Modernizing Development Operations

Building secure, observable, and efficient CI/CD pipelines through intelligent automation and developer-centric platforms

The Challenge We're Solving

Today's Reality

- Every team builds pipelines differently
- Security checks happen too late or not at all
- Secrets are often hardcoded or poorly managed
- When things break, we can't see why
- Debugging takes hours instead of minutes

Our Vision

- One standardized approach across all teams
- Security automated from day one
- Secrets protected and rotated automatically
- Complete visibility into system behavior
- Issues detected and resolved quickly

Standardizing CI/CD Pipelines with Backstage

Think of Backstage as your team's control center for development. Instead of each developer creating their own deployment scripts and processes, Backstage provides a unified Internal Developer Portal (IDP) where everyone works from the same playbook. When a new service is created, it automatically comes with a standardized pipeline that includes testing, security scanning, and deployment steps—no manual setup required.

What is Backstage?

Developer Portal

A centralized platform where developers find everything they need: service catalogs, documentation, templates, and tools—all in one place.

Service Catalog

Every microservice, API, and library is registered and searchable. Know who owns what, see dependencies, and understand the full architecture at a glance.

Software Templates

Pre-built project scaffolds that include CI/CD pipelines, security configurations, and best practices. Start new projects in minutes, not days.

A Real Example: Before Backstage

1

Day 1: Setup Begins

Sarah joins the team and needs to deploy her first service. She asks around for the "standard" pipeline configuration.

2

Day 2: Copy-Paste Engineering

She finds three different pipeline examples from three teams. Each one is slightly different. She picks one and modifies it.

3

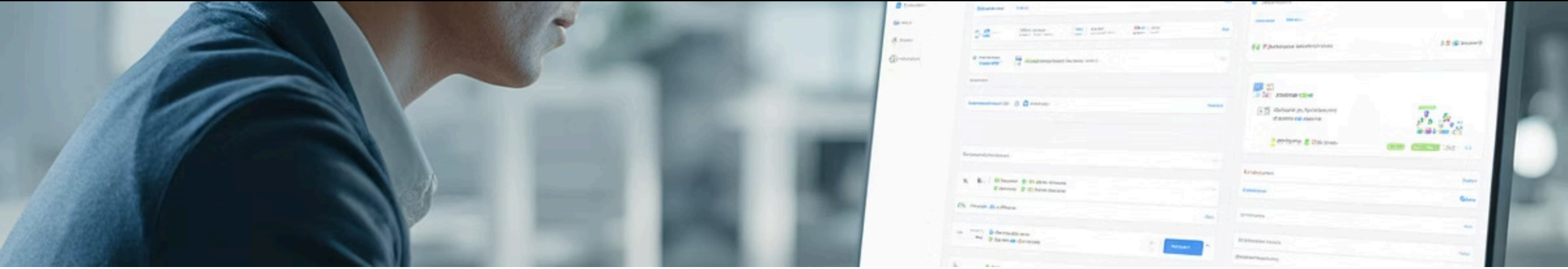
Day 3: The Gotchas

Her pipeline fails because it's missing security scanning. She adds it manually, but uses an outdated scanner version.

4

Day 5: Finally Deployed

After multiple iterations and help from senior engineers, her service is deployed—but it doesn't match team standards.



The Same Example: With Backstage

Minute 1: Choose Template

Sarah opens Backstage, clicks "Create New Service," and selects the Node.js microservice template.

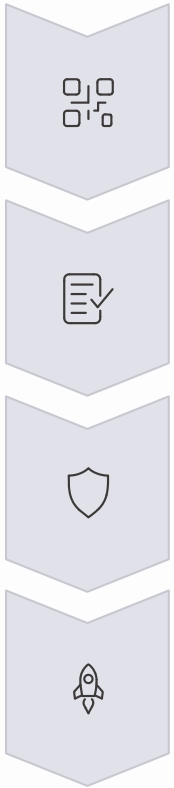
Minute 5: Repository Ready

The template generates a complete repository with CI/CD pipeline, security scanning, tests, and documentation—all pre-configured.

Minute 15: First Deployment

She commits her code, and the standardized pipeline automatically runs tests, security scans, and deploys to staging. Everything just works.

The Automated Pipeline Flow



Code Commit

Developer pushes code to repository. Pipeline automatically triggers within seconds.

Automated Tests

Unit tests, integration tests, and end-to-end tests run automatically. Any failure stops the pipeline immediately.

Security Scanning

SBOM generation, SAST, DAST, and CVE checks run in parallel. Vulnerabilities block deployment.

Deployment or Rollback

If all checks pass, deploy to production. If issues are detected post-deployment, automatic rollback initiates.

Understanding Security Scanning Components

SBOM - Software Bill of Materials

A complete inventory of every component, library, and dependency in your application. Think of it like a detailed ingredient list on food packaging—you know exactly what's inside.

Example: Your app uses Express.js 4.18.2, which depends on 25 other packages. SBOM lists them all.

SAST - Static Analysis

Scans your source code without running it, looking for security vulnerabilities like SQL injection risks, hardcoded secrets, or insecure cryptography.

Example: Detects when you accidentally write `password = "admin123"` in your code.

DAST - Dynamic Analysis

Tests your running application like an attacker would, trying to exploit vulnerabilities through actual HTTP requests and interactions.

Example: Attempts SQL injection on your login form to see if it's properly protected.

CVE - Known Vulnerabilities

Checks if any of your dependencies have publicly disclosed security vulnerabilities in the CVE database.

Example: Alerts you that the version of Log4j you're using has a critical remote code execution vulnerability.

Real-World Scenario: Automated Protection

> On a Friday afternoon, a developer commits code with a new npm package that contains a high-severity vulnerability. Before they even grab coffee, the pipeline has already blocked the deployment, created a ticket, and sent a Slack notification explaining the issue and suggesting safer alternatives. Production remains secure, and the developer fixes the issue Monday morning—no emergency weekend calls.

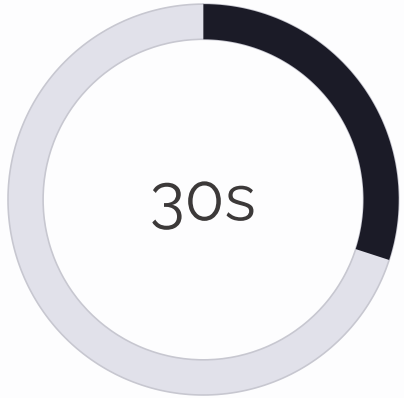
Automatic Rollback: Your Safety Net

How It Works

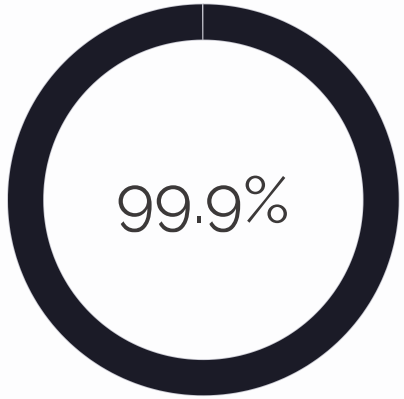
After deployment, the system continuously monitors key health metrics: error rates, response times, and resource usage. If any metric exceeds predefined thresholds, an automatic rollback is triggered within seconds.

Rollback Triggers

- Error rate spikes above 5%
- Response time increases by 200%
- Failed health checks for 30 seconds
- Critical logs detected in first 5 minutes



Average Rollback Time



Uptime Maintained

Introducing Observability Concepts

Observability is about understanding what's happening inside your systems by examining their outputs. Unlike traditional monitoring that answers "Is it working?", observability answers "Why isn't it working?" and "What changed?" It gives you the ability to ask any question about your system's behavior, even questions you didn't anticipate when you built it.

The Three Pillars of Observability

📊 Metrics

What: Numerical measurements over time

Example: CPU usage at 75%, response time of 245ms, 1,247 requests per second

Use Case: Tracking trends, setting alerts, capacity planning

📝 Logs

What: Timestamped records of discrete events

Example: "User 12345 logged in at 14:23:01", "Payment failed: insufficient funds"

Use Case: Debugging specific issues, audit trails, understanding event sequences

🔍 Traces

What: The journey of a single request through your system

Example: API call → Authentication (45ms) → Database query (120ms) → Cache check (12ms) → Response (5ms)

Use Case: Finding bottlenecks, understanding dependencies, diagnosing latency

Monitoring vs. Observability: The Key Difference

Traditional Monitoring

You define specific checks in advance: "Alert me if CPU exceeds 80%" or "Notify if response time is above 1 second." It answers predetermined questions.

Limitations

- Only catches problems you anticipated
- Struggles with complex, distributed systems
- Hard to debug novel issues
- Reactive rather than exploratory

Observability

You instrument your system to emit rich data, then ask questions as problems arise: "Why are checkouts slow for users in California?" It enables exploration and discovery.

Advantages

- Investigate unexpected behaviors
- Understand complex system interactions
- Correlate metrics, logs, and traces
- Proactive problem-solving

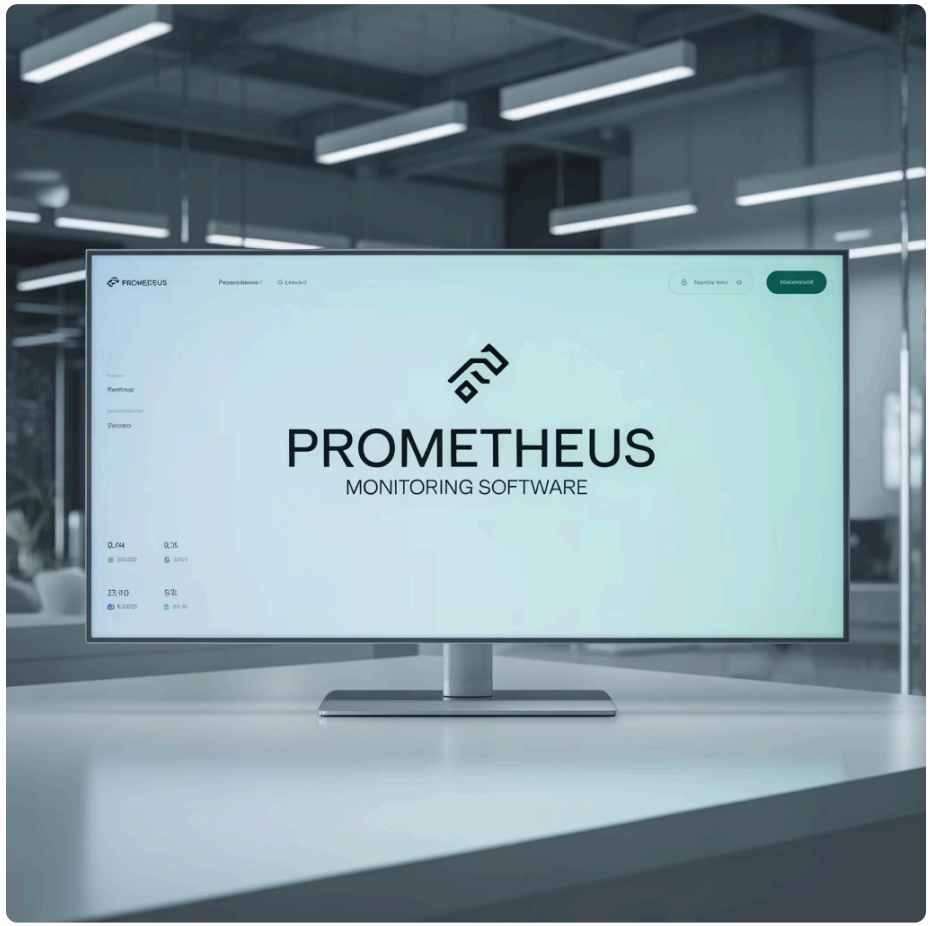
A Real-World Observability Story

> **The Problem:** On Tuesday morning, checkout completion rates dropped from 95% to 78%. Traditional monitoring showed all systems green—servers healthy, databases responsive, no errors in logs.
>
> **The Investigation:** Using observability tools, engineers discovered that a recent deployment increased the checkout page load time from 1.2s to 3.1s—but only for users on mobile devices with slow connections. The issue? A new JavaScript library added 800KB of uncompressed code.
>
> **The Resolution:** Within 20 minutes, they identified the problematic commit, rolled back the deployment, and restored checkout rates. Without observability, this would have taken hours of guesswork and blind debugging.

The Observability Stack

Building comprehensive observability requires assembling the right tools for collecting, storing, and visualizing your system's telemetry data. Our stack combines industry-leading open-source tools that work together seamlessly: Prometheus for metrics, Grafana for visualization, EFK or ELK for logs, and OpenTelemetry for unified instrumentation.

Prometheus: Metrics Collection and Storage



What It Does

Prometheus is a time-series database that collects and stores metrics from your applications and infrastructure. It "scrapes" metrics endpoints at regular intervals (typically every 15 seconds), storing data efficiently for quick querying.

Key Features

- **Pull-based model:** Prometheus pulls metrics from your services
- **Powerful query language (PromQL):** Slice and dice your metrics data
- **Built-in alerting:** Define rules that trigger notifications
- **Service discovery:** Automatically finds new services to monitor

Grafana: Visualization and Dashboards

Beautiful Dashboards

Transform raw metrics into intuitive visualizations. Create dashboards that update in real-time, showing exactly what matters to your team.

Cross-Team Collaboration

Share dashboards across teams with role-based access. Everyone from engineers to executives can see relevant metrics in their preferred format.

Alerting Integration

Configure alerts that integrate with Slack, PagerDuty, email, and more. Get notified where you work, with context and suggested actions.

Logging Stack: EFK vs ELK

EFK Stack

Elasticsearch + Fluentd + Kibana

- **Elasticsearch:** Stores and indexes logs for fast searching
- **Fluentd:** Collects logs from all sources and forwards them
- **Kibana:** Web interface for searching and visualizing logs

Best For

Cloud-native environments, Kubernetes deployments, high log volume

ELK Stack

Elasticsearch + Logstash + Kibana

- **Elasticsearch:** Stores and indexes logs for fast searching
- **Logstash:** Processes and enriches logs before storage
- **Kibana:** Web interface for searching and visualizing logs

Best For

Complex log parsing requirements, data transformation needs, traditional infrastructure

Both stacks provide powerful log aggregation and analysis. Fluentd is lighter and more cloud-native, while Logstash offers more sophisticated data processing capabilities.

OpenTelemetry: Unified Instrumentation

OpenTelemetry (OTel) is an open-source framework that provides a single, vendor-neutral way to instrument your applications for metrics, logs, and traces. Instead of using different libraries for different observability tools, OTel gives you one API and SDK that works with everything.

Single Instrumentation

Write instrumentation code once using OTel libraries. Switch observability backends (Prometheus, Jaeger, Datadog, etc.) without changing application code.

Automatic Instrumentation

OTel provides auto-instrumentation for popular frameworks like Express, Django, Spring Boot. Add one library, get comprehensive telemetry automatically.

Context Propagation

Automatically traces requests across microservices, maintaining context as calls flow through your distributed system.

Putting It All Together: Complete Stack Architecture

Applications Instrumented with OpenTelemetry

Your microservices emit metrics, logs, and traces using OTel SDKs

Collection Layer

Prometheus scrapes metrics, Fluentd/Logstash collects logs, OTel Collector aggregates traces

Storage Layer

Prometheus stores metrics, Elasticsearch stores logs, Jaeger stores traces

Visualization & Analysis

Grafana dashboards combine metrics and logs, Kibana searches logs, Jaeger UI explores traces

Integration with Backstage IDP

The true power emerges when we integrate observability directly into Backstage. For every service in your catalog, developers can see live metrics, recent logs, and active traces—all without leaving the portal.

Service Health at a Glance

Each service page in Backstage shows embedded Grafana panels with key metrics like error rates, latency, and throughput. Red, yellow, green indicators provide instant status visibility.

Deep Linking

Click from a service in Backstage directly into detailed Grafana dashboards, Kibana log searches pre-filtered to that service, or Jaeger traces showing request flows.

Contextual Alerts

Active alerts for a service appear on its Backstage page. Developers see what's broken and why, with links to runbooks and suggested fixes.

Team Ownership

Observability dashboards automatically show data for services your team owns. New team members get access to the right monitoring without manual setup.

Before and After: Developer Experience

Before: The Old Way

"The checkout service is slow. Let me investigate..."

1. Check which team owns checkout (ask in Slack)
2. Find the Grafana dashboard URL (search Wiki)
3. Guess which metrics matter (trial and error)
4. Search logs in Kibana (build query from scratch)
5. Look for traces (if you know Jaeger exists)
6. Correlate information across 4 different tools
7. **Time to insight: 30-60 minutes**

After: With Integrated Observability

"The checkout service is slow. Let me investigate..."

1. Open checkout service page in Backstage
2. See elevated p95 latency on embedded dashboard
3. Click "View Logs" → pre-filtered to slow requests
4. Identify database query taking 2+ seconds
5. Click trace ID → see full request flow in Jaeger
6. Realize missing index on recent table change
7. Time to insight: 5 minutes

Our Path Forward

Phase 1: Foundation (Month 1-2)

1

Deploy Backstage IDP with service catalog. Standardize CI/CD pipeline templates with basic security scanning. Implement centralized secrets management.

Phase 2: Security Hardening (Month 2-3)

2

Integrate SBOM generation, SAST/DAST, and CVE scanning into all pipelines. Enable automatic rollback mechanisms. Enforce secrets rotation policies.

Phase 3: Observability Basics (Month 3-4)

3

Deploy Prometheus, Grafana, and EFK/ELK stack. Instrument core services with OpenTelemetry. Create initial dashboard templates.

Phase 4: Full Integration (Month 4-6)

4

Embed observability data into Backstage service pages. Complete instrumentation of all microservices. Train teams on observability practices and troubleshooting workflows.

By following this roadmap, we'll transform how we build, deploy, and operate software—making it more secure, observable, and maintainable. Every developer will have the tools and visibility they need to ship confidently.