## Service 2 Only CPU Limit Reduction





I tried the following kubernetes command to find what exactly happens at particular timestamps:

“kubectl get events --sort-by='.lastTimestamp'”

Then, the output is like below:



This describes:

1. No Warnings or Failures

There are no **Warning** or **Failed e**vent types (e.g., CrashLoopBackOff, OOMKilled, Unhealthy) in this output. That rules out a crash or eviction directly causing the latency drops.

1. All Events are Normal

Most are related to CronJob execution:

* *reduce-cpu-request-*, reduce-mem-\* jobs being scheduled, pulling images, starting containers, and completing successfully.
* Other events show Deployments scaling up and down and some pods being killed normally (expected behavior during deployment updates).

### Spike at 05-11 22

* Confirmed as the point where CPU limit is reduced, and:
  + We are seeing a momentary latency spike due to throttling kicking in abruptly.
  + Event log shows reduce-cpu-request-s1-cronjob executing here - that aligns with the spike timing.
* Spike is expected and aligns with CPU throttling. No pod crash or GC pause—just the CPU limit drop and system adjusting.

### Latency Drops near 05-12 00:00 and 05-12 04:00

* No Kubernetes events around those exact times. That suggests it wasn't caused by a pod restart, image pull, or crash.

1. Synthetic Load Generator Reset (External Cause)

During this time, Locust Restarted with the reduction of CPU limit.

* It is restarted, paused & finished a round.
* During this time:
  + No/very few requests are received → measured latency approaches 0.
  + After the tool resumes, latency returns to the throttled baseline.

1. Latency Recording Artifact

The latency is being measured externally using Prometheus scrape interval, then:

* A momentary gap in recording or lack of requests can register as 0 latency.
* This happens due to:
  + Data point interpolation when request count is very low

### Then, why there is a very small value for the Latency?

1. Latency is measured only when a request is received

Even if requests are extremely rare (e.g. 1 every 10 seconds), any non-zero request will still generate a latency data point.

So:

* If 1 request arrives in a time window (15s Prometheus scrape interval),
* And that request was handled instantly (e.g., 80 µs),
* Then average latency for that window is ~80 µs - not zero.

So the graph shows a very small but non-zero latency due to a *tiny number* of super-fast requests.

1. Go Echo service is highly responsive at idle

The Go Echo server (being a minimal HTTP handler) has:

* No business logic or blocking IO
* No load at that moment

So under idle or near-idle conditions:

* It responds extremely fast (microsecond-scale).
* This fast response is what you see as "very small latency" - not zero, but almost negligible.

1. Prometheus-style interpolation

We are scraping metrics every 15s:

* Latency metrics are interpolated between timestamps.
* If few or no new requests happen, the rate of change in latency becomes almost flat or dips close to zero.

Latency goes near-zero (but not exactly zero) because one or two fast requests per interval are still being processed by the Go Echo service with minimal delay — and latency is only recorded for those requests.

### Latency Drop Near 05-12 06

* CPU Usage stays low, and CPU Request/Limit remain at reduced levels (from earlier throttle steps).
* No significant jump or change in CPU usage.
* No major GC spike or allocation burst.
* So the cause is not memory pressure.

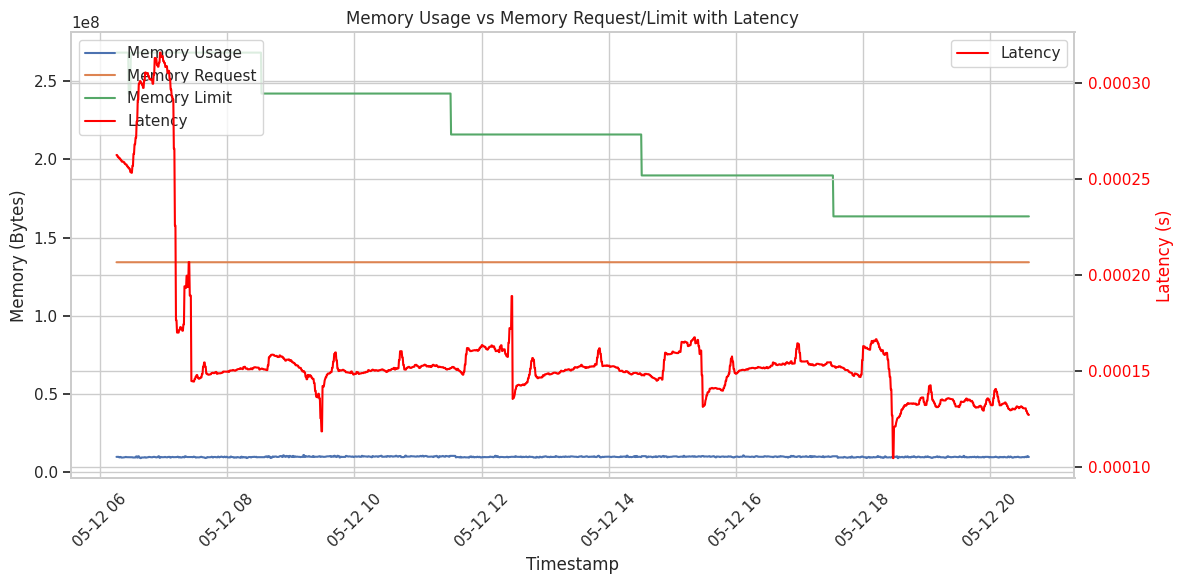
Here, we are finishing our cronjob.

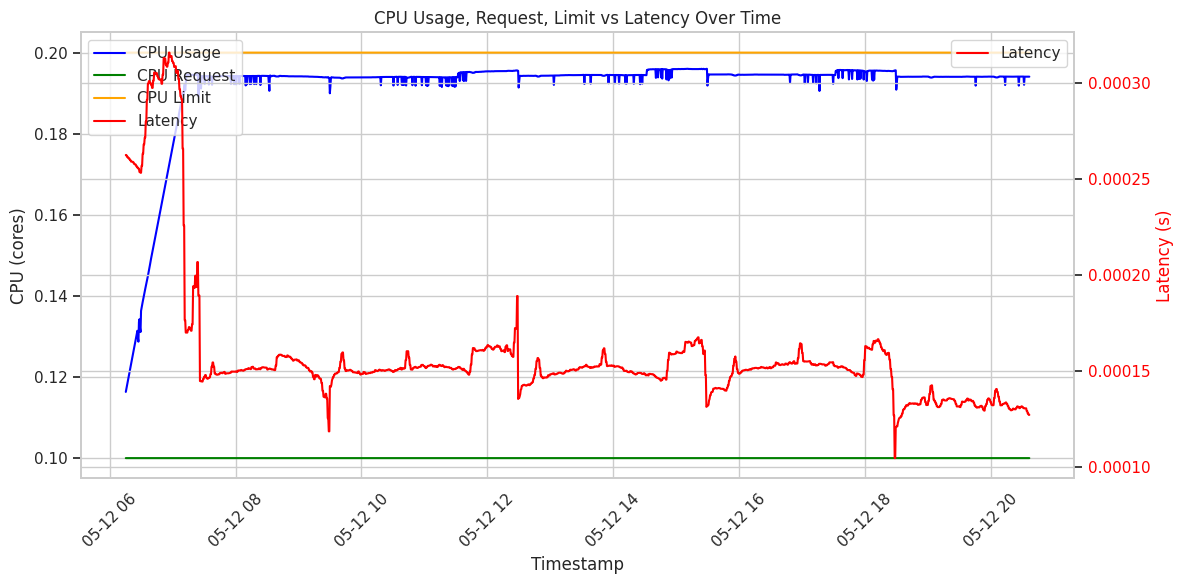
From kubernetes events:

* Multiple CronJobs and Jobs executed.
* Pods started, completed, and cleaned up.

So, one cronjob (service 2 cpu limit reduction) was terminated. That would reduce or eliminate incoming traffic -> latency drops.

## Service 2 Only Memory Limit Reduction





* As memory limits are reduced (step-wise), latency remains nearly flat, indicating that the application is not memory-intensive.
* The service likely fits comfortably within even the lowest memory limits tested here as memory usage is very low.
* So, no risk of memory pressure
* Since CPU limits were not reduced, CPU usage remains stable and fully satisfies the service’s compute needs.
* This is why we don’t observe any CPU-induced latency.
* Unlike some Java or Python services that show latency spikes with memory reduction (due to GC & memory fragmentation), the Go Echo service maintains consistent latency—this suggests:
  + Efficient memory access patterns
  + Minimal reliance on memory buffers or caching
  + Little or no garbage collection pressure

Why latency is reduced after 05-12 18?

1. Post-Warmup Optimization

* The Go Echo service may have undergone:
  + Just-in-time optimization (if compiled with runtime tuning or warm-up behavior).
  + Route caching**,** template compilation, or connection pooling warm-ups.
* These can make the service more efficient after initial traffic.
* Even though Go is compiled, some frameworks (e.g., echo, fiber) do internal boot-time caching.

1. Garbage Collection Settling

* Go uses a garbage collector (GC). Early in execution, memory allocations and GC activity may be higher, causing:
  + Small latency increases due to GC pauses.
  + Eventually, as heap usage stabilizes and the service stops creating many short-lived objects, GC pressure drops.
* After that point, latency smooths out.

1. No Resource Pressure

* At that stage:
  + CPU limits are high and unchanged.
  + Memory limit is still generous.
* The service isn’t being throttled, so it can operate at peak efficiency.

Why reducing memory limit leads to spikes in CPU usage and latency?

1. Go Runtime Adjusts to Lower Heap Space

When we reduce the memory limit, even if the application isn’t using much memory, Go's garbage collector (GC) detects less available headroom.

This leads to:

* More frequent GC runs to stay under the tighter memory cap.
* Short-lived pause-the-world GC phases -> increased latency.
* Higher CPU usage to do those GC cycles more often.

Go's GC is concurrent but still does short "stop-the-world" pauses.

2Container Memory Pressure Triggers GC Sooner

Kubernetes doesn’t kill the pod until memory usage hits the limit, but the Go runtime reads the cgroup memory limits and adjusts its GC behavior proactively.

* If the container has less headroom, GC becomes more aggressive.
* Even with low actual memory usage, the runtime will react defensively.

Go runtime uses the memory limit as a soft signal, not the actual memory used.

3CPU Spikes Are GC + Cache/Allocator Work

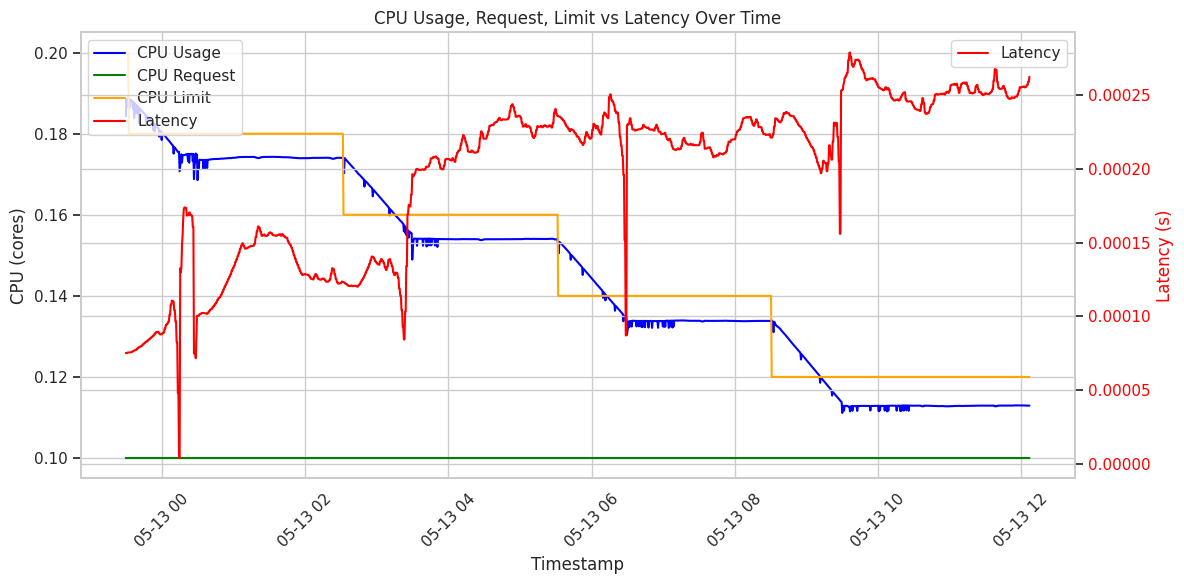
* Lowering memory limits cause memory allocators and caches to:
  + Shrink pools.
  + Clear or reallocate buffer caches.
* This can cause short CPU bursts.

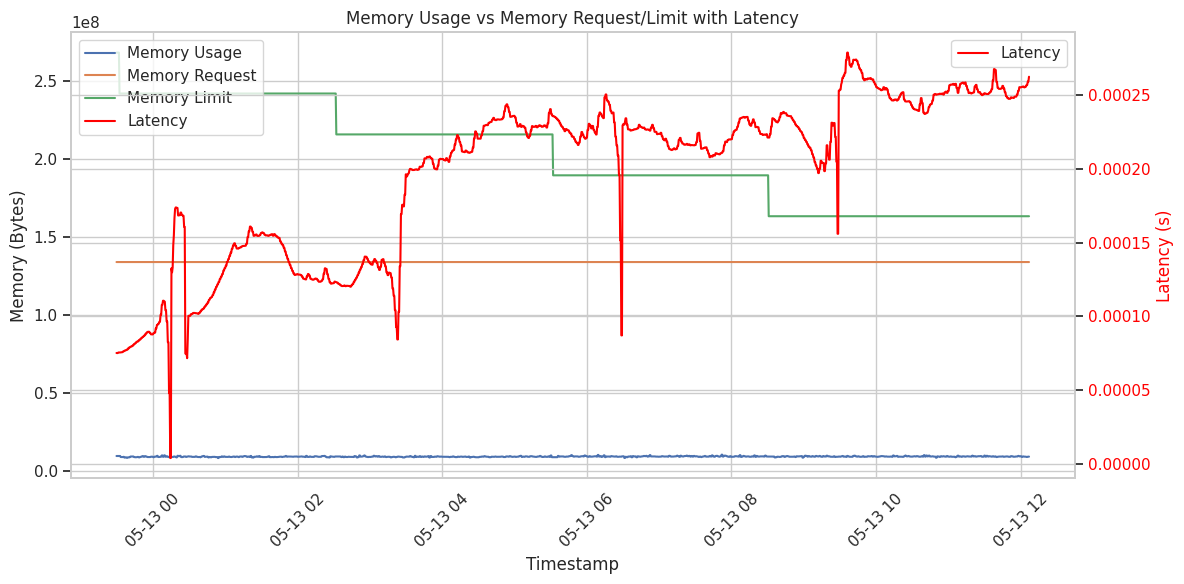
4Latency Spikes from GC or Allocator Throttling

The latency increase is typically not due to the app doing more logic - it’s because:

* GC interrupts request handling briefly.
* Threads get scheduled out or paused during memory adjustments.

## Service 2 Both Resource Limits Reduction





* CPU Usage closely tracks the limit - clearly saturating the available CPU after each reduction.
* Latency rises correlatively with CPU saturation and each limit drop.
* Memory Usage is very stable and consistently below memory limits.
* Latency mirrors the CPU plot—no new behavior is seen here related to memory.

Latency Spikes Strongly Correlate with CPU Limit Drops

Each CPU limit reduction is followed by:

* Immediate CPU usage saturation.
* A clear, stepwise increase in latency.

This repeating pattern is CPU throttling - the application needs more CPU than it’s allowed, so it queues work, resulting in:

* Slower request handling.
* Increased end-to-end latency.

Memory Usage Is Flat and Never Hits Limits

Memory usage is:

* Smooth and stable.
* Not close to the limit at any time.
* Not followed by latency jumps.

Memory is not a constraining factor. Reductions in memory limits have no observed negative impact.

Latency Floor Increases Over Time

* Even between CPU limit drops, latency gradually trends upward.
* Suggests cumulative CPU stress over time:
  + Increased garbage collection time due to CPU constraints.
  + Java threads not getting enough CPU time.
  + Minor queuing effects compounding over minutes.