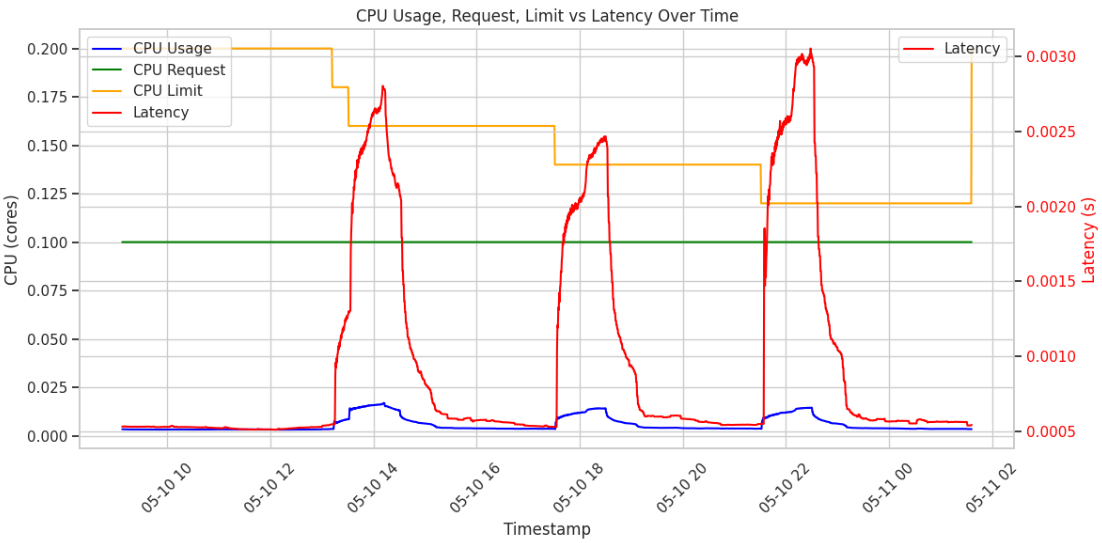
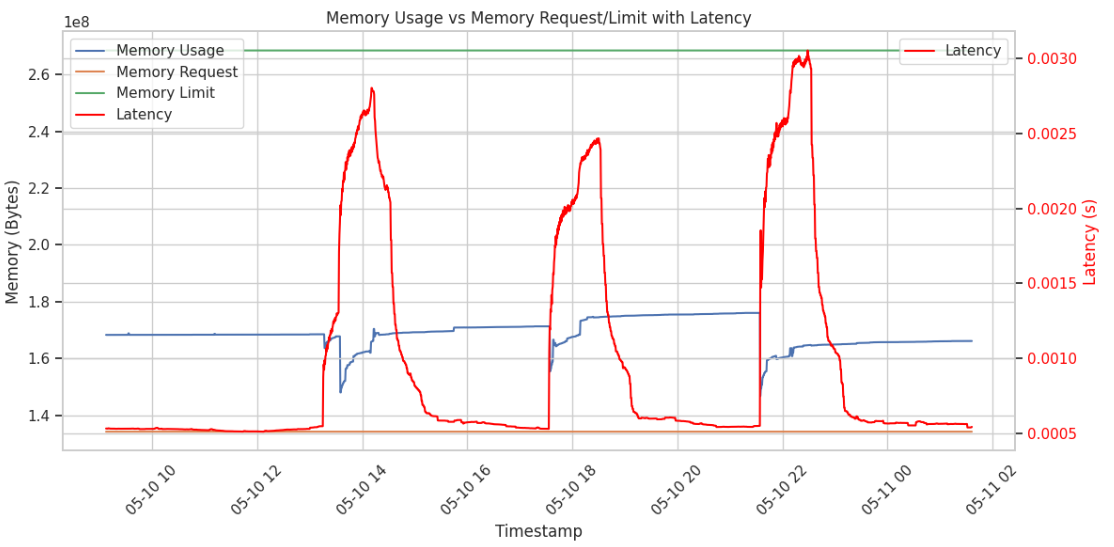
# HashGen Service Only CPU Limit Reduction





* Latency correlates strongly with CPU usage spikes. So, this service is CPU intensive.
* Post-spike, once CPU usage drops, latency returns to low values, indicating no long-term system degradation.
* A slight latency increment at the end indicates:
  + Memory/CPU fragmentation
  + Increased queuing delays due to reduction of CPU
  + Application-level slowdown due to frequent container resource patching (As we’re dynamically reducing limits)
* At each latency spike, memory usage falls rapidly, implying memory is being released & container is restarted.
* The increasing slope before the drop suggests gradual memory accumulation, e.g., from unfreed objects or request buffers not being released.

Why is this graph different from service 1 & service 2 graphs?

1. Sharp, Periodic Latency Spikes

* Unlike previous services that may have had steady or gradually increasing latency, this graph shows distinct, periodic spikes.
* These spikes align precisely with CPU and memory usage peaks, suggesting a strong resource-bound behavior — especially CPU.

This pattern is common for stateless**,** compute-bound workloads (like hash generators), where each request triggers intense CPU processing (SHA). Once requests queue up faster than they can be processed (due to CPU limits), latency surges rapidly — then drops once load subsides.

1. Latency Drops Immediately After Peaks

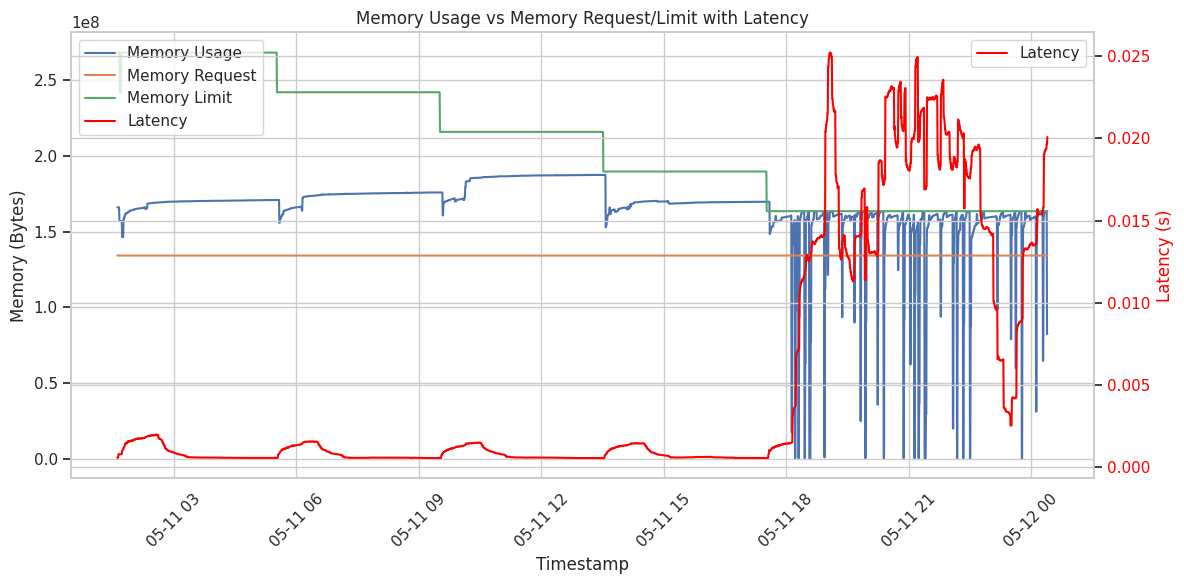
The latency returns to near-zero immediately after usage drops, which differs from services where latency lingers due to cache misses, DB backlogs, or I/O saturation.

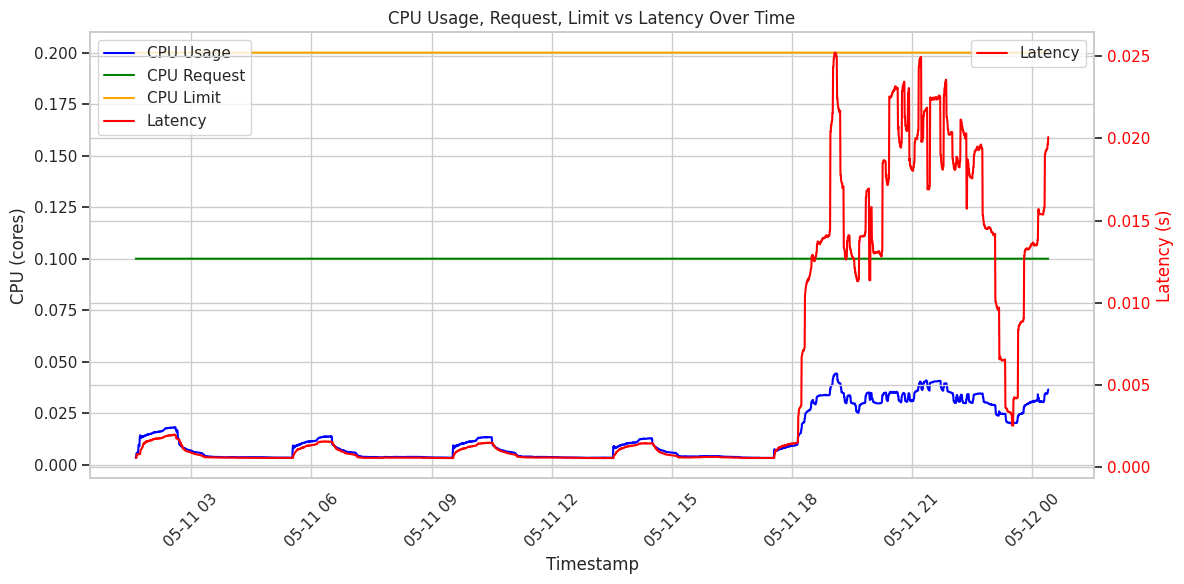
Hash generation doesn't rely on external services (e.g., DB, caches), so once CPU/memory pressure is relieved, it can process new requests quickly. Previous services might have had downstream dependencies that cause persistent latency even after load drops.

1. Latency spikes are tightly tied to CPU/mem exhaustion and drop instantly when load eases.

1. Previous services hadstateful, or multi-tier architecture influences.

# HashGen Service Only Memory Limit Reduction





* After memory usage hits the memory limit:
  + Memory usage starts fluctuating sharply (likely due to GC activity).
  + Latency rises significantly, with noisy and prolonged spikes.
  + Memory requests remain static, suggesting they aren't tuned dynamically.
* Around 05-11 18, CPU usage increases, tracking load, and then latency spikes.

Latency Spike After 05-11 18: Trigger = Memory Limit Reduction

* This spike coincides with memory limits being reduced below the threshold required for stable GC operations in a Java service.
* What happened:
  + Java's Garbage Collector (GC), lsenses increased memory pressure.
  + Smaller heap = more frequent GC = GC pauses -> latency spikes.
  + We also observe Full GCs if the heap size becomes too small to sustain concurrent collection.
* Even if memory usage is below limit, the JVM knows about the limit (via cgroups) and triggers GC early and often under memory stress.

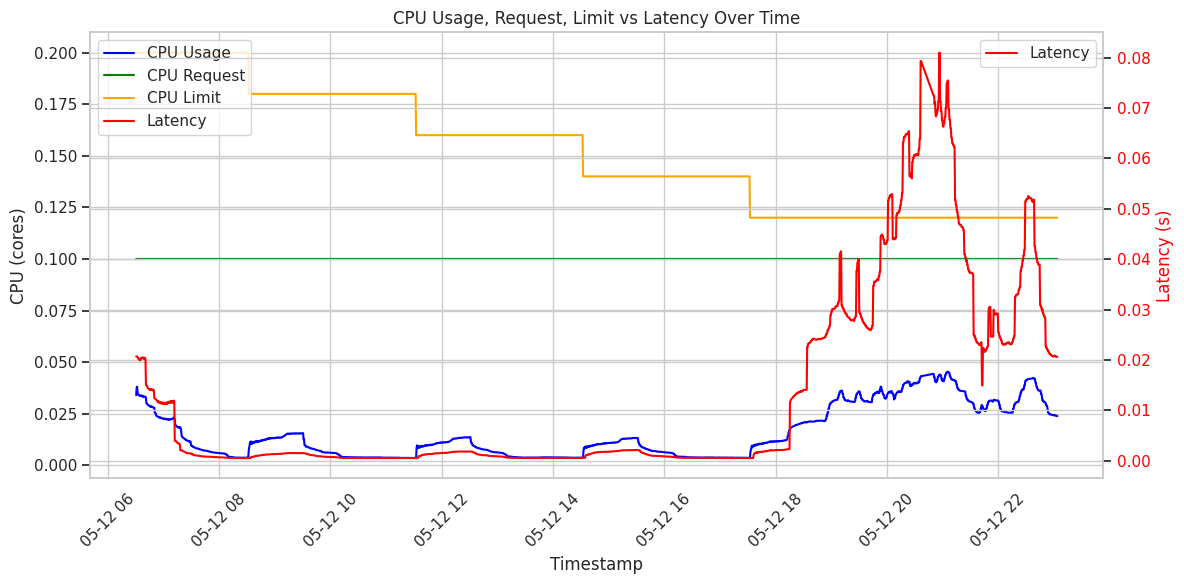
Memory Usage "Jitter" (Fluctuations)

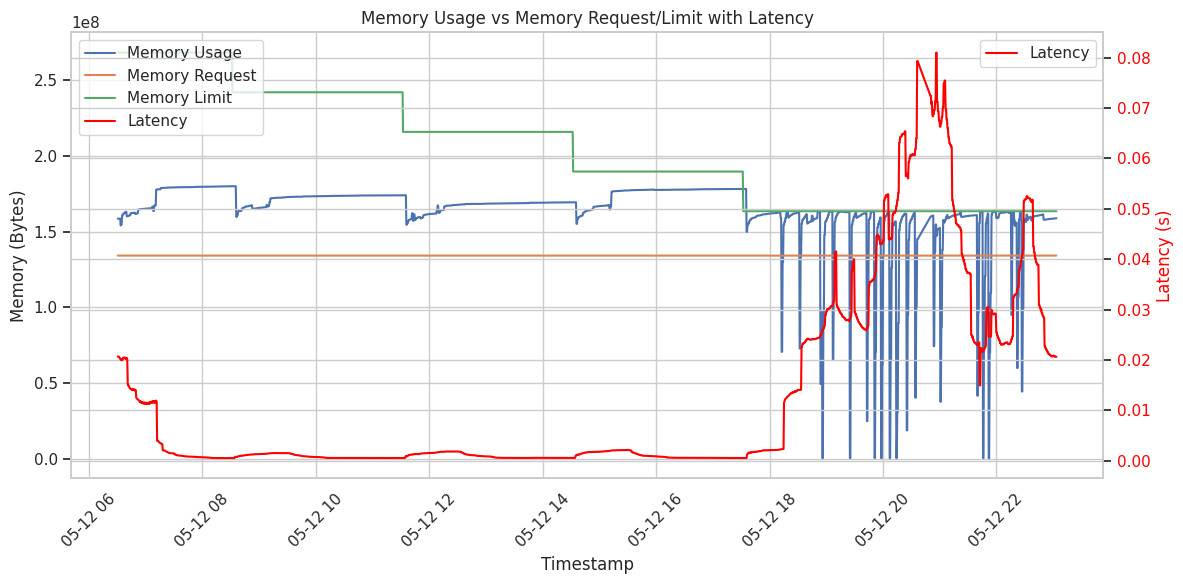
* The sharp drop & rise in memory usage after limit reductions is classic GC behavior:
  + Heap fills -> GC -> memory drops -> heap fills again -> repeat.
* Sharp downward spikes mean memory is reclaimed, but too frequently.
* JVM is working harder to stay within reduced limits.

CPU Usage Increase Post 05-11 18

* Likely a result of:
  + More GC cycles consuming CPU.
  + JVM thread management overhead under constrained memory.
* This is classic “CPU/GC fight” - Java consumes CPU for GC trying to compensate for memory shortage.

# HashGen Service Both Resource Limits Reduction





* CPU usage is low at first, then rises and tracks the limit closely after ~05-22 18.
* Latencyspikes sharply and persistently post ~05-22 18, exactly where CPU limit reductions reach critical thresholds.
* Memory Usage Spikes suddenly after 05-22 18 and becomes erratic, with extreme oscillations indicating OOM killer or GC stress.
* Latency is perfectly correlated with the onset of these memory usage spasms and high CPU usage.

1. Latency Begins Climbing When CPU Saturates (Post ~05-22 18)

* CPU usage begins rising and tracks just below the reduced CPU limit.
* Latency starts climbing soon after, suggesting the service can no longer compute hashes fast enough under CPU constraint.
* CPU is the primary bottleneck initiator - this is when your hash service begins slowing down.

1. Latency Spikes Dramatically When Memory Usage Becomes Unstable

* Around 05-12 20, memory usage suddenly:
  + Becomes noisy and erratic (due to GC thrashing).
  + Drops rapidly (due to memory eviction & OOM killer resets).
* At the same time:
  + Latency spikes from 0.01s to 0.08s - 8× increase!
  + This indicates memory pressure has now exacerbated CPU delay, due to:
    - GC pauses from frequent allocations.
    - JVM resizing heap or struggling with limited memory.
    - Failed memory allocations (temporary or fatal).

1. Strong Compounding Effect Between CPU & Memory Bottlenecks

* When only CPU was constrained (05-22 18), latency began increasing.
* When memory also began failing (05-22 20), latency spiked massively.
* CPU was the primary bottleneck, memory became a secondary amplifying factor once usage crossed a stress threshold.

Key Observations

* CPU Limit - Primary trigger - CPU usage tracked the limit - queue builds - latency increases.
* Memory Limit - Amplifier - only impacted latency when usage neared or exceeded the limit, triggering instability.
* Memory Usage - Smooth initially -> chaotic under stress -> correlates with the worst latency spike.
* Latency Trend - Steady until system resource stress kicks in, then increases exponentially.

Behavioural Summary

* This hash generator service is CPU-heavy, with some memory sensitivity under load.
* Latency remained low while CPU and memory were under their thresholds.
* Once CPU became saturated and memory started oscillating, latency spiked — the system was cascading into performance degradation.