Java Backend Interview Q & A

Distributed Systems Debugging Under Production Load

15 Real-World Scenarios with Code Examples

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

The most common culprit is the database. Local environments typically use in-memory or local databases with minimal data, while production has large datasets, network latency, and concurrent connections. Query execution plans differ dramatically between environments.

Root Causes:

- Missing database indexes on production tables
- N+1 query problems that only surface with real data volume
- Connection pool misconfiguration - too few connections for production load
- Slow queries without proper EXPLAIN ANALYZE review
- Network latency between app servers and database in production

Solutions:

- Enable query logging and analyze slow queries with EXPLAIN ANALYZE
- Verify connection pool size matches production concurrency needs
- Add missing indexes based on query patterns
- Use caching for frequently accessed data
- Consider read replicas for read-heavy workloads

Code Example:

```
"color: #6a9955;">// Anti-pattern: N+1 query problem
public List<Order> getOrdersWithItems() {
    List<Order> orders = orderRepository.findAll(); "color: #6a9955;">// 1 query
    for (Order order : orders) {
        "color: #6a9955;">// N queries - one per order!
        List<OrderItem> items = orderItemRepository.findByOrderId(order.getId());
        order.setItems(items);
```

```
    }
    return orders;
}

"color: #6a9955;">// Solution: Use JOIN FETCH or EntityGraph
@Query("SELECT o FROM Order o LEFT JOIN FETCH o.items WHERE o.status = :status")
List<Order> findOrdersWithItems(@Param("status") OrderStatus status);

"color: #6a9955;">// Or use EntityGraph to avoid N+1
@EntityGraph(attributePaths = {"items", "items.product"})
List<Order> findByStatus(OrderStatus status);

"color: #6a9955;">// Connection pool configuration (HikariCP)
@Bean
public HikariDataSource dataSource() {
    HikariConfig config = new HikariConfig();
    config.setJdbcUrl("jdbc:postgresql:"color: #6a9955;">//prod-db:5432/mydb");
    config.setUsername("user");
    config.setPassword("pass");
    "color: #6a9955;">// Production tuning
    config.setMaximumPoolSize(50);      "color: #6a9955;">// Match your concurrency
    config.setMinimumIdle(10);           "color: #6a9955;">// Keep warm connections
    config.setConnectionTimeout(30000);  "color: #6a9955;">// 30s max wait
    config.setIdleTimeout(600000);       "color: #6a9955;">// 10 min idle timeout
    config.setMaxLifetime(1800000);      "color: #6a9955;">// 30 min max lifetime
    config.setLeakDetectionThreshold(60000); "color: #6a9955;">// Detect leaks
    return new HikariDataSource(config);
}
```

> ***Key Takeaway:*** *Always profile database queries in production-like data volumes. Local performance is never representative.*

2. Kafka consumers are running normally, but message lag keeps increasing. Why can this happen?

Consumer lag increases when the rate of message production exceeds the rate of consumption. Even if consumers appear 'running', they may not be processing messages fast enough due to slow processing logic, insufficient partitions, or consumer rebalancing issues.

Root Causes:

- Consumer processing logic is too slow (blocking I/O, heavy computation)
- Insufficient consumer instances for the partition count
- Consumer rebalancing storms due to frequent pod restarts
- Poison pill messages causing consumer to get stuck
- Commit failures causing consumers to reprocess messages

Solutions:

- Scale out consumers (ensure partition count >= consumer count)
- Optimize message processing - use async processing where possible
- Increase max.poll.records to process more per batch
- Implement dead letter queues for poison messages
- Monitor consumer lag with alerts (Burrow or Kafka exporter)

Code Example:

```
"color: #6a9955;">// Anti-pattern: Synchronous blocking processing
@KafkaListener(topics = "orders", groupId = "order-processor")
public void consume(ConsumerRecord<String, Order> record) {
    "color: #6a9955;">// Blocking call to slow downstream service
    paymentService.processPayment(record.value()); "color: #6a9955;">// 2-3 seconds!
    sendEmailNotification(record.value());         "color: #6a9955;">// Another 1 second
}

"color: #6a9955;">// Solution: Async processing with CompletableFuture
@KafkaListener(topics = "orders", groupId = "order-processor")
public void consumeBatch(List<ConsumerRecord<String, Order>> records) {
    List<CompletableFuture<Void>> futures = records.stream()
        .map(record -> CompletableFuture.runAsync(() -> {
            try {
                processOrder(record.value());
            } catch (Exception e) {
```

```
                sendToDeadLetterQueue(record, e);
            }
        }, executorService))
        .toList();

    "color: #6a9955;">// Wait for batch to complete
    CompletableFuture.allOf(futures.toArray(new CompletableFuture[0]))
        .join();
}

"color: #6a9955;">// Proper consumer configuration
@Bean
public ConsumerFactory<String, Order> consumerFactory() {
    Map<String, Object> props = new HashMap<>();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka:9092");
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "order-processor");
    props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500); "color: #6a9955;">// Process
    props.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, 300000); "color: #6a9955;">// 5
    props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 45000);
    props.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 15000);
    "color: #6a9955;">// Disable auto-commit for better control
    props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
    return new DefaultKafkaConsumerFactory<>(props);
}

"color: #6a9955;">// Manual commit after successful processing
@KafkaListener(topics = "orders")
public void consume(ConsumerRecord<String, Order> record, Acknowledgment ack) {
    try {
        processOrder(record.value());
        ack.acknowledge(); "color: #6a9955;">// Only commit after success
    } catch (Exception e) {
        sendToDLQ(record, e);
        ack.acknowledge(); "color: #6a9955;">// Commit to skip poison pill
    }
}
```

> ***Key Takeaway:*** *Consumer lag is a throughput problem. Measure processing time per message and ensure (messages/sec * processing time) < consumer capacity.*

3. Database connections suddenly get exhausted during peak traffic. What could cause this?

Connection exhaustion occurs when all connections in the pool are in use and new requests must wait or fail. This often happens due to connection leaks, long-running transactions, or pool size being too small for traffic spikes.

Root Causes:

- Connection leaks - connections not closed in finally blocks
- Long-running transactions holding connections
- Pool size too small for concurrent request volume
- Slow queries keeping connections occupied
- Connection pool not configured with proper timeouts

Solutions:

- Always use try-with-resources for connections
- Set transaction timeouts and query timeouts
- Size connection pool based on (max concurrent requests * avg query time)
- Enable connection leak detection in HikariCP
- Use circuit breakers for database calls

Code Example:

```
"color: #6a9955;">// Anti-pattern: Connection leak
public User getUser(Long id) {
    Connection conn = dataSource.getConnection(); "color: #6a9955;">// Leaked if exceptio
    PreparedStatement stmt = conn.prepareStatement("SELECT * FROM users WHERE id = ?");
    stmt.setLong(1, id);
    ResultSet rs = stmt.executeQuery();
    if (rs.next()) {
        return mapUser(rs); "color: #6a9955;">// If exception here, conn never closed!
    }
    return null;
}

"color: #6a9955;">// Solution: Try-with-resources
public User getUser(Long id) {
    String sql = "SELECT * FROM users WHERE id = ?";
    try (Connection conn = dataSource.getConnection();
```

```
         PreparedStatement stmt = conn.prepareStatement(sql)) {
        stmt.setLong(1, id);
        try (ResultSet rs = stmt.executeQuery()) {
            if (rs.next()) {
                return mapUser(rs);
            }
        }
    } catch (SQLException e) {
        throw new DataAccessException("Failed to fetch user", e);
    }
    return null;
}

"color: #6a9955;">// Even better: Use JPA with @Transactional timeout
@Service
public class UserService {

    @Transactional(timeout = 5) "color: #6a9955;">// 5 seconds max
    public User getUserWithOrders(Long userId) {
        User user = userRepository.findById(userId)
            .orElseThrow(() -> new UserNotFoundException(userId));
        "color: #6a9955;">// Hibernate will manage connection lifecycle
        return user;
    }

    @Transactional(timeout = 10,
                   propagation = Propagation.REQUIRES_NEW,
                   isolation = Isolation.READ_COMMITTED)
    public void processPayment(Long orderId) {
        "color: #6a9955;">// Short, focused transaction
        Order order = orderRepository.findById(orderId).orElseThrow();
        order.setStatus(PAID);
        orderRepository.save(order);
        "color: #6a9955;">// Transaction commits here, connection returned to pool
    }
}

"color: #6a9955;">// HikariCP monitoring
@Component
public class ConnectionPoolMonitor {

    @Autowired
    private HikariDataSource dataSource;

    @Scheduled(fixedRate = 60000)
    public void logPoolMetrics() {
        HikariPoolMXBean poolMXBean = dataSource.getHikariPoolMXBean();
        System.out.println("Active: " + poolMXBean.getActiveConnections());
        System.out.println("Idle: " + poolMXBean.getIdleConnections());
        System.out.println("Waiting: " + poolMXBean.getThreadsAwaitingConnection());
        System.out.println("Total: " + poolMXBean.getTotalConnections());
    }
}
```

> ***Key Takeaway:*** *Connections are a finite resource. Keep transactions short, use try-with-resources, and size your pool based on actual concurrency*

> *measurements.*

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

Adding more instances doesn't help if the bottleneck isn't CPU. Common issues include database contention, lock contention, shared resource saturation, or the new pods themselves causing cascading problems like cache thrashing.

Root Causes:

- Database connection pool saturation - more pods = more connections needed
- Shared resource contention (locks, semaphores, distributed caches)
- Cache thrashing - each new pod has cold cache
- Thread pool exhaustion in downstream services
- Network bandwidth saturation

Solutions:

- Identify the real bottleneck with distributed tracing (Jaeger/Zipkin)
- Use connection pooling at the service mesh level
- Implement rate limiting to protect shared resources
- Warm up caches before bringing pods into rotation
- Scale database tier independently from application tier

Code Example:

```
"color: #6a9955;">// Anti-pattern: Shared lock causing contention
@Service
public class InventoryService {
    private final Object lock = new Object(); "color: #6a9955;">// Shared across all pods

    public void updateStock(Long productId, int quantity) {
        synchronized (lock) { "color: #6a9955;">// Only works within ONE JVM!
            "color: #6a9955;">// In multi-pod setup, this doesn&#039;t prevent concurrent
            inventoryRepository.updateStock(productId, quantity);
        }
    }
}

"color: #6a9955;">// Solution: Distributed locking with Redis/Database
@Service
```

```
public class InventoryService {

    @Autowired
    private StringRedisTemplate redisTemplate;

    public void updateStock(Long productId, int quantity) {
        String lockKey = "lock:inventory:" + productId;
        String token = UUID.randomUUID().toString();

        "color: #6a9955;">// Try to acquire distributed lock
        Boolean locked = redisTemplate.opsForValue()
            .setIfAbsent(lockKey, token, Duration.ofSeconds(10));

        if (Boolean.TRUE.equals(locked)) {
            try {
                "color: #6a9955;">// Use optimistic locking with version field
                inventoryRepository.updateStockWithVersion(productId, quantity);
            } finally {
                "color: #6a9955;">// Release lock only if we own it
                String currentValue = redisTemplate.opsForValue().get(lockKey);
                if (token.equals(currentValue)) {
                    redisTemplate.delete(lockKey);
                }
            }
        } else {
            throw new ConflictException("Concurrent update in progress");
        }
    }
}

"color: #6a9955;">// Rate limiting to protect downstream
@Component
public class RateLimitingFilter extends OncePerRequestFilter {

    private final Map<String, RateLimiter> limiters = new ConcurrentHashMap<>();

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain chain) throws ServletException, IOExcepti
        String clientId = request.getHeader("X-Client-ID");
        RateLimiter limiter = limiters.computeIfAbsent(clientId,
            k -> RateLimiter.create(100)); "color: #6a9955;">// 100 req/sec per client

        if (limiter.tryAcquire()) {
            chain.doFilter(request, response);
        } else {
            response.setStatus(429); "color: #6a9955;">// Too Many Requests
            response.getWriter().write("Rate limit exceeded");
        }
    }
}

"color: #6a9955;">// Circuit breaker for downstream calls
@CircuitBreaker(name = "paymentService", fallbackMethod = "paymentFallback")
public PaymentResult processPayment(PaymentRequest request) {
    return paymentClient.charge(request);
}
```

```
private PaymentResult paymentFallback(PaymentRequest request, Exception ex) {
    return PaymentResult.queued(request.getId());
}
```

> ***Key Takeaway:*** *Scaling out only works when the bottleneck is CPU. Use distributed tracing to find the real constraint before adding more instances.*

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

Retries without idempotency guarantees are dangerous for non-idempotent operations like payments. If the request succeeds on the server but the response is lost, the client retries and creates a duplicate.

Root Causes:

- Retrying non-idempotent operations without idempotency keys
- Network timeouts where request actually succeeded
- Client-side retries without server-side deduplication
- Lack of idempotency key validation
- Race conditions in idempotency check implementation

Solutions:

- Generate idempotency keys for every payment request
- Store idempotency keys with TTL in Redis/database
- Use database unique constraints on idempotency keys
- Implement idempotency at the API gateway level
- Design payment operations to be naturally idempotent

Code Example:

```
"color: #6a9955;">// Anti-pattern: Retry without idempotency
public class PaymentClient {
    @Retryable(value = {IOException.class}, maxAttempts = 3, backoff = @Backoff(delay = 1
    public PaymentResult charge(BigDecimal amount) {
        "color: #6a9955;">// If this succeeds on server but response times out,
        "color: #6a9955;">// retry creates DUPLICATE payment!
        return restTemplate.postForObject("/payments", amount, PaymentResult.class);
    }
}

"color: #6a9955;">// Solution: Idempotency key pattern
@Service
public class PaymentService {

    @Autowired
    private IdempotencyKeyRepository idempotencyRepo;

    @Autowired
```

```
    private PaymentRepository paymentRepo;

    @Transactional
    public PaymentResult processPayment(PaymentRequest request, String idempotencyKey) {
        "color: #6a9955;">// 1. Check if we&#039;ve seen this key before
        Optional<IdempotencyKey> existing = idempotencyRepo.findByKey(idempotencyKey);
        if (existing.isPresent()) {
            "color: #6a9955;">// Return cached response
            return existing.get().getResponse();
        }

        "color: #6a9955;">// 2. Process payment (with DB unique constraint as safety net)
        Payment payment = new Payment();
        payment.setAmount(request.getAmount());
        payment.setIdempotencyKey(idempotencyKey);
        payment.setStatus(PaymentStatus.PROCESSING);

        try {
            PaymentResult result = paymentGateway.charge(request);
            payment.setStatus(PaymentStatus.COMPLETED);
            payment.setTransactionId(result.getTransactionId());

            "color: #6a9955;">// 3. Save idempotency key with response
            IdempotencyKey key = new IdempotencyKey();
            key.setKey(idempotencyKey);
            key.setResponse(result);
            key.setExpiresAt(Instant.now().plus(24, ChronoUnit.HOURS));
            idempotencyRepo.save(key);

            return result;
        } catch (Exception e) {
            payment.setStatus(PaymentStatus.FAILED);
            throw e;
        } finally {
            paymentRepo.save(payment);
        }
    }
}

"color: #6a9955;">// Entity with unique constraint
@Entity
@Table(name = "payments",
       uniqueConstraints = @UniqueConstraint(columnNames = "idempotency_key"))
public class Payment {
    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "idempotency_key", nullable = false, unique = true)
    private String idempotencyKey;

    private BigDecimal amount;
    private String transactionId;
    @Enumerated(EnumType.STRING)
    private PaymentStatus status;
}

"color: #6a9955;">// Client-side: Generate and persist idempotency key
```

```
public class PaymentClient {
    public PaymentResult charge(BigDecimal amount) {
        String idempotencyKey = generateIdempotencyKey();

        "color: #6a9955;">// Store key locally before sending (in case of crash)
        pendingKeysStore.save(idempotencyKey, amount);

        HttpHeaders headers = new HttpHeaders();
        headers.set("Idempotency-Key", idempotencyKey);
        HttpEntity<PaymentRequest> entity = new HttpEntity<>(
            new PaymentRequest(amount), headers);

        try {
            PaymentResult result = restTemplate.postForObject(
                "/payments", entity, PaymentResult.class);
            pendingKeysStore.remove(idempotencyKey);
            return result;
        } catch (Exception e) {
            "color: #6a9955;">// Key remains in store; on restart we can check status
            throw e;
        }
    }

    private String generateIdempotencyKey() {
        return UUID.randomUUID().toString() + "-" + System.currentTimeMillis();
    }
}
```

> ***Key Takeaway:*** *Never retry non-idempotent operations without an idempotency key. The key must be checked atomically before processing.*

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

When you scale from one instance to multiple, a simple @Scheduled annotation runs on every instance. You need distributed locking to ensure only one instance executes the job at a time.

Root Causes:

- @Scheduled runs on every pod instance without coordination
- Lack of distributed locking mechanism
- Job execution time exceeding scheduling interval
- Failed job not releasing lock properly
- Clock skew between instances

Solutions:

- Use ShedLock or Quartz with JDBC job store for distributed scheduling
- Implement distributed locking with Redis/ZooKeeper
- Use a dedicated job scheduler service (Kubernetes CronJob)
- Ensure locks have TTL to prevent stuck locks
- Use leader election pattern for singleton jobs

Code Example:

```
"color: #6a9955;">// Anti-pattern: Simple @Scheduled runs on every instance
@Component
public class ReportGenerator {

    @Scheduled(cron = "0 0 2 * * ?") "color: #6a9955;">// Runs on EVERY pod!
    public void generateDailyReport() {
        "color: #6a9955;">// With 5 pods, this runs 5 times!
        reportService.generateDailyReport();
    }
}

"color: #6a9955;">// Solution 1: ShedLock for distributed scheduling
@Component
public class ReportGenerator {

    @Scheduled(cron = "0 0 2 * * ?")
    @SchedulerLock(name = "dailyReport", lockAtMostFor = "PT30M", lockAtLeastFor = "PT5M"
    public void generateDailyReport() {
```

```
        "color: #6a9955;">// Only one instance will execute this
        reportService.generateDailyReport();
    }
}

"color: #6a9955;">// Configuration
@Configuration
@EnableScheduling
@EnableSchedulerLock(defaultLockAtMostFor = "PT10M")
public class SchedulerConfig {

    @Bean
    public LockProvider lockProvider(DataSource dataSource) {
        return new JdbcTemplateLockProvider(
            JdbcTemplateLockProvider.Configuration.builder()
                .withJdbcTemplate(new JdbcTemplate(dataSource))
                .usingDbTime() "color: #6a9955;">// Use database time to avoid clock skew
                .build()
        );
    }
}

"color: #6a9955;">// Solution 2: Redis-based distributed lock
@Component
public class DistributedJobScheduler {

    @Autowired
    private StringRedisTemplate redisTemplate;

    public void executeWithLock(String jobName, Runnable job) {
        String lockKey = "job-lock:" + jobName;
        String token = UUID.randomUUID().toString();

        "color: #6a9955;">// Try acquire lock with 5 min TTL
        Boolean acquired = redisTemplate.opsForValue()
            .setIfAbsent(lockKey, token, Duration.ofMinutes(5));

        if (Boolean.TRUE.equals(acquired)) {
            try {
                job.run();
            } finally {
                "color: #6a9955;">// Extend lock during execution
                redisTemplate.expire(lockKey, Duration.ofMinutes(5));
                "color: #6a9955;">// Release lock
                String current = redisTemplate.opsForValue().get(lockKey);
                if (token.equals(current)) {
                    redisTemplate.delete(lockKey);
                }
            }
        } else {
            log.info("Job {} is already running on another instance", jobName);
        }
    }
}

"color: #6a9955;">// Solution 3: Quartz with JDBC job store
@Configuration
public class QuartzConfig {
```

```
    @Bean
    public SchedulerFactoryBean schedulerFactoryBean(DataSource dataSource) {
        SchedulerFactoryBean factory = new SchedulerFactoryBean();
        factory.setDataSource(dataSource);
        factory.setQuartzProperties(quartzProperties());
        factory.setOverwriteExistingJobs(true);
        factory.setAutoStartup(true);
        return factory;
    }

    private Properties quartzProperties() {
        Properties props = new Properties();
        props.put("org.quartz.jobStore.class", "org.quartz.impl.jdbcjobstore.JobStoreTX")
        props.put("org.quartz.jobStore.driverDelegateClass",
                  "org.quartz.impl.jdbcjobstore.StdJDBCDelegate");
        props.put("org.quartz.jobStore.isClustered", "true");
        props.put("org.quartz.jobStore.clusterCheckinInterval", "20000");
        props.put("org.quartz.scheduler.instanceId", "AUTO");
        return props;
    }
}
```

> ***Key Takeaway:*** *In a scaled environment, always use distributed scheduling. @Scheduled alone only works reliably on a single instance.*

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

Without proper isolation, a slow or failing downstream service can cascade and exhaust resources in the calling service. Thread pools fill up, connections hang, and eventually the entire application becomes unresponsive.

Root Causes:

- Synchronous calls to slow services without timeouts
- Shared thread pools for all downstream calls
- No circuit breaker to fail fast
- Cascading failure due to resource exhaustion
- Lack of bulkhead pattern implementation

Solutions:

- Implement circuit breakers with Resilience4j or Spring Cloud Circuit Breaker
- Use bulkhead pattern to isolate thread pools per service
- Set aggressive timeouts for downstream calls
- Use async/reactive patterns to avoid thread blocking
- Implement graceful degradation with fallbacks

Code Example:

```
"color: #6a9955;">// Anti-pattern: Synchronous call without protection
@Service
public class OrderService {

    public Order createOrder(OrderRequest request) {
        "color: #6a9955;">// If payment service is slow, this thread hangs!
        PaymentResult payment = paymentClient.process(request.getPayment());

        "color: #6a9955;">// If inventory service is down, this also hangs!
        inventoryClient.reserve(request.getItems());

        "color: #6a9955;">// If shipping service times out, another hanging thread!
        ShippingQuote shipping = shippingClient.getQuote(request.getAddress());

        return buildOrder(payment, shipping);
    }
}

"color: #6a9955;">// Solution: Circuit Breaker + Bulkhead + Timeout
```

```java
@Service
public class OrderService {

    private final CircuitBreaker paymentCircuitBreaker;
    private final CircuitBreaker inventoryCircuitBreaker;
    private final ThreadPoolBulkhead paymentBulkhead;

    public OrderService() {
        this.paymentCircuitBreaker = CircuitBreaker.ofDefaults("payment");
        this.inventoryCircuitBreaker = CircuitBreaker.ofDefaults("inventory");

        this.paymentBulkhead = ThreadPoolBulkhead.of("payment",
            ThreadPoolBulkheadConfig.custom()
                .maxThreadPoolSize(10)
                .coreThreadPoolSize(5)
                .queueCapacity(20)
                .build());
    }

    public Order createOrder(OrderRequest request) {
        "color: #6a9955;">// Each service has its own circuit breaker and thread pool
        CompletableFuture<PaymentResult> paymentFuture =
            paymentBulkhead.executeSupplier(() ->
                paymentCircuitBreaker.executeSupplier(() ->
                    paymentClient.processWithTimeout(request.getPayment(), Duration.ofSec
                )
            );

        CompletableFuture<InventoryResult> inventoryFuture =
            CompletableFuture.supplyAsync(() ->
                inventoryCircuitBreaker.executeSupplier(() ->
                    inventoryClient.reserve(request.getItems())
                ), inventoryExecutor
            );

        "color: #6a9955;">// Wait with overall timeout
        try {
            PaymentResult payment = paymentFuture.get(3, TimeUnit.SECONDS);
            InventoryResult inventory = inventoryFuture.get(3, TimeUnit.SECONDS);

            return buildOrder(payment, inventory);
        } catch (TimeoutException e) {
            "color: #6a9955;">// Return gracefully degraded response
            return createOrderWithAsyncPayment(request);
        }
    }

    "color: #6a9955;">// Fallback when payment service is down
    private Order createOrderWithAsyncPayment(OrderRequest request) {
        Order order = new Order();
        order.setStatus(OrderStatus.PENDING_PAYMENT);
        order.setItems(request.getItems());
        "color: #6a9955;">// Queue payment for retry later
        paymentRetryQueue.enqueue(order);
        return order;
    }
}
```

```
"color: #6a9955;">// Spring Boot Resilience4j configuration
@Configuration
public class ResilienceConfig {

    @Bean
    public Customizer<Resilience4JCircuitBreakerFactory> circuitBreakerCustomizer() {
        return factory -> factory.configure(builder -> builder
            .circuitBreakerConfig(CircuitBreakerConfig.custom()
                .failureRateThreshold(50)
                .waitDurationInOpenState(Duration.ofSeconds(30))
                .permittedNumberOfCallsInHalfOpenState(5)
                .slidingWindowSize(100)
                .build())
            .timeLimiterConfig(TimeLimiterConfig.custom()
                .timeoutDuration(Duration.ofSeconds(3))
                .build()),
            "payment", "inventory", "shipping");
    }

    @Bean
    public Customizer<ThreadPoolBulkheadRegistry> bulkheadCustomizer() {
        return registry -> registry.addConfiguration("default",
            ThreadPoolBulkheadConfig.custom()
                .maxThreadPoolSize(20)
                .coreThreadPoolSize(10)
                .queueCapacity(50)
                .build());
    }
}

"color: #6a9955;">// Annotation-based approach (Spring Cloud Circuit Breaker)
@Service
public class PaymentServiceClient {

    @CircuitBreaker(name = "payment", fallbackMethod = "fallback")
    @Bulkhead(name = "payment", type = Bulkhead.Type.THREADPOOL)
    @TimeLimiter(name = "payment")
    public PaymentResult process(PaymentRequest request) {
        return restTemplate.postForObject("/payments", request, PaymentResult.class);
    }

    public PaymentResult fallback(PaymentRequest request, Exception ex) {
        return PaymentResult.pending(request.getId());
    }
}
```

> ***Key Takeaway:*** *Isolate failures with circuit breakers and bulkheads. A problem in one service should never take down the entire platform.*

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

Random 500s with healthy infrastructure usually point to application-level issues: race conditions, memory leaks causing GC pauses, dependency version mismatches, or intermittent downstream failures that aren't handled gracefully.

Root Causes:

- Race conditions in concurrent code
- Memory pressure causing long GC pauses
- Unhandled exceptions in async code paths
- Intermittent downstream failures not caught
- Resource leaks (memory, file handles, connections)

Solutions:

- Add comprehensive error handling and logging
- Use structured logging with correlation IDs
- Monitor JVM GC metrics and heap usage
- Implement proper exception handling in async flows
- Use chaos engineering to find intermittent issues

Code Example:

```
"color: #6a9955;">// Anti-pattern: Unhandled exceptions in async code
@Service
public class OrderProcessor {

    public void processOrders(List<Order> orders) {
        orders.parallelStream().forEach(order -> {
            "color: #6a9955;">// Exception here kills the stream silently!
            paymentService.charge(order.getPayment());
            notificationService.send(order);
        });
    }
}

"color: #6a9955;">// Solution: Proper exception handling in async flows
@Service
public class OrderProcessor {

    private static final Logger log = LoggerFactory.getLogger(OrderProcessor.class);
```

```
    public ProcessingResult processOrders(List<Order> orders) {
        List<CompletableFuture<OrderResult>> futures = orders.stream()
            .map(order -> CompletableFuture.supplyAsync(() -> processOrder(order))
                .exceptionally(ex -> {
                    log.error("Failed to process order {}", order.getId(), ex);
                    return OrderResult.failed(order.getId(), ex.getMessage());
                }))
            .toList();

        "color: #6a9955;">// Collect results - don&#039;t lose failures
        List<OrderResult> results = futures.stream()
            .map(CompletableFuture::join)
            .toList();

        long successCount = results.stream().filter(OrderResult::isSuccess).count();
        long failureCount = results.size() - successCount;

        return new ProcessingResult(successCount, failureCount, results);
    }

    private OrderResult processOrder(Order order) {
        try {
            PaymentResult payment = paymentService.charge(order.getPayment());
            notificationService.send(order);
            return OrderResult.success(order.getId(), payment.getTransactionId());
        } catch (PaymentException e) {
            log.error("Payment failed for order {}", order.getId(), e);
            "color: #6a9955;">// Don&#039;t rethrow - handle gracefully
            return OrderResult.failed(order.getId(), "PAYMENT_FAILED: " + e.getMessage())
        }
    }
}

"color: #6a9955;">// Global exception handler with correlation IDs
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleException(
            Exception ex, HttpServletRequest request) {

        String correlationId = MDC.get("correlationId");

        ErrorResponse error = ErrorResponse.builder()
            .timestamp(Instant.now())
            .status(HttpStatus.INTERNAL_SERVER_ERROR.value())
            .error("Internal Server Error")
            .message("An unexpected error occurred")
            .path(request.getRequestURI())
            .correlationId(correlationId)
            .build();

        "color: #6a9955;">// Log full stack trace internally
        log.error("Unhandled exception [correlationId={}] at {}",
                  correlationId, request.getRequestURI(), ex);

        "color: #6a9955;">// Return safe response to client
```

```
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                .body(error);
        }

        @ExceptionHandler(ConstraintViolationException.class)
        public ResponseEntity<ErrorResponse> handleValidation(
                ConstraintViolationException ex) {

            List<String> violations = ex.getConstraintViolations().stream()
                .map(v -> v.getPropertyPath() + ": " + v.getMessage())
                .toList();

            return ResponseEntity.badRequest()
                .body(ErrorResponse.builder()
                    .status(400)
                    .error("Validation Failed")
                    .details(violations)
                    .build());
        }
    }

    "color: #6a9955;">// JVM monitoring for GC issues
    @Component
    public class JvmHealthIndicator implements HealthIndicator {

        @Override
        public Health health() {
            MemoryMXBean memoryMXBean = ManagementFactory.getMemoryMXBean();
            MemoryUsage heapUsage = memoryMXBean.getHeapMemoryUsage();

            long used = heapUsage.getUsed();
            long max = heapUsage.getMax();
            double usagePercent = (double) used / max * 100;

            "color: #6a9955;">// Get GC info
            List<GarbageCollectorMXBean> gcBeans =
                ManagementFactory.getGarbageCollectorMXBeans();
            long gcTime = gcBeans.stream()
                .mapToLong(GarbageCollectorMXBean::getCollectionTime)
                .sum();

            Health.Builder builder = Health.up();
            builder.withDetail("heap.usage.percent", String.format("%.2f", usagePercent));
            builder.withDetail("gc.total.time.ms", gcTime);

            if (usagePercent > 85) {
                builder.down().withDetail("reason", "High heap usage");
            }

            return builder.build();
        }
    }
```

> ***Key Takeaway:*** *Random 500s need structured logging with correlation IDs to trace across services. Don't just check infrastructure - instrument your*

> *application deeply.*

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

Liveness/readiness probes often only check if the application process is running, not if it's actually serving requests correctly. The app might be up but unable to connect to the database, process messages, or serve actual traffic.

Root Causes:

- Health checks only verify process is alive, not functional
- Database connections failing but not checked in health endpoint
- Thread pool exhaustion not reflected in health status
- Dependency services down but health check doesn't verify them
- Readiness probe not configured to remove pod from service

Solutions:

- Implement deep health checks that verify database, cache, and key dependencies
- Separate liveness (is process alive) from readiness (can serve traffic)
- Add custom health indicators for critical dependencies
- Configure readiness probes to fail when dependencies are unhealthy
- Implement startup probes for slow-starting containers

Code Example:

```
"color: #6a9955;">// Anti-pattern: Superficial health check
@RestController
public class HealthController {

    @GetMapping("/health")
    public String health() {
        return "OK"; "color: #6a9955;">// Always returns OK even if DB is down!
    }
}

"color: #6a9955;">// Solution: Comprehensive health checks with Spring Boot Actuator
@Component
public class DatabaseHealthIndicator implements HealthIndicator {

    @Autowired
    private DataSource dataSource;
```

```
    @Override
    public Health health() {
        try (Connection conn = dataSource.getConnection()) {
            if (conn.isValid(5)) {
                return Health.up()
                    .withDetail("database", "PostgreSQL")
                    .withDetail("validationQuery", "SELECT 1")
                    .build();
            }
        } catch (SQLException e) {
            return Health.down()
                .withDetail("error", e.getMessage())
                .build();
        }
        return Health.down().build();
    }
}

@Component
public class KafkaHealthIndicator implements HealthIndicator {

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    @Override
    public Health health() {
        try {
            "color: #6a9955;">// Check if we can reach the cluster
            Map<String, Object> metrics = kafkaTemplate.metrics();
            if (metrics != null && !metrics.isEmpty()) {
                return Health.up()
                    .withDetail("kafka", "Connected")
                    .build();
            }
        } catch (Exception e) {
            return Health.down()
                .withDetail("error", e.getMessage())
                .build();
        }
        return Health.down().build();
    }
}

@Component
public class ExternalServiceHealthIndicator implements HealthIndicator {

    @Autowired
    private PaymentClient paymentClient;

    @Override
    public Health health() {
        try {
            "color: #6a9955;">// Lightweight health check to external service
            HealthResponse response = paymentClient.health();
            if (response.isHealthy()) {
                return Health.up()
                    .withDetail("paymentService", "Healthy")
```

```
                    .withDetail("version", response.getVersion())
                    .build();
            }
        } catch (Exception e) {
            return Health.down()
                .withDetail("paymentService", "Unreachable")
                .withDetail("error", e.getMessage())
                .build();
        }
        return Health.down().build();
    }
}

"color: #6a9955;">// Custom composite health check
@Component
public class ApplicationReadinessIndicator implements HealthIndicator {

    @Autowired
    private List<HealthIndicator> dependencyIndicators;

    @Override
    public Health health() {
        Map<String, Health> results = new HashMap<>();
        boolean allUp = true;

        for (HealthIndicator indicator : dependencyIndicators) {
            Health health = indicator.health();
            results.put(indicator.getClass().getSimpleName(), health);
            if (health.getStatus() != Status.UP) {
                allUp = false;
            }
        }

        Health.Builder builder = allUp ? Health.up() : Health.down();
        results.forEach((name, health) -> builder.withDetail(name, health));

        return builder.build();
    }
}

"color: #6a9955;">// Kubernetes probe configuration
"color: #6a9955;">/*
  livenessProbe:
    httpGet:
      path: /actuator/health/liveness
      port: 8080
    initialDelaySeconds: 60
    periodSeconds: 10
    failureThreshold: 3

  readinessProbe:
    httpGet:
      path: /actuator/health/readiness
      port: 8080
    initialDelaySeconds: 30
    periodSeconds: 5
    failureThreshold: 3
```

```
  startupProbe:
    httpGet:
      path: /actuator/health/liveness
      port: 8080
    initialDelaySeconds: 10
    periodSeconds: 5
    failureThreshold: 30  "color: #6a9955;">// Allow 2.5 min for slow startup
*/
```

> ***Key Takeaway:*** *Health checks must verify functionality, not just process liveness. Separate liveness, readiness, and startup probes for proper Kubernetes orchestration.*

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

Cache staleness occurs when the cache and database get out of sync. This happens due to missing invalidation, race conditions between writes and reads, or TTL-based expiration that doesn't account for data changes.

Root Causes:

- Cache writes without invalidation on data updates
- Race conditions between cache read and write operations
- TTL expiration longer than data change frequency
- Multiple cache layers with inconsistent invalidation
- Database updates outside the application bypassing cache

Solutions:

- Implement cache-aside with proper invalidation
- Use write-through or write-behind patterns
- Implement optimistic locking with versioning
- Use cache invalidation on all write paths
- Consider using Redis with pub/sub for cross-instance invalidation

Code Example:

```
"color: #6a9955;">// Anti-pattern: Cache without invalidation
@Service
public class ProductService {

    @Autowired
    private CacheManager cacheManager;

    @Cacheable("products")
    public Product getProduct(Long id) {
        return productRepository.findById(id).orElseThrow();
    }

    "color: #6a9955;">// Missing @CacheEvict - cache becomes stale!
    public Product updateProduct(Long id, ProductUpdate update) {
        Product product = productRepository.findById(id).orElseThrow();
        product.setName(update.getName());
        product.setPrice(update.getPrice());
        return productRepository.save(product);
```

```
    }
}

"color: #6a9955;">// Solution: Proper cache invalidation
@Service
public class ProductService {

    @Autowired
    private ProductRepository productRepository;

    @Autowired
    private CacheManager cacheManager;

    @Autowired
    private RedisTemplate<String, Object> redisTemplate;

    @Cacheable(value = "products", key = "#id")
    public Product getProduct(Long id) {
        return productRepository.findById(id).orElseThrow();
    }

    @CacheEvict(value = "products", key = "#id")
    @CachePut(value = "products", key = "#id")
    public Product updateProduct(Long id, ProductUpdate update) {
        Product product = productRepository.findById(id).orElseThrow();
        product.setName(update.getName());
        product.setPrice(update.getPrice());
        product.setVersion(product.getVersion() + 1);
        return productRepository.save(product);
    }

    @CacheEvict(value = "products", key = "#id")
    public void deleteProduct(Long id) {
        productRepository.deleteById(id);
    }

    "color: #6a9955;">// Distributed cache invalidation across instances
    public void updateProductWithBroadcast(Long id, ProductUpdate update) {
        Product product = productRepository.findById(id).orElseThrow();
        product.setName(update.getName());
        product.setPrice(update.getPrice());
        productRepository.save(product);

        "color: #6a9955;">// Broadcast invalidation to all instances
        redisTemplate.convertAndSend("cache:invalidate",
            "products::" + id);
    }
}

"color: #6a9955;">// Cache event listener for distributed invalidation
@Component
public class CacheInvalidationListener {

    @Autowired
    private CacheManager cacheManager;

    @EventListener
    public void handleCacheInvalidation(CacheInvalidateEvent event) {
```

```
        Cache cache = cacheManager.getCache(event.getCacheName());
        if (cache != null) {
            cache.evict(event.getKey());
        }
    }
}

"color: #6a9955;">// Redis pub/sub configuration
@Configuration
public class CacheConfig {

    @Bean
    public RedisMessageListenerContainer redisContainer(
            RedisConnectionFactory connectionFactory,
            CacheInvalidationListener listener) {

        RedisMessageListenerContainer container =
            new RedisMessageListenerContainer();
        container.setConnectionFactory(connectionFactory);

        "color: #6a9955;">// Subscribe to invalidation channel
        container.addMessageListener(
            (message, pattern) -> {
                String key = new String(message.getBody());
                "color: #6a9955;">// Parse cache name and key
                String[] parts = key.split("::");
                listener.handleCacheInvalidation(
                    new CacheInvalidateEvent(parts[0], parts[1]));
            },
            new PatternTopic("cache:invalidate")
        );

        return container;
    }
}

"color: #6a9955;">// Cache-aside with versioning to prevent stale reads
@Service
public class OrderService {

    public Order getOrder(Long orderId) {
        String cacheKey = "order:" + orderId;
        CachedOrder cached = (CachedOrder) redisTemplate.opsForValue().get(cacheKey);

        Order order = orderRepository.findById(orderId).orElseThrow();

        if (cached != null && cached.getVersion() == order.getVersion()) {
            return cached.toOrder(); "color: #6a9955;">// Cache hit with version match
        }

        "color: #6a9955;">// Cache miss or stale version - update cache
        CachedOrder toCache = CachedOrder.from(order);
        redisTemplate.opsForValue().set(cacheKey, toCache, Duration.ofMinutes(10));
        return order;
    }
}
```

> ***Key Takeaway:*** *Caching without invalidation is a cache bug waiting to happen. Every write path must invalidate or update the cache. Consider distributed invalidation for multi-instance deployments.*

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

Without distributed tracing and correlation IDs, logs from different services are just isolated streams. You can't follow a single request's journey across the system, making root cause analysis nearly impossible.

Root Causes:

- No correlation ID propagated between services
- Different log formats across services
- Logs scattered across multiple systems without aggregation
- Missing context in log messages (user ID, request params)
- No distributed tracing (spans, traces) implemented

Solutions:

- Generate and propagate correlation IDs across all service calls
- Use structured logging (JSON) with consistent schema
- Centralize logs with ELK stack or similar
- Implement distributed tracing with OpenTelemetry/Jaeger
- Add business context to every log entry

Code Example:

```
"color: #6a9955;">// Anti-pattern: Unstructured, context-less logging
@Service
public class OrderService {
    private static final Logger log = LoggerFactory.getLogger(OrderService.class);

    public Order createOrder(OrderRequest request) {
        log.info("Creating order"); "color: #6a9955;">// No context!

        PaymentResult payment = paymentClient.process(request.getPayment());
        log.info("Payment processed"); "color: #6a9955;">// Which order? Which payment?

        return new Order();
    }
}

"color: #6a9955;">// Solution: Structured logging with correlation IDs
@Component
public class CorrelationIdFilter extends OncePerRequestFilter {
```

```
    public static final String CORRELATION_ID_HEADER = "X-Correlation-ID";
    public static final String CORRELATION_ID_MDC = "correlationId";

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain chain) throws ServletException, IOExcepti

        String correlationId = request.getHeader(CORRELATION_ID_HEADER);
        if (correlationId == null) {
            correlationId = UUID.randomUUID().toString();
        }

        MDC.put(CORRELATION_ID_MDC, correlationId);
        response.setHeader(CORRELATION_ID_HEADER, correlationId);

        try {
            chain.doFilter(request, response);
        } finally {
            MDC.clear();
        }
    }
}

"color: #6a9955;">// Feign client interceptor to propagate correlation ID
@Component
public class CorrelationIdInterceptor implements RequestInterceptor {

    @Override
    public void apply(RequestTemplate template) {
        String correlationId = MDC.get(CorrelationIdFilter.CORRELATION_ID_MDC);
        if (correlationId != null) {
            template.header(CorrelationIdFilter.CORRELATION_ID_HEADER, correlationId);
        }
    }
}

"color: #6a9955;">// Structured logging with context
@Service
public class OrderService {

    private static final Logger log = LoggerFactory.getLogger(OrderService.class);

    public Order createOrder(OrderRequest request) {
        String correlationId = MDC.get("correlationId");
        String userId = SecurityContextHolder.getContext().getAuthentication().getName();

        "color: #6a9955;">// Structured log with all context
        log.info("Order creation started",
            StructuredArguments.kv("correlationId", correlationId),
            StructuredArguments.kv("userId", userId),
            StructuredArguments.kv("orderRequest", request),
            StructuredArguments.kv("itemsCount", request.getItems().size()));

        try {
            PaymentResult payment = paymentClient.process(request.getPayment());
```

```
            log.info("Payment completed",
                StructuredArguments.kv("correlationId", correlationId),
                StructuredArguments.kv("transactionId", payment.getTransactionId()),
                StructuredArguments.kv("amount", payment.getAmount()));

            Order order = buildOrder(request, payment);
            Order saved = orderRepository.save(order);

            log.info("Order created successfully",
                StructuredArguments.kv("correlationId", correlationId),
                StructuredArguments.kv("orderId", saved.getId()),
                StructuredArguments.kv("status", saved.getStatus()));

            return saved;

        } catch (PaymentException e) {
            log.error("Payment failed for order",
                StructuredArguments.kv("correlationId", correlationId),
                StructuredArguments.kv("error", e.getMessage()),
                StructuredArguments.kv("paymentMethod", request.getPayment().getMethod()),
                e);
            throw e;
        }
    }
}

"color: #6a9955;">// OpenTelemetry tracing configuration
@Configuration
public class TracingConfig {

    @Bean
    public OpenTelemetry openTelemetry() {
        Resource resource = Resource.getDefault()
            .merge(Resource.create(Attributes.of(
                ResourceAttributes.SERVICE_NAME, "order-service",
                ResourceAttributes.SERVICE_VERSION, "1.0.0")));

        "color: #6a9955;">// Jaeger exporter
        JaegerGrpcSpanExporter jaegerExporter = JaegerGrpcSpanExporter.builder()
            .setEndpoint("http:"color: #6a9955;">//jaeger:14250")
            .setTimeout(30, TimeUnit.SECONDS)
            .build();

        SdkTracerProvider tracerProvider = SdkTracerProvider.builder()
            .addSpanProcessor(BatchSpanProcessor.builder(jaegerExporter).build())
            .setResource(resource)
            .build();

        return OpenTelemetrySdk.builder()
            .setTracerProvider(tracerProvider)
            .buildAndRegisterGlobal();
    }

    @Bean
    public Tracer tracer(OpenTelemetry openTelemetry) {
        return openTelemetry.getTracer("order-service");
    }
}
```

```
"color: #6a9955;">// Tracing in service methods
@Service
public class TracedOrderService {

    @Autowired
    private Tracer tracer;

    public Order createOrder(OrderRequest request) {
        Span span = tracer.spanBuilder("createOrder")
            .setAttribute("user.id", getCurrentUserId())
            .setAttribute("order.itemCount", request.getItems().size())
            .startSpan();

        try (Scope scope = span.makeCurrent()) {
            Span paymentSpan = tracer.spanBuilder("processPayment")
                .setParent(Context.current().with(span))
                .startSpan();

            PaymentResult payment;
            try (Scope paymentScope = paymentSpan.makeCurrent()) {
                payment = paymentClient.process(request.getPayment());
                paymentSpan.setAttribute("payment.transactionId",
                    payment.getTransactionId());
            } finally {
                paymentSpan.end();
            }

            return buildAndSaveOrder(request, payment);

        } catch (Exception e) {
            span.recordException(e);
            span.setStatus(StatusCode.ERROR);
            throw e;
        } finally {
            span.end();
        }
    }
}
```

> ***Key Takeaway:*** *Logs without correlation IDs are just noise. Implement distributed tracing with OpenTelemetry to follow requests across services and understand latency at each hop.*

12. JVM memory usage slowly increases after every deployment.

Gradual memory growth typically indicates a memory leak. Objects are being created and referenced but never released. Common causes include static collections growing unbounded, thread-local variables not cleaned up, or resources not being closed.

Root Causes:

- Static collections accumulating objects without cleanup
- ThreadLocal variables not removed after use
- Listeners/observers registered but never unregistered
- Large objects cached without TTL or size limits
- Classloader leaks from hot reload or dynamic class generation

Solutions:

- Use heap dumps and analysis tools (Eclipse MAT, VisualVM)
- Implement proper cleanup in finally blocks
- Use WeakReference for caches that should allow GC
- Set maximum size and TTL on all caches
- Remove ThreadLocal values in a finally block

Code Example:

```
"color: #6a9955;">// Anti-pattern: Unbounded static cache
public class UserCache {
    "color: #6a9955;">// This map grows forever - memory leak!
    private static final Map<Long, User> cache = new HashMap<>();

    public static void put(Long id, User user) {
        cache.put(id, user); "color: #6a9955;">// No eviction!
    }

    public static User get(Long id) {
        return cache.get(id);
    }
}

"color: #6a9955;">// Solution: Bounded cache with TTL
@Service
public class UserCache {
```

```
    private final Cache<Long, User> cache;

    public UserCache() {
        this.cache = Caffeine.newBuilder()
            .maximumSize(10_000)              "color: #6a9955;">// Max 10k entries
            .expireAfterWrite(Duration.ofMinutes(30)) "color: #6a9955;">// 30 min TTL
            .removalListener((key, value, cause) -> {
                log.debug("User {} removed from cache: {}", key, cause);
            })
            .recordStats() "color: #6a9955;">// Enable hit/miss stats
            .build();
    }

    public User get(Long id) {
        return cache.get(id, userRepository::findById);
    }

    public void invalidate(Long id) {
        cache.invalidate(id);
    }

    public CacheStats getStats() {
        return cache.stats();
    }
}

"color: #6a9955;">// Anti-pattern: ThreadLocal without cleanup
public class RequestContext {
    private static final ThreadLocal<Map<String, Object>> context =
        ThreadLocal.withInitial(HashMap::new);

    public static void set(String key, Object value) {
        context.get().put(key, value);
    }

    public static Object get(String key) {
        return context.get().get(key);
    }
    "color: #6a9955;">// Missing remove() - leaks when using thread pools!
}

"color: #6a9955;">// Solution: Proper ThreadLocal cleanup
@Component
public class RequestContextFilter extends OncePerRequestFilter {

    private static final ThreadLocal<Map<String, Object>> context =
        ThreadLocal.withInitial(HashMap::new);

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain chain) throws ServletException, IOExcepti
        try {
            context.get().put("correlationId",
                request.getHeader("X-Correlation-ID"));
            context.get().put("startTime", System.currentTimeMillis());
            chain.doFilter(request, response);
        } finally {
```

```
            "color: #6a9955;">// CRITICAL: Always clean up ThreadLocal
            context.remove();
        }
    }
}

"color: #6a9955;">// Memory leak detection with heap dumps
@Component
public class MemoryMonitor {

    @Scheduled(fixedRate = 3600000) "color: #6a9955;">// Every hour
    public void checkMemory() {
        Runtime runtime = Runtime.getRuntime();
        long used = runtime.totalMemory() - runtime.freeMemory();
        long max = runtime.maxMemory();
        double usagePercent = (double) used / max * 100;

        log.info("Memory usage: {}% ({}MB / {}MB)",
            String.format("%.2f", usagePercent),
            used / 1024 / 1024,
            max / 1024 / 1024);

        if (usagePercent > 80) {
            triggerHeapDump();
        }
    }

    private void triggerHeapDump() {
        try {
            String fileName = "heap-dump-" + System.currentTimeMillis() + ".hprof";
            HotSpotDiagnosticMXBean mxBean = ManagementFactory
                .getPlatformMXBean(HotSpotDiagnosticMXBean.class);
            mxBean.dumpHeap(fileName, true);
            log.warn("Heap dump triggered: {}", fileName);
        } catch (IOException e) {
            log.error("Failed to create heap dump", e);
        }
    }
}

"color: #6a9955;">// Proper resource cleanup with try-with-resources
public class FileProcessor {

    public void processLargeFile(String filePath) {
        "color: #6a9955;">// Old way - potential leak
        BufferedReader reader = null;
        try {
            reader = new BufferedReader(new FileReader(filePath));
            String line;
            while ((line = reader.readLine()) != null) {
                process(line);
            }
        } catch (IOException e) {
            throw new ProcessingException(e);
        } finally {
            if (reader != null) {
                try {
                    reader.close(); "color: #6a9955;">// Might throw, masking original ex
```

```
                } catch (IOException e) {
                    log.error("Failed to close reader", e);
                }
            }
        }
    }

    "color: #6a9955;">// Modern way - guaranteed cleanup
    public void processLargeFileSafe(String filePath) {
        try (BufferedReader reader = new BufferedReader(new FileReader(filePath));
             "color: #6a9955;">// Multiple resources, all closed automatically
             Connection conn = dataSource.getConnection();
             PreparedStatement stmt = conn.prepareStatement("INSERT ...")) {

            String line;
            while ((line = reader.readLine()) != null) {
                stmt.setString(1, process(line));
                stmt.addBatch();
            }
            stmt.executeBatch();

        } catch (IOException | SQLException e) {
            throw new ProcessingException(e);
        }
        "color: #6a9955;">// All resources closed automatically, even if exception occurs
    }
}
```

> ***Key Takeaway:*** *Memory leaks are almost always reference leaks. Use bounded caches, clean up ThreadLocals, and leverage try-with-resources. Profile with heap dumps when growth is observed.*

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

API gateways add layers of routing, authentication, rate limiting, SSL termination, and request transformation. Issues often stem from header manipulation, payload size limits, timeout configurations, or certificate problems that differ between environments.

Root Causes:

- Request/response size limits differ between environments
- SSL/TLS certificate issues in production
- Gateway timeout shorter than service timeout
- Missing or incorrect headers added by gateway
- CORS configuration different in production
- IP whitelist or firewall rules blocking traffic

Solutions:

- Mirror production gateway configuration in staging
- Test with exact same headers and SSL settings
- Configure gateway timeouts longer than service timeouts
- Use path-based routing tests in CI/CD
- Implement health checks that traverse the full gateway path

Code Example:

```
"color: #6a9955;">// Anti-pattern: Assuming direct access and gateway access are the same
@RestController
public class OrderController {

    @PostMapping("/api/v1/orders")
    public ResponseEntity<Order> createOrder(@RequestBody OrderRequest request) {
        "color: #6a9955;">// Works fine when testing directly against service
        "color: #6a9955;">// But fails behind gateway due to size limits or headers
        return ResponseEntity.ok(orderService.create(request));
    }
}

"color: #6a9955;">// Solution: Gateway-aware controller with proper headers
@RestController
@RequestMapping("/api/v1")
public class OrderController {
```

```
    private static final Logger log = LoggerFactory.getLogger(OrderController.class);

    @PostMapping("/orders")
    public ResponseEntity<Order> createOrder(
            @RequestBody @Valid OrderRequest request,
            @RequestHeader(value = "X-Forwarded-For", required = false) String forwardedF
            @RequestHeader(value = "X-Request-ID", required = false) String requestId,
            HttpServletRequest servletRequest) {

        "color: #6a9955;">// Log gateway headers for debugging
        log.debug("Client IP: {}, Request ID: {}, Content-Length: {}",
            forwardedFor != null ? forwardedFor : servletRequest.getRemoteAddr(),
            requestId,
            servletRequest.getContentLengthLong());

        "color: #6a9955;">// Check payload size (gateway might have different limits)
        if (servletRequest.getContentLengthLong() > 10 * 1024 * 1024) {
            return ResponseEntity.status(HttpStatus.PAYLOAD_TOO_LARGE)
                .body(null);
        }

        Order order = orderService.create(request);

        return ResponseEntity.ok()
            .header("X-Request-ID", requestId != null ? requestId : UUID.randomUUID().toS
            .body(order);
    }
}

"color: #6a9955;">// Gateway timeout configuration awareness
@Service
public class OrderService {

    @Value("${gateway.timeout.seconds:30}")
    private int gatewayTimeoutSeconds;

    @Value("${service.timeout.seconds:25}")
    private int serviceTimeoutSeconds;

    @PostConstruct
    public void validateTimeouts() {
        if (serviceTimeoutSeconds >= gatewayTimeoutSeconds) {
            throw new IllegalStateException(
                "Service timeout (" + serviceTimeoutSeconds +
                "s) must be less than gateway timeout (" +
                gatewayTimeoutSeconds + "s)");
        }
    }

    public Order create(OrderRequest request) {
        "color: #6a9955;">// Use timeout less than gateway&#039;s timeout
        return CompletableFuture.supplyAsync(() -> processOrder(request))
            .orTimeout(serviceTimeoutSeconds, TimeUnit.SECONDS)
            .join();
    }
}
```

```
"color: #6a9955;">// Spring Cloud Gateway configuration
@Configuration
public class GatewayConfig {

    @Bean
    public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route("order-service", r -> r
                .path("/api/v1/orders/**")
                .filters(f -> f
                    .stripPrefix(0)
                    .addRequestHeader("X-Gateway-Time",
                        Instant.now().toString())
                    .circuitBreaker(config -> config
                        .setName("orderService")
                        .setFallbackUri("forward:/fallback"))
                    .retry(retryConfig -> retryConfig
                        .setRetries(3)
                        .setBackoff(Duration.ofMillis(100),
                            Duration.ofMillis(1000), 2, true))
                    .requestSize(10 * 1024 * 1024L)) "color: #6a9955;">// 10MB limit
                .uri("lb:"color: #6a9955;">//order-service"))
            .build();
    }

    @Bean
    public GlobalFilter loggingFilter() {
        return (exchange, chain) -> {
            ServerHttpRequest request = exchange.getRequest();
            log.info("Gateway request: {} {} from {}",
                request.getMethod(),
                request.getURI(),
                request.getRemoteAddress());
            return chain.filter(exchange);
        };
    }
}

"color: #6a9955;">// Integration test that simulates gateway behavior
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@AutoConfigureMockMvc
public class GatewayIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void shouldHandleGatewayHeaders() throws Exception {
        mockMvc.perform(post("/api/v1/orders")
                .header("X-Forwarded-For", "10.0.0.1")
                .header("X-Request-ID", UUID.randomUUID().toString())
                .header("X-Real-IP", "192.168.1.1")
                .contentType(MediaType.APPLICATION_JSON)
                .content(orderJson()))
            .andExpect(status().isOk())
            .andExpect(header().exists("X-Request-ID"));
    }
```

```
    @Test
    void shouldRejectOversizedPayload() throws Exception {
        String largePayload = "a".repeat(15 * 1024 * 1024); "color: #6a9955;">// 15MB

        mockMvc.perform(post("/api/v1/orders")
                .contentType(MediaType.APPLICATION_JSON)
                .content(largePayload))
            .andExpect(status().isPayloadTooLarge());
    }
}
```

> ***Key Takeaway:*** *Never assume gateway behavior is transparent. Test with the same headers, SSL, timeouts, and payload limits that production uses.*

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

Thread exhaustion with low CPU indicates threads are blocked waiting for I/O (database, HTTP calls, file system) rather than doing computation. The threads are alive but idle, waiting for external resources.

Root Causes:

- Threads blocked on database queries without timeouts
- HTTP client calls without connection or read timeouts
- Synchronized blocks causing thread contention
- Blocking queue operations without timeouts
- File I/O operations on slow storage

Solutions:

- Use async/non-blocking I/O where possible (WebFlux, CompletableFuture)
- Set aggressive timeouts on all blocking operations
- Use separate thread pools for different operation types (bulkhead)
- Monitor thread states (BLOCKED, WAITING, TIMED_WAITING)
- Use connection pooling with proper timeout configuration

Code Example:

```
"color: #6a9955;">// Anti-pattern: Blocking calls consuming threads
@Service
public class ReportService {

    private final ExecutorService executor = Executors.newFixedThreadPool(10);

    public List<Report> generateReports(List<Long> reportIds) {
        return reportIds.stream()
            .map(id -> executor.submit(() -> {
                "color: #6a9955;">// Each thread blocks here for 5+ seconds!
                Data data = fetchFromSlowDatabase(id); "color: #6a9955;">// Blocking
                Chart chart = callExternalChartService(data); "color: #6a9955;">// Blocki
                PdfDocument pdf = renderPdf(chart); "color: #6a9955;">// Blocking I/O
                return new Report(pdf);
            }))
            .map(future -> {
                try {
                    return future.get(); "color: #6a9955;">// Also blocks!
                } catch (Exception e) {
```

```
                    throw new RuntimeException(e);
                }
            })
            .toList();
    }
}

"color: #6a9955;">// Solution: Reactive programming with WebFlux
@Service
public class ReactiveReportService {

    @Autowired
    private DatabaseClient databaseClient;

    @Autowired
    private WebClient webClient;

    public Mono<List<Report>> generateReports(List<Long> reportIds) {
        return Flux.fromIterable(reportIds)
            .flatMap(this::generateReport, 10) "color: #6a9955;">// Concurrency = 10
            .collectList();
    }

    private Mono<Report> generateReport(Long reportId) {
        return fetchData(reportId)
            .flatMap(this::generateChart)
            .flatMap(this::renderPdf)
            .map(Report::new)
            .timeout(Duration.ofSeconds(10)); "color: #6a9955;">// Fail fast
    }

    private Mono<Data> fetchData(Long reportId) {
        return databaseClient.sql("SELECT * FROM reports WHERE id = :id")
            .bind("id", reportId)
            .map((row, metadata) -> mapData(row))
            .one()
            .timeout(Duration.ofSeconds(5));
    }

    private Mono<Chart> generateChart(Data data) {
        return webClient.post()
            .uri("/charts/generate")
            .bodyValue(data)
            .retrieve()
            .bodyToMono(Chart.class)
            .timeout(Duration.ofSeconds(5));
    }

    private Mono<PdfDocument> renderPdf(Chart chart) {
        return Mono.fromCallable(() -> pdfRenderer.render(chart))
            .subscribeOn(Schedulers.boundedElastic()) "color: #6a9955;">// Offload blocki
            .timeout(Duration.ofSeconds(10));
    }
}

"color: #6a9955;">// Thread pool monitoring
@Component
public class ThreadPoolMonitor {
```

```
    @Scheduled(fixedRate = 30000)
    public void logThreadStats() {
        ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();

        long[] threadIds = threadMXBean.getAllThreadIds();
        ThreadInfo[] threadInfos = threadMXBean.getThreadInfo(threadIds);

        Map<String, Long> stateCounts = Arrays.stream(threadInfos)
            .filter(Objects::nonNull)
            .collect(Collectors.groupingBy(
                ThreadInfo::getThreadState,
                Collectors.counting()));

        log.info("Thread states: {}", stateCounts);

        "color: #6a9955;">// Alert if too many BLOCKED threads
        long blockedCount = stateCounts.getOrDefault(Thread.State.BLOCKED, 0L);
        if (blockedCount > 10) {
            log.warn("High number of BLOCKED threads: {}", blockedCount);
            "color: #6a9955;">// Capture thread dump for analysis
            captureThreadDump();
        }
    }

    private void captureThreadDump() {
        ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();
        long[] threadIds = threadMXBean.getAllThreadIds();
        ThreadInfo[] threadInfos = threadMXBean.getThreadInfo(threadIds, true, true);

        for (ThreadInfo info : threadInfos) {
            if (info != null && info.getThreadState() == Thread.State.BLOCKED) {
                log.warn("BLOCKED thread: {} - waiting on: {}",
                    info.getThreadName(),
                    info.getLockName());
            }
        }
    }
}

"color: #6a9955;">// Properly configured HTTP client with timeouts
@Configuration
public class HttpClientConfig {

    @Bean
    public WebClient webClient() {
        HttpClient httpClient = HttpClient.create()
            .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 5000)
            .responseTimeout(Duration.ofSeconds(10))
            .doOnConnected(conn -> conn
                .addHandlerLast(new ReadTimeoutHandler(10))
                .addHandlerLast(new WriteTimeoutHandler(10)));

        return WebClient.builder()
            .clientConnector(new ReactorClientHttpConnector(httpClient))
            .filter(logRequest())
            .build();
    }
```

```
    private ExchangeFilterFunction logRequest() {
        return ExchangeFilterFunction.ofRequestProcessor(request -> {
            log.debug("Request: {} {}", request.method(), request.url());
            return Mono.just(request);
        });
    }
}

"color: #6a9955;">// Virtual threads (Java 21+) for I/O bound workloads
@Service
public class VirtualThreadService {

    private final ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();

    public List<Report> generateReports(List<Long> reportIds) {
        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
            List<StructuredTaskScope.Subtask<Report>> subtasks = reportIds.stream()
                .map(id -> scope.fork(() -> generateReport(id)))
                .toList();

            scope.join();
            scope.throwIfFailed();

            return subtasks.stream()
                .map(StructuredTaskScope.Subtask::get)
                .toList();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private Report generateReport(Long reportId) {
        "color: #6a9955;">// Virtual threads are cheap - blocking here doesn&#039;t exhau
        Data data = fetchFromSlowDatabase(reportId);
        Chart chart = callExternalChartService(data);
        return new Report(chart);
    }
}
```

> ***Key Takeaway:*** *Low CPU with thread exhaustion means I/O blocking. Use reactive programming, virtual threads, or properly sized dedicated thread pools for I/O operations.*

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

Circuit breakers only protect the specific call they're configured for. Cascading failures happen when downstream failures cause resource exhaustion (threads, memory, connections) in the calling service before the circuit breaker can trip.

Root Causes:

- Circuit breaker timeout longer than caller's timeout
- Thread pools shared between different downstream calls
- Circuit breaker not configured for all failure paths
- Slow accumulation of requests before breaker trips
- Fallback methods that themselves call failing services

Solutions:

- Use bulkheads to isolate thread pools per dependency
- Set circuit breaker timeout shorter than upstream timeout
- Configure circuit breaker with low failure threshold
- Ensure fallback methods are simple and don't call external services
- Implement rate limiting at the edge

Code Example:

```
"color: #6a9955;">// Anti-pattern: Shared resources defeat circuit breaker
@Service
public class OrderService {

    private final RestTemplate restTemplate = new RestTemplate();

    @CircuitBreaker(name = "payment")
    public PaymentResult processPayment(PaymentRequest request) {
        "color: #6a9955;">// Uses shared connection pool - can still exhaust!
        return restTemplate.postForObject("/payments", request, PaymentResult.class);
    }

    @CircuitBreaker(name = "inventory")
    public InventoryResult reserveInventory(List<Item> items) {
        "color: #6a9955;">// Same shared pool - payment failures affect inventory calls!
        return restTemplate.postForObject("/inventory/reserve", items, InventoryResult.cl
    }
}
```

```
"color: #6a9955;">// Solution: Bulkheads + Circuit Breakers + Timeouts
@Configuration
public class ResilienceConfig {

    "color: #6a9955;">// Separate thread pool for each dependency
    @Bean("paymentExecutor")
    public ThreadPoolExecutor paymentExecutor() {
        return new ThreadPoolExecutor(5, 10, 60, TimeUnit.SECONDS,
            new ArrayBlockingQueue<>(100),
            new ThreadFactoryBuilder().setNameFormat("payment-%d").build(),
            new ThreadPoolExecutor.CallerRunsPolicy());
    }

    @Bean("inventoryExecutor")
    public ThreadPoolExecutor inventoryExecutor() {
        return new ThreadPoolExecutor(5, 10, 60, TimeUnit.SECONDS,
            new ArrayBlockingQueue<>(100),
            new ThreadFactoryBuilder().setNameFormat("inventory-%d").build(),
            new ThreadPoolExecutor.CallerRunsPolicy());
    }

    @Bean
    public Customizer<Resilience4JCircuitBreakerFactory> circuitBreakerFactoryCustomizer(
        return factory -> factory.configure(builder -> builder
            .circuitBreakerConfig(CircuitBreakerConfig.custom()
                .failureRateThreshold(50)        "color: #6a9955;">// Open after 50% fail
                .slowCallRateThreshold(80)       "color: #6a9955;">// Count slow calls as
                .slowCallDurationThreshold(Duration.ofSeconds(2))
                .waitDurationInOpenState(Duration.ofSeconds(30))
                .permittedNumberOfCallsInHalfOpenState(5)
                .slidingWindowSize(100)
                .build())
            .timeLimiterConfig(TimeLimiterConfig.custom()
                .timeoutDuration(Duration.ofSeconds(3)) "color: #6a9955;">// Must be < ca
                .build()),
            "payment", "inventory", "shipping");
    }
}

@Service
public class ResilientOrderService {

    @Autowired
    @Qualifier("paymentExecutor")
    private ExecutorService paymentExecutor;

    @Autowired
    @Qualifier("inventoryExecutor")
    private ExecutorService inventoryExecutor;

    @CircuitBreaker(name = "payment", fallbackMethod = "paymentFallback")
    @Bulkhead(name = "payment", type = Bulkhead.Type.THREADPOOL)
    @TimeLimiter(name = "payment")
    public CompletableFuture<PaymentResult> processPayment(PaymentRequest request) {
        return CompletableFuture.supplyAsync(() -> {
            return paymentClient.charge(request);
        }, paymentExecutor);
```

```
    }

    private CompletableFuture<PaymentResult> paymentFallback(PaymentRequest request, Exce
        "color: #6a9955;">// Simple fallback - NO external calls!
        return CompletableFuture.completedFuture(
            PaymentResult.pending(request.getId()));
    }

    @CircuitBreaker(name = "inventory", fallbackMethod = "inventoryFallback")
    @Bulkhead(name = "inventory", type = Bulkhead.Type.THREADPOOL)
    public CompletableFuture<InventoryResult> reserveInventory(List<Item> items) {
        return CompletableFuture.supplyAsync(() -> {
            return inventoryClient.reserve(items);
        }, inventoryExecutor);
    }

    private CompletableFuture<InventoryResult> inventoryFallback(
            List<Item> items, Exception ex) {
        "color: #6a9955;">// Return cached inventory levels - no external calls
        return CompletableFuture.completedFuture(
            InventoryResult.fromCache(items));
    }

    public Order createOrder(OrderRequest request) {
        "color: #6a9955;">// Use CompletableFuture to compose async calls
        CompletableFuture<PaymentResult> paymentFuture =
            processPayment(request.getPayment());

        CompletableFuture<InventoryResult> inventoryFuture =
            reserveInventory(request.getItems());

        "color: #6a9955;">// Combine results with timeout
        try {
            return CompletableFuture.allOf(paymentFuture, inventoryFuture)
                .thenApply(v -> buildOrder(
                    paymentFuture.join(),
                    inventoryFuture.join()))
                .orTimeout(5, TimeUnit.SECONDS) "color: #6a9955;">// Overall timeout
                .join();
        } catch (CompletionException e) {
            "color: #6a9955;">// At least one service failed or timed out
            return createPendingOrder(request);
        }
    }
}

"color: #6a9955;">// Rate limiting at the edge to prevent overload
@Component
public class EdgeRateLimiter {

    private final RateLimiter rateLimiter = RateLimiter.create(1000); "color: #6a9955;">/

    public boolean tryAcquire() {
        return rateLimiter.tryAcquire();
    }
}

"color: #6a9955;">// Monitoring circuit breaker states
```

```java
@Component
public class CircuitBreakerMonitor {

    @Autowired
    private CircuitBreakerRegistry circuitBreakerRegistry;

    @Scheduled(fixedRate = 30000)
    public void logCircuitBreakerStates() {
        circuitBreakerRegistry.getAllCircuitBreakers().forEach(cb -> {
            CircuitBreaker.Metrics metrics = cb.getMetrics();
            log.info("CB {}: state={}, failureRate={}, slowCallRate={}, bufferedCalls={}"
                cb.getName(),
                cb.getState(),
                metrics.getFailureRate(),
                metrics.getSlowCallRate(),
                metrics.getNumberOfBufferedCalls());
        });
    }
}
```

> ***Key Takeaway:*** *Circuit breakers alone aren't enough. Combine with bulkheads (isolation), timeouts (fail fast), and rate limiting (prevent overload) for true resilience.*