# 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.

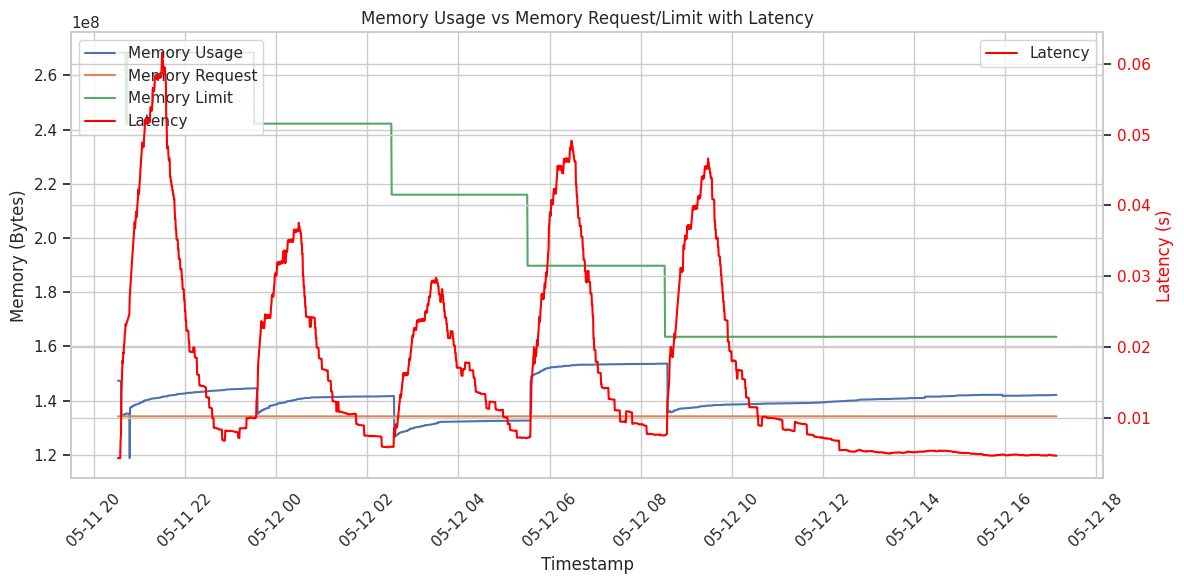
1. Garbage Collection (GC) Kicks In More Aggressively

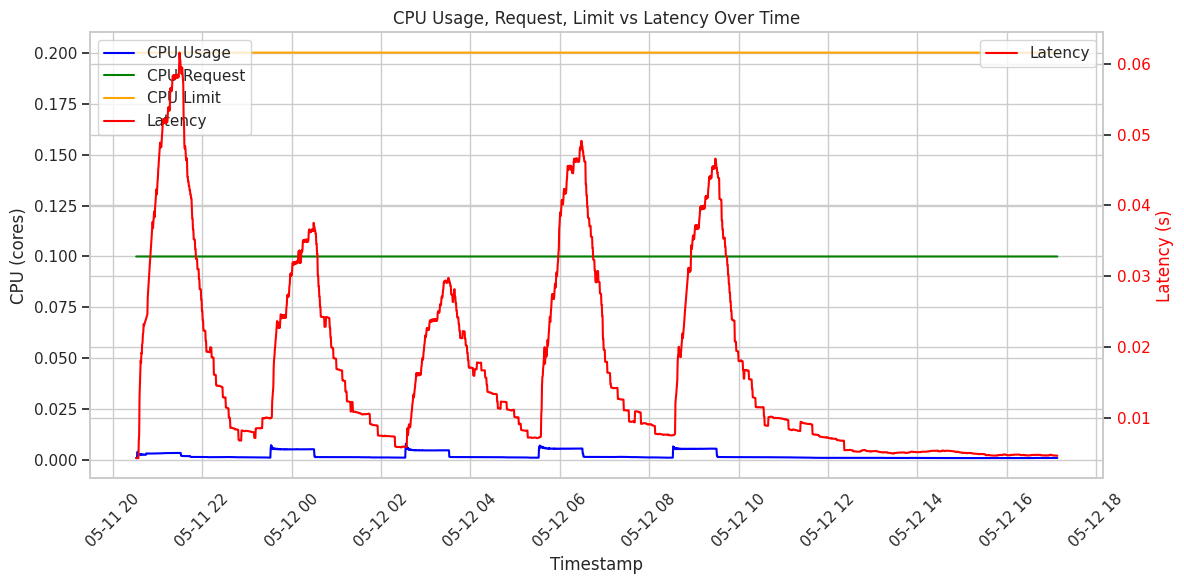
* With lower CPU, the app slows down, which may delay object creation.
* Some runtimes (Java) have background GC which now gets more breathing room (since fewer user threads are active).
* That leads to more frequent memory cleanup, resulting in visible drops in memory usage.

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.

why latency spike at 05-12 04 is lower than the spike at 05-12 00?

1. At 05-12 00: First Major Spike

* Memory Limit is still high, but reducing from initial values.
* The system is experiencing its first "shock" of pressure as memory is no longer abundant.
* Garbage Collection (GC) may not yet be optimized to the new heap size - JVM tuning typically adapts over time.
* Full GC as memory pressure hit unexpectedly during burst.
* CPU also shows a sharper increase, indicating high CPU-GC contention.
* Higher latency due to initial heap pressure + GC adaptation lag.

1. At 05-12 04: Second Spike

* Memory limit is lower, but memory usage is also reduced slightly.
* The service and JVM have adapted to tighter limits:
  + Shorter-lived objects.
  + More frequent, but shorter GCs.
* CPU usage rises again but is less spiky, implying GCs are quicker, possibly due to fewer major heap promotions.
* There also be less concurrent request overlap compared to earlier.
* Even though memory is tighter, the GC cycles are shorter and smoother, leading to a lower latency spike.

Why memory usage has this relevant pattern?

* Memory usage rises gradually, peaks, then drops sharply.
* This pattern repeats periodically, with the memory limit reductions (until the memory limit gets too low).
* Latency seems to correlate somewhat with memory usage spikes (due to GC events).

**Deep Dive into the Cause**

1. JVM Heap Allocation & Object Lifecycle

* The Java Virtual Machine allocates memory for objects primarily in the young generation.
* As requests come in (10 - 20 req/s), objects are created, filling up heap space.
* Once a threshold is reached, a GC is triggered (minor or major):
  + Short-lived objects get collected.
  + Some surviving objects are moved to the old generation.
* After GC, heap memory drops.
* The rise-and-fall pattern in memory usage is caused by regular object creation + GC cleanup cycles.

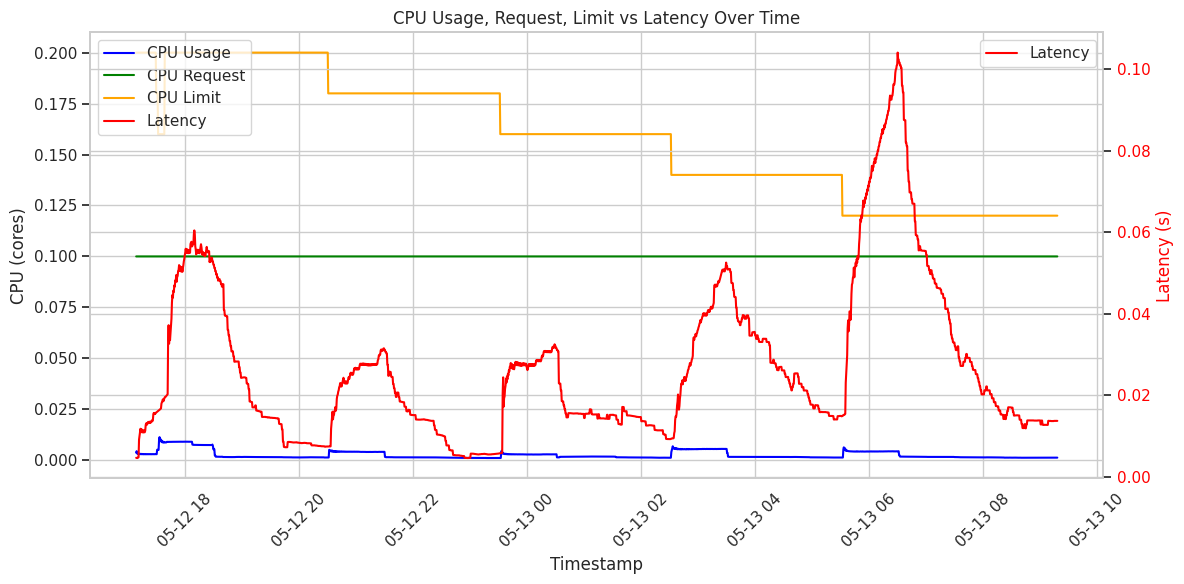
1. Service-Specific Workload Pattern

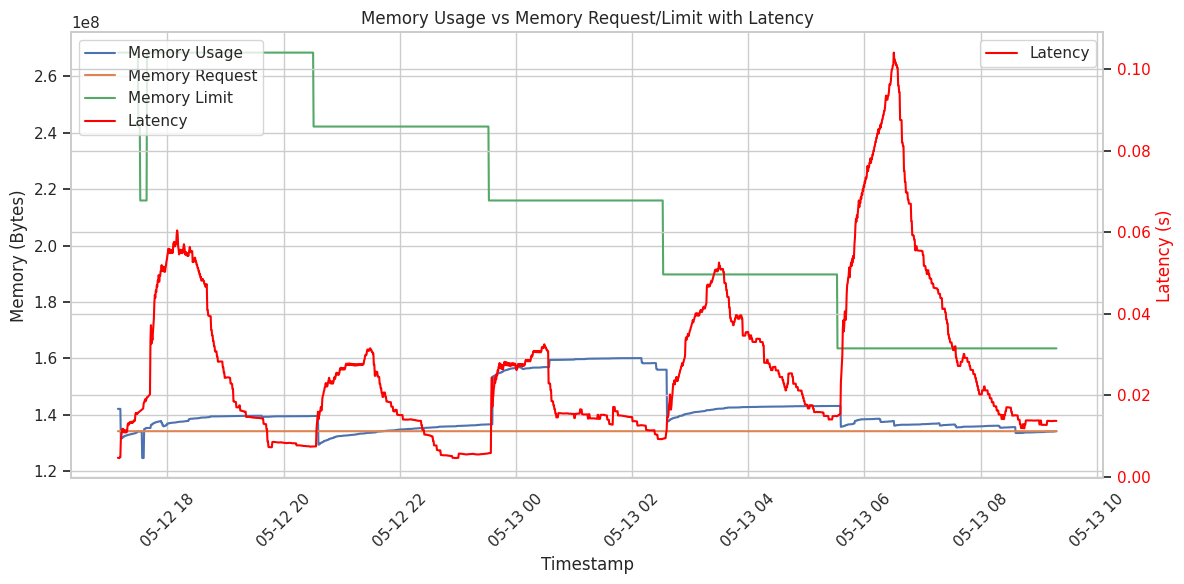
* Our random password generator:
  + Creates strings and arrays with some cryptographic entropy.
  + These objects are short-lived (transient strings), perfect for young gen GC.
* Every few seconds:
  + Requests accumulate allocations -> heap fills -> GC -> memory drops.
* The usage is naturally cyclical due to the structure of the work and object lifespan.

1. GC Frequency and Heap Tuning

* The JVM performs GC based on thresholds in Eden, Survivor, and Old generations.
* As memory limits are reduced, the frequency of GC can:
  + Increase (less space, GC triggers earlier),
  + Or cause larger spikes if GC is delayed due to CPU contention or slow promotion.
* The frequency of the memory usage waves matches the GC rhythm, adapting to memory constraints.

# RandPwGen Service Both Resource Limits Reduction





* Latency Increases in clear waves corresponding to moments of load, peaking sharply after ~05-13 06.
* Memory Usage follows a cyclical wave pattern mirroring the latency pattern.
* Latency again, increases during peak memory usage - especially when memory is closer to or briefly exceeds request levels.

Latency Analysis

1. Latency Shows Periodic Spikes Matching Workload Activity

* Each latency peak aligns with increases in both CPU usage and memory usage.
* This suggests the app’s password generation has periodic bursts of compute and allocation.

1. CPU is Not Fully Utilized - Not the Main Bottleneck

* CPU usage is well below limit, even during latency spikes.
* This means latency isn't from CPU throttling. Instead:
  + CPU might be idle waiting on GC or memory.
  + Or workload is not CPU-intensive, but memory allocation-heavy.

1. Latency Spikes Strongly Correlate With Memory Usage Peaks

* When memory usage surges, latency also rises dramatically.
* After ~05-13 06, a latency peak to ~0.10s occurs as:
  + Memory usage spikes near or slightly above the reduced limit.
  + Suggests GC pressure, heap resizing, or allocation delays are stalling the service.

1. Latency Decreases Right After Memory Limit Is Relaxed

* Right after each memory limit plateau is hit, latency tends to fall again.
* This confirms that memory headroom relieves pressure and reduces service delay.

Summary Table

|  |  |  |
| --- | --- | --- |
| Metric | Behaviour | Latency Impact |
| CPU Limit | Reduced but still sufficient | Not the bottleneck |
| CPU Usage | Low and stable | Correlates, but not causal |
| Memory Limit | Gradually reduced | Correlated with spikes |
| Memory Usage | Wavelike pattern, near limits | Directly tied to latency |
| Latency | Periodic spikes, strongest during high memory usage | GC/allocation delays likely |

## Explanation of Latency Reduction After 05-12 22:

Observations at 05-12 22:

* CPU and memory limits are lower than earlier stages.
* But latency drops significantly, becoming lower than 05-12 18 to 22.
* CPU usage and memory usage show a more stable and smoother pattern.
* No aggressive spikes in latency despite lower resources.

Reasons for Latency Reduction

1. Garbage Collection (GC) Stabilization

* Java applications often exhibit high latency due to GC pressure under higher memory limits.
* Earlier phases might have allowed excessive memory allocation -> more frequent or full GC.
* After memory limits were reduced, the JVM may have started allocating more conservatively, leading to:
  + Fewer object promotions to old gen
  + Shorter GC pause times
  + More frequent minor GCs, but less disruptive.

1. CPU Cache Efficiency

* With less CPU available, the JVM and OS scheduler:
  + Schedule threads more predictably.
  + Reduce CPU thrashing from excessive parallelism or context switching.
* This may improve instruction cache hit ratio or thread locality, indirectly reducing latency.

1. Load Flattening / Stabilization

* The load is constant (10–20 req/s) overall, but actual request distribution have been more bursty earlier (surges).
* Post-05-12 22, the load became more uniform, the app:
  + Better utilize CPU without sudden pressure.
  + Keep queues short -> lower response time.

1. Memory Limit Shrinkage Preventing Over-Allocation

* JVM over-allocation under high memory limits  cause:
  + High GC time.
  + Memory fragmentation.
* Reduced limits have disciplined the allocator, resulting in:
  + Lower memory usage variance
  + Tighter heap layout
  + Faster memory access (less paging, no major GC triggers).

1. Background Optimization Effects

* JVM’s adaptive optimizations (JIT compiler, tiered compilation) have kicked in later:
  + Heavily used code gets compiled to native over time.
  + Optimizations such as inlining, loop unrolling, or escape analysis improve runtime performance.
  + These would be more visible after long uptime, around the 05-12 22 mark.

1. Kubernetes Pod Stabilization

* Early phases have included pod CPU throttling, cold start effects, or background init tasks.
* After 05-12 22, the container be fully stabilized, and the **Linux CFS scheduler** more fairly assigns CPU slices.
* CPU limits enforced properly by Kubernetes -> less preemption-> consistent performance.