ACT 1 · Three Production Fixes

ACT 2 · The AI Layer We Built On Top

From Firefighting to Foresight

Re-engineering Our P1 Incident Response Pipeline

Rajesh Balaji

Software Engineer III (Senior Software Engineer) — Costco Wholesale Corporation, USA

Independent Researcher | Seattle, WA, USA — Cloud Architect | AIOps Practitioner | AI/ML & Cybersecurity Researcher

IEEE Senior Member | Member, Institution of Engineering and Technology (MIET) | ACM Professional Member | Sigma Xi Member | Raptors Fellow

30-minute talk · Pre-recorded · Production-grade engineering

2:14 AM

PagerDuty. P1. Site-impacting.

> It wasn't bad code. It was a deployment nobody outside one team could see, going out without an enforced checklist, into a system whose first warning sign was a customer complaint — not a dashboard.

STEPPING BACK

Different Incidents. Same Three Gaps.

Every P1 had a different proximate cause — a missed flag, a bad migration order, a rollback that didn't roll back a dependency. But asking "where did this become unrecoverable" surfaced the same three answers, every time:

✗

No enforced checklist

Tribal knowledge lived in people's heads, not the pipeline. Senior engineers knew the steps. Junior engineers on a Friday night deploy did not.

✗

No unified view

Incident commanders spent the first fifteen minutes of every P1 just assembling a picture of what was happening — stitching together five different tools mid-crisis.

✗

No forward signal

Nobody could see what was about to deploy next, org-wide. A hold on one service meant nothing if another team's deploy was scheduled to hit it in twenty minutes.

01

Deployment Checklist Gate

An enforced stop in the pipeline — not a wiki page, not a Confluence doc. The pipeline hard-stops if required steps aren't completed. No exceptions.

02

Unified Incident Dashboard

One correlated view of service health, recent deploys, and active alerts — time-aligned so you can visually see an alert light up after a specific deploy went out.

03

Org-wide Deployment Tracker

Forward visibility into every team's upcoming deploys, in one place. The forward-looking complement to the dashboard, which only ever shows the past.

PROBLEM 1

Fix 1 — The Deployment Checklist Gate

⚠ THE GAP

- Every senior engineer could recite the pre-deploy checklist from memory. That was exactly the problem.
- Tribal knowledge doesn't survive a 2 AM deploy. It doesn't transfer when the on-call engineer is junior, or covering for someone on PTO, or just distracted.
- The checklist existed. It lived in a Confluence doc that was last updated eighteen months ago. Nobody checked it under pressure. And the pipeline had no way to enforce it.

✓ THE FIX

- An enforced gate inside the deployment pipeline itself.
- If a required step isn't checked, the pipeline hard-stops — no workarounds, no 'I'll do it after.' It returns to the step that failed.
- Scoped deliberately: only the steps that kept showing up in post-mortems as 'this would have caught it.' Not every possible check — just the ones that mattered.

RESULT

Missed-step P1s — the category that traces to 'we forgot to verify X' — dropped to near zero within a month. Because forgetting became structurally impossible.

PROBLEM 2

Fix 2 — The Unified Incident Dashboard

⚠ THE GAP

- During a live P1, incident commanders were losing fifteen minutes at the start of every incident just assembling a picture of what was happening.
- Deployment logs in one tool. Service health in another. Active alerts in a third. A fast-moving Slack thread that was simultaneously the main comms channel and the main source of noise.
- Fifteen minutes lost is fifteen minutes the incident runs unmanaged. At 2 AM, that's expensive.

✓ THE FIX

- One dashboard that time-aligns existing signals: current service health, what deployed recently and where, and which alerts are active — correlated by timestamp.
- No new data sources. The signals already existed. The win was putting them in one place and aligning them on a shared timeline.
- If an alert fires twelve minutes after a specific service deployed, you can see that relationship visually, immediately, without opening five tabs.

RESULT

Time from 'something's wrong' to 'I know roughly why' collapsed significantly. Every signal already existed. Correlating them by time, in one view, was the entire win.

PROBLEM 3

Fix 3 — The Org-wide Deployment Tracker

⚠ THE GAP

- Picture this: you've stabilized an incident at 2 AM. The system is holding. You're monitoring carefully.
- Twenty minutes later, an unrelated team's scheduled deploy hits a service your stabilization plan depends on staying still. Nobody did anything wrong. Nobody had visibility.
- The dashboard only shows what's happened. It has no opinion about what's about to happen. That blind spot cost us incidents we had no structural way to prevent.

✓ THE FIX

- A live, org-wide view of every team's upcoming deploys — not just your pipeline, but everyone's — accessible in one place.
- Includes: which service, which team, scheduled time, blast radius estimate.
- One-click hold request: an incident commander can flag a conflict and request a hold without a Slack thread, without tracking down the deploying team's on-call, without a delay.

RESULT

The conversation that used to happen too late — or not at all — now happens before impact. This fix required the most political capital of the three. It was also the one that mattered most.

Lessons learned building all three

Visibility, not tooling

Every fix started as 'add more monitoring' and ended up being about process and intent, not system state. We didn't need better tools. We needed the existing tools to talk to each other, and the existing processes to be enforced by the system rather than remembered by humans.

Build order matters — and it's political

We built the checklist first because it felt tractable. The tracker mattered most but required the most political capital. If we'd built the tracker first, we might not have had the credibility to push teams to adopt it. Sequence your fixes by what builds trust, not just what's technically easiest.

Green metrics can still page you

You can be fully green on every dashboard and still be twenty minutes away from a P1. Because what's breaking isn't a system — it's coordination. Visibility gaps don't show up as alerts. They show up as incidents that could have been prevented if two teams had known about each other.

THE THREE FIXES WORKED

✓ Missed-step P1s dropped to near zero

✓ Fifteen-minute assembly period mostly gone

✓ Cross-team deploy conflicts caught proactively

✓ Engineers had better information, faster

But something was still missing.

The checklist stopped bad deploys.
The dashboard showed what had happened.
The tracker showed what was coming next.

None of them could tell you that a deploy which looked completely fine was quietly causing a memory drift that would breach your SLO six hours later.

THE QUESTION THAT REMAINED

The data to predict that memory drift was already in our telemetry. The metric was climbing from minute thirty after the deploy. Steadily. Slowly. But climbing.

Our tools stored it. Nobody asked the right question of it.

So we built Act 2.

Not to replace the three fixes — those stayed. But to add a layer on top: one that could look at the telemetry we were already collecting and say, before the breach, that something was heading in the wrong direction.

Stop asking 'what broke.'
Start asking 'what's about to.'

From threshold alerts to drift forecasting

BEFORE — Threshold-based (reactive)

- ✗ Metric stored. Threshold not crossed. Nothing happens.
- ✗ Alert fires when breach occurs — after the damage
- ✗ Manual log analysis under pressure, across multiple tools
- ✗ No link between deploy events and metric drift
- ✗ Knowledge of past incidents: in engineers' heads

VS

AFTER — Trend-based (predictive)

- ✓ Metric stored AND rate-of-change computed continuously
- ✓ Predicted breach surfaced 6–48 hrs ahead, with an ETA
- ✓ AI auto-links metric drift to recent deployments
- ✓ RAG copilot answers 'what's most likely to break this week?
- ✓ Every resolved incident captured to a searchable knowledge base

The goal: shift the on-call engineer from incident responder → incident preventer. The three fixes made response better. This layer makes prevention possible.

From raw telemetry to a specific breach prediction — five steps

INGEST

Dynatrace streams GKE + AKS logs, metrics and traces into a FastAPI backend every 30 seconds. Every metric, every service.

→

TREND

Rolling window analysis (6h / 24h / 7d) computes rate-of-change on each metric. Not 'where is it now' — 'where is it going'.

→

FORECAST

Linear + exponential smoothing projects each metric forward. Output: predicted value at T+N hours, with a confidence band and a breach ETA.

→

CORRELATE

Drift events are cross-referenced with deploy history. Metric drifting within 2h of a deploy? Flagged deployment-linked — automatically.

→

SURFACE

A forecast card: metric name, current value, predicted breach time, linked deploy, confidence tier, recommended action from similar past incidents.

WHAT A FORECAST CARD LOOKS LIKE:

📊 E-Commerce · Avg Response Time · FORECAST Confidence: ● URGENT

Current: 760 ms ↑ Trending: +38.6%/hr Predicted breach (1,000 ms SLO): 6h 22min from now

Linked deploy: v2.14 pushed 3h 47min ago by E-Commerce team

Recommended action (from 8 similar past incidents): Scale node pool · Restart affected pods

The Confidence Score Problem

We shipped accurate predictions. Engineers didn't act on them. Here's why — and how we fixed it.

Failure 1 — Engineers ignored low-confidence predictions

Scores ranged 60–95%. Engineers mentally filtered anything under 80%. A real 73%-confidence prediction for a genuine memory drift was dismissed. The P1 fired six hours later. The system was right. The percentage made it look uncertain enough to ignore.

"If the AI isn't sure, why should I stop what I'm doing?"

Failure 2 — Engineers over-trusted high-confidence predictions

A 91%-confidence prediction triggered an engineer to scale a node pool immediately. The predicted breach never came — the trend self-corrected. The premature scaling caused its own brief performance blip. Confidently wrong is still wrong.

"It said 91%. I acted without checking."

Failure 3 — One confidence score across all metric types

85% confidence on stable memory utilization is meaningful. 85% confidence on network latency — which naturally swings 40% on a normal day — means almost nothing. We had one calibration for everything. That was a design error.

"85% on memory ≠ 85% on latency."

✓ The fix: Three-tier traffic light — Watch / Act / Urgent — with per-metric volatility calibration. Same predictions, better communication design. Engineer trust went from 42% acting on predictions to 87%.

"Ask the System" — answers from your own incident data, not from a model's imagination

What RAG means — simply:

Ask: Engineer types: "What's most likely to break in the next seven days?"

Retrieve: Vector database finds 8–10 most relevant historical incidents + current live forecast data for all teams

Augment: That context is injected into the model prompt — the LLM reads your actual data, not general knowledge

Generate: Model answers from what was retrieved: "E-Commerce: 3 signals trending toward breach within 7 days. Highest risk: avg response time — recommend node scaling."

Why we ran the model locally (Ollama):

- Live telemetry can't leave our network perimeter
- Sub-second inference — real-time chat feels responsive

Real questions engineers asked it:

"What's most likely to break in the next 7 days?"

"Is this memory drift related to the v2.14 deploy?"

"Show me similar past incidents to this CPU spike"

"Which team has the highest P1 risk right now?"

The LLM isn't bringing outside wisdom. It's organizing and surfacing your own operational data. Quality of answers = quality of your incident documentation.

Wins, surprises, and one near-failure — honest production story

✓ What worked

First real catch — 4 PM, not 10 PM

System predicted a response time breach in E-Commerce 6+ hours ahead. Engineer scaled at 4 PM. No P1. No customer impact. The incident that never happened is the one that mattered.

Ownership debate: eliminated

Memory drift auto-linked to a deployment 3 hours earlier. The 20–40 minute 'which team owns this?' conversation didn't happen. The card arrived with the answer already in it.

Junior engineer, first on-call shift

Used the copilot to navigate a live situation without calling anyone. Said it was 'like having a senior engineer explain things without the Slack thread.' That's institutional memory working.

What surprised us

Week 1: almost abandoned

Watch threshold was too sensitive. Too many low-urgency predictions. Engineers were close to mentally opting out. Three days of calibration saved the whole project. Start narrow.

Copilot gave a wrong answer

RAG retrieved a marginally relevant incident. Confident-sounding answer, wrong for the situation. We added source citations that week. Explainability is not optional. Build it first.

Not all metrics are forecastable

Payment error spikes and auth failures are event-driven, not trend-driven. Trend models on bursty metrics produce nonsense. Classify metrics before you build forecasts for them.

What we'd do differently if we started over today

ACT 1

Sequence fixes by trust, not technical ease

We built the checklist first because it was self-contained. Right call. The tracker required every team to change behavior — we needed the credibility from the checklist win to push that adoption. Think about what each fix earns you politically, not just technically.

ACT 1

Accept that 'green metrics' is not the same as 'safe'

Build forward-looking visibility from the beginning, not as an afterthought. The deployment tracker felt like a luxury feature until we traced two incidents to exactly the gap it would have closed. It wasn't a luxury. It was the most important fix.

ACT 2

Classify every metric before building forecasts for any of them

A single upfront exercise — steady-state or bursty? — would have saved two weeks of building the wrong thing and another week of cleaning up the noise it created. Do this first. It takes one day.

ACT 2

Explainability is not a feature. It is a precondition.

Source citations, evidence panels, the 'why does it think this?' button — these should be built before the first user ever sees an AI answer. Not after the first wrong answer. We learned this the hard way. Don't.

KEY TAKEAWAYS

Six things — three from each act — worth carrying home

01

Fix the foundation before you add AI

The checklist gate, unified dashboard, and tracker made the data trustworthy and the process disciplined. Without that foundation, the AI layer would have had nothing reliable to build on.

02

Stop asking 'what broke.' Start asking 'where did this become invisible.'

Almost every incident has a moment where it became unrecoverable — usually a visibility gap, not a technical failure. Find that moment. Fix that gap.

03

Green metrics can still page you

Coordination failures don't show up on dashboards. They show up as incidents. Forward-looking visibility isn't optional — it's the thing that closes the gap your dashboards structurally can't.

04

Trend matters more than threshold

Rate of change is more informative than current value. The data to predict your next P1 is already in your telemetry. Compute the direction, not just the position.

05

Confidence scores are a UX problem, not a model problem

How you display uncertainty shapes human behavior more than the accuracy of the uncertainty estimate itself. 42% to 87% trust — from a design change, not a model change.

06

Explainability first, AI answer second

If you can't show where an AI answer came from, you haven't given the engineer enough to decide whether to act on it. Build source citations before you ship the feature, not after the first mistake.

Questions — and my honest answers

I'll do my best to be honest rather than confident.

Q: Do you need Dynatrace specifically for any of this?

A: No. You need a telemetry source. Prometheus, Datadog, CloudWatch, anything that gives you time-series metrics works. The forecasting layer doesn't care which tool collected the data.

Q: How long does Act 1 take to build — realistically?

A: The checklist gate: one to two weeks if your pipeline is reasonably accessible. The unified dashboard: two to four weeks — the correlation logic is the hard part. The tracker: four to six weeks, with most of that time spent on organizational adoption, not code.

Q: We don't have good incident documentation. Can we still build the RAG copilot?

A: Start with the forecasting layer — it only needs live telemetry, not history. Build the copilot in parallel by documenting every incident from now on. Even twenty well-documented incidents is a start. Don't wait for a perfect archive.

Q: Did you consider LSTM or Prophet for the forecasting instead of linear models?

A: Yes — we tested Prophet. Better on 7–30 day windows, slower on short-range latency. For 6–48 hour operational windows, linear projection plus exponential smoothing was accurate enough, significantly faster to tune, and much easier to debug when it was wrong. YAGNI.

Thank You

From Firefighting to Foresight: Re-engineering Our P1 Incident Response Pipeline

Software Engineer III (Senior Software Engineer) — Costco Wholesale Corporation, USA

Independent Researcher | Seattle, WA, USA — Cloud Architect | AIOps Practitioner | AI/ML & Cybersecurity Researcher

IEEE Senior Member | Member, Institution of Engineering and Technology (MIET) | ACM Professional Member | Sigma Xi Member | Raptors Fellow

🔗 linkedin.com/in/rajesh-balaji-74787932

🎓 ORCID: 0009-0008-9481-5687

📧 vasubalajimca@gmail.com

"Stop asking 'what broke.'

Start asking 'where did this become invisible.'"

— Rajesh Balaji, Raptors 2026