From "What Broke" to "What Changed"

Michael Beemer, Dynatrace
Parth Suthar, DevCycle

hosted at: https://beeme1mr.github.io/kubecon-us-25-flag-observablity/

KubeCon | CloudNativeCon
North America 2025

Michael Beemer

Senior Product Manager

beeme1mr

Parth Suthar

Software Engineer

suthar26

3:25 PM Monday

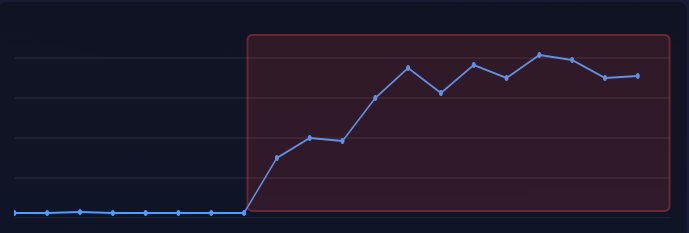
A familiar story...

- **ERROR** 3:25 PM
  Failure rate increase detected
- **INVESTIGATION** 3:30 PM
  No recent deployments found
- **INVESTIGATION** 4:05 PM
  No code changes in last 48h
- **INVESTIGATION** 4:50 PM
  All dependencies healthy
- **ESCALATION** 5:20 PM
  All hands on deck
- **ROOT CAUSE** 6:25 PM
  "Oh, I toggled that feature flag"

ERROR **Failure rate increase**

Nov 11, 2025, 3:25 PM

The error rate increased to 7.14%. All dynamic requests have increased failure rates.



AFFECTED SERVICE

checkout-service

⚠ **The Missing Context**

Feature flag changes are **invisible** to traditional monitoring

Feature flags are hidden from observability tools
making it difficult to pinpoint changes
as the root cause of incidents

The Observability Gap

What traditional monitoring misses

What We See

- Error rates spike
- Latency increases
- Failed requests
- Resource exhaustion

Traditional metrics & traces

What's Hidden

- Which flag changed?
- When was it toggled?
- Who was affected?
- What variant was served?

Feature flag context

A Real-World Example: OpenTelemetry Demo

Feature flags that trigger production-like issues

Astronomy Shop

- Full **microservices e-commerce** platform
- 10+ services, multiple languages
- Production-grade observability stack
- Uses **OpenFeature** for feature flagging

🎯 **Perfect Test Bed**

Includes flags that deliberately enable problems to demonstrate observability challenges

Problem Scenarios

⚠ `recommendationServiceCacheFailure`

Memory leak → 1.4x exponential growth → OOM crashes

⚠ `paymentServiceUnreachable`

Invalid endpoint → all payment requests fail

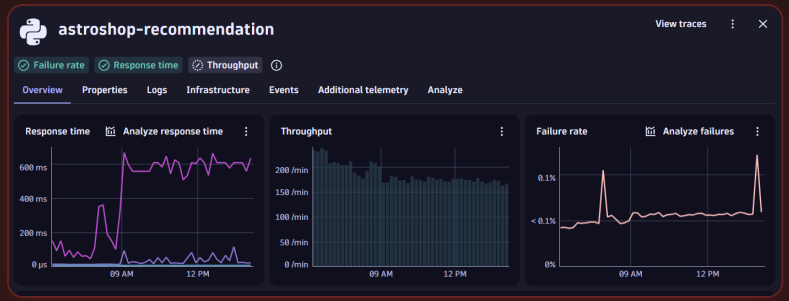
⚠ `imageSlowLoad`

Fault injection → 5s image load latency

Impact vs Root Cause

Observability shows the symptoms, but hides the diagnosis

What Your Dashboard Shows



🚨 Clear Impact: Response time spiked from 145ms to 600ms!
❓ Unknown Cause: No code changes, no deploys...

What's Actually Happening

Production

Targeting Rule `All Users` Modified

- Name: All Users
- Filters:
  - Filter 1:
    - Type: All Users
- Distribution:
  - Variation: Variation Off
  - Percentage: 100%
- Bucketing Key: user_id

→

- Name: All Users
- Filters:
  - Filter 1:
    - Type: All Users
- Distribution:
  - Variation: Variation On
  - Percentage: 100%
- Bucketing Key: user_id

The Hidden Truth

✅ Quick fix: Toggle flag off → problem gone
⏱️ Gives time to debug properly
🎯 But monitoring tools can't see this connection

The Mitigation Problem:

Hours spent restarting pods, rolling back code, and debugging — when a **30-second flag toggle** would have stopped the bleeding

Why This Matters

The hidden cost of invisible feature flags

Slower Recovery

MTTR increases when the easiest fix is invisible

Wrong Direction

Debugging code that isn't actually broken

Escalation

Simple toggle becomes all-hands incident

We need feature flags as **first-class concept** in observability

Feature Flag Observability

A progressive approach for teams looking to gain visibility into flags

Level 0
Flying Blind

Level 1
Broadcast Blast

Level 2
Manual Events

Level 3
Auto Mapping

Level 4
Trace-Level

Level 0: Flying Blind 🙈

You see something is wrong, but have no idea why

The Problem

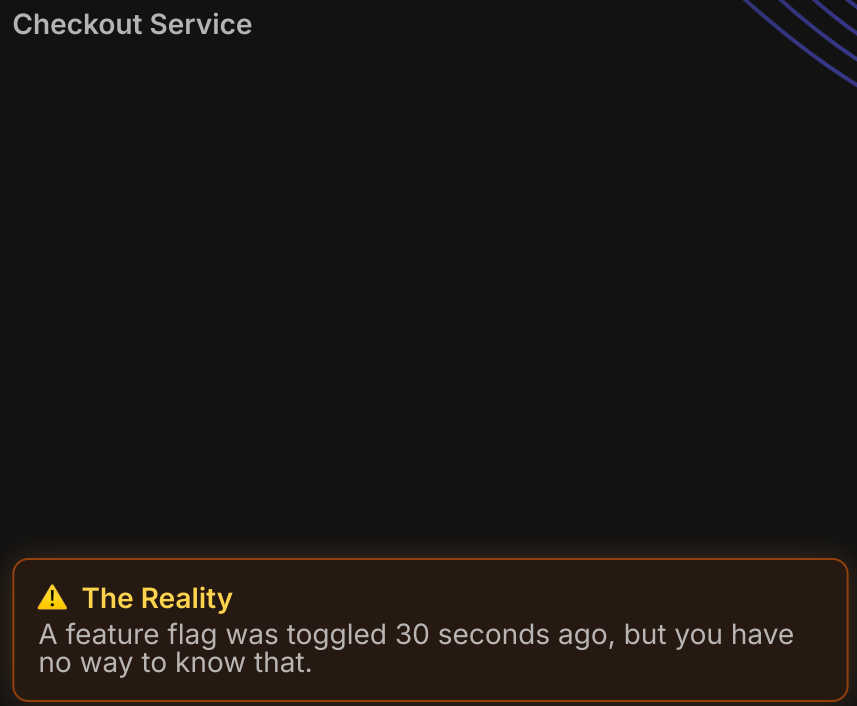
- No visibility into flag changes
- Pure guesswork during incidents
- Hours of manual hunting

What You See

Failure rate spiking!

But what changed? No deployments, no code changes...

Time to check logs, metrics, Slack, coffee machine...



The Reality

A feature flag was toggled 30 seconds ago, but you have no way to know that.

Level 1: Broadcast Blast 📣

Same flag change annotated on all services... but only one is actually using the flag

Checkout service

Payment service

Recommendation service

⚠ The Red Herring Problem

When you manually configure events, you might send them to services that **don't use the flag**. This creates noise during incidents and can send investigations in the wrong direction.

"We see the flag changed at the same time, but these services look fine... maybe it's not the flag?"

Level 2: Manual Change Events ✍️

Send events to specific services... but requires manual mapping and can become outdated

checkout-service **payment-service** **recommendation-service**

⚠ The Manual Mapping Problem

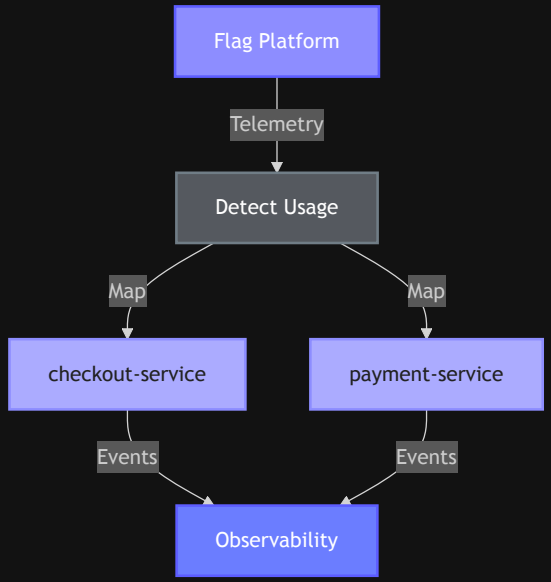
You can **manually configure** which services receive flag change events, but this requires maintaining a mapping. As your system evolves, these mappings become **outdated** — events might go to services that no longer use the flag, or miss new services that started using it.

"We configured this 6 months ago... did we update it when we refactored?"

Level 3: Automatic Event Mapping 🤖

Telemetry-driven routing eliminates manual configuration

Intelligent Routing



Benefits

- ✓ **Automatic discovery**
  Detects where flags are evaluated
- ✓ **Smart routing**
  Events only to affected services
- ✓ **Zero maintenance**
  Stays current as code evolves
- ⚠ **Correlation ≠ Causation**
  Good indicator, but can't measure exact impact

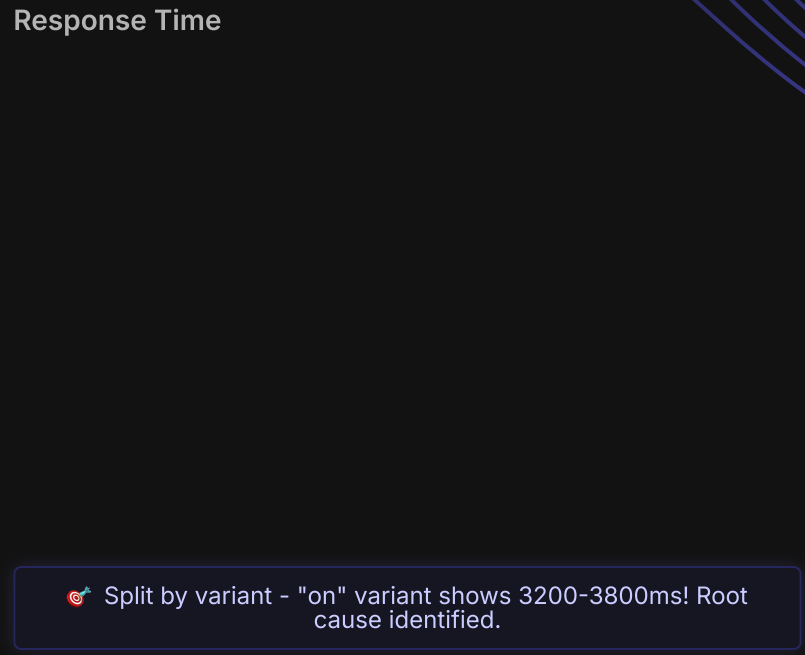
Solves the scaling problem, but requires a more complicated setup. How can it be improved further?

Level 4: Trace-Level Observability 🎯

Feature flags as a first-class observability concept

Fine-Grained Insights

- **See flag evaluations**
  Every trace shows which flags were evaluated
- **Realtime mapping of flag presence per service**
  Filter by traces with specific flag presence
- **Filter by flag presence**
  Isolate requests that used a specific flag
- **Compare variants**
  See performance differences in real-time
- **Instant root cause**
  Immediately identify which variant caused issues



Trace-level observability makes feature flags a first-class citizens in your monitoring stack

Feature Flag Observability Standards

Bridging feature flags and observability through open standards

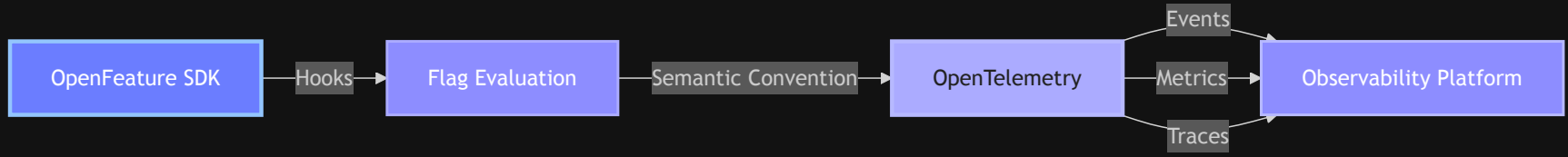
OpenFeature 🤝 OpenTelemetry

OpenFeature

- Open standard for **feature flagging**
- Vendor-neutral SDK
- **Hook system** for extensibility

OpenTelemetry

- Open standard for **observability**
- Metrics, traces, and logs
- **Semantic conventions** for consistency



Feature Flag Semantic Convention

Standardized attributes that make flag evaluations observable

Obvious Attributes

`feature_flag.key`
The flag identifier (e.g., "new-checkout")

`feature_flag.result.variant`
Which variant was served (e.g., "on", "treatment")

`feature_flag.result.value`
The value returned by the flag (e.g., true, false, "blue")

`feature_flag.provider.name`
Flag provider (e.g., "flagd", "launchdarkly", "devcycle")

`feature_flag.result.reason`
Why this variant? (e.g., "targeting_match", "default")

Nuanced Attributes

`feature_flag.set.id`

Human-readable logical identifier for where this flag is managed
`"acme-org/web-app/production"`

`feature_flag.context.id`

Provider's context identifier (fallback: targeting key)
Used by providers like DevCycle to lookup user evaluations and simulation

`feature_flag.version`

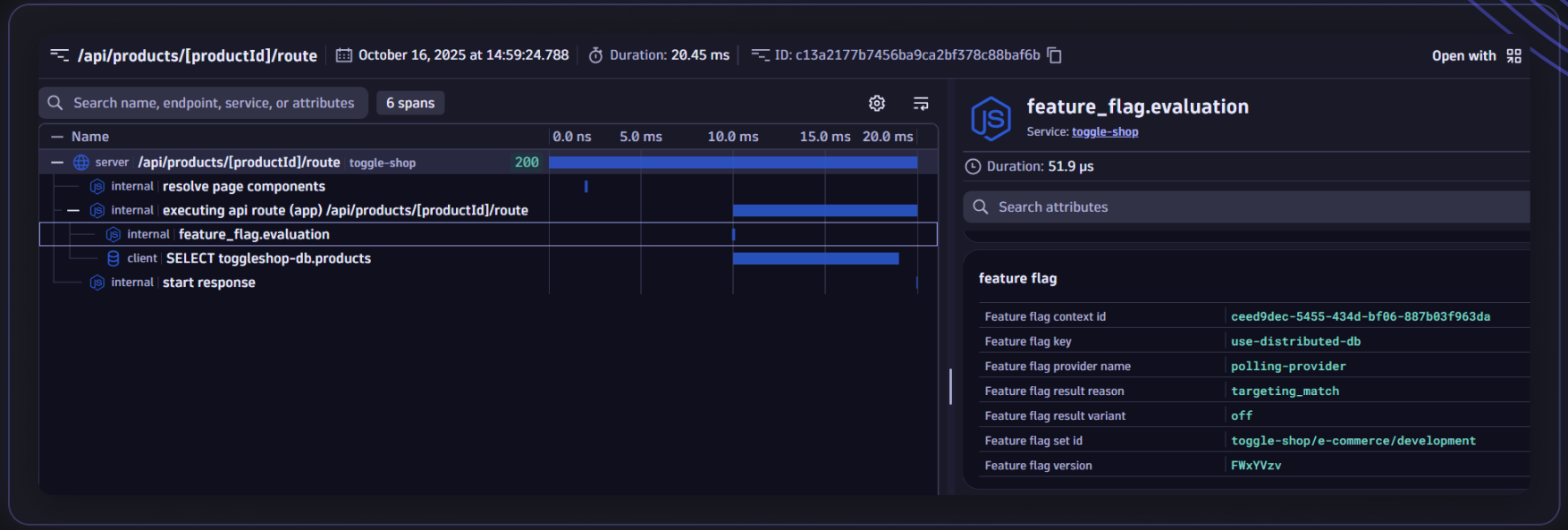
Ruleset version at evaluation time
Track which configuration was active (number, hash, etc.)

How It Works: OpenFeature Hooks

```
import { MyProvider } from 'my-flag-provider';
import { EventHook } from '@openfeature/open-telemetry-hooks';

// Initialize OpenFeature with your flag provider
await OpenFeature.setProviderAndWait(new MyProvider());

// Register the OpenTelemetry hook
OpenFeature.addHooks(new EventHook());
const client = OpenFeature.getClient();

// Prepare evaluation context
const context = {
  targetingKey: 'user_123',
  tier: 'premium',
  environment: 'production'
};

// Evaluate flag - automatically traced!
const variant = await client.getStringValue(
  'new-checkout-flow',
  'control',
  context
);
```

Seeing It in Action

Feature flag evaluation captured in distributed tracing



Why Standards Matter

Interoperability
Works across vendors and tools

Consistency
Same attributes everywhere

Adoption
Easy to implement and adopt

Progressive Delivery with Observability

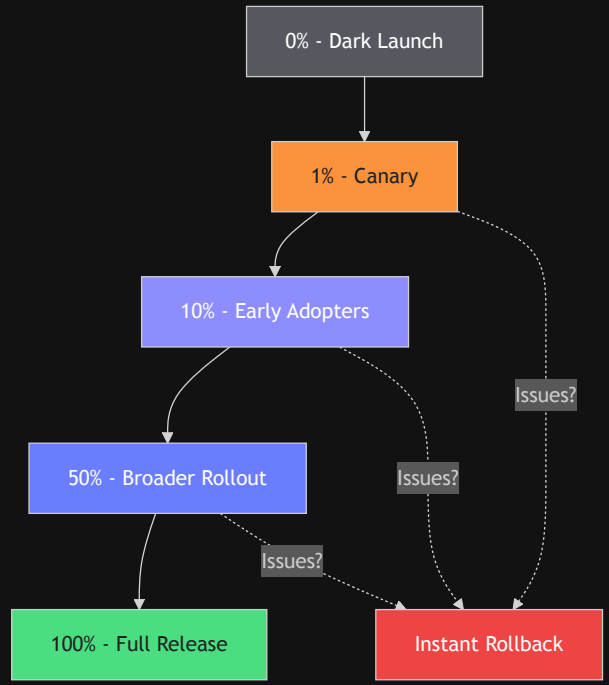
Transforming theory into practice

What Is Progressive Delivery?

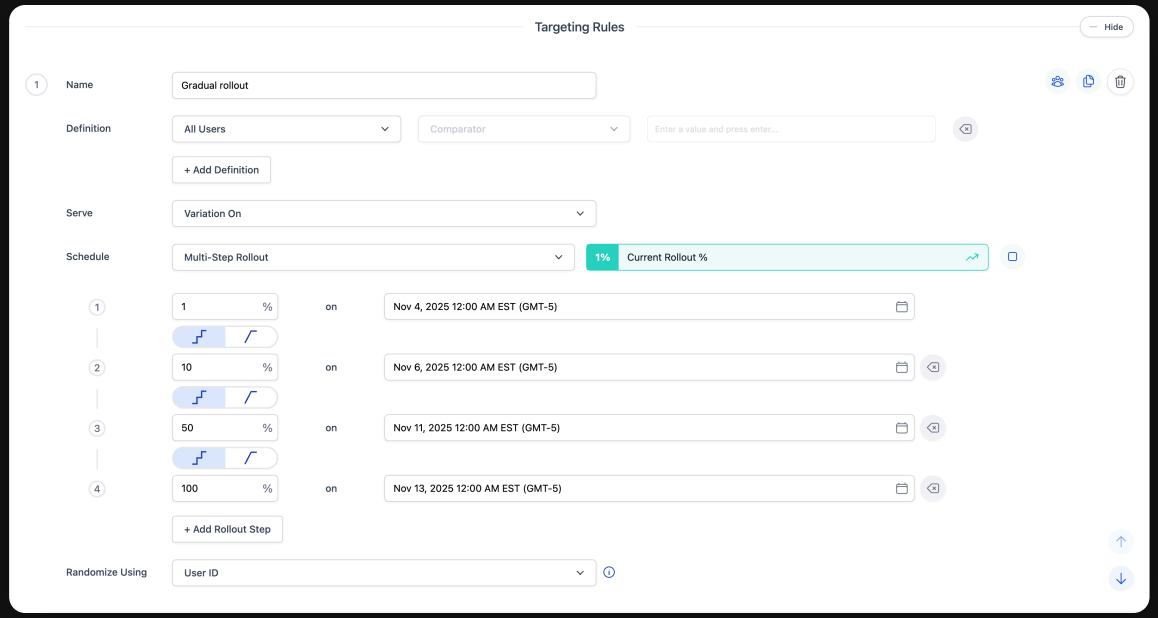
Concepts

- Gradually expose customers to new features
- Monitor impact at each step
- Roll back instantly if problems arise
- Minimize risk, maximize confidence

Rollout Stages



Rollout targeting setup

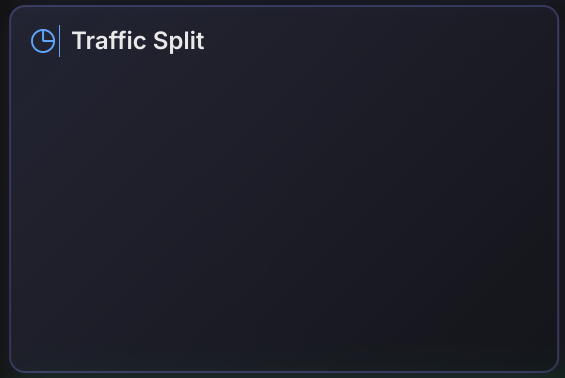


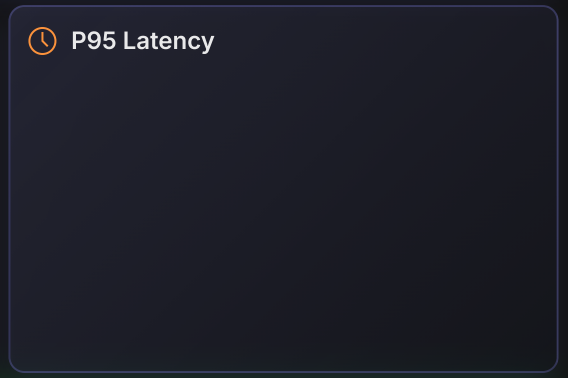
Observability Enables Progressive Delivery

Variant-level visibility reveals the root cause

`new-checkout-flow` @ `checkout-service` — Rolled Back

0% rollout (100% control / 0% treatment)





✅ Rollback Triggered - Treatment Variant Disabled

❌ **Without Flag Observability:**
Hours debugging aggregate metrics, unclear which change caused the issue

✅ **With Flag Observability:**
Instant variant identification & rollback in seconds

Key Benefits

Without Observability

- Blind rollouts
- Slow to detect issues
- Manual rollback decisions
- Can't compare variants
- High-risk deployments

With Observability

- Data-driven rollouts
- Instant issue detection
- Automated rollback triggers
- Real-time variant comparison
- Low-risk deployments

Observability is the **key** to progressive delivery

Challenges and Opportunities

The journey isn't complete

⚠ Current Limitations

Scaling Challenges

Users must handle deduplication, sampling, and aggregation strategies themselves

Provider Dependency

Requires OpenFeature providers to include semantic convention attributes

Opportunities Ahead

Metrics Convention

Standardize flag evaluation metrics for consistent monitoring

Change Events

Define standard format for flag configuration changes

Flags as Entities

Model feature flags as first-class entities in observability platforms

Let’s Build the Future Together

Feature flags are powerful tools for modern software delivery

But they need to be visible in our observability stack

OpenFeature + OpenTelemetry make this possible with open standards

Join us in transforming how we understand and deploy software

From “What Broke” to “What Changed”

Thank You!

Questions?

Find us after the talk or online:

Resources

- openfeature.dev
- github.com/open-openfeature/
- opentelemetry.io

Connect

- CNCF Slack: #openfeature
- CNCF Slack: #opentelemetry
- @openfeature / @opentelemetry