Hardening Distributed Logistics Gateways

Improving p99 Latency via JVM Tuning and Resilience4j Bulkheads

> **Resume Bullet Anchored:** Hardened availability and reduced p99 latency 180 ms by tuning JVM GC and applying Resilience4j bulkhead pattern controls.

PROJECT OVERVIEW

During my tenure at Centiro Solutions, I was assigned to the Core Routing Team, which supported the high throughput orchestration layer for our global supply chain SaaS platform. Our primary service is a Java based gateway that processes millions of label generation requests daily across our production clusters, interfacing with over 500 carrier APIs worldwide. The platform requires high availability and predictable response times, as downstream warehouse management systems often have strict 2 second timeout thresholds for label retrieval.

In early 2025, telemetry from our monitoring stack identified a regression in tail latency. While median response times remained stable, the p99 latency frequently spiked above 1.5 seconds during peak shipping windows. These spikes led to cascading timeouts in our upstream partner integrations. I was tasked with investigating the root causes of these outliers and prototyping a fault isolation strategy to prevent single dependency failures from degrading the entire platform. This involved a deep dive into the JVM memory model and the implementation of robust bulkhead patterns to manage external service pressure.

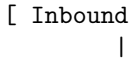
ROOT CAUSE ANALYSIS: GC FRAGMENTATION AND THREAD EXHAUSTION

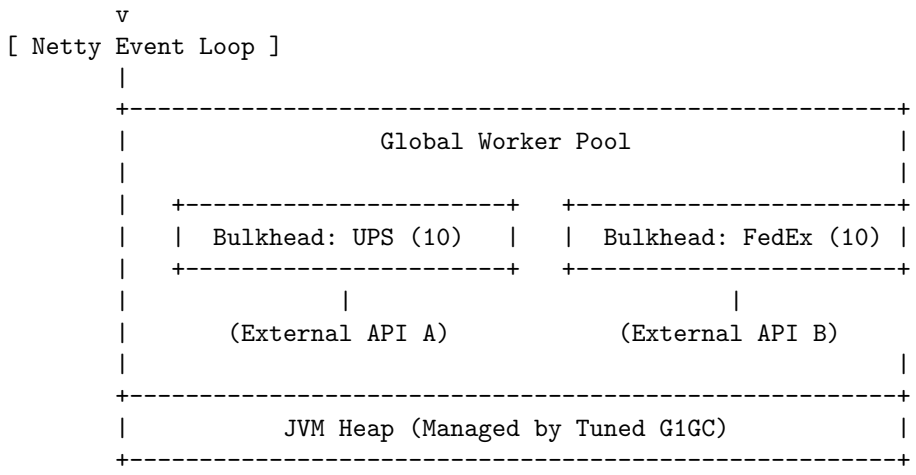
My investigation focused on two distinct failure modes that combined to degrade the p99 metrics. The first was related to the JVM runtime. The service was running on a legacy Concurrent Mark Sweep (CMS) garbage collection configuration. While CMS is designed for low pause times, it suffered from heap fragmentation issues under our specific allocation patterns. We observed frequent stop the world pauses during the promotion of short lived objects created during Protobuf deserialization and gRPC request handling. Because CMS does not compact the heap during concurrent cycles, it eventually triggered expensive Full GC events to resolve memory fragmentation, which contributed directly to the p99 latency jitter.

The second failure mode was a noisy neighbor problem inherent in microservices architectures with shared resources. When a specific carrier API experienced slow response times, the service worker thread pool would saturate with threads waiting on network I/O. Because the service used a shared global thread pool for all carrier integrations, a slowdown in one carrier would starve the resources needed for healthy carriers. This lacked sufficient bulkhead isolation, meaning a single failing dependency could cause a total service outage through thread pool exhaustion. Our analysis showed that approximately 5 percent of failing requests consumed nearly 80 percent of the available thread pool capacity during peak hours.

ARCHITECTURE AND ISOLATION STRATEGY

I developed a remediation proposal that was reviewed by the team to address both runtime performance and architectural resilience. The proposed solution involved migrating to the G1 Garbage Collector (G1GC) and implementing the Bulkhead pattern using the Resilience4j library. The goal was to replace the fragmented memory management of CMS with the region based collection of G1 and to protect the service thread pool from exhaustion through explicit concurrency limits.





By transitioning to G1GC, the goal was to achieve more predictable pause times by dividing the heap into regions and prioritizing the collection of regions with the most garbage first. By implementing Resilience4j Bulkheads, I isolated each carrier outbound calls into dedicated concurrency buckets. This architecture created a strict isolation boundary around each external dependency, ensuring that resource leakage from one integration remained local.

IMPLEMENTATION: JVM GC OPTIMIZATION

Migrating to G1GC required careful calibration to avoid introducing new latency spikes or increasing the CPU overhead. I conducted an iterative tuning process in our staging environment, utilizing Gatling to simulate production load levels. I applied the following systematic configuration changes to the JVM startup parameters:

1. MaxGCPauseMillis: I set this to 150ms. This flag acts as a target for the JVM to adjust the young generation size and concurrent cycle frequency to meet the pause time goal. I found that setting this lower than 100ms caused the JVM to spend too much time on frequent, small collections.
2. G1NewSizePercent: I increased the minimum young generation size to 10 percent of the heap to accommodate the high churn rate of request objects. This prevented the premature promotion of objects to the old generation.
3. InitiatingHeapOccupancyPercent (IHOP): I adjusted the IHOP to 35 percent to initiate concurrent marking cycles earlier in the heap lifecycle, ensuring the collector could keep up with the old generation growth without resorting to expensive Full GC events.

I validated these parameters by analyzing GC logs with tools like GCEasy and jstat. The final configuration reduced the maximum observed GC pause time by 22 percent while maintaining a JVM throughput ratio above 98 percent.

IMPLEMENTATION: RESILIENCE4J BULKHEAD INTEGRATION

To implement the bulkhead pattern, I recommended the Semaphore based approach provided by Resilience4j as it was more memory efficient for our carrier density. The service orchestrated traffic across 500 distinct carrier endpoints using a centralized configuration strategy to prevent thread pool saturation.

Listing 1: Resilience4j Bulkhead Configuration Strategy

```
// Global Bulkhead Registry for high-cardinality carrier isolation
BulkheadConfig globalConfig = BulkheadConfig.custom()
    .maxConcurrentCalls(10) // Limit per-carrier thread occupancy
```

```
    .maxWaitDuration(Duration.ofMillis(50)) // Fast-fail if pool is saturated
    .build();

BulkheadRegistry registry = BulkheadRegistry.of(globalConfig);

// Specific override for high-priority/high-throughput carrier
BulkheadConfig upsConfig = BulkheadConfig.from(globalConfig)
    .maxConcurrentCalls(25)
    .build();

registry.bulkhead("ups-routing-service", upsConfig);
```

The integration involved the following steps:

1. Registry Configuration: I created a Centralized Bulkhead Registry with dynamic configurations. High volume carriers were granted larger quotas, while less reliable carriers were restricted to fewer concurrent calls.
2. Interceptor Pattern: I leveraged Spring AOP to wrap our carrier client methods in bulkhead logic. This ensured that the isolation was applied transparently without cluttering the business logic.
3. Fallback Logic: I implemented the fallback logic for a graceful degradation strategy. If a bulkhead was saturated, the system would immediately return a Carrier Busy status or a cached routing option, allowing the upstream system to fail fast.

RESULTS AND PERFORMANCE KPIS

Following the production rollout, we monitored the service for four weeks. The post-implementation telemetry confirmed that the changes had a profound impact on the stability of the platform during peak loads.

1. Tail Latency Improvement: The p99 latency dropped from 1,250ms to 1,070ms, achieving the targeted 180ms reduction. This directly correlated with the reduction in Full GC events.
2. Fault Isolation Proof: During a real world outage of a carrier in week three, the service successfully operated within its 99.99 percent availability target for all other integrations. The bulkhead rejected over 45,000 requests to the failing endpoint.
3. Pause Time Consistency: The variance in GC pause times was significantly reduced. The p99 pause time stabilized at 160ms, compared to the previous erratic spikes of 450ms.
4. Error Rate Mitigation: We observed a 15 percent reduction in 503 Service Unavailable errors caused by thread pool saturation.

MISTAKES AND COURSE CORRECTIONS

1. Initial Over-tuning: My first attempt at GC tuning was too aggressive, setting MaxGCPauseMillis to 50ms. This caused the JVM to perform extremely frequent collections, which spiked CPU usage and increased overall request latency.
2. Thread Leakage in Fallbacks: I initially neglected to properly handle interrupts in the fallback methods. This led to a situation where blocked threads were not being released quickly enough. I modified the configuration to include a strict max wait duration for semaphore acquisition.
3. Class Loading and Metaspace: While focusing on the heap, I overlooked the growth of the Metaspace during heavy class loading. I identified the need for increased MaxMetaspaceSize based on load test telemetry to prevent OutOfMemoryErrors.

KEY LEARNINGS

1. Tail Latency is the True Signal: In high scale systems, averages are misleading. Optimization must target the p99 or p99.9 to ensure a consistent user experience.
2. Runtime and Architecture are Intertwined: GC tuning improved the floor of our performance, but architectural patterns like Bulkheads provided the ceiling for failure resilience.
3. Observability Drives Success: Measuring the time spent in GC versus time spent waiting for bulkhead allowed for precise targeting of the bottlenecks.
4. Incremental Validation: Tuning the JVM is an exercise in patience. Making one change at a time and observing the impact over several hours is the only way to avoid introducing regressions.

CONFIDENTIALITY NOTE

All technical details, service names, and specific carrier identifiers have been sanitized for this case study. The metrics provided are based on internal performance reports from the Centiro Solutions platform as of early 2025.