🚨 Debugging Production Outages Like an SRE Pro! 🔥

⚠️ When everything is on fire, where do you start?

As an SRE, I've faced multiple high-severity production outages, some caused by code, others by misconfigured infrastructure, and a few by unexpected behavior in distributed systems.

One of the worst outages I dealt with was a critical database slowdown that led to cascading failures across multiple microservices.

🚀 In this post, I'll take you through:
✅ My structured, step-by-step approach to debugging production incidents
✅ A real-world incident where I solved a severe database performance issue
✅ How observability (logs, metrics, traces) helped in RCA (Root Cause Analysis)
✅ Fixes that improved performance & stability after the outage
✅ Best practices for preventing such issues in the future

📌 The Incident: High Latency & API Failures

🚨 The Problem Statement

A Java-based microservice running on Kubernetes started failing 60% of API requests in production. Users complained about slow responses, and error logs were flooded with timeouts and connection errors.

💥 Impact:

- Customer-facing APIs were returning 500 errors.
- Database latency spiked from 100ms to over 5 seconds.
- Kubernetes HPA (Horizontal Pod Autoscaler) started spinning up more pods, worsening the situation.
- No recent deployments or configuration changes—so what triggered the failure?

🛠️ My Step-by-Step Debugging Approach

Step 1️⃣ Incident Validation – Is It a Real Issue?

📌 Before jumping to conclusions, I first confirmed the issue's severity and scope:

✅ Checked Monitoring Dashboards (Dynatrace, Prometheus, Datadog)

- Observed an increase in API latency (from 100ms → 4.5s).
- CPU usage spiked 80% → 95% on the database node.
- High connection errors detected on the application side.

✅ Checked if the Issue Was Widespread

- Did the issue impact all requests or specific regions?
- Was it affecting one microservice or multiple services?

💡 Findings:

- The outage started at 2:05 PM UTC.
- All API requests depending on the database were slowing down.
- No recent code deployments or infrastructure changes.

Step 2️⃣ Check Logs for Application Errors (Log Correlation & Pattern Analysis)

Since the issue was related to high latency & failures, I pulled logs using the ELK stack (Elasticsearch, Logstash, Kibana) & Grafana Loki.

🔍 Errors found in logs:

[ERROR] 2025-03-07T14:05:32 Connection timeout: Database connection pool exhausted

[WARN] 2025-03-07T14:06:12 Retry attempt failed - Connection refused

[ERROR] 2025-03-07T14:07:45 ReadTimeoutException: Query took too long to execute

📌 Observations:

- The application was timing out while connecting to the PostgreSQL database.
- Retries were overwhelming the database, making things worse.
- Kubernetes HPA (autoscaling) was triggering pod restarts, further worsening the situation.

At this point, I suspected the database was the bottleneck. But why was it suddenly slow?

Step 3️⃣ Correlate Logs with Tracing (Request Flow Analysis)

I used OpenTelemetry tracing to track the request journey across microservices.

📍 Request Flow Observations:

- 🚦 User Request → API Gateway → Microservice → Database
- Database query times jumped from 100ms to 5 seconds.
- Database connection pool was exhausted, leading to timeouts.
- Increased API retries further stressed the database.

🎯 Suspected Root Cause:

- Database performance degradation due to slow queries.

At this stage, I was confident the root cause was within the database layer.

Step 4️⃣ Analyze Database Performance (Find the Bottleneck!)

To dig deeper, I analyzed PostgreSQL performance metrics using:
✅ pg_stat_activity – Check active queries.
✅ pg_stat_statements – Identify long-running queries.
✅ CloudSQL insights (GCP) – Monitor DB CPU & slow queries.

🔎 Findings:

- A slow SQL query was causing table locks & high CPU utilization (95%).
- This query was scanning millions of records due to missing indexing.
- The issue started after a new analytics dashboard was introduced, which triggered frequent unoptimized queries.

🛑 Root Cause Identified!

- A missing index on a large table (orders table) caused full table scans, locking rows and slowing queries.
- API retries flooded the database, further exhausting DB resources.

🚀 Step 5️⃣ Implement the Fix & Recovery

🔹 Immediate Mitigation:
✅ Killed the slow query to stop database locks.
✅ Increased connection pool size temporarily to reduce timeouts.
✅ Disabled automatic retries in the application to prevent additional load.

🔹 Permanent Fixes:
✅ Added proper indexing (CREATE INDEX idx_status ON orders(status);).
✅ Optimized database queries (avoided SELECT *, used pagination).
✅ Set connection pooling best practices (HikariCP tuning).
✅ Implemented circuit breakers (Resilience4j) to stop retry storms.