



GC Intelligence Report

 perf_seda_141_G1_400mb_gc.zip

 Duration: 13 hrs 26 min 15 sec

 Congratulations! Your application's GC activity is healthy.

Recommendations

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

- ✔ **5 sec 50 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:
 1. 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.
 2. 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.



- ✔ **70.0 ms** of GC pause time is triggered by '**Metadata GC Threshold**' event. This type of GC event is triggered under two circumstances:
 1. Configured metaspace size is too small than the actual requirement
 2. There is a classloader leak (very unlikely, but possible).

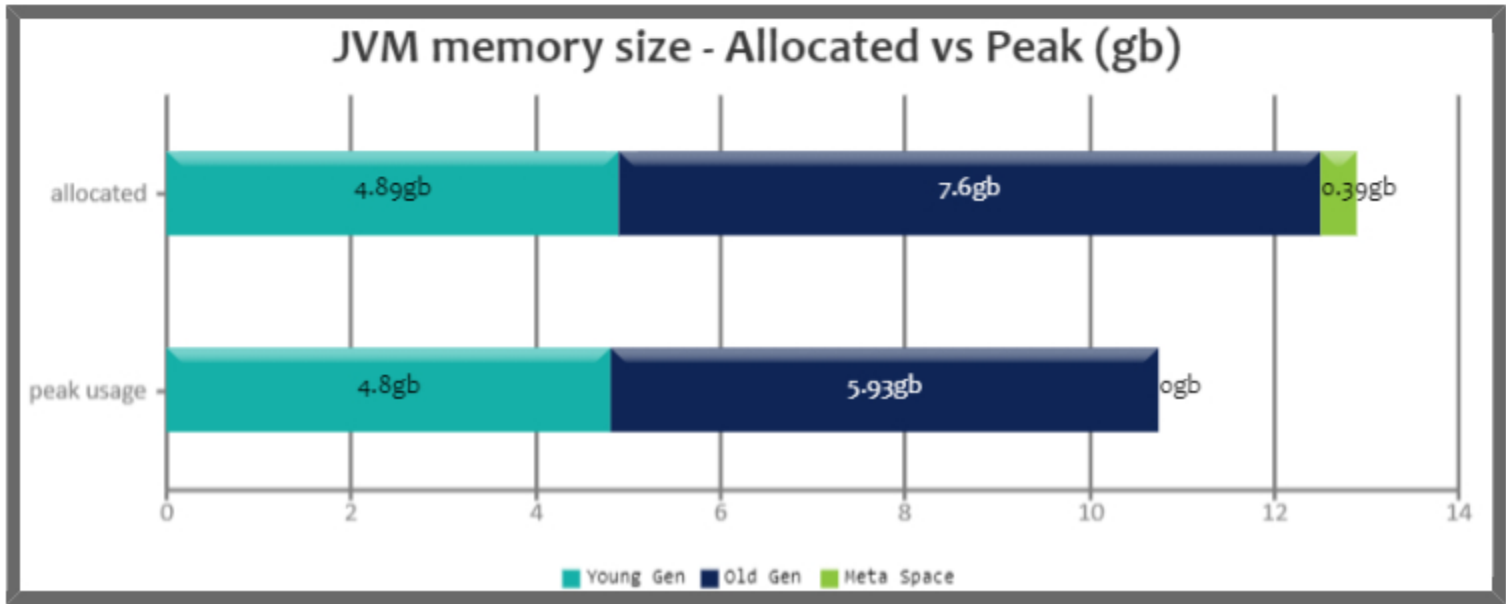
Solution:

You may consider setting '-XX:MetaspaceSize' to a higher value. If this property is not present already please configure it. Setting these this 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 capture heap dump from your application and analyze it. You can learn how to do heap dump analysis 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](#)
- ✔ -XX:+UseCompressedOops is not required to be passed, if you are running in Java SE update 23 and later. Compressed oops is supported and enabled by default in Java SE 6u23 and later versions. For more details, [refer here](#).

JVM memory size

Generation	Allocated 	Peak 
Young Generation	4.89 gb	4.8 gb
Old Generation	7.6 gb	5.93 gb
Meta Space	400 mb	n/a
Young + Old + Meta space	8.39 gb	7.66 gb



🔍 Key Performance Indicators

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

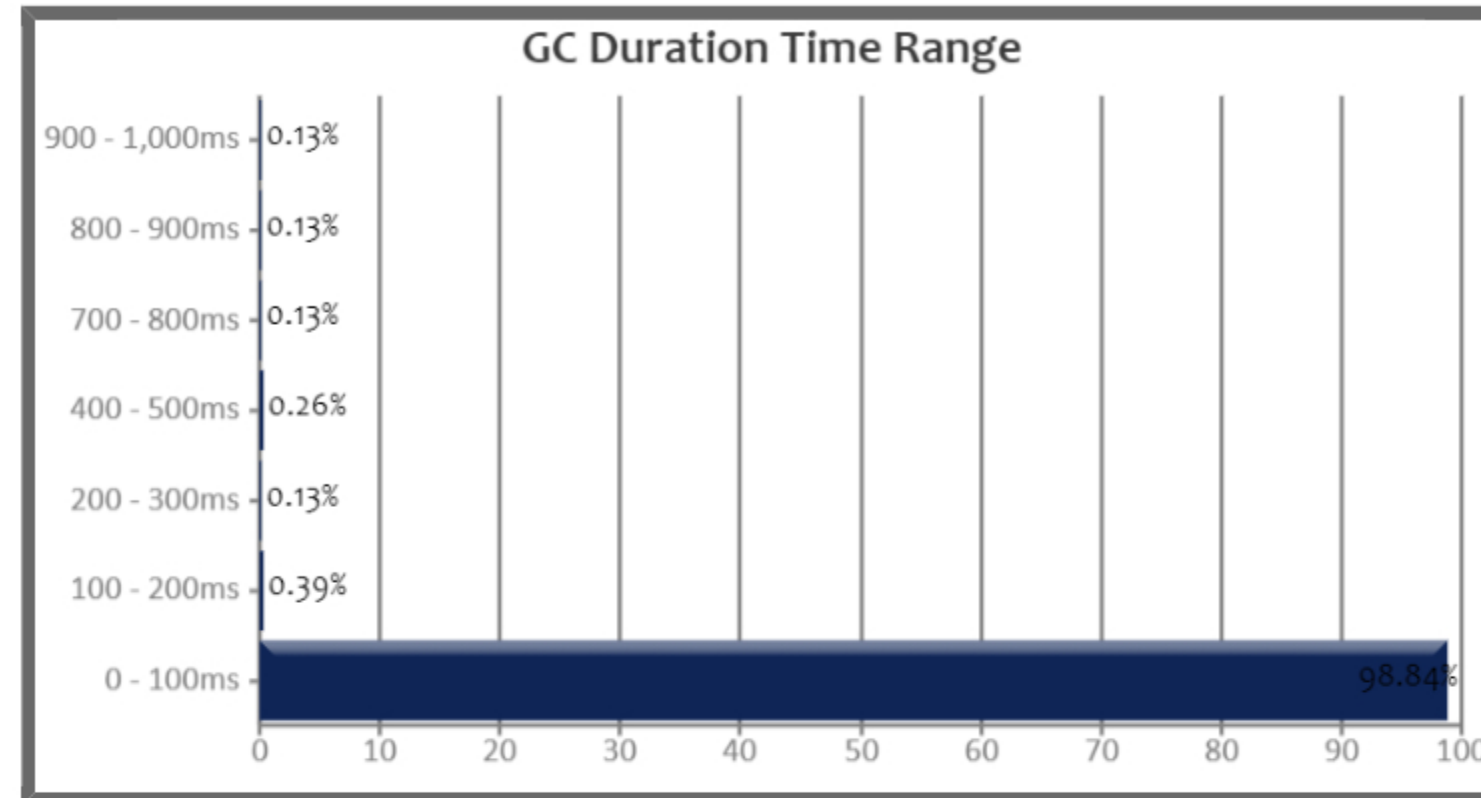
❶ Throughput 📄 : 99.944%

❷ Latency:

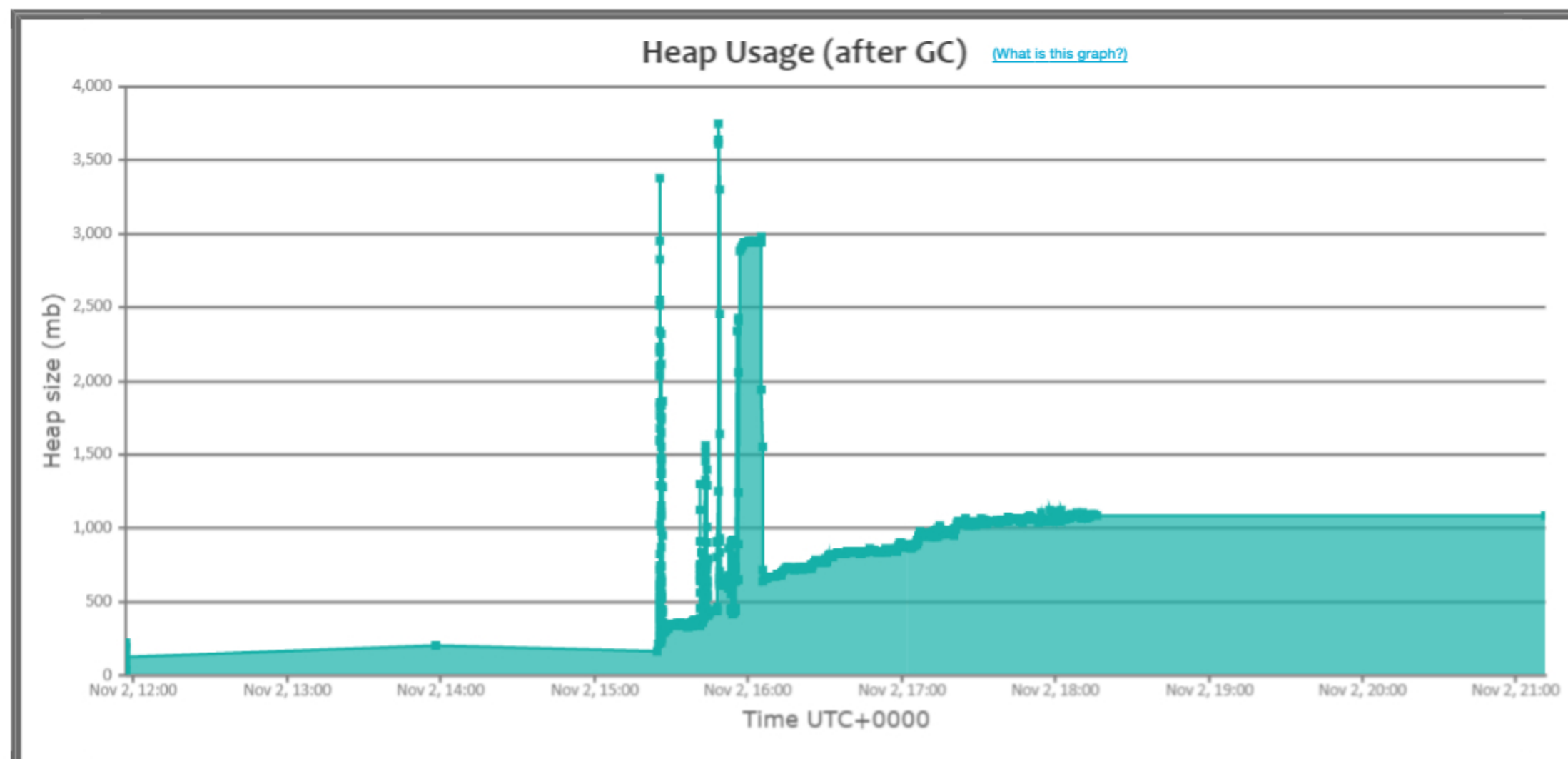
Avg Pause GC Time 📄	34.8 ms
Max Pause GC Time 📄	990 ms

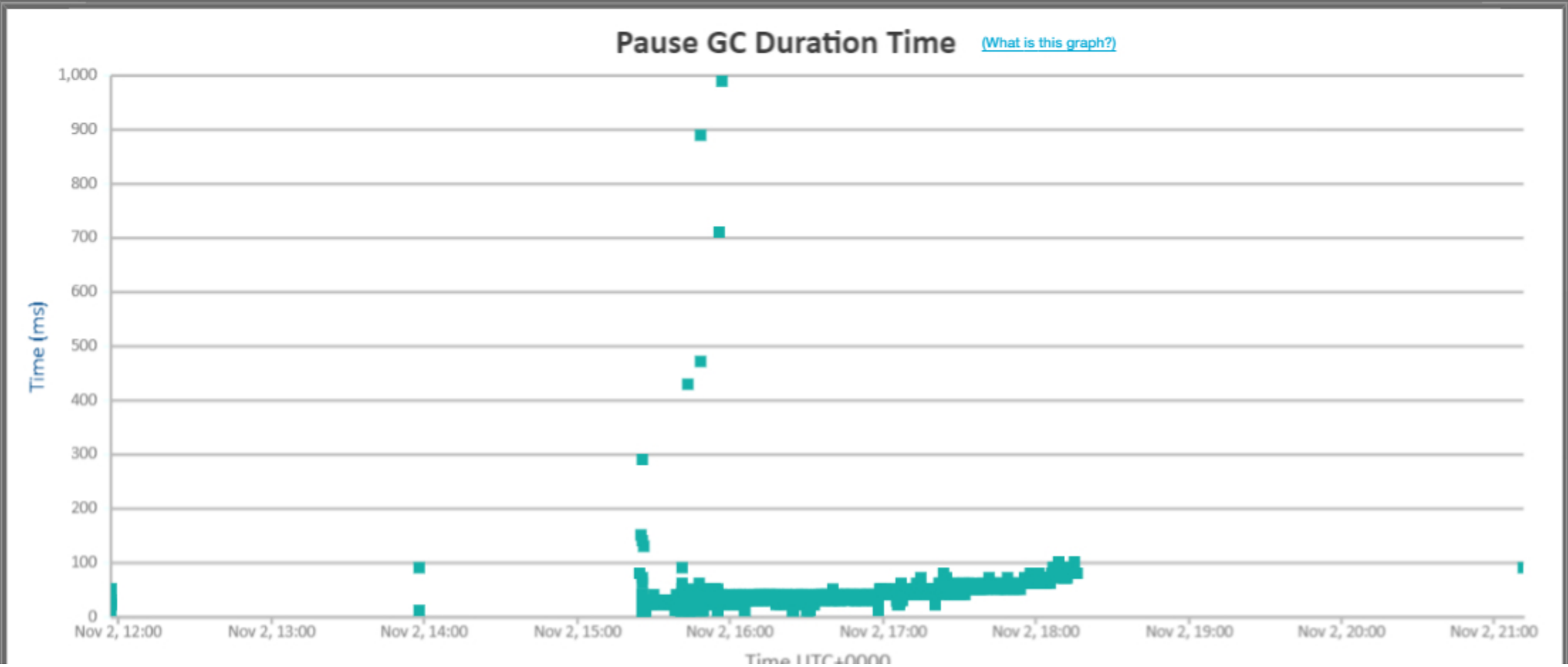
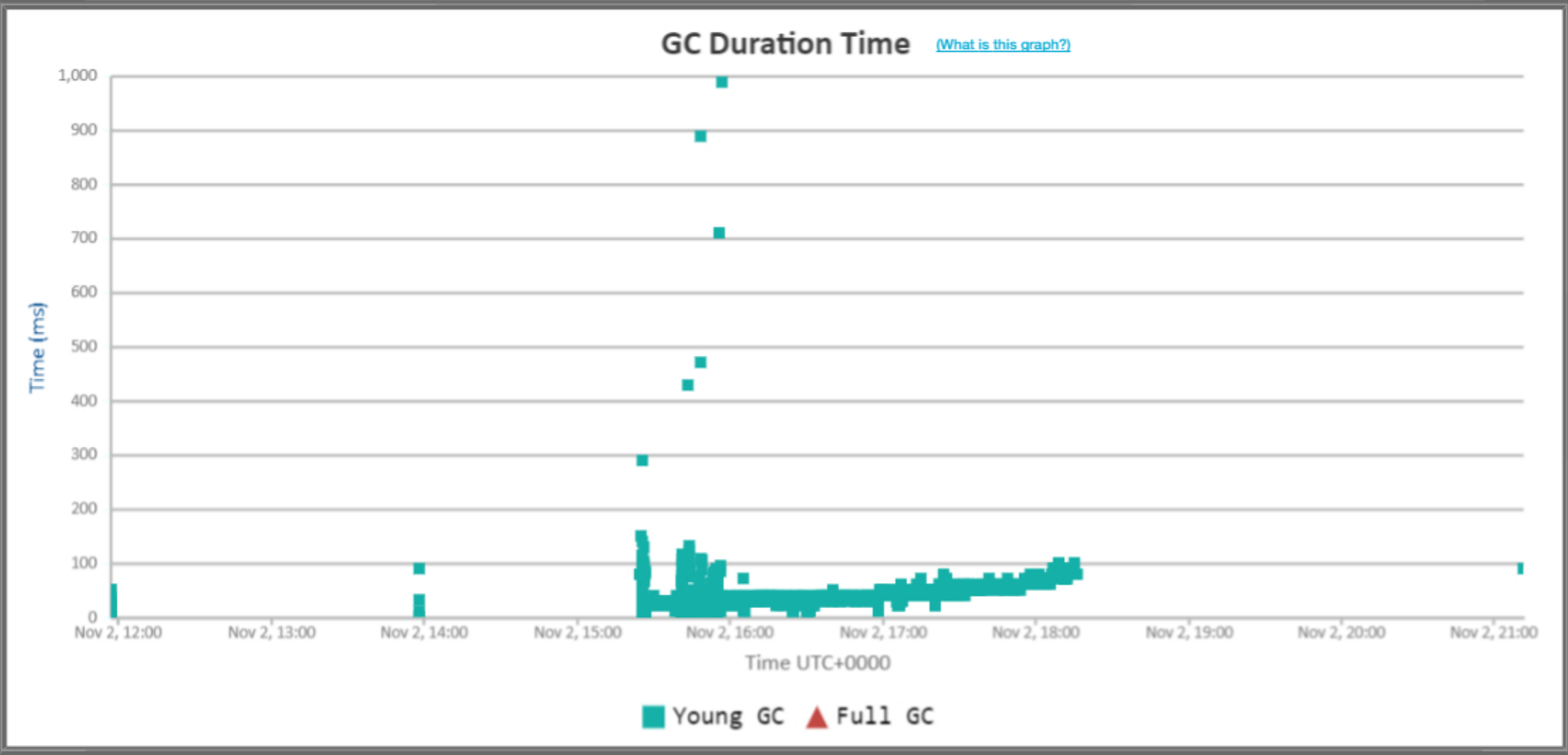
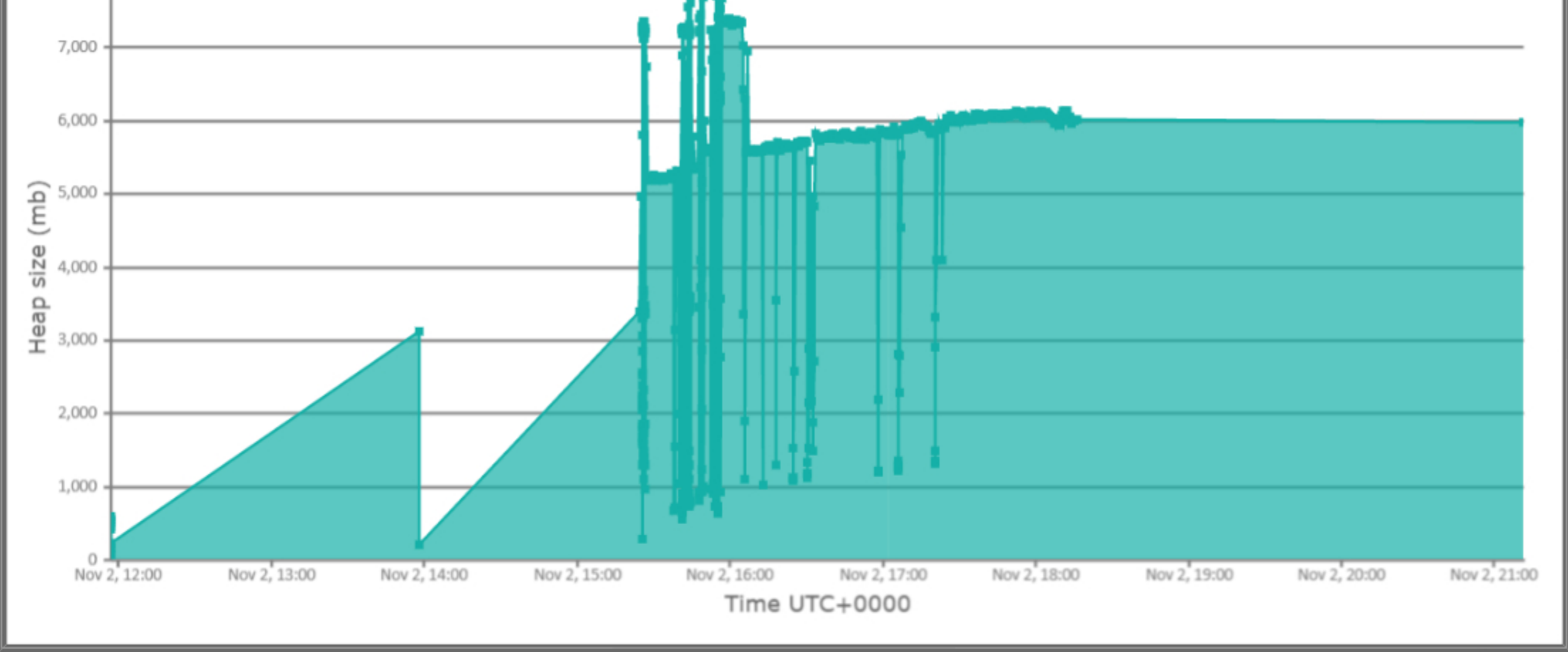
GC Pause Duration Time Range 📄:

Duration (ms)	No. of GCs	Percentage
100 ms Change		
0 - 100	765	98.84%
100 - 200	3	0.39%
200 - 300	1	0.13%
400 - 500	2	0.26%
700 - 800	1	0.13%
800 - 900	1	0.13%
900 - 1,000	1	0.13%

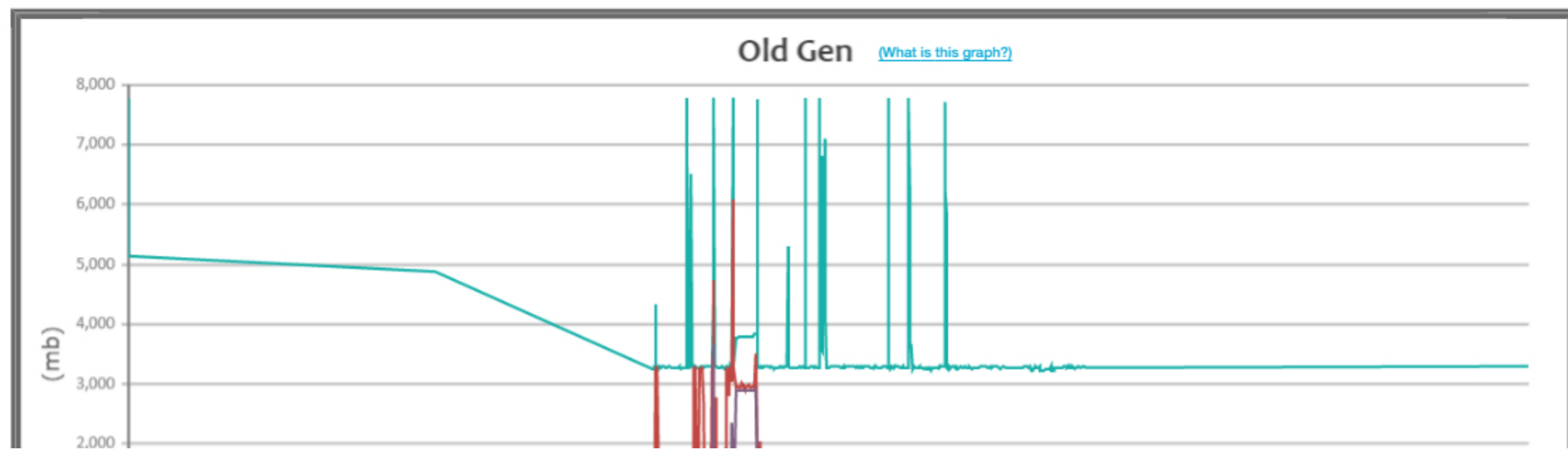
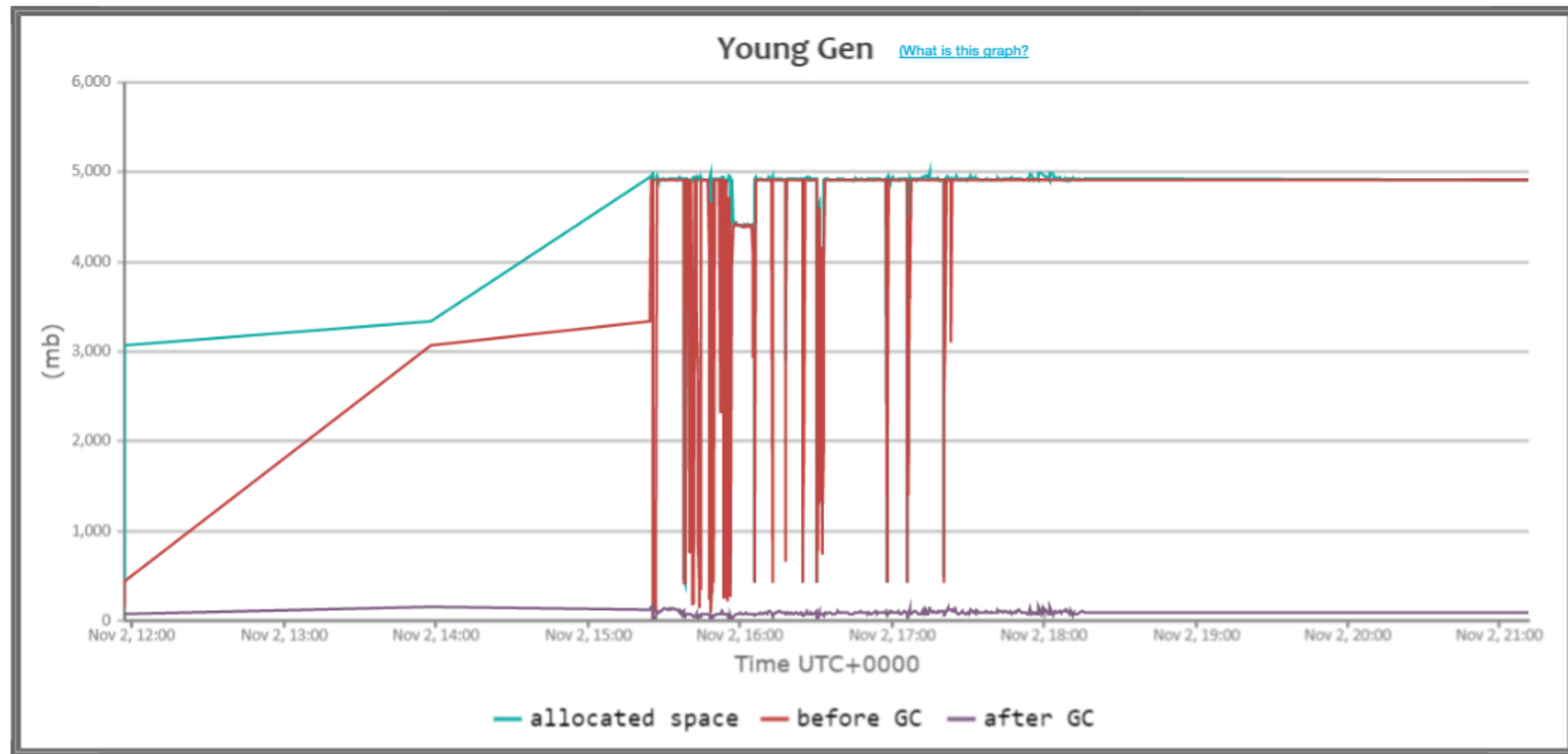
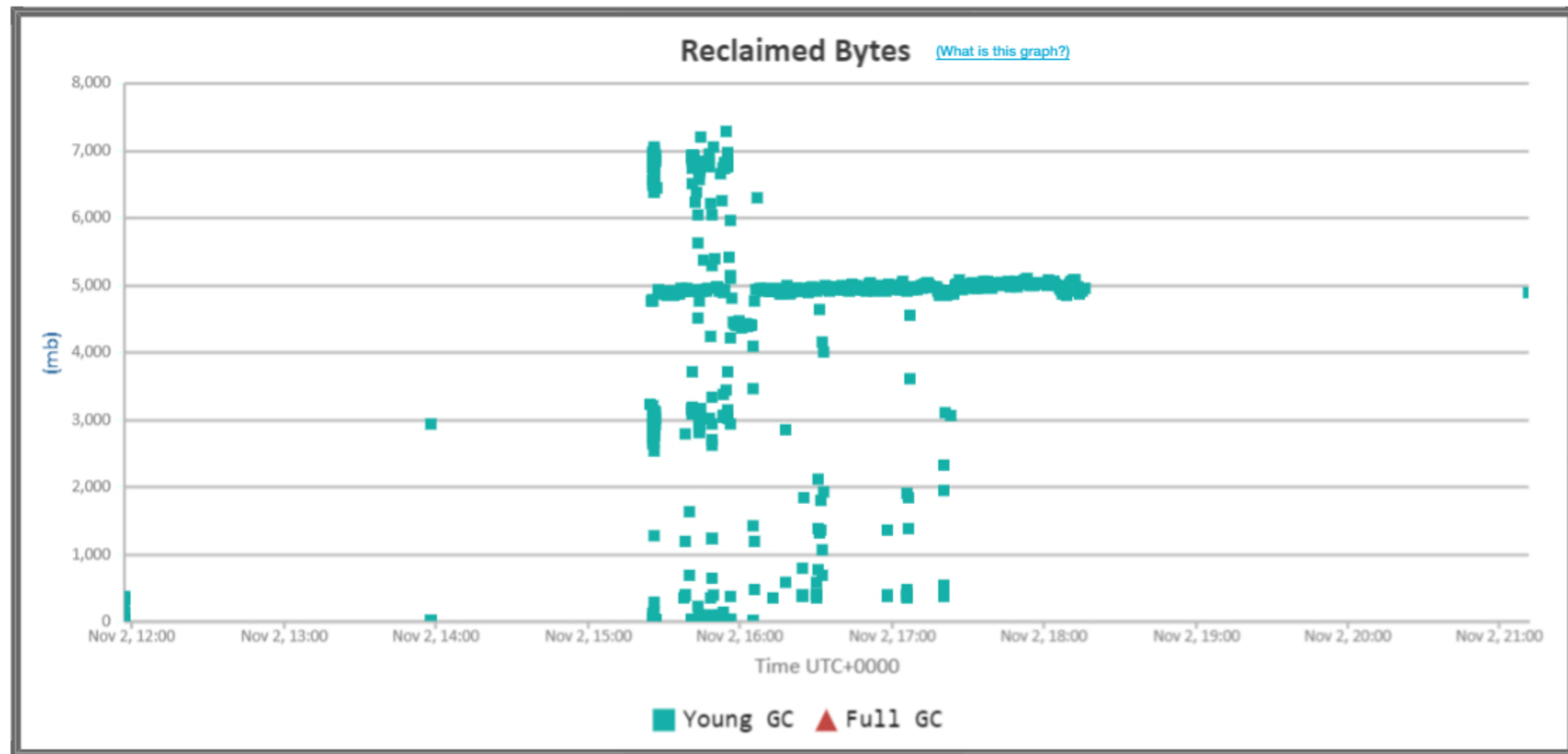


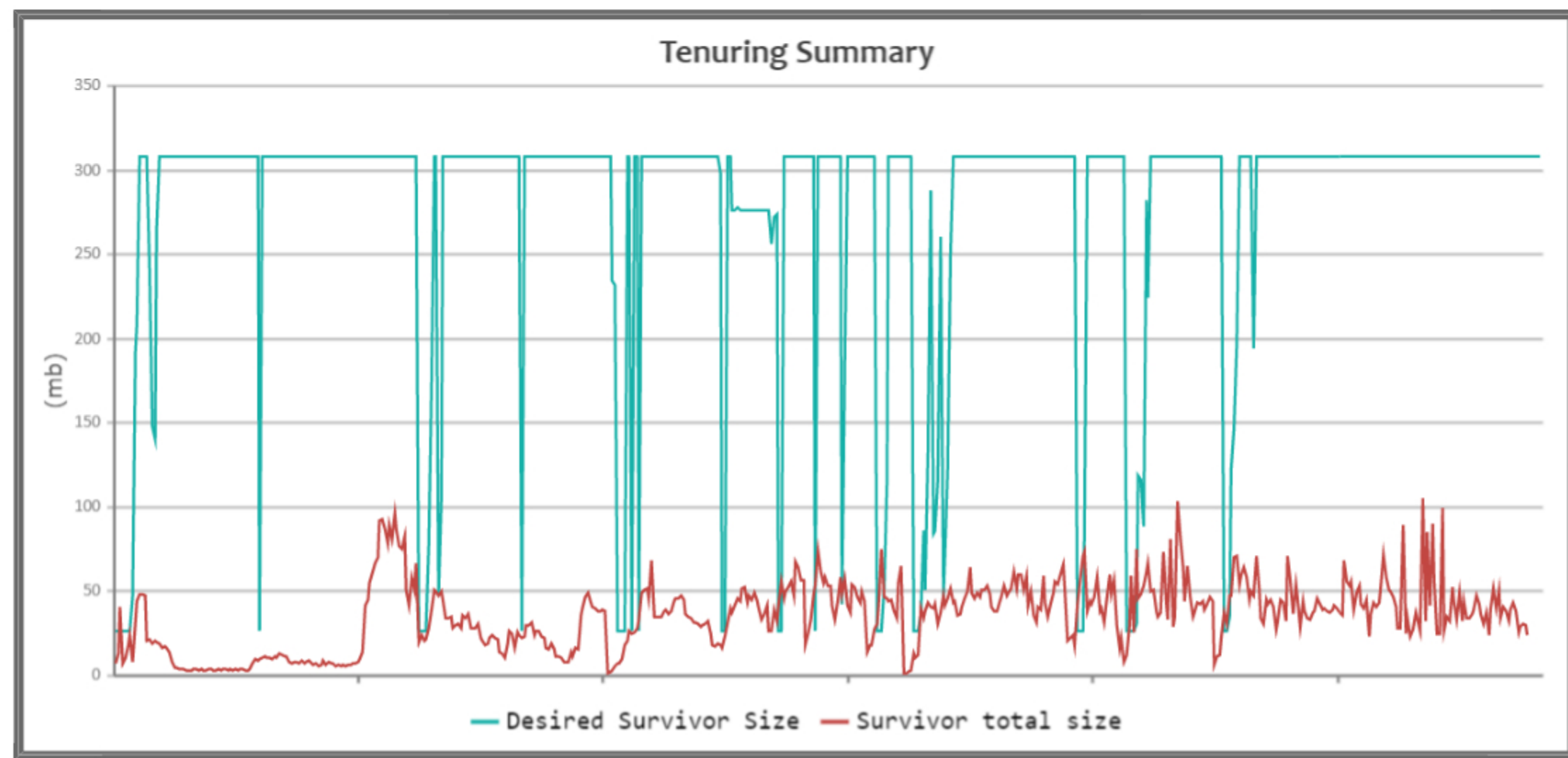
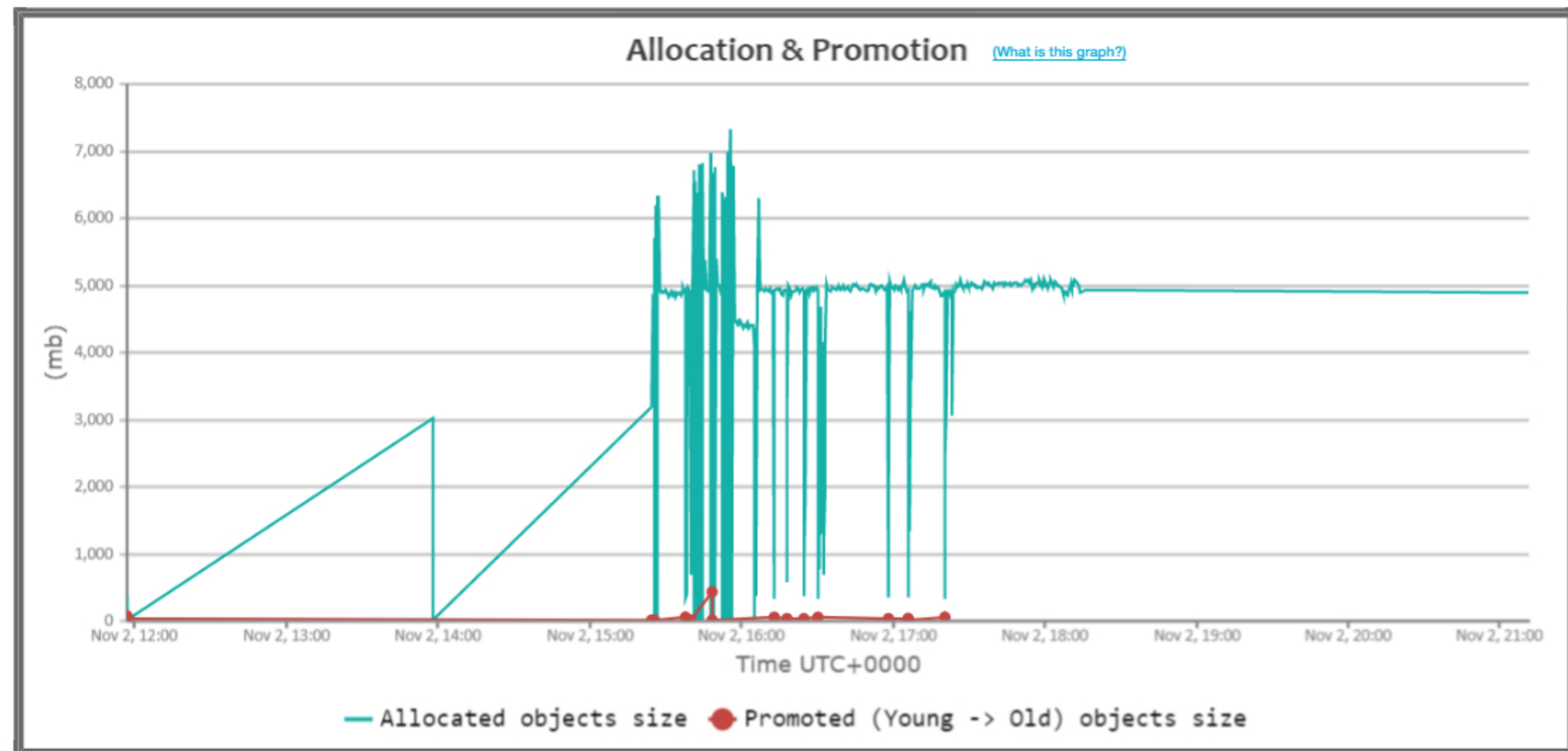
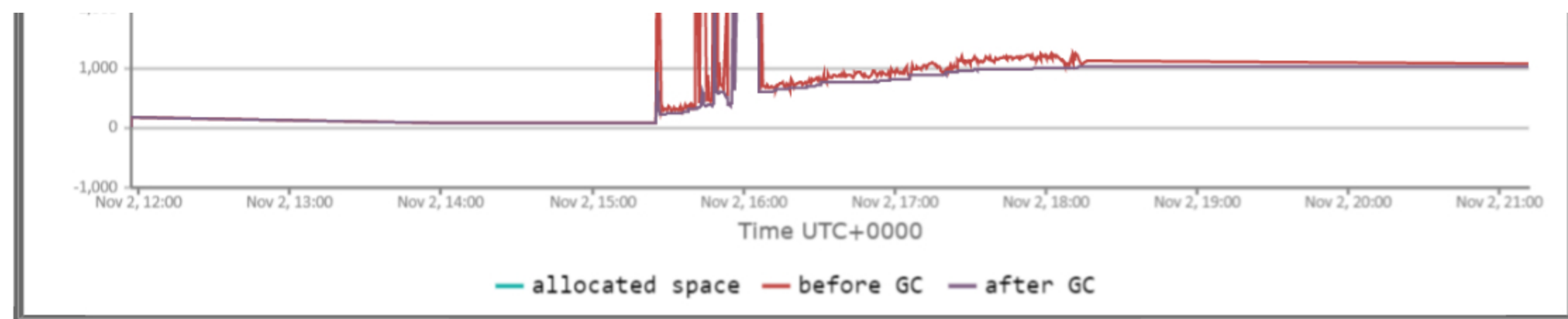
📊 Interactive Graphs [\(How to zoom graphs?\)](#)





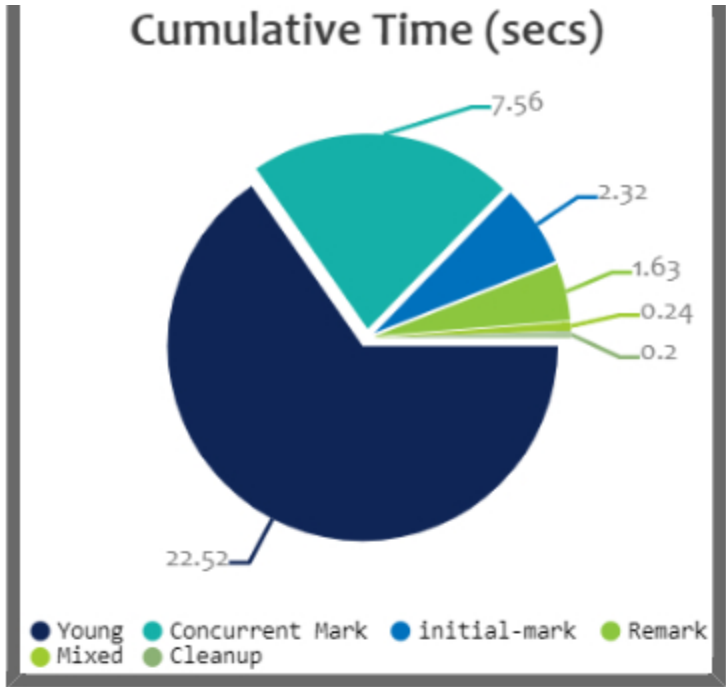
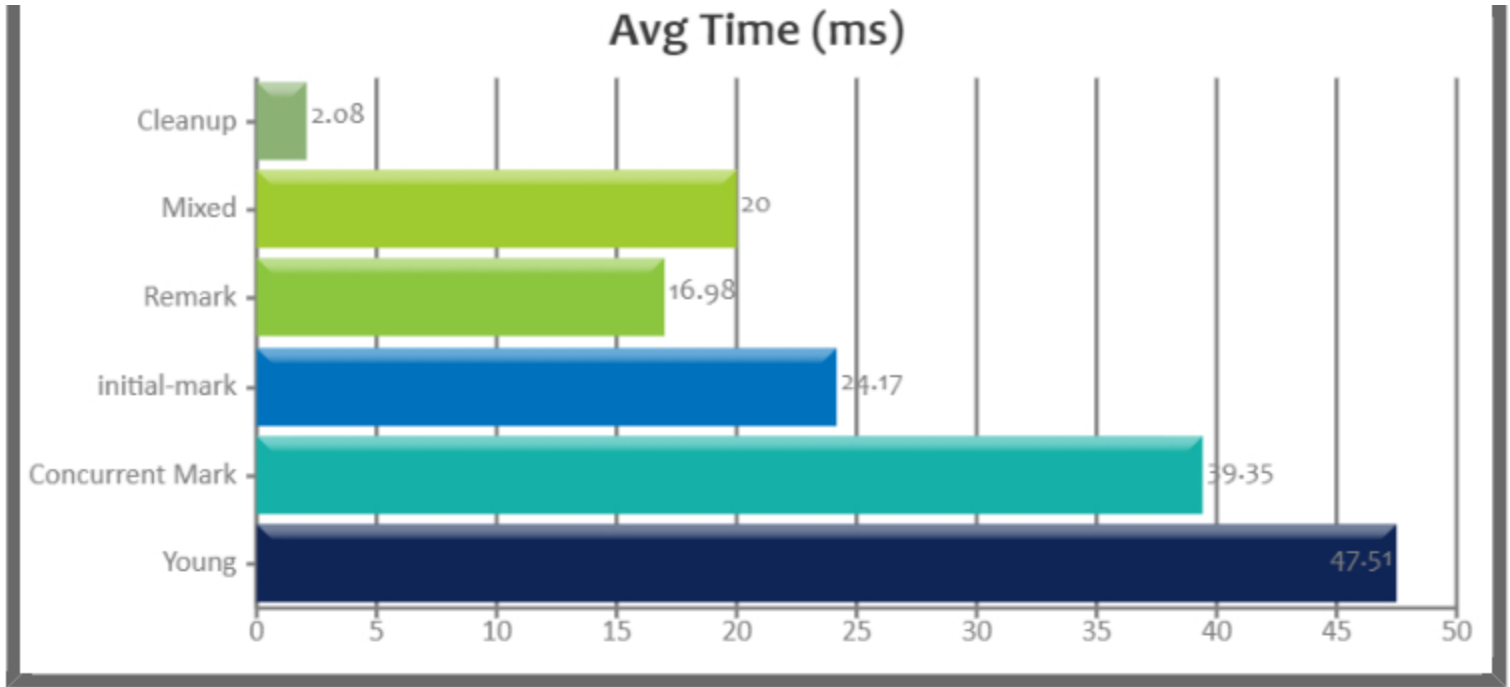
■ Young GC ▲ Full GC





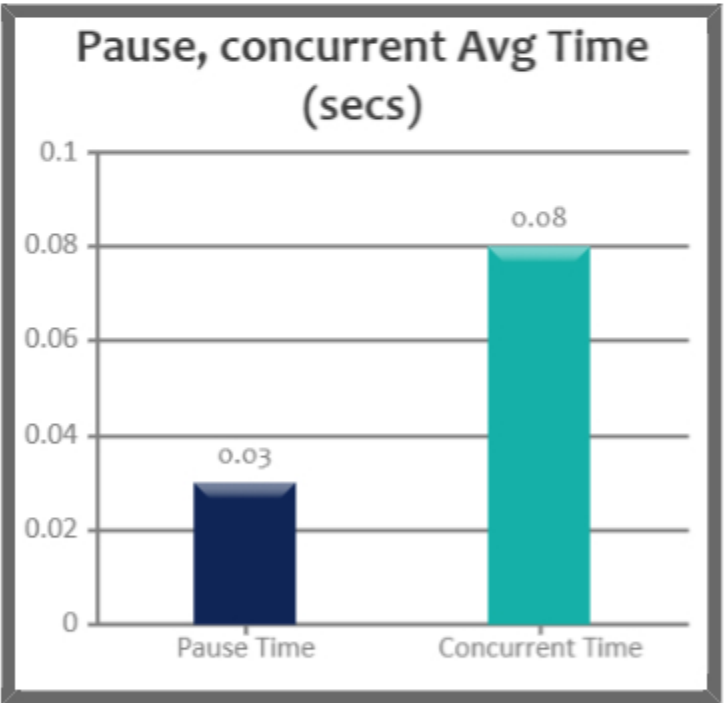
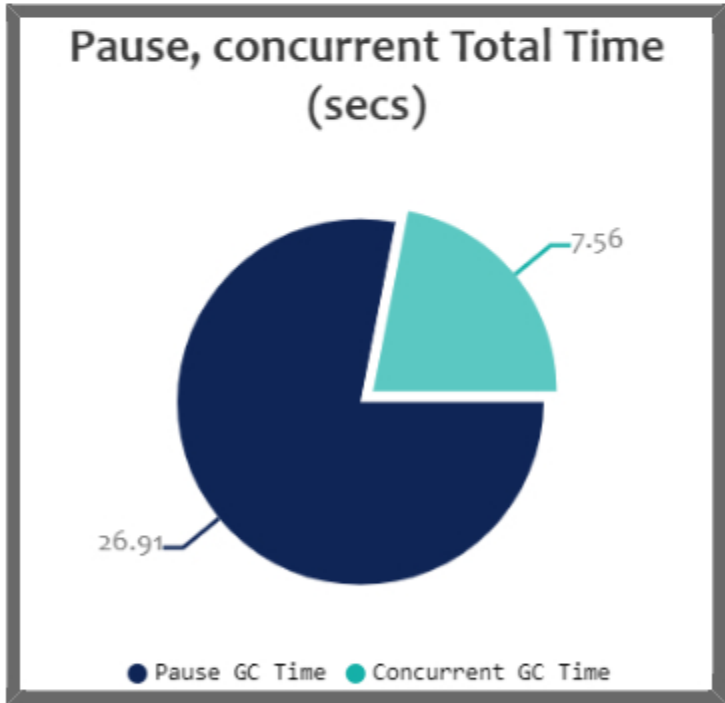
🔒 G1 Collection Phases Statistics

(One G1 GC event has multiple phases. This section provides detailed statistics of each G1 phases.)



	Young ⓘ	Concurrent Mark	initial-mark ⓘ	Remark ⓘ	Mixed ⓘ	Cleanup ⓘ	Total
Count ⓘ	474	192	96	96	12	96	966
Total GC Time ⓘ	22 sec 520 ms	7 sec 555 ms	2 sec 320 ms	1 sec 630 ms	240 ms	200 ms	34 sec 465 ms
Avg GC Time ⓘ	47.5 ms	39.3 ms	24.2 ms	17.0 ms	20.0 ms	2.08 ms	35.7 ms
Avg Time std dev	74.8 ms	41.8 ms	16.1 ms	5.97 ms	12.9 ms	4.06 ms	58.0 ms
Min/Max Time ⓘ	0 / 990 ms	0 / 132 ms	0 / 90.0 ms	0 / 30.0 ms	0 / 40.0 ms	0 / 10.0 ms	0 / 990 ms
Avg Interval Time ⓘ	1 min 10 sec 376 ms	1 min 18 sec 59 ms	2 min 36 sec 939 ms	2 min 36 sec 940 ms	3 min 37 sec 10 ms	2 min 36 sec 940 ms	1 min 39 sec 284 ms

🔍 G1 GC Time



Pause Time ⓘ





Total Time	26 sec 910 ms
Avg Time	34.8 ms
Std Dev Time	61.3 ms
Min Time	0
Max Time	990 ms

Concurrent Time ⓘ

Total Time	7 sec 555 ms
Avg Time	77.9 ms
Std Dev Time	21.2 ms
Min Time	0.531 ms
Max Time	132 ms

Object state

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

Total created bytes 	2.49 tb
Total promoted bytes 	856.6 mb
Avg creation rate 	54.02 mb/sec
Avg promotion rate 	18 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.)

Consecutive Full GC

None.

Long Pause

None.

Safe Point Duration

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

	Total Time	Avg Time	% of total duration
Total time for which app threads were stopped	37.654 secs	0.002 secs	0.078 %
Time taken to stop app threads	2.031 secs	0.0 secs	0.004 %


Allocation stall metrics

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

Not Reported in the log.

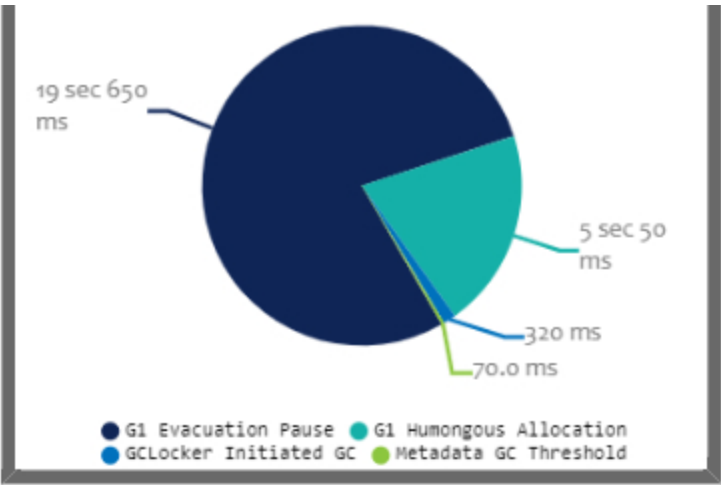
GC Causes

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

Cause	Count	Avg Time	Max Time	Total Time
G1 Evacuation Pause 	395	49.7 ms	990 ms	19 sec 650 ms

GC Causes (Total Time)

G1 Humongous Allocation ?	175	28.9 ms	430 ms	5 sec 50 ms
GCLocker Initiated GC ?	10	32.0 ms	60.0 ms	320 ms
Metadata GC Threshold ?	3	23.3 ms	30.0 ms	70.0 ms



⌘ Tenuring Summary ?

Desired Survivor Size: 308.0 mb,

Max Threshold: 15

Age	Survival Count	Average size (kb)	Average Total 'To' size (kb)
age 1	578	17052.33	17052.33
age 2	569	5092.62	22285.73
age 3	556	3426.35	25829.68
age 4	543	2499.46	28436.73
age 5	531	1702.12	30337.92
age 6	519	1126.33	31586.05
age 7	507	992.68	32587.71
age 8	495	883.02	33197.06
age 9	482	785.28	33526.12
age 10	470	746.64	33830.49
age 11	459	724.48	34245.13
age 12	447	709.54	34659.19
age 13	435	635.21	35097.85
age 14	424	629.02	35571.29
age 15	414	610.39	35927.25

📄 Command Line Flags ?

```
XX:+PrintGCTimeStamps -XX:+PrintTenuringDistribution -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseG1GC
```