# Observability and Monitoring Strategy

**Target Audience:** DevOps, SREs, Support Engineers
**Purpose:** Ensuring our ability to ask questions about the system's behavior and performance in real-time. "Why is it slow?" "Why did this payment fail?"

---

# 1. The Three Pillars of Observability

We implement the core triad: **Logs**, **Metrics**, and **Traces**.

## 1.1 Logging Strategy (The "What")

- **Format:** Structured **JSON Logging**.
  - *Bad:* `print("User failed login")` (Hard to parse).
  - *Good:* `logger.info("login_failed", user_id="123", reason="bad_password", ip="10.0.0.1")`
- **Levels:**
  - `DEBUG` : Verbose (Local dev only).
  - `INFO` : Standard operational flow (e.g., "Request Received").
  - `WARNING` : Recoverable issues (e.g., "Retry attempt 1").
  - `ERROR` : Actionable failures (e.g., "Database timeout").
- **Aggregation:** All containers pipe `stdout` to a central collector (e.g., Loki or ELK Stack). No SSH-ing into servers to read files.
- **Correlation:** Every log line must include a `trace_id` to link it to a specific request across services.

## 1.2 Metrics Collection (The "Health")

We use the **RED Method** for all microservices:

- **R - Rate:** Number of requests per second (Throughput).
- **E - Errors:** Number of failed requests per second.
- **D - Duration:** Latency distributions (P50, P95, P99).
- **Technology:** Prometheus (Scraping `/metrics` endpoints).

- **Business Metrics:** We also track domain stats:
    - `tickets_sold_total`
    - `payment_revenue_total`
    - `active_buses_count`

### 1.3 Distributed Tracing (The "Where")

- **Problem:** A user clicks "Buy", but gets an error. Was it the Gateway? The Auth Service? Or the Database?
- **Solution: OpenTelemetry**.
- **Mechanism:**
    1. Gateway generates a unique `trace_id` (e.g., `abc-123`).
    2. This ID is passed in HTTP Headers (`X-Trace-Id`) to every downstream service.
    3. Spans are collected in a backend (Jaeger/Tempo).
- **Usage:** Visualizing the "Waterfall" of a request to pinpoint exactly which microservice caused the latency or error.

---

## 2. Error Monitoring (Sentry)

- **Tool:** Sentry (or equivalent).
- **Role:** Capture Unhandled Exceptions and stack traces.
- **Process:**
    1. Code crashes (`NullReferenceException`).
    2. SDK captures variables, user context, and stack trace.
    3. Alert sent to Slack/Teams.
    4. Developers click link to see exactly *line of code* that failed.

---

## 3. Operational Dashboards (Grafana)

We maintain distinct dashboards for different stakeholders.

### 3.1 The "Control Tower" (for Operations)

- **Focus:** Business Health.
- **Widgets:**
    - Live Map of Buses.
    - Revenue Ticker (Last Hour).

- Active User Count.
- "Red/Green" status of all Payment Gateways.

### 3.2 The "Engine Room" (for Engineers)

- **Focus:** System Health.
- **Widgets:**
  - CPU/Memory Usage per Container.
  - Database Connection Pool depth.
  - RabbitMQ Queue Length (Backlog).
  - HTTP 5xx Error Rate.

---

## 4. Alerting Strategy

"Don't wake me up unless it's urgent."

- **Severity 1 (Critical):** System Down, Payments Failing, Data Loss Risk.
  - *Action:* PagerDuty call to On-Call Engineer (24/7).
- **Severity 2 (Warning):** Queue backlog growing, Latency increasing.
  - *Action:* Slack notification to Engineering Channel (Business Hours).
- **Severity 3 (Info):** A single cron job failed (Automatic retry likely).
  - *Action:* Log entry only.

---

## 5. Incident Diagnosis Workflow

1. **Alert Fires:** "High Error Rate on Ticketing Service".
2. **Check Dashboard:** Is it correlated with high traffic? Is the DB CPU high?
3. **Check Traces:** Find a specific failed request trace. See that `Payment Service` is timing out.
4. **Check Logs:** Filter logs for `Payment Service` errors. See "Connection Refused from M-Pesa API".
5. **Conclusion:** The external provider is down.
6. **Action:** Enable "Maintenance Mode" or switch Payment Provider route.