KubeCon CloudNativeCon India 2026

#KubeCon #CloudNativeCon

From Flaky Builds to Self-Healing Deployments

An OTel-Driven CI/CD Tale

Chamod Perera & Shivay Lamba

KubeCon CloudNativeCon India 2026

Chamod Perera

Software Engineer II at Circles
CNCF Ambassador
Jenkins GSoC Mentor

Shivay Lamba

CNCF Ambassador
Jenkins GSoC Org Admin
Docker Captain

The Lie the Dashboard Tells

99.9%

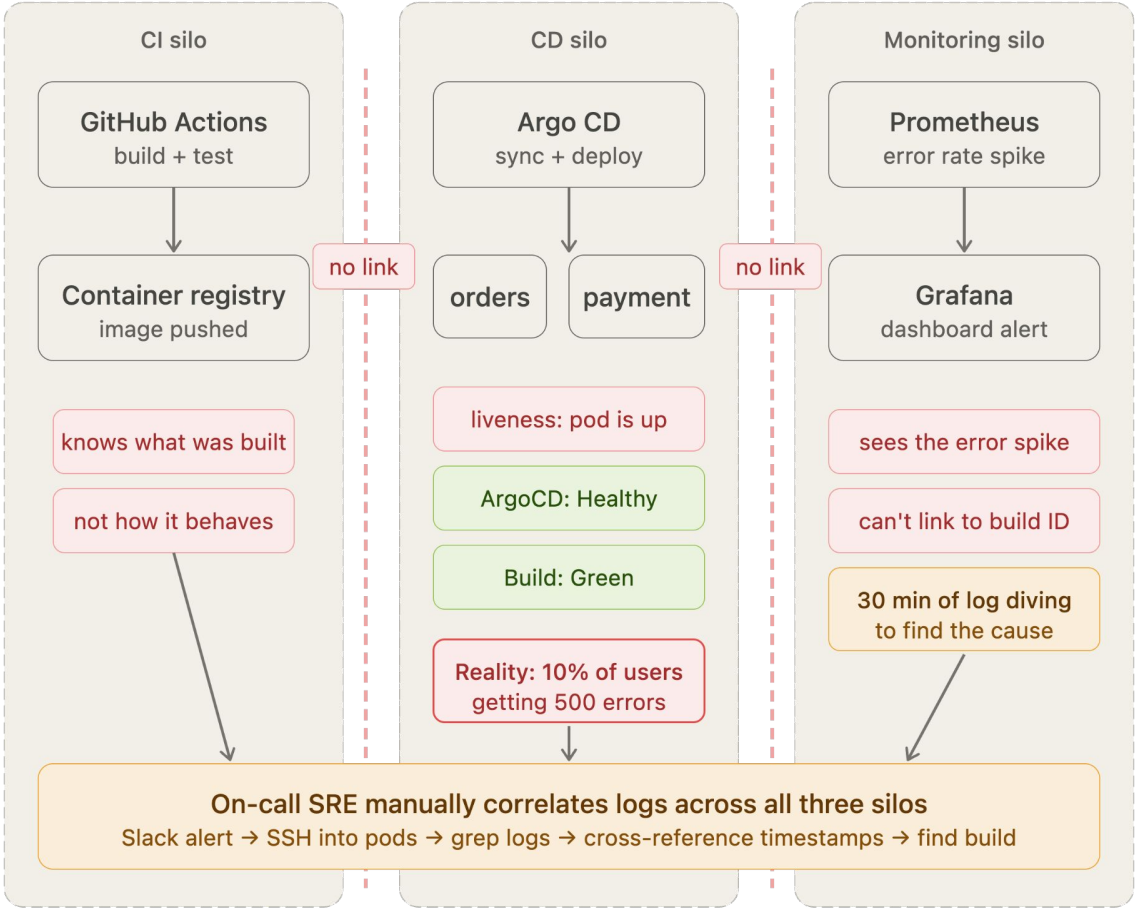
"uptime" on the dashboard

Meanwhile…

developers quietly lose hours to flaky builds and silent deployment failures.

Sound familiar?

- ✓ Dashboards say everything is green
- ⚠ Builds flagged "flaky" - rerun and pray
- ✕ Deployments "succeed" but quietly break
- ⏱ Hours lost to manual log-diving to find out why



Each tool sees one layer. No shared identity connects build → deploy → runtime.

github actions log

```
[14:31:05] Checkout commit d3a4db4
[14:32:41] Build payment image — done
[14:33:20] Push ghcr.io/org/payment
[14:33:45] Job succeeded — SUCCESS
```

payment stdout

```
14:35:17 approved (42ms)
14:35:19 ERROR: dependency
mismatch: inventory
schema v2 vs payment v3
14:35:21 approved (39ms)
```

orders stdout

```
14:35:19 status=500
  build_id=unknown
14:35:19 ERROR: payment
unavailable
```

the developer's reasoning

"build succeeded at 14:33:45..."

"error in payment at 14:35:19, is that the same deploy?"

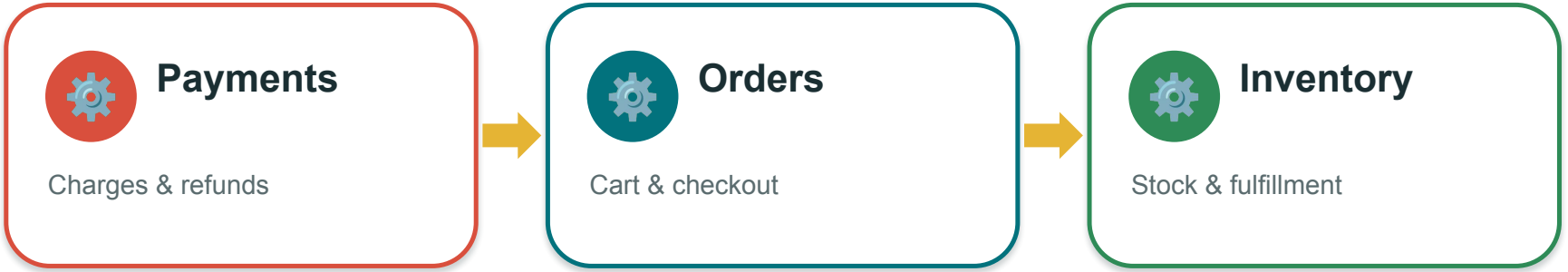
"orders just says build_id=unknown — no help"

"is this commit d3a4db4 or an older one still running?"

~30 min later: still guessing

The Setup: Our Microservices

One CI/CD pipeline running across three linked services on multiple clusters



⚡ **One shared CI/CD pipeline** — build → test → deploy — fans out across all three services and every cluster.

A failure in one service can ripple through the others, and the pipeline reports each stage in isolation.

The Incident: It Was a Friday (of course)

A high-priority update ships to the **payments** microservice…

✓ **CI**

Build marked **GREEN**

✓ **CD**

Deployment marked **HEALTHY**

! **The reality: 10% of users hit 500 errors**

Every dashboard insisted nothing was wrong while the business silently bled money on failed transactions.

Anatomy of a “Gray Failure”

What actually broke

- A subtle dependency mismatch in the new code
- Triggered only under specific production load
- Invisible to Kubernetes liveness probes
- System stayed “Green” the entire time

Why liveness probes lied

✓ What it checked

Is the process **up**? - Yes.

✕ What it never checked

Is the process actually **functional**? - Nobody asked.

The Real Problem: Observability Silos

Three tools, three blind spots, zero shared context

CI

Knew WHAT was built but nothing about how it behaved in production.

Kubernetes

Knew the container was running but not WHY it was slow or degraded.

Prometheus

Showed the error spike but couldn’t link it to a Build ID or Git commit.

→ 30 minutes of manual log-diving just to connect an error back to a commit.

The Insight: The Trace ID as the Golden Thread

Make one **Trace ID** the thread that stitches Build → Deploy → Runtime into a single distributed trace.

traceparent: 00-4bf92f3577b34da6a3ce929d0e0e4736-...

One ID, carried end-to-end, means any runtime error points straight back to the exact build and commit that caused it.

Re-Architecture · Part 1

Trace Context Propagation

KubeCon CloudNativeCon India 2026

GitHub Actions becomes the trace root

1. GitHub Actions builds the image, tags it with build_id, and pushes to GHCR
2. The workflow patches the Argo Rollout, injecting build_id as a pod env var
3. Build-time metadata (build_id, git_sha) rides along into runtime spans

ci.yml (excerpt)

```
- run: docker push $IMAGE:$BUILD_ID
- run: kubectl patch rollout payment \
    --type=json -p='[{"op":"replace",
    "value":"$BUILD_ID"}]'
  # build_id + git_commit attached
  # OTLP span emitted to Collector
```

Re-Architecture · Part 2

The Golden Signal Check

Instead of “Is the pod running?”, the gate asks the data that actually matters:

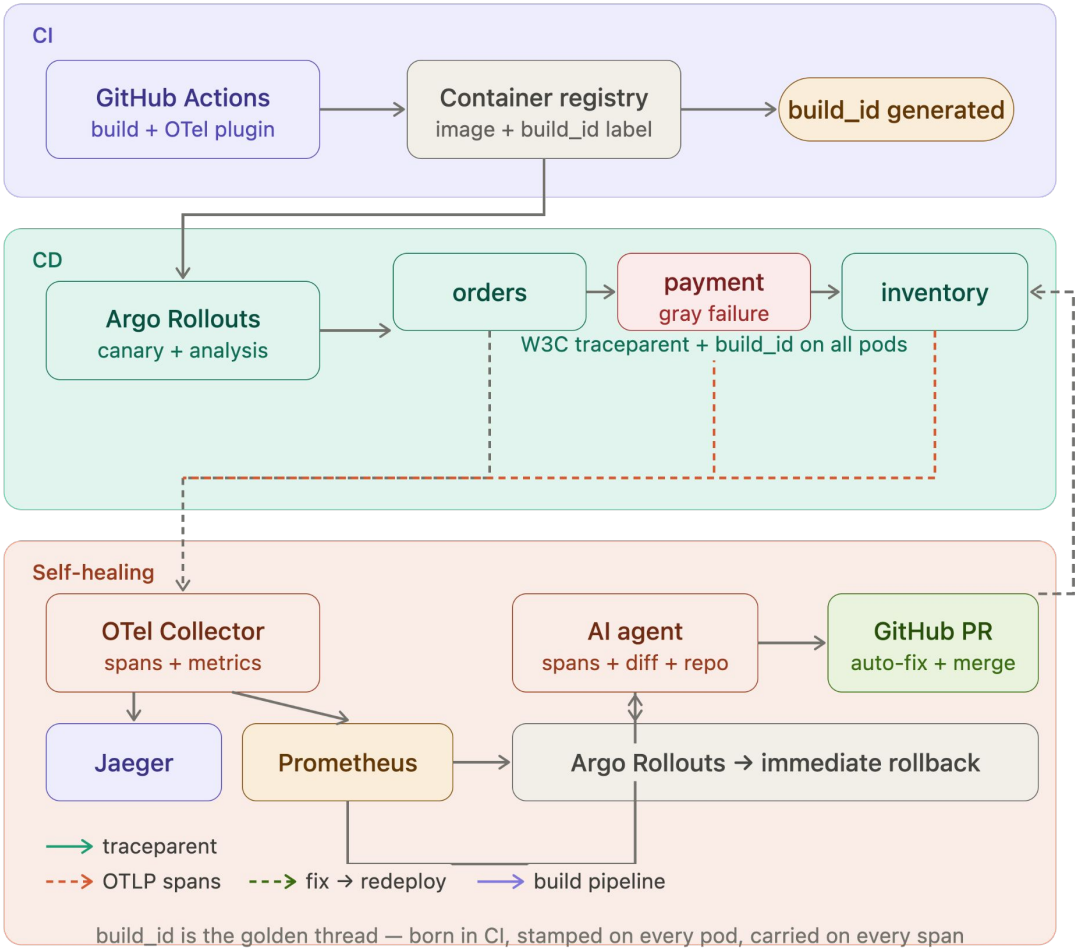
“Are the spans tagged with this build_id reporting an error rate > 1%?”

Before

Argo Rollouts only checked pod liveness — a healthy process passed the gate even while users got errors.

After

A custom AnalysisTemplate queries the OTel backend for live error rate on this exact build before promoting.


CI
GitHub Actions
build + OTel plugin
Container registry
image + build_id label
build_id generated
CD
Argo Rollouts
canary + analysis
orders
payment
gray failure
inventory
W3C traceparent + build_id on all pods
Self-healing
OTel Collector
spans + metrics
AI agent
spans + diff + repo
GitHub PR
auto-fix + merge
Jaeger
Prometheus
Argo Rollouts → immediate rollback
traceparent
OTLP spans
fix → redeploy
build pipeline
build_id is the golden thread — born in CI, stamped on every pod, carried on every span

Before & After: Catching the Failed Trace

✕ Before

- Build, deploy and runtime logged in separate silos
- No shared ID linking an error to its commit
- ~30 min of manual log-diving per incident
- Failures surface only after users complain

✓ After

- Build metadata stitched to runtime spans automatically
- Every span carries build_id + git_sha
- Failed trace points straight to the bad commit
- Degradation caught on the canary, before rollout

Two Weeks Later: The Bad Commit Returns

Another regression gets pushed but this time the platform is watching.

GitHub Actions builds it

pushes to GHCR and patches the canary with build_id

OTel Collector sees it first

span errors appear, tagged with that specific build_id

SLA breach detected

error rate crosses the 1% threshold on the canary

The Self-Healing Moment

Rollback triggered automatically

The GitOps controller saw the error spans, recognized the SLA breach, and reverted the canary on its own.

…before a human even looked at the Slack alert.

0	<1%	seconds
humans paged	users impacted	to roll back

The Fix-Forward Moment

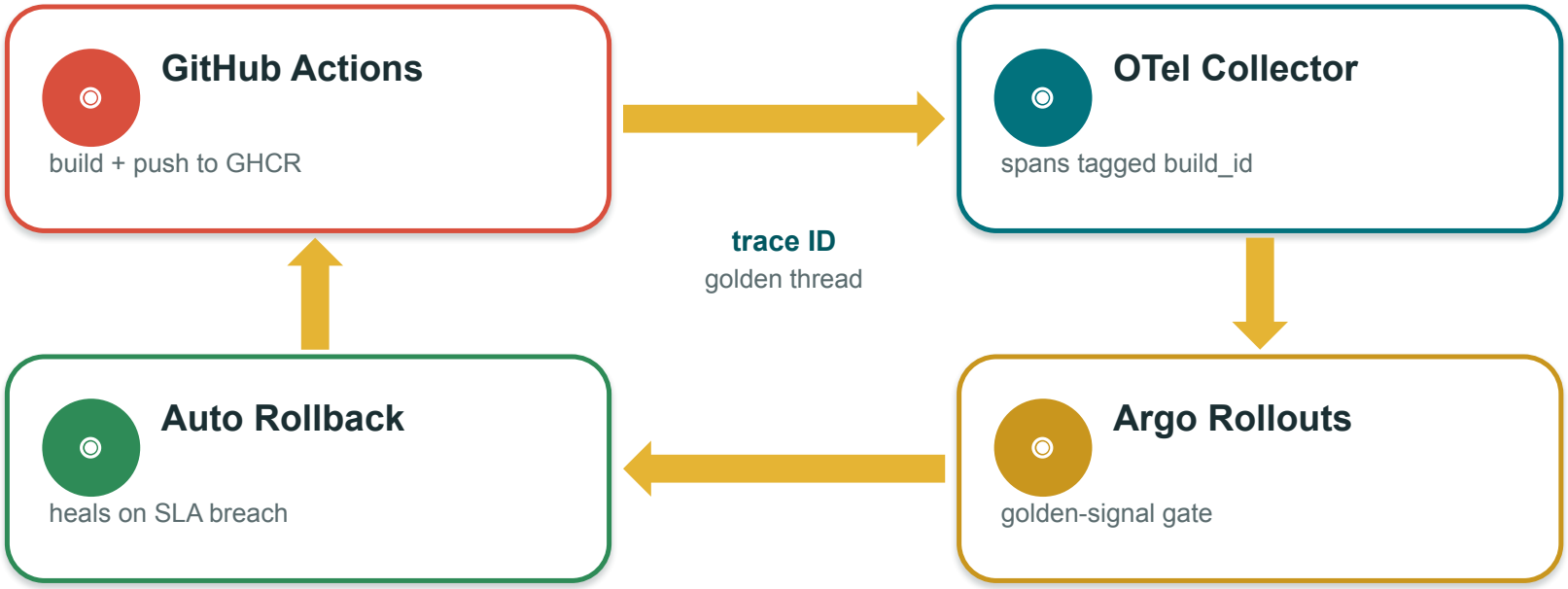
The AI SRE agent investigates and opens a fix

It reads Loki logs and Prometheus metrics scoped to the failed build_id, names the offending commit, and opens a pull request with the fix.

Merging the PR redeploys through the same OTel gate.

1	1	same gate
PR opened, root cause named	keystroke to approve and merge	validates the fix before promoting

Architecture Recap: Closing the Loop



Benefit to the Ecosystem

Vendor-neutral

Built on OpenTelemetry - works with any backend, no lock-in.

Faster MTTR

Errors point straight to the commit; no more 30-minute log hunts.

Truly self-healing

Pipelines roll themselves back before customers or SREs notice.

Pipelines as traces

Treat CI/CD as a distributed system - observe it like one.

Key Takeaways

KubeCon CloudNativeCon India 2026

1

Make the Trace ID your golden thread

Carry one ID from build through deploy to runtime.

2

Check golden signals, not just liveness

“Running” is not “working” - gate on error rate.

3

Let the platform page itself

Wire OTel traces into GitOps so rollbacks happen automatically.

KubeCon CloudNativeCon India 2026

Thank You

Connect with us:
Shivay: @howdevelop
Chamod: @chamodshehanka