


GC Intelligence Report

 **gc-poc-tester-SERVER-192.168.1.80-id-0-2023-10-17-04-32-55.log**

 **Duration: 1 hr 36 min 46 sec**

Memory Leak



Our analysis tells that your application is suffering from memory leak. It can cause OutOfMemoryError, JVM to freeze, poor response time and high CPU consumption. Read our recommendations to [resolve memory leak](#)

Poor Throughput



Our analysis tells that your application is spending too much time on GC. **21.25%** of time is spent on GC. Too much GC activity degrades response time + consumes CPU. It's ideal to keep GC time under **10.0%**. Read our recommendations to [increase throughput](#)

Recommendations

(**CAUTION:** Please do thorough testing before implementing below recommendations.)

✔ **20 min 31 sec 24 ms** of GC pause time is triggered by '**G1 Evacuation Pause**' event. This GC is triggered when copying live objects from one set of regions to another set of regions. When Young generation regions are only copied then Young GC is triggered. When both Young + Tenured regions are copied, Mixed GC is triggered..

Solution:

- Evacuation failure might happen because of over tuning. So eliminate all the memory related properties and keep only min and max heap and a realistic pause time goal (i.e. Use only -Xms, -Xmx and a pause time goal -XX:MaxGCPauseMillis). Remove any additional heap sizing such as -Xmn, -XX:NewSize, -XX:MaxNewSize, -XX:SurvivorRatio, etc.
- If the problem still persists then increase JVM heap size (i.e. -Xmx).
- If you can't increase the heap size and if you notice that the marking cycle is not starting early enough to reclaim the old generation then reduce -XX:InitiatingHeapOccupancyPercent. The default value is 45%. Reducing the value will start the marking cycle earlier. On the other hand, if the marking cycle is starting early and not reclaiming, increase the -XX:InitiatingHeapOccupancyPercent threshold above the default value.
- You can also increase the value of the '-XX:ConcGCThreads' argument to increase the number of parallel marking threads. Increasing the concurrent marking threads will make garbage collection run fast.
- Increase the value of the '-XX:G1ReservePercent' argument. Default value is 10%. It means the G1 garbage collector will try to keep 10% of memory free always. When you try to increase this value, GC will be triggered earlier, preventing the Evacuation pauses. Note: G1 GC caps this value at 50%.

✔ **530 ms** of GC pause time is triggered by '**G1 Humongous Allocation**' event. Humongous allocations are allocations that are larger than 50% of the region size in G1. Frequent humongous allocations can cause couple of performance issues:

- If the regions contain humongous objects, space between the last humongous object in the region and the end of the region will be unused. If there are multiple such humongous objects, this unused space can cause the heap to become fragmented.
- Until Java 1.8u40 reclamation of humongous regions were only done during full GC events. Where as in the newer JVMs, clearing humongous objects are done in cleanup phase.

Solution:

You can increase the G1 region size so that allocations would not exceed 50% limit. By default region size is calculated during startup based on the heap size. It can be overridden by specifying '-XX:G1HeapRegionSize' property. Region size must be between 1 and 32 megabytes and has to be a power of two. Note: Increasing region size is sensitive change as it will reduce the number of regions. So before increasing new region size, do thorough testing.

✔ **50.0 ms** of GC pause time is triggered by '**Metadata GC Threshold**' event. This type of GC event is triggered under two circumstances:

- Configured metaspace size is too small than the actual requirement
- There is a classloader leak (very unlikely, but possible).

Solution:

You may consider setting '-XX:MaxMetaspaceSize' to a higher value. If this property is not present already, please configure it. Setting these arguments to a higher value will

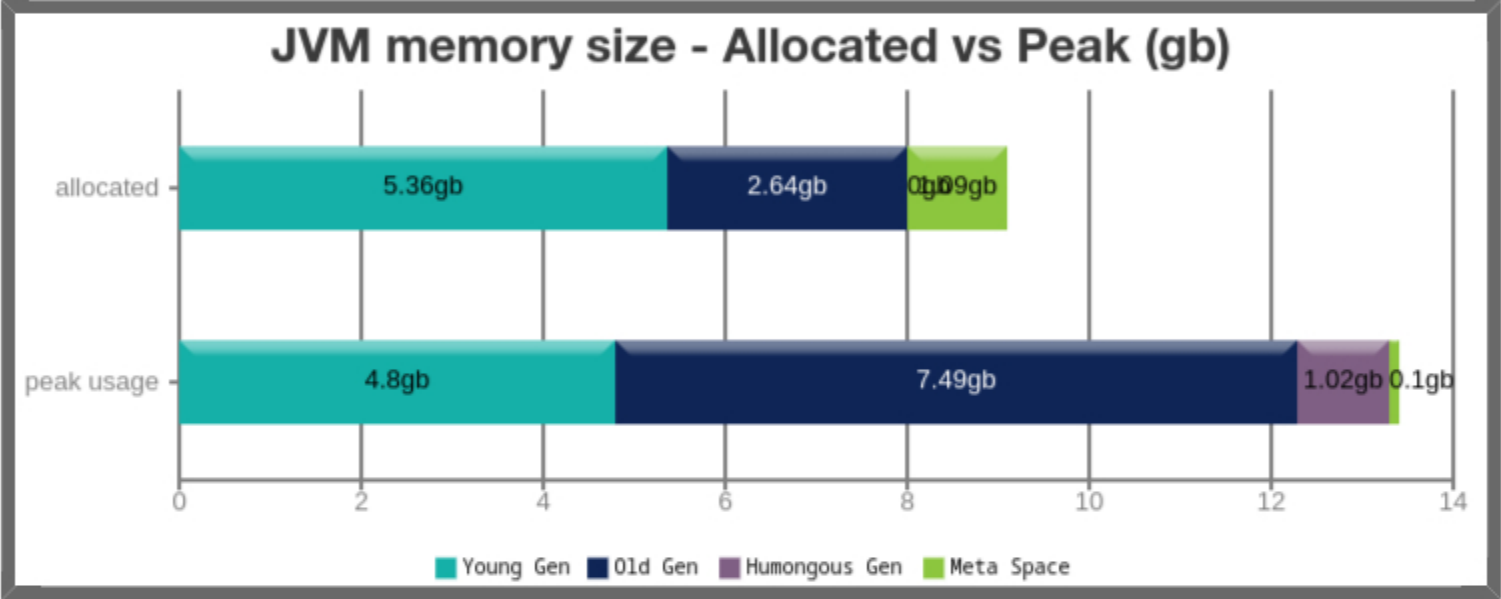
You may consider setting `-XX:MaxMetaspaceSize` to a higher value. If this property is not present already please configure it. Setting these arguments to a higher value will reduce 'Metadata GC Threshold' frequency. If you still continue to see 'Metadata GC Threshold' event reported, then you need to inspect metaspace contents. Learn how to inspect metaspace contents from [this article](#).

- ✔ It looks like you are using G1 GC algorithm. If you are running on Java 8 update 20 and above, you may consider passing `-XX:+UseStringDeduplication` to your application. It will remove duplicate strings in your application and has potential to improve overall application's performance. You can learn more about this property in [this article](#).
- ✔ This application is using the G1 GC algorithm. If you are looking to tune G1 GC performance even further, here are the [important G1 GC algorithm related JVM arguments](#)

☰ JVM memory size

(To learn about JVM Memory, [click here](#))

Generation	Allocated	Peak
Young Generation	5.36 gb	4.8 gb
Old Generation	2.64 gb	7.49 gb
Humongous	n/a	1.02 gb
Meta Space	1.09 gb	103.37 mb
Young + Old + Meta space	9.09 gb	8.1 gb



🔑 Key Performance Indicators

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

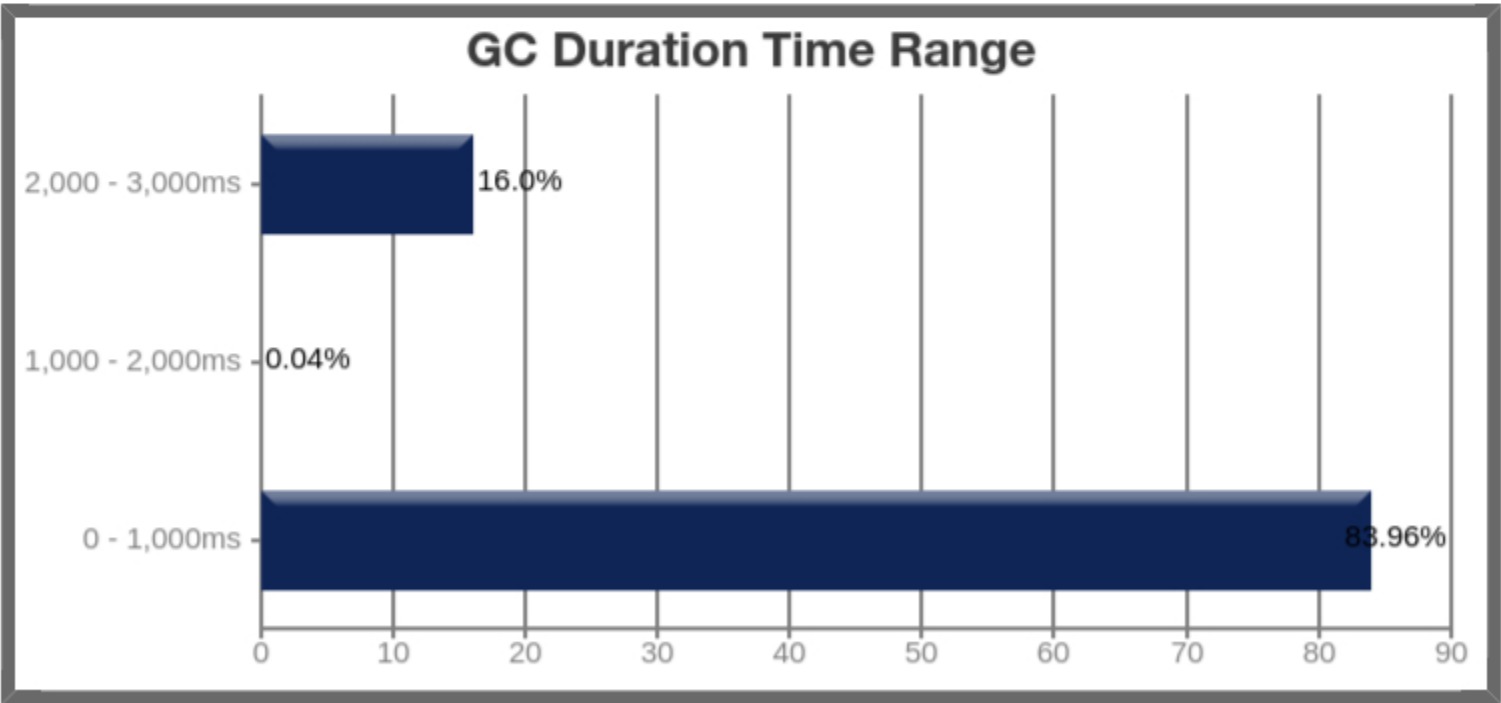
1 Throughput : 78.753%

2 Latency:

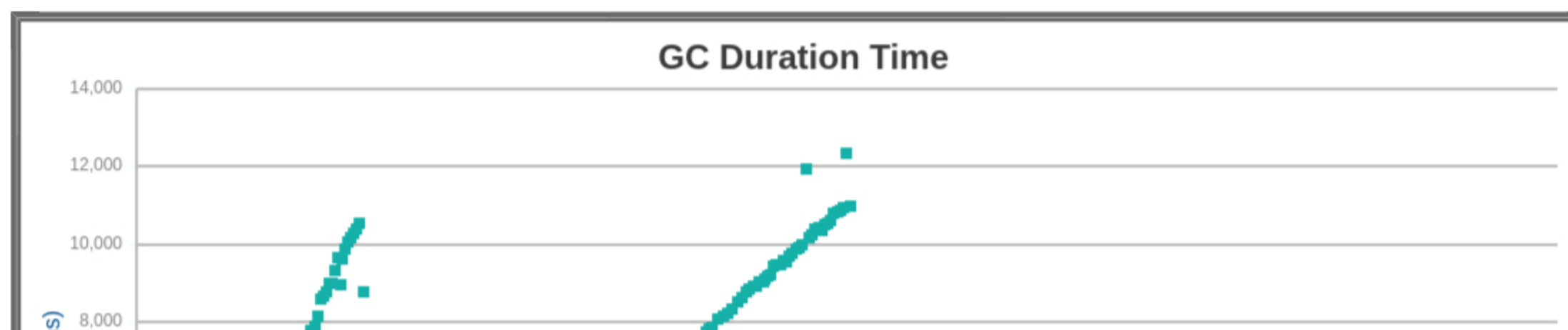
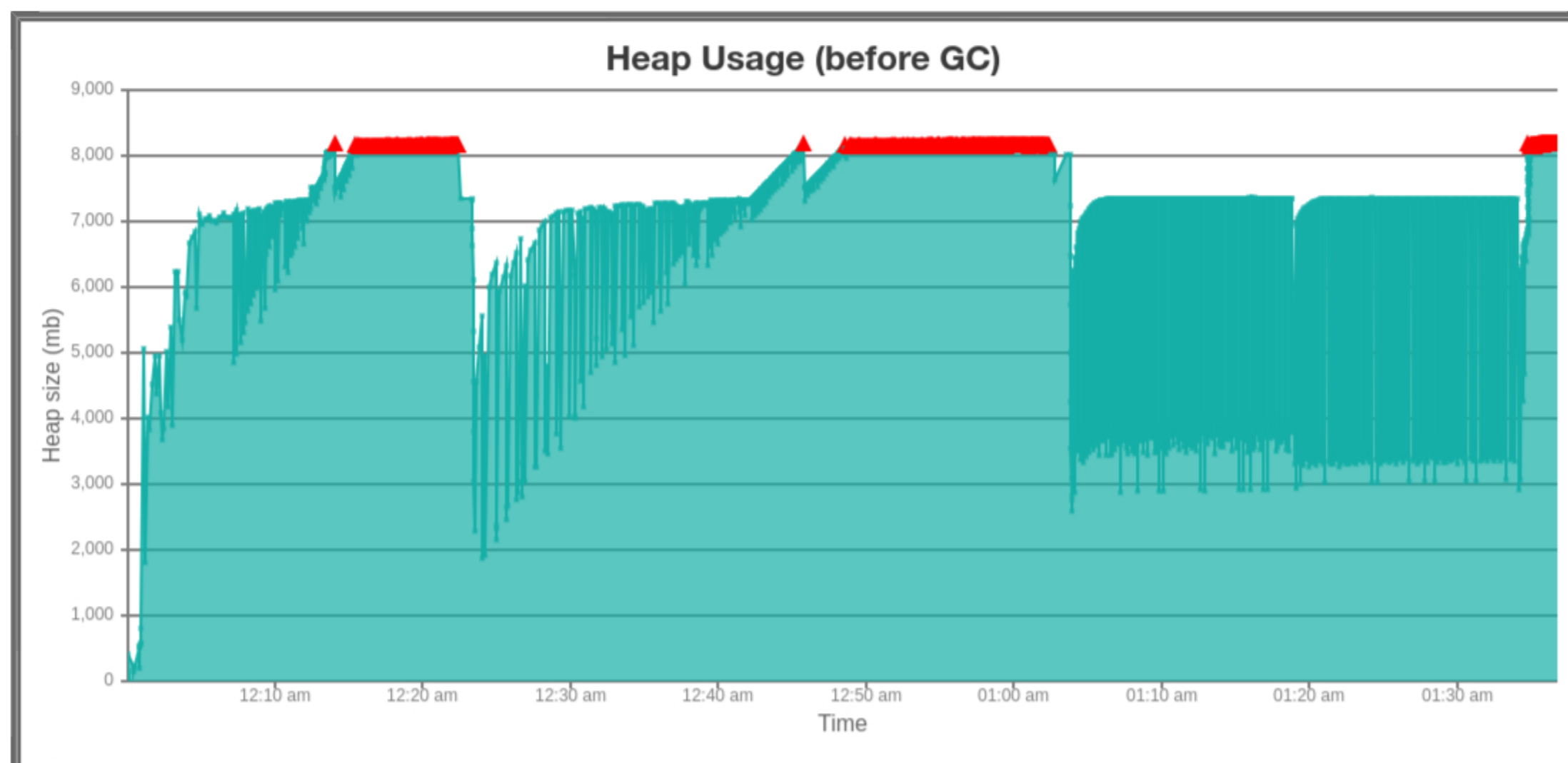
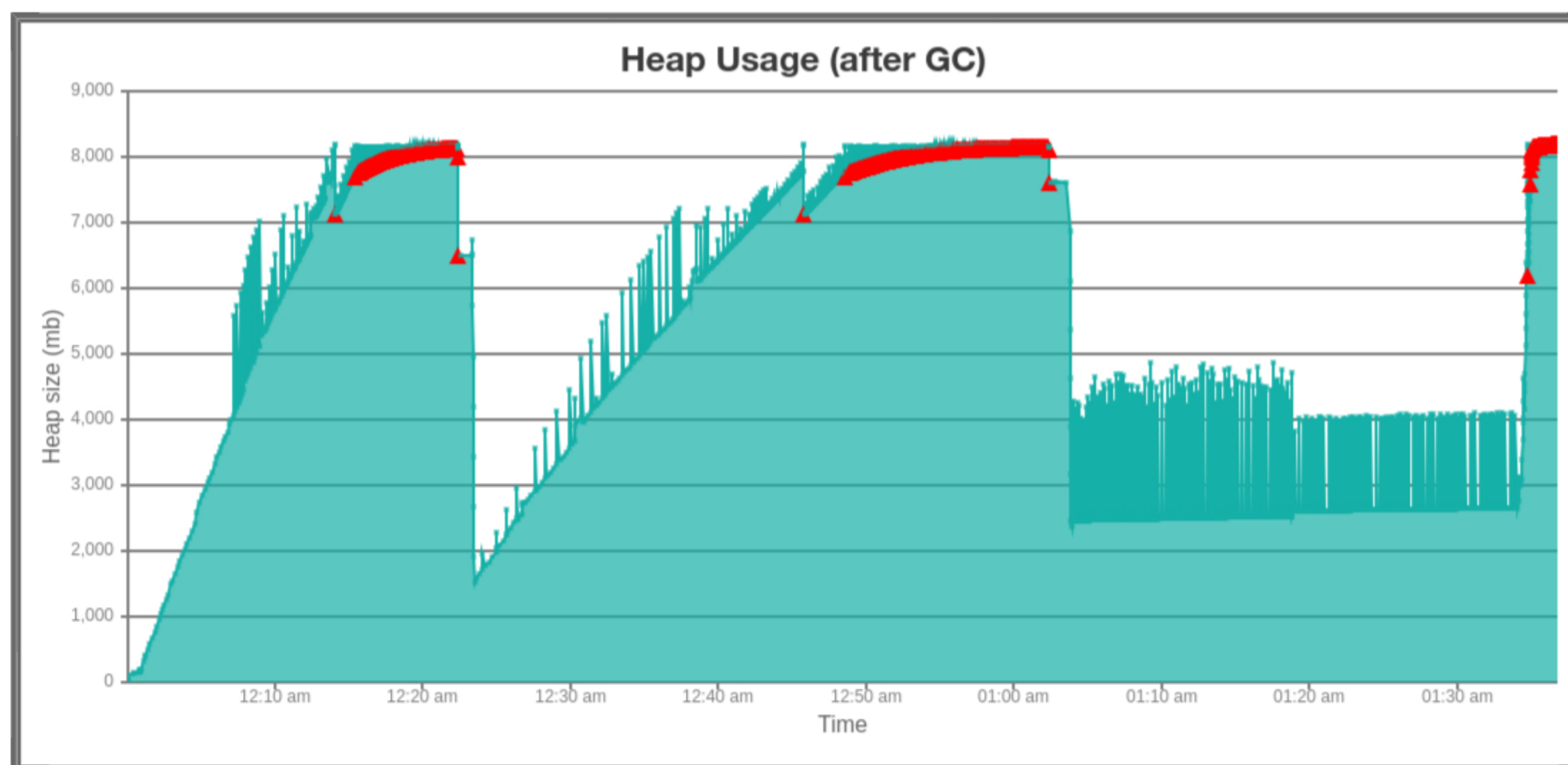
Avg Pause GC Time	466 ms
Max Pause GC Time	2 sec 780 ms

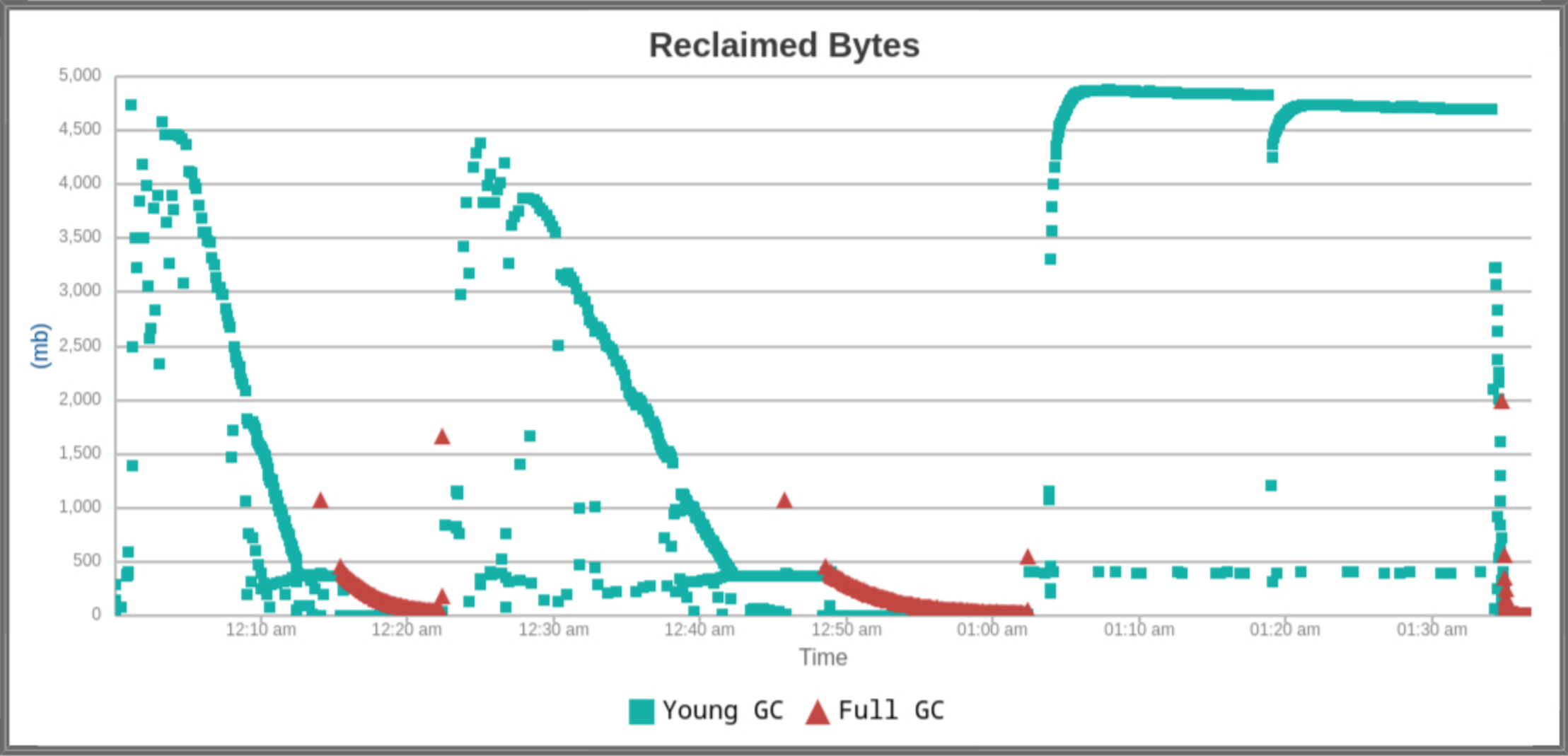
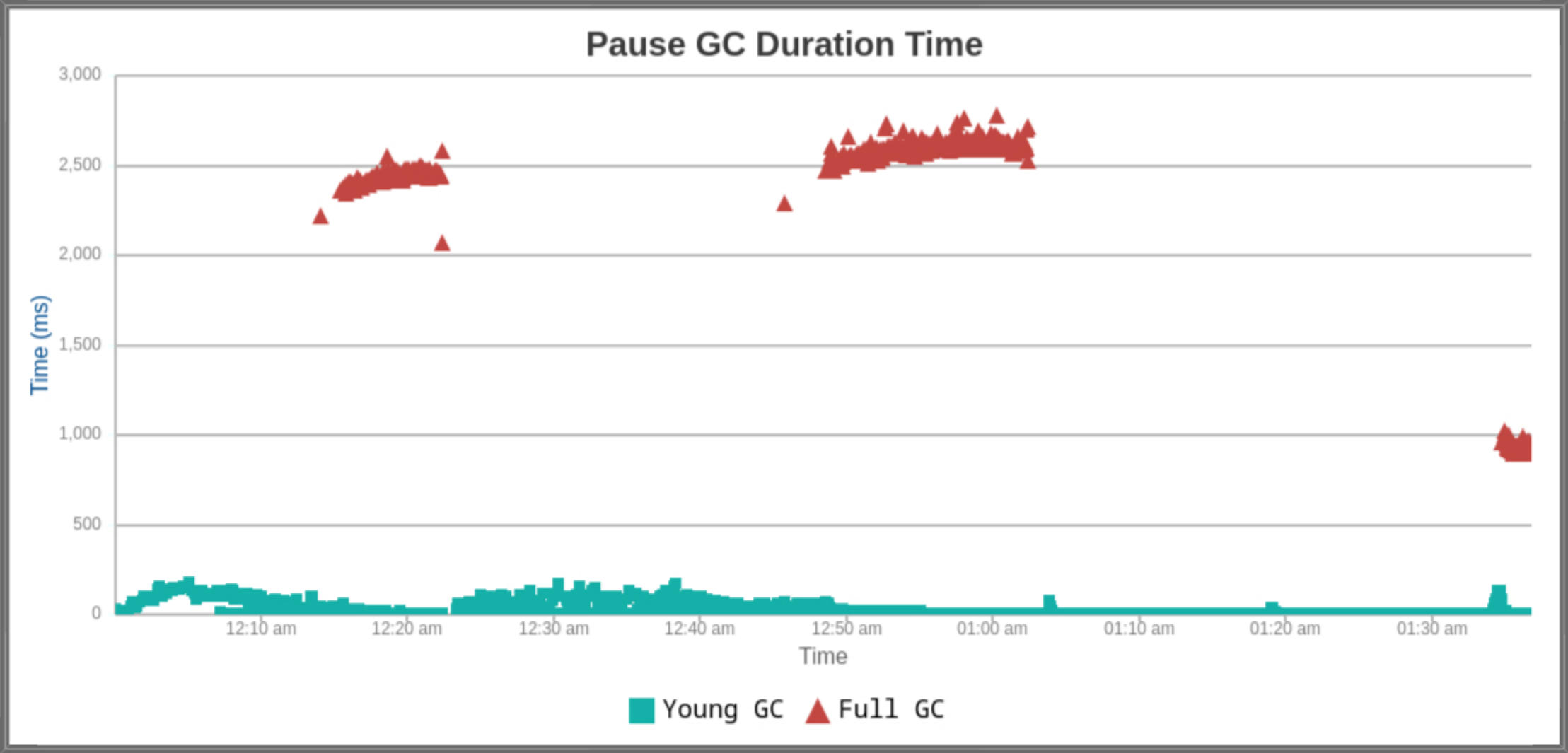
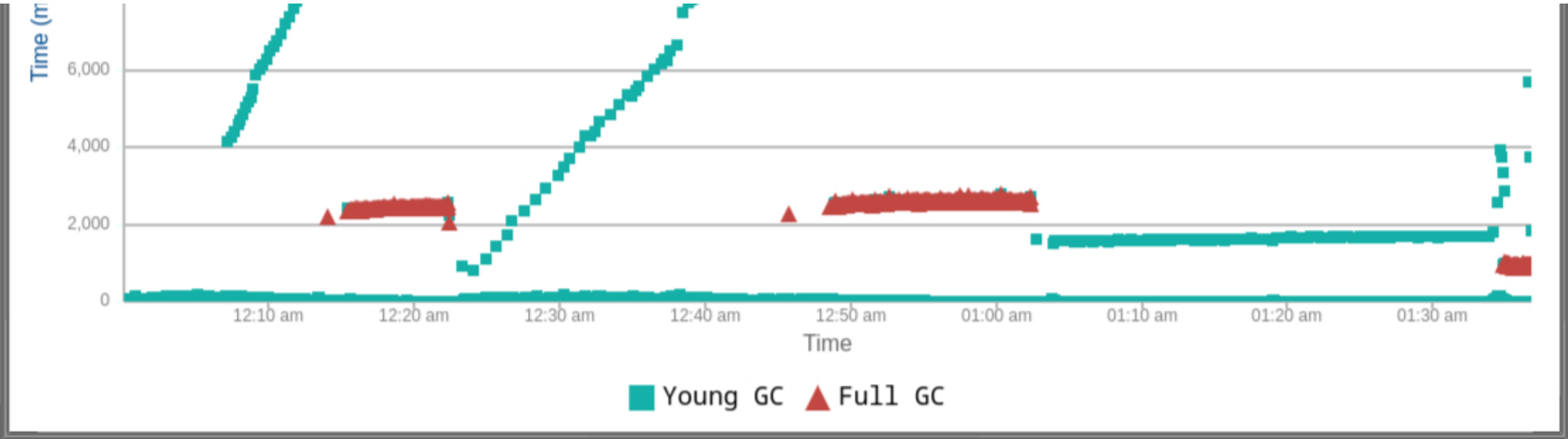
GCPauseDuration Time Range

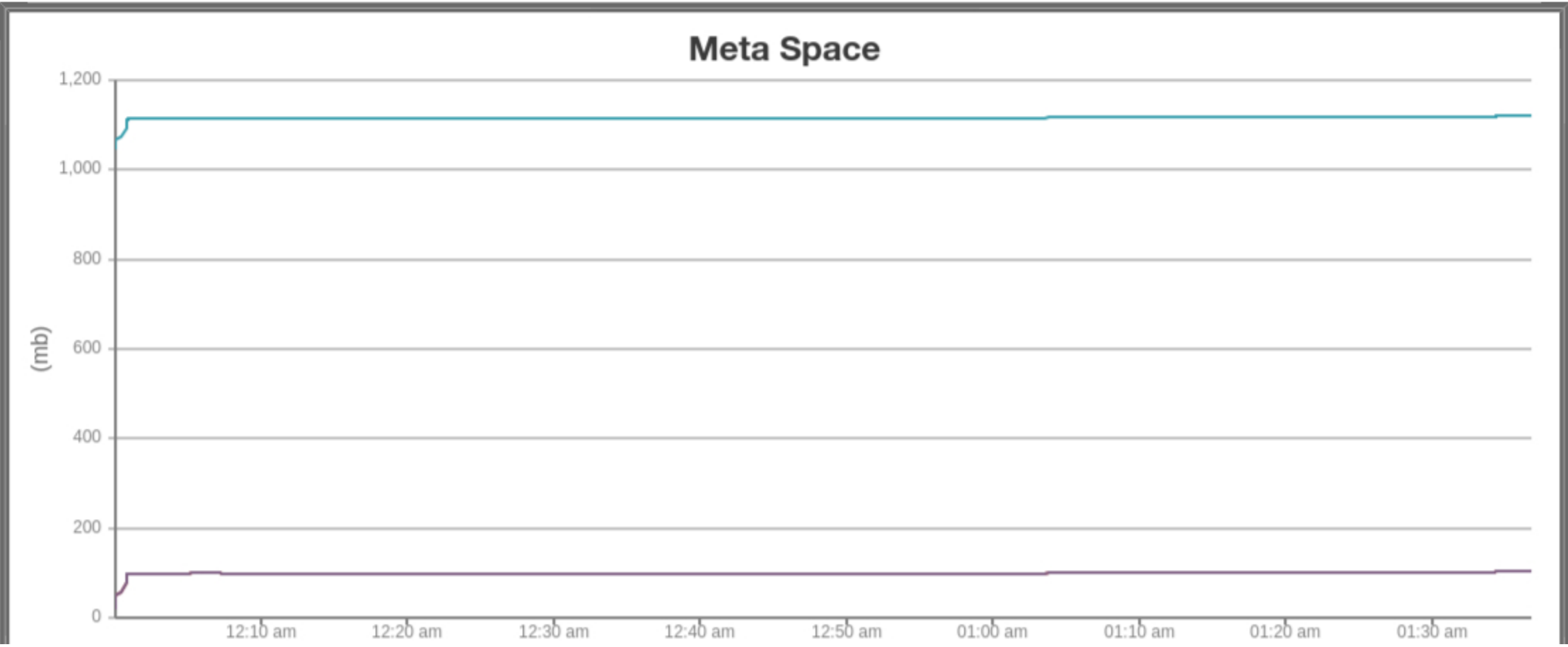
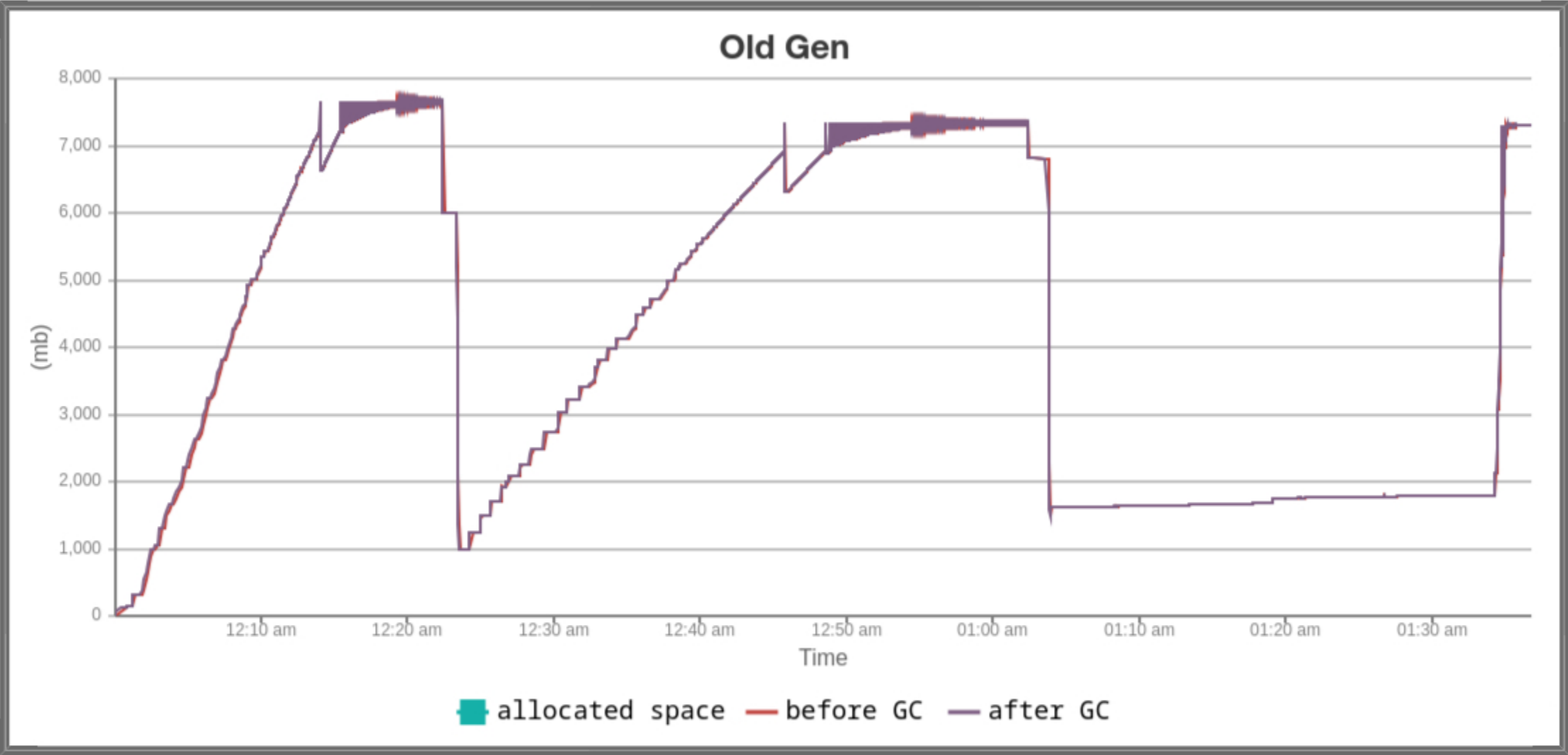
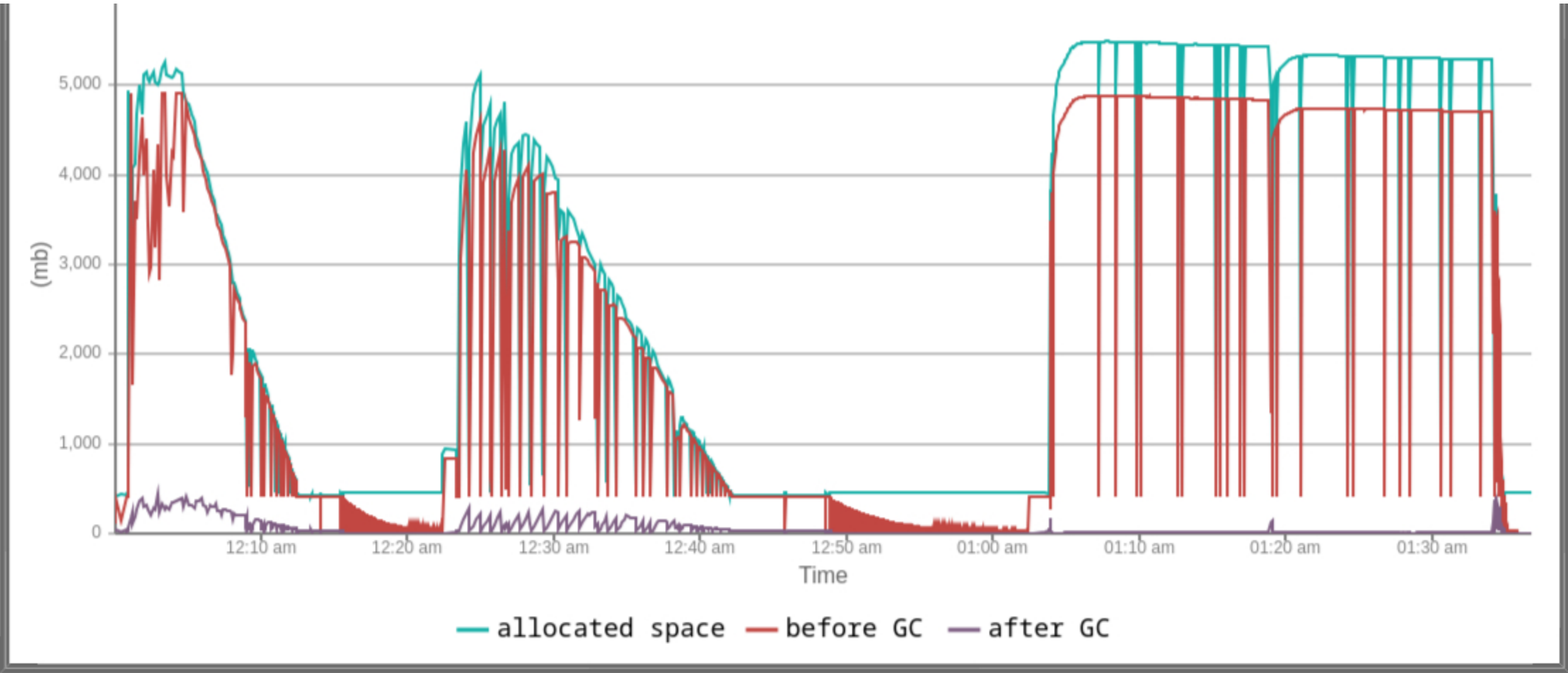
Duration (ms)	No. of GCs	Percentage
1,000 ms Change		
0 - 1,000	2225	83.96%
1,000 - 2,000	1	0.04%
2,000 - 3,000	424	16.0%

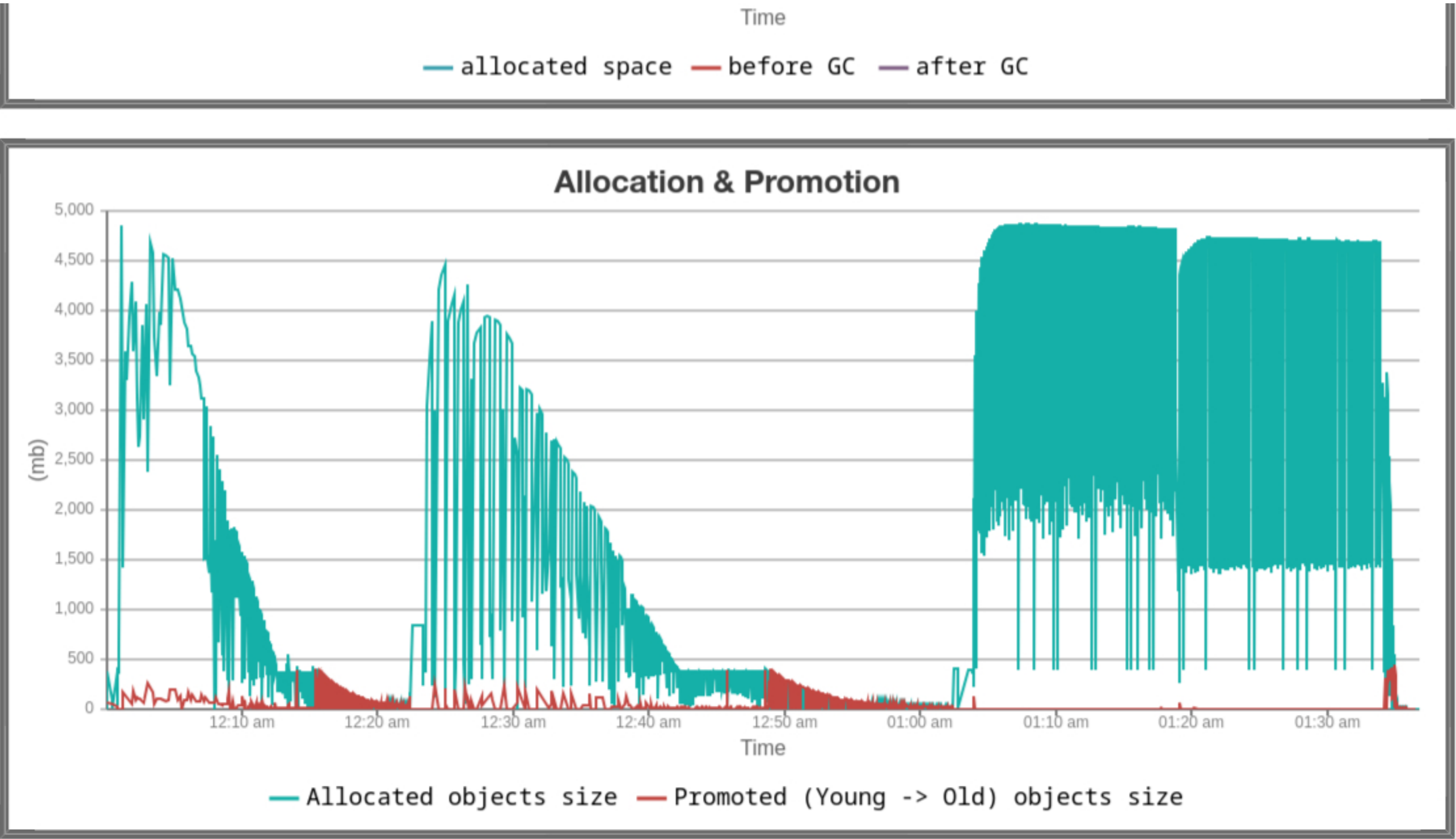


📊 Interactive Graphs [\(How to zoom graphs?\)](#)

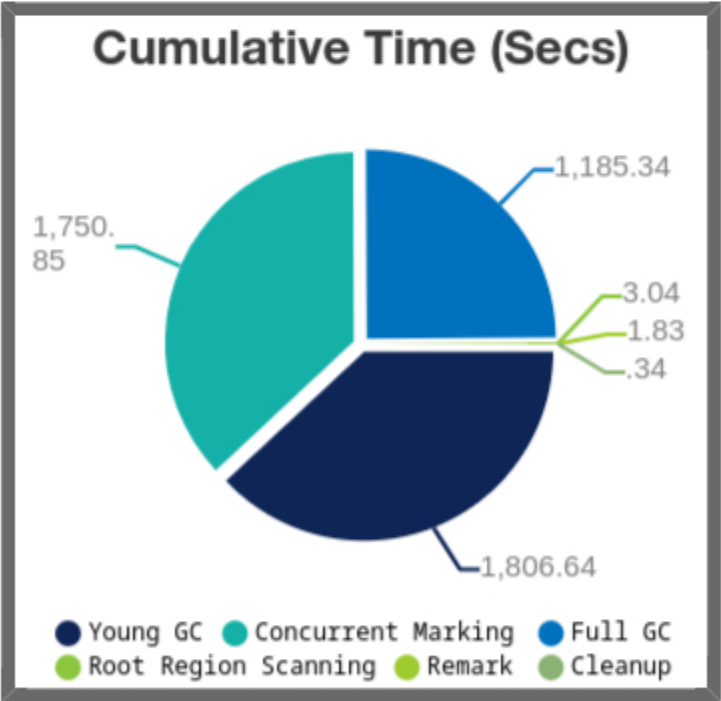
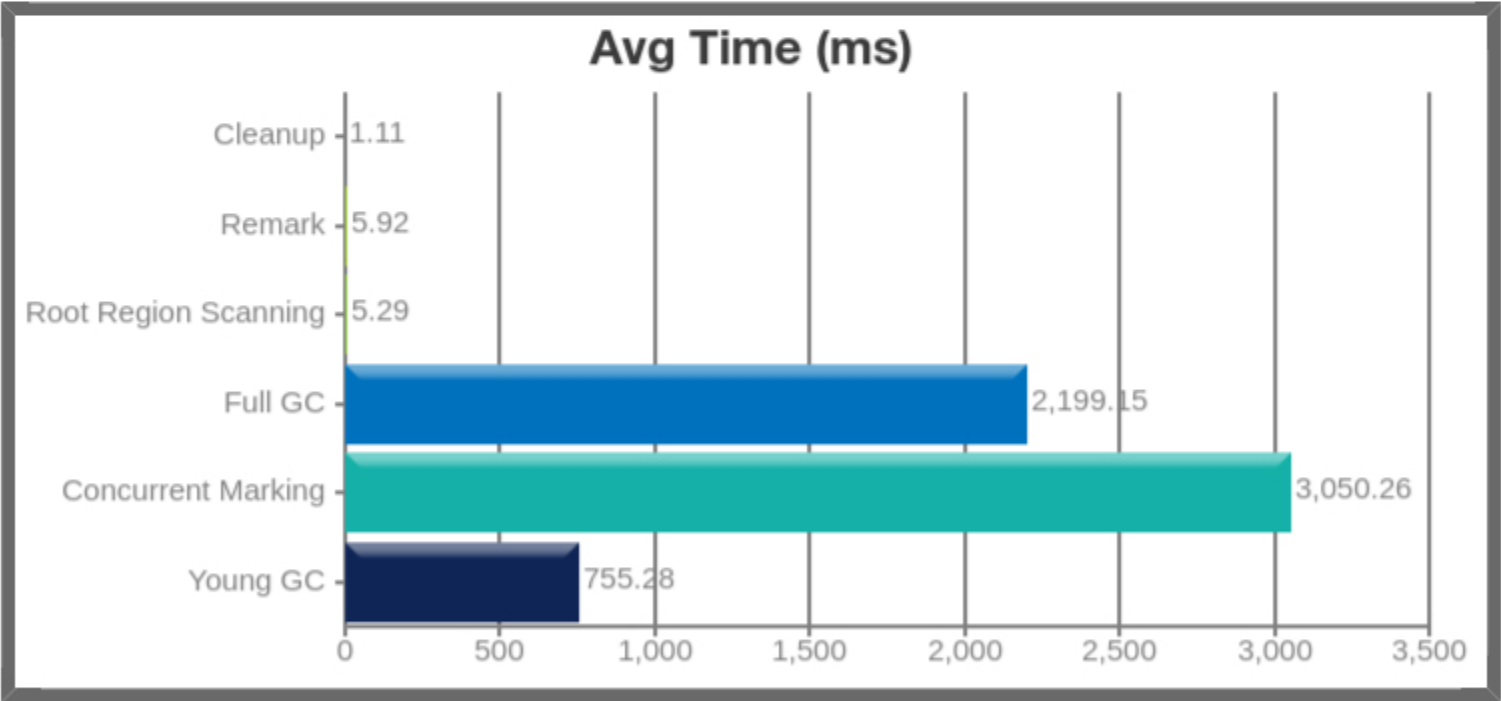






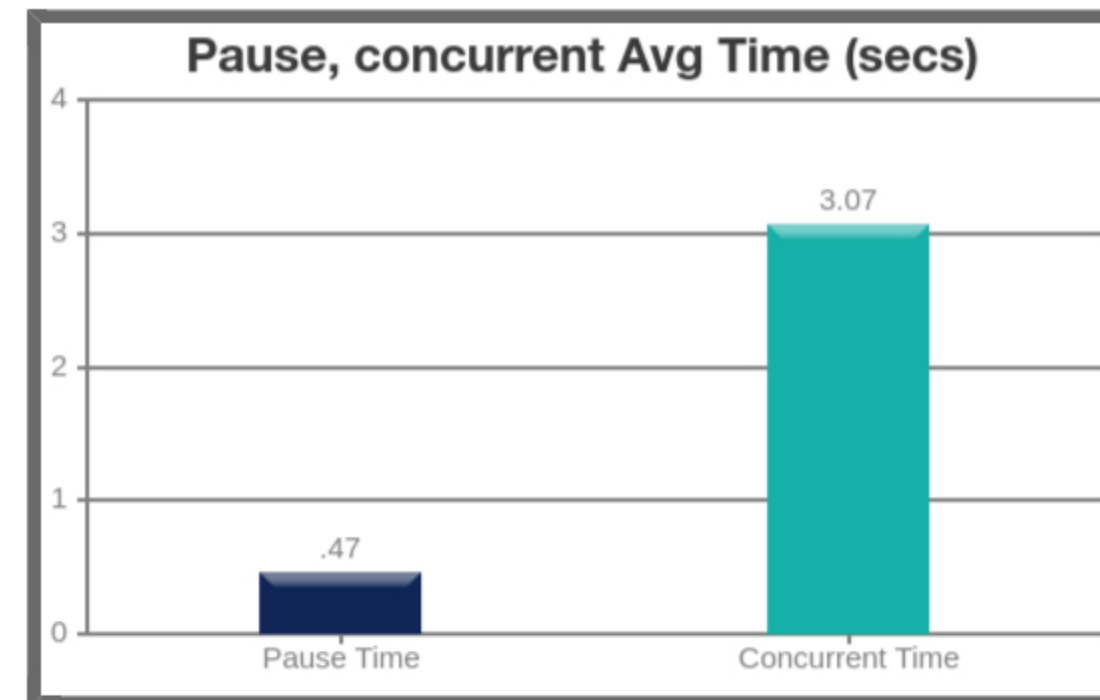
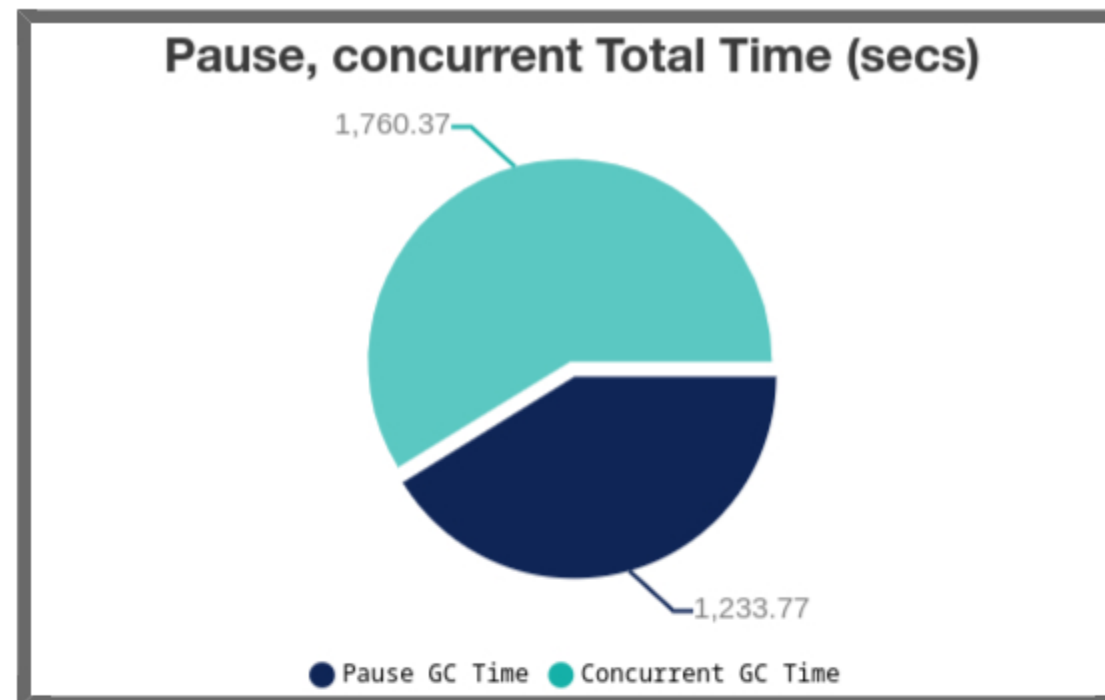


G1 Collection Phases Statistics



	Young GC 	Concurrent Marking	Full GC 	Root Region Scanning	Remark 	Cleanup
Total Time 	30 min 6 sec 641 ms	29 min 10 sec 849 ms	19 min 45 sec 341 ms	3 sec 35 ms	1 sec 831 ms	338 ms
Avg Time 	755 ms	3 sec 50 ms	2 sec 199 ms	5.29 ms	5.92 ms	1.11 ms
Std Dev Time	1 sec 795 ms	2 sec 523 ms	662 ms	11.5 ms	1.26 ms	0.327 ms
Min Time 	0	30.8 ms	900 ms	0.00300 ms	1.08 ms	0.232 ms
Max Time 	12 sec 335 ms	12 sec 323 ms	2 sec 780 ms	98.0 ms	14.3 ms	2.11 ms
Interval Time 	2 sec 426 ms	10 sec 123 ms	9 sec 226 ms	10 sec 123 ms	18 sec 474 ms	18 sec 765 ms
Count 	2392	574	539	574	309	304

🕒 G1 GC Time



Pause Time ?

Total Time	20 min 33 sec 769 ms
Avg Time	466 ms
Std Dev Time	926 ms
Min Time	0.232 ms
Max Time	2 sec 780 ms

Concurrent Time ?

Total Time	29 min 20 sec 370 ms
Avg Time	3 sec 67 ms
Std Dev Time	2 sec 529 ms
Min Time	36.2 ms
Max Time	12 sec 335 ms

⚙️ Object Stats ?

Total created bytes ?	303.91 gb
Total promoted bytes ?	76.17 gb
Avg creation rate ?	53.59 mb/sec
Avg promotion rate ?	13.43 mb/sec

☰ CPU Stats ? (To learn more about CPU stats, [click here](#))

CPU Time: ?	4 hrs 15 min 31 sec
User Time: ?	4 hrs 15 min 29 sec
Sys Time: ?	2 sec 660 ms

⏴ Consecutive Full GC ?

None.

⏴ Long Pause ?

None.

🕒 Safe Point Duration ?

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

Not Reported in the log

Not reported in the log.

Allocation stall metrics

(To learn more about Allocation Stall, [click here](#))




Not Reported in the log.

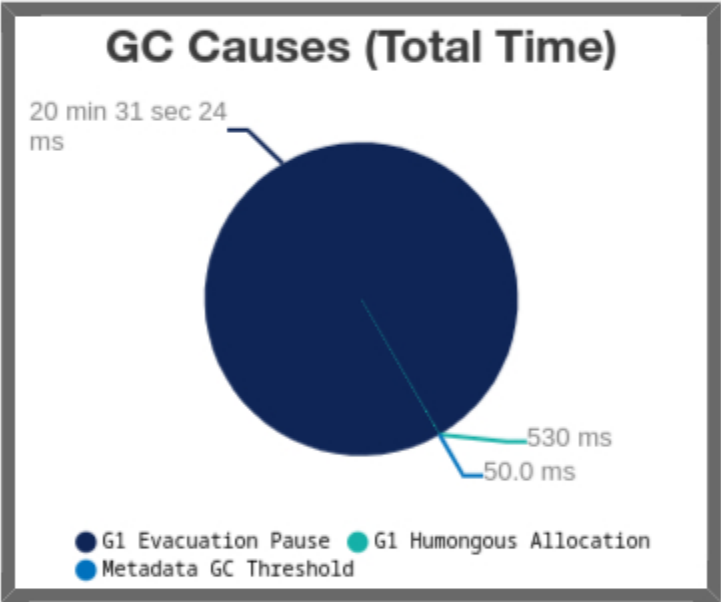
String Deduplication Metrics

Not Reported in the log.

GC Causes

(What events caused GCs & how much time they consumed?)

Cause	Count	Avg Time	Max Time	Total Time
G1 Evacuation Pause 	2346	525 ms	2 sec 780 ms	20 min 31 sec 24 ms
G1 Humongous Allocation 	8	66.3 ms	140 ms	530 ms
Metadata GC Threshold 	3	16.7 ms	20.0 ms	50.0 ms



Tenuring Summary

Not reported in the log.

JVM Arguments

(To learn about JVM Arguments, [click here](#))

Not reported in the log.