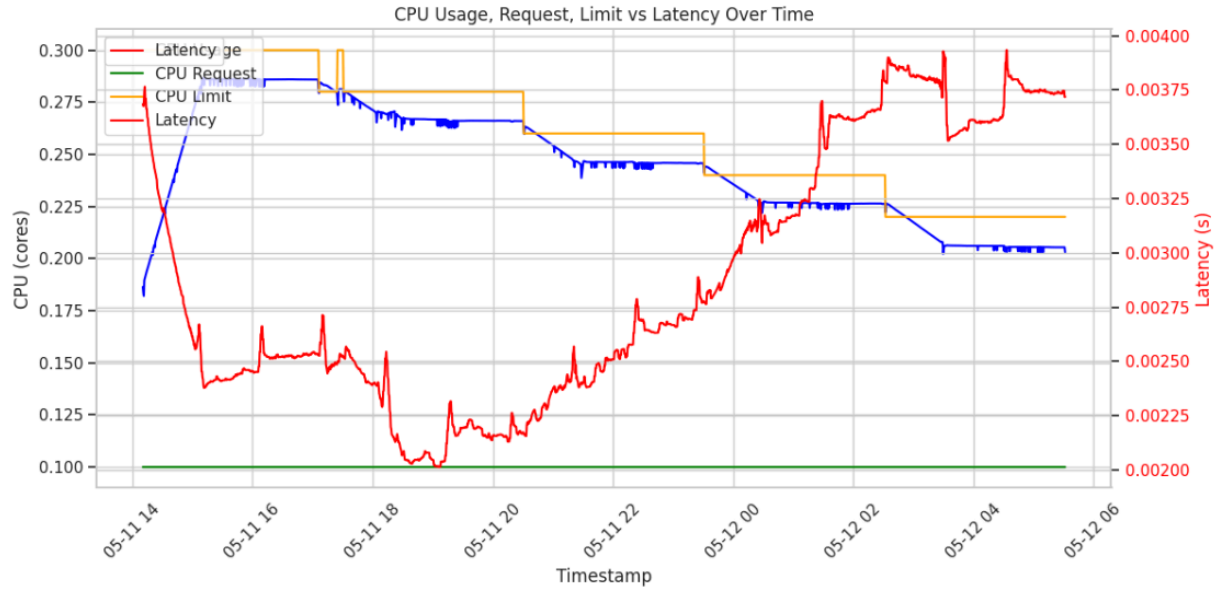
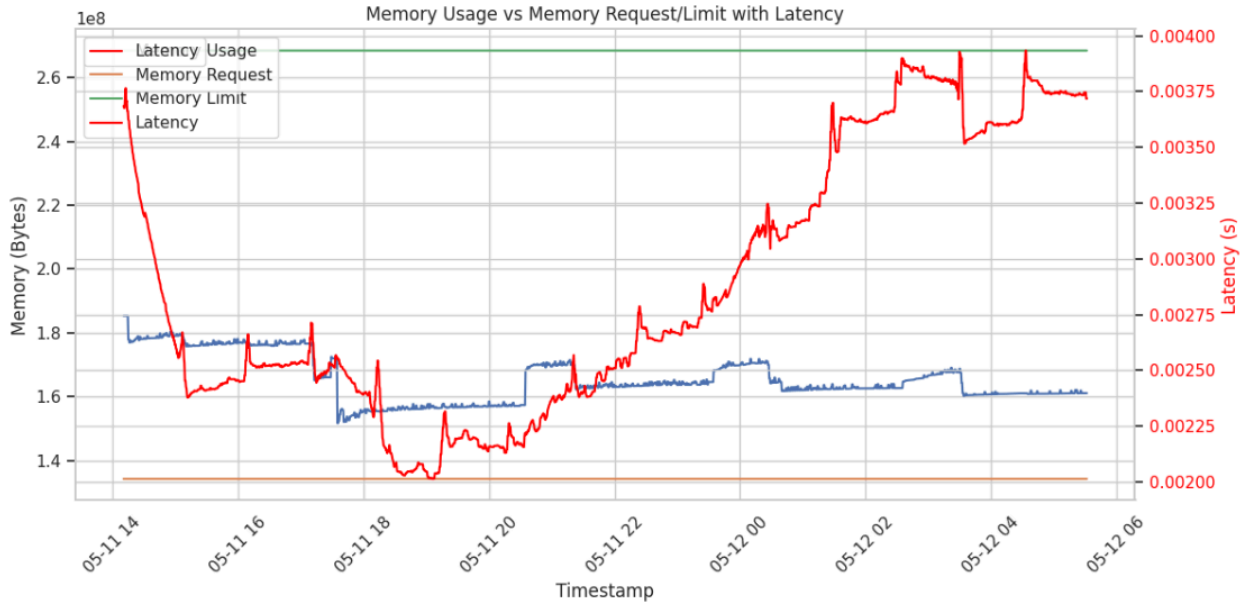
Deep Analysis

Service 1





Latency Spikes Throughout the graph

1. CPU Throttling & Scheduling Delays
    - As we reduce the CPU limit, Kubernetes enforces it strictly.
    - When the application demands more CPU than allowed, it gets throttled.
    - This causes:

    - Thread queuing
    - Context switching delays
    - Slower request handling
- Latency spikes, even if usage appears “low.” It’s not that the app doesn’t need CPU—it’s being denied it.

2. Garbage Collection (GC) Delays in Java
    - Your service is Java-based, meaning GC plays a major role in runtime latency.
    - GC requires CPU time. When CPU is throttled:
        - GC runs less frequently or for longer durations.
        - Heap space fills up -> **minor GC becomes major GC** -> latency spikes.
        - Threads may pause during GC (especially with `Stop-The-World` events).
    - Intermittent but large latency spikes, especially when memory usage increases or GC is delayed.

3. Jitter from Background Services

- Java services may have background threads for:
    - Logging
    - Health checks
    - Internal thread pools
- These compete with the main request-processing thread, especially when CPU is limited.
- Any spike in background task CPU demand can slow down response latency.
- Short-lived but frequent latency spikes, seen as jitter.

4. Thread Pool Saturation

- Java web services often use thread pools (e.g., Tomcat, Jetty).
- If CPU is insufficient, request threads:
    - Take longer to process
    - Build up in the queue
- Eventually, the queue becomes saturated, forcing:
    - Rejected requests
    - Slow throughput -> high latency
- Latency spikes increase in magnitude the longer CPU remains under-provisioned.

5. Dynamic Load or External Triggers

- We are sending multiple parallel requests (e.g., 10/s from a client).

- If there's even slight load imbalance, one pod may receive a burst.
- Combined with CPU limits, this causes temporary overload -> spike.
- Spikes appear even if average load is low, due to micro-bursts.

6. Heap Memory Pressure -> CPU Demand Loop

- When memory usage increases, the JVM:
    - Allocates more memory
    - Increases GC frequency
    - GC needs CPU -> which is already constrained
- This forms a feedback loop: memory increase -> GC -> CPU -> latency -> more memory usage…
- Repeated latency spikes as heap usage and CPU limits fight each other.

Latency Drop at 05-11 20

- This is not a natural decline over time but a sharp, step-like drop.
- At this time increased CPU limits, the container:
    -