# RandPwGen Service Only CPU Reduction





* CPU usage (blue) rises gradually, peaks, then falls.
* Latency (red) follows a smoother, less spiky curve compared to the hash generator.
* Unlike hash generation, latency here increases proportionally with load, not abruptly.
* This indicates the password generator is likely less CPU-intensive per request and can handle small bursts without immediately saturating CPU. Latency degradation is more graceful, suggesting:
* Lightweight password generation logic (random sampling)
* Efficient concurrency (e.g., async)
* No blocking computation (unlike hashing)
* No correlation between memory and latency (unlike the hash generator).
* Latency is **purely CPU-bound** here, not memory-bound. You’re not dealing with:
* GC pressure
* Fragmentation
* Memory-related queuing
* Like the hash generator, latency drops fast after CPU usage declines.
* Indicates:
  + No queued backlog
  + Stateless request handling
  + Possibly short-lived connections

Why spike at 05-11 06 is lower than the previous ones?

1. Load Have Been Slightly Lower

* Memory usage at 05-11 06 peaks lower than at earlier timestamps.
* Implies that the incoming request rate was possibly slightly reduced, or the load was less bursty.
* Less burst = fewer concurrent requests to queue or throttle -> latency spike was dampened.

1. CPU Usage Decay Was Smoother

* After peaking, CPU usage drops more gradually, suggesting fewer queued requests or a more stable serving pattern.
* This smoother behavior avoids congestion collapse, helping latency remain manageable.

Why memory usage increased in this service?

Workload Nature: CPU-Bound vs Memory-Bound

* Hash generation (SHA) is often CPU-bound but also involve:
  + Temporary large objects (hash buffers, salts, input copies).
* If CPU is throttled:
  + Requests start queuing up (because they take longer).
  + These pending requests hold onto memory (e.g., input buffers, request objects).
  + Memory usage increases over time.

Why memory usage drops when CPU limits drop?

1. Reduced CPU -> Less Work Done -> Less Memory Needed

When we reduce CPU limits:

* Our application gets less CPU time.
* It processes fewer requests per unit time.
* Fewer requests = fewer in-flight objects, smaller queues, and less transient memory allocation.

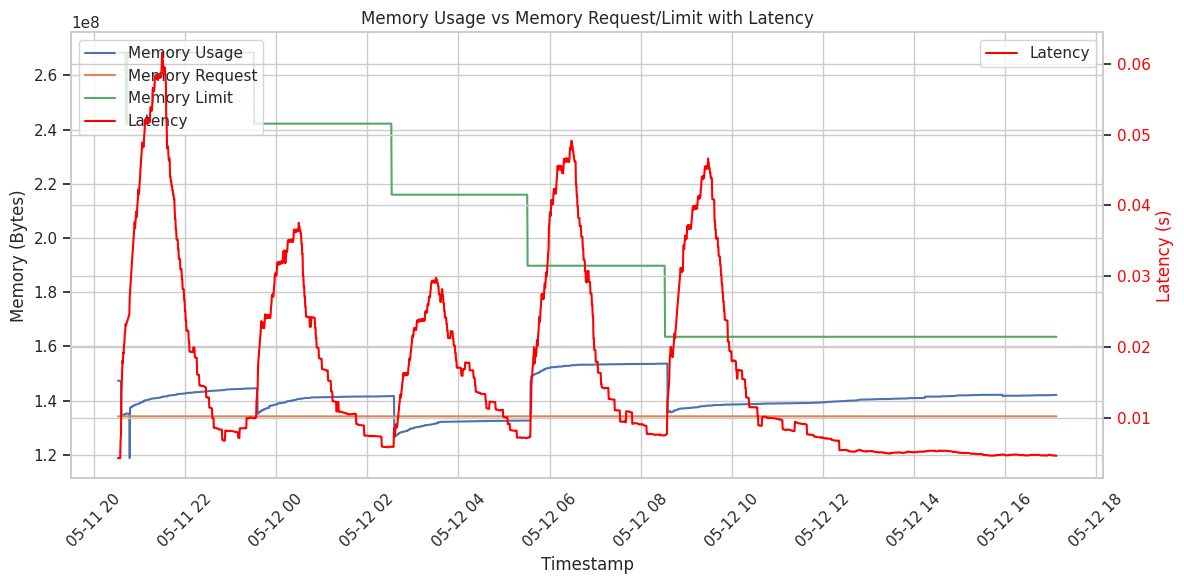
1. Garbage Collection (GC) Kicks In More Aggressively

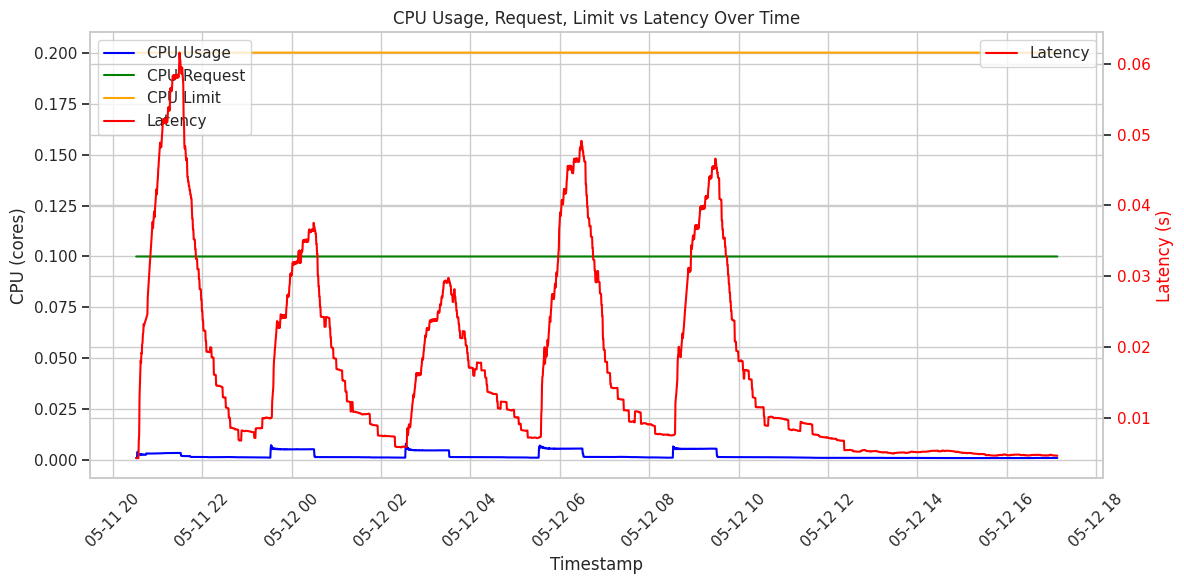
* With lower CPU, the app slows down, which may delay object creation.
* Some runtimes (Java) have background GC which now gets more breathing room (since fewer user threads are active).
* That leads to more frequent memory cleanup, resulting in visible drops in memory usage.

1. Latency Increases = Request Backoff or Drop

* In real systems (with scaling), high latency can lead to:
  + Clients backing off.
  + Requests being dropped or rejected.
  + Work being deferred.
* This results in less memory pressure, because fewer request contexts are active.

# RandPwGen Service Only Memory Reduction





Latency Patterns

* Latency peaks sharply and periodically, synchronized with CPU and memory usage peaks.
* As memory limit decreases step by step:
  + Latency peaks get slightly taller and wider.
  + After each step down in memory limit, latency becomes more sensitive to load spikes.
* The service starts encountering memory pressure as memory limit nears actual memory usage, leading to GC stress and CPU contention, both of which increase latency.

Memory Usage Dynamics

* This shows periodic bursts in memory usage, peaking well below the memory limit.
* However, memory usage aligns tightly with CPU usage, indicating memory usage is work-driven (likely per-request object allocations).
* After limit reductions, usage **s**till stays below the new limit, suggesting the app doesn't leak or scale memory linearly with load.
* The service allocates and releases memory efficiently, but reduced limits compress heap, leading to more frequent GC -> latency impact.

CPU Usage Dynamics

* CPU usage peaks match memory usage and latency.
* The base CPU usage is very low, but spikes up during bursts.
* CPU request/limit remain unchanged, so there’s no CPU throttling, but increased CPU is likely due to GC activity.

Why Memory Limit Reduction → Higher Latency?

Java’s GC is heap size-sensitive:

* Smaller heap -> less space for allocation -> more frequent GCs.
* Frequent GCs = higher CPU usage and pause times -> higher latency.
* When memory usage touches or nears the limit, the JVM experiences backpressure, leading to longer GC cycles or full GCs.

Even though memory usage appears “safe,” this doesn't mean latency is unaffected.

The frequency and duration of GC cycles increase subtly until they start impacting response times.