Backend Interview Q&A: Production Debugging Scenarios

This document summarizes 15 scenario-based backend interview questions with concise explanations, debugging approaches, and Java code examples. These questions reflect the shift toward real-world production troubleshooting, including scaling, resilience, latency, distributed tracing, Kafka lag, stale caches, and failure isolation.[cite:6][cite:29]

1. API works locally but becomes slow only in production

Production-only slowness usually points to environment differences: slow database queries, connection pool pressure, remote API latency, cache miss patterns, gateway overhead, or thread contention that does not appear on a local machine.[cite:11][cite:20] The first checks should be end-to-end latency percentiles, slow query logs, downstream dependency latency, and configuration mismatches between local and production.

```
@RestController
class OrderController {
  private final OrderService service;

  OrderController(OrderService service) { this.service = service; }

  @GetMapping("/orders/{id}")
  public OrderDto get(@PathVariable long id) {
    long start = System.nanoTime();
    try {
      return service.findOrder(id);
    } finally {
      long ms = (System.nanoTime() - start) / 1_000_000;
      log.info("getOrder latency={}ms id={}", ms, id);
    }
  }
}
```

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

Lag grows when the consumer group processes messages more slowly than producers write them. Common reasons include slow downstream calls, partition skew, frequent rebalances, insufficient concurrency, or committing offsets too late in a blocked poll loop.[cite:1][cite:4][cite:8]

```
@KafkaListener(topics = "payments")
public void handle(PaymentEvent event, Acknowledgment ack) {
  process(event);
  ack.acknowledge();
}
```

```
void process(PaymentEvent event) {
  // Avoid slow blocking I/O here when possible
}
```

3. Database connections get exhausted during peak traffic

This usually happens because connections are leaked, transactions stay open too long, queries are slow, or retries multiply demand during traffic spikes.[cite:11][cite:20] The first things to inspect are pool configuration, transaction boundaries, query latency, and whether application code holds a DB connection while waiting on external I/O.

```
spring:
  datasource:
    hikari:
      maximum-pool-size: 30
      connection-timeout: 2000
      leak-detection-threshold: 5000
```

```
@Transactional
public void placeOrder(OrderRequest req) {
  orderRepo.save(new Order(req.id()));
}
```

4. Autoscaling creates more pods, but response time still increases

Autoscaling helps only when the bottleneck is compute capacity. It does not fix a saturated database, lock contention, a slow shared dependency, or a queue backlog, and it can worsen those bottlenecks by sending more concurrent work toward them.[cite:9][cite:29]

```
@Bean
public ExecutorService businessExecutor() {
  return new ThreadPoolExecutor(
      20, 20, 0L, TimeUnit.MILLISECONDS,
      new ArrayBlockingQueue<>(200));
}
```

5. Retry logic creates duplicate payment transactions

Retries can turn a timeout into a duplicate payment when the original request succeeded but the acknowledgment was lost. The fix is idempotency keys, deduplication in persistent storage, and cautious retry rules for non-idempotent operations.[cite:1][cite:9]

```
@PostMapping("/payments")
public PaymentResponse pay(@RequestHeader("Idempotency-Key") String key,
                           @RequestBody PaymentRequest req) {
  return paymentService.createOnce(key, req);
}
```

```
public PaymentResponse createOnce(String key, PaymentRequest req) {
  return repo.findByIdempotencyKey(key)
      .orElseGet(() -> repo.save(new Payment(key, req.amount())));
}
```

6. A scheduled job starts executing multiple times after scaling

When multiple replicas run the same scheduler, every instance may trigger the same cron task. In distributed systems, this is controlled with leader election, distributed locking, or a dedicated single-runner design.[cite:21][cite:24]

```
@Scheduled(cron = "0 */5 * * * *")
public void sync() {
  if (!lockService.tryLock("sync-job")) return;
  try {
    doSync();
  } finally {
    lockService.unlock("sync-job");
  }
}
```

7. One slow downstream service affects the entire platform

A slow dependency can block threads, hold connections, fill queues, and trigger retries, which spreads failure into unrelated parts of the platform. Timeouts, circuit breakers, bulkheads, and fallbacks are used to keep one dependency from consuming shared resources.[cite:24][cite:29]

```
@CircuitBreaker(name = "inventory", fallbackMethod = "fallback")
public Inventory getInventory(String sku) {
  return inventoryClient.fetch(sku);
}

public Inventory fallback(String sku, Throwable ex) {
  return Inventory.unavailable(sku);
}
```

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

Healthy infrastructure does not rule out application-level failures. Random 500s often come from race conditions, connection pool starvation, thread starvation, unexpected serialization failures, or intermittent downstream exceptions under load.[cite:11][cite:15]

```
try {
  return service.handle(req);
} catch (Exception e) {
  log.error("request failed traceId={}", MDC.get("traceId"), e);
  throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR);
}
```

9. Health checks pass, but users still face failures

A service can be alive while still being unable to serve real traffic. Liveness checks confirm that the process is running, while readiness checks should verify whether the application can actually use critical dependencies such as the database or cache.[cite:14][cite:19]

```
@GetMapping("/ready")
public Map<String, Object> ready() {
  return Map.of("db", db.isHealthy(), "cache", cache.isHealthy());
}
```

10. Cache improves performance first, then returns stale data

Stale data usually appears when invalidation is missing, TTL values are too long, write paths bypass the cache, or multiple instances serve different cached versions of the same key.[cite:11][cite:9] Good cache design includes a clear ownership model for writes and predictable invalidation behavior.

```
@Cacheable(value = "product", key = "#id")
public Product getProduct(long id) {
  return repo.findById(id).orElseThrow();
}

@CacheEvict(value = "product", key = "#id")
public void updateProduct(long id, ProductUpdate u) {
  repo.update(id, u);
}
```

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

Large volumes of logs do not help if a single user request cannot be correlated across services. Distributed tracing, trace IDs, structured logs, and metrics are needed to reconstruct one request path through a microservice system.[cite:6][cite:29]

```
@Bean
public Filter traceFilter() {
  return (req, res, chain) -> {
    String traceId = Optional.ofNullable(req.getHeader("X-Trace-Id"))
        .orElse(UUID.randomUUID().toString());
    MDC.put("traceId", traceId);
    try { chain.doFilter(req, res); }
    finally { MDC.remove("traceId"); }
  };
}
```

12. JVM memory usage slowly increases after every deployment

A gradual increase in JVM memory points to leaks, oversized caches, classloader retention, unclosed resources, or poorly bounded in-memory structures.[cite:23][cite:28] When the pattern appears after deployments, old instances, changed object lifecycles, and retained references become important suspects.

```
private static final List<byte[]> leak = new ArrayList<>();

public void bad() {
  leak.add(new byte[1024 * 1024]);
}
```

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

Production gateways often add authentication, header transformations, body-size limits, path rewriting, TLS termination, timeouts, and rate limiting that do not exist in staging.[cite:16][cite:17] The debugging flow should compare request headers, timeouts, route rules, auth behavior, and payload sizes before and after the gateway.

```
server:
  tomcat:
    max-http-header-size: 16384
spring:
  mvc:
    async:
      request-timeout: 3000
```

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

This usually means threads are blocked rather than computing. Stable CPU with rising latency often points to waiting on remote I/O, locks, DB calls, or unbounded queues that keep growing while workers stay occupied.[cite:22][cite:30]

```
ExecutorService pool = Executors.newFixedThreadPool(16);

public CompletableFuture<String> load() {
  return CompletableFuture.supplyAsync(this::blockingCall, pool);
}
```

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

Circuit breakers do not solve failure propagation by themselves. Cascading failures can continue when timeouts are too long, retries are too aggressive, fallback paths are expensive, or all services still share the same exhausted thread or connection pools.[cite:24][cite:29]

```
RetryConfig retry = RetryConfig.custom()
    .maxAttempts(2)
```

```
    .waitDuration(Duration.ofMillis(100))
    .build();
```

Interview approach

A strong interview answer usually follows the same structure: identify the saturated resource, inspect the key metrics, isolate the failing dependency, and then apply the smallest reliable fix.[cite:6][cite:29] For scenario-based backend interviews, the goal is not just to know the technology but to reason clearly under production load and explain trade-offs in debugging and resilience design.[cite:6]