Devops Shack

OBSERVABILITY VS. MONITORING in DevOps

By DevOps Shack

Click here for DevSecOps & Cloud DevOps Course

DevOps Shack

Observability vs. Monitoring in DevOps

Table of content

1. Introduction to Observability and Monitoring

In the fast-paced world of DevOps, achieving reliability, performance, and rapid issue resolution is critical. Two concepts central to these goals are **Monitoring** and **Observability**. While often used interchangeably, they are fundamentally different and serve complementary roles. This section lays the foundation by defining each and explaining why the distinction is vital.

1.1 What is Monitoring?

Monitoring refers to the **collection, analysis, and alerting** of predefined metrics to detect system performance issues or failures. It is typically:

- **Rule-based**: You define what to monitor—like CPU usage, memory, or error rates.
- **Alert-driven**: When a metric crosses a defined threshold, alerts are triggered.
- **Limited in scope**: It answers "what is wrong" but often not "why it's wrong."

Example: Monitoring CPU usage on a server and triggering an alert if it exceeds 90% for over 5 minutes.

1.2 What is Observability?

Observability, on the other hand, is a **broader, systemic capability** that enables teams to understand the internal states of a system based on its external outputs (like logs, metrics, and traces). It allows teams to:

- **Ask new questions** about system behavior **without predefining metrics**.
- **Diagnose root causes** in complex, distributed environments.
- Gain insights into **emergent issues** that weren't previously anticipated.

Key Differentiator: Observability doesn't require you to know what to look for in advance.

1.3 Why This Distinction Matters in DevOps

In modern DevOps environments—especially with microservices, cloud-native architectures, and containerization—the complexity is too vast to monitor every potential failure scenario. Here's why understanding this distinction is crucial:

- **Monitoring is proactive but rigid**; it's useful for known issues.
- **Observability is dynamic and flexible**, supporting debugging of unknown failures.
- Teams relying solely on monitoring often struggle with troubleshooting in production.
- Observability improves **Mean Time to Resolution (MTTR)** and supports **continuous delivery** by increasing system transparency.

Takeaway: You need both—monitoring for visibility and alerting, and observability for exploration and diagnosis.

2. Core Concepts and Definitions

To build a strong understanding of the differences and interplay between observability and monitoring, it's important to define the key elements that make up these practices. This section covers the foundational components that guide how systems are observed and monitored in modern DevOps environments.

2.1 Metrics, Logs, and Traces Explained

These are the **three primary types of telemetry data** used to gain insight into system behavior:

- **Metrics**: Numerical values measured over time (e.g., CPU usage, request latency, memory consumption).
  - Fast to query and great for triggering alerts.
  - Aggregated and lightweight, making them ideal for dashboards.
- **Logs**: Timestamped records of discrete events in the system (e.g., errors, warnings, user actions).
  - Useful for debugging specific events.
  - Can be structured (JSON) or unstructured (plain text).
- **Traces**: A record of a request as it flows through multiple services and components (distributed tracing).
  - Helps visualize request paths and bottlenecks.
  - Crucial in microservices and distributed environments.

These three together are often referred to as the **"Three Pillars of Observability."**

2.2 The Three Pillars of Observability

These pillars form the backbone of modern observability systems:

- **Metrics** give a high-level view of system health and performance.

- **Logs** offer detailed context for specific events or failures.
- **Traces** reveal the end-to-end journey of transactions or requests.

Why they matter together:

- Metrics tell you **something is wrong**.
- Logs and traces tell you **why it is wrong**.
- When used in concert, they offer **rich visibility** into system internals.

📌 Pro Tip: An effective observability strategy should integrate and correlate all three data types.

2.3 Comparing Monitoring and Observability

Feature	**Monitoring**	**Observability**
Purpose	Detect known issues	Understand unknown failures
Scope	Specific metrics and thresholds	Full system state insight
Approach	Reactive (alerts when thresholds break)	Proactive and exploratory
Tooling	Prometheus, Nagios, Zabbix	OpenTelemetry, Datadog, Honeycomb
Adaptability	Low: requires pre-defined rules	High: allows ad-hoc queries and insights

Monitoring is a **subset** of observability. You monitor what you already know can go wrong. Observability helps you discover and fix what you didn't know could go wrong.

3. Monitoring: Use Cases and Limitations

Monitoring is an essential component of system management, especially in traditional infrastructure and predictable environments. However, as systems grow in complexity, relying solely on monitoring can lead to blind spots. This section explores how monitoring works in practice, when it's effective, and where it begins to fall short.

3.1 Reactive Incident Detection

Monitoring shines when it comes to **detecting and alerting** on predefined conditions. These include:

- **System health metrics** like CPU, memory, disk usage, and network latency.
- **Application-level metrics** such as request count, error rate, and response time.
- **Service availability checks** using uptime monitors or health probes.

How it works:

- Engineers set thresholds.
- If the threshold is crossed (e.g., CPU > 90%), an alert is triggered.
- Teams respond based on playbooks or incident response plans.

Example: A sudden spike in response time triggers an alert, prompting a quick investigation.

3.2 Predefined Alerts and Dashboards

Monitoring tools typically offer customizable dashboards and alerting systems:

- **Dashboards** provide real-time visualization of critical metrics.
- **Alerts** notify teams via email, SMS, or integrations (e.g., Slack, PagerDuty).
- **Synthetic monitoring** can simulate user interactions to detect downtime or poor performance.

While useful, this method depends heavily on **foreknowledge**:

- You must know what's important to monitor.
- You must guess the right thresholds for alerts.
- Too many alerts can cause **alert fatigue**; too few can lead to **missed issues**.

3.3 Limitations in Dynamic and Distributed Systems

Modern systems—especially those built on **microservices**, **containers**, or **serverless platforms**—bring new challenges:

- **Ephemeral infrastructure**: Containers or pods may exist for seconds, making traditional monitoring difficult.
- **Distributed architectures**: A single transaction may touch dozens of services, each with its own telemetry.
- **Unknown unknowns**: You can't monitor what you don't expect or haven't configured.

Real-world limitation:
A dashboard shows high latency, but doesn't explain which microservice caused the bottleneck or why it happened. Without deeper observability, you're left guessing.

🔍 **Key Insight**: Monitoring is essential for system **awareness**, but not sufficient for system **understanding**.

4. Observability: Deep Insights and Root Cause Analysis

While monitoring answers "**What's wrong?**", observability answers "**Why is it wrong?**". In dynamic and distributed DevOps environments, observability provides the **context-rich insights** needed for rapid troubleshooting, deep root cause analysis, and continuous improvement. This section highlights how observability enables deeper system understanding and facilitates robust, scalable operations.

4.1 Understanding System Behavior Holistically

Observability offers a **comprehensive view** of how different components in your system interact. It allows DevOps teams to:

- **Correlate metrics, logs, and traces** to reconstruct incidents.
- **Identify dependencies** between services during real-time issues.
- **Explore anomalies** without preconfigured thresholds or alerts.

Scenario: A spike in latency occurs. With observability, you trace the issue to a single downstream service, observe the deployment that caused it, and correlate it with error logs—**all within minutes**.

Benefit: Holistic visibility turns guesswork into actionable insight.

4.2 Ad-hoc Queries and Dynamic Exploration

Unlike monitoring, observability tools allow engineers to ask **open-ended questions** about systems:

- What changed before this latency spike?
- Which service is causing cascading failures?
- What's the performance trend for user requests from Europe?

With platforms like **Grafana**, **New Relic**, **Honeycomb**, or **OpenTelemetry**:

- You can build queries **on the fly**.
- You don't need to predefine everything.

- The system supports **exploratory debugging**, crucial for unknown failures.

🧠 **Insight**: Observability empowers teams to **investigate**, not just react.

4.3 Benefits for Microservices and Cloud-Native Environments

Modern applications often consist of dozens (or hundreds) of microservices, spread across containers and orchestrated by platforms like **Kubernetes**. Observability is especially crucial in these environments because:

- **Traditional monitoring breaks down** with frequent service changes.
- **Distributed tracing** maps out request flow across services, making it easier to detect bottlenecks or dropped transactions.
- **Real-time debugging** is possible even when infrastructure is transient.

Example: In a Kubernetes-based system, a user request passes through six services. A trace shows one service causing a 3-second delay. Observability tools let you instantly drill down into logs and metrics for just that service instance.

☑ **Summary**: Observability enhances your ability to:

- Debug complex issues quickly.
- Gain real-time insights in dynamic environments.
- Make confident, data-driven operational decisions.

5. Tools and Technologies Comparison

A successful DevOps strategy relies on the right tools. As organizations evolve from traditional monitoring to modern observability, the tools used for telemetry collection, visualization, alerting, and debugging also evolve. This section breaks down the most commonly used tools in both categories and highlights how they differ or integrate.

5.1 Monitoring Tools: Nagios, Zabbix, Prometheus

These are some of the most established tools for traditional monitoring:

🛠 **Nagios**

- One of the earliest monitoring solutions.
- Strong in host and service monitoring via plugins.
- Good for legacy infrastructure but lacks native support for cloud-native systems.

📊 **Zabbix**

- Offers real-time monitoring of networks, servers, VMs.
- Combines metrics collection and alerting.
- Not well-suited for dynamic, container-based environments.

🔍 **Prometheus**

- Designed for dynamic infrastructure.
- Pull-based time series database, perfect for Kubernetes.
- Offers service discovery, powerful query language (PromQL), and alerting integration (Alertmanager).

Limitation: These tools are metric-focused and may lack integrated logging/tracing support unless paired with other tools (e.g., Grafana, Loki, Jaeger).

5.2 Observability Platforms: Grafana, New Relic, Datadog, OpenTelemetry

Observability tools aim to provide **full-stack visibility** through unified views of metrics, logs, and traces.

📈 **Grafana Stack (Grafana + Loki + Tempo)**

- Visualization dashboard (Grafana), log aggregation (Loki), tracing (Tempo).
- Flexible, open-source, extensible via plugins.
- Ideal for teams already using Prometheus.

🧠 **New Relic**

- Full observability platform: application performance monitoring (APM), logs, traces, infrastructure.
- AI-assisted insights and anomaly detection.
- SaaS-based; suitable for enterprises.

📊 **Datadog**

- Unified observability: metrics, logs, traces, security.
- Deep cloud-native integrations (Kubernetes, AWS, etc.).
- Includes real user monitoring (RUM) and synthetic testing.

🌐 **OpenTelemetry**

- Open standard for collecting telemetry (metrics, logs, traces).
- Vendor-neutral; integrates with many observability platforms.
- Becoming the industry standard for instrumentation.

🔁 **Insight**: Observability tools are **integrated**, **exploratory**, and **designed for dynamic, distributed systems**.

5.3 Tooling Evolution: From Monitoring to Observability

Feature	Traditional Monitoring	Modern Observability
Focus	Metrics and alerts	Metrics, logs, and traces
Instrumentation	Manual setup and plugins	Auto-instrumentation with SDKs

Feature	Traditional Monitoring	Modern Observability
Flexibility	Predefined thresholds	Dynamic querying and correlation
Architecture Fit	Static and monolithic	Microservices and cloud-native
Tool Examples	Nagios, Zabbix, Prometheus	Grafana, Datadog, New Relic

The shift from monitoring to observability is **not just about tools**, but about embracing a mindset focused on **proactive understanding** rather than just reactive detection.

6. Implementation in DevOps Pipelines

Implementing observability and monitoring into your DevOps CI/CD pipelines ensures proactive visibility across every stage of your software delivery lifecycle. From code to deployment, telemetry must be baked into the workflow. This section explores how to embed monitoring and observability within DevOps pipelines using practical examples and code snippets.

6.1 Integrating Observability in CI/CD Workflows

To ensure observability is **automated and continuous**, telemetry hooks must be included in:

- **Build processes**
  Capture metrics like build duration, success/failure status.
- **Deployment stages**
  Automatically log deployment metadata, like version, environment, and timestamp.
- **Post-deployment monitoring**
  Enable health checks, error tracking, and performance tracing immediately after release.

✅ **GitHub Actions Example: Add Observability to CI/CD**

```
name: CI-CD Pipeline with Observability


on:
  push:
    branches: [main]


jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
      uses: actions/checkout@v3

    - name: Build Application
      run: |
        echo "Starting Build..."
        ./scripts/build.sh
        echo "Build completed"

    - name: Push Deployment Info to Logs
      run: |
        curl -X POST https://logs.example.com/api/deployments \
          -H "Authorization: Bearer ${{ secrets.LOG_API_KEY }}" \
          -H "Content-Type: application/json" \
          -d '{
            "app": "my-app",
            "env": "production",
            "version": "${{ github.sha }}",
            "deployedBy": "${{ github.actor }}"
          }'
```

📌 This sends deployment metadata to your observability backend (e.g., Datadog, Loki, or ELK).

6.2 Instrumenting Applications for Monitoring and Tracing

Modern apps should be instrumented to export metrics, logs, and traces. Libraries like **OpenTelemetry** make this process streamlined.

🚀 **Node.js Example: Tracing with OpenTelemetry**

```
const opentelemetry = require('@opentelemetry/api');
const { NodeTracerProvider } = require('@opentelemetry/sdk-trace-node');
const { SimpleSpanProcessor } = require('@opentelemetry/sdk-trace-base');
const { JaegerExporter } = require('@opentelemetry/exporter-jaeger');

const provider = new NodeTracerProvider();
const exporter = new JaegerExporter({
  serviceName: 'my-node-app',
});

provider.addSpanProcessor(new SimpleSpanProcessor(exporter));
provider.register();

const tracer = opentelemetry.trace.getTracer('my-app-tracer');

// Example span
const span = tracer.startSpan('process_request');
doSomething();
span.end();
```

🔍 This sends trace data to Jaeger, helping you understand how a request flows through your app.

6.3 Dashboards and Alerting Automation

After telemetry is in place, configure:

- **Dashboards** that auto-update based on tags or labels (e.g., service name, environment).
- **Alert rules** for anomalies post-deployment (e.g., 5xx rate increase, latency spikes).

📊 **Prometheus + Alertmanager Example: Alert Rule**

```
groups:
 - name: app-alerts
  rules:
   - alert: HighErrorRate
    expr: rate(http_requests_total{status=~"5.."}[5m]) > 0.05
    for: 2m
    labels:
     severity: critical
    annotations:
     summary: "High error rate detected"
     description: "More than 5% of HTTP requests failed in the last 5 minutes."
```

💡 This triggers an alert when your service starts producing a high rate of 5xx errors.

7. Challenges and Best Practices

Implementing observability and monitoring in a DevOps pipeline isn't without its hurdles. From tool sprawl to performance overhead, many organizations face challenges in scaling visibility effectively. This section outlines the **common pitfalls**, and provides **practical best practices** to ensure a successful and sustainable implementation.

7.1 Common Pitfalls in Implementation

Here are some of the typical mistakes organizations make when implementing observability and monitoring:

❌ **Over-instrumentation**

- Capturing too many metrics or logs without purpose.
- Results in **storage bloat**, increased costs, and slower dashboards.

❌ **Alert Fatigue**

- Too many noisy, non-actionable alerts lead to desensitization.
- Critical incidents may go unnoticed amidst minor warnings.

❌ **Tool Fragmentation**

- Using disconnected tools for logs, metrics, and traces.
- Leads to **context-switching**, data silos, and slow root cause analysis.

🔍 **Real-world issue**: Teams install Prometheus for metrics, ELK for logs, and Jaeger for traces — but without a unified dashboard, visibility is fragmented.

7.2 Best Practices for Effective Adoption

To avoid pitfalls and maximize value, apply these best practices:

☑ **Adopt Open Standards (like OpenTelemetry)**

- Avoid vendor lock-in.
- Easily export telemetry data to multiple backends (Datadog, Prometheus, etc.).

☑ **Define SLOs and SLIs Early**

- Service Level Indicators (e.g., latency < 200ms).
- Service Level Objectives (e.g., 99.9% uptime).
- Use these to build alerts and performance baselines.

✅ **Establish a Culture of Observability**

- Make observability part of developer responsibility.
- Train teams to read traces and investigate logs early in the dev cycle.
- Incorporate observability feedback into retrospectives and root cause reviews.

7.3 Building a Scalable Observability Strategy

Observability must grow with your infrastructure. Here's how to scale effectively:

🧩 **Centralize Telemetry Collection**

- Use unified platforms (e.g., Grafana Stack, Datadog).
- Ensure cross-team visibility and reuse of dashboards/alerts.

🏗 **Leverage Infrastructure-as-Code (IaC)**

- Define alerts, dashboards, and log pipelines as code using Terraform, Helm, etc.

Terraform Example: Datadog Monitor

```
resource "datadog_monitor" "cpu_high" {
  name              = "High CPU usage on {{host.name}}"
  type              = "metric alert"
  query             = "avg(last_5m):avg:system.cpu.user{*} > 80"
  message           = "CPU usage is too high"
  tags              = ["env:prod"]
  notify_no_data    = true
  renotify_interval = 10
}
```

📦 **Sample and Retain Smartly**

- Apply **log sampling** and **trace tailing** to retain only what's needed.
- Use **aggregated metrics** for long-term dashboards, and detailed telemetry only for debugging.

✅ **Key Takeaway**: Successful observability isn't just about collecting data — it's about collecting the **right data**, making it **actionable**, and embedding it into your **engineering culture and processes**.

8. Future Trends and Recommendations

As software systems become increasingly complex, observability and monitoring must evolve to meet the demands of **real-time, intelligent, and automated** DevOps environments. This section explores where the industry is heading and how teams can prepare for what's next.

8.1 AI and ML in Observability

Artificial Intelligence and Machine Learning are transforming observability by adding **predictive** and **automated** capabilities to traditional workflows.

Key Use Cases:

- **Anomaly Detection**: Identify outliers in traffic, latency, and CPU usage before thresholds are breached.
- **Root Cause Analysis**: ML models analyze logs/traces to pinpoint failure causes.
- **Auto-Remediation**: Trigger automated actions (e.g., rollbacks, scaling) based on learned patterns.

Example: Datadog's Watchdog uses machine learning to identify performance regressions and notify teams automatically, reducing MTTR drastically.

8.2 Shift from Reactive to Proactive DevOps

Modern observability empowers a **shift-left** approach — enabling teams to surface performance and reliability issues **earlier** in the development lifecycle.

What This Looks Like:

- **Pre-deployment checks** for baseline compliance (SLOs, security policies).
- **Observability in CI/CD** to catch regressions before they hit users.
- **Proactive capacity planning** based on usage trends.

✅ **Tools supporting this shift:**

- **Chaos Engineering frameworks** like Litmus or Gremlin combined with observability dashboards.
- **Service Reliability tools** that simulate traffic and test scaling.

8.3 Recommendations for Teams and Organizations

🧑‍💻 For Dev Teams:

- **Instrument everything that matters** — not everything you can.
- Learn to read traces like a stack trace.
- Use observability for **feature flag testing**, **A/B analysis**, and **perf optimization**.

🏢 For Organizations:

- Appoint an **Observability Champion** in each team.
- Embed observability reviews in sprint planning and incident postmortems.
- Standardize tooling around **OpenTelemetry** for long-term sustainability.

📊 For Leadership:

- Track business-impacting SLIs (e.g., checkout latency, user session errors).
- Invest in **developer observability training**.
- Treat observability as a **competitive advantage**, not just a cost center.