**Structured Analysis**

**1. Potential Root Causes (at least 5 areas)**

1. Application-Level Issues
   * Inefficient code paths, blocking calls, memory leaks, or unoptimized database queries.
   * Some requests may hit heavy endpoints (e.g., reports, analytics) leading to slow response.
2. Database Performance
   * Slow queries, missing indexes, lock contention, or overloaded database connections.
   * Random spikes can happen if a small % of requests hit a problematic query.
3. Container / Host Resource Bottlenecks
   * EC2 instances running containers might face CPU throttling, memory pressure, or noisy neighbor issues.
   * 20% slowdown could indicate one instance/container occasionally hitting resource limits.
4. Load Balancer Issues
   * Application Load Balancer (ALB) may unevenly distribute traffic, sending more requests to a degraded instance.
   * Slow health checks or sticky sessions can worsen the problem.
5. Network Latency / AWS Infrastructure
   * Intermittent packet loss, AZ-specific latency, or cross-zone routing issues.
   * Users may randomly hit a slower path.
6. External Dependencies (bonus)
   * API calls to third-party services delaying responses.
   * Without proper timeouts/retries, upstream delays can propagate.

**2. Troubleshooting Approach**

Step-by-step investigation with tools:

1. Confirm User Reports
   * Collect logs from ALB and application.
   * Measure response time distribution (p50, p90, p99).
2. Check ALB Metrics
   * Use CloudWatch to analyze TargetResponseTime, 5xx errors, and request distribution.
   * Ensure load balancing is even across targets.
3. Inspect Application Containers
   * Use docker stats or ECS/EKS metrics.
   * Look for CPU/memory spikes during slow periods.
4. Database Investigation
   * Enable query logging (e.g., MySQL slow query log).
   * Check for spikes in query latency.
   * Run EXPLAIN on heavy queries.
5. Network Checks
   * Use VPC Flow Logs to detect anomalies.
   * Ping/traceroute between services to spot packet loss or latency.
6. Correlate with Time
   * Look for patterns (time of day, batch jobs, autoscaling events).

**Tools used:**

* AWS CloudWatch & X-Ray (latency tracing).
* Prometheus + Grafana (metrics).
* Jaeger or OpenTelemetry (distributed tracing).
* ELK/Opensearch stack for logs.

**3. Monitoring Solution**

A comprehensive monitoring strategy:

* Metrics
  + Collect container CPU, memory, network I/O.
  + ALB request latency and target health.
  + Database query performance.
* Distributed Tracing
  + Use AWS X-Ray or OpenTelemetry to trace requests across microservices.
  + Identify which hop (DB, external API, or container) is slow.
* Logging
  + Structured logging (JSON) sent to CloudWatch Logs or ELK.
  + Correlate logs with request IDs for full request visibility.
* Alerting
  + CloudWatch alarms for p95 latency > 1s.
  + Alerts for CPU > 80% or DB connections saturation.
* Dashboards
  + Grafana dashboards with latency heatmaps.
  + Compare normal vs degraded requests.

**4. Incident Response Plan**

Escalation Procedures

1. First-level support (on-call engineer) investigates metrics/dashboards.
2. If app-level suspected → escalate to backend dev team.
3. If infra-level suspected → escalate to SRE/DevOps.
4. If third-party dependency suspected → contact vendor/support.

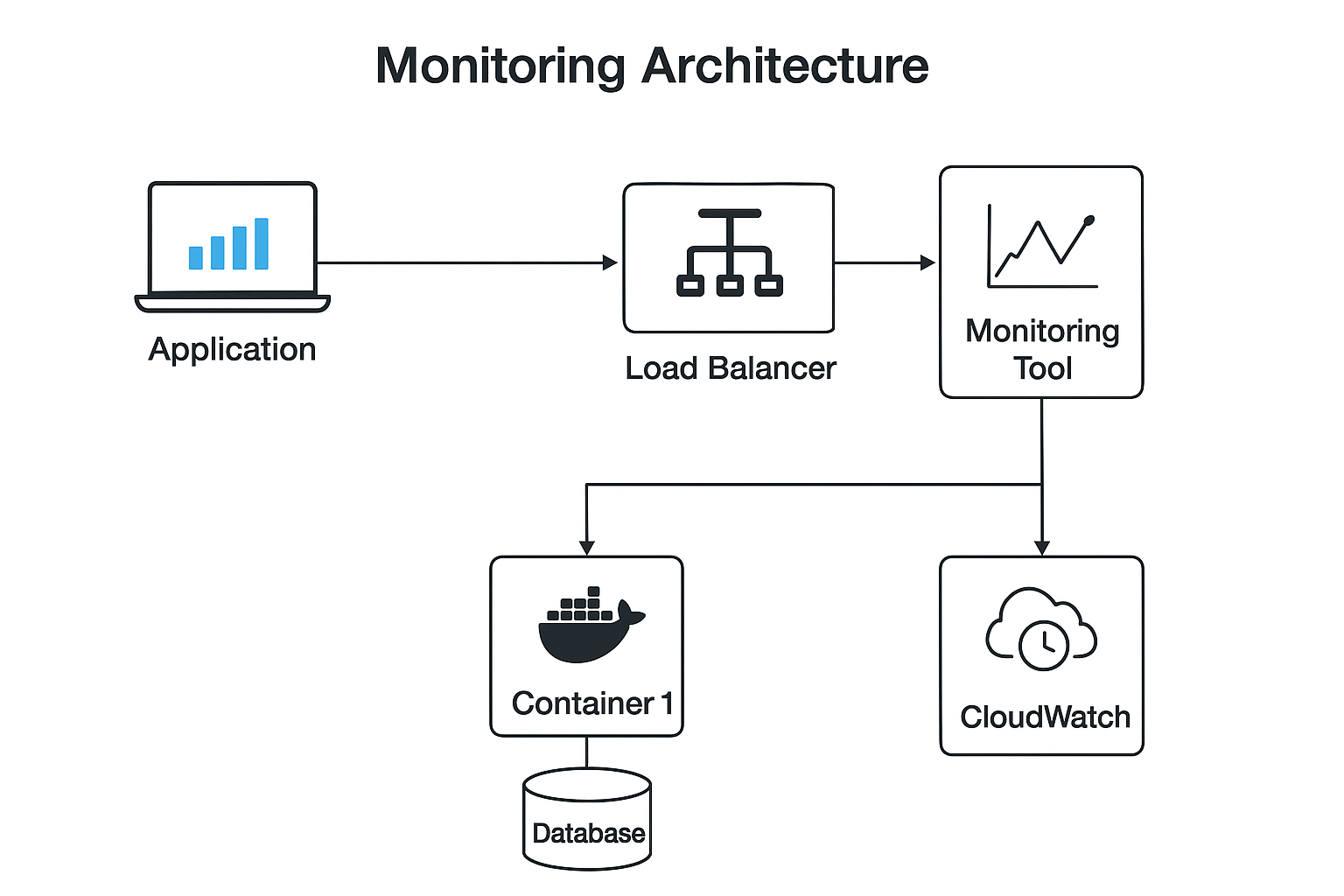
Communication Protocols

* Post updates in incident Slack/Teams channel every 15–30 mins.
* Update status page if >10% of users impacted.
* Write incident report post-resolution with root cause and corrective actions.

**5. Success Metrics**

We measure success by:

* Latency Improvement: p95 latency consistently < 800ms.
* Error Rate: <1% failed requests over 24h.
* Stability: No more random 20% slowdowns.
* Alert Responsiveness: MTTR (mean time to recover) < 15 minutes.
* User Satisfaction: Reduced complaints and improved monitoring signals.
* **Monitoring architecture diagram:**



* **Incident Response and Troubleshooting Flowchart:**

