SEV-1 POST-INCIDENT REVIEW · 2026-06-21

# The 503 Outage

How one sick node took down `lastplayed.io` for 43 minutes — and the alert that should have caught it in two.

ROOT CAUSE · TIMELINE · PREVENTION

**SEVERITY 1 · ~43 MIN FULL OUTAGE · MITIGATED 16:27 UTC**

# Contents

PART ONE

# What happened

A factual account: what broke, who it affected, and the order events occurred — before any interpretation.

AUDIENCE · ANYONE ON CALL · 5-MINUTE READ

CHAPTER 01

# Executive summary

> A single unhealthy cluster node simultaneously killed the application's running pods and took down the private image registry they needed to restart, leaving lastplayed.io returning 503 for ~43 minutes.

Classified **SEV-1** — a total outage of a public-facing service. On 2026-06-21, `lastplayed.io` (and a second hostname that serves the same app) returned **HTTP 503 Service Unavailable** from **15:44 UTC to 16:27 UTC**. The public edge had **no healthy application pods** behind it for the duration.

The trigger was the health of **one cluster node**. Two faults on that node fired at once:

1. its **overlay network (the cluster CNI)** was crash-looping, and
2. its **node-level DNS resolution** was timing out.

Those faults produced two independent failures that combined into the outage:

> THE ONE-SENTENCE CAUSE
>
> An overlay-network fault on one node at ~15:44 UTC made the app's `/api/health` probe time out — so the orchestrator killed both app pods — and at the same moment took down the in-cluster private image registry that lives on that node, so the killed pods could not pull their image to restart.

Because the registry's storage **pins it to that one node**, it could not relocate to a healthy node, and because the app pulled its image with `imagePullPolicy: Always`, the cached image already on the worker nodes was ignored on restart. The result: the app deployment sat at **0 of 2 replicas** in `ImagePullBackOff`, and the edge returned 503.

Service was restored at 16:27 by switching the app to **serve from its cached image** (`imagePullPolicy: IfNotPresent`), which does not depend on the registry. The registry itself then recovered on its own at ~16:30 once the node's DNS cleared for long enough — but it remains **fragile**: still a single replica on an unhealthy node whose overlay network is still flapping. The user-facing outage is over; the **durable fix — repairing that node — is still outstanding**.

> HEADLINE FINDING
>
> The outage began at 15:44 and was not noticed until **16:06** — a 22-minute time-to-detect — and only because a human looked. **No automated alert fired.** For a public endpoint, an external uptime check would have paged within ~1 minute. This is the single most valuable fix to come out of this incident.

CHAPTER 02

# Impact & blast radius

> The user-visible outage was one app, but the failed dependency degraded a wider slice of the cluster.

## Directly impacted (user-facing)

| Surface | Impact | Window |
|---|---|---|
| `lastplayed.io` | **Full outage** — HTTP 503 | 15:44 → 16:27 UTC (~43 min) |
| Alternate hostname (same app, same edge) | **Full outage** — HTTP 503 | 15:44 → 16:27 UTC |

## Indirectly impacted (shared dependency)

While the registry was down, **no workload in the cluster could pull a private image**. The genuinely incident-caused collateral was contained:

| Workload | State | Caused by this incident? |
|---|---|---|
| Private container registry + Git host | Down — CI/CD push & pull blocked | **Yes** |
| Sign-in / SSO (identity service) | **Down** — pod killed 15:45, could not re-pull; blocked deploy-tool login | **Yes** |
| SSO reverse-proxy (fronts other apps) | CrashLoop — could not reach the identity service to initialise | **Yes** (knock-on) |
| Two scheduled jobs (the app's data syncs) | Failed — their trigger could not reach the 503'ing app | **Yes** |
| Three other internal apps | `ImagePullBackOff` | No — **pre-existing for days**; recovery blocked |

> DON'T OVER-ATTRIBUTE
>
> Three unrelated internal apps were **already** in `ImagePullBackOff` for days before 15:44 — a separate, pre-existing problem, not fallout from this incident. The registry outage *prevented their recovery* but did not cause their failure. They are called out so they are not mistaken for collateral.

> **WHY THIS MATTERS**
>
> A self-hosted registry is not "just another app" — it is a **control-plane dependency for every other app**. When it is down, nothing can (re)start from a cold image. The blast radius of a registry outage is the whole cluster, not one service.

## What was *not* affected

No data was lost or at risk. The database, the registry's storage volume, and all application state were intact throughout — this was an availability incident, not a data incident.

CHAPTER 03

# Timeline of events

> All times UTC, 2026-06-21. Timestamps are taken from orchestrator pod/container state and event records, not reconstructed from memory.

| Time (UTC) | Event | Evidence |
|---|---|---|
| **08:47–08:48** | The app pods start and run healthy on the worker nodes. Site is up. | container start timestamps |
| **~15:44** | An overlay-network fault on one node begins (CNI agent restart + node DNS restart). | CNI restart climb; node DNS restart |
| **15:44:00 / 15:44:10** | Both app pods' `/api/health` liveness probe times out; the orchestrator **SIGKILLs both** within 10 s of each other. | container exit code 137; "liveness probe failed: context deadline exceeded" |
| **15:44+** | The orchestrator tries to restart the pods. With `imagePullPolicy: Always`, it must pull the image — but the registry is now unreachable. Pods enter `ImagePullBackOff`. | image pull to the private registry times out |
| **15:44 → 16:27** | The app deployment is **0/2 ready**; its Service has **no endpoints**; the edge returns **503** for `lastplayed.io`. | Service endpoints empty |
| **16:06** | Outage detected **manually** while looking at something else. No alert had fired. | `curl https://lastplayed.io` → 503 |
| **16:06–16:18** | Investigation walked the chain: 503 → 0 endpoints → `ImagePullBackOff` → registry Service has 0 endpoints → registry pod stranded on the unhealthy node → CNI sandbox failure + node DNS timeout. | orchestrator describe across the app, registry, and CNI |
| **16:18** | Root cause confirmed: single sick node, dual fault, registry pinned by node-local storage. | see Chapter 04 |
| **~16:25** | Attempted clean recovery (recreate the registry pod) **failed** — the new pod could not initialise on the unhealthy node (an init-container pull was blocked by the node's DNS). Confirmed node-level repair is required for a full fix. | init-container `ImagePullBackOff`; DNS timeout |
| **16:26** | Stopgap applied: switched the app to `imagePullPolicy: IfNotPresent` to serve the **cached image**. | deployment patched |
| **16:27** | New pods come up **1/1 Ready** on the worker nodes; endpoints restored; site returns **HTTP 200** (5/5 probes). | rollout complete; `curl` 200 ×5 |
| **~16:30** | The recreated registry pod finishes initialising during a DNS-recovery window and goes **1/1 Ready**; the registry serves again. But it remains **fragile**, on a node whose overlay network is still flapping. | registry endpoint present; healthy auth challenge |
| **~16:45** | Sign-in (SSO) found to be down from the same 15:45 kill — blocking deploy-tool login. Restored with the same cached-image stopgap; identity discovery returns **200**, the SSO proxy recovers. Operator unblocked. | SSO discovery endpoint → 200 |

PART TWO

# Why it happened

The mechanism in detail — the node fault, the two failures it caused, and the design choices that turned a blip into an outage.

AUDIENCE · OPERATORS & THE PLATFORM OWNER

CHAPTER 04

# Root cause analysis

> One node's health caused two failures that, combined with two configuration choices, became a 43-minute outage.

## The trigger: one unhealthy node

The affected node also hosts the private image registry. Two host-level faults were active:

- **Overlay-network (CNI) instability.** The node's CNI agent had restarted ~20 times (most recently *during* the incident). The cause is specific and node-local: the cluster's pod network rides an overlay tunnel bound to a **per-node network interface that kept disappearing on this one node**, so the CNI agent panicked trying to rebuild the tunnel over a missing interface. On each restart the node's CNI state briefly vanished, so the orchestrator could not wire up new pod sandboxes — every new pod on that node failed to get a network sandbox.
- **Node DNS failure.** External name resolution on the node was timing out at its local resolver, so anything that needed to fetch a public base image from that node failed.

## The two failures it caused

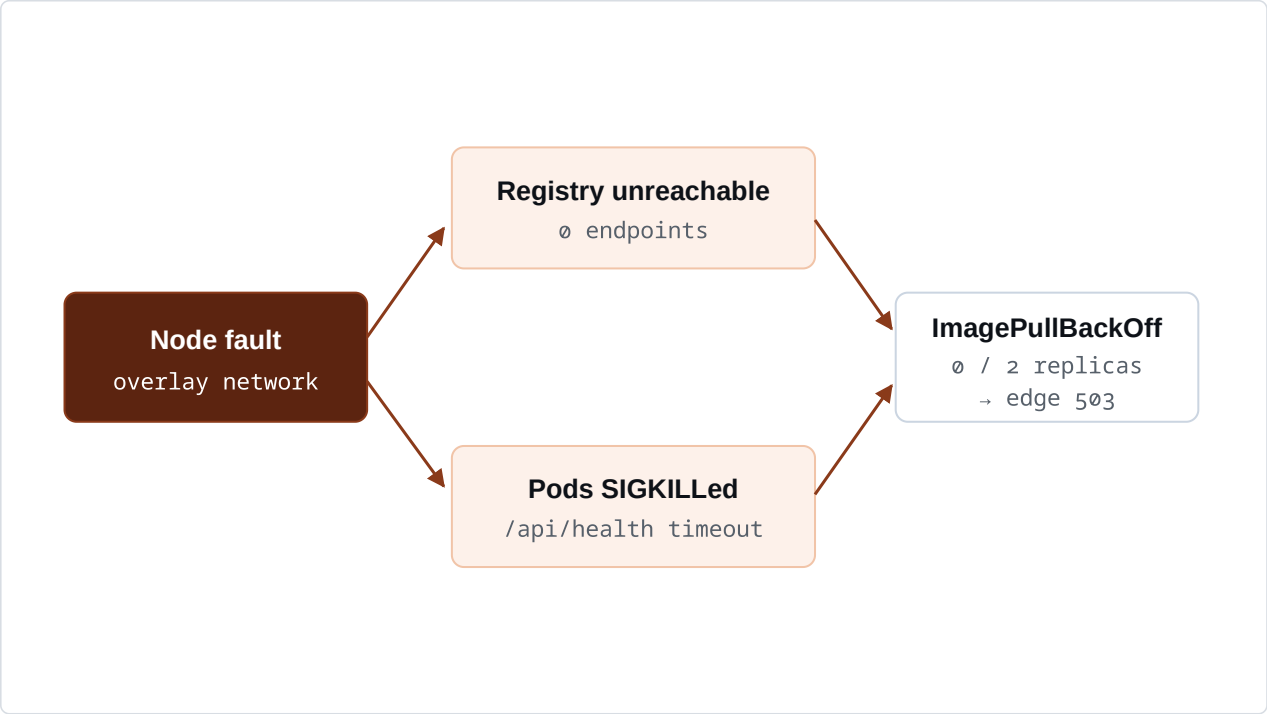


*Figure 4.1 — One node fault → two failures → one outage.*

**Failure A — the running pods were killed.** The app's liveness probe is an HTTP GET to `/api/health`. When the node's networking degraded, that probe started timing out (`context deadline exceeded`). The orchestrator treats a failed liveness probe as "the container is wedged, restart it," and **SIGKILLed both replicas** (exit code 137) within ten seconds — even though the application process itself was fine.

**Failure B — the registry went down.** The registry pod runs on the affected node. The overlay-network fault left it unable to (re)create its pod sandbox, so it sat in an unknown state and its Service had **zero endpoints**. Every request to the registry returned 503 (or timed out from inside the cluster).

## Why the two failures combined into an outage

Either failure alone would have been survivable. Together they were not, because of two design choices covered in Chapter 05:

1. The killed pods needed to **pull** to restart (Failure A needed the registry) — but the registry was down (Failure B).
2. The registry **could not move** to a healthy node to recover itself.

> **THE CORE COUPLING**
>
> The thing that recovers the apps (the registry) lived on, and shared the fate of, the same node whose failure required the apps to recover. A dependency that is needed precisely *when* it is broken is a guaranteed outage amplifier.

CHAPTER 05

# Contributing factors

> The node fault was the trigger; these four choices are what made it a 43-minute outage instead of a self-healing blip.

## 1 · The registry is pinned to one node by node-local storage

The registry's data volume uses **node-local storage** — a directory on one node's local disk. A read-write-once node-local volume can only be mounted on the node that holds it, so the single-replica registry is effectively **bolted to that node** and cannot be rescheduled when it is unhealthy.

## 2 · The app pulls `:latest` with `imagePullPolicy: Always`

`Always` forces a registry round-trip on **every** container start, even when an identical image is already cached on the node. So although the working image was sitting in the local image cache on both worker nodes (the pods had run it for ~7 hours), the restart still demanded the registry — and failed. A pinned digest with `IfNotPresent` would have let the cache serve.

## 3 · Liveness probe couples app life to a flaky dependency

A liveness failure is a death sentence — the container is killed. Tying liveness to `/api/health` (which can hang on transient node/network/dependency issues) means a **network blip kills all replicas at once**, exactly when you least want to lose capacity. There is no startup probe, generous `failureThreshold`, or separation between "the process is dead" (liveness) and "the dependency is slow" (readiness).

## 4 · Single points of failure with no redundancy

The registry runs **one replica** with no anti-affinity, disruption budget, or replicated storage. There is no pull-through cache or mirror, so there is no second path to an image when the one registry is down.

> **REFRAME**
>
> Three of these four are cheap to change and would each, independently, have prevented the outage: pin a digest + `IfNotPresent` (apps survive a registry outage), relax the liveness probe (a blip doesn't kill all replicas), and run a pull-through cache (a registry outage stops blocking pulls).

CHAPTER 06

# Detection & response

> The fix was fast once someone was looking; the problem is that no one was told to look.

## Detection was manual and late

The outage started at 15:44 and was found at 16:06 — **22 minutes**, and only because a human happened to check. There was **no page, no email, no chat alert**. For a public endpoint this is the weakest link in the whole incident: the metrics needed to fire an alert *already existed* (edge 5xx counters, cluster-state metrics, a running metrics stack) — they simply were not wired to a rule and a notifier.

## Diagnosis was efficient

Once investigating, the path from symptom to root cause took ~12 minutes by walking the dependency chain backward:

```
503 at the edge
  → the app Service has 0 endpoints
    → pods in ImagePullBackOff (0/2)
      → image pull to the private registry times out
        → the registry Service has 0 endpoints
          → the registry pod is stranded on the unhealthy node
            → CNI sandbox failure + node DNS timeout
```

> **WHY "503" WAS A RED HERRING**
>
> A 503 from the edge almost never means "the app crashed." It means "**no ready endpoint to route to.**" The right first question is always "does this Service have endpoints?", not "show me the app logs" — the app never got far enough to log anything.

## Response friction: the dashboards were degraded too

The observability dashboards sit behind the same SSO reverse-proxy that was itself crash-looping during the incident. The dashboard service's own process was up, but the **access path was degraded exactly when it was needed**. Investigation proceeded via direct cluster access; a less-equipped responder would have been blocked at the front door.

RECOVERY-DEPENDENCY LOOP

The same node fault also took down the **sign-in / SSO** service (its pod was killed at 15:45 and could not re-pull). That blocked logging into the **deploy tooling**, which is where the manifests, the app builds, and the very change that would *apply the fix* live. In other words, the incident disabled the tooling needed to recover from it. Restoration was only possible because direct cluster access is an **out-of-band path that does not depend on the SSO**. The lesson: every recovery path must have a fallback the incident itself cannot also take down — and the operator should not have to log in through the failing front door to fix the back end.

CHAPTER 07

# How service was restored

> Restore the user-facing service first with a reversible stopgap; repair the root cause second.

## The stopgap (applied 16:26 UTC)

The killed pods could not pull, but the image they needed was **already cached** on the worker nodes (they had been running it since 08:48). Switching the pull policy to `IfNotPresent` let the orchestrator use that cache instead of the dead registry. New pods started from cache on the worker nodes, passed readiness, and the Service regained endpoints. `lastplayed.io` returned **HTTP 200** within ~60 seconds, verified across five consecutive probes (2.5 s warming → 0.45 s warm).

> **THIS IS A STOPGAP, NOT THE FIX**
>
> With `IfNotPresent` on a `:latest` tag, the app will keep serving the cached image but **will not pick up new deploys** until the registry is healthy and the policy is reconsidered. The registry, the scheduled jobs, and the other private-image apps are still down. The durable fix (Chapter 09) is still required.

## What did *not* work (and why it's informative)

Recreating the registry pod *initially* failed: it rescheduled onto the same unhealthy node (forced there by its node-local volume) and its init-container could not pull because of the node's DNS fault — which is why the user-facing fix had to bypass the registry entirely. That same recreated pod then **succeeded a few minutes later** (~16:30), once the node's DNS happened to clear long enough for the init pull, and the registry came back. The lesson stands: recovery depended on the node's intermittent good luck, not on any deliberate repair — the registry **still cannot survive that node being unhealthy**, so the host-level fix in Chapter 09 remains mandatory.

PART THREE

# Preventing recurrence

The gaps this incident exposed, and the concrete work that closes them — ordered by leverage.

AUDIENCE · THE PLATFORM OWNER · BACKLOG INPUT

CHAPTER 08

# Observability gaps

> Every gap below is something that, had it existed, would have caught or shortened this incident. The metrics mostly already exist — what is missing is alerting rules and a few synthetic checks.

## A · Detection — nothing told anyone the site was down

| # | Gap | Signal that should exist |
|---|---|---|
| A1 | **No external/synthetic uptime check** on `lastplayed.io`. The 22-minute TTD is entirely this. | Blackbox HTTP probe from outside the cluster, alert on non-200, <1 min. |
| A2 | **No alert on edge 5xx rate.** The edge already exports it; 100% 503 on a host went unwatched. | Alert on the 5xx request-rate per host. |
| A3 | **No alert on "Service has zero endpoints."** This is the most direct predictor of a 503. | Alert when an edge-backed Service has 0 ready endpoints. |
| A4 | **No alert on Deployment availability.** The app sat at 0/2 silently. | Alert when available replicas < desired. |

## B · Diagnosis — the failing dependencies were all silent

| # | Gap | Signal that should exist |
|---|---|---|
| B1 | **No alert on `ImagePullBackOff` / pods not Ready.** Many pods were stuck with no page. | Alert on the count of pods waiting on image pulls. |
| B2 | **No health check on the registry itself.** A control-plane dependency had 0 endpoints, unwatched. | Synthetic pull probe of the registry; alert on its endpoint count. |
| B3 | **No alert on CNI / node flapping.** The overlay agent restarted ~20× and emitted sandbox-creation failures with no signal. | Alert on CNI-agent restart rate and Node `Ready` flaps. |
| B4 | **No alert on node-level DNS failure.** The resolution timeouts were invisible. | Per-node probe of external resolution; cluster-DNS error/latency. |

## C · Meta — the monitoring stack and the apps themselves under-signal

| # | Gap | Signal that should exist |
|---|---|---|
| C1 | **The dashboards depend on a failing auth path.** Observability behind a crash-looping SSO proxy is unreachable during an incident. | A break-glass/no-auth read path, or decouple SSO from the observability UI. |
| C2 | **No app-side anomaly alert.** Product analytics and error tracking are wired, but a drop-to-zero in events / spike in server errors did not page. | "No events received in N min" + an error-rate alert. |
| C3 | **No alert on scheduled-job failure.** The data-sync jobs failed silently. | Alert on failed jobs / stale last-success timestamps. |
| C4 | **No SLO / error-budget burn alert** for the public endpoint, so there is no policy threshold that forces a page. | Multi-window burn-rate alert on a 99.x% availability SLO. |

> THE 80/20
>
> Three rules would have caught this within a minute: **(A1)** external uptime on lastplayed.io, **(A2)** edge 5xx rate, and **(A3)** zero-endpoints. Everything else shortens diagnosis; these three are the difference between a 1-minute and a 22-minute time-to-detect.

CHAPTER 09

# Action items & the durable fix

> Ordered by leverage: stop the bleeding, then make this class of failure self-healing, then make it impossible to miss.

## Now — finish recovering (still outstanding)

1. **Repair the affected node — stabilise its overlay-network interface.** The CNI crash-loop has a precise cause: the pod-network overlay rides a per-node interface that keeps dropping on this node, and the CNI agent panics trying to rebuild over a missing interface. Fixing that interface is the keystone — the node's DNS time-outs almost certainly share the same cause, since cross-node traffic traverses the overlay.
2. **Verify the registry stays healthy.** It has already recovered — but it will go down again on the next overlay flap. Do not treat it as resolved until the node is fixed.
3. **Revert the stopgap deliberately.** Once the registry is healthy, move the app to a **pinned digest +** `IfNotPresent` (not back to `Always` on `:latest`) so a future registry outage can never again block a restart.

## Next — make a registry outage survivable (resilience)

4. **Decouple app restart from the registry** everywhere: pin digests, prefer `IfNotPresent`, and consider pre-pulling critical images onto every node.
5. **Run a pull-through cache / mirror** so image pulls have a second source.
6. **Get the registry off node-local storage** (replicated/shared volume or an off-node managed registry) so it can reschedule; add a disruption budget and, if feasible, a second replica.
7. **Fix the liveness probe**: add a startup probe, raise `failureThreshold` / `timeoutSeconds`, and ensure liveness reflects only "process is wedged," not "a dependency is briefly slow." Bound the rollout so probes can't take down all replicas at once.

## Then — make detection automatic (the highest-leverage fix)

8. **Wire the three detection alerts** (Chapter 08: A1 external uptime, A2 edge 5xx, A3 zero-endpoints) to a notifier that actually pages.
9. **Add the diagnosis alerts** (B1–B4) so the next responder is told *where* it broke, not just *that* it broke.
10. **Give observability a break-glass path** (C1) so dashboards/logs are reachable when SSO is down, and add the app-side anomaly + job-failure + SLO alerts (C2–C4).

> **IF ONLY THREE THINGS GET DONE**
>
> (8) external uptime alert, (3/4) pinned-digest + `IfNotPresent`, and (7) a saner liveness probe. The first means you find out in a minute; the second and third mean this exact chain can't form again.

**STATUS AT TIME OF WRITING**

`lastplayed.io` is **up** (2/2, serving the cached image) and the registry has **come back on its own** — but on borrowed time: it is single-replica on an unhealthy node whose overlay network is **still flapping**, so the same chain can re-form at any moment. Until that node is actually repaired (action items 1–2, which need host-level access), this is a **mitigated** SEV-1, not a **resolved** one. Treat the node fix as urgent, not optional.