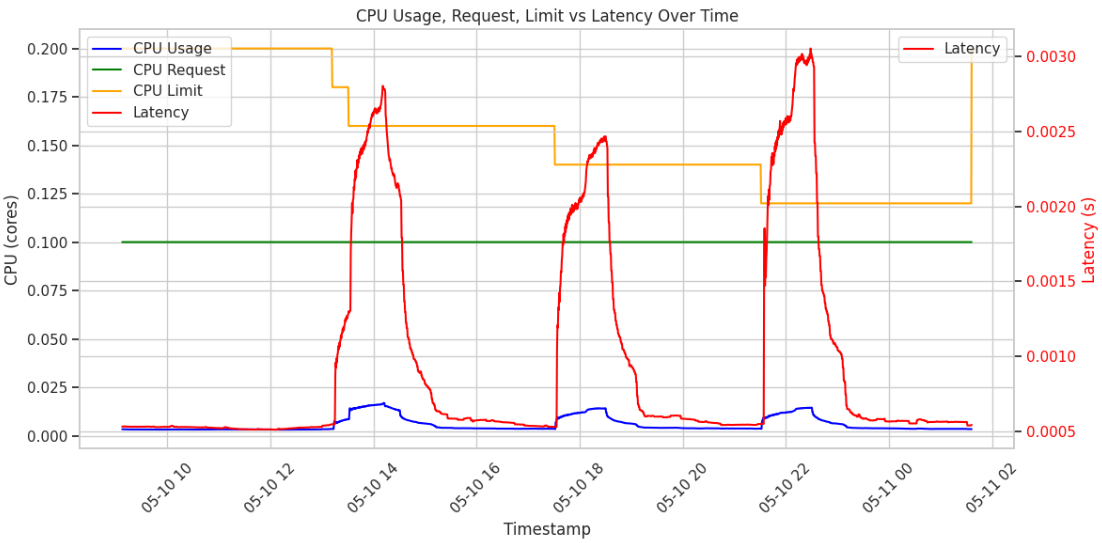
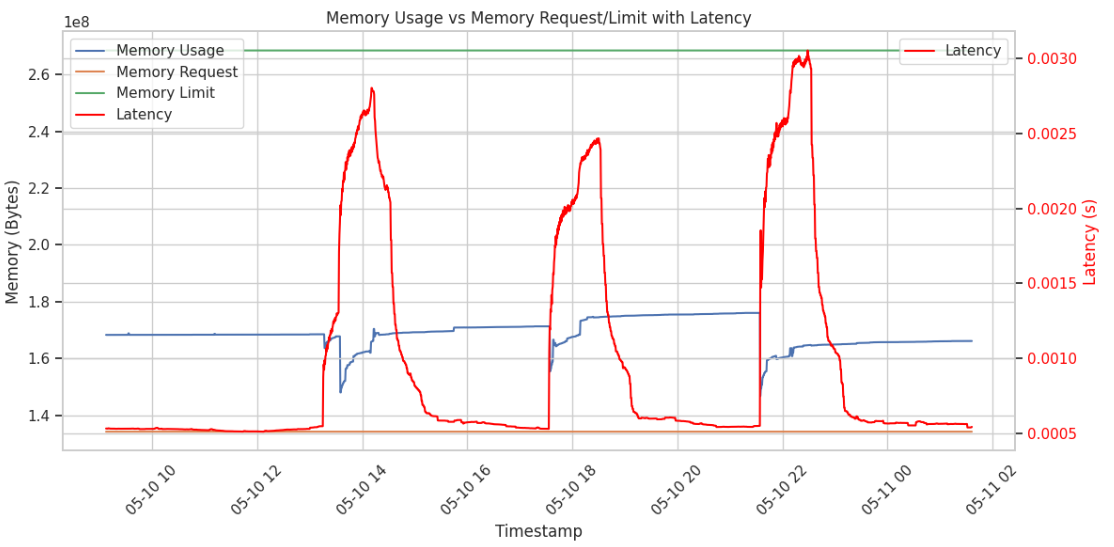
# HashGen Service Only CPU Limit Reduction





* 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

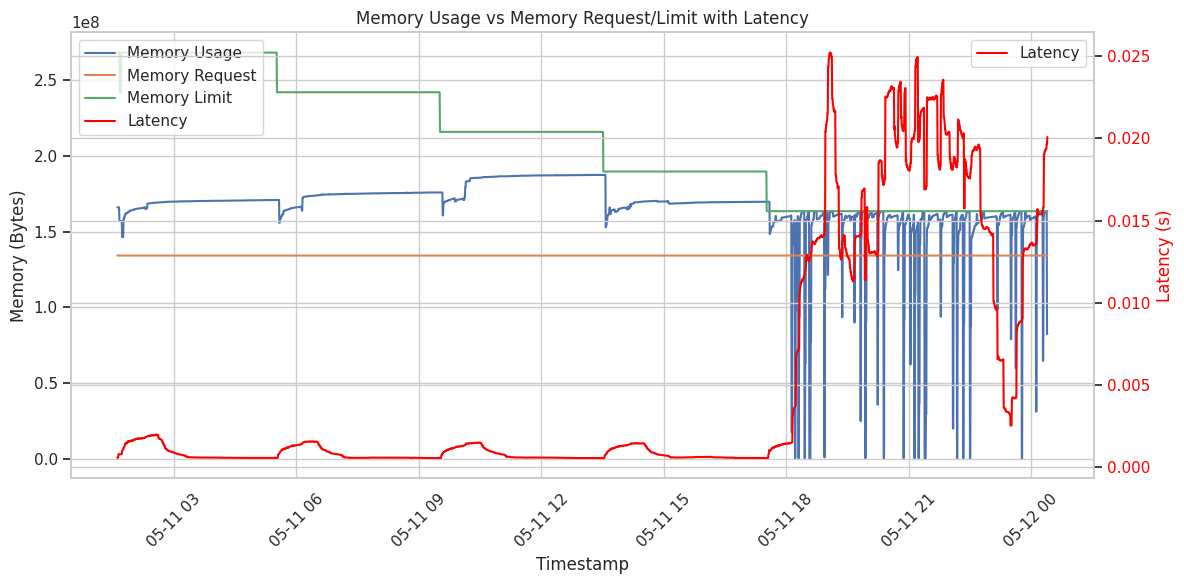
The latency returns to near-zero immediately after usage drops, which differs from services where latency lingers due to cache misses, DB backlogs, or I/O saturation.

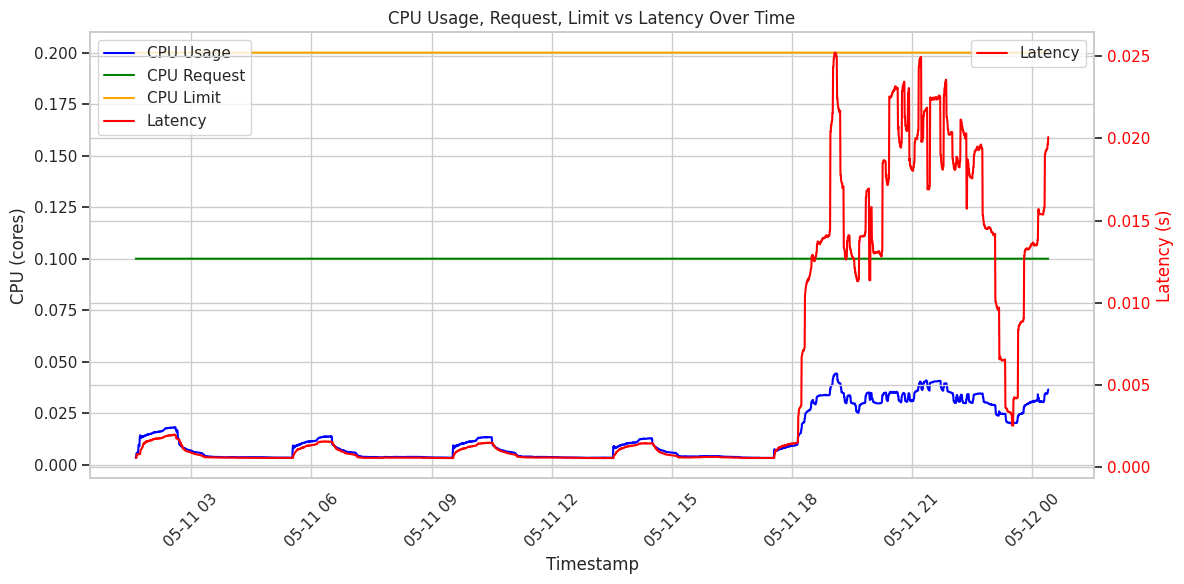
Hash generation doesn't rely on external services (e.g., DB, caches), so once CPU/memory pressure is relieved, it can process new requests quickly. Previous services might have had downstream dependencies that cause persistent latency even after load drops.

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

