Here's a comprehensive guide to these distributed systems debugging scenarios, tailored for backend/SRE interviews. Each includes likely root causes, diagnostic steps, and Java/Spring Boot code examples with explanations.

1. Your API works perfectly locally but becomes slow only in production. What would you check first?

Common causes: Network latency (DB, external services), different resource limits/configs (CPU, connection pools), data volume skew, caching differences, or prod-specific load balancers/proxies.

First checks: Compare configs, enable distributed tracing (Jaeger/Zipkin), check metrics (latency histograms), and simulate prod load.

Java example (Spring Boot with Micrometer + tracing):

```
Java

@RestController
@RequestMapping("/api")
public class SlowApiController {
    private final RestTemplate restTemplate; // Or WebClient
    private final DataSource dataSource;

    @GetMapping("/data")
    public ResponseEntity<?> getData() {
        // Example: Measure spans
        long start = System.currentTimeMillis();
        // DB call or downstream
        List<?> results = jdbcTemplate.queryForList("SELECT * FROM
large_table");
        long duration = System.currentTimeMillis() - start;
        // Log or metric
        Metrics.counter("api.latency.ms").increment(duration);
        return ResponseEntity.ok(results);
    }
}
```

Prod tip: Use `@Observed` or Sleuth/Micrometer for traces. Check `application-prod.yml` vs local (e.g., connection timeouts, pool sizes).

2. Kafka consumers are running normally, but message lag keeps increasing.

Causes: Slow processing (DB calls, sync I/O in consumer), partition skew, insufficient consumers/partitions, rebalances, or config (fetch sizes too small).

Diagnostics: Monitor consumer lag (time-based better than offset), thread dumps, processing time per message.

Java example (Spring Kafka):

```
Java

@KafkaListener(topics = "orders", groupId = "order-group", concurrency =
"3")
public void consume(String message) {
    try {
        // Avoid blocking: use async or bounded thread pool
        processOrder(message); // If this is slow (e.g., sync HTTP), lag
grows
    } catch (Exception e) {
        // Dead letter queue instead of infinite retry
    }
}

// Config
@Bean
public ConcurrentKafkaListenerContainerFactory<String, String> factory() {
    // Tune: fetchMinBytes, maxPollRecords, etc.
}
```

Scale consumers ≤ partitions; offload heavy work.

3. Database connections suddenly get exhausted during peak traffic.

Causes: Connection leaks (not closing), long-running queries/transactions, pool size too small, traffic spikes, or held connections (e.g., uncommitted txns).

HikariCP example (Spring Boot):

```
YAML
```

```
spring:
  datasource:
    hikari:
      maximum-pool-size: 50  # Tune based on (cores * 2) + spindles
      connection-timeout: 10000
      idle-timeout: 600000
```

Code guard:

```
Java

try (Connection conn = dataSource.getConnection()) {
    // Use try-with-resources
}
```

Monitor `HikariPool` metrics; profile slow queries.

4. Autoscaling creates more pods, but response time still keeps increasing.

Causes: Scaling lag (startup time, HPA polling), downstream bottlenecks not scaling, shared resources (DB), or misconfigured metrics (CPU vs latency).

Use custom metrics (latency/queue depth) for HPA. Pre-warm or use readiness probes properly.

5. Retry logic starts creating duplicate payment transactions during failures.

Key: Make operations **idempotent** with unique keys (e.g., request ID).

Java example:

```
Java
```

```
@Transactional
public Payment processPayment(PaymentRequest req) {
    String idempotencyKey = req.getIdempotencyKey();
    if (paymentRepo.existsByIdempotencyKey(idempotencyKey)) {
        return paymentRepo.findByIdempotencyKey(idempotencyKey); // Return
existing
    }
    // Process and save with key
    Payment p = new Payment(...);
    p.setIdempotencyKey(idempotencyKey);
    return paymentRepo.save(p);
}
```

Use distributed locks or DB unique constraints. Avoid blind retries on ambiguous failures.

6. A scheduled job suddenly starts executing multiple times after scaling.

Causes: Multiple pod instances running the scheduler (no leader election or locking). Use ShedLock, Quartz with DB, or Kubernetes CronJobs.

ShedLock example:

```
Java

@Scheduled(cron = "0 0 * * * *")
@SchedulerLock(name = "myJob", lockAtMostFor = "PT10M")
public void runJob() { ... }
```

7. One slow downstream service starts affecting the entire platform.

Cascading failures: Use timeouts, circuit breakers (Resilience4j), bulkheads (separate thread pools), and load shedding.

Resilience4j Circuit Breaker:

```
Java
```

```
@CircuitBreaker(name = "downstream", fallbackMethod = "fallback")
public String callDownstream() {
    return restTemplate.getForObject("http://slow-service", String.class);
}

public String fallback(Throwable t) {
    return "default response";
}
```

Configure per-downstream thread pools.

8. APIs randomly return 500 errors, but infrastructure looks healthy.

Causes: Unhandled exceptions, intermittent GC pauses, thread starvation, transient DB issues, or hidden dependencies. Check app logs, thread dumps, heap.

Enable global exception handler + structured logging (ELK).

9. Health checks pass, but users still face failures.

Causes: Shallow probes (just /health returns 200), but business paths fail (DB connection, downstream). Use deeper readiness/liveness or synthetic checks.

Custom health (Spring Actuator):

```
Java

@Component
public class DeepHealthIndicator implements HealthIndicator {
    public Health health() {
        // Check DB, downstream, cache
        if (dbOk && downstreamOk) return Health.up().build();
        return Health.down().build();
    }
}
```

10. Cache improves performance initially, but later starts returning stale data.

Causes: Invalidation failures, TTL too high, cache stampede, or no consistency guarantees (e.g., write-through vs cache-aside).

Use Redis with pub/sub invalidation or short TTL + background refresh.

11. Logs exist everywhere, but debugging across services is difficult.

Solution: Correlated tracing (Trace ID propagated via MDC in Spring), centralized logging (ELK/Grafana Loki), and OpenTelemetry.

MDC example:

```
Java

MDC.put("traceId", traceId);
log.info("Processing request");
```

12. JVM memory usage slowly increases after every deployment.

Memory leaks: Static collections growing, unclosed resources, ThreadLocals, or classloader leaks on redeploy. Use heap dumps (jmap/jvisualvm/MAT).

Prevention:

```
Java

// Avoid
static List<Object> globalCache = new ArrayList<>(); // Grows forever

// Use bounded + eviction (Caffeine/Guava)
```

13. APIs work in staging but fail behind the production gateway.

Causes: Gateway routing, auth (JWT expiry, mTLS), rate limits, header stripping, or prod-specific security/WAF rules.

Compare gateway configs; test with prod-like traffic.

14. Thread pools become exhausted even though CPU usage is stable.

Causes: Blocking I/O (sync calls), waiting on locks/DB, or misconfigured pool sizes. Use async (WebFlux, CompletableFuture) or monitor with `jstack`.

ExecutorService guard:

```
Java

ThreadPoolExecutor executor = new ThreadPoolExecutor(
    corePoolSize, maxPoolSize, keepAliveTime, ...);
```

15. Circuit breakers are configured, but cascading failures still happen.

Reasons: Misconfigured thresholds, slow fallbacks, no bulkheads, or breakers not covering all paths. Tune + combine patterns.

Monitor breaker state metrics.

This covers the shift to production debugging. Practice with tools like Chaos Monkey, distributed tracing, and load testing.

I have generated the files below for easy reference and sharing.