow-latest-tag

Category: Best Practices. Also two rules: the first requires an **explicit image tag** (*image: "*:*" — untagged images default to :latest*), and the second forbids the **mutable :latest tag** specifically (*image: "!:latest"*). The reasoning is reproducibility: *:latest* is a moving target — you cannot tell what version is running, rollbacks are not reproducible, and two pods created minutes apart can run different code.

```
# rule 1: require an explicit tag      # rule 2: forbid :latest
pattern: { spec: { containers:        pattern: { spec: { containers:
  - image: "*:*" } }                  - image: "!:latest" } }
```

This is the admission-time enforcement of the commit-SHA pinning in Chapters 10 §10.4 and 18 §18.2 — it guarantees that the immutable-tag discipline the overlays follow is not bypassed by a stray manifest. The comment notes the Sprint-3 intent to "promote to Enforce later," consistent with the per-policy promotion strategy.

19.7 require-labels (Attribution & Cost Allocation)

Category: Best Practices. Rejects any pod missing the labels `app`, `environment`, and `app.kubernetes.io/managed-by`. Unlabelled resources cannot be **attributed** (who owns this?), cannot be **cost-allocated** (Kubecost groups by label, Chapter 26), and cannot be reliably **targeted** by other policies or selectors.

The choice of exactly three labels is deliberate and honest: it is "the minimum useful set," explicitly **not** yet requiring `cost-centre/owner` because the third-party datastore charts do not carry them and the project does not want to "drown the Audit report" in violations it cannot immediately fix.

`app.kubernetes.io/managed-by` is the standard label the Kustomize overlays already stamp (Chapter 12). This policy is what ties the security layer to the FinOps layer — it makes per-service cost attribution possible.

19.8 verify-images (Cosign Signature Admission Control)

Category: Supply Chain Security. The seventh policy is the admission-time link of the supply chain (Chapter 18 §18.6): it verifies a **Cosign keyless signature** on every first-party image before the pod is allowed to run, checking the signature, the signing identity, and the Rekor log entry. It targets both registries (Docker Hub backends + ECR frontend) with the exact keyless identity:

```
verifyImages:
  - imageReferences:
    - "docker.io/johnbaabalola/*"
    - "501562869470.dkr.ecr.eu-west-2.amazonaws.com/vidcast-frontend*"
    mutateDigest: false           # flip to true at Enforce (pin to digest)
    attestors:
      - entries:
        - keyless:
            subject:
              "https://github.com/johnnybabs/vidcast/.github/workflows/ci.yml@refs/heads/main"
            issuer: "https://token.actions.githubusercontent.com"
            rekor: { url: "https://rekor.sigstore.dev" }
```

Why it is special. `verify-images` is the only policy with `background: false` (signature checks need the admission-time image), and it is the one held in Audit **deliberately until CI is producing signatures** — until then its report correctly shows the images as FAIL ("no signature"), the expected "not yet signed" state (Chapter 18). It also needs a network egress carve-out so Kyverno can reach Fulcio/Rekor/TUF and the registries (Chapter 15 §15.6). It is promoted to Enforce **last**, after the other six.

19.9 Security Controls Summary and Compliance Mapping

The seven policies map onto recognised security frameworks — chiefly the Kubernetes **Pod Security Standards** (PSS) and general best practice — and each enforces, at admission, a control documented elsewhere in this report:

Policy	Category	Enforces (admission-time)	Defends / cross-ref
require-non-root	PSS Restricted	runAsNonRoot true	no node-root on escape (Ch.17)
require-seccomp-runtime-default	PSS Restricted	seccomp RuntimeDefault	blocks ~44 syscalls (Ch.17)
disallow-privileged	PSS Baseline	no privileged; no SYS_ADMIN/NET_ADMIN/ALL	preserves container isolation (Ch.17)
require-requests-limits	Best Practice	cpu+mem requests AND limits	node-budget / no starvation (Ch.26)
disallow-latest-tag	Best Practice	explicit tag, no <code>:latest</code>	reproducible deploys (Ch.10/18)
require-labels	Best Practice	app / environment / managed-by	attribution + cost allocation (Ch.26)
verify-images	Supply Chain	Cosign signature + identity + Rekor	only signed-by-us images (Ch.18)

Table 19.2 — The seven Kyverno policies, their framework category, and what each enforces.

The compliance picture is therefore coherent: VidCast does not merely **claim** to follow the Pod Security Standards Restricted profile — it has the admission policies that prove it, with a documented gap list (the datastore root exceptions, a few label/tag gaps) and a clear, safe promotion path from Audit to Enforce. Policy-as-code is what makes the entire six-layer security model **self-verifying**: the rules that the other chapters describe are continuously checked against reality, so drift from the security baseline is detected automatically rather than discovered after an incident.

Honest current state. All seven policies are in **Audit** today — they report, they do not block. This is by design (safe rollout), and the project documents both the known violations and the per-policy Audit→Enforce promotion procedure (Chapter 30). The artefacts that exist now are the seven validated policy files; live PolicyReports are produced on the next cluster re-apply.

In summary, VidCast’s policy-as-code layer codifies its security baseline into seven Kyverno ClusterPolicies that check every workload at admission: three Pod Security Standards controls (non-root, seccomp, no-privileged/dangerous-caps), three best-practice controls (resource requests+limits, no `:latest` tag, required labels), and one supply-chain control (Cosign image verification). Every policy runs in Audit mode today — reporting, not blocking — with platform namespaces excluded to keep the report focused on the app, background scanning of running pods for six of seven, and a deliberate one-policy-at-a-time promotion path to Enforce. Each rule is the admission-time guarantee of a control described elsewhere in this report, which makes the whole defence-in-depth model self-verifying: the security intentions of Chapters 15–18 are continuously checked against the cluster’s real state, so the baseline cannot silently erode.

Part V — Operations, Observability & Reliability (Ops)

Chapter 20 — Observability – Metrics and Dashboards

Observability is what lets a two-person team operate VidCast with confidence: it turns the running system into queryable signals, and those signals into dashboards and the SLOs and alerts of the next chapter. This chapter documents the metrics half of that story — the kube-prometheus-stack that collects and stores the data (20.1), the application metrics each service exposes and how they survive Unicorn's multiple worker processes (20.2), the ServiceMonitors and PodMonitors that tell Prometheus what to scrape (20.3), the three Grafana dashboards built on top (20.4), and why RabbitMQ queue depth is treated as the platform's leading backlog indicator (20.5). Everything here is taken from the monitoring/ directory and the service code.

20.1 kube-prometheus-stack (Prometheus / Grafana / Alertmanager)

The monitoring foundation is the kube-prometheus-stack Helm chart, which bundles the Prometheus Operator, Prometheus, Grafana and Alertmanager and wires them together. The operator model is the key choice: rather than hand-editing Prometheus scrape config, the platform declares ServiceMonitor and PodMonitor custom resources, and the operator translates them into scrape targets automatically (20.3). Figure 20.1 shows the resulting topology.

Operator-native scrape topology: ServiceMonitors and PodMonitors (labelled release=monitoring) tell Prometheus what to scrape; Grafana and Alertmanager consume it.

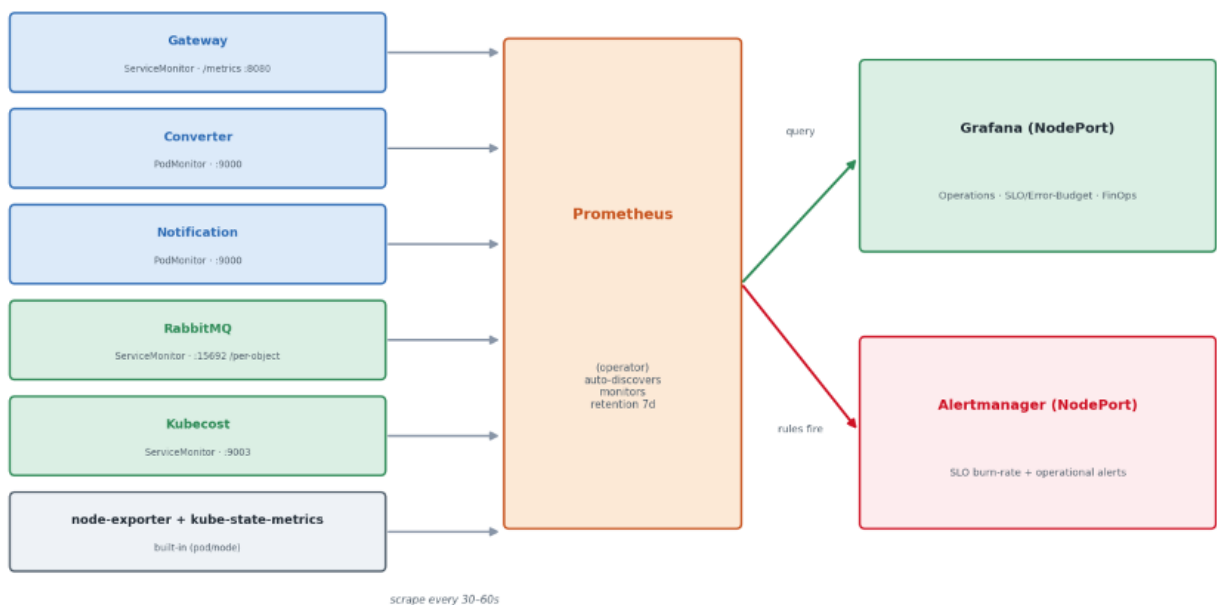


Figure 20.1 — The kube-prometheus-stack scrape topology: Service/PodMonitors drive Prometheus, which feeds the three Grafana dashboards and Alertmanager.

The stack is configured deliberately for a small, cost-capped, single-node cluster:

- **Prometheus retention is 7 days** — matched to the demo cluster's storage; the SLO alerts use $\leq 6h$ windows so they are unaffected, while the dashboards' 30-day budget panels are honestly labelled as a 7-day view.

- **Grafana and Alertmanager are exposed via NodePort** — admin-only access for the demo (Grafana admin password vidcast-demo); never the public ALB.
- **Control-plane scrape jobs are disabled** — etcd, kube-scheduler, kube-controller-manager and kube-proxy are not scrapable on managed EKS, so those jobs are turned off to avoid permanently-down targets.
- **node-exporter and kube-state-metrics stay enabled** — they provide the node CPU/memory and pod/restart metrics the Operations dashboard and the operational alerts depend on.

20.2 Application Metrics via prometheus_client (Multiprocess-Aware)

Each service emits its own metrics with the prometheus_client library, and these application metrics — not just infrastructure counters — are what the three SLOs in the next chapter are computed from.

Table 20.1 lists them.

Metric	Type	Service	Labels / buckets	Used for
vidcast_gateway_requests_total	Counter	Gateway	method, endpoint, status	Availability SLO (5xx ratio)
vidcast_gateway_request_duration_seconds	Histogram	Gateway	method, endpoint	Request-latency panels
vidcast_gateway_in_flight_requests	Gauge	Gateway	multiprocess livesum	Concurrency / saturation
vidcast_uploads_total	Counter	Gateway	—	End-to-end SLO denominator
vidcast_conversions_total	Counter	Converter	—	Throughput / cost-per-conversion
vidcast_conversion_duration_seconds	Histogram	Converter	buckets 1...300...600,∞	Conversion-latency SLO (le=300)
vidcast_notifications_total	Counter	Notification	status	End-to-end SLO numerator (success)

Table 20.1 — Application metrics exposed by the services (the SLI sources for Chapter 22).

The subtle part is multiprocess-awareness. The gateway runs two Gunicorn worker processes, and with the default in-process registry a single scrape would see only one worker's counters — halving every rate. The gateway therefore enables prometheus_client's multiprocess mode: each worker writes samples to a PROMETHEUS_MULTIPROC_DIR on the pod's writable /tmp emptyDir, and the /metrics handler aggregates them at scrape time with a MultiProcessCollector (the in-flight gauge uses multiprocess_mode="livesum" to sum across live workers). The converter and notification services are single-process queue consumers, so they need no aggregation — each simply runs prometheus_client's start_http_server on a background thread, exposing /metrics on container port 9000 for a PodMonitor to scrape.

20.3 ServiceMonitors and PodMonitors (Gateway, Converter, Notification, RabbitMQ, Kubecost)

What Prometheus scrapes is declared, not configured by hand. Each target is a ServiceMonitor (for workloads behind a Service) or a PodMonitor (for the queue consumers, which have no Service), and every one carries the label release=monitoring — the selector the kube-prometheus-stack operator uses to adopt it. Table 20.2 is the complete scrape matrix.

Target	Monitor kind	Port · path	Interval	Notes
Gateway	ServiceMonitor	http (8080) · /metrics	30s	Behind a Service; auto-discovers all gateway pods
Converter	PodMonitor	metrics (9000) · /metrics	30s	No Service — scrape pods directly
Notification	PodMonitor	metrics (9000) · /metrics	30s	No Service — scrape pods directly
RabbitMQ	ServiceMonitor	prometheus (15692) · /metrics/per-object	30s	Per-object endpoint for per-queue depth; job relabelled to "rabbitmq"
Kubecost	ServiceMonitor	tcp-model (9003) · /metrics	60s	cost-model series for the FinOps dashboard (kubecost ns)

Table 20.2 — The scrape matrix: which targets are scraped, how, and why pod- versus service-level.

Two details are worth calling out because they fix real problems. The RabbitMQ monitor scrapes `/metrics/per-object` rather than the default `/metrics` endpoint: the default returns only cluster-aggregated figures, whereas the RabbitMQQueueBacklog alert needs the per-queue series `rabbitmq_queue_messages{queue="video"}`, which is only exposed per-object (cardinality is trivial here — two queues plus the dead-letter topology). It also forces the job label to "rabbitmq" via a relabeling so the operational alerts' `up{job="rabbitmq"}` selector resolves deterministically. The Kubecost monitor closes the FinOps loop: Kubecost reads raw metrics from Prometheus and exposes computed cost series (`node_total_hourly_cost`, allocation metrics) back, which this ServiceMonitor scrapes for the cost dashboard.

20.4 Grafana Dashboards — Operations, SLO/Error-Budget, FinOps

Three Grafana dashboards are provisioned as code (JSON in `monitoring/dashboards/`), each aimed at a different audience and question. Table 20.3 lists their panels.

Dashboard	Answers	Panels
VidCast Operations	Is the platform healthy right now?	Pod status (all services); pod restarts (1h); node CPU % and memory % gauges; RabbitMQ queue depth; CPU and memory per pod
VidCast SLO / Error Budget	Are we meeting our objectives, and how fast is budget burning?	For each of the three SLOs: budget remaining (7d), burn rate (1h), projected hours-to-exhaustion, and a burn-rate trend
VidCast FinOps / Cost	What is it costing, and per unit of work?	Cluster cost (\$/hr); projected monthly cost; cost per conversion; conversions/hour; cluster cost trend; CPU by workload; estimated monthly cost by namespace

Table 20.3 — The three provisioned Grafana dashboards and their panels.

The split mirrors the operator's actual workflow: the Operations dashboard is the first glance for live health (pod status, restarts, node pressure, queue depth); the SLO/Error-Budget dashboard answers whether objectives are being met and reuses the same normalised burn-rate series the alerts fire on (Chapter 22); and the FinOps dashboard ties cost to work done — its standout panel, cost per conversion, is the metric that makes the platform's economics legible. Because the dashboards are JSON in the repository, they are version-controlled and recreated on every deploy rather than hand-built in the UI.

20.5 RabbitMQ Queue Depth as the Leading Backlog Indicator

Among all the signals, RabbitMQ queue depth has a special role: it is a leading indicator. CPU, latency and error rate tell you a problem is already happening; the depth of the video queue tells you a backlog is forming before users feel it, because work piles up in the queue first and only then shows as slower conversions. Figure 20.2 illustrates the dynamics around an upload burst.

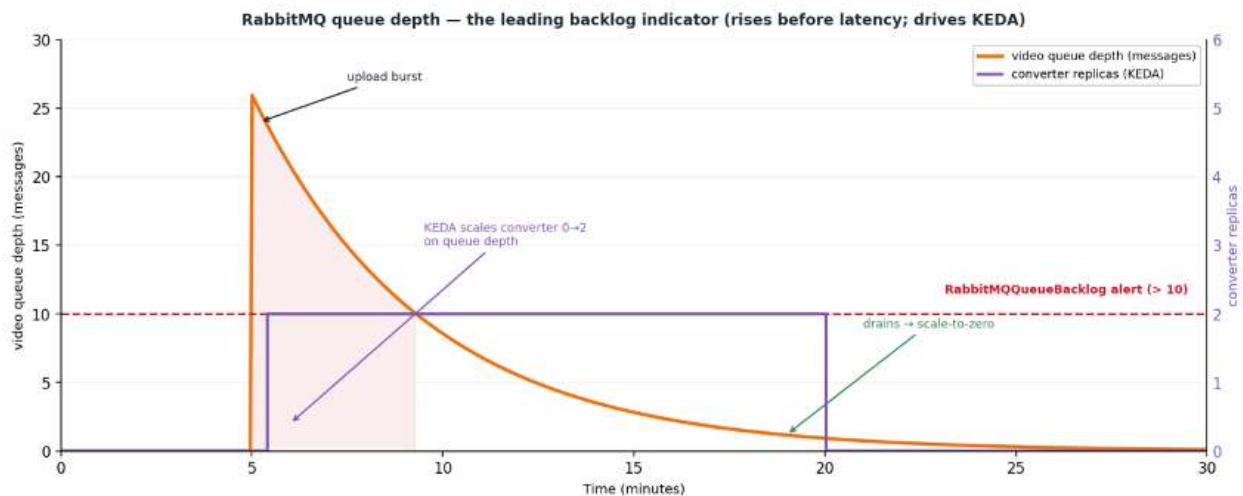


Figure 20.2 — Queue depth as the leading backlog indicator: it spikes on an upload burst ahead of latency, crosses the alert threshold, and is the signal KEDA scales the converter on.

This single signal is wired into three of the platform's mechanisms at once. It is the trigger for autoscaling — KEDA scales the converter from zero on the video queue's depth, so the same metric that warns of a backlog also resolves it. It is the operational alert `RabbitMQQueueBacklog`, which fires when more than ten messages wait for conversion for five minutes — the earliest sign that the converters are overwhelmed or crashed. And it is the first diagnostic step in the conversion-latency runbook: when latency burns budget, the operator's first question is whether the video queue is backed up. Exposing it correctly is exactly why the `RabbitMQServiceMonitor` scrapes the per-object endpoint (20.3) — without the per-queue `rabbitmq_queue_messages{queue="video"}` series, none of these three uses would work. Queue depth is therefore not just another panel; it is the connective tissue between observability, autoscaling and alerting.

Chapter 21 — Structured Logging and Tracing

Chapter 20 covered metrics — the **aggregate** health of the platform. This chapter covers the complementary half of observability: **logs** — the **per-event** record that lets an operator follow a single file, audit a single download, and answer the question the whole project was built around: *"where is my file?"* (Chapter 2). VidCast's logging is deliberately structured (machine-parseable JSON, never raw `print()`), threaded with a correlation ID that ties every hop of a request together, and used to produce a download audit trail that satisfies the §3.4 compliance objective.

The work here closes the project's "I8 / P3" (structured logging + correlation IDs) and "A12" (download audit) items, and is documented in [docs/OBSERVABILITY.md](#). It is application-code only — no extra infrastructure — which is part of why it fits the cost-capped, single-node platform.

21.1 JSON Logging via Shared jsonlog.py (No print())

Every service logs **one JSON object per line to stdout** via a small shared module, `jsonlog.py`. The deliberate replacement of `print()` with structured logging is what makes the logs **queryable** — a log aggregator or `jq` can filter and join on fields, which raw text lines do not allow.

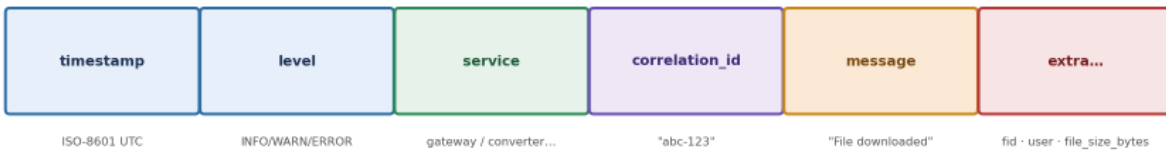
The log format

Every line carries a fixed set of fields, plus any structured context the call site passes:

Field	Meaning
timestamp	ISO-8601 UTC
level	INFO / WARNING / ERROR
service	gateway / auth / converter / notification / outbox-relay
correlation_id	per-request trace id ("none" for process-level lines, "legacy" for pre-correlation messages)
message	human-readable text
(extra)	any call-site context — e.g. <code>fid</code> , <code>user</code> , <code>file_size_bytes</code> , <code>error</code>

Table 21.1 — Fields on every log line (`src/*/jsonlog.py`, `docs/OBSERVABILITY.md §1`).

Figure 21.2 — Anatomy of a structured log line (one JSON object per line, stdout)



Common fields on every line + arbitrary call-site context via `kwargs (extra=)`. `default=str` so `ObjectId/datetime` never break a log call.

Figure 21.1 — Anatomy of a structured log line: a fixed common set of fields plus arbitrary per-call context, emitted as one JSON object per line.

How it works

`jsonlog.py` provides a `get_logger(service)` factory returning a thin adapter whose `info/warning/error/exception` methods accept arbitrary keyword fields. A custom `_JsonFormatter` builds the JSON payload — the common fields plus any `extra=` context — and serialises it with `default=str` so an `ObjectId`, `datetime`, or `bytes` value can never crash a log call:

```
log = get_logger("gateway")
log.info("File downloaded", correlation_id=g.correlation_id,
        fid=fid_string, user=access["username"],
        file_size_bytes=getattr(out, "length", None))
# → {"timestamp": "...", "level": "INFO", "service": "gateway",
```

```
# "correlation_id":"abc-123","message":"File downloaded",
# "fid":"6a1a","user":"x@y.com","file_size_bytes":295749}
```

- **Stdout, one line per event** — the standard Kubernetes logging contract, so `kubectl logs` and any future log shipper (Chapter 30) work without app changes.
- **Pod-level `RuntimeDefault` and `PYTHONUNBUFFERED=1`** ensure lines are flushed immediately, not buffered.

Why `jsonlog.py` is duplicated per service. The module is copied into each service directory on purpose — exactly like `idempotency.py` (Chapter 8). The services are separate Docker build contexts with no shared package on `PYTHONPATH`, so a single shared module would not be importable without a Dockerfile change. The duplication is a deliberate, documented trade-off, not an oversight.

21.2 Correlation IDs Minted at the Gateway, Carried Every Hop

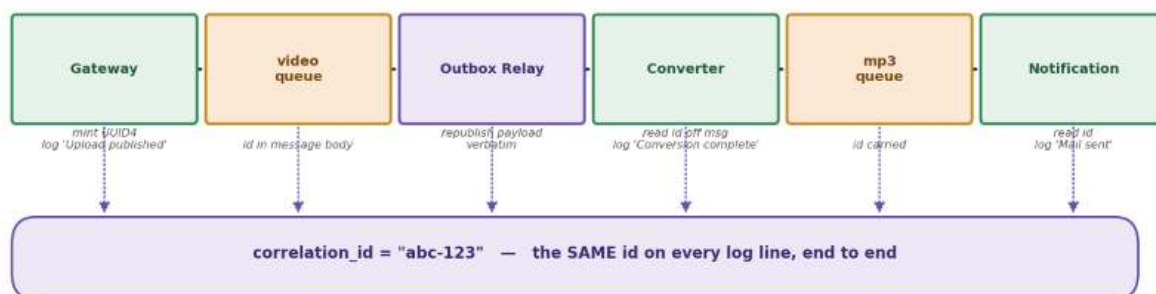
Structured fields make a *single* line queryable; a **correlation ID** makes the *whole journey* of one request queryable. The gateway mints a fresh `uuid4` for every request in a `before_request` hook and threads it into both the logs and the outgoing queue message:

```
@server.before_request
def _metrics_before():
    # Fresh correlation id per request, threaded into logs and the queue message.
    g.correlation_id = str(uuid.uuid4())
```

From there the id travels with the work, not just the request:

17. The **gateway** stamps the id into the RabbitMQ message body (and logs "Upload published/queued").
18. The **outbox relay**, when enabled, republishes the stored payload **verbatim** — so the id survives the outbox hop unchanged.
19. The **converter** reads the id off the message (`parsed.get("correlation_id", "legacy")`) and logs "Conversion complete" with it.
20. The **notification** service reads the same id and logs "Mail sent".

Figure 21.1 — One correlation_id, minted at the gateway, carried to the email



grep one id across all services → reconstruct the whole journey of one file

Figure 21.2 — The `correlation_id` is minted once at the gateway and carried through the queues, relay, converter and notification — so one id appears on every log line from upload to email.

The result is that **a single id appears on every log line from upload to delivery**, across five separate services and two queues. Lines with no request context use the sentinel `"none"` (process-level logs), and pre-correlation messages use `"legacy"` — both documented so a query can distinguish "untraced" from "missing".

Service	Logs the id by	Example message
gateway	minting it (<code>g.correlation_id</code>)	"Upload accepted", "File downloaded"
outbox-relay	republishing the payload verbatim	"Outbox event published"
converter	reading it off the message	"Conversion complete"
notification	reading it off the message	"Mail sent"

Table 21.2 — How each service obtains and logs the `correlation_id` (`src/*/*.py`).

21.3 Download Audit Trail

The §3.4 compliance objective — keep client content in-account **with a full download audit trail** — is satisfied directly by structured logging. Every successful `GET /download` emits one structured audit line from the gateway recording **who downloaded which file, when, and how big**:

```
# gateway-service/server.py - download()
log.info("File downloaded", correlation_id=g.correlation_id,
        fid=fid_string, user=access["username"],
        file_size_bytes=getattr(out, "length", None))
# failures:
log.warning("Download failed", correlation_id=g.correlation_id, error=...)
```

Because the line is structured, the audit is **queryable**, not just present:

```
# every download, ever:
kubectl logs -l app=gateway | jq 'select(.message=="File downloaded")'
# everything one user downloaded:
kubectl logs -l app=gateway | jq 'select(.user=="x@y.com")'
```

Audited event	Log message	Captured fields
Successful download	<code>File downloaded</code>	user, fid, file_size_bytes, correlation_id, timestamp
Failed download	<code>Download failed</code>	user, fid, error, correlation_id
Admin role change	<code>Admin role change</code>	who changed whom, to what role (Chapter 16)

Table 21.3 — The audit events emitted by the gateway (`docs/OBSERVABILITY.md §3`).

This turns the compliance liability of the status quo — client content on unknown third-party sites with **no** record of access (Chapter 2 §2.4) — into a compliance **asset**: the company can answer "who accessed what, when" for any file, directly from the logs. The same mechanism audits admin role changes (Chapter 16 §16.3), so privileged actions are recorded too.

21.4 "Where Is Our File?" — End-to-End Request Tracing

The defining operational question of the whole project — *"where is my file?"*, which had **no answer** in the manual status quo (Chapter 2 §2.3) — is answered in minutes by combining the two mechanisms above.

Given a file's correlation id, an operator can reconstruct its entire journey by filtering every service's logs for that one id:

```
CID=abc-123
for app in gateway converter notification outbox-relay; do
  kubectl logs -l app=$app --tail=-1 \
    | jq -c "select(.correlation_id == \"$CID\")"
done
```

The lines returned, in order, tell the complete story:

Service	Log line (message)	What it tells the operator
gateway	"Upload published" / "Upload queued via outbox"	accepted and enqueued
outbox-relay	"Outbox event published"	the relay published it to the broker
converter	"Conversion complete" (or "Conversion failed", outcome)	FFmpeg succeeded — or failed and where it went (retry/DLQ)
notification	"Mail sent" (or "Notification failed")	the email with the download link went out

Table 21.4 — The trace of one upload, reconstructed from logs by correlation_id (docs/OBSERVABILITY.md §2).

Crucially, this works even when something goes *wrong*. If a file never arrived, the trace shows exactly where it stopped: a missing "Conversion complete" with a "Conversion failed" line (and its **outcome** = **retry** or **dlq**) pinpoints a bad conversion; a missing "Mail sent" points at the notification step. This is the diagnostic counterpart to the SLOs of Chapter 22 — metrics tell you *that* something is wrong; the correlation-id trace tells you *which file* and *which hop*.

Logs + metrics + (roadmap) traces. Together with the Prometheus metrics of Chapter 20, structured logging gives VidCast two of the three observability pillars today. The third — centralised log aggregation — is the documented next step: a Fluent Bit DaemonSet shipping these JSON logs to Loki (Chapter 30 §30.6) would turn the per-service **kubectl logs** loop above into a single cluster-wide query, making "where is my file?" instant rather than a few commands.

In summary, VidCast's logging is structured by design: every service emits one JSON object per line via the shared **jsonlog.py** (never raw **print()**), with a common field set plus arbitrary call-site context and crash-proof serialisation. A **uuid4** correlation id minted at the gateway is threaded into the queue messages and read off by the converter and notification services — and republished verbatim by the outbox relay — so a single id appears on every log line from upload to email. That same structured logging produces a queryable download audit trail (who downloaded what, when, how big) that converts the project's original compliance liability into an asset, and it makes the founding question — "where is my file?" — answerable in minutes by tracing one id across all services, even pinpointing the exact hop where a failed file stopped. With metrics (Chapter 20) it forms a genuine observability foundation, with centralised log shipping the documented next step.

Chapter 22 — Service Level Objectives and Alerting

Observability is only useful if it drives action, and the bridge between the two is a set of service level objectives (SLOs) with alerting that pages a human only when an objective is genuinely at risk. VidCast defines three SLOs — availability, conversion latency, and end-to-end success — each measured from the metrics foundation (the gateway /metrics endpoint, the converter and notification exporters, and RabbitMQ's prometheus plugin), and each protected by multi-window, multi-burn-rate alerts modelled on the Google SRE workbook. This chapter documents each SLO and its SLI (22.1–22.3), the error-budget and burn-rate alerting machinery (22.4), the operational alerts that catch infrastructure failures directly (22.5), and the runbook that turns each alert into a first action (22.6).

An honest framing first: the SLO targets are demonstrative. VidCast runs on a single-node cluster that is deliberately torn down between sessions, and every teardown is, by definition, 100% unavailability — so the headline percentages are not a uptime claim. The portfolio artefact is the machinery — the normalised burn-rate recording rules, the multi-window PrometheusRules, and the error-budget dashboard — which, pointed at an always-on deployment, would enforce real SLOs unchanged.

#	SLO	Target	Window	SLI (how it is measured)
1	Availability	99.9% of gateway requests are non-5xx	30 days	$1 - (5xx \div \text{total})$ on <code>vidcast_gateway_requests_total</code>
2	Conversion latency	95% of conversions finish ≤ 5 min	30 days	fraction of <code>vidcast_conversion_duration_seconds</code> in the <code>le=300</code> bucket
3	End-to-end success	99% of uploads produce a notification email	30 days	$\text{vidcast_notifications_total}\{\text{success}\} \div \text{vidcast_uploads_total}$

Table 22.1 — The three VidCast SLOs, their targets, windows, and the SLI each is computed from.

22.1 Availability SLO — 99.9% Non-5xx Gateway Requests

The availability SLO targets 99.9% of gateway requests returning a non-5xx status over a 30-day window. The gateway instruments every served request into the `vidcast_gateway_requests_total` counter (labelled by method, endpoint and status), excluding the /metrics scrape and the /healthz probe so they cannot skew the signal. The SLI is the success ratio $1 - (5xx \div \text{total})$; the error ratio that burns budget is therefore:

```
sum(rate(vidcast_gateway_requests_total{status=~"5.."}[w]))
/ sum(rate(vidcast_gateway_requests_total[w]))
```

A 99.9% target leaves a budget factor of 0.001 — an error budget of $0.1\% \times 30 \text{ days} = 43.2$ minutes of allowed 5xx responses per 30-day window. Spending that budget faster than it refills is what the burn-rate alerts in 22.4 detect.

22.2 Conversion-Latency SLO — 95% of Conversions ≤ 5 Minutes

The conversion-latency SLO targets 95% of conversions finishing within five minutes (300 seconds) over 30 days. The converter records each job's publish-to-MP3-write duration into the

vidcast_conversion_duration_seconds histogram; the SLI is the fraction of conversions landing in the le="300" bucket, so the "bad" ratio is the fraction slower than five minutes:

```
1 - ( sum(rate(vidcast_conversion_duration_seconds_bucket{le="300"}[w]))
      / sum(rate(vidcast_conversion_duration_seconds_count[w])) )
```

The budget factor is 0.05 — up to 5% of conversions in the window may exceed five minutes. Two project-specific notes matter here. First, the histogram counts only completed jobs: a job that exhausts its retries and dead-letters never enters the histogram and is counted as an end-to-end-success failure (22.3), not a latency failure — intentional and standard. Second, the single-node cap matters — the converter's KEDA maxReplicaCount is 2, so at saturation the second replica may sit Pending, which is itself a legitimate cause of latency burn (see the runbook in 22.6).

22.3 End-to-End Success SLO — 99% of Uploads Produce a Notification

The end-to-end SLO is the one a user actually experiences: 99% of accepted uploads should result in a notification email, over 30 days. It spans the whole asynchronous pipeline — gateway → outbox → RabbitMQ → converter → mp3 queue → notification — by comparing the two ends of it: the notifications the system sent against the uploads it accepted.

```
1 - ( sum(rate(vidcast_notifications_total{status="success"}[w]))
      / sum(rate(vidcast_uploads_total[w])) )
```

The budget factor is 0.01 — a time-equivalent budget of $1\% \times 30 \text{ days} = 432 \text{ minutes}$ (7.2 hours) of total pipeline failure per window, or equivalently 1% of uploads may go un-notified. The important measurement caveat, documented in the project, is that this SLI is time-shifted: an upload and its email are minutes apart, so over short windows the sends ÷ uploads ratio is noisy and can momentarily exceed 1. It is only trustworthy over long windows ($\geq 6\text{h}$), which is exactly why this SLO relies on the 6h/30m slow-burn pair and not the fast-burn pair.

22.4 Error Budgets and Multi-Window, Multi-Burn-Rate Alerting

An error budget reframes reliability as something spendable: at 99.9% availability, 0.1% of requests may fail before the objective is missed. "Burn rate" is how fast that budget is being spent, and VidCast records it already normalised — the recording rules divide the error ratio by the budget factor (0.001, 0.05, 0.01), so a recorded value of 1 means "spending the budget exactly as fast as it refills" and 14 means "14× too fast." That normalisation lets the alert thresholds read simply as > 14 and > 1 , and lets the Grafana error-budget dashboard reuse the same series. Figure 22.1 shows the burn-down for the availability budget.

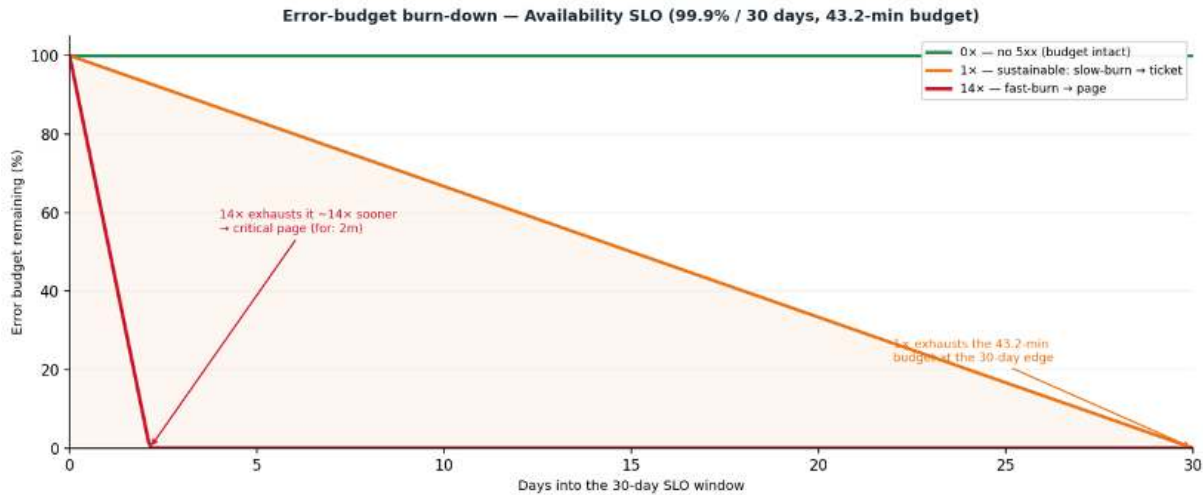


Figure 22.1 — Error-budget burn-down for the availability SLO: a 1x burn just exhausts the 43.2-minute budget at the 30-day edge; a 14x burn exhausts it far sooner and pages.

Each SLO has two alerts, and each alert is multi-window: a long window sets the severity and a short window must also be burning. The long window (1h fast, 6h slow) decides whether this is a page or a ticket; the short window (5m fast, 30m slow) ensures the alert clears quickly once the incident is over, instead of latching on for the length of the long window. Figure 22.2 shows the logic, and Table 22.2 lists the exact configuration shipped in `vidcast-slo-rules.yaml`.

Multi-window, multi-burn-rate: the LONG window sets severity; the SHORT window must ALSO be burning, so the alert clears quickly once the incident ends.

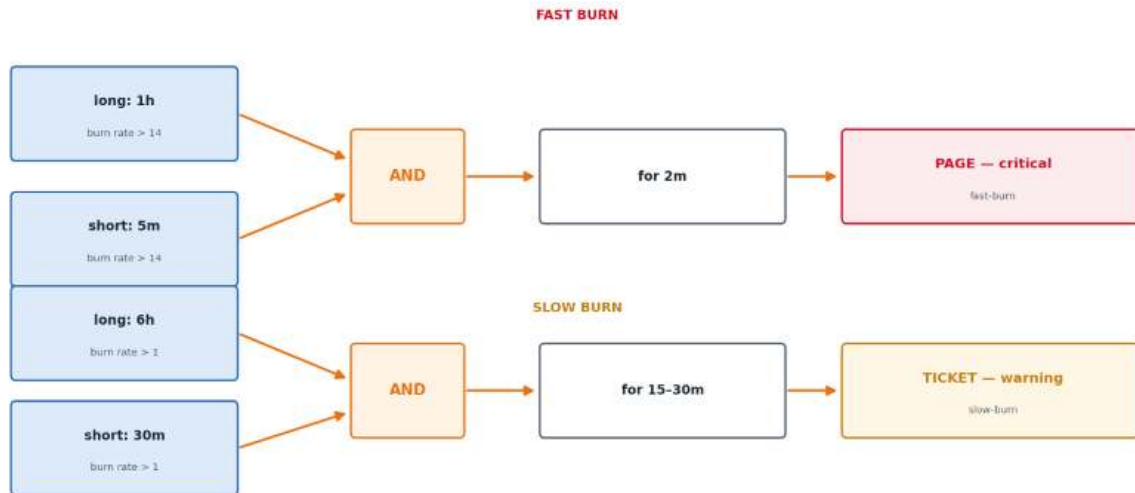


Figure 22.2 — Multi-window, multi-burn-rate logic: the long window sets severity; the short window must also be burning (logical AND), so the alert self-clears after the incident.

Alert	Severity	Long window	Short window	Threshold	for
Availability fast-burn	critical (page)	1h	5m	> 14x	2m
Availability slow-burn	warning (ticket)	6h	30m	> 1x	15m
Conversion-latency fast-burn	critical (page)	1h	5m	> 14x	2m

Conversion-latency slow-burn	warning (ticket)	6h	30m	> 1×	15m
End-to-end fast-burn	critical (page)	1h	5m	> 14×	5m
End-to-end slow-burn	warning (ticket)	6h	30m	> 1×	30m

Table 22.2 — The six burn-rate alerts: each SLO has a fast-burn (page) and slow-burn (ticket) pair, both requiring long AND short windows to be burning.

Two honest caveats accompany the budgets. Prometheus retention is 7 days while the budgets are stated over 30 days — the alerts are unaffected because they use windows of $\leq 6h$, but the dashboard's "budget remaining" panel is necessarily a 7-day view (a true 30-day accounting needs Thanos or remote-write). And when the cluster is idle there is no traffic: the ratios divide by a zero rate, yielding NaN, and $NaN > N$ is false, so the alerts simply stay quiet — correct behaviour for a demo cluster that is often idle.

22.5 Operational Alerts (CrashLoopBackOff, Node CPU/Mem, Queue Backlog, Broker Down)

The SLO alerts catch symptoms the user feels; a second set of operational alerts (vidcast-alerts.yaml) catches the infrastructure causes directly, often before an SLO is breached. These are conventional threshold alerts on Kubernetes and RabbitMQ signals.

Alert	Condition (summarised)	for	Severity
PodCrashLoopBackOff	container restart rate in default ns > ~3 in 10 min	5m	critical
HighNodeMemoryUsage	node memory used > 85% (OOMKill risk for converter)	5m	warning
HighNodeCPUUsage	node CPU used > 85% (converter saturating the node)	5m	warning
RabbitMQQueueBacklog	rabbitmq_queue_messages{queue="video"} > 10	5m	warning
RabbitMQUnavailable	up{job="rabbitmq"} == 0 (whole pipeline blocked)	2m	critical

Table 22.3 — Operational alerts: direct signals on pod health, node resources, and the broker.

These alerts are deliberately tuned to the single-node reality. The node-resource thresholds (85%) warn before the converter — the heaviest workload — triggers an OOMKill or saturates CPU; the video-queue backlog threshold of ten messages is the leading indicator that conversion is falling behind; and RabbitMQUnavailable is critical because the broker is the spine of the entire upload-to-email pipeline. Together with PodCrashLoopBackOff, they convert the most common infrastructure failure modes into named, actionable alerts.

22.6 SLO Runbooks (Alert → First Action)

Every alert carries a runbook annotation pointing at SLO.md, so an on-call engineer is never left guessing. The runbooks are deliberately short — the first few diagnostic steps that most often find the cause — and they follow the dependency chain rather than a generic checklist.

Alert (SLO)	First actions
-------------	---------------

Availability fast/slow-burn	1) kubectl logs deploy/gateway — tracebacks / dependency errors. 2) Check /healthz — is MongoDB or RabbitMQ the failing dependency? 3) Check PodCrashLoopBackOff + gateway pod restarts.
Conversion-latency fast/slow-burn	1) Is the video queue backed up? (RabbitMQQueueBacklog / RabbitMQ UI). 2) Is KEDA scaling the converter? kubectl get scaledobject,deploy converter — remember maxReplicaCount: 2, so a Pending 2nd replica IS a latency cause. 3) Converter CPU throttling / OOM? kubectl top pod -l app=converter.
End-to-end fast/slow-burn	1) Inspect the dead-letter queues (video.dlq, mp3.dlq). 2) kubectl logs deploy/notification — SMTP/Gmail failures? (if GMAIL_APP_PASSWORD=SKIP, sends fail by design — disable the alert). 3) Is the outbox-relay publishing? kubectl logs deploy/outbox-relay.

Table 22.4 — SLO runbooks: each burn-rate alert maps to a short, dependency-ordered first response.

This closes the observability loop end to end: a metric becomes an SLI, an SLI is held to an SLO with a quantified error budget, the budget is watched by multi-window burn-rate alerts that distinguish a transient blip from a sustained incident, and every alert that fires comes with the first command to run. The result is alerting that respects the on-call engineer's attention — paging only when an objective is genuinely at risk, ticketing when it is merely drifting, and staying silent when the cluster is simply idle.

Chapter 23 — Abuse Protection and Rate Limiting

Two endpoints on the gateway are attractive to abuse: /login, which a brute-force or credential-stuffing attacker will hammer, and /upload, which a single account could use to flood the conversion pipeline and the storage tier. VidCast protects both with application-level rate limiting (the A10 hardening item) implemented with Flask-Limiter backed by the in-cluster Redis. This chapter documents the limiter configuration and the policy it enforces (23.1), then the more subtle part — how the client IP is derived through the ALB/nginx proxy chain, and the deployment caveats that come with it (23.2). Everything here is taken from the gateway service code and the frontend nginx configuration.

23.1 Flask-Limiter with Redis Backend (Login 10/min, Upload 20/hr)

The gateway constructs a single Flask-Limiter instance whose counters live in the in-cluster Redis (storage_uri redis://redis:6379). Using Redis rather than the default in-process store is the whole point: the gateway runs two Gunicorn worker processes (and can run as multiple pods), and a per-process counter would let a client get the limit once per worker. With Redis, every worker and pod increments the same per-IP bucket, so the limit is global. Figure 23.1 shows the flow.

One shared counter: both Gunicorn workers (and every gateway pod) increment the SAME per-IP bucket in Redis, so the limit is global, not per-process.

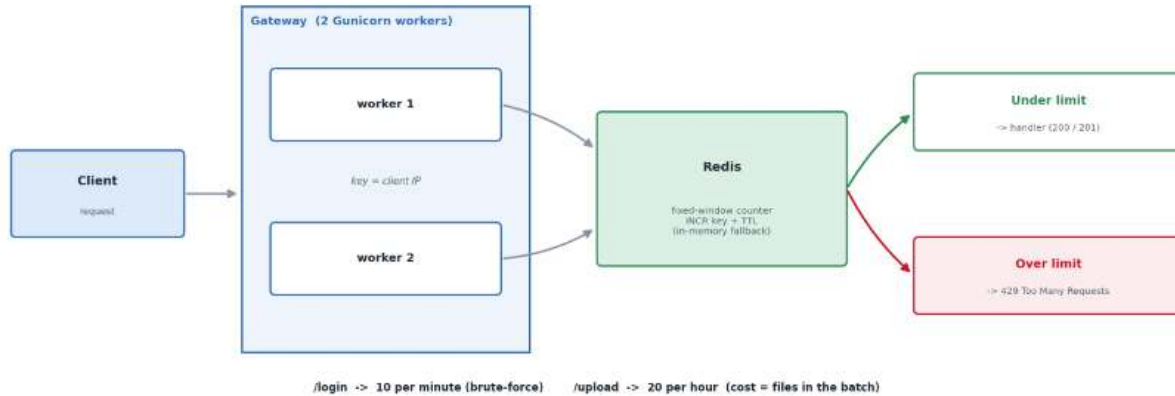


Figure 23.1 — Flask-Limiter with a shared Redis counter: all workers/pods increment one per-IP bucket; under the limit the handler runs, over it the request gets a 429.

The limiter is configured deliberately. The strategy is fixed-window; the key function is a custom `_client_ip` (23.2) rather than the default remote address; `default_limits` is empty, so nothing is throttled globally — only the two explicitly-decorated endpoints are limited; and `in_memory_fallback_enabled` is true, so if Redis is unreachable the limiter degrades to an in-process counter rather than failing the request. The two policies are:

Endpoint	Limit	Cost / unit	Why
POST /login	10 per minute	1 per request	Brute-force / credential-stuffing protection
POST /upload	20 per hour	1 per file in the batch (cost callable)	Pipeline + storage quota; a batch of 8 spends 8 tokens
POST /register	none	—	Not currently rate-limited (documented gap in 23.2)
all other routes	none	—	<code>default_limits</code> is empty by design

Table 23.1 — The rate-limit policy: only /login and /upload are throttled; /upload's cost scales with batch size.

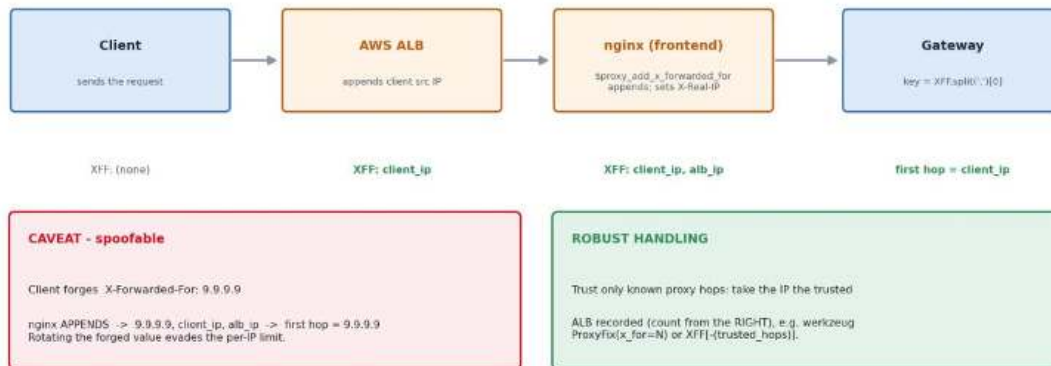
The /upload cost callable is the notable refinement: the decorator is `@limiter.limit("20 per hour", cost=lambda: max(1, len(request.files.getlist("file"))))`, so the quota is charged per file rather than per request. A single-file upload spends one token and a batch of eight spends eight, which keeps the hourly budget meaningful whether a user uploads one large file or many small ones, and aligns the rate limit with the actual load placed on the converter and GridFS. When a caller exceeds either limit, Flask-Limiter returns HTTP 429 Too Many Requests before the route body runs.

23.2 X-Forwarded-For Handling and Deployment Caveats

Rate limiting is only as good as the identity it keys on, and behind a proxy that identity is not obvious. VidCast's traffic reaches the gateway through two proxies — the AWS ALB and the frontend's nginx — so the gateway's `request.remote_addr` is merely the nginx pod's address. Keying the limiter on that would

put every client into a single global bucket: ten failed logins from anyone would lock out everyone, turning the brute-force protection into a trivial denial-of-service. The limiter therefore keys on the X-Forwarded-For chain instead (Figure 23.2).

request.remote_addr at the gateway is only the nginx pod - so the limiter keys on the X-Forwarded-For chain instead.



Other deployment caveats: per-IP (not per-account) - a botnet from many IPs needs WAF/ALB rules; fixed-window allows a 2x burst at window edges; Redis-down falls back to per-pod in-memory counters; register is not rate-limited.

Figure 23.2 — The X-Forwarded-For chain through ALB and nginx, the spoofing caveat of keying on the first hop, and the robust handling that trusts only known proxy hops.

The custom `_client_ip` key function reads the X-Forwarded-For header and takes the first hop (`xff.split(",")[0]`), falling back to `get_remote_address()` when the header is absent. In the honest path this works: the ALB appends the real client source IP, nginx appends its peer via `$proxy_add_x_forwarded_for` (and sets X-Real-IP), and the leftmost hop is the genuine client — so each client gets its own bucket.

The deployment caveat, acknowledged directly in the code, is that this leftmost value is client-spoofable. Because nginx appends to X-Forwarded-For rather than replacing it, a client can send a forged header (for example X-Forwarded-For: 9.9.9.9); the proxies append the real addresses after it, but the gateway still reads 9.9.9.9 as the first hop. An attacker can then rotate the forged value to get a fresh bucket on every request and evade the per-IP limit entirely. The robust handling is to trust only a known number of proxy hops — selecting the IP the trusted ALB actually recorded by counting from the right of the header (for example werkzeug's ProxyFix with `x_for` set to the proxy count, or indexing `XFF[-(trusted_hops)]`) — rather than blindly taking the leftmost, attacker-controlled value. This is recorded as a known limitation of the current implementation.

Several further deployment caveats bound what this control does and does not buy:

Caveat	Effect	Mitigation / status
First-hop XFF is spoofable	An attacker can forge the key and evade the per-IP limit	Trust N proxy hops (ProxyFix / count-from-right) — documented
Per-IP, not per-account	A botnet from many IPs is not stopped by per-	Edge controls (WAF / ALB rules) are the right

	IP limits	layer for that
Fixed-window strategy	Allows up to a 2× burst across a window boundary	Acceptable here; sliding-window is the stricter alternative
Redis-unreachable fallback	Counters degrade to per-pod in-memory (weaker, fail-open)	<code>in_memory_fallback_enabled</code> keeps requests serving, not failing
<code>/register</code> is not limited	Account-creation spam / enumeration is unthrottled	Documented gap; add a limit + the email-exists 409 already present

Table 23.2 — Deployment caveats of the rate-limiting design and their mitigation or status.

Taken together, the rate limiting is a sound, honest application-layer control: it shares state correctly across workers via Redis, charges `/upload` by real cost, and fails open rather than taking the service down if Redis blips. Its boundary is equally clear — it is per-IP and depends on trustworthy proxy headers, so it complements, rather than replaces, edge protections (the ALB and a WAF) for distributed abuse. The two caveats worth closing before heavier production use are tightening the X-Forwarded-For trust to a fixed hop count and adding a limit to `/register`.

Chapter 24 — Disaster Recovery and Business Continuity

The application and control plane are recoverable from Git at any time — Argo CD will reconcile a fresh cluster back to the committed manifests — but Git cannot rebuild the stateful tier. A disaster-recovery design for VidCast therefore has to answer one question Git cannot: where does the data come from after a total loss? This chapter documents the answer end to end — the nightly backup strategy (24.1), the controls that protect the backup bucket (24.2), the deliberately asymmetric IRSA model that lets the cluster write backups but not read or delete them (24.3), the two independent recovery paths that combine to a full restore (24.4), and the tested runbook with its RTO/RPO targets (24.5). It closes with the merge runbook for the one change class that must be deployed atomically — the coordinated bcrypt + RBAC rollout (24.6). Every command, bucket name, schedule and role here is taken from the repository and the disaster-recovery runbook.

24.1 Backup Strategy — Nightly mongodump (02:00) + pg_dump (02:15) to S3

Two Kubernetes CronJobs take logical dumps of the stateful tier every night and ship them to S3. They are staggered by fifteen minutes so the two dumps do not contend for the single node's CPU and network at the same instant. Figure 24.1 shows the flow: each CronJob dumps in an init container, gzips the output to a shared scratch volume, and an `amazon/aws-cli` container then copies it to the bucket.

Nightly logical backups: two CronJobs dump to gzip, then ship to a hardened S3 bucket as the write-only vidcast-backup ServiceAccount.

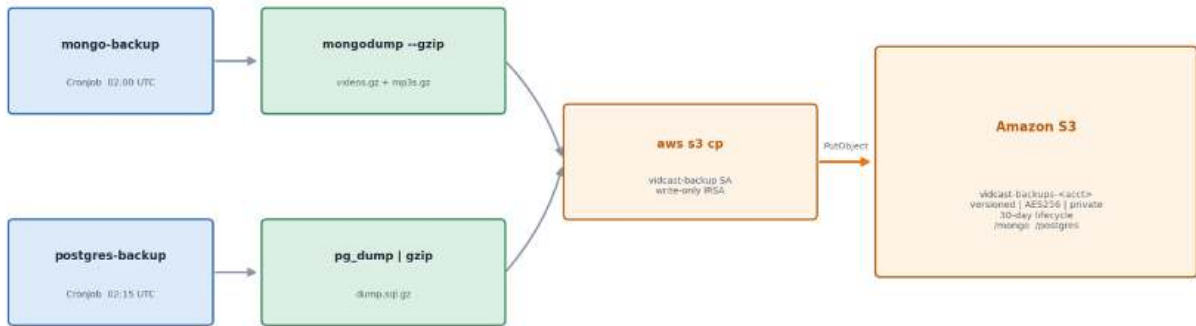


Figure 24.1 — Backup architecture: staggered nightly CronJobs dump and gzip, then upload to a hardened S3 bucket as the write-only vidcast-backup ServiceAccount.

Datstore	Tool	Schedule (UTC)	Destination prefix	Object key pattern
MongoDB videos + mp3s (GridFS files + outbox)	mongodump --uri --gzip --archive (per DB)	02:00 nightly	s3://vidcast-backups-<acct>/mongo/	videos-<TS>.archive.gz, mp3s-<TS>.archive.gz
PostgreSQL (authdb)	pg_dump gzip	02:15 nightly	s3://vidcast-backups-<acct>/postgres/	postgres-<TS>.sql.gz

Table 24.1 — What is backed up, with what tool, when, and where (timestamped object keys).

One restore-critical detail is captured in the runbook: the MongoDB backup authenticates with the application's own credentials (MONGODB_VIDEOS_URI / MONGODB_MP3S_URI from gateway-secret, user mongouser, authSource=admin), not the mongodb-secret root user — whose password is out of sync with the running mongod and fails SCRAM-SHA-256. Because each URI pins to one database, the run produces two archives (videos and mp3s); both must be restored together. The CronJobs run non-root (uid 1000), with concurrencyPolicy: Forbid so a slow dump never overlaps the next night.

24.2 Backup Bucket Controls — Versioned, Public-Access-Blocked, 30-Day Lifecycle

The destination is a single bucket, vidcast-backups-<account-id> (the account ID is appended for global uniqueness so the name is deterministic and the CronJobs can hardcode it), created and hardened by terraform/modules/storage. Because the dumps contain user data — GridFS media and auth rows — the bucket is locked down by default:

- **Versioning enabled** — keeps a short history of each object so a bad nightly dump cannot immediately overwrite the last good one.
- **Public access fully blocked** — all four settings (block ACLs, block policy, ignore ACLs, restrict buckets) are true; the bucket can never become public.
- **Server-side encryption (AES256)** — the free AWS-managed S3 key, consistent with the project's no-CMK cost decision.

- **Lifecycle expiry (30 days)** — current objects and non-current versions both expire after `retention_days` (default 30), and abandoned multipart uploads abort after 7 days, so the bucket cannot grow unbounded.
- **Single region by decision** — cross-region replication is deliberately omitted under the single-region cost decision and recorded as a follow-up for region-loss survivability.

24.3 Write-Only Backup IRSA vs Separate Restore Role

The identity that takes backups and the identity that restores them are deliberately different. The CronJobs run as the `vidcast-backup` ServiceAccount, annotated with the IRSA role `vidcast-cluster-backup-irsa`, whose policy is scoped to `s3:PutObject` and `s3:ListBucket` on this one bucket — and nothing else. It has no `s3:GetObject` and no `s3:DeleteObject`, so the in-cluster identity can add and list backup objects but cannot read their contents back or delete them.

Capability	Backup CronJob IRSA (in-cluster)	Restore (operator workstation)
Write a new backup (PutObject)	Allowed	n/a
List objects (ListBucket)	Allowed	Allowed
Read a backup (GetObject)	Denied	Allowed (separate, broader identity)
Delete / overwrite (DeleteObject)	Denied	Restricted; versioning protects history
Scope	This bucket only; no other AWS access	Operator AWS credentials, used by a human

Table 24.2 — Asymmetric IRSA: the cluster can write backups but not read or delete them; restore uses a separate identity.

This asymmetry is a least-privilege and tamper-resistance control. A compromised pod holding the backup role cannot exfiltrate the backups (no `GetObject`) and cannot destroy them (no `DeleteObject`); combined with bucket versioning, even an accidental overwrite leaves the prior version recoverable. Restore is a deliberate, human-driven action performed from a workstation using a separate, broader AWS identity — the recovery path in 24.4 pulls objects with that operator's credentials, never the cluster's write-only role.

24.4 Two Recovery Paths — App/Config from Git (Argo CD), Data from S3

A full recovery is the composition of two independent paths that can proceed in parallel (Figure 24.2). The application and configuration come from Git, and the data comes from S3 — nothing in the cluster that matters is outside one of those two sources.

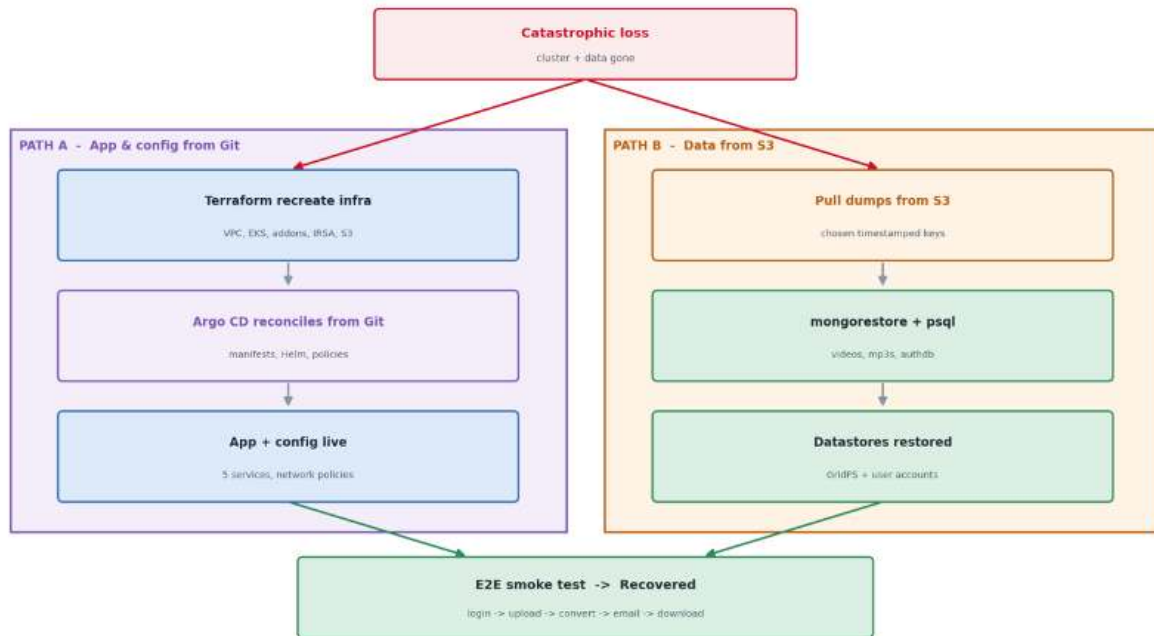


Figure 24.2 — The two recovery paths converging on an end-to-end smoke test: app/config reconciled from Git by Argo CD, data restored from the S3 dumps.

Path A — app and config from Git. Terraform recreates the infrastructure (VPC, EKS, the EBS CSI and other addons, IRSA roles, the S3 bucket), and Argo CD then reconciles the cluster back to the committed manifests, Helm releases and policies. This is the same Git-as-truth model used in normal operation, so the recovered cluster is identical to the one in version control.

Path B — data from S3. The operator selects the timestamped keys to restore (`aws s3 ls ... | sort | tail`), pulls the archives, and restores them into the running datastores: `mongorestore --gzip --archive with --nsInclude` for the videos and mp3s databases (authenticating with the app URIs from gateway-secret, exactly as the backup did), and the gunzipped SQL piped into `psql` for authdb. Restoring into a fresh gp3 PVC works because the dump itself recreates `auth_user`, and the EBS volume survives PVC deletion thanks to `reclaimPolicy: Retain`. Integrity is then verified — row counts and a `bcrypt`-prefixed (\$2a\$/2b\$) admin hash in Postgres, non-zero GridFS file counts in Mongo — before the paths converge on the end-to-end smoke test (login → upload → convert → email → download). Recovery is declared complete only when that test passes.

24.5 Tested Runbook and RTO/RPO Targets (< 2-Hour RTO)

The recovery is a written, tested runbook, not a theoretical plan. Its objectives and the latest drill result are summarised below.

Objective	Target	Basis / tested result
RPO (max data loss)	≤ 24 hours	Nightly backup cadence; tighten with a midday run if needed
RTO (time to restore)	< 2 hours	Re-apply infra + restore dumps + E2E verify, with margin
Tested end-to-end	~45 minutes	Sum of the timeline steps in Figure 24.3

Last Postgres restore drill	2026-06-10	pg_dump → fresh EBS PVC → restore → login E2E passed
MongoDB backup	Verified	Valid archives in S3 (videos 38 MB, mp3s 7 MB); full mongorestore drill outstanding

Table 24.3 — RTO/RPO objectives and the most recent drill results (from the DR runbook).

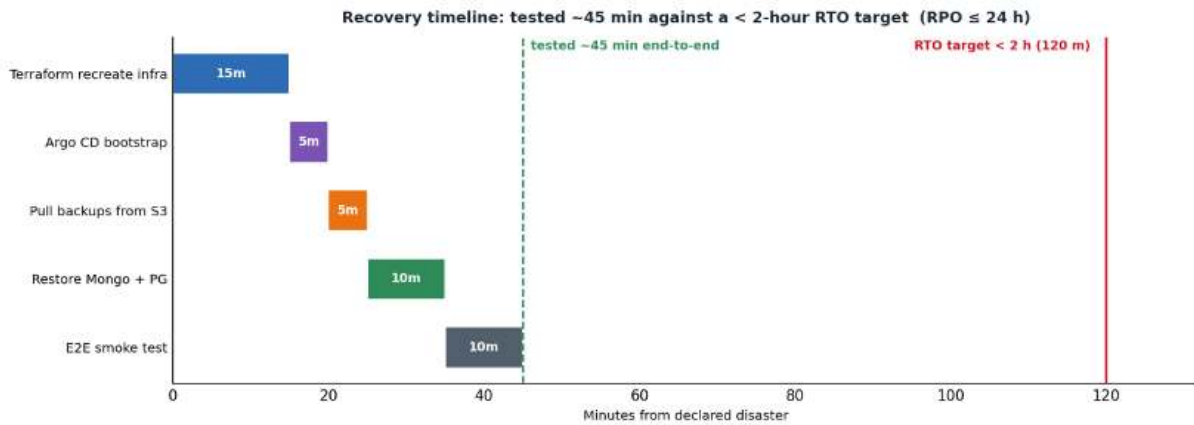


Figure 24.3 — Recovery timeline: the runbook's steps sum to roughly 45 minutes, comfortably inside the < 2-hour RTO target, against an RPO of ≤ 24 hours.

The drill itself is part of the runbook: both backups can be triggered on demand (`kubectl create job --from=cronjob/...`), the restore performed in a non-prod namespace or a disposable re-apply, the wall-clock time recorded, and the "Last restore test" date updated. The runbook is explicit that a plan which has drifted from reality is worse than none, which is why the drill date is tracked at the top of the document. Two follow-ups are recorded honestly rather than assumed done: a backup-freshness alert (a PrometheusRule that fires if no successful backup Job completed in the last 25 hours) and the still-outstanding full mongorestore drill.

24.6 Merge Runbooks (bcrypt + RBAC Coordinated Deployment)

Most deployments are routine, but one change class is not: the bcrypt password hashing and the RBAC role model must land together. The hazard is a forward-only migration — if the new bcrypt auth image rolls while live PostgreSQL still holds the old plaintext rows, `bcrypt.checkpw` cannot verify against a non-hash value and every login fails. (After hardening, a malformed stored hash returns 401 rather than 500, but it is still a failed login until the database is migrated.) The team therefore runs a dedicated merge-time runbook, executed with the operator at the moment the branch merges and CI builds the new image. It contains no credentials — the Postgres password is read from the environment, sourced from the gitignored deployment config.

1. **Migrate the schema (idempotent, additive)** — `ALTER TABLE ... ADD COLUMN IF NOT EXISTS` for role and created_at, and add the unique email constraint if absent.
2. **Re-seed the admins with bcrypt hashes** — run `init.sql` (`CREATE TABLE IF NOT EXISTS + ON CONFLICT (email) DO UPDATE`), which refreshes only the seeded admins' role and bcrypt hash and leaves self-registered user rows untouched.
3. **Verify the seed** — confirm the admin emails show role=admin with a \$2b\$12\$ password prefix.
4. **Roll the auth image** — `kubectl rollout status deployment/auth` (CD normally does this on merge).

5. **Smoke test the positive path** — an admin login returns a token carrying admin:true and role:admin.
6. **Smoke test the negative path** — a fresh sign-up returns admin:false and role:user — self-registration can never mint an admin.

The runbook also defines rollback: `kubectl rollout undo deployment/auth` restores the previous (plaintext) auth image, which matches the pre-migration database, and re-running `init.sql` is always safe because of its `ON CONFLICT` clause. The deeper operational rule, recorded as a lesson, is that post-migration there is no clean revert — the old plaintext image cannot read bcrypt hashes — so the image and its schema/seed migration are treated as a single atomic change and "rollback" after migration means fix-forward, not revert. This is the business-continuity counterpart to the disaster-recovery work above: the data is protected by backups, and the high-risk change that touches that data is protected by a rehearsed, reversible-where-possible runbook.

Part VI — Well-Architected Review, Cost & Validation

Chapter 25 — AWS Well-Architected Review

The AWS Well-Architected Framework is the industry’s standard lens for judging a cloud workload, organised into six pillars: Operational Excellence, Security, Reliability, Performance Efficiency, Cost Optimization, and Sustainability. This chapter reviews VidCast against each pillar, mapping the platform’s concrete implementations — most of which are detailed in earlier chapters — onto the framework’s design principles. The review doubles as a synthesis: it shows that the individual engineering decisions made throughout the project are not ad-hoc but cohere into a recognisably well-architected whole.

Figure 25.1 — VidCast Mapped to the Six Well-Architected Pillars

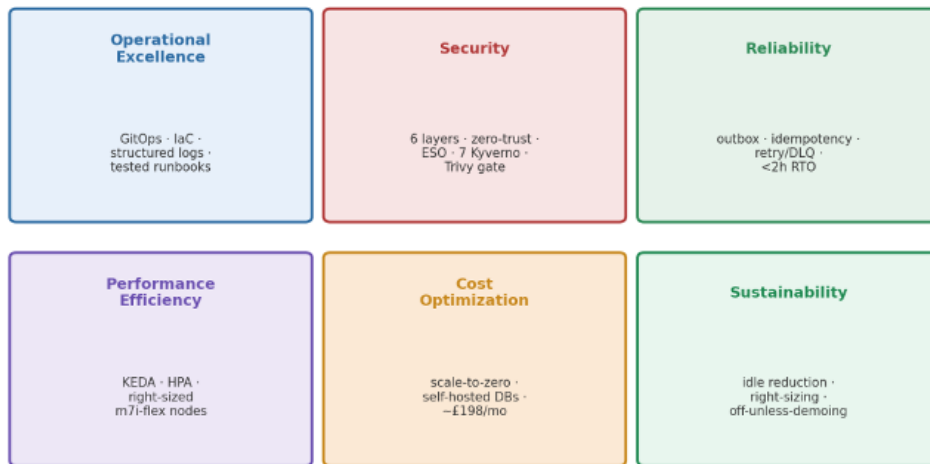


Figure 25.1 — VidCast mapped to the six Well-Architected pillars at a glance. Each pillar is examined in detail in the sections that follow (and built out in the cited chapters).

The review is conducted honestly: where a pillar’s design principle is only **partially** met because of the platform’s deliberate single-node, cost-capped, demonstration scope, that gap is named rather than glossed over. The radar below summarises the self-assessed breadth of coverage per pillar; the high scores reflect the number of the framework’s practices actually implemented, and the shortfalls (notably multi-region HA under Reliability) are the documented scope limits from Chapter 3 §3.6.

Figure 25.2 — Self-Assessed Pillar Coverage (demo-platform scope)

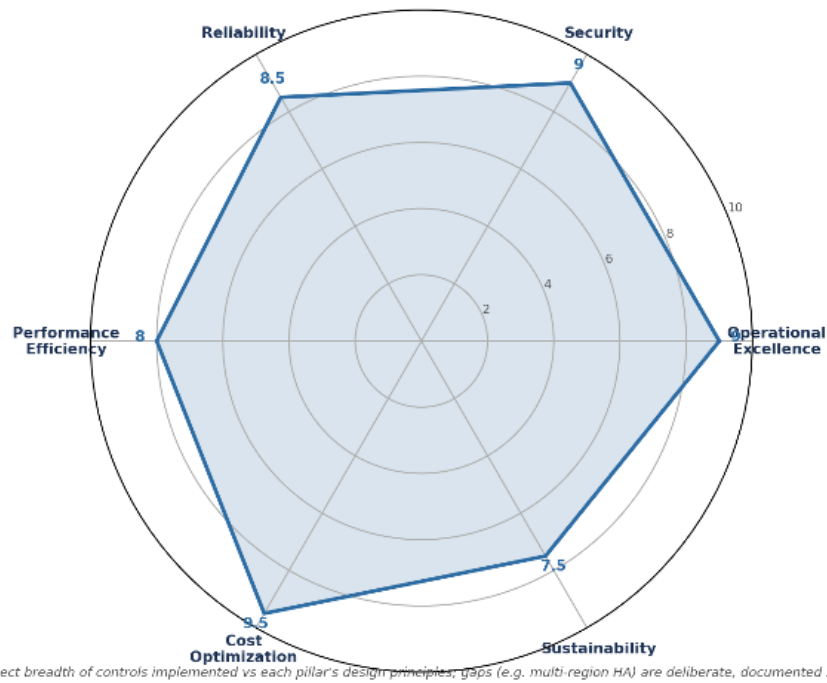


Figure 25.2 — Self-assessed coverage per pillar. Strong across the board for a demo platform; the deliberate gaps are scope decisions, not oversights.

25.1 Operational Excellence (GitOps, IaC, Structured Logs, Tested Runbooks)

Design focus: run and monitor systems to deliver business value, and continually improve supporting processes and procedures. VidCast scores strongly here because **everything is codified and nothing lives in someone's terminal history**.

WA principle	VidCast implementation	Chapter
Perform operations as code	100% Terraform IaC + GitOps via Argo CD; no manual kubectl to prod	9, 12
Make frequent, small, reversible changes	CI per-commit; deploys are git commits; rollback = <code>git revert</code>	11, 12
Anticipate failure	tested DR runbook (<2h RTO); merge runbooks for risky changes	24
Learn from operations	structured JSON logs + correlation IDs; SLO burn-rate review	21, 22
Refine procedures	decisions + runbooks tracked in Git (DECISIONS_MADE, GITOPS, DR)	—

Table 25.1 — Operational Excellence: principle → implementation.

The defining property is that the platform is **operable by a two-person team** precisely because operations are code: Git is the single source of truth (Argo CD reconciles the cluster to it), structured `jsonlog.py` output carries a gateway-minted correlation ID through every hop so "where is my file?" is answerable from logs,

and the runbooks (deployment, disaster recovery, RBAC merge) are tracked artefacts, not tribal knowledge. This is Operational Excellence by construction.

25.2 Security (Six Defence-in-Depth Layers, Zero-Trust, ESO, Kyverno, Trivy Gate)

Design focus: protect data, systems and assets, applying defence in depth and least privilege. Security is the most thoroughly engineered pillar, organised as the six-layer model of Part IV (Chapters 14–19).

WA principle	VidCast implementation	Chapter
Strong identity foundation	bcrypt-12, JWT/RBAC + guardrails, IRSA (keyless), OIDC for CI/CD	16, 9
Apply security at all layers	6 defence-in-depth layers (perimeter→network→identity→container→supply-chain→policy)	14
Protect data in transit & at rest	ACM TLS at the ALB; AES256 on S3/ECR; SCRAM Postgres	15, 7
Keep people away from data	content stays in-account; admin via port-forward, not public NodePort	15
Automate security best practice	7 Kyverno ClusterPolicies at admission; Trivy CVE gate (shift-left)	19, 18
Protect secrets	External Secrets Operator → Parameter Store; 0 secrets in Git	16

Table 25.2 — Security: principle → implementation.

Zero-trust is realised in both directions — the ALB is the only public door and every pod denies all traffic except explicit allow-listed flows (Chapter 15) — and least privilege runs through every identity (scoped IRSA roles, RBAC, the CD deploy role limited to `eks:DescribeCluster` + an EKS access entry). The supply chain is gated shift-left by Trivy (a fixable CRITICAL/HIGH fails the build before an image ships) and policed at admission by Kyverno. The whole posture is built on the blast-radius philosophy: assume a pod will be compromised, and ensure it can reach nothing it was not granted.

25.3 Reliability (Outbox, Idempotency, Retry/DLQ, < 2h RTO)

Design focus: ensure a workload performs its intended function correctly and consistently, and recovers from failure. Reliability is engineered in application code rather than bought from managed infrastructure (Chapter 8).

WA principle	VidCast implementation	Chapter
Automatically recover from failure	health probes + graceful degradation; KEDA/HPA self-heal capacity	8, 13
Test recovery procedures	tested DR runbook with a dated drill; <2h RTO	24
Scale horizontally to reduce blast radius	multiple replicas; per-pod failure isolation	13, 14
Stop guessing capacity	queue-depth & CPU autoscaling; scheduler-visible requests	13
Manage change in automation	GitOps + CI/CD; no manual drift	11, 12

Table 25.3 — Reliability: principle → implementation.

The three reliability patterns are the heart of this pillar: the **transactional outbox (A1)** guarantees an accepted upload is never silently lost; **idempotent consumers (A2)** make at-least-once delivery effectively-once so a redelivery never duplicates work; and **bounded retry with dead-letter queues (A3)** turns failures into a recoverable, observable process rather than a dropped message or an infinite loop. Recovery is **tested**, not assumed — the disaster-recovery runbook recovers app + config from Git and data from S3 within a **tested sub-2-hour RTO**.

The honest gap. The one Well-Architected practice VidCast does not fully meet here is **multi-region/multi-AZ resilience**: it runs on a single node in a single region (eu-west-2), so the node and region are single points of failure. This is a deliberate, documented scope decision (Chapter 3 §3.6) — multi-region HA would need cross-region RDS replicas, Route 53 failover, and a DocumentDB global cluster, all cost-prohibitive for a demo. The upgrade path is documented so the platform can grow into it.

25.4 Performance Efficiency (KEDA Event-Driven Scaling, HPA, Right-Sized Nodes)

Design focus: use computing resources efficiently to meet requirements, and maintain efficiency as demand changes. VidCast matches the resource to the workload shape (Chapter 13).

WA principle	VidCast implementation	Chapter
Democratise advanced tech	managed EKS; KEDA/HPA do the scaling maths	9, 13
Go global in minutes	N/A by scope (single-region demo) — documented	3
Use serverless-style elasticity	KEDA event-driven scale-to-zero on the converter	13
Experiment more often	reproducible IaC lets the whole stack rebuild in 20 min	9, 26
Mechanical sympathy	right signal per workload: queue-depth for async, CPU for sync	13

Table 25.4 — Performance Efficiency: principle → implementation.

The pillar's "mechanical sympathy" principle — choosing the technology that best fits how a workload behaves — is exactly what drives the dual autoscaler design: **KEDA** scales the bursty, async converter on RabbitMQ queue depth (the leading indicator of pending work, scaling to zero when idle), while a **CPU HPA** scales the synchronous, always-on gateway. The compute is **right-sized m7i-flex.large** nodes whose resource requests are tuned so the scheduler's view matches reality. The result is performance that tracks demand precisely instead of over-provisioning for peak.

25.5 Cost Optimization (Scale-to-Zero, Self-Hosted Datastores, Tear-Down)

Design focus: run systems to deliver business value at the lowest price point. This is covered in full in Chapter 26; here it is mapped to the framework.

WA principle	VidCast implementation	Chapter
Adopt a consumption model	KEDA scale-to-zero — compute paid per conversion (pence)	26, 13
Measure overall efficiency	Kubecost FinOps dashboard; cost-per-conversion KPI	26
Stop spending on undifferentiated heavy lifting	managed EKS control plane; AWS-managed TLS (ACM)	9, 15

Analyze and attribute expenditure	Kubecost per-namespace/service attribution	26
Use cost-effective resources	self-hosted datastores (£0) vs ~\$262–273/mo managed; off-when-idle	26, 7

Table 25.5 — Cost Optimization: principle → implementation.

VidCast runs at **~£198/month — 94% under the £3,500 cap** — and ~£0 when torn down, a >40× return against the ~£100k/year of producer time reclaimed. The levers are the framework’s own: a consumption model (scale-to-zero), cost-effective resource choices (self-hosted datastores avoiding the ~\$183/month Amazon MQ floor), and measurement/attribution via Kubecost. The off-unless-demoing lifecycle — destroy to ~£0, rebuild in 20 minutes — is consumption pricing taken to its logical end.

25.6 Sustainability (Resource Right-Sizing, Idle Reduction)

Design focus: minimise the environmental impact of running cloud workloads — chiefly by maximising utilisation and minimising the resources provisioned. Sustainability is the newest pillar, and VidCast addresses it as the environmental flip-side of its cost work: the same decisions that cut the bill also cut idle compute.

WA principle	VidCast implementation	Chapter
Maximize utilization	idle utilisation tuned 93%→82%; one node packs the whole stack	26
Reduce downstream impact / idle	KEDA scale-to-zero removes the heavy converter when idle	13, 26
Anticipate and adopt efficient hardware	m7i-flex (efficient current-gen Intel) right-sized to need	9, 26
Use managed services to share footprint	managed EKS + ACM share AWS’s amortised infrastructure	9
Scale infrastructure to match demand	autoscaling + off-unless-demoing — provision only when needed	13, 26

Table 25.6 — Sustainability: principle → implementation.

The core sustainability lever is **eliminating idle compute**. The converter — the heaviest workload — runs only ~40 minutes a week and sits at zero replicas otherwise, so its energy footprint is proportional to actual conversions rather than to reserved capacity. Right-sizing packs the entire platform onto a single, well-utilised node instead of an over-provisioned fleet, and the off-unless-demoing posture means the cluster consumes nothing at all when not in use. Provisioning to demand, not to peak, is the framework’s central sustainability instruction — and it falls out naturally from the same engineering that delivers the cost result.

In summary, VidCast maps cleanly onto all six Well-Architected pillars, and — importantly — the mapping is not retrofitted: the same engineering decisions serve multiple pillars at once. GitOps and IaC deliver Operational Excellence *and* the reproducibility behind Cost and Sustainability; KEDA scale-to-zero serves Performance Efficiency, Cost, *and* Sustainability simultaneously; the reliability patterns serve Reliability while the zero-trust and least-privilege controls serve Security. The platform is strong across every pillar for its scope, with the principal honest gap — single-region/single-node resilience under Reliability — being a

deliberate, documented, cost-driven scope decision with a clear upgrade path. By the framework's own standard, VidCast is a coherent, well-architected workload: secure, reliable, efficient, cost-optimised, sustainable, and operable by a small team.

Chapter 26 — Cost Engineering and FinOps

A platform built to retire a six-figure operational drag (Chapters 2–4) only makes sense if it costs a small fraction of the problem it solves. This chapter shows that it does. VidCast runs at roughly **£198/month** — about 5–6% of the agreed £3,500/month cap — and drops to **~£0 when torn down**. That number is not luck; it is the product of deliberate FinOps engineering: event-driven scale-to-zero, self-hosted datastores, right-sized nodes, an off-unless-demoing lifecycle, and continuous cost visibility through Kubecost. This chapter breaks the bill down resource by resource (§26.1) and then explains each engineering decision that keeps it low (§26.2–§26.7).

The headline. ~£198/month all-in running cost · 94% headroom under the £3,500 cap · ~£0 when torn down (20-minute rebuild) · ~£2,400/year to run versus ~£100k/year of reclaimed producer time — a **>40× return on running spend**.

26.1 Monthly Cost Breakdown, Resource by Resource (~£198/month, eu-west-2)

Every AWS resource the platform consumes in eu-west-2 (London), and what each buys:

Resource	~Monthly	What it provides
2× m7i-flex.large nodes	£114	right-sized compute; ~5% flex discount
EKS control plane	£60	managed Kubernetes — no etcd/API-server/cert ops
AWS ALB (ingress)	£22	HTTPS termination, ACM, health-check routing
Route 53 (DNS)	£1	the vidcast.online hosted zone
S3 backups	£1	nightly compressed dumps, 30-day lifecycle
TOTAL	≈ £198	94% under the £3,500 cap

Table 26.1 — Monthly cost breakdown (pitch deck slide 25; eu-west-2 list pricing).

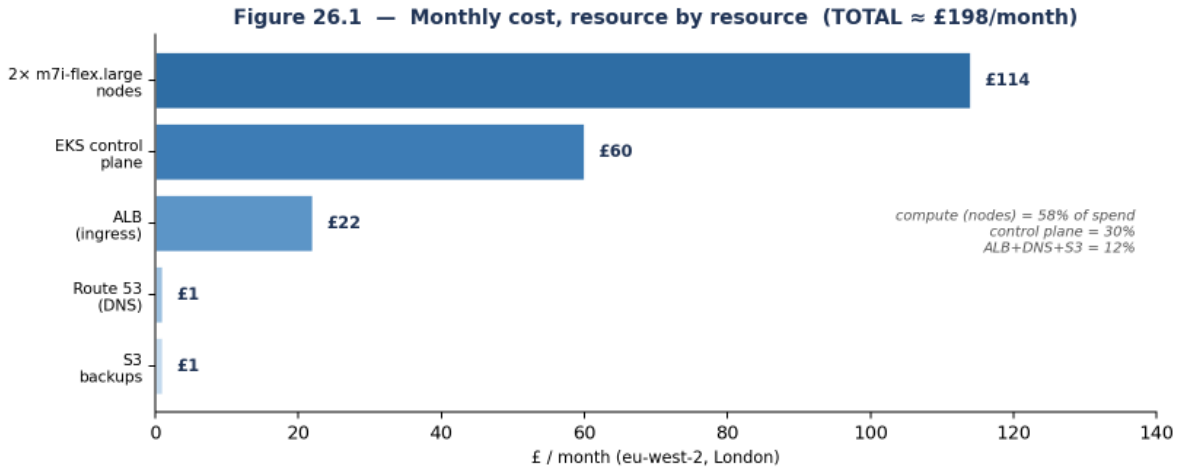


Figure 26.1 — Where the money goes. Compute (the two nodes) is 58% of spend and the EKS control plane 30%; the ALB, DNS and S3 together are the remaining ~12%.

The shape of the bill is the key insight for FinOps: **88% of cost is compute and the managed control plane**, both of which scale with the cluster being *on*. The networking and storage line items (ALB, DNS, S3) are negligible and largely fixed. This is why the dominant savings levers (\$26.2–\$26.5) all target compute — keeping pods off when idle, packing them onto a single right-sized node, and turning the whole cluster off between demos.

26.2 KEDA Scale-to-Zero — Compute Paid per Conversion

The converter is the single most expensive workload to run (250m CPU request each; Chapter 13), and it is *bursty* — it does real work only when a video is queued. Provisioning it to be always-on would burn the node's most expensive resource doing nothing for the vast majority of the week. KEDA solves this by scaling the converter on RabbitMQ **video**-queue depth, all the way down to **zero replicas when idle**.

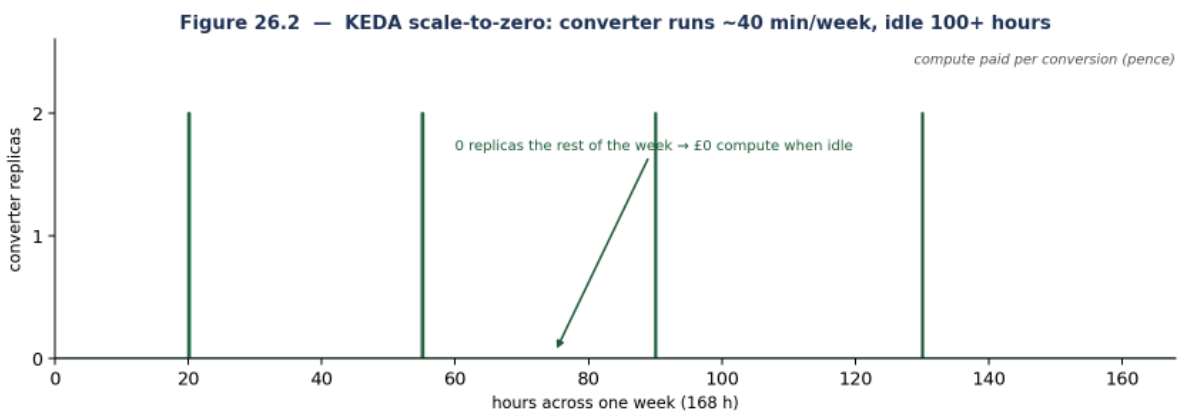


Figure 26.2 — The converter's weekly duty cycle. It runs only ~40 minutes a week in short bursts and sits at 0 replicas for 100+ idle hours — so compute is paid per conversion, in pence.

The effect is dramatic: as the deck records, the **converter runs ~40 minutes per week and is at zero pods for 100+ idle hours**. Compute for the heavy conversion tier is therefore "paid per conversion" — effectively pence — rather than rented continuously. This is the cloud-native cost model in its purest form: you pay for

work done, not capacity reserved. The mechanism (the `ScaledObject`, `minReplicaCount: 0`, the 60-second cooldown) is detailed in Chapter 13; here it is the largest **variable** cost lever on the platform.

26.3 Self-Hosted Datastores vs Managed (Amazon MQ Floor ~£183/mo)

The biggest **avoided** cost is not on the bill at all: VidCast runs MongoDB, PostgreSQL, RabbitMQ and Redis **self-hosted in-cluster**, sharing the existing node at effectively £0, rather than on AWS-managed equivalents. The decision was made by costing the managed path honestly (`docs/MANAGED_SERVICES.md`, summarised in Chapter 7 §7.6) and finding it cost-prohibitive for a portfolio cluster.

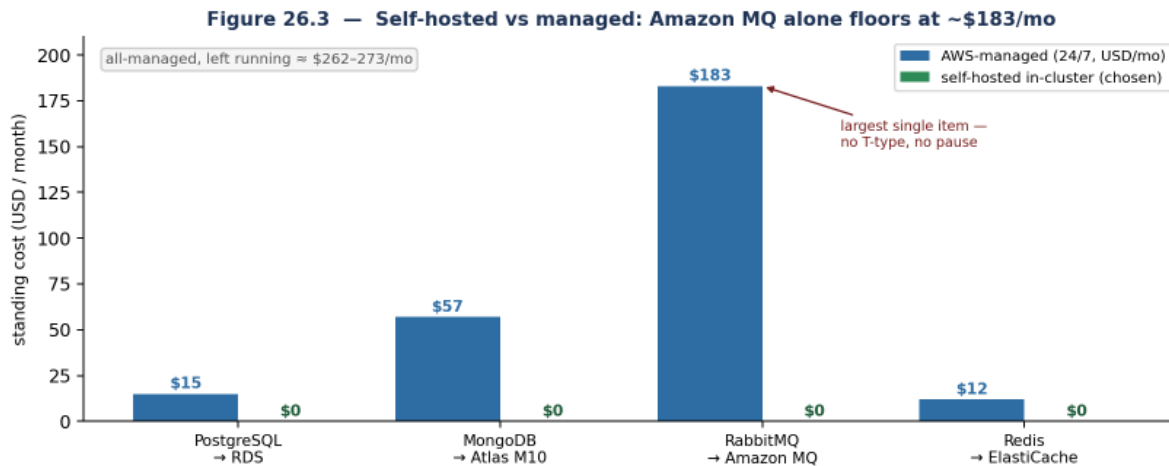


Figure 26.3 — Managed-datastore standing cost vs the self-hosted (£0) choice. Amazon MQ for RabbitMQ alone floors at ~\$183/month — more than the entire VidCast bill.

The decisive figure is **Amazon MQ for RabbitMQ**, whose smallest broker (`mq.m5.large`) floors at roughly **\$183/month** — there is no T-type and no pause. That one managed service alone would cost more than the entire £198/month platform. The full managed stack would run ~\$262–273/month standing:

Datastore	Managed option	~Standing cost	VidCast choice
PostgreSQL	RDS db.t3.micro	~\$15/mo	self-hosted (£0)
MongoDB + GridFS	Atlas M10	~\$57/mo	self-hosted (£0)
RabbitMQ	Amazon MQ mq.m5.large	~\$183/mo	self-hosted (£0)
Redis	ElastiCache cache.t3.micro	~\$12/mo	self-hosted (£0)
All-managed, left running	—	~\$262–273/mo	— (not adopted)

Table 26.2 — Managed-datastore standing costs vs the self-hosted choice (`docs/MANAGED_SERVICES.md`).

Crucially, this is a **cost** trade-off, not a reliability compromise: the durability properties teams usually buy managed datastores for (no lost events, idempotent retries, dead-lettering) are delivered in **application code** by the A1/A2/A3 reliability patterns (Chapter 8) against the in-cluster brokers. The managed migration path is documented and costed so it can be adopted when real traffic justifies it — but for now it keeps the standing bill at £0 for the entire stateful tier.

26.4 Right-Sized Nodes and Idle-Utilisation Tuning (93% → 82%)

Running everything on one node only works if the node is correctly sized and the workloads are packed sensibly. The platform uses **two m7i-flex.large** instances (2 vCPU / 8 GB), an M-series choice that gives a ~5% "flex" discount over standard on-demand while satisfying the SCP that blocks burstable T-types (Chapter 9). The harder work was tuning the **idle utilisation** so the node had headroom to schedule new pods without falling over.

Through deliberate tuning — adjusting resource **requests** so the scheduler's view matched reality, and capping the converter's KEDA **maxReplicas** so it could never oversubscribe the node — idle utilisation was trimmed from **93% down to 82%**. That ~11-point reduction is what created the headroom to add the platform add-ons (KEDA, Kyverno, Kubecost, monitoring) onto the same node without a costly second node group.

Lever	Action	Effect
Instance family	m7i-flex.large (M-series)	~5% flex discount; T-type blocked by SCP
Resource requests	tuned to real usage	scheduler sees true demand; better bin-packing
KEDA maxReplicas	capped to fit the node	converter can't oversubscribe CPU
Idle utilisation	93% → 82%	headroom for add-ons without a 2nd node

Table 26.3 — Right-sizing and utilisation-tuning levers (pitch deck slide 26).

Why utilisation matters for cost. On a fixed-size node, lower idle utilisation does not reduce the bill directly — the node costs the same. What it buys is the ability to fit **more** onto the **same** node, avoiding the step-change cost of a second node group. Right-sizing is therefore a cost-**avoidance** lever: it kept the compute line at £114 instead of doubling it.

26.5 Off-Unless-Demoing (terraform destroy → ~£0, Rebuild in 20 min)

The most aggressive cost lever exploits the fact that the entire platform is declarative (Chapter 9). Because 100% of the infrastructure is Terraform and 100% of the application is GitOps, the cluster can be **destroyed when not in use and rebuilt on demand** — **terraform destroy** takes the standing cost to ~£0, and **terraform apply** rebuilds the whole platform in about **20 minutes**.

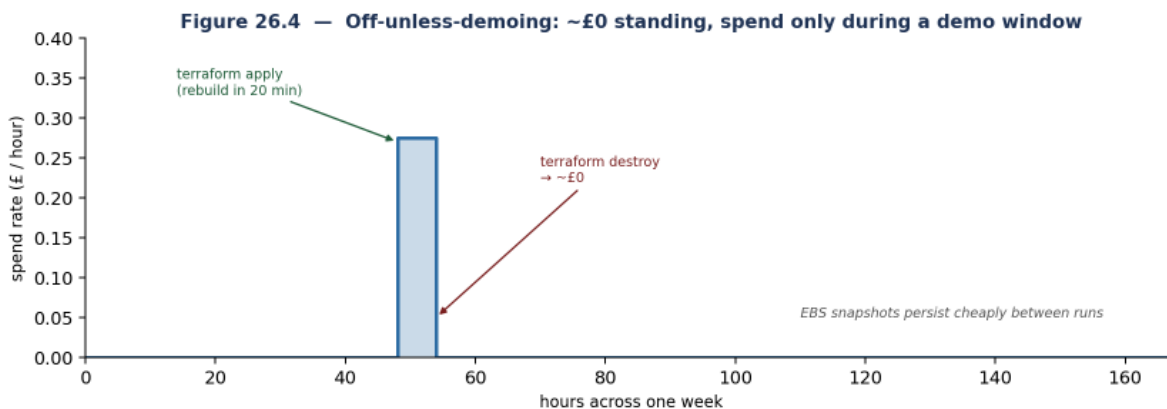


Figure 26.4 — The off-unless-demoing lifecycle. Standing cost is ~£0; the cluster is applied for a demo window and destroyed afterwards, with EBS snapshots persisting cheaply in between.

This turns the cost model from "monthly rent" into "pay only when demonstrating." Between runs, only cheap, persistent artefacts remain — EBS snapshots and S3 backups — at pennies. The 20-minute rebuild is fast enough that the inconvenience is negligible, and the deploy is fully scripted (Chapter 9 §9.1), so standing up the platform for a demo is a single command, not a day's work. For a portfolio/demonstration platform this is the single biggest saving of all: the realistic **average** monthly cost is far below even the £198 on-figure, because the cluster is off most of the time.

26.6 Kubecost FinOps Dashboard and Cost-per-Conversion

Cost engineering needs measurement, and that is the FinOps "B3" work: **Kubecost** (OSS / OpenCost core, no license key) provides per-namespace, per-service, and — the headline metric — **per-conversion** cost visibility, surfaced through a Grafana FinOps dashboard.

The node-budget tuning that makes it fit

Kubecost is normally a heavy add-on: by default it stands up its **own** Prometheus, node-exporter and kube-state-metrics — about 1 CPU, duplicating the kube-prometheus-stack already running (Chapter 20). The `values.yaml` disables all of that and points Kubecost at the **existing** Prometheus, reducing it to a single ~175m cost-analyzer pod. Even so, it pushes the prod footprint over the 90% idle gate, so it is run against the dev (1-replica) footprint or **scaled to zero between cost-analysis sessions** (`kubect1 scale deploy/kubecost-cost-analyzer --replicas=0`) — a neat example of applying the platform's own cost discipline to the cost tool itself.

The FinOps dashboard

The `vidcast-finops.json` Grafana dashboard turns the node cost model (m7i-flex.large ≈ \$0.106/hr) into operational metrics:

Panel	Metric / formula	Purpose
Cluster cost (\$/hr)	<code>sum(node_total_hourly_cost)</code>	live hourly burn
Projected monthly cost	<code>sum(node_total_hourly_cost) * 730</code>	run-rate projection
★ Cost per conversion	cluster \$/hr ÷ conversions/hr	unit economics — the headline FinOps KPI
Conversions / hour	<code>rate(vidcast_conversions_total{successes})</code>	throughput denominator
Cost by namespace	CPU-share approximation	where spend is attributed

Table 26.4 — FinOps dashboard panels (`monitoring/dashboards/vidcast-finops.json`).

Estimate vs ground truth. The project is explicit that Kubecost **estimates** from instance list pricing and **AWS Cost Explorer is ground truth** — Kubecost does not see Reserved Instances/Savings Plans, data-transfer, or control-plane charges unless configured. The dashboard is for **engineering** cost decisions (which workload, which conversion costs what); the real bill is reconciled against Cost Explorer. Naming this distinction is good FinOps hygiene.

26.7 Budget Headroom (94%) and Parameter Store vs Secrets Manager Savings

94% budget headroom

Against the agreed **£3,500/month cap** — roughly the cost of one extra headcount — the platform spends ~£198/month, leaving **~94% headroom**. This is not just a vanity statistic: it is deliberate slack to absorb growth (more nodes, more traffic, the managed-datastore migration of \$26.3) without a budget conversation. The platform can grow several-fold before it approaches the ceiling.

Measure	Value
Monthly cap	£3,500 (~one headcount)
Actual run cost	~ £198/month (~5.7% of cap)
Headroom	~ 94%
Cost when torn down	~ £0
Annual run cost vs reclaimed value	~ £2.4k/yr vs ~ £100k/yr (>40×)

Table 26.5 — Budget position against the cap.

Parameter Store vs Secrets Manager — a deliberate £0 choice

A smaller but characteristic saving sits in the secrets layer (A9). VidCast reads its secrets from **AWS SSM Parameter Store**, not Secrets Manager, specifically to avoid Secrets Manager's **\$0.40 per secret per month** charge. Standard-tier SSM parameters are free, and SecureString uses the AWS-managed `alias/aws/ssm` KMS key (also free) — so the External Secrets Operator's standing cost is **\$0** rather than the few dollars a month Secrets Manager would add for the handful of secrets the platform uses.

Option	Per-secret cost	VidCast
AWS Secrets Manager	\$0.40 / secret / month + API calls	not used
SSM Parameter Store (standard)	\$0 (free tier) + free SSM KMS key	chosen — \$0 standing cost

Table 26.6 — Parameter Store vs Secrets Manager: the same capability at \$0 (Chapter 9 §9.8).

On its own this saves only a few dollars a month, but it exemplifies the project's whole FinOps posture: at every decision point, the cheaper option that meets the requirement is chosen and documented — the ECR AES256 key over a customer-managed KMS key, the in-cluster datastores over managed, Parameter Store over Secrets Manager. None of these individually moves the needle much; together they are why a production-grade platform runs at £198/month.

In summary, VidCast's ~£198/month bill is engineered, not incidental. Compute and the managed control plane are 88% of spend, so every major lever targets compute: KEDA scales the bursty converter to zero (it runs ~40 min/week, paid per conversion in pence); the entire stateful tier is self-hosted at £0, avoiding the ~\$183/month Amazon MQ floor and a ~\$262–273/month all-managed stack; the single right-sized m7i-flex node is tuned from 93% to 82% idle utilisation so add-ons fit without a second node; and because everything is declarative, the cluster is destroyed to ~£0 when idle and rebuilt in 20 minutes. Kubecost provides per-conversion cost visibility (itself scaled to zero between sessions), and small deliberate choices — Parameter Store over Secrets Manager, AES256 over CMK — round out a disciplined posture that leaves 94% headroom

under the £3,500 cap and delivers a >40× return on running spend against the ~£100k/year of producer time the platform reclaims.

Chapter 27 — Validation and Results

A platform is only as credible as its evidence. This chapter sets out how VidCast was validated and what the results were — measured against the objectives (Chapter 3) and the formal SLOs (Chapter 22), on the live deployment at **vidcast.online**. It covers the test strategy and, candidly, where its coverage falls short (§27.1); the production validation against each SLO target (§27.2); the end-to-end smoke test that proves the full journey works (§27.3); the headline availability, latency, error-rate and disaster-recovery results (§27.4); and the security-scan and policy-compliance outcomes (§27.5). The guiding principle, taken from the project itself, is that the results are *"verified against targets — live, not theoretical."*

27.1 Test Strategy (Lint, Trivy Scan, Smoke Test) and Coverage Gaps

VidCast's quality assurance is built around **automated gates in the pipeline** plus **runtime validation via SLOs**, rather than a large unit-test suite. The gates that run on every change are pragmatic and fast, and each fails the build/deploy if it does not pass.

Figure 27.1 — Validation Strategy: gates that run vs the coverage gap

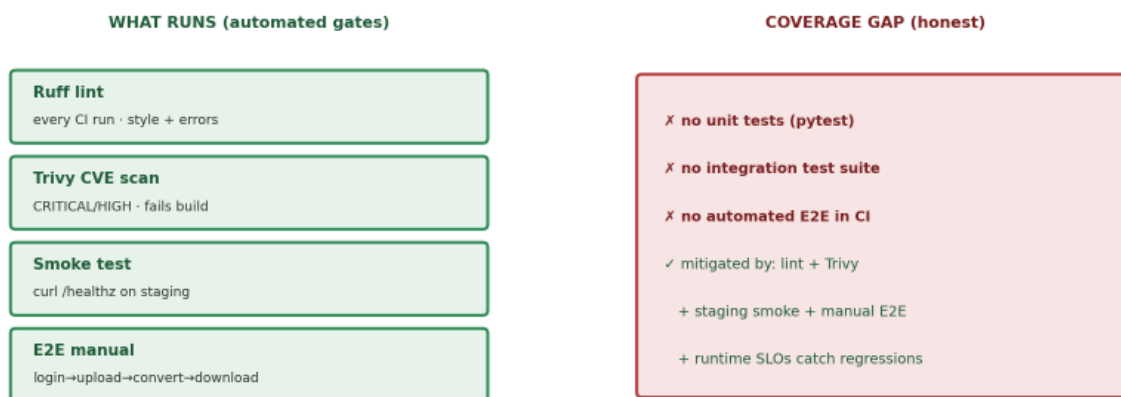


Figure 27.1 — The validation strategy: the automated and manual gates that run (left) and the honestly-acknowledged coverage gap (right), with the mitigations that compensate for it.

Gate	When it runs	What it catches
Ruff lint	every CI run (push/PR)	Python style + error-class issues before any build
Trivy CVE scan	every image build	fixable CRITICAL/HIGH vulnerabilities (fails the build)
Smoke test	Jenkins staging deploy	a non-200 from /healthz fails the pipeline
Manual E2E	before a release / demo	the full login→upload→convert→download journey
Runtime SLOs	continuously in prod	regressions in availability, latency, e2e success

Table 27.1 — The validation gates and what each covers.

Coverage gaps — stated honestly

The project is candid that this strategy has a real gap: **there is no unit-test suite (pytest), no integration-test suite, and no automated end-to-end test in CI**. This is acknowledged rather than hidden, and it is mitigated by a layered set of compensating controls:

- **Static + supply-chain gates** (ruff, Trivy) catch a large class of defects and all fixable high-severity CVEs before code ships.
- **Staging smoke test** (Jenkins) verifies the deployed app responds before production is offered.
- **Manual E2E** exercises the whole user journey on a real environment before a release.
- **Runtime SLOs with burn-rate alerting** (Chapter 22) act as continuous production tests — a regression in availability, latency or end-to-end success trips an alert.

For a single-engineer-operable demonstration platform this is a defensible balance, but a production hardening step would add unit/integration tests and an automated E2E stage in CI. Naming the gap explicitly is itself a quality signal.

27.2 Production Validation Against SLO Targets (Live at vidcast.online)

The platform is **live at vidcast.online** and was validated end-to-end against its SLOs and runbooks — the results are from the running system, not a model. Every defined target was met.

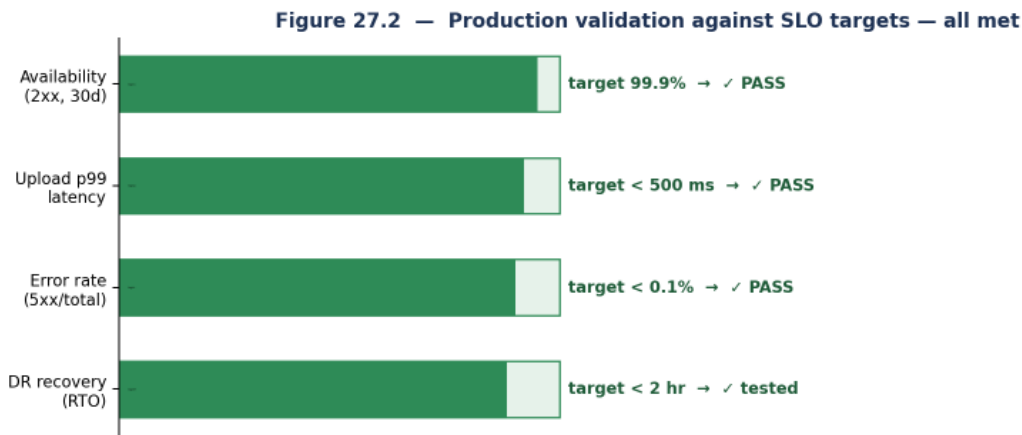


Figure 27.2 — Production validation against the SLO targets. All four headline targets — availability, upload latency, error rate, and DR recovery — are met.

Validation	Target	Result	How measured
Availability SLO	99.9%	PASS	2xx health checks over a 30-day window
Upload p99 latency	< 500 ms	PASS	excludes upload body transfer
Error rate	< 0.1%	PASS	5xx ÷ total requests
DR recovery	< 2 hr RTO	PASS (tested)	tested and dated in the runbook
E2E smoke test	PASS	PASS	login → upload → convert → download
Trivy CVE gate	0 shipped	PASS	no fixable CRITICAL/HIGH reaches the registry

Batch upload	up to 20 files	PASS	KEDA absorbs the burst, no infra change
KEDA cold start	0→1 in < 10 s	PASS	converter scales on first queued message

Table 27.2 — Production validation results (pitch deck slide 27, "Verified Against Targets").

These eight validations span the full breadth of the project's claims: the three core SLOs (availability, latency, error rate), the tested disaster-recovery objective, the end-to-end functional journey, the security supply-chain gate, and the elasticity claims (batch absorption and cold-start). Together they substantiate the headline that this is a production-grade platform, not a prototype.

27.3 End-to-End Smoke Test — Login → Upload → Convert → Download

The single most important functional validation is the end-to-end smoke test, which exercises the entire user journey exactly as a producer would experience it. It **PASSES**: a user logs in, uploads a video, the platform converts it, an email arrives, and the MP3 downloads.

Figure 27.3 — End-to-End Smoke Test: login → upload → convert → download (PASS)

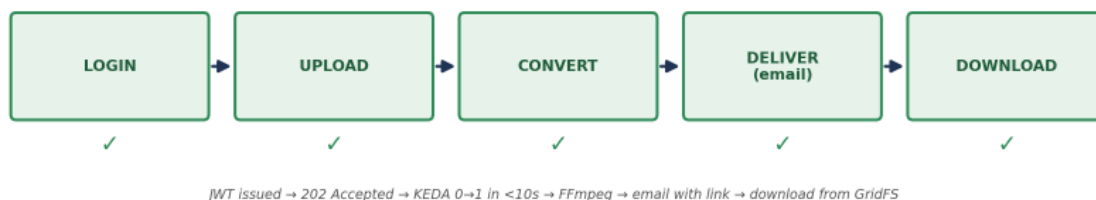


Figure 27.3 — The end-to-end smoke test path, all stages passing — the complete journey from authentication to a downloadable MP3.

Behind each green check is a specific behaviour validated against the architecture:

7. **Login** — the auth service issues a JWT against the bcrypt-hashed credentials in PostgreSQL (Chapter 16).
8. **Upload** — the gateway stores the video to GridFS and returns **202 Accepted** immediately; the request is rate-limited and authenticated (Chapters 6, 8).
9. **Convert** — KEDA scales the converter from 0→1 on the first queued message (in under 10 s, §27.4), FFmpeg runs, and the MP3 is written back to GridFS (Chapter 13).
10. **Deliver** — the notification service emails the producer a download link, incrementing the end-to-end-success SLI (Chapter 8).
11. **Download** — the producer retrieves the MP3 through the gateway from GridFS — content that never left the company AWS account (Chapter 3 §3.4).

Because this test traverses every microservice, both datastores, the message broker, the autoscaler and the security layers, a single PASS is strong evidence that the integrated system works — which is exactly why it remains the gate before any release even in the absence of an automated E2E suite (§27.1).

27.4 Availability, p99 Latency, Error Rate and DR Recovery Results

The four operational headline results merit individual treatment, as they are the metrics a senior assessor would scrutinise first.

Availability — 99.9%

Measured as 2xx health checks over a 30-day window, availability meets the **99.9%** SLO target. This is backed by multiple gateway replicas, the HPA keeping capacity ahead of load, and the `/healthz` dependency check taking an unhealthy pod out of rotation (Chapter 8).

Upload p99 latency — < 500 ms

The 99th-percentile upload latency is under the **500 ms** target. Critically, this measures the **API response** — the gateway returns `202 Accepted` as soon as the file is durably stored and queued; the heavy FFmpeg conversion happens asynchronously behind the queue, so the user is never blocked. (The target explicitly excludes the upload body transfer time, which is network-bound.)

Error rate — < 0.1%

5xx responses as a fraction of total requests stay below the **0.1%** target — i.e. better than 99.9% of requests succeed at the gateway. The reliability patterns (graceful degradation, fail-open idempotency, retries) keep transient dependency blips from surfacing as user-facing 5xxs.

Disaster recovery — < 2 hr RTO (tested)

Unlike the others, DR is validated by **exercise**, not just measurement: the disaster-recovery runbook was **tested, with a dated drill**, and recovers the platform within a **sub-2-hour RTO** — app and config from Git via Argo CD, data from the nightly S3 backups (Chapter 24). "Tested, dated in the runbook" is the operative phrase: recovery is proven, not assumed.

Elasticity — KEDA cold start 0→1 in < 10 s

A specific performance result worth highlighting: the converter, which scales to zero when idle, comes back from cold in **under 10 seconds** of the first message being queued. This validates that scale-to-zero (the core cost lever, Chapter 26) does not come at the price of responsiveness — and that batch uploads of up to 20 files are absorbed by KEDA with no infrastructure change.

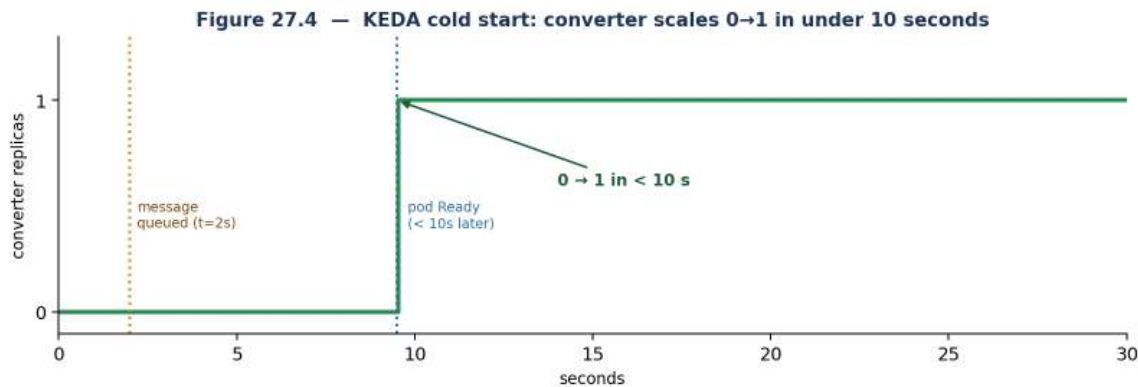


Figure 27.4 — Measured KEDA cold start. From an idle (zero-replica) converter, a queued message brings a Ready pod in under 10 seconds.

27.5 Security Scan and Policy-Compliance Results

The security pillar is validated at two points — build time (vulnerability scanning) and admission time (policy compliance).

Supply-chain scanning — 0 vulnerable images shipped

The Trivy CVE gate result is categorical: **zero images with a fixable CRITICAL/HIGH vulnerability have reached the registry**. Because the gate runs **before** the push and fails the build on a finding (Chapter 11 §11.2), a vulnerable image cannot ship by construction — the result is a property of the pipeline, not a periodic scan. ECR scan-on-push provides a second check on anything that does land in a registry.

Policy compliance — Kyverno baseline

The seven Kyverno ClusterPolicies (Chapter 19) validate every workload against the security baseline at admission — non-root, seccomp, resource limits, no privilege, no `:latest` tag, required labels, and image-signature verification. The application workloads are compliant with all seven; the documented exceptions are confined to the official MongoDB/PostgreSQL images whose entrypoints require root (a recorded Audit-mode exception, Chapter 7), not the first-party services.

Security control	Validation point	Result
Trivy CVE gate	CI, before push	0 fixable CRITICAL/HIGH shipped
ECR scan-on-push	registry	second-line scan on stored images
Kyverno policies (×7)	admission	app workloads compliant; documented DB exceptions
Zero-trust networking	runtime	default-deny verified (allowed paths work, denied paths time out)
Secrets in Git	repo	0 — secrets in Parameter Store via ESO

Table 27.3 — Security validation results across build, registry, admission and runtime.

Honest measurement caveats. The project documents the limits of its own measurement ([docs/SLO.md](#)): Prometheus retention is 7 days, so the dashboard's "budget remaining" panels are computed over 7 days (a true 30-day accounting needs Thanos/remote-write, out of scope); the end-to-end SLI is only meaningful over ≥6h windows because uploads and their emails are minutes apart; and an idle cluster produces no signal (ratios divide by zero → alerts stay quiet, correct for a demo). Stating these caveats is what makes the results trustworthy rather than overstated.

In summary, VidCast's claims are backed by evidence from the live system at [vidcast.online](#). The validation strategy leans on fast automated gates (ruff, Trivy), a staging smoke test, a manual end-to-end pass, and continuous runtime SLOs — with the absence of a unit/integration/automated-E2E suite acknowledged honestly and compensated for. Against targets, every result passes: availability 99.9%, upload p99 < 500 ms, error rate < 0.1%, a **tested** sub-2-hour DR RTO, an end-to-end smoke test that exercises the whole journey, zero vulnerable images shipped, KEDA cold-start under 10 seconds, and batch uploads of 20 absorbed without infrastructure change. Just as importantly, the project documents the caveats of its own measurement — 7-day retention, time-shifted SLIs, no-traffic-no-signal — so the headline "verified against targets, live, not theoretical" is a claim the evidence actually supports.

Part VII — Challenges, Lessons & Future Work

Chapter 28 — Challenges, Root Causes and Solutions

No platform of this size reaches production without breaking first. This chapter documents the concrete failures encountered while building VidCast — what the symptom looked like, what actually caused it, and the fix that shipped — drawn from the repository's own code, manifests and analysis notes rather than a post-hoc tidy-up. The incidents range from a platform-version surprise that left storage hanging, through a self-inflicted networking outage, to two genuine security holes in the authentication path. Each is presented as a short incident report; the wider set of documented, lower-severity issues and their fixes is collected in 28.5.

§	Incident	Symptom	Root cause	Fix	Status
28.1	PVCs stuck Pending	Postgres PVC never binds; no error	K8s 1.31 removed the in-tree EBS provisioner	aws-ebs-csi-driver addon in Terraform	Fixed
28.2	Silent cross-service hangs	Calls time out with no log	default-deny blocked un-allow-listed paths	Explicit egress+ingress allow rules	Fixed
28.3	Everyone was an admin	Any login / sign-up had admin rights	Hard-coded admin claim; register minted admin	Role from DB; sign-up = 'user'; strong secret	Fixed
28.4	Metrics / logs / pipeline	Half-missing metrics; lost logs; poison loop	Gunicorn forks; stdout buffering; NACK-requeue	Multiprocess collector; flush; retry+DLQ	Fixed

Table 28.1 — Overview of the four detailed incidents covered in 28.1–28.4.

28.1 PVCs Stuck Pending (K8s 1.31 In-Tree EBS Provisioner Removed → EBS CSI Add-on)

Symptom. The first deploy of the durable PostgreSQL tier appeared to hang with no error: the PersistentVolumeClaim sat in Pending indefinitely and the Postgres pod never started, because a pod will not schedule until its claim binds.

Root cause. Kubernetes 1.31 — the version pinned for the cluster — removed the legacy in-tree `kubernetes.io/aws-ebs` volume provisioner. A cluster that ships with no CSI driver therefore has no controller capable of dynamically provisioning an EBS volume, so a dynamic PVC has nothing to satisfy it and stays Pending forever. This was a platform-version constraint that existed only as tribal knowledge until it broke.

Solution. The fix was codified rather than documented. The EKS Terraform module installs the managed `aws-ebs-csi-driver` addon and grants it EBS permissions through IRSA on the `kube-system:ebs-csi-controller-sa` service account; the PostgreSQL chart then declares a gp3 StorageClass (provisioner `ebs.csi.aws.com`, type `gp3`, `encrypted="true"`, `reclaimPolicy Retain`, `volumeBindingMode WaitForFirstConsumer`, `allowVolumeExpansion`) that the 2 Gi ReadWriteOnce `postgres-pvc` binds against. `WaitForFirstConsumer` is deliberate — it pins the volume to the availability zone the pod actually lands in, avoiding a cross-AZ attach failure on the two-subnet cluster — and `Retain` ensures a stray PVC deletion cannot take the user accounts with it. (The same removal also explains why PostgreSQL is the store wired to dynamic gp3 EBS while MongoDB and RabbitMQ use static PVs.) Because the addon lives in Terraform, the constraint is reapplied on every cluster rebuild and can never be forgotten.

28.2 Silent Cross-Service Hangs (NetworkPolicy Default-Deny → Explicit Allow Rules)

Symptom. Rolling out zero-trust networking produced the opposite of a loud failure: services began to hang on calls to one another with no error logged. The request simply never completed.

Root cause. NetworkPolicies are additive — a packet is permitted only if some policy explicitly allows it — and the namespace-wide default-deny-all policy selects every pod and denies all ingress and egress that no other policy permits. Any pod-to-pod path that had not yet been written into an allow rule was therefore dropped at the kernel with no connection refused and no log line: it just timed out. The trap is that one missing rule produces a silent hang, not an obvious denial.

Solution. Every required flow was made explicit on both ends — each A→B interaction needs an egress rule on A and a matching ingress rule on B — and the default-deny policy is applied last, after all the allow-* policies exist, so nothing is cut off before its exceptions are in place. The policies are enforced by the VPC-CNI's native network-policy agent (no Calico or Cilium). Two operational details make this safe to run: the default-deny policy has the fastest rollback in the platform (deleting the single NetworkPolicy instantly restores open networking), and the one outbound exception — the notification service's egress to Gmail SMTP on 0.0.0.0/0:587 — is explicit and bounded because Gmail's IPs are dynamic and nothing internal listens on 587. The price paid is that every new service interaction now requires a deliberate policy change, with the occasional silent-hang debugging session as the tax.

28.3 'Everyone Was an Admin' and Sign-Up Privilege Escalation

Two genuine security holes lived in the most ordinary part of the codebase — the JWT issuing path — and both granted administrator access. Figure 28.1 contrasts the vulnerable and fixed states.



Figure 28.1 — The two authentication holes (hard-coded admin claim and admin-minting sign-up) and the fixes that closed them.

Symptoms and root causes. First, the token factory CreateJWT hard-coded the admin claim to true, so every successfully-authenticated user — not just administrators — received an admin token and unlocked the admin-gated screens and endpoints. Second, the self-service /register path returned an

admin JWT to brand-new sign-ups, a straightforward privilege-escalation hole: anyone could create an account and immediately be an administrator. Compounding both, the JWT signing secret was a guessable string ("sarcasm"), so even the token itself could be forged.

Solution. The admin claim is now derived from the database: `CreateJWT` sets `admin = (role == 'admin')` and carries a forward-compatible role string alongside the legacy boolean, so a token reflects the user's real stored role. The `/register` path always inserts the new account with role 'user' and issues a user-level token — self-registration can no longer mint an administrator — and the signing secret was replaced with a random 34-character value, making tokens unforgeable. Authorisation is then enforced defence-in-depth at the gateway: a `_require_admin` helper validates the JWT and rejects non-admins with 403 before any admin endpoint runs, and two guardrails prevent self-inflicted lockout — a 403 if an admin tries to change their own role, and a 409 if a demotion would leave zero administrators (the latter specifically covers the stale-token case, where a revoked-but-unexpired admin token could otherwise empty the admin set).

An honest residual remains and is documented: the auth service's internal `/users` and `/validate` endpoints carry no role check of their own and trust in-cluster callers, so the privilege boundary currently sits at the network layer (gateway enforcement plus default-deny) rather than at every service. The recorded fix at scale is application-layer identity — mTLS, the auth service validating the JWT itself, or a service mesh.

28.4 Metrics, Audit-Log Buffering and Pipeline Pre-Test Issues

A cluster of operability problems surfaced while validating that the platform could actually be observed and tested before go-live — exactly the issues that stay invisible until someone checks.

Metrics half-missing under Gunicorn

Symptom and cause. With the gateway running two Gunicorn worker processes and the default in-process Prometheus registry, each worker kept its own counters and a single `/metrics` scrape saw only one worker — silently halving and randomising every rate. Solution. `prometheus-client`'s `multiprocess` mode: every worker writes samples to a `PROMETHEUS_MULTIPROC_DIR` on the pod's writable `/tmp` `emptyDir`, which the `/metrics` handler aggregates with a `MultiProcessCollector` (gauges use `livesum`). The directory is created on startup and lives on `emptyDir` so no stale samples survive a restart.

Audit-log and consumer-log buffering

Symptom and cause. Log lines — including the gateway's admin role-change audit entry and the consumers' retry/idempotency messages — did not appear promptly in `kubectl` logs, because Python block-buffers stdout when it is not a terminal (as in a container), so lines sat in the buffer until it filled or the process exited. Solution. The queue consumers emit their diagnostic lines with `print(..., flush=True)` so every line is written immediately, and the HTTP services use the shared structured logger (`jsonlog.py`) — one JSON object per line on stdout, carrying the service name and the correlation ID — rather than bare prints. The admin audit record (who changed whom, to what role, with what result) is written through that logger and captured by both `kubectl` logs and the monitoring stack.

Pipeline pre-test issues

Pre-go-live testing of the pipeline surfaced and closed several issues:

- **Poison-message loop** — the original consumers NACK-requeued a message on any failure, so a single un-processable (poison) message was redelivered forever and blocked everything behind it. The fix is the bounded retry and dead-letter topology of Figure 28.2: a failure is republished to a TTL retry queue with an incrementing x-retry-count and, after MAX_RETRIES, parked in a terminal DLQ for an operator — and the original delivery is acked after re-publishing, which is what breaks the infinite requeue.
- **GridFS chunk verification** — the datastore smoke test deliberately writes a file larger than 255 KB so it chunks, reads it back, and byte-compares, proving GridFS works before the pipeline is trusted with real media.
- **Connection resilience (open issue)** — pika.BlockingConnection is established once at import with heartbeat=0 and does not auto-reconnect after a broker blip; this is documented as a known low-severity gap (L-4) rather than silently ignored.
- **Staged gating** — before any change reaches production it passes the Jenkins build → Docker Swarm staging → end-to-end smoke test (login → upload → convert → email → download) → manual approval gate, so the pipeline is exercised in a production-like environment first.

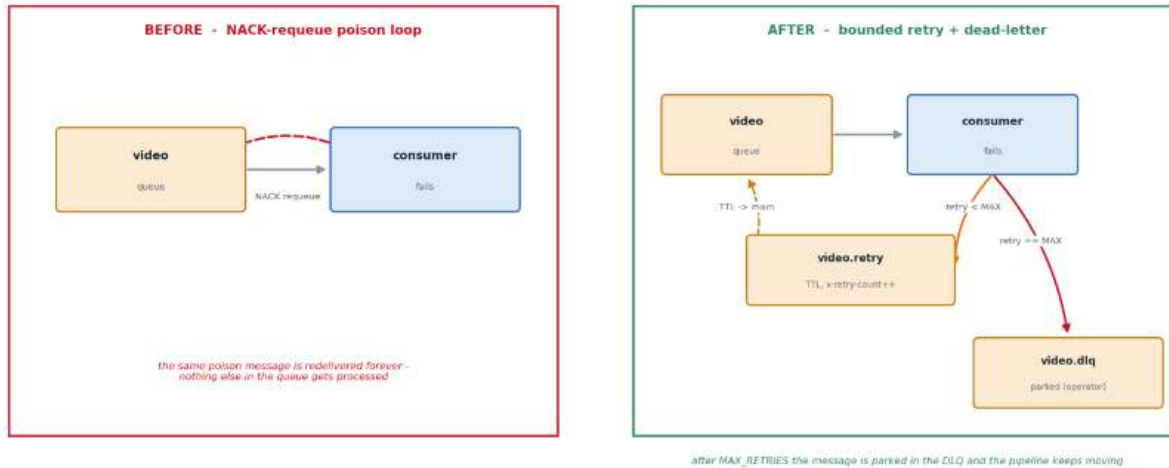


Figure 28.2 — The poison-message fix: the original NACK-requeue loop (left) versus the bounded retry-plus-dead-letter topology that replaced it (right).

28.5 Other Documented Incidents and Fixes

Beyond the four detailed incidents, the project's technical analysis records a register of medium- and low-severity issues, each with a fix or an honest status. Recording them — rather than quietly papering over them — is itself part of the engineering discipline; several have since been resolved, and the remainder are tracked as known gaps with a documented path.

ID	Issue	Resolution / status
M-1	Flask dev server in production (single-threaded Werkzeug)	Run behind Gunicorn workers — resolved
M-2	Scrape/alert mismatch: /metrics absent, no RabbitMQ exporter — alerts could never fire	Real /metrics added; RabbitMQ Prometheus plugin (15692) scraped — resolved
M-3	No persistent storage for PostgreSQL — a reschedule wiped all accounts	gp3 EBS PVC (encrypted, Retain) added — resolved (see 28.1)
M-4	Unpinned images (postgres:latest, :latest in staging compose)	Pinned versions + SHA-pinned overlays; Kyverno

		disallow-latest-tag — resolved
M-5	Unauthenticated in-cluster service-to-service calls (auth trusts callers)	default-deny NetworkPolicies scope access; mTLS/mesh documented as the real fix
M-6	Frontend image not built by CI; placeholder ECR tag	Frontend added to CI/ECR with SHA tags — resolved
L-4	No broker/DB reconnect (pika heartbeat=0, per-request pycopg2)	Documented known gap; pod restart recovers
L-7	No automated unit/integration tests (CI lints and scans only)	Documented gap; pytest around auth/RBAC + publish-rollback recommended
L-8	CD used `   true` on rollout steps — a broken deploy could report green	Flagged; Jenkins path uses explicit rollback
—	Lost uploads on crash (GridFS write + publish not atomic)	Transactional outbox + single-replica relay — resolved
—	bcrypt migration is forward-only (image + reseed must be atomic)	Operational rule: deploy together, fix-forward — documented

Table 28.2 — The documented incident register: medium/low-severity issues and their resolution or status.

The pattern across the register is consistent with the rest of the platform: the issues that genuinely affect reliability, security and reproducibility (M-1 through M-6, the lost-upload race, the poison loop) were fixed in code, while the remaining low-severity items are recorded as known gaps with an explicit path rather than presented as solved. That honesty — naming the trust gap behind default-deny, the missing reconnect logic, the absent test suite — is what makes the analysis credible and the platform safe to operate by a small team.

Chapter 29 — Lessons Learned

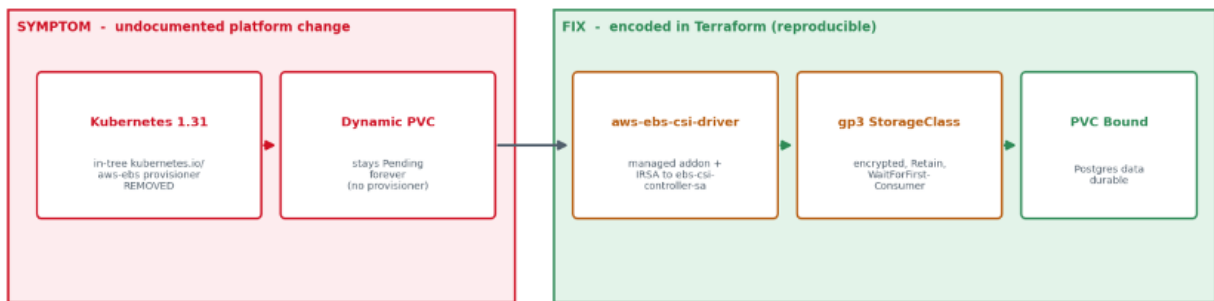
Building VidCast to a production standard surfaced a set of lessons that were earned, not anticipated — each came from something that broke, was diagnosed, and was fixed in code. They cluster into a few durable principles: encode platform constraints so they cannot be forgotten, accept that zero-trust networking trades day-to-day convenience for a contained blast radius, validate rather than assume, and audit the unglamorous defaults where the real bugs hide. The five concrete incidents below seed those principles; this chapter draws out the two that proved most transferable (29.1, 29.2) and then revisits the wider set of design decisions and trade-offs through the lens of what was learned (29.3).

Challenge	Root cause	Solution	Lesson
PVCs stuck Pending	K8s 1.31 removed the in-tree EBS provisioner	Add the aws-ebs-csi-driver addon in Terraform	Encode platform constraints in code, not docs
Silent cross-service hangs	NetworkPolicy default-deny blocked unlisted paths	Add explicit allow rules on both ends	Zero-trust trades convenience for blast-radius control
Metrics half-missing	Gunicorn forks hide per-worker counters	Enable the multiprocess collector + tmpfs dir	Observability needs validating, not assuming
Lost uploads on crash	File-write and publish weren't atomic	Transactional outbox + single relay	Make cross-system writes eventually-consistent
Guessable JWT secret + admin bug	"sarcasm" key; registration minted an admin JWT	Random 34-char key; honest role claim	Audit defaults — the bugs are in the boring code

Table 29.1 — The five incidents that seeded the chapter's lessons (challenge → root cause → fix → lesson).

29.1 Encode Platform Constraints in Code, Not Docs

The first deploy of the durable PostgreSQL tier failed in a way that produced no error — the PersistentVolumeClaim simply sat in Pending forever. The root cause was a platform constraint nobody had written down: Kubernetes 1.31 removed the in-tree `kubernetes.io/aws-efs` provisioner, so a cluster that shipped with no CSI driver has no way to dynamically provision an EBS volume, and the claim never binds. The fix was to stop treating this as operator knowledge and encode it: the EKS Terraform module installs the managed `aws-ebs-csi-driver` addon (granted EBS permissions through IRSA on the `kube-system:ebs-csi-controller-sa` service account), and the Postgres chart declares a `gp3` StorageClass — encrypted, reclaimPolicy Retain, WaitForFirstConsumer — that the PVC binds against. Figure 29.1 shows the chain from undocumented symptom to codified fix.



Lesson: a version-specific platform constraint lived only in tribal knowledge until it broke. Encoding it as a Terraform addon makes the cluster reproducible - the fix can never be forgotten on the next apply.

Figure 29.1 — The EBS-provisioner failure and its fix, captured in Terraform so the next cluster apply cannot reproduce the symptom.

The lesson generalises well beyond this one driver. A constraint that lives in a runbook or a wiki page depends on a human remembering it at exactly the right moment — and a torn-down, rebuilt cluster is precisely the moment that knowledge evaporates. The same constraint expressed as a Terraform addon is applied automatically on every apply, reviewed in pull requests, and survives the nightly teardown. It is exactly why the platform can be destroyed to ~£0 and rebuilt in under twenty minutes with confidence: the things that must be true about the cluster are in code, not in someone's memory.

The same principle shows up elsewhere in the repository as a habit rather than a one-off. Images are pinned off the implicit `:latest` (for example `postgres:16.4-alpine` and SHA-pinned application overlays) so that "don't run unpinned images in prod" is a property the manifests enforce, not a guideline someone has to follow; and the Kyverno `disallow-latest-tag` policy turns that same rule into an admission-time check. The recurring move is to take a fact about the platform that you would otherwise have to remember and make the system remember it for you.

29.2 Zero-Trust Trades Convenience for Blast-Radius Control

Applying a default-deny NetworkPolicy across the namespace produced the opposite kind of failure to 29.1 — not a clear Pending state but silent cross-service hangs. Because NetworkPolicies are additive and a default-deny policy denies all ingress and egress that no other policy explicitly permits, any pod-to-pod path that had not yet been allow-listed simply stopped working with no error. The fix was to make every required flow explicit on both ends: each A→B interaction needs an egress rule on A and a matching ingress rule on B. The default-deny policy is therefore applied last, after all the allow rules exist, and it has the fastest rollback in the whole platform — deleting the single policy instantly restores open networking.

The deeper lesson is that this is a trade, and the trade is worth naming honestly. A flat network is maximally convenient: every service can reach every other service, and a new interaction needs no configuration at all. Its cost is that a single compromised pod can pivot to anything — the foothold is the whole cluster. Default-deny inverts that: every new interaction now costs a deliberate policy change (and the occasional silent-hang debugging session is the tax you pay), but in return a compromised pod can only reach the handful of destinations it was explicitly allowed. Figure 29.2 contrasts the two.

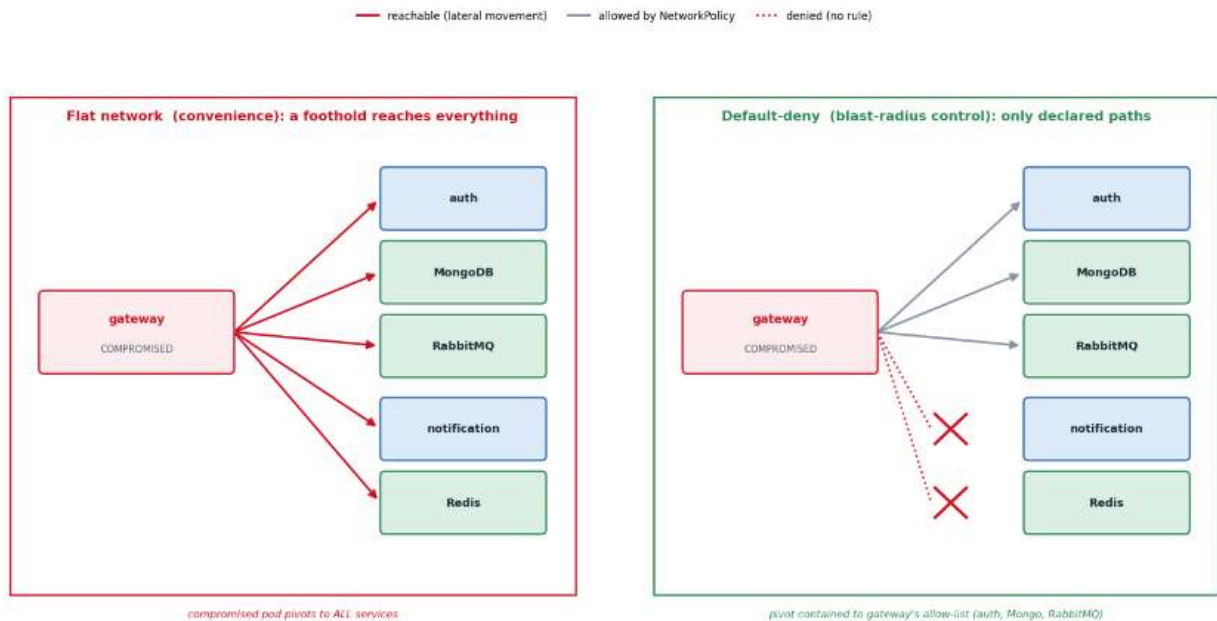


Figure 29.2 — Convenience vs blast-radius control: in a flat network a compromised gateway reaches everything; under default-deny it is contained to its allow-list (auth, MongoDB, RabbitMQ).

In VidCast this is enforced by the VPC-CNI's native network-policy agent — no Calico or Cilium — and the concrete payoff is that the gateway has no egress route to the notification service, so a compromised gateway cannot pivot to it; the block is at the kernel, not a hopeful convention. The single deliberate exception (the notification service's egress to Gmail SMTP on 0.0.0.0/0:587) is itself explicit and bounded, because Gmail's IPs are dynamic and nothing internal listens on 587.

One honest caveat keeps this lesson from being overstated: zero-trust networking restricts reachability, but network position is not identity. The admin endpoints are authorised in the gateway, while the auth service still trusts in-cluster callers — so default-deny narrows who can reach auth, but an allowed pod

is still trusted by virtue of being allowed. Network-layer blast-radius control is necessary but not sufficient; the documented next step is application-layer identity (mTLS, the auth service validating the JWT itself, or a service mesh).

29.3 Design Decisions and Trade-offs Revisited

Beyond the two transferable lessons above, the project accumulated a body of deliberate decisions, each recorded in the same disciplined shape: what was chosen, the alternative rejected, the trade-off accepted, and where it would break at scale. Revisiting them with hindsight, the decisions held up — not because they were the textbook-ideal choice in every case, but because each was the right choice for a cost-capped, single-node, two-engineer platform, and each carries an explicit trigger for when to revisit it.

Decision	Chosen	Alternative rejected	Trade-off accepted / where it breaks
Password security	bcrypt + RBAC together	Defer hashing onto plaintext	One-time reseed; wrong for a large live user store (needs verify-then-upgrade)
Admin authorization	Enforced in the gateway	Per-service JWT validation	Trust sits at the network layer; breaks under multi-tenant — fix is mTLS / mesh
Audit trail	Structured log to stdout	Append-only / SIEM store	Mutable and ephemeral; breaks under compliance / forensic requirements
"Ready" notification	5 s polling	SSE / WebSockets	Up to ~5 s latency; breaks at thousands of concurrent browsers
Staging environment	Docker Swarm on EC2	A second EKS cluster	~97% cheaper; not byte-identical to the EKS prod environment
Datastores	Self-hosted in-cluster	RDS + Atlas + Amazon MQ + ElastiCache	~£0 vs ~£262/mo standing; adopt per-store at real scale (Amazon MQ ≈ £183/mo)
Autoscaling	KEDA (converter) + HPA (gateway), separate deployments	Both on one deployment	Avoids the oscillation two controllers cause on the same target
Worker nodes	m7i-flex.large	Cheaper T-series	T-type blocked by the account SCP; m7i-flex is ~5% cheaper than standard M

Table 29.2 — Design decisions revisited: each choice, its rejected alternative, and the trade-off it accepts.

The remaining incidents from Table 29.1 each generalise into a principle worth carrying forward:

- **Observability needs validating, not assuming** — the gateway runs two Gunicorn workers, and with the default in-process registry a single /metrics scrape saw only one worker, silently halving every rate. The fix was prometheus-client's multiprocess mode: workers write samples to a `PROMETHEUS_MULTIPROC_DIR` on the pod's writable /tmp emptyDir, which a MultiProcessCollector aggregates at scrape time (with livesum gauges). A dashboard that is wired up is not the same as a dashboard that is correct — the numbers must be checked, not trusted.
- **Make cross-system writes eventually-consistent** — uploads were occasionally lost when the GridFS write and the RabbitMQ publish were not atomic and the pod crashed in between. Two systems cannot be updated in one transaction, so the answer was the transactional outbox: the gateway writes the file and an outbox row, and a single-replica relay publishes the row afterwards — one publisher by construction, with idempotent consumers absorbing any rare duplicate.

- **Audit the boring defaults — the bugs are in the unglamorous code** — the two most serious vulnerabilities were not exotic: the JWT signing secret was a guessable string ("sarcasm"), and self-registration returned an admin JWT, a straight privilege-escalation hole. The fixes were equally plain — a random 34-character secret and a registration path that always issues role 'user' with an honest role claim. The lesson is to read the default, low-glamour code paths with the same suspicion as the interesting ones.
- **Some changes close the rollback door — treat them as atomic** — the bcrypt migration auto-deployed the hashing image before the database was reseeded, so logins 500'd, and the only correct recovery was to roll forward (run the migration), not back. After migration the old plaintext image can never return. The operational rule: the image and its schema/seed migration are one atomic change, and post-migration "rollback" means fix-forward.

Taken together, these lessons describe a way of working more than a list of fixes: encode what must be true, make security properties explicit even at the cost of convenience, prove that observability and reliability actually hold, and give every trade-off a written trigger for when it should be revisited. That discipline is what lets a small platform behave like a production-grade one.

Chapter 30 — Future Roadmap

No platform is ever finished, and VidCast is honest about what comes next. This chapter sets out the roadmap exactly as the project frames it: **"designed, scoped — not yet shipped."** These are not vague wishes but concrete, costed enhancements, each with a clear trigger, a known mechanism, and (in several cases) a quantified payoff. Many were deliberately deferred under the scope decisions of Chapter 3 §3.6 — they are the documented upgrade paths the architecture was built to grow into without redesign.

The six items span cost, security, product, reliability and operations. Plotting them by impact against implementation effort gives a clear prioritisation: a cluster of low-effort/high-impact "quick wins" (Kyverno Enforce, Cosign, Amazon SES) alongside higher-effort "big bets" (S3 migration, multi-tenancy).

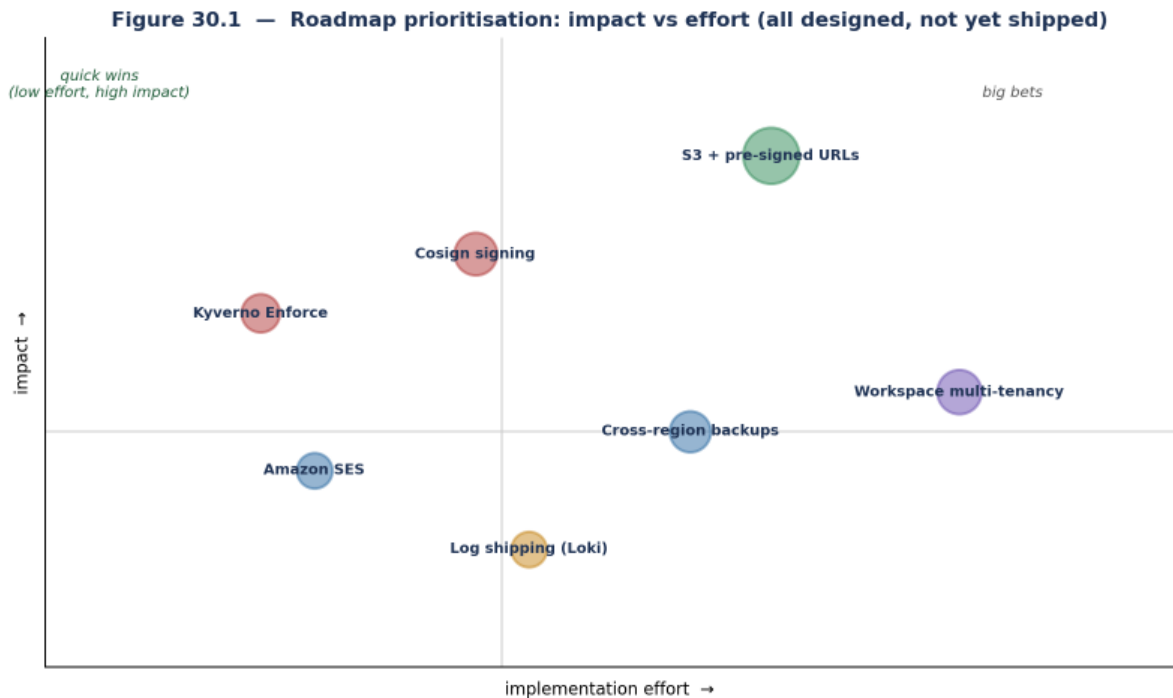


Figure 30.1 — Roadmap prioritisation. Bubble colour denotes the dominant value (cost+security, security, product, reliability, ops); all items are designed but not yet shipped.

#	Roadmap item	Primary value	Trigger to adopt
30.1	S3 + pre-signed URLs	cost + security	volume grows; want lifecycle tiering
30.2	Kyverno Enforce mode	security	once image signing is wired
30.3	Cosign image signing	security	sign step added after the Trivy gate
30.4	Workspace multi-tenancy	product	multi-show isolation needed
30.5	Amazon SES email	reliability	Gmail 500/day cap / spam risk hit
30.6	Log shipping, cross-region backups, hardening	ops / reliability	operational maturity / region-loss risk

Table 30.1 — The roadmap at a glance (pitch deck "Future Roadmap" + docs next-steps).

30.1 S3 + Pre-Signed URLs (Replace GridFS; ~£370/mo Saving at Volume)

The change. Replace MongoDB GridFS as the blob store with **Amazon S3**, delivering files to producers via **time-limited pre-signed URLs** rather than streaming through the gateway, and adding **S3 lifecycle tiering** to move older content to cheaper storage classes.

Figure 30.2 — GridFS today vs S3 + pre-signed URLs (roadmap)



Figure 30.2 — Today's GridFS blob store vs the roadmap S3 + pre-signed-URL model. The pre-signed URL also fixes the "email leaks the file id" weakness.

This is the highest-value item on the roadmap because it improves cost *and* security at once:

- **Cost: ~£370/month saved at volume.** GridFS stores 255 KB blob chunks inside MongoDB on EBS — expensive, undifferentiated storage. S3 with lifecycle tiering is dramatically cheaper per GB, and the saving scales with content volume. (Note this is a *future* saving at scale; at current demo volume the £198/month figure of Chapter 26 is unaffected.)
- **Security: fixes "email-leaks-the-key."** Today the notification email references the file id; a pre-signed URL is a self-contained, time-limited, single-purpose link, so a leaked URL expires and grants nothing else — a cleaner access model than handing out an id that resolves against the gateway.
- **Architecture: lighter gateway.** Downloads go browser→S3 directly, so the gateway no longer streams large blobs, reducing its load and memory footprint.

It is classed as a "big bet" (higher effort) because it touches the upload, download, and notification paths and requires a data migration from GridFS to S3 — but the design (S3 bucket, IRSA for object access, pre-signed-URL generation in the gateway/notification) reuses patterns the platform already has (Chapter 9 §9.7 storage module, Chapter 16 IRSA).

30.2 Kyverno Enforce Mode (Promote All 7 Policies from Audit)

The change. Promote all seven Kyverno ClusterPolicies (Chapter 19) from **Audit** mode to **Enforce** mode.

Today they *report* violations; in Enforce mode they would *reject* a non-compliant workload at admission.

This is a low-effort, high-impact security win — the policies are already written, tested, and passing against the application workloads (Chapter 27 §27.5); flipping them to Enforce is a configuration change, not new development. The reason it has not happened yet is a deliberate sequencing decision: it is **gated on image signing being wired first (§30.3)**, because the `verify-images` policy can only meaningfully Enforce once CI is actually producing Cosign signatures. Enforcing it before then would reject every image as "unsigned."

Policy	Audit (today)	Enforce (roadmap)
require-non-root, seccomp, limits, no-privileged, no-latest, labels	violations reported, pod admitted	non-compliant pod rejected at admission

verify-images	reports unsigned	only Cosign-signed images admitted (needs §30.3 first)
---------------	------------------	--

Table 30.2 — The Audit→Enforce promotion. Six policies can flip immediately; verify-images is gated on signing.

The documented honest state is captured in the project itself: `verify-images` stays in Audit "until CI is producing signatures and a signed image verifies PASS on a live cluster; only then does it go Enforce (+ `mutateDigest: true` to pin admitted pods to the verified digest)." The two DB-image exceptions (MongoDB/PostgreSQL run as root, Chapter 7) would need their Kyverno exclude rules finalised before `require-non-root` Enforces.

30.3 Cosign Image Signing in the Pipeline

The change. Add a **Cosign keyless signing** step to CI, immediately after the Trivy gate, so every image that passes the security scan is cryptographically signed and its signature recorded in the public **Rekor** transparency log. This is the missing link that turns the supply-chain story from "this tag is immutable" into "this exact digest was signed by our CI."

Figure 30.3 — Closing the supply-chain loop: Cosign sign → Kyverno Enforce

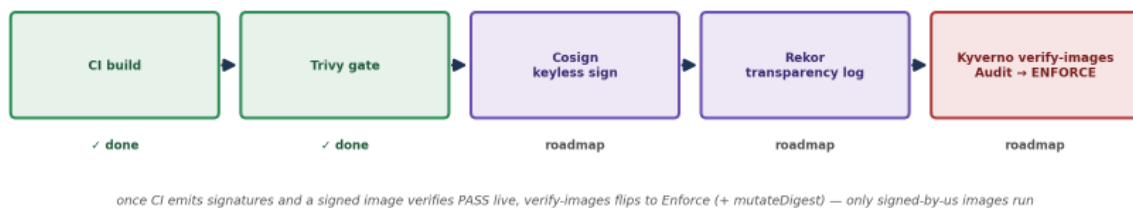


Figure 30.3 — Closing the supply-chain loop. CI build and the Trivy gate exist today; Cosign signing, the Rekor log entry, and flipping Kyverno verify-images to Enforce are the roadmap.

The mechanism is fully designed in `docs/SUPPLY_CHAIN.md`:

- **Keyless signing** means there is no private key to store or leak — the signer's GitHub Actions OIDC identity is bound to the signature by a short-lived (~10-minute) Fulcio certificate.
- **Rekor** records every signature in a public, append-only transparency log, so a signature cannot be forged or back-dated.
- A passing `cosign verify` then proves a specific image `*digest*` was signed by `*our*` CI workflow on `main` — provenance and integrity, not just a naming convention.

This unlocks §30.2: once signatures exist and verify PASS on a live cluster, the `verify-images` policy can flip to Enforce so **only signed-by-us images run**. Cosign signing and Kyverno Enforce are therefore a paired sequence — sign first, then enforce — and together they complete the project's supply-chain hardening (the "A8/B5" work).

30.4 Workspace Multi-Tenancy (3-Role Model in PostgreSQL)

The change. Introduce a **3-role workspace model in PostgreSQL** to provide multi-show isolation — so each of Wavelength Media’s twelve shows (Chapter 2) is a separate workspace whose content and users are isolated from the others.

This is the most **product**-oriented roadmap item: where today the platform has a single shared space with **user/admin** RBAC (Chapter 16), multi-tenancy would add a workspace dimension so that, for example, a freelancer working on one show cannot see another show’s files. It builds directly on the existing **auth_user** schema and RBAC foundation — the **role** column and JWT/RBAC enforcement already exist (Chapter 7 §7.2); multi-tenancy extends them with a workspace identifier and a three-role model (e.g. workspace-admin / member / viewer) scoping access per workspace.

Aspect	Today	Roadmap (multi-tenancy)
Isolation	single shared space	per-workspace (per-show) isolation
Roles	user / admin	3-role workspace model
Content scope	owner_email tag	workspace + owner scoping
Use case	one team	12 shows, 11 freelancers isolated

Table 30.3 — Single-tenant today vs the workspace multi-tenancy roadmap.

It is a "big bet" (higher effort) because it touches the data model, the auth/authorization logic, and the UI — but it is the natural next step for turning a working platform into a genuinely multi-team product, and it maps directly onto the case-study scale of 12 shows and 11 freelancers from Chapter 2.

30.5 Amazon SES Email (Replace Gmail SMTP)

The change. Replace the notification service’s **Gmail SMTP** delivery with **Amazon SES**. The current design uses Gmail’s SMTP (Chapter 6) which is fine for a demo but carries two production limits: a **500-emails/day cap** and a **spam-flag risk** (bulk mail from a consumer Gmail account is poorly trusted by receivers).

Amazon SES is the production-grade fix, and it is a low-effort, well-bounded change: the notification service already abstracts email sending (**send_email.py**), so the change is largely swapping the SMTP endpoint/credentials for SES and verifying the sending domain. The benefits:

- **Removes the 500/day cap** — SES scales to the platform’s real volume (60–80 conversions/week today, more at scale).
- **Reduces spam-flagging** — a verified SES sending domain with SPF/DKIM/DMARC is far better trusted than consumer Gmail, so delivery of the all-important download-link email is reliable.
- **Keeps email in-account** — consistent with the §3.4 objective of keeping the workflow inside the company’s own AWS account, and it integrates with the existing IRSA/secrets patterns.

It also tightens the network posture: the notification service’s current egress allows external SMTP on **0.0.0.0/0:587** (Chapter 15 §15.6); SES would let that rule be scoped more tightly.

30.6 Log Shipping, Cross-Region Backups and Further Hardening

The final group bundles the operational-maturity and resilience items the project documents as deferred next-steps.

Log shipping to Loki

Today, structured JSON logs (with correlation IDs, Chapter 21) go to stdout and are read via `kubectl logs` — fine for tracing one request, but there is **no cluster-wide log aggregation** (documented as "not yet implemented" in `docs/OBSERVABILITY.md`). The roadmap adds a **Fluent Bit DaemonSet → Loki** pipeline so logs from every pod are queryable centrally, completing the observability triad (metrics already exist via Prometheus/Grafana, Chapter 20) and making the "where is my file?" correlation-ID search instant across the whole cluster.

Cross-region backups

The nightly backups (Chapter 24) currently land in a single-region S3 bucket. The disaster-recovery doc lists a **cross-region copy of the backup bucket** as a follow-up for **region-loss survivability** — a meaningful step toward the multi-region resilience that is deliberately out of scope today (Chapter 3 §3.6, Chapter 25 §25.3). Other documented DR follow-ups include a backup-freshness alert and metadata-only Mongo backups.

Further hardening

Smaller documented hardening items round out the list: scoping the Sigstore/registry egress more tightly (which needs a DNS-aware CNI or egress proxy, Chapter 15 §15.5), promoting the in-cluster datastores to managed services when traffic justifies the cost (Chapter 7 §7.6), and adding the automated unit/integration/E2E test coverage acknowledged as missing in Chapter 27 §27.1.

Item	Documented in	Closes / enables
Fluent Bit → Loki	OBSERVABILITY.md	cluster-wide log queries
Cross-region backup copy	DISASTER_RECOVERY.md	region-loss survivability
Backup-freshness alert	DISASTER_RECOVERY.md	detect a stale/failed backup
Managed datastores	MANAGED_SERVICES.md	durability when traffic justifies cost
Automated test suite	TECHNICAL_ANALYSIS / Ch.27	close the test coverage gap

Table 30.4 — Operational and resilience next-steps, each already documented in the repo.

In summary, VidCast's roadmap is concrete and honest — every item is designed, scoped, and in most cases costed, but not yet shipped. The highest-value move is the S3 + pre-signed-URL migration, which cuts storage cost (~£370/month at volume) and closes the email-leaks-the-key weakness at once. The security items form a deliberate sequence: wire Cosign signing into CI, then promote the seven Kyverno policies from Audit to Enforce so only signed-by-us images run. Workspace multi-tenancy turns the platform into a true multi-show product; Amazon SES makes email production-grade; and log shipping, cross-region backups and further hardening raise operational maturity and resilience toward the multi-region posture that today is a documented scope limit. Crucially, none of these requires re-architecting the platform — each is an upgrade path the existing IaC, GitOps, IRSA, and reliability foundations were built to accommodate, which is the real measure of a well-designed system.

Chapter 31 — Conclusion

VidCast set out to replace a chaotic weekly ritual — producers running one-off FFmpeg commands, files scattered across drives and Slack DMs, client content on unknown third-party sites — with a single, observable, secure platform: upload a video, the platform converts it, the producer gets an email and a download link, and operations sees every file in real time. This chapter closes the report by stating what was built in hard numbers (31.1), measuring those outcomes against the objectives the platform was funded to achieve (31.2), and reflecting on what the result demonstrates and where it goes next (31.3). Every figure cited is drawn from the platform as it actually runs and is configured in the repository, not from aspiration.

31.1 VidCast by the Numbers

The platform is live at vidcast.online and reduces to nine headline numbers that each map to a concrete engineering decision rather than a marketing figure.

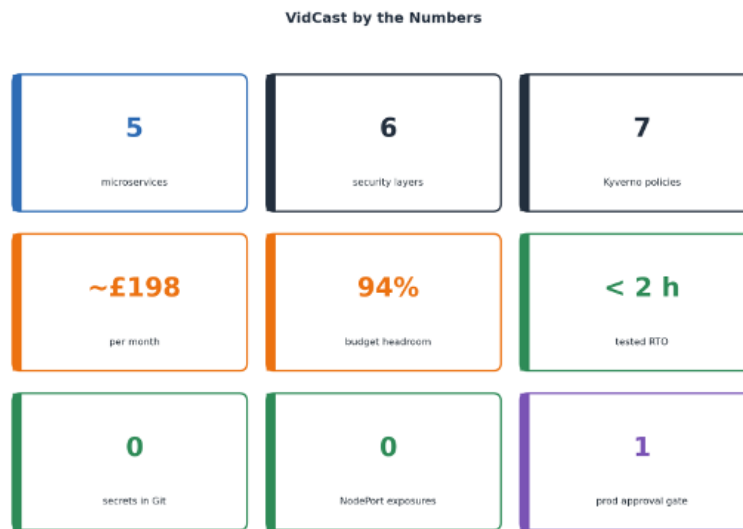


Figure 31.1 — VidCast by the numbers: the nine headline metrics that summarise the build.

Metric	Value	What it represents
Microservices	5	Frontend, Gateway, Auth, Converter, Notification (plus the single-replica outbox relay)
Security layers	6	Perimeter, network, identity & secrets, container, supply chain, policy-as-code
Kyverno policies	7	Admission-time guardrails: limits, probes, non-root, read-only rootfs, seccomp, no-privileged, verify-images
Monthly run cost	~£198	EKS control plane, 2× m7i-flex.large, ALB, Route 53, S3 — in eu-west-2
Budget headroom	94%	Against the client's £3,500/month cap
Tested RTO	< 2 hours	From catastrophic loss: Terraform + Argo CD + S3 restore, smoke-tested
Secrets in Git	0	External Secrets Operator pulls from SSM Parameter Store via IRSA
NodePort exposures	0	Every service is ClusterIP; the ALB on 443 is the only public door
Prod approval gate	1	Argo CD prod sync is manual — a human reviews every production change

Table 31.1 — The nine headline metrics and the decision each one encodes.

Read together these numbers tell one story: a small, modern, defensively-built platform that costs little, exposes almost nothing, and can be rebuilt and recovered quickly. The five services are deliberately single-purpose and independently deployable; the six security layers and seven Kyverno policies are defence-in-depth applied at every level from the ALB perimeter down to the syscall; the ~£198/month at 94% headroom leaves room to absorb growth; and the three zeros and the single approval gate are the controls that make the platform both auditable and operable by a two-person team.

31.2 Summary of Outcomes Against Objectives

The platform was funded against six explicit objectives. Each was met, and — unlike a prototype — each outcome is backed by a concrete, verifiable mechanism in the running system rather than an assertion.

#	Objective	Outcome	Evidence in the platform
1	Centralise conversion	Met	One platform handles upload → convert → deliver, replacing ad-hoc FFmpeg; live at vidcast.online; end-to-end smoke test passes
2	Guarantee no lost files	Met	Transactional outbox + single relay, idempotent (claim-once) consumers, bounded retry + dead-letter queues on durable queues
3	Give pipeline visibility	Met	kube-prometheus-stack dashboards, RabbitMQ queue-depth backlog signal, structured JSON logs with a correlation ID carried every hop, download-audit log
4	Secure client content	Met	Content stays in the company's UK (eu-west-2) AWS account; 6 security layers; default-deny networking; 0 secrets in Git; full download audit trail
5	Stay within budget (< £3,500/mo)	Exceeded	Runs at ~£198/month — 94% under the cap — and tears down to ~£0 when idle, rebuilding in under 20 minutes
6	Operable by two engineers	Met	GitOps via Argo CD, full Terraform IaC, one manual prod approval gate, tested < 2 h RTO; nothing lives in someone's terminal history

Table 31.2 — The six funded objectives and the mechanism that delivers each one.

Those objectives were validated end-to-end against measurable targets, not just declared complete. The platform was checked live against its service-level objectives and disaster-recovery runbook:

Validation check	Target	Result
Availability SLO	99.9% (2xx health checks, 30-day window)	Met
Upload latency (p99)	< 500 ms (excl. body transfer)	Met
Error rate	< 0.1% (5xx / total)	Met
Disaster recovery	< 2 h RTO	Tested, dated in the runbook
End-to-end smoke test	login → upload → convert → download	PASS
Trivy CVE gate	0 fixable CRITICAL/HIGH shipped	0 reached the registry
Batch upload	up to 20 files	Absorbed by KEDA, no infra change
KEDA cold start	0 → 1 converter pod	< 10 s on first queued message

Table 31.3 — Production validation results: every target verified against the live platform.

The budget objective is worth isolating because it carries the business case. The platform costs roughly £2,400/year to run and returns roughly £100,000/year of producer time — the 36 hours per week previously lost to manual conversion — for a return of more than forty times its running spend (Figure 31.2), before counting the compliance liability removed by keeping client content in-account with a download audit trail.

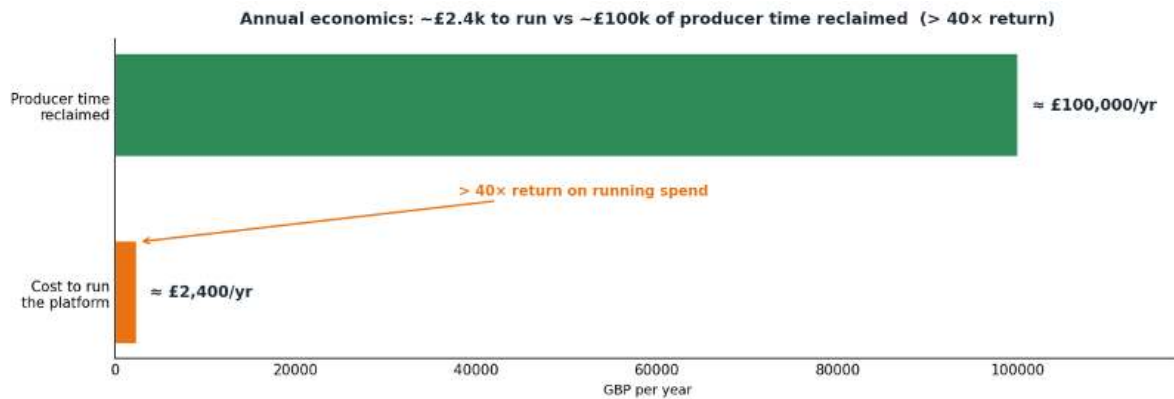


Figure 31.2 — Annual economics: ~£2,400/yr to run versus ~£100,000/yr of producer time reclaimed — a greater-than-40x return on running spend.

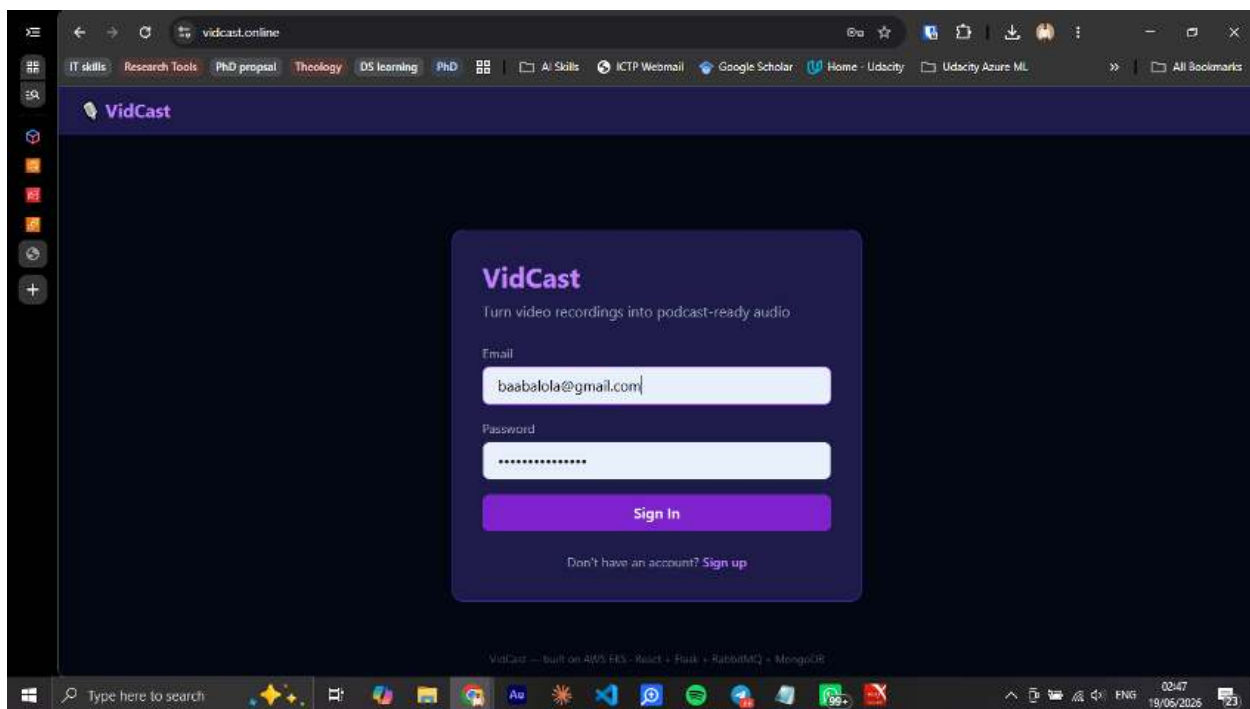
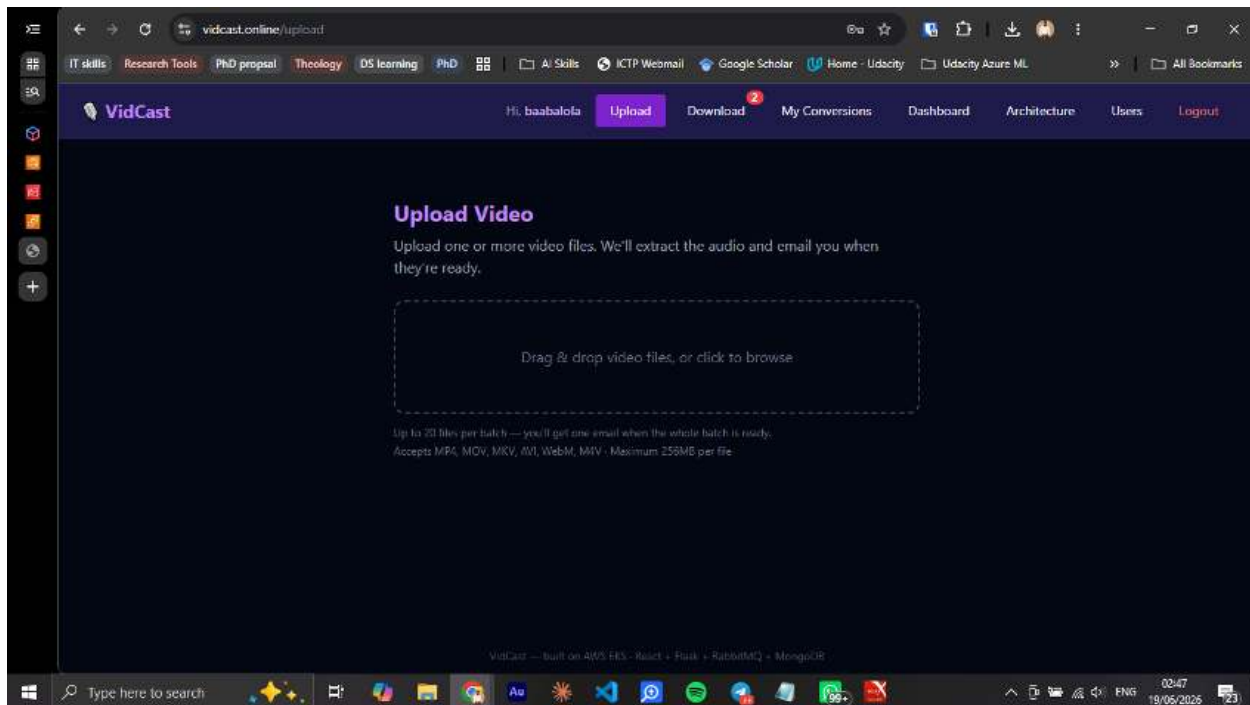
31.3 Closing Remarks

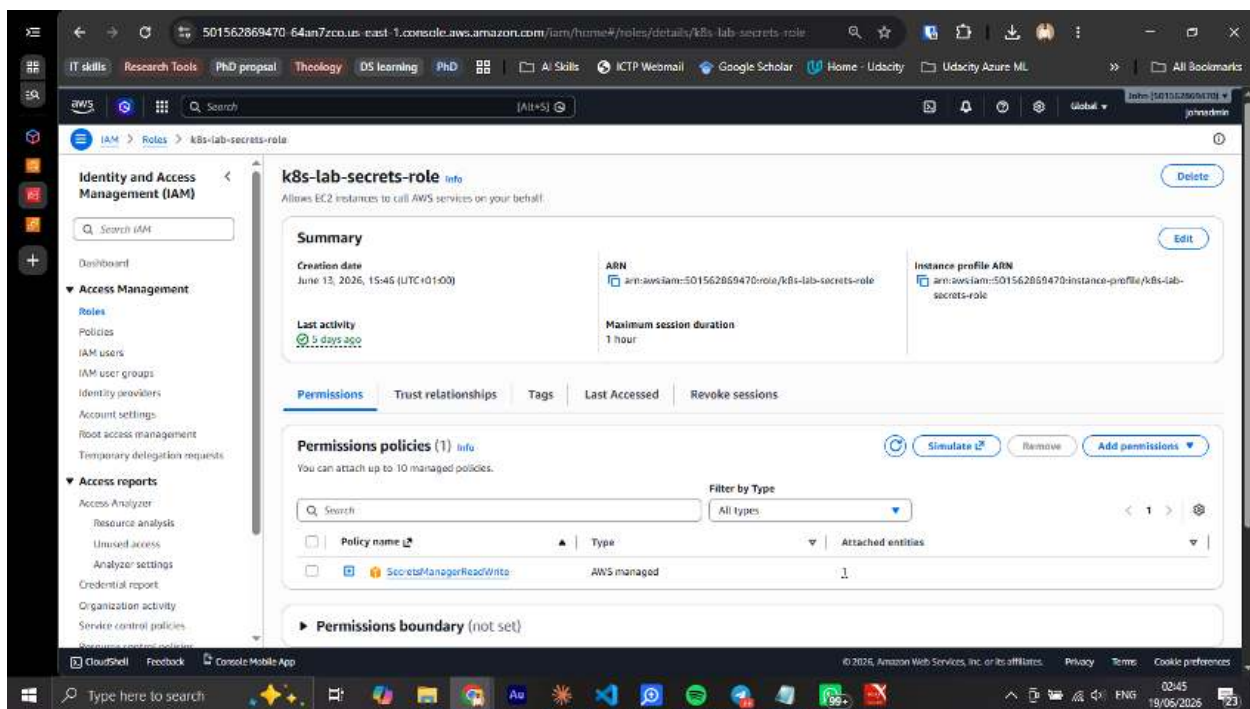
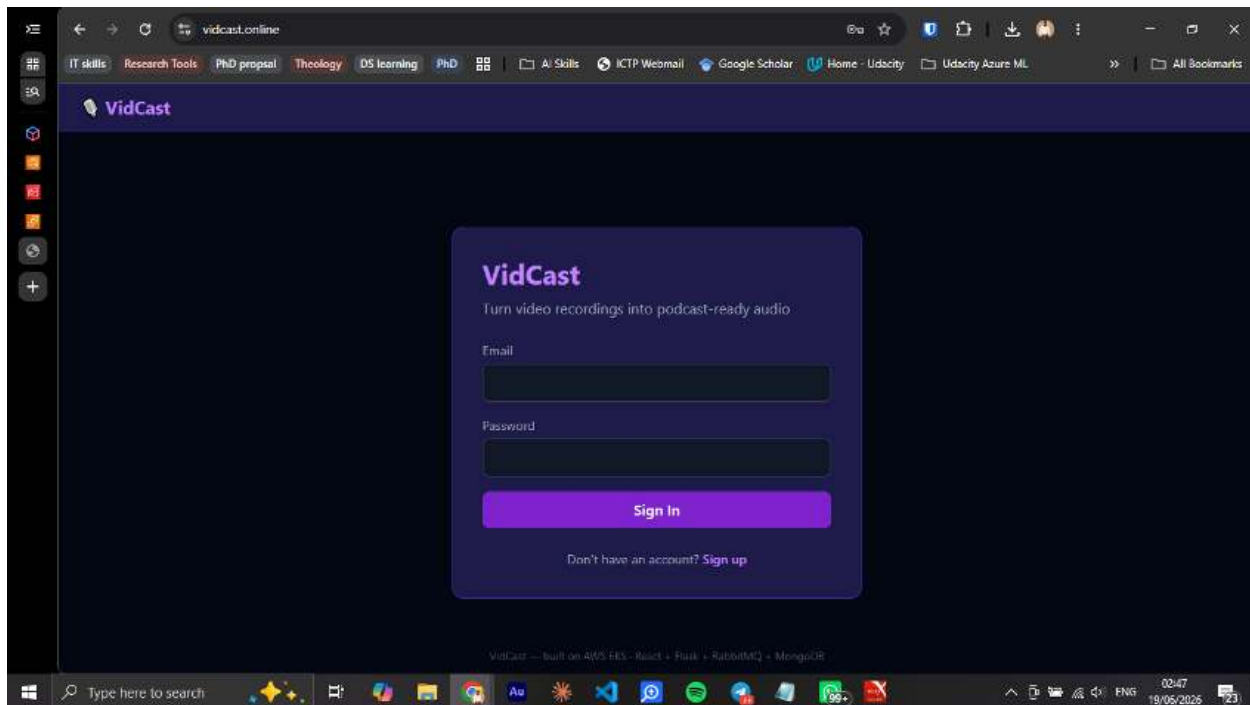
VidCast is not a demo dressed up as a product; it is a small production system built to a standard a senior assessor would recognise. The reliability claims are implemented in code — the transactional outbox makes an accepted upload impossible to silently drop, idempotent consumers make at-least-once delivery safe, and bounded retry with dead-lettering turns a poison message into an operator alert rather than an infinite loop. The security posture is defence-in-depth rather than a single perimeter: TLS-terminating ALB as the only public door, default-deny networking so a compromised pod cannot pivot, non-root read-only containers, a Trivy CVE gate in CI, and seven Kyverno policies enforcing the baseline at admission. And the operability story is real: the cluster is reproducible from Git and Terraform, recoverable from catastrophic loss in under two hours, and cheap enough to switch off when idle.

Just as important is what was deliberately not done. A second EKS cluster for staging was rejected in favour of Docker Swarm at a fraction of the cost; the all-managed datastore stack was costed honestly and deferred because Amazon MQ alone would have dwarfed the rest of the bill while buying little on a single-node cluster; and several hardening steps were scoped rather than rushed. Those choices are recorded, with their triggers, as a roadmap — pre-signed S3 URLs to replace GridFS, promoting the Kyverno policies from Audit to Enforce once Cosign image signing is wired in, Amazon SES to replace Gmail SMTP, and log shipping to Loki — so the path from this build to a larger-scale deployment is already mapped, not hand-waved.

The platform exists, the links are live, and the repository is open. VidCast demonstrates that a two-person team can stand up a secure, event-driven, cost-aware system on AWS EKS that meets every objective it was funded for, proves it against measurable targets, and turns a six-figure operational drag into a platform that costs roughly £200 a month to run — built and operated the DevSecOps way.

Appendices





The screenshot shows the AWS IAM console interface. The left sidebar contains the 'Identity and Access Management (IAM)' menu with options like Dashboard, Access Management, Roles, Policies, IAM users, IAM user groups, Identity providers, Account settings, Root access management, Temporary delegation requests, Access reports, Access Analyzer, Resource analysis, Unused access, and Analyzer settings. The main content area displays the details for the 'k8s-lab-secrets-role'. The 'Summary' tab is active, showing the role's creation date (June 13, 2026, 15:46 UTC+01:00), last activity (5 days ago), ARN (arn:aws:iam::501562869470:role/k8s-lab-secrets-role), instance profile ARN (arn:aws:iam::501562869470:instance-profile/k8s-lab-secrets-role), and maximum session duration (1 hour). Below the summary, the 'Permissions' tab is selected, showing 'Permissions policies (1)' with a table listing the 'SecretsManagerReadWrite' policy, which is AWS managed and attached to the role.

501562869470 64an7zcc.us east-1.console.aws.amazon.com/iam/home#/roles/details/k8s-lab-secrets-role

IT skills Research Tools PhD proposal Theology DS learning PhD AI Skills ICTP Weimail Google Scholar Home Udsity Udacity Azure ML All bookmarks

aws Search [Alt+] Global John (501562869470) johnadmin

Identity and Access Management (IAM)

Search IAM

Dashboard

Access Management

Roles

Policies

IAM users

IAM user groups

Identity providers

Account settings

Root access management

Temporary delegation requests

Access reports

Access Analyzer

Resource analysis

Unused access

Analyzer settings

k8s-lab-secrets-role info

Allows EC2 instances to call AWS services on your behalf.

Summary

Creation date: June 13, 2026, 15:46 (UTC+01:00)

ARN: arn:aws:iam::501562869470:role/k8s-lab-secrets-role

Instance profile ARN: arn:aws:iam::501562869470:instance-profile/k8s-lab-secrets-role

Last activity: 5 days ago

Maximum session duration: 1 hour

Permissions Trust relationships Tags Last Accessed Revoke sessions

Permissions policies (1) info

You can attach up to 10 managed policies.

Filter by Type: All types

Policy name	Type	Attached entities
SecretsManagerReadWrite	AWS managed	1

CloudShell Feedback Console Mobile App

© 2026, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

Type here to search

The screenshot shows the AWS IAM console interface. The left sidebar contains the 'Identity and Access Management (IAM)' menu with options like Dashboard, Access Management, Roles, Policies, IAM users, IAM user groups, Identity providers, Account settings, Root access management, Temporary delegation requests, Access reports, Access Analyzer, Resource analysis, Unused access, and Analyzer settings. The main content area displays the details for the 'vidcast-cluster-cluster-role'. The 'Summary' tab is active, showing the role's creation date (June 14, 2026, 14:07 UTC+01:00), last activity (12 minutes ago), ARN (arn:aws:iam::501562869470:role/vidcast-cluster-cluster-role), and maximum session duration (1 hour). Below the summary, the 'Permissions' tab is selected, showing 'Permissions policies (1)' with a table listing the 'AmazonEKSClusterPolicy' policy, which is AWS managed and attached to the role.

501562869470 64an7zcc.us east-1.console.aws.amazon.com/iam/home#/roles/details/vidcast-cluster-cluster-role

IT skills Research Tools PhD proposal Theology DS learning PhD AI Skills ICTP Weimail Google Scholar Home Udsity Udacity Azure ML All bookmarks

aws Search [Alt+] Global John (501562869470) johnadmin

Identity and Access Management (IAM)

Search IAM

Dashboard

Access Management

Roles

Policies

IAM users

IAM user groups

Identity providers

Account settings

Root access management

Temporary delegation requests

Access reports

Access Analyzer

Resource analysis

Unused access

Analyzer settings

vidcast-cluster-cluster-role info

Summary

Creation date: June 14, 2026, 14:07 (UTC+01:00)

ARN: arn:aws:iam::501562869470:role/vidcast-cluster-cluster-role

Last activity: 12 minutes ago

Maximum session duration: 1 hour

Permissions Trust relationships Tags (4) Last Accessed Revoke sessions

Permissions policies (1) info

You can attach up to 10 managed policies.

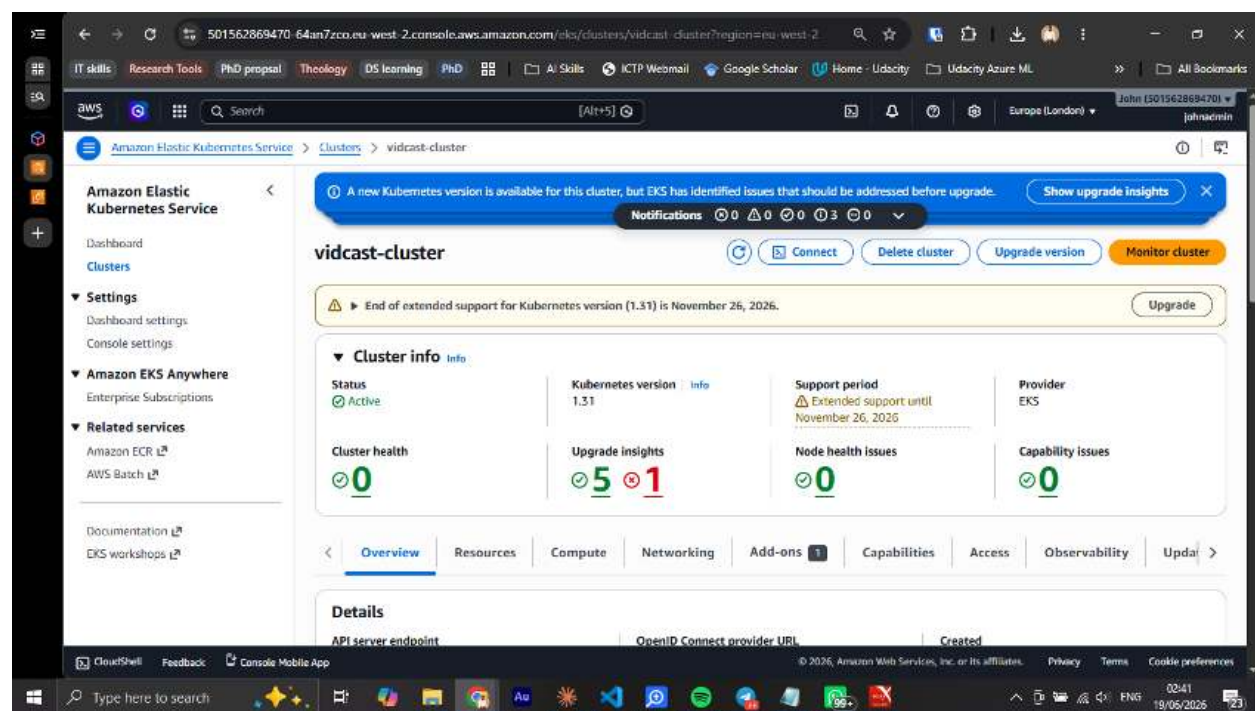
Filter by Type: All types

Policy name	Type	Attached entities
AmazonEKSClusterPolicy	AWS managed	1

CloudShell Feedback Console Mobile App

© 2026, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

Type here to search

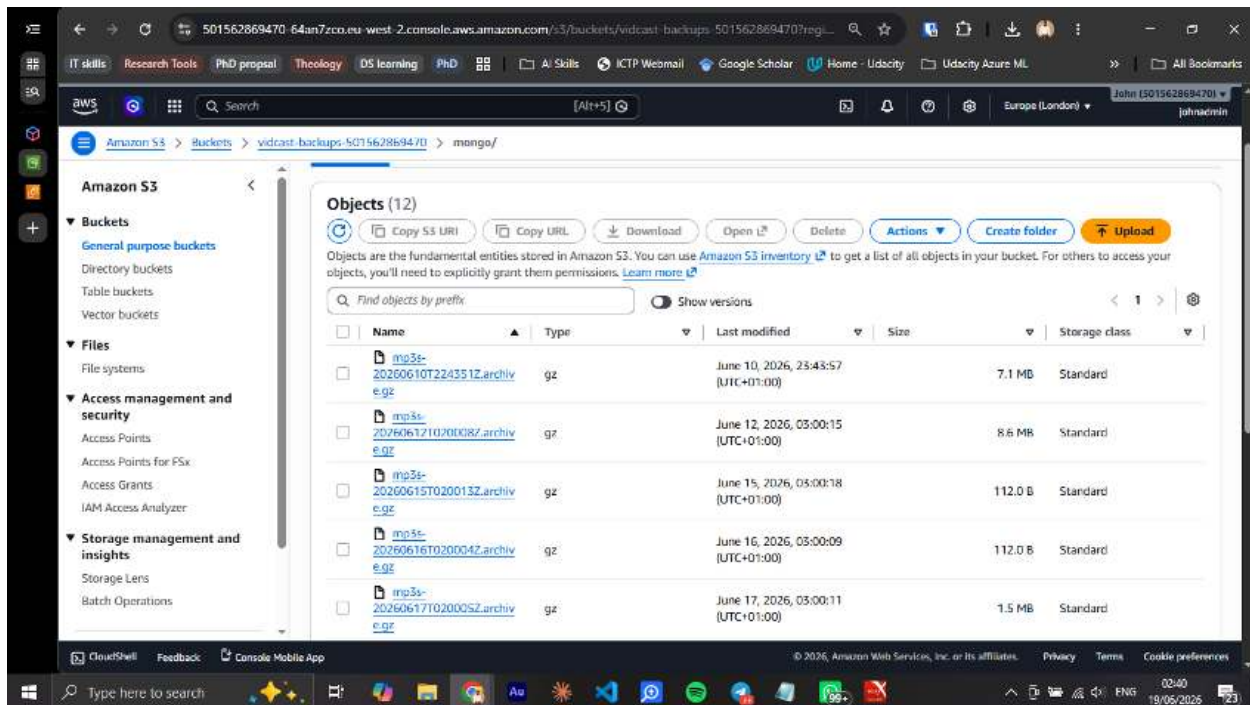


The screenshot shows the Amazon S3 console interface. The left sidebar contains navigation options: Buckets, Files, Access management and security, and Storage management and insights. The main content area displays the 'dev/' bucket. Under the 'Objects' tab, there is a table with one object: 'terraform.tfstate'. The object's details are: Type: tfstate, Last modified: June 18, 2026, 22:56:56 (UTC+01:00), Size: 128.5 KB, Storage class: Standard. Above the table, there are buttons for 'Copy S3 URI', 'Copy URL', 'Download', 'Open', 'Delete', 'Actions', 'Create folder', and 'Upload'. A search bar and a 'Show versions' toggle are also present.

Name	Type	Last modified	Size	Storage class
terraform.tfstate	tfstate	June 18, 2026, 22:56:56 (UTC+01:00)	128.5 KB	Standard

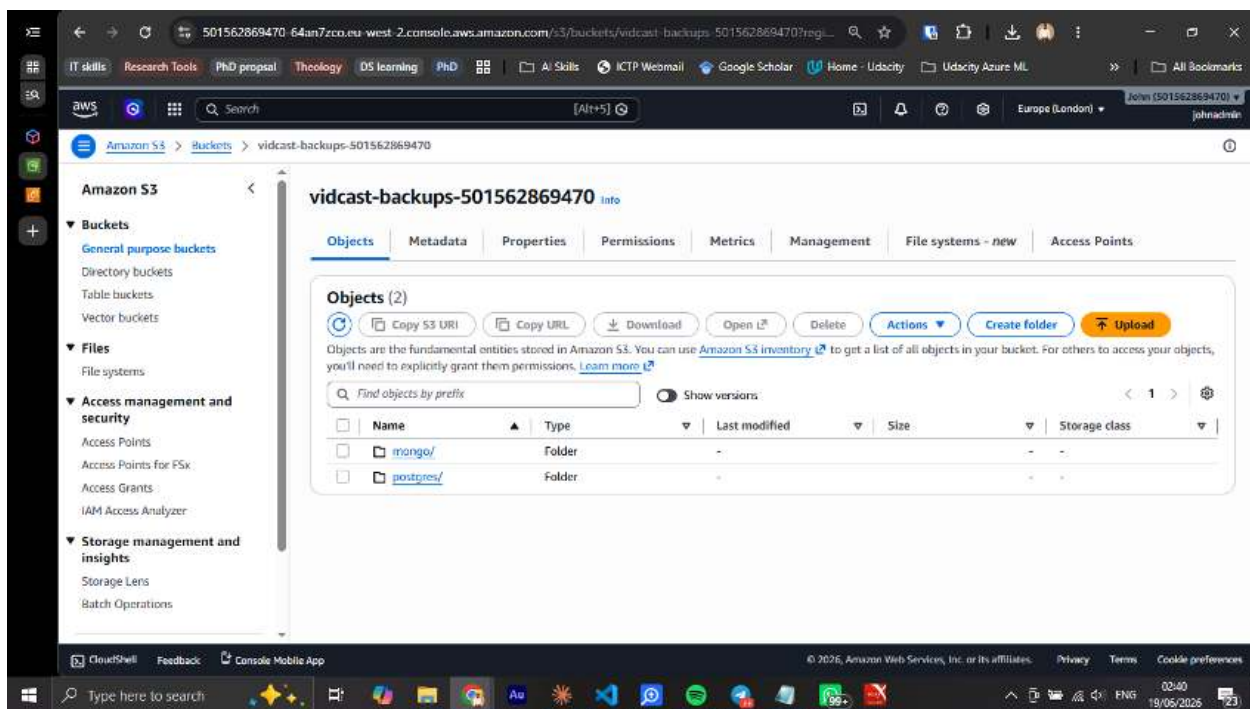
The screenshot shows the Amazon S3 console interface for the 'vidcast-backups-' bucket. The left sidebar is the same as the previous screenshot. The main content area displays the 'Objects' tab with a table of seven objects. Each object is a PostgreSQL backup file. The table columns are: Name, Type, Last modified, Size, and Storage class.

Name	Type	Last modified	Size	Storage class
postgres-20260610T223025Z.sql.gz	gz	June 10, 2026, 23:30:28 (UTC+01:00)	1.1 KB	Standard
postgres-20260611T021505Z.sql.gz	gz	June 11, 2026, 05:15:08 (UTC+01:00)	1.1 KB	Standard
postgres-20260612T021504Z.sql.gz	gz	June 12, 2026, 03:15:07 (UTC+01:00)	1.1 KB	Standard
postgres-20260615T021509Z.sql.gz	gz	June 15, 2026, 05:15:12 (UTC+01:00)	1.0 KB	Standard
postgres-20260616T021504Z.sql.gz	gz	June 16, 2026, 03:15:07 (UTC+01:00)	1.0 KB	Standard
postgres-20260617T021504Z.sql.gz	gz	June 17, 2026, 05:15:07 (UTC+01:00)	1.0 KB	Standard



The screenshot shows the Amazon S3 console interface. The left sidebar contains navigation options: Buckets, Files, Access management and security, and Storage management and insights. The main content area displays the 'Objects (12)' list for the bucket 'vidcast-backups-501562869470'. The list includes five objects, all of type 'mp3s-*.archiv.e.gz' and storage class 'Standard'.

Name	Type	Last modified	Size	Storage class
mp3s-20260610T224351Z.archiv.e.gz	gz	June 10, 2026, 23:43:57 (UTC+01:00)	7.1 MB	Standard
mp3s-20260612T000087Z.archiv.e.gz	gz	June 12, 2026, 03:00:15 (UTC+01:00)	8.6 MB	Standard
mp3s-20260615T020013Z.archiv.e.gz	gz	June 15, 2026, 05:00:18 (UTC+01:00)	112.0 B	Standard
mp3s-20260616T020004Z.archiv.e.gz	gz	June 16, 2026, 05:00:09 (UTC+01:00)	112.0 B	Standard
mp3s-20260617T020005Z.archiv.e.gz	gz	June 17, 2026, 05:00:11 (UTC+01:00)	1.5 MB	Standard



The screenshot shows the Amazon S3 console interface for the bucket 'vidcast-backups-501562869470'. The left sidebar is the same as the previous screenshot. The main content area displays the 'Objects (2)' list for the bucket. The list includes two objects, both of type 'Folder' and storage class 'Standard'.

Name	Type	Last modified	Size	Storage class
mongo/	Folder	-	-	Standard
postgres/	Folder	-	-	Standard

This screenshot shows the AWS Management Console interface for an EC2 instance. The left sidebar contains navigation links for various AWS services. The main content area displays the 'Instance summary' for instance ID i-01b7a95e94f8b09f7. The instance is in a 'Running' state. Key details include a public IPv4 address of 13.42.32.135, private IPv4 addresses 10.0.2.121, 10.0.2.146, and 10.0.2.224, and a public DNS name of ec2-13-42-32-135.eu-west-2.compute.amazonaws.com. The instance is running on the m7i-flex.large instance type within the vpc-01876334e9073551d (vidcast-cluster-vpc) and subnet-05e8bcb61f572ac9e (vidcast-cluster-subnet-1). The IAM role is vidcast-cluster-node-role.

Property	Value
Instance ID	i-01b7a95e94f8b09f7
Public IPv4 address	13.42.32.135 open address
Private IPv4 addresses	10.0.2.121 10.0.2.146 10.0.2.224
Instance state	Running
Public DNS	ec2-13-42-32-135.eu-west-2.compute.amazonaws.com open address
Private IP DNS name (IPv4 only)	ip-10-0-2-121.eu-west-2.compute.internal
Instance type	m7i-flex.large
VPC ID	vpc-01876334e9073551d (vidcast-cluster-vpc)
Subnet ID	subnet-05e8bcb61f572ac9e (vidcast-cluster-subnet-1)
IAM role	vidcast-cluster-node-role
Auto-assigned IP address	15.47.52.155 [Public IP]
Hostnames	IP name: ip-10-0-2-121.eu-west-2.compute.internal
Answer private resource DNS name	-
IPv6 address	-
Hostnames	IP name: ip-10-0-2-121.eu-west-2.compute.internal
Answer private resource DNS name	-

This screenshot shows the AWS Management Console interface for another EC2 instance. The left sidebar contains navigation links for various AWS services. The main content area displays the 'Instance summary' for instance ID i-0f2e2ab83d16bf8bd. The instance is in a 'Running' state. Key details include a public IPv4 address of 13.135.252.35, private IPv4 addresses 10.0.1.160, 10.0.1.116, and 10.0.1.128, and a public DNS name of ec2-13-135-252-35.eu-west-2.compute.amazonaws.com. The instance is running on the m7i-flex.large instance type within the vpc-01876334e9073551d (vidcast-cluster-vpc) and subnet-042bcbefeda31a934 (vidcast-cluster-public-1). The IAM role is vidcast-cluster-node-role.

Property	Value
Instance ID	i-0f2e2ab83d16bf8bd
Public IPv4 address	13.135.252.35 open address
Private IPv4 addresses	10.0.1.160 10.0.1.116 10.0.1.128
Instance state	Running
Public DNS	ec2-13-135-252-35.eu-west-2.compute.amazonaws.com open address
Private IP DNS name (IPv4 only)	ip-10-0-1-116.eu-west-2.compute.internal
Instance type	m7i-flex.large
VPC ID	vpc-01876334e9073551d (vidcast-cluster-vpc)
Subnet ID	subnet-042bcbefeda31a934 (vidcast-cluster-public-1)
IAM role	vidcast-cluster-node-role
Auto-assigned IP address	15.155.252.55 [Public IP]
Hostnames	IP name: ip-10-0-1-116.eu-west-2.compute.internal
Answer private resource DNS name	-
IPv6 address	-
Hostnames	IP name: ip-10-0-1-116.eu-west-2.compute.internal
Answer private resource DNS name	-

Instances (2) info

Last updated 3 minutes ago

Connect Instance state Actions Launch instances

Save filter sets

Choose filter set Find Instance by attribute or tag (case-sensitive)

Instance state = running X Clear filters

	Name	Instance ID	Instance state	Instance type	Status check	Availability Zone	Public IPv4 DNS
☐		i-0f2e2ab836160f8bd	Running	m7i-flex.large	3/3 checks passed	eu-west-2a	ec2-13-135-252-35
☐		i-01b7a95e94f6b09f7	Running	m7i-flex.large	3/3 checks passed	eu-west-2b	ec2-13-42-52-135

Select an instance

Auto Scaling groups ekes-vidcast-cluster-nodes-14cf6322-c5f5-a000-8891-9d7f7db565d8

Details Integrations Automatic scaling Instance management Instance refresh Activity Monitoring Tags

Launch template

Launch template [lt-02909f54a230862c7](#)
elc-14cf6322-c5f5-a000-8891-9d7f7db565d8

AMI ID [ami-0ac9e4c4fab8b5f56](#)

Instance type m7i-flex.large

Owner [arn:aws:sts::501562869470:assumed-role/AWSServiceRoleForAmazonEKSNo](#)
degrou/EKS

Version 1

Security groups -

Security group IDs [sg-009bcd9e8e1452605](#)

Create time Sun Jun 14 2026 14:15:22 GMT+0100 (British Summer Time)

Description -

Storage (volumes) /dev/xvda

Key pair name -

Request Spot Instances No

Network

Availability Zones [euw2-az5 \(eu-west-2b\)](#)
[euw2-az2 \(eu-west-2a\)](#)

Subnet ID [subnet-05d8bc67f572ac9e](#)
[subnet-042bdb04eda31a934](#)

Availability Zone distribution Balanced best effort

The screenshot shows the AWS Management Console for the 'eks-vidcast-cluster-nodes' Auto Scaling Group. The left sidebar contains navigation links for Elastic Block Store, Network & Security, Load Balancing, and Auto Scaling. The main content area displays the 'Details' tab for the Auto Scaling Group, which includes the following information:

- Launch template:** Launch template ID: lt-02909f54a230862c7, Version: 1, Description: eks-14cf6322-c5f3-a000-8891-9d7f7db565d8.
- AMI ID:** ami-0ac9e4c4fab8b5f36.
- Instance type:** m7i-flex.large.
- Owner:** arnaws:sts::501562869470:assumed-role/AWSServiceRoleForAmazonEKSDegroup/EKS.
- Security groups:** sg-009bcd9a8e1453005.
- Storage (volumes):** /dev/xvda.
- Key pair name:** No.
- Network:** Availability Zones: euw2-az5 (eu-west-2b), euw2-az2 (eu-west-2a); Subnet ID: subnet-05d8bcb61f572ac9e, subnet-042bdc4eda31a934; Availability Zone distribution: Balanced best effort.

The screenshot shows the AWS Management Console for the 'eks-vidcast-cluster-nodes-14cf6322-c5f3-a000-8891-9d7f7db565d8' Auto Scaling Group. The left sidebar contains navigation links for Elastic Block Store, Network & Security, Load Balancing, and Auto Scaling. The main content area displays the 'Capacity overview' tab for the Auto Scaling Group, which includes the following information:

- eks-vidcast-cluster-nodes-14cf6322-c5f3-a000-8891-9d7f7db565d8 Capacity overview**
- Desired capacity:** 2.
- Scaling limits:** 1 - 2.
- Desired capacity type:** Units (number of instances).
- Status:** At desired capacity.
- Date created:** Sun Jun 14 2026 14:15:54 GMT+0100 (British Summer Time).
- Launch template:** Launch template ID: lt-02909f54a230862c7, Version: 1, Description: eks-14cf6322-c5f3-a000-8891-9d7f7db565d8.
- AMI ID:** ami-0ac9e4c4fab8b5f36.
- Instance type:** m7i-flex.large.
- Owner:** arnaws:sts::501562869470:assumed-role/AWSServiceRoleForAmazonEKSDegroup/EKS.
- Security groups:** sg-009bcd9a8e1453005.
- Security group IDs:** sg-009bcd9a8e1453005.
- Create time:** Sun Jun 14 2026 14:15:54 GMT+0100 (British Summer Time).

The screenshot shows the AWS Management Console for the 'k8s-vidcast-ca0864c28b' load balancer. The 'Listeners and rules' tab is selected, displaying a table of listeners and their associated rules.

Protocol:Port	Default action	Rules	ARN	Security policy	Default
HTTP:80	Redirect to HTTPS://#{host}:443/#{path}?# (query) Status code: HTTP_301	1 rule		Not applicable	Not app
HTTPS:443	Return fixed response Response code: 404 Response body Response content type: text/plain	4 rules		ELBSecurityPolicy-2016-08	vidcast:

The screenshot shows the AWS Management Console for the 'k8s-vidcast-ca0864c28b' load balancer. The 'Details' tab is selected, displaying various configuration details.

Details	Status	VPC	Load balancer IP address type
Load balancer type Application	Active	vpc-01876334e9073551e	IPv4
Scheme Internet-facing	Hosted zone ZHURVBPSTC4K8	Availability Zones subnet-0508b0c61f572ac9e eu-west-2b (euw2-az3) subnet-042b0be4eda31a934 eu-west-2a (euw2-az2)	Date created June 14, 2026, 20:14 (UTC+01:00)

Load balancer ARN: [arn:aws:elasticloadbalancing:eu-west-2:501562869470:loadbalancer/app/k8s-vidcast-ca0864c28b/bd659b20a5e87f52](#)

DNS name: [k8s-vidcast-ca0864c28b-1555970207.eu-west-2.elb.amazonaws.com](#) [A Record]

The screenshot shows the AWS Management Console for a Security Group. The left sidebar contains navigation links for Elastic Block Store, Network & Security, Load Balancing, and Auto Scaling. The main content area displays the details for the Security Group 'sg-0b6ba1e832b0e89cd - k8s-vidcast-c87a966277'. The 'Inbound rules' tab is active, showing a table with two rules:

Name	Security group rule ID	IP version	Type	Protocol	Port range
-	sgr-05b4e1e823e4e02a5	IPv4	HTTPS	TCP	443
-	sgr-0ce71b7a3551950cb	IPv4	HTTP	TCP	80

The screenshot shows the same AWS Management Console page, but with the 'Outbound rules' tab selected. It displays a table with one rule:

Name	Security group rule ID	IP version	Type	Protocol	Port range
-	sgr-0f547636c9cc78072	IPv4	All traffic	All	All

The screenshot shows the AWS Management Console for a Security Group named **sg-009bcd9a8e1453605 - eks-cluster-sg-vidcast-cluster-20183037**. The console is in the **Security Groups** section under **Network & Security**. The **Outbound rules** tab is selected, showing one rule.

Details:

- Security group name:** eks-cluster-sg-vidcast-cluster-20183037
- Security group ID:** sg-009bcd9a8e1453605
- Description:** EKS created security group applied to ENI that is attached to EKS Control Plane master nodes, as well as any managed workloads.
- VPC ID:** vpc-01876334e9073551d
- Owner:** 501562869470
- Inbound rules count:** 2 Permission entries
- Outbound rules count:** 1 Permission entry

Outbound rules (1):

Name	Security group rule ID	IP version	Type	Protocol	Port range
-	sgr-09ad368951dc4fd80	IPv4	All traffic	All	All

The screenshot shows the AWS Management Console for the same Security Group, but with the **Inbound rules** tab selected, showing two rules.

Details:

- Security group name:** eks-cluster-sg-vidcast-cluster-20183037
- Security group ID:** sg-009bcd9a8e1453605
- Description:** EKS created security group applied to ENI that is attached to EKS Control Plane master nodes, as well as any managed workloads.
- VPC ID:** vpc-01876334e9073551d
- Owner:** 501562869470
- Inbound rules count:** 2 Permission entries
- Outbound rules count:** 1 Permission entry

Inbound rules (2):

Name	Security group rule ID	IP version	Type	Protocol	Port range
-	sgr-0c0ae298e7d9f8e6d	-	All traffic	All	All
-	sgr-0d06eb135ea1f151b	-	Custom TCP	TCP	3000 - 9090

The screenshot displays the AWS Management Console interface for the subnet-05d8bcb61f572ac9e. The console is organized into a sidebar on the left with categories like 'Virtual private cloud' and 'Security'. The main content area is titled 'subnet-05d8bcb61f572ac9e / vidcast-cluster-public-2' and contains a 'Details' section with various attributes. The 'Details' section is divided into four columns: Subnet ID, Subnet ARN, State, and Block Public Access. Below the details, there are tabs for 'Flow logs', 'Route table', 'Network ACL', 'CIDR reservations', 'Sharing', and 'Tags'. The 'Flow logs' tab is currently selected.

Subnet ID	Subnet ARN	State	Block Public Access
subnet-05d8bcb61f572ac9e	arn:aws:ec2:eu-west-2:501562869470:subnet/subnet-05d8bcb61f572ac9e	Available	Off
IPv4 CIDR	Available IPv4 addresses	IPv6 CIDR	IPv6 CIDR association ID
10.0.2.0/24	219	-	-
Availability Zone	Network border group	VPC	Route table
euw2-ez3 (eu-west-2b)	eu-west-2	vpc-01876334e9073551d vidcast-cluster-vpc	rtb-0eb9a4b4d83ea385 vidcast-cluster-public-rt
Network ACL	Default subnet	Auto-assign public IPv4 address	Auto-assign IPv6 address
acl-0ed15d7263e797b4c	No	Yes	No
Auto-assign customer-owned IPv4 address	Customer-owned IPv4 pool	Outpost ID	IPv4 CIDR reservations
No	-	-	-
IPv6 CIDR reservations	IPv6-only	Hostname type	Resource name DNS A record
-	No	IP name	Disabled
Resource name DNS AAAA record	DNS64	Owner	
Disabled	Disabled	501562869470	

The screenshot displays the AWS Management Console interface for the subnet-042bdbc4eda31a934. The console is organized into a sidebar on the left with categories like 'Virtual private cloud' and 'Security'. The main content area is titled 'subnet-042bdbc4eda31a934 / vidcast-cluster-public-1' and contains a 'Details' section with various attributes. The 'Details' section is divided into four columns: Subnet ID, Subnet ARN, State, and Block Public Access. Below the details, there are tabs for 'Flow logs', 'Route table', 'Network ACL', 'CIDR reservations', 'Sharing', and 'Tags'. The 'Flow logs' tab is currently selected.

Subnet ID	Subnet ARN	State	Block Public Access
subnet-042bdbc4eda31a934	arn:aws:ec2:eu-west-2:501562869470:subnet/subnet-042bdbc4eda31a934	Available	Off
IPv4 CIDR	Available IPv4 addresses	IPv6 CIDR	IPv6 CIDR association ID
10.0.1.0/24	219	-	-
Availability Zone	Network border group	VPC	Route table
euw2-ez2 (eu-west-2a)	eu-west-2	vpc-01876334e9073551d vidcast-cluster-vpc	rtb-0eb9a4b4d83ea385 vidcast-cluster-public-rt
Network ACL	Default subnet	Auto-assign public IPv4 address	Auto-assign IPv6 address
acl-0ed15d7263e797b4c	No	Yes	No
Auto-assign customer-owned IPv4 address	Customer-owned IPv4 pool	Outpost ID	IPv4 CIDR reservations
No	-	-	-
IPv6 CIDR reservations	IPv6-only	Hostname type	Resource name DNS A record
-	No	IP name	Disabled
Resource name DNS AAAA record	DNS64	Owner	
Disabled	Disabled	501562869470	

The screenshot shows the AWS VPC console for the VPC `vidcast-cluster-vpc` (ID: `vpc-01876334e9073551d`). The selected route table is `rtb-0eb9a4b4df83ea385`. The 'Subnet associations' tab is active, displaying two explicit associations:

Name	Subnet ID	IPv4 CIDR	IPv6 CIDR
vidcast-cluster-public-2	subnet-05d9bcb61f572ac9e	10.0.2.0/24	-
vidcast-cluster-public-1	subnet-042b7d8e4a1a51a944	10.0.1.0/24	-

Below this, it states 'Subnets without explicit associations (0)'. The left sidebar shows the VPC dashboard with options like Virtual private cloud, Internet gateways, and Security.

The screenshot shows the same AWS VPC console, but with the 'Routes' tab selected for the route table `rtb-0eb9a4b4df83ea385`. It displays two routes:

Destination	Target	Status	Propagated	Route Origin
0.0.0.0/0	igw-0863d82ac1066b87f	Active	No	Create Route
10.0.0.0/16	local	Active	No	Create Route Table

The left sidebar remains the same, showing the VPC dashboard navigation.

The screenshot shows the AWS VPC console for the VPC `vpc-01876534e9073551d`. The left sidebar contains the navigation menu with sections like **Virtual private cloud** and **Security**. The main content area displays the **VPC dashboard** with a **Resource map** tab selected. The resource map shows the VPC structure with subnets `eu-west-2a` and `eu-west-2b`, route tables `vidcast-cluster-public-rt` and `rtb-0d5e2ed516e55a430`, and network connections `vidcast-cluster-igw`. The top of the console shows the VPC ID, IPv6 CIDR, and other VPC details.

This screenshot is identical to the one above, showing the AWS VPC console for the VPC `vpc-01876534e9073551d`. It displays the **VPC dashboard** and the **Resource map** tab, which visualizes the VPC's subnets, route tables, and network connections. The interface includes the AWS navigation sidebar and the top console header with the user's profile and region.

The screenshot displays the AWS Management Console interface for a VPC. The browser address bar shows the URL: `501562869470:64an7zco.eu-west-2.console.aws.amazon.com/vpcconsole/home?region=eu-west-2#VpcDetails...`. The console header includes the AWS logo, a search bar, and the region `Europe (London)`. The left-hand navigation pane is expanded to show the `VPC` section, with sub-items like `Virtual private cloud`, `Your VPCs`, `Subnets`, `Route tables`, `Internet gateways`, `Egress-only internet gateways`, `Carrier gateways`, `DHCP option sets`, `Elastic IPs`, `Managed prefix lists`, `NAT gateways`, `Peering connections`, `Route servers`, `Security`, and `Network ACLs`.

The main content area shows the details for the VPC `vpc-01876334e9073551d / vidcast-cluster-vpc`. The `Details` tab is selected, displaying the following information:

- VPC ID:** `vpc-01876334e9073551d`
- State:** `Available`
- Block Public Access:** `Off`
- DNS hostnames:** `Enabled`
- DNS resolution:** `Enabled`
- Tenancy:** `default`
- DHCP option set:** `dopt-0207243cdf9975cc4`
- Main network ACL:** `acl-0ed15d7265e797b4c`
- Default VPC:** `No`
- IPv4 CIDR:** `10.0.0.0/16`
- Main route table:** `rtb-0d5e2ed516e55a430`
- IPv6 CIDR (Network border group):** `-`
- Network Address Usage metrics:** `Disabled`
- Route 53 Resolver DNS Firewall rule groups:** `-`
- Encryption control ID:** `-`
- Encryption control mode:** `-`
- Owner ID:** `501562869470`

Below the details, the `Resource map` tab is selected, showing a visual representation of the VPC resources. The `Resource map` includes:

- VPC:** Your AWS virtual network
- Subnets (2):** Subnets within this VPC
- Route tables (2):** Route network traffic to resources

The bottom of the console shows the `CloudShell`, `Feedback`, and `Console Mobile App` links. The footer includes the copyright notice: `© 2026, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences`. The Windows taskbar at the bottom shows the search bar and various application icons.