Complete Observability System Report

Introduction

In modern software architectures, particularly microservices and cloud-native systems, maintaining **reliability, performance, and fault detection** is critical. Observability allows developers and operations teams to monitor applications comprehensively by analyzing **metrics, logs, and traces**.

This project focuses on building a complete observability stack for a Node.js demo application, integrating **metrics collection**, **centralized logging**, and **distributed tracing**, all visualized through Grafana dashboards.

Abstract

The aim of this project is to create an **integrated monitoring system** that allows real-time analysis of application behavior.

The system collects: - **Performance metrics**: HTTP request counts, success and error rates, latency distributions.
- **Centralized logs**: Structured logs from the application aggregated through Loki.
- **Distributed traces**: Request-level traces for debugging and latency analysis via Jaeger.

By combining these elements, we can **identify bottlenecks, trace errors, and improve reliability**, providing actionable insights for developers and DevOps engineers.

Tools Used

- **Node.js**: Demo application generating metrics and logs.
- **Docker & Docker Compose**: Containerized deployment of app and observability stack.
- **Prometheus**: Scrapes application metrics at regular intervals.
- **Grafana**: Dashboard visualization for metrics, logs, and traces.
- **Loki**: Centralized logging system, aggregates structured logs.
- **Promtail**: Agent that tails log files and sends them to Loki.
- **Jaeger**: Distributed tracing to visualize end-to-end request flow.
- **OpenTelemetry**: Automatic instrumentation for exporting traces to Jaeger.
- **Pino Logger**: Fast, structured logging for Node.js applications.

Steps Involved in Building the Project

1. Containerize the Sample App

- Created a Node.js demo application with three routes: `/`, `/work`, `/error`.
- Integrated **Prometheus client** to track:
- Total requests per route

- Request duration (histogram)
- Success and error counters
- Used **Pino logger** for structured logging in JSON format.
- Instrumented **OpenTelemetry** to export traces to Jaeger via OTLP protocol.

2. Setup Docker Compose

- Defined services: `app`, `Prometheus`, `Grafana`, `Loki`, `Promtail`, `Jaeger`.
- Mounted `/logs` directory from host to container for Promtail.
- Exposed required ports: Grafana (3001), Prometheus (9090), Jaeger UI (16686), Loki (3100).

3. Configure Prometheus

- Prometheus configured to scrape metrics from Node.js app (`/metrics`).
- Metrics collected:
- `app_http_requests_total`
- `app_http_success_total`
- `app_http_error_total`
- `app_http_request_duration_seconds`

4. Setup Loki & Promtail

- Promtail tails application logs and sends them to Loki.
- Loki stores logs and exposes query interface in Grafana.
- Logs include **timestamp**, **HTTP method**, **route**, **status code**, **response time**.

5. Setup Jaeger

- OpenTelemetry automatically instruments all HTTP routes.
- Traces exported to Jaeger, showing request duration, route, and any errors.
- Jaeger UI allows end-to-end visualization of traces for debugging.

6. Configure Grafana Dashboards

- Created panels for:
- HTTP Requests per Second (by route and status code)
- Error Rate (%)
- Success Rate (%)
- 95th Percentile Latency
- Centralized logs (Loki)
- Request traces (Jaeger)
- Dashboard automatically refreshes every 10 seconds for real-time monitoring.

7. Validation & Testing

- Verified metrics accuracy via Prometheus query expressions.
- Confirmed logs appear in Grafana (Loki).
- Triggered `/work` route failures to test error rate visualization.
- Checked traces for each request in Grafana (Jaeger panel).

Observed Insights

- **Latency Analysis**: The `/work` endpoint occasionally spikes due to simulated delays (0–500ms).
- **Error Rate Tracking**: 10% of `/work` requests fail randomly, reflected in error rate panel.
- **Success Rate**: Remaining requests confirm service reliability.
- **Correlation**: Traces and logs correlate request errors and latency spikes, making root cause analysis easier.

Conclusion

This project successfully implemented a **complete observability system** for a Node.js application:

- **Metrics** provide high-level performance insights.
- **Centralized logs** allow structured monitoring of requests and errors.
- **Traces** enable request-level debugging and latency analysis.

Deliverables include: - `docker-compose.yml`
- Grafana dashboard JSON
- Logs and trace samples
- Project report (this document)

The integrated observability stack demonstrates **how modern tools can provide actionable insights**, improve reliability, and simplify debugging in distributed applications.