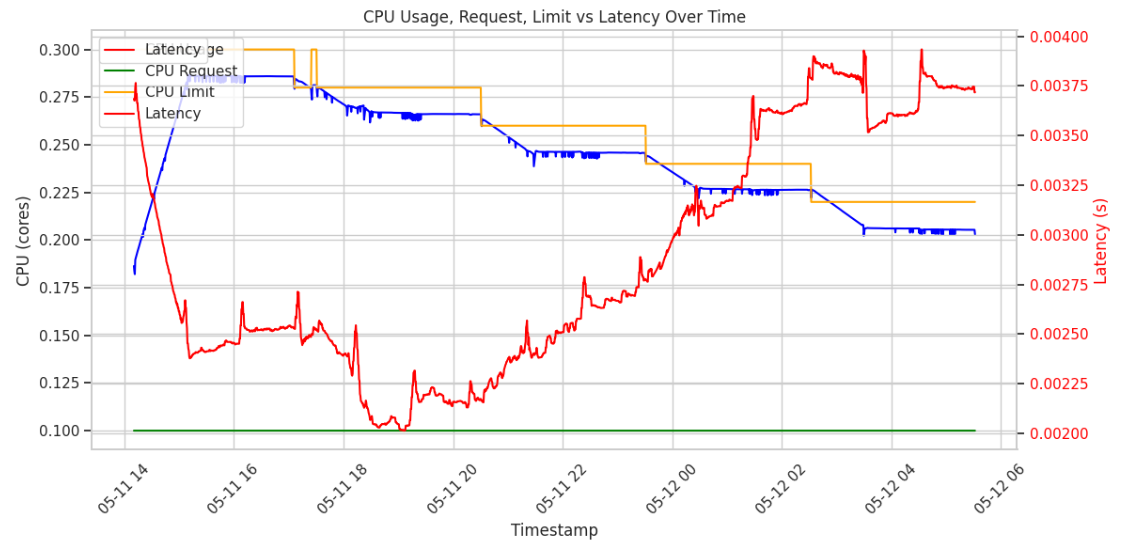
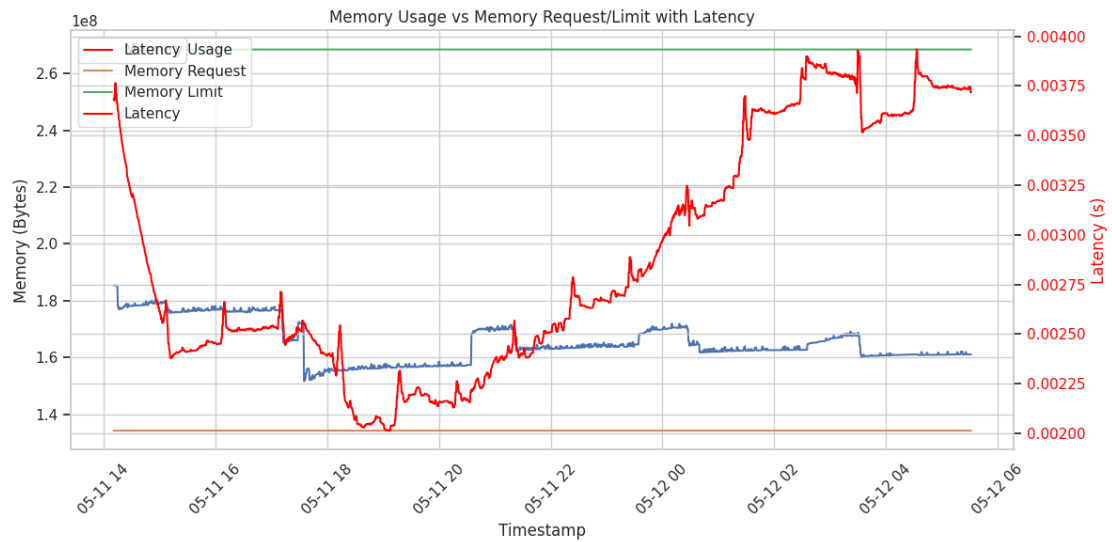
# Deep Analysis

### 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.

1. 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.

1. 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.

1. 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.

#### 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.

1. 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:
  + Stopped being throttled
  + Could serve requests faster
  + GC runs completed faster, reducing pauses
  + Latency dropped quickly
* Drop in CPU usage % may seem counterintuitive but indicates more headroom was available, so no throttling.
* With higher limits, actual CPU demand is met without delay -> lower latency.

#### Latency Drop at 05-12 04

1. Just-In-Time (JIT) Compilation Kicked In

* Java JIT Compiler (HotSpot) is known to:
  + Compile "hot" methods (those called frequently) at runtime into optimized native code.
  + This often happens after some sustained activity - like several invocations.
* Latency drops suddenly after optimized code replaces interpreted bytecode.
* The drop near 05-12 04:00 suggests that JIT optimizations became active, improving response time without needing more CPU.

#### Why Latency is increasing after a the above drop

1. CPU Limit Throttling Reasserts Itself

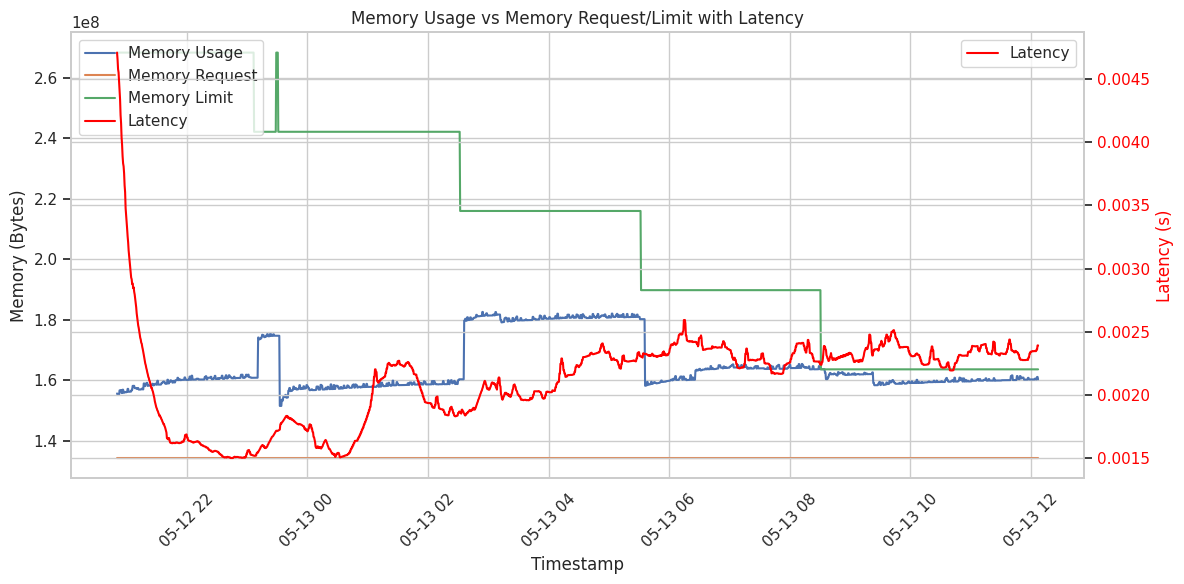
* This is common in long-running Java services, and it can result from a combination of dynamic runtime factors:
  + Even with optimized bytecode, we’re still operating under a constrained CPU limit.
  + Especially during peak processing or multiple requests, the OS scheduler throttles the container, leading to increased latency again.

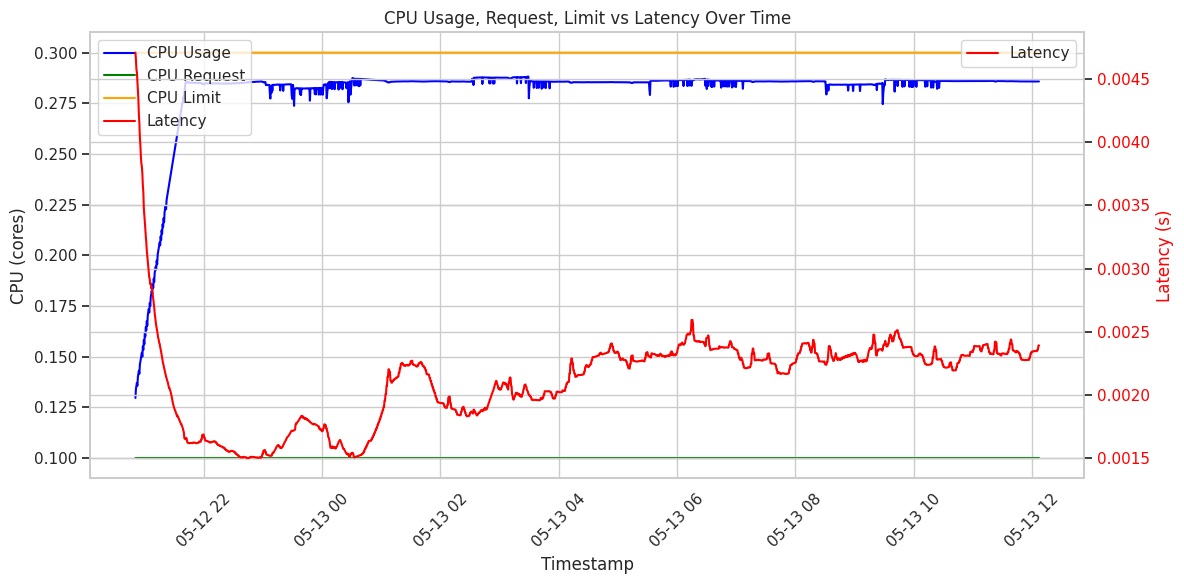
1. Garbage Collection Pressure Builds Up Again

* JIT optimizations often allocate more short-lived objects as they inline and unroll loops.
* This can cause:
  + More frequent young GCs
  + Occasional longer full GCs

1. Then internal request queues may grow again, increasing latency.

### Service 1 - Only Memory Limit Reduction





* After each memory reduction:
  + No sudden latency spike.
  + However, **l**atency rises gradually as memory is reduced.
* Interpretation:
  + JVM doesn't immediately break under low memory.
  + But less heap space **->** more frequent GC **->** longer GC pauses -> gradual latency increase.
* This fits classic Java GC behavior:
  + The heap is small and GC has to work harder to reclaim memory, latency increases due to more frequent and longer GC pauses.
* CPU usage remains flat.
* So latency increases despite plenty of CPU, again pointing to memory pressure as the source.

Why there is a higher memory usage from 05-13 02 to 05-13 06 more than other places?

1. More Complex Requests (Data-Dependent Workload)

* This indicates input data changed during this time, likelarger numbers - the app will have:
  + Built larger in-memory structures (arrays, strings, logs, etc.).
  + Run more iterations or used recursive methods.
  + Consumed more heap space per request.

1. JVM Memory Fragmentation or Delayed GC

* During this window, GC may not have run aggressively - possibly to avoid long pauses.

* The memory spike between 05-13 02 and 05-13 06 likely results from accumulating memory pressure due to tight memory limits and temporarily ineffective GC cycles, possibly worsened by complex input data or background activity.

### Service 1 - Both Resource Limits Reduction

