

## Consecutive Full GC

Our analysis tells that Full GCs are consecutively running in your application. It might cause intermittent OutOfMemoryErrors or degradation in response time or high CPU consumption or even make application unresponsive.

Read our recommendations to [resolve consecutive Full GCs](#)

---

## Application waiting for resources

It looks like your application is waiting due to lack of compute resources (either CPU or I/O cycles). Serious production applications shouldn't be stranded because of compute resources. In 4 GC event(s), 'real' time took more than 'usr' + 'sys' time.

Timestamp	User Time (secs)	Sys Time (secs)	Real Time (secs)
00:02:11	4.47	0.01	5.44
00:02:26	4.84	0.01	5.69

00:02:32	4.36	0.01	5.48
00:02:59	4.65	0.01	5.54

Read our recommendations to [fix this problem](#)

---

## 💡 Tips to reduce GC Time

(**CAUTION:** Please do thorough testing before implementing out the recommendations. These are generic recommendations & may not be applicable for your application.)



- ✓ **5.18%** of GC time (i.e 4 sec 350 ms) is caused by '**Concurrent Mode Failure**'. The CMS collector uses one or more garbage collector threads that run simultaneously with the application threads with the goal of completing the collection of the tenured generation before it becomes full. In normal operation, the CMS collector does most of its tracing and sweeping work with the application threads still running, so only brief pauses are seen by the application threads. However, if the CMS collector is unable to finish reclaiming the unreachable objects before the tenured generation fills up, or if an allocation cannot be satisfied with the available free space blocks in the tenured generation, then the application is paused and the collection is completed with all the application threads stopped. The inability to complete a collection concurrently is referred to as concurrent mode failure and indicates the need to adjust the CMS collector parameters. Concurrent mode failure typically triggers Full GC..

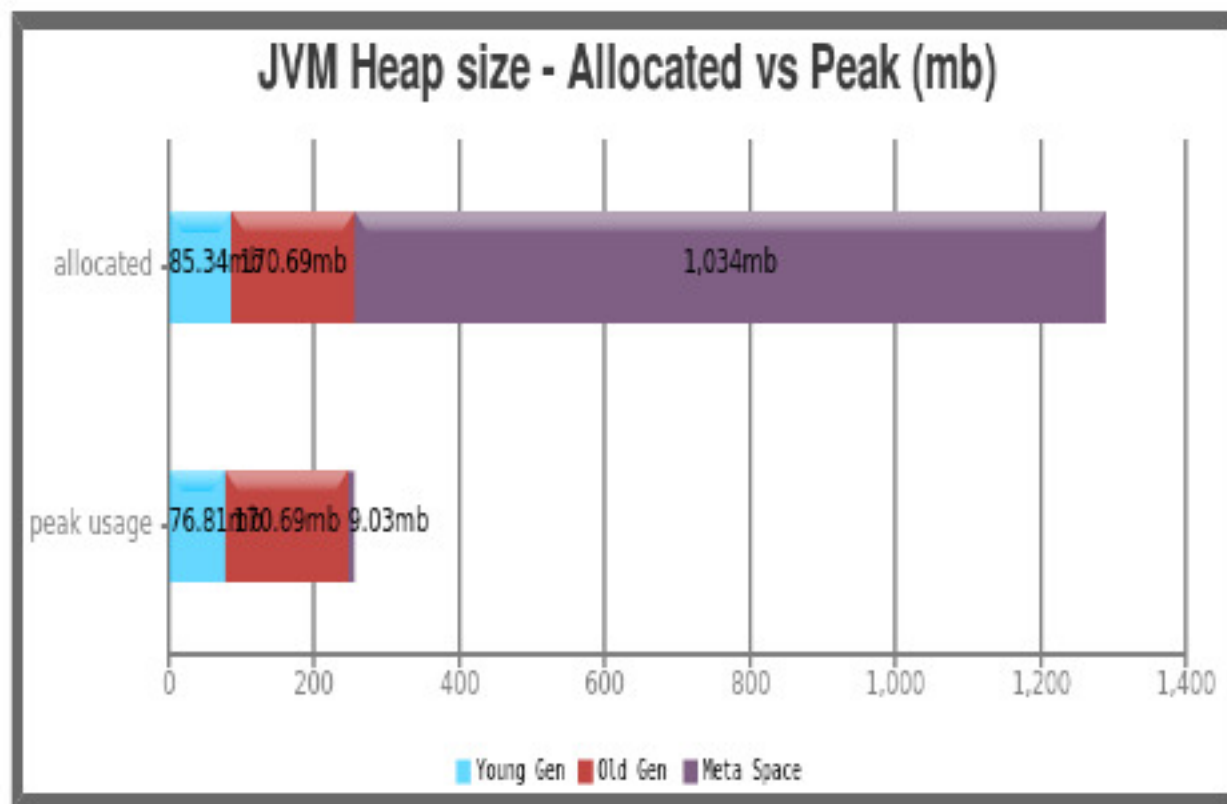
### **Solution:**

The concurrent mode failure can either be avoided by increasing the tenured generation size or initiating the CMS collection at a lesser heap occupancy by setting `CMSInitiatingOccupancyFraction` to a lower value and setting `UseCMSInitiatingOccupancyOnly` to true. `CMSInitiatingOccupancyFraction` should be chosen carefully, setting it to a low value will result in too frequent CMS collections.

---

## JVM Heap Size

Generation	Allocated 	Peak 
Young Generation	85.34 mb	76.81 mb
Old Generation	170.69 mb	170.69 mb
Meta Space	1.01 gb	9.03 mb
Young + Old + Meta space	1.26 gb	256.53 mb

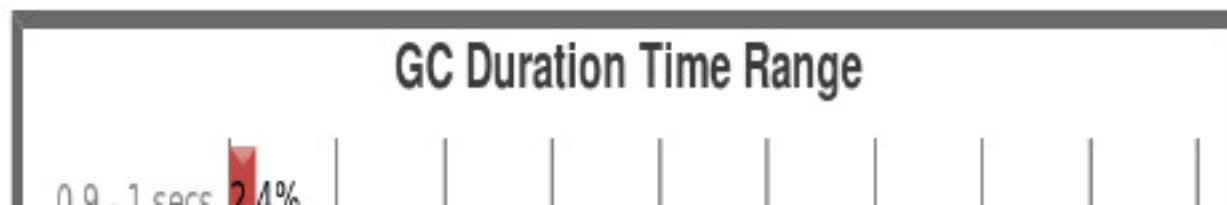


## 🔑 Key Performance Indicators

(Important section of the report. To learn more about KPIs, [click here](#))

1 Throughput  : 94.663%

2 Latency:

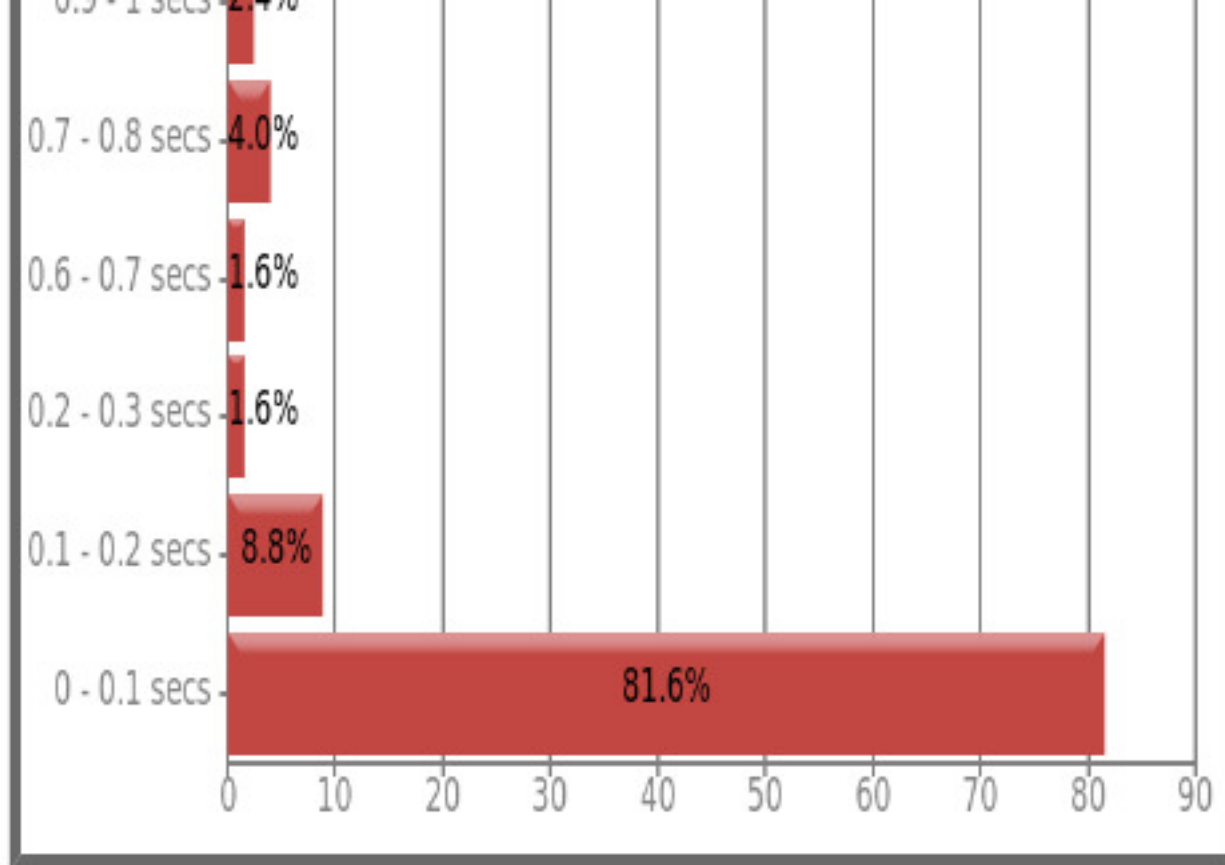


Avg Pause GC Time ⓘ 117 ms

Max Pause GC Time ⓘ 970 ms

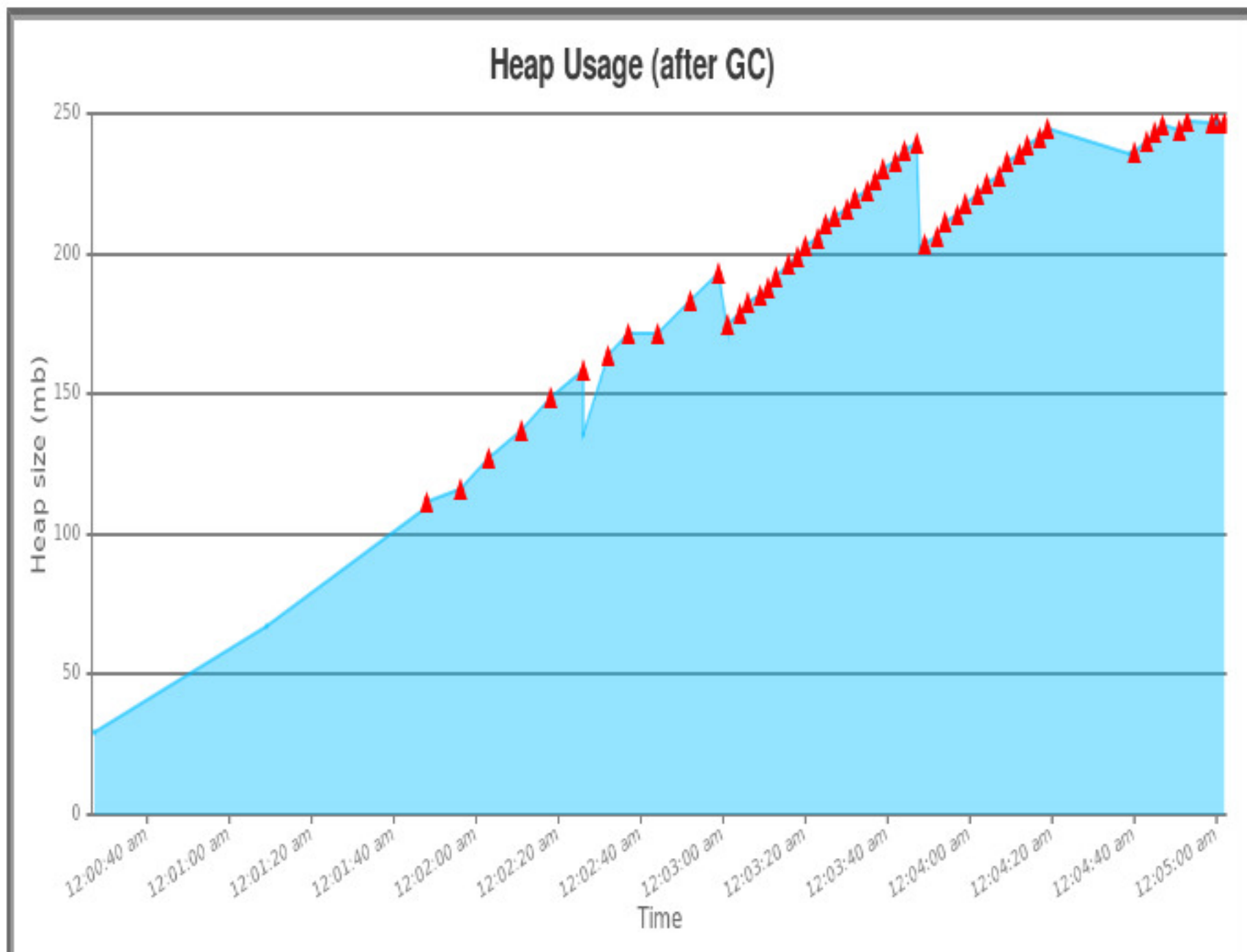
GC Pause Duration Time Range ⓘ:

Duration (secs)	No. of GCs	Percentage
0 - 0.1	102	81.6%
0.1 - 0.2	11	9.0%
0.2 - 0.3	2	1.6%
0.6 - 0.7	2	1.6%
0.7 - 0.8	5	4.0%
0.9 - 1	3	2.4%

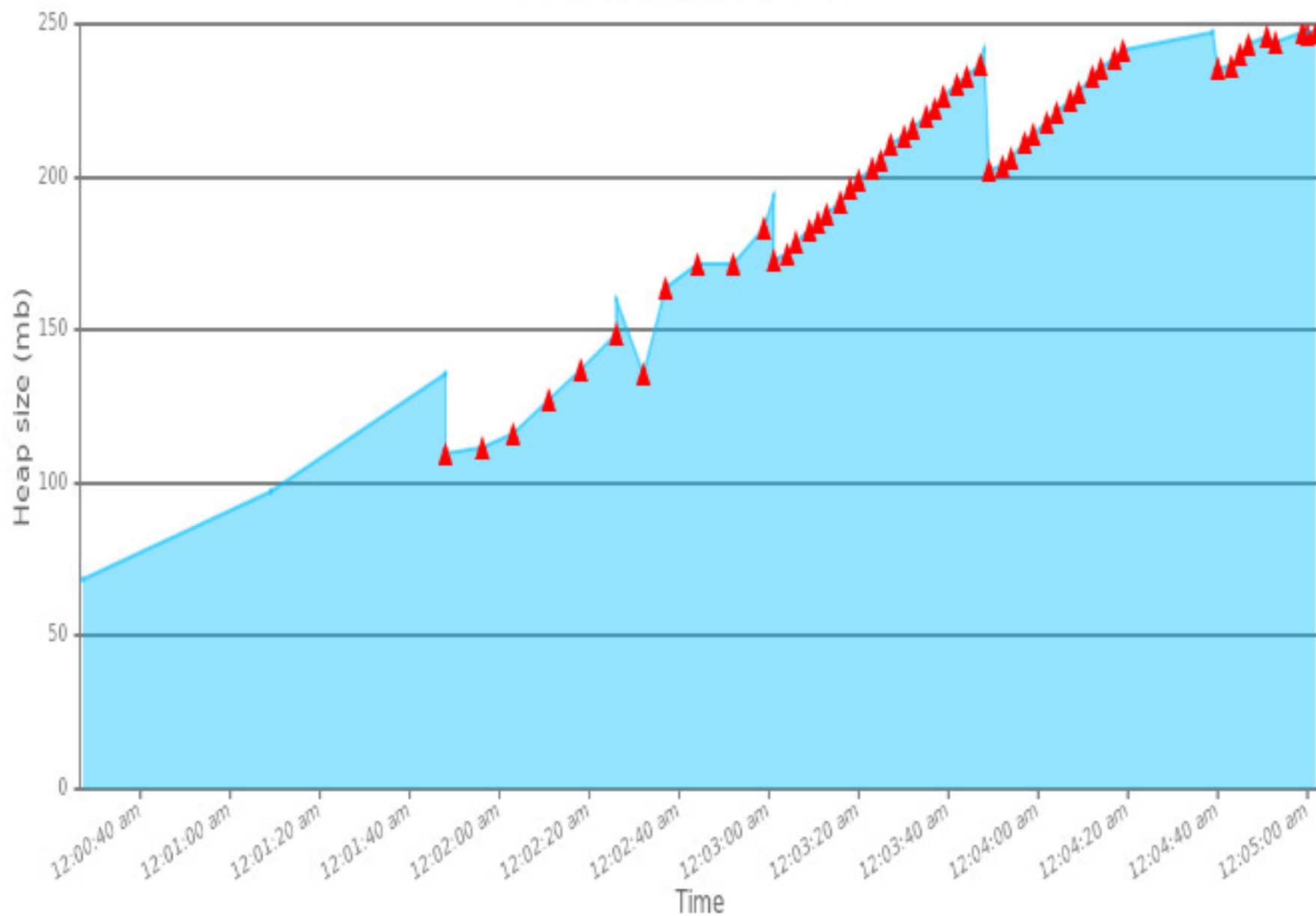


# Interactive Graphs

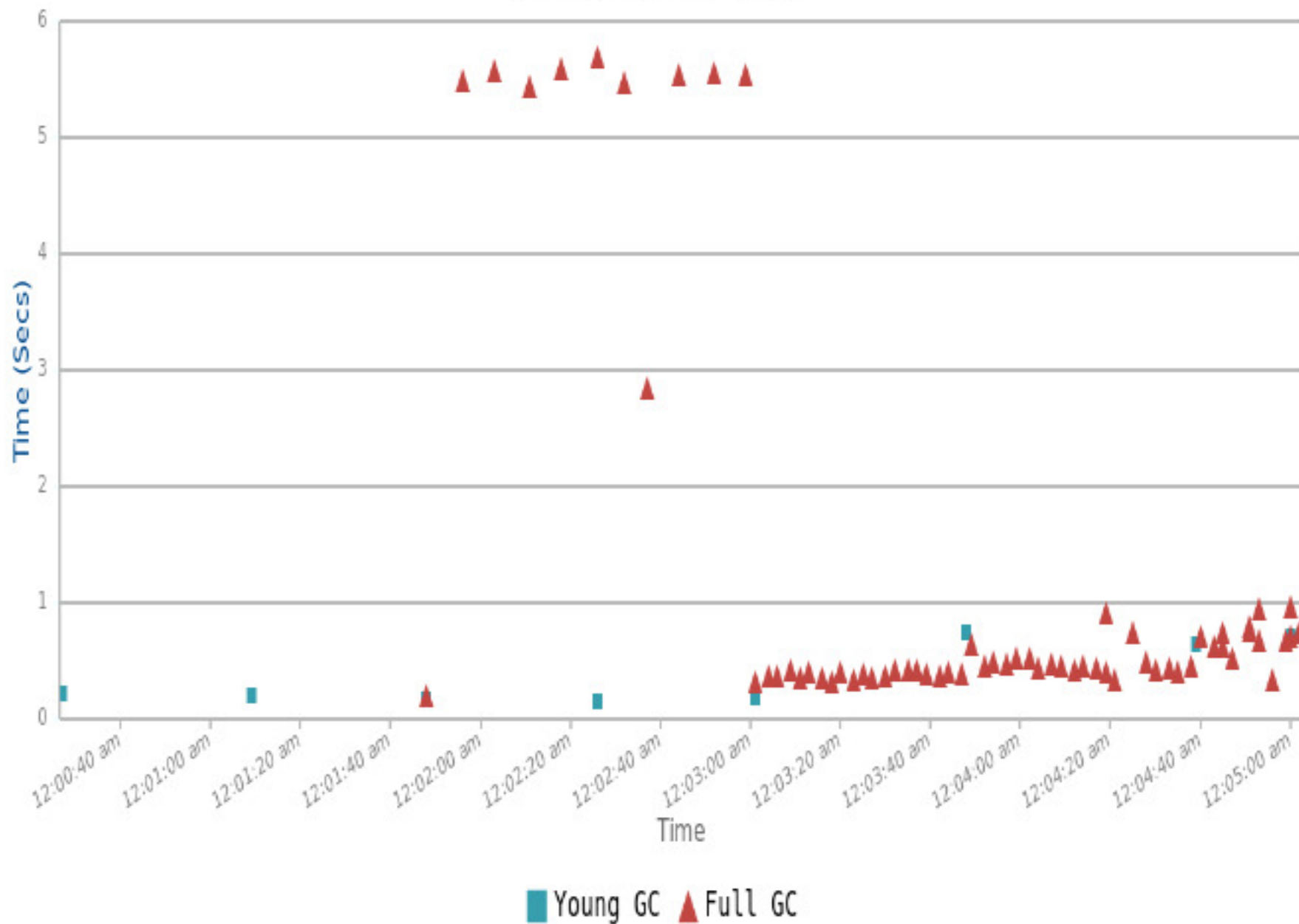
(All graphs are zoomable)



# Heap Usage (before GC)

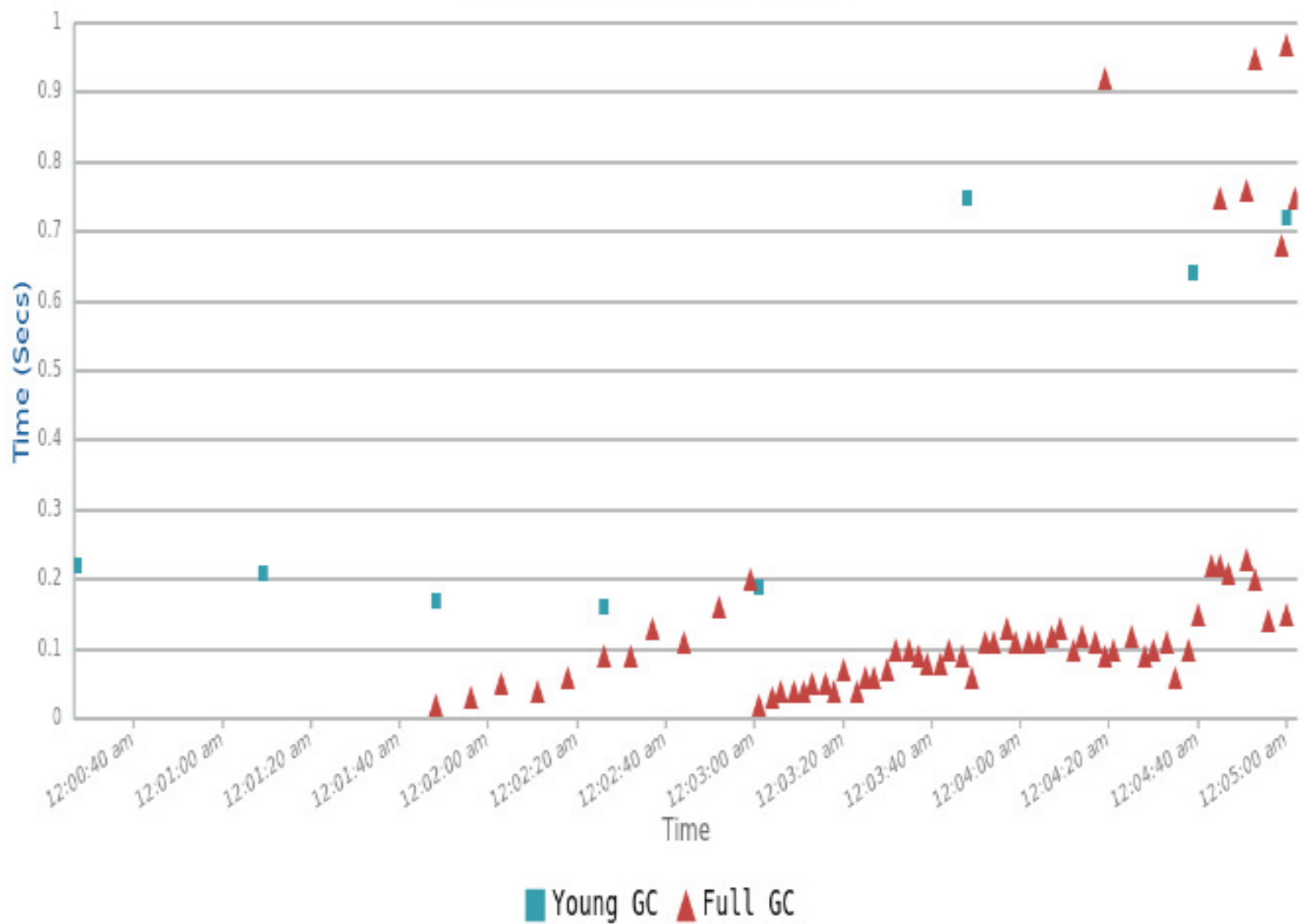


# GC Duration Time



# Pause GC Duration Time

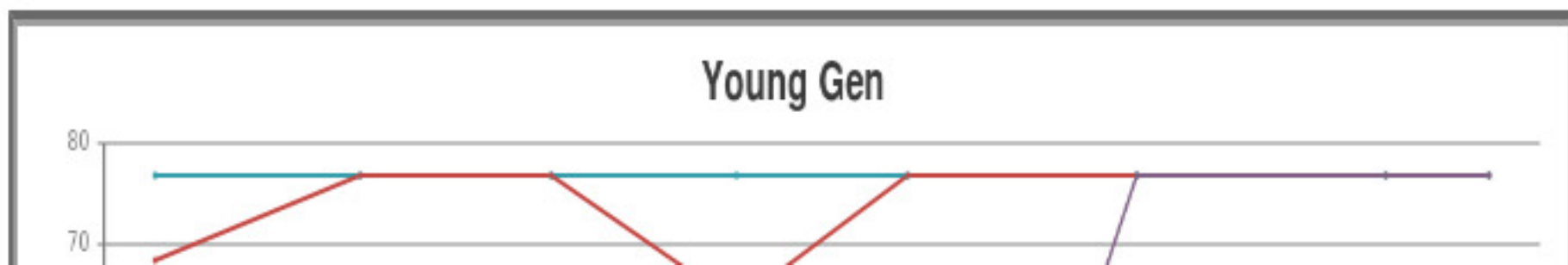
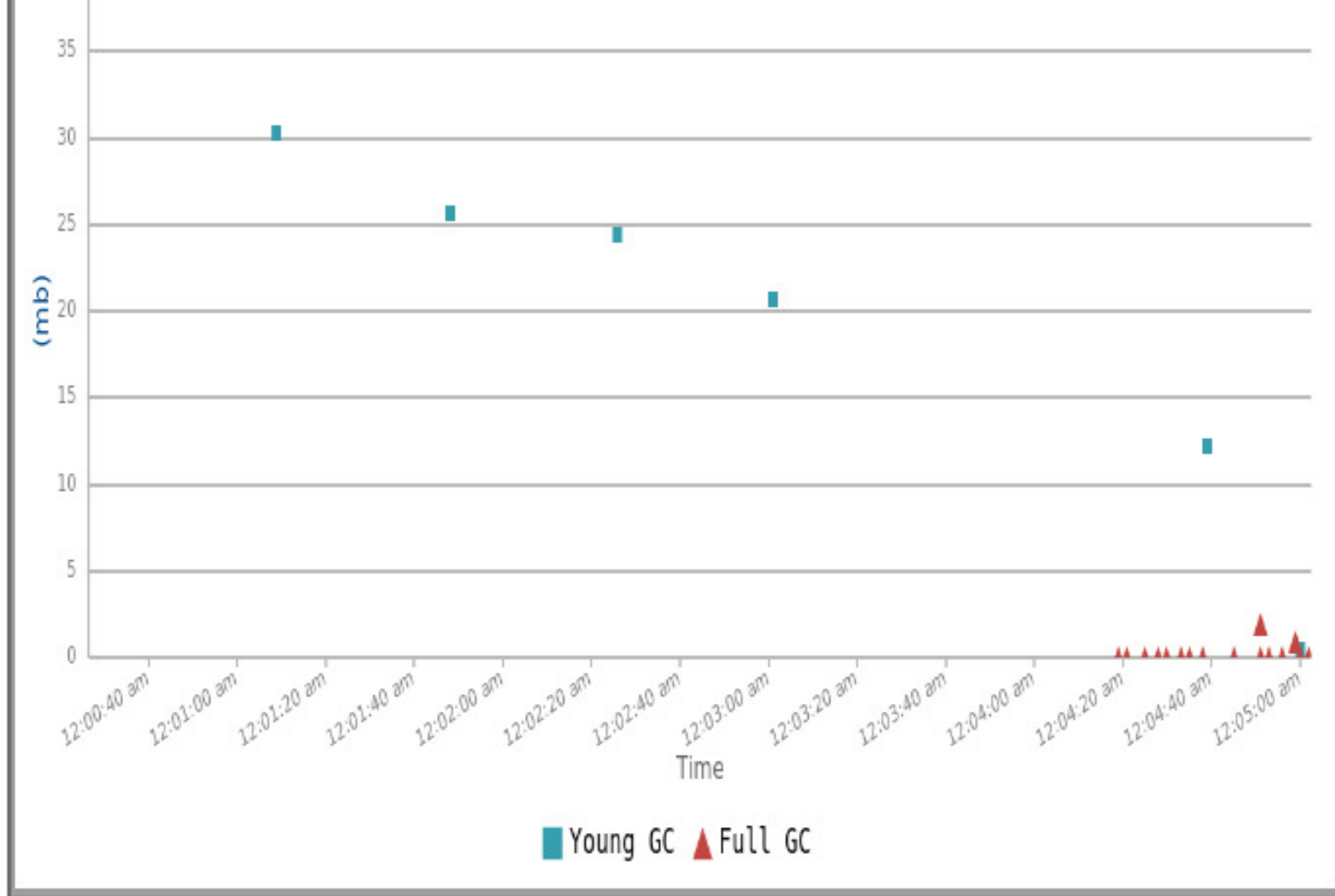
Pause GC Duration Time

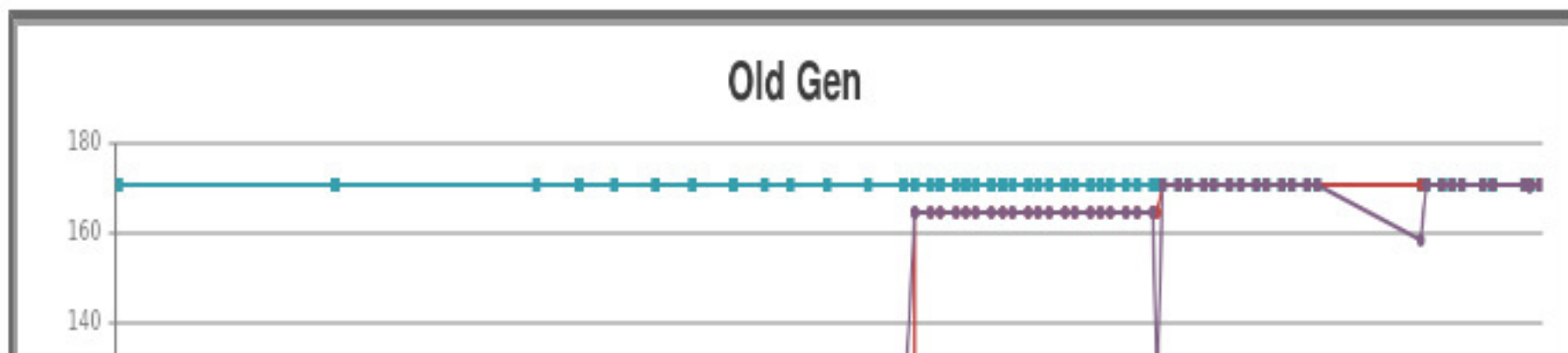
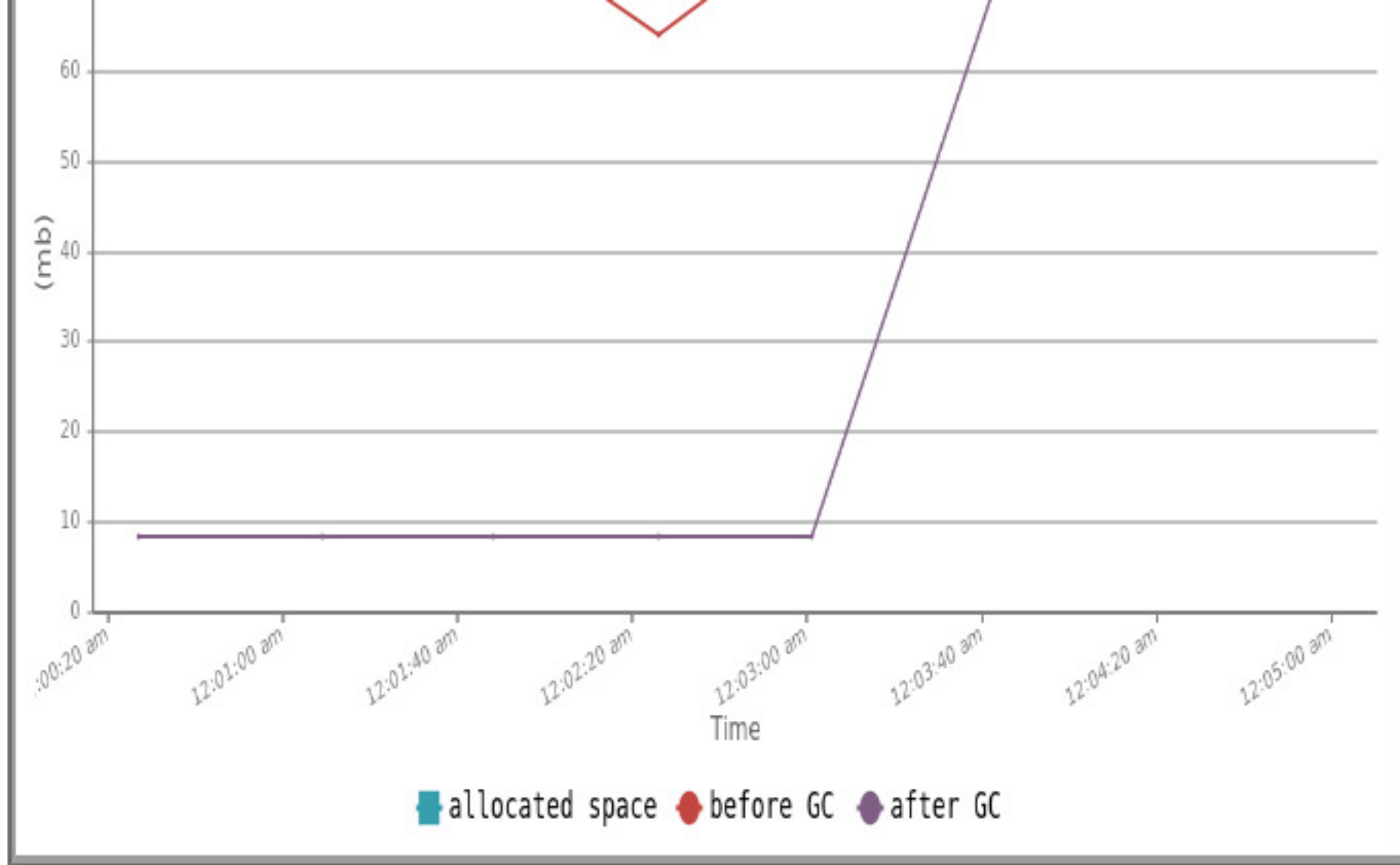


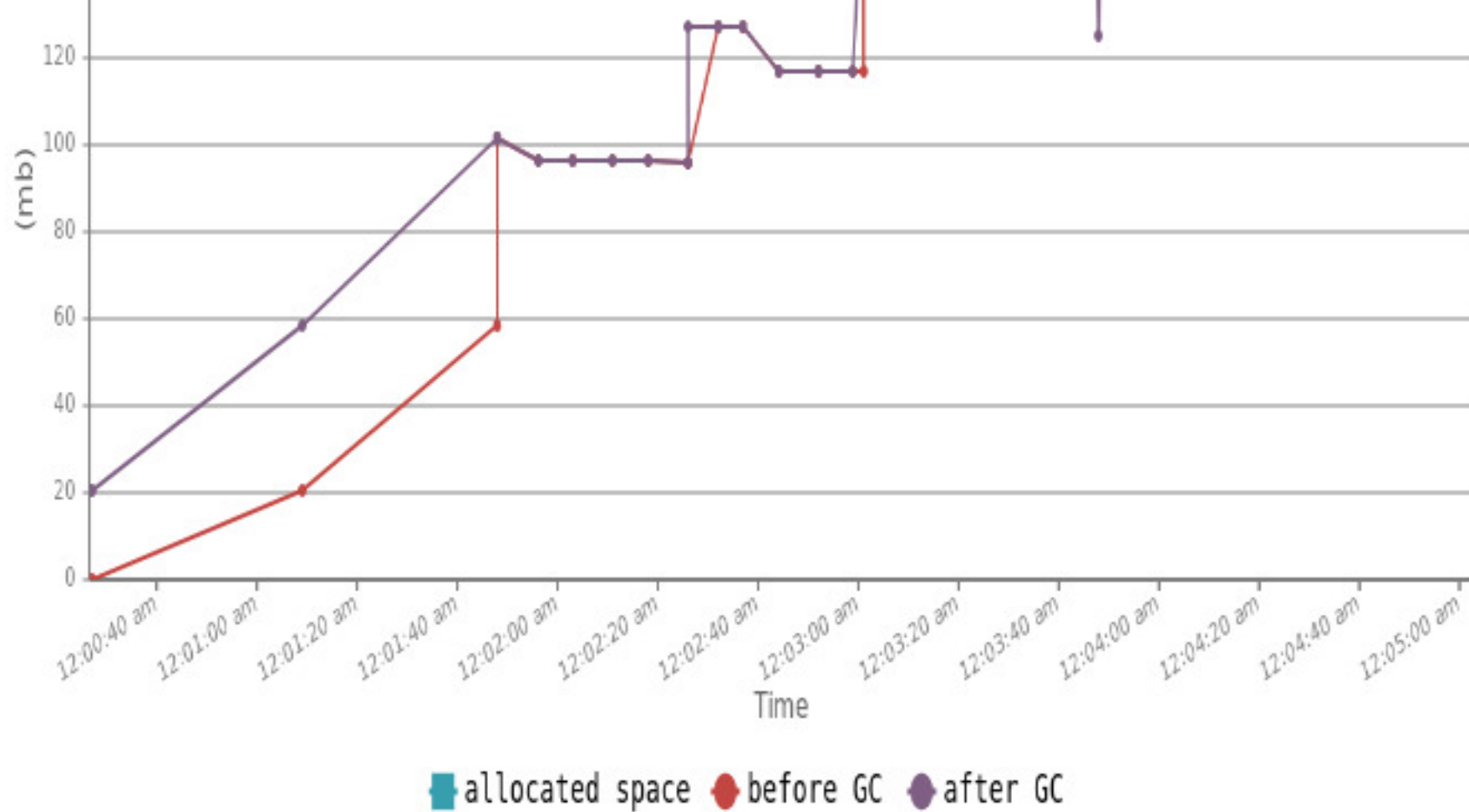
Reclaimed Bytes





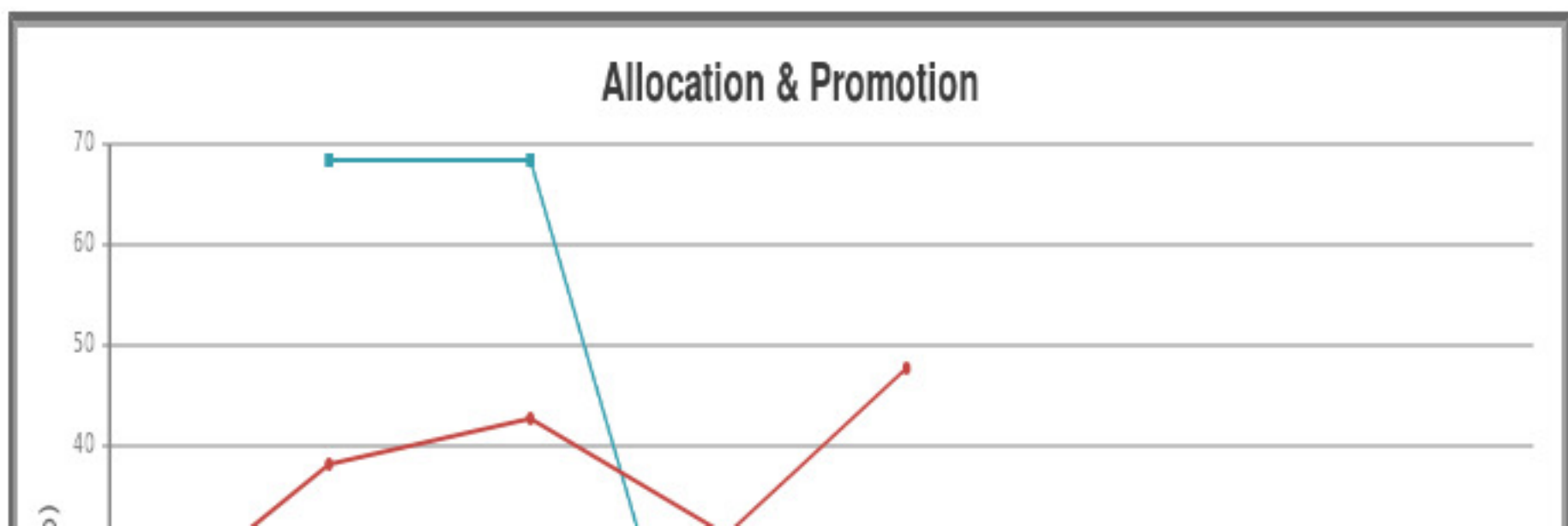
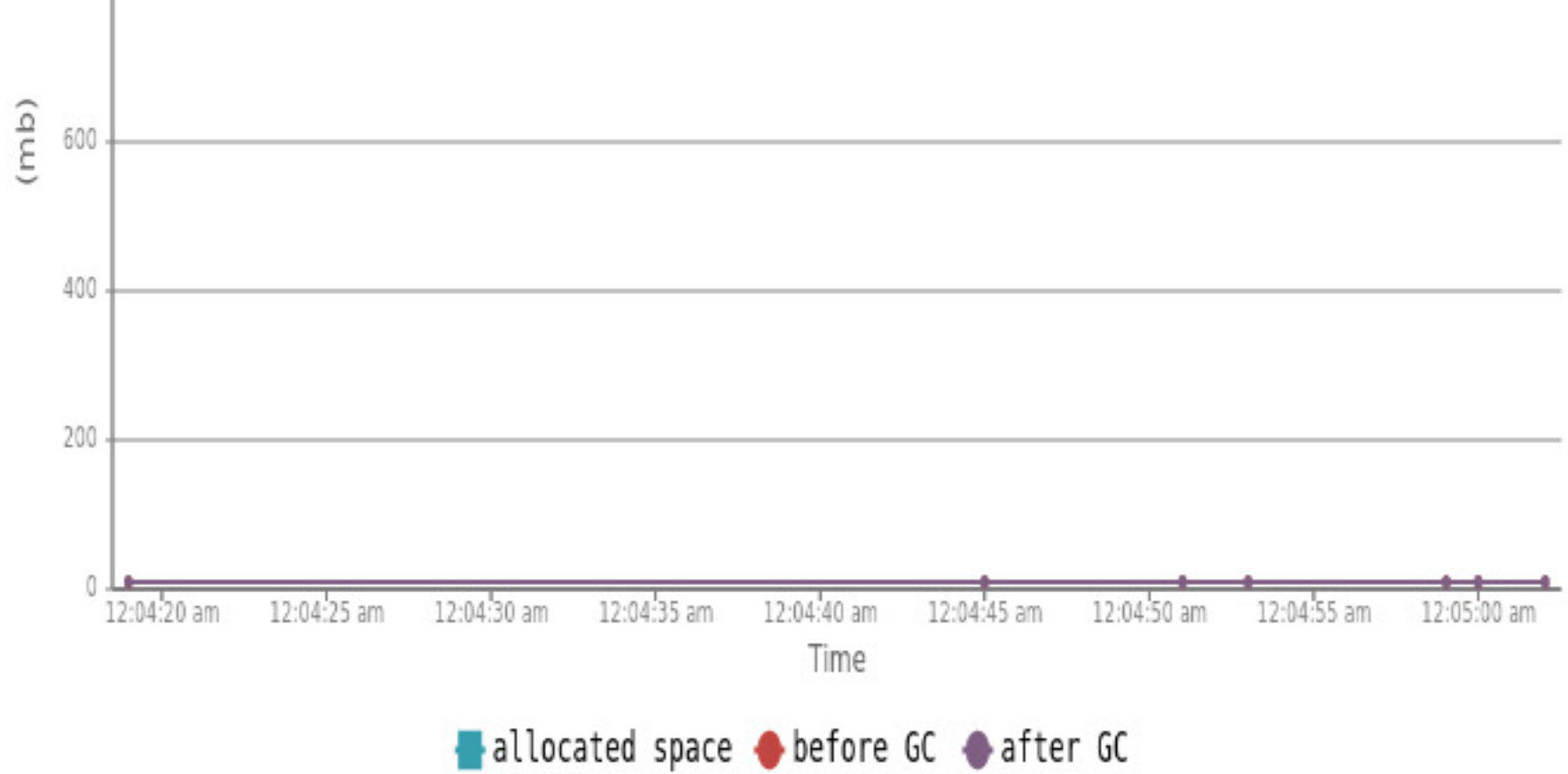






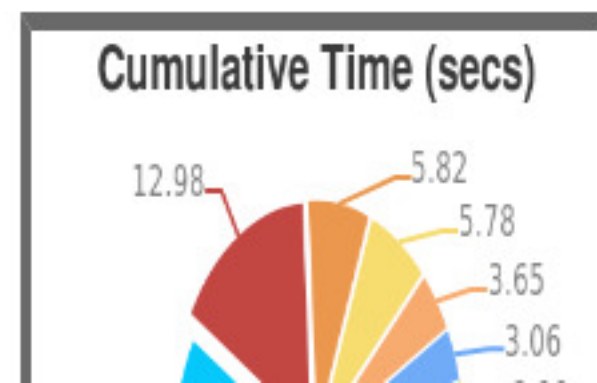
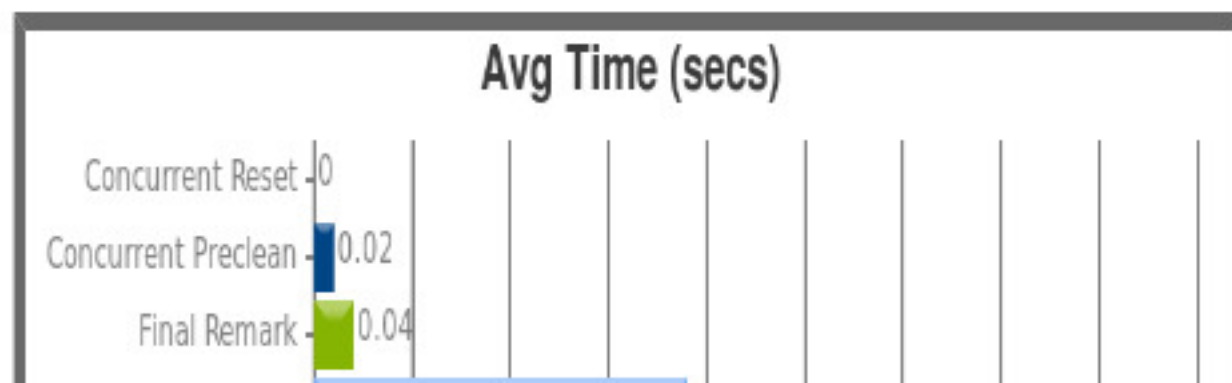
### Meta Space

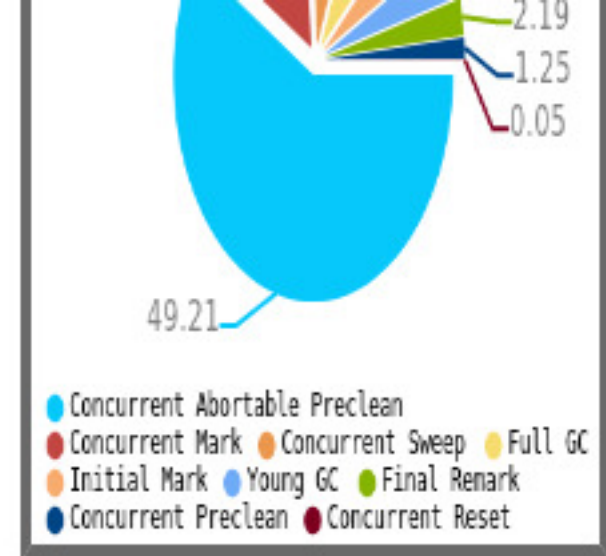
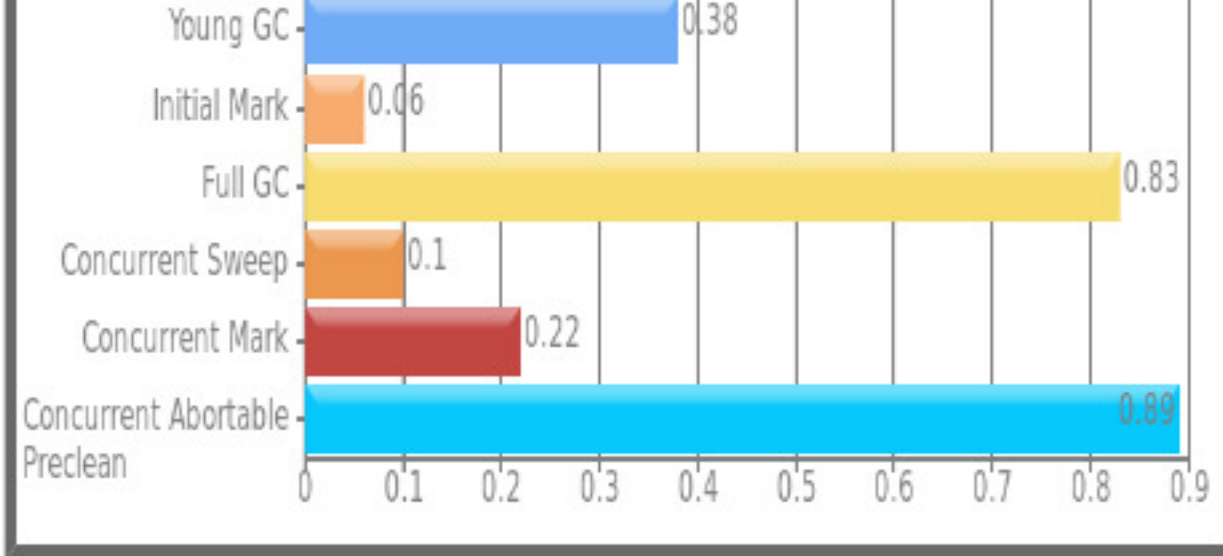






## 🔗 CMS Collection Phases Statistics

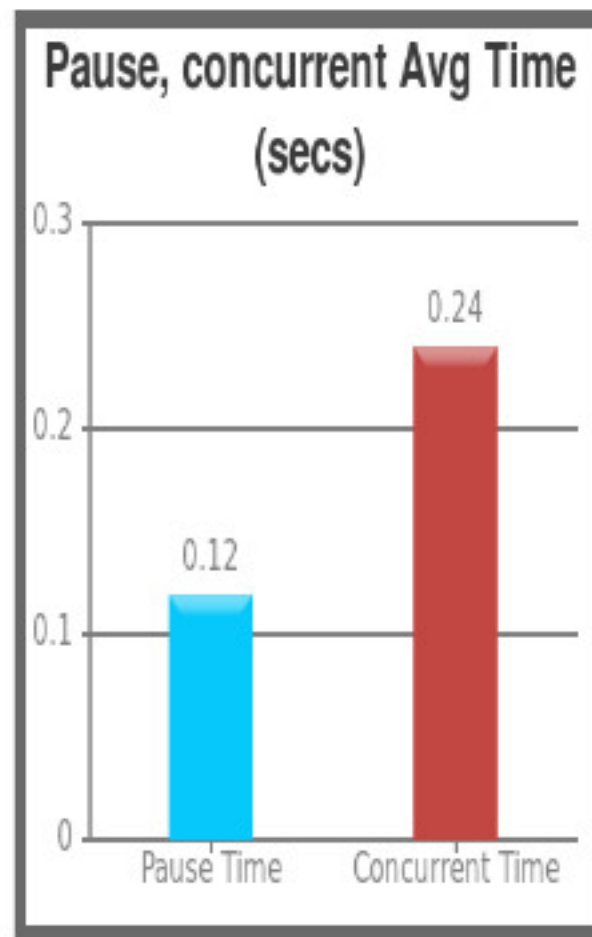
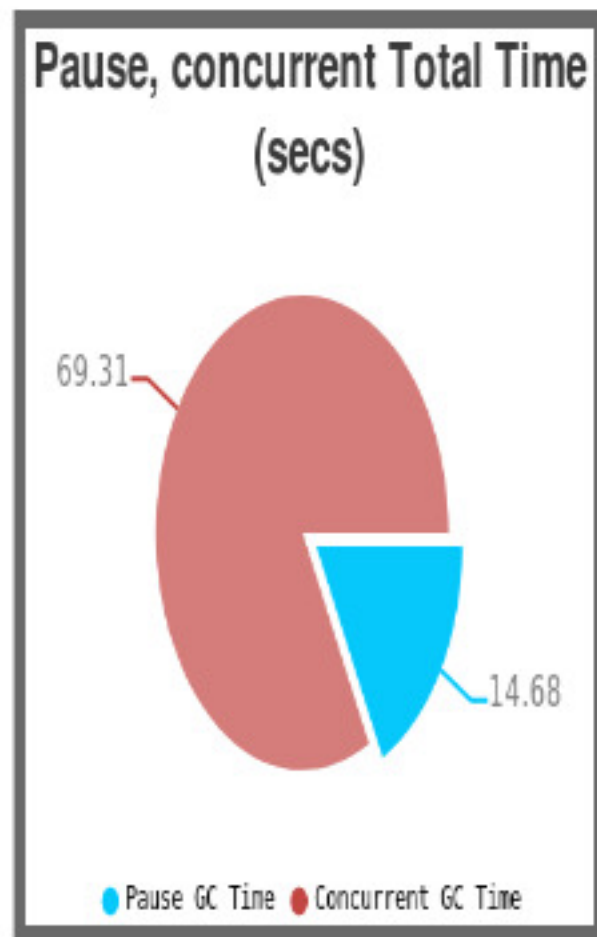




	Concurrent Abortable Preclean	Concurrent Mark	Concurrent Sweep	Full GC	Initial Mark	Young GC	Final Remark	Concurrent Preclean	Concurrent Reset
<b>Total Time</b>	49 sec 210 ms	12 sec 980 ms	5 sec 820 ms	5 sec 780 ms	3 sec 650 ms	3 sec 60 ms	2 sec 190 ms	1 sec 250 ms	50 ms
<b>Avg Time</b>	895 ms	220 ms	102 ms	826 ms	62 ms	383 ms	43 ms	22 ms	1 ms
<b>Std Dev Time</b>	1 sec 931 ms	40 ms	17 ms	108 ms	35 ms	251 ms	25 ms	69 ms	3 ms
<b>Min Time</b>	0	130 ms	60 ms	680 ms	10 ms	160 ms	10 ms	0	0

Max Time ❗	5 sec 380 ms	350 ms	120 ms	970 ms	140 ms	750 ms	100 ms	250 ms	10 ms
Count ❗	55	59	57	7	59	8	51	57	55

## 🔗 CMS GC Time



## Pause Time ?

Total Time	14 sec 680 ms
Avg Time	117 ms
Std Dev Time	205 ms
Min Time	10 ms
Max Time	970 ms

## Concurrent Time ?

Total Time	1 min 9 sec 310 ms
Avg Time	245 ms
Std Dev Time	913 ms
Min Time	0
Max Time	5 sec 380 ms

---

## ⚙ Object Stats

(These are perfect [micro-metrics](#) to include in your performance reports)

Total created bytes ?	209.33 mb
Total promoted bytes ?	180.13 mb
Avg creation rate ?	779 kb/sec



Avg promotion rate 	670 kb/sec
--	------------

---

## Memory Leak

No major memory leaks.

(**Note:** there are [8 flavours of OutOfMemoryErrors](#). With GC Logs you can diagnose only 5 flavours of them (java heap space, GC overhead limit exceeded, Requested array size exceeds VM limit, Permgen space, Metaspace). So in other words, your application could be still suffering from memory leaks, but need other tools to diagnose them, not just GC Logs.)

---

## Long Pause

None.

---

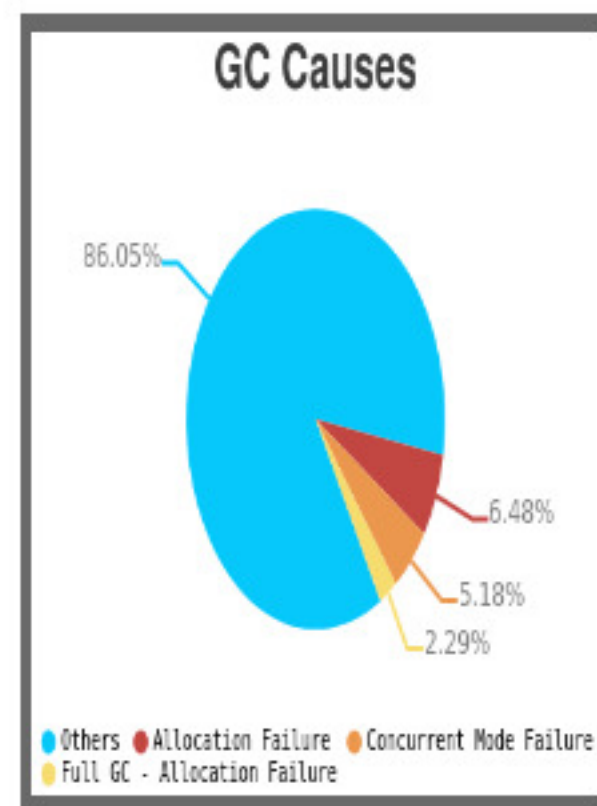
## Safe Point Duration

(To learn more about SafePoint duration, [click here](#))

## ? GC Causes ?

(What events caused the GCs, how much time it consumed?)

Cause	Count	Avg Time	Max Time	Total Time	Time %
Others	54	n/a	n/a	1 min 12 sec 280 ms	86.06%
Allocation Failure ?	11	495 ms	750 ms	5 sec 440 ms	6.48%
Concurrent Mode Failure ?	5	870 ms	970 ms	4 sec 350 ms	5.18%
Full GC - Allocation Failure ?	4	480 ms	750 ms	1 sec 920 ms	2.29%
Total	74	n/a	n/a	1 min 23 sec 990 ms	100.01%



Not reported in the log.

---

## Command Line Flags ?

```
%X:GCLogFileSize=10485760 -XX:InitialHeapSize=268435456 -XX:MaxHeapSize=268435456 -XX:MaxNewSize=89481216 -XX:MaxTenuringThreshold=6  
%X:NewSize=89481216 -XX:OldPLABSize=16 -XX:OldSize=178954240 -XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UseCompressedClassPointers  
%X:+UseCompressedOops -XX:+UseConcMarkSweepGC -XX:+UseParNewGC
```

---

