Analysis Report

Tips to reduce GC Time

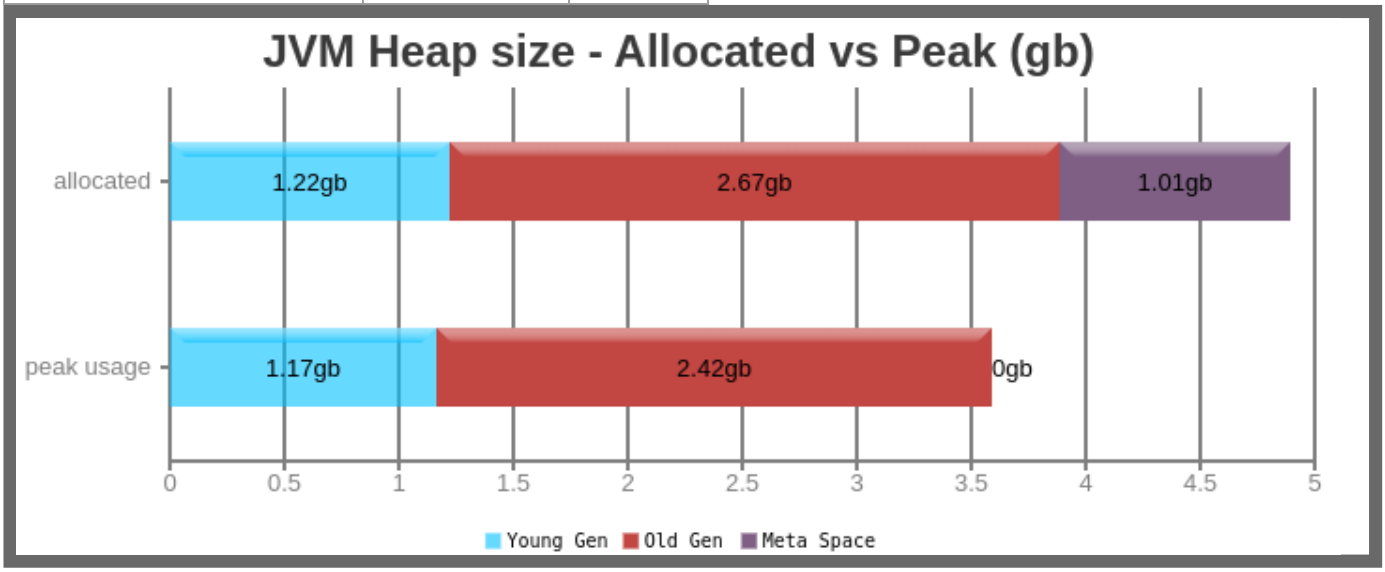
(**CAUTION:** Please do thorough testing before implementing out the recommendations. These are generic recommendations & may not be applicable for your application.)

- **30.67%** of GC time (i.e 1 sec 362 ms) is caused by **'Evacuation Failure'**. When there are no more free regions to promote to the old generation or to copy to the survivor space, and the heap cannot expand since it is already at its maximum, an evacuation failure occurs. For G1 GC, an evacuation failure is very expensive - a. For successfully copied objects, G1 needs to update the references and the regions have to be tenured. b. For unsuccessfully copied objects, G1 will self-forward them and tenure the regions in place.
  Solution:
  1. 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.
  2. If the problem still persists then increase JVM heap size (i.e. -Xmx)
  3. 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.
  4. If concurrent marking cycles are starting on time, but takes long time to finish then increase the number of concurrent marking thread count using the property: '-XX:ConcGCThreads'.
  5. If there are lot of 'to-space exhausted' or 'to-space overflow' GC events, then increase the -XX:G1ReservePercent. The default is 10% of the Java heap. Note: G1 GC caps this value at 50%.

- **22.77%** of GC time (i.e 1 sec 11 ms) is caused by **'G1 Humongous Allocation'**. 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 overriden 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.

JVM Heap Size

Generation	Allocated ❓	Peak ❓
Young Generation	1.22 gb	1.17 gb
Old Generation	2.67 gb	2.42 gb
Meta Space	1.01 gb	3.79 mb
Young + Old + Meta space	4.9 gb	3.05 gb



🔑 Key Performance Indicators

(Important section of the report. To learn more about KPIs, click here (https://blog.gceasy.io/2016/10/01/garbage-collection-kpi/))

1. **Throughput ❓ : 74.193%**

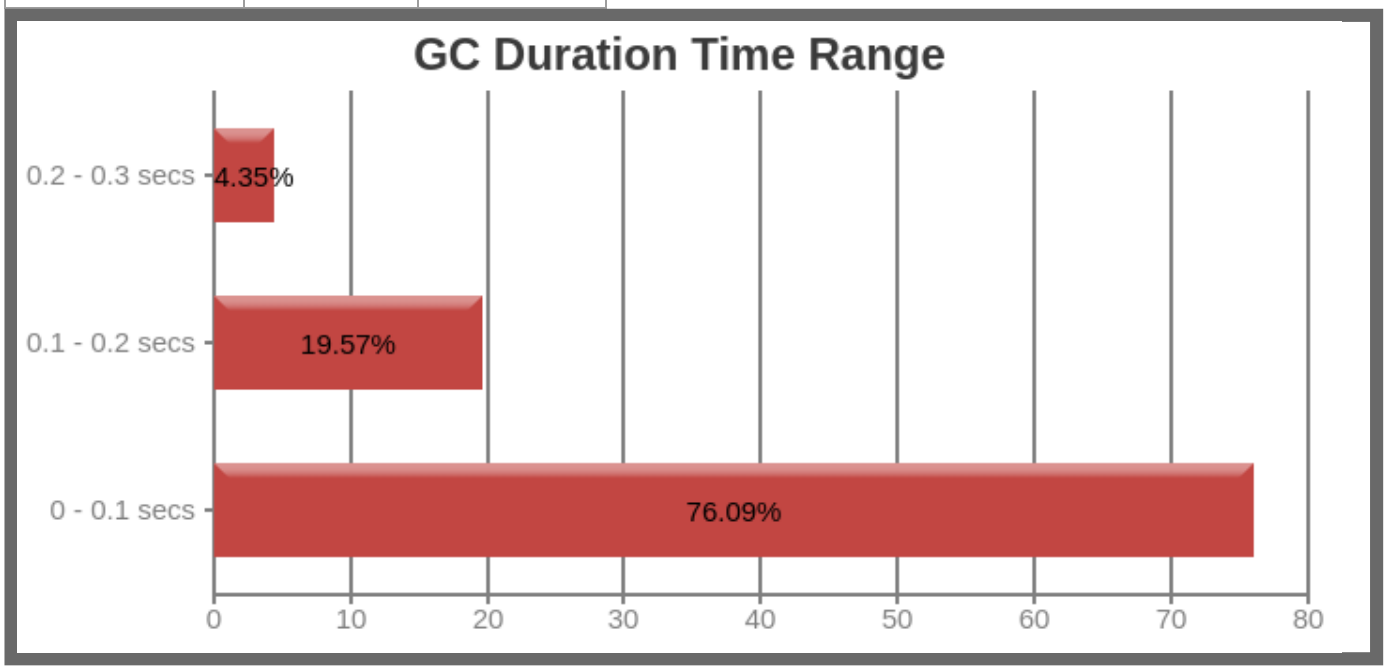
2. **Latency:**

Avg Pause GC Time ❓	**46 ms**
Max Pause GC Time ❓	**240 ms**

GC **Pause** Duration Time Range ❓:

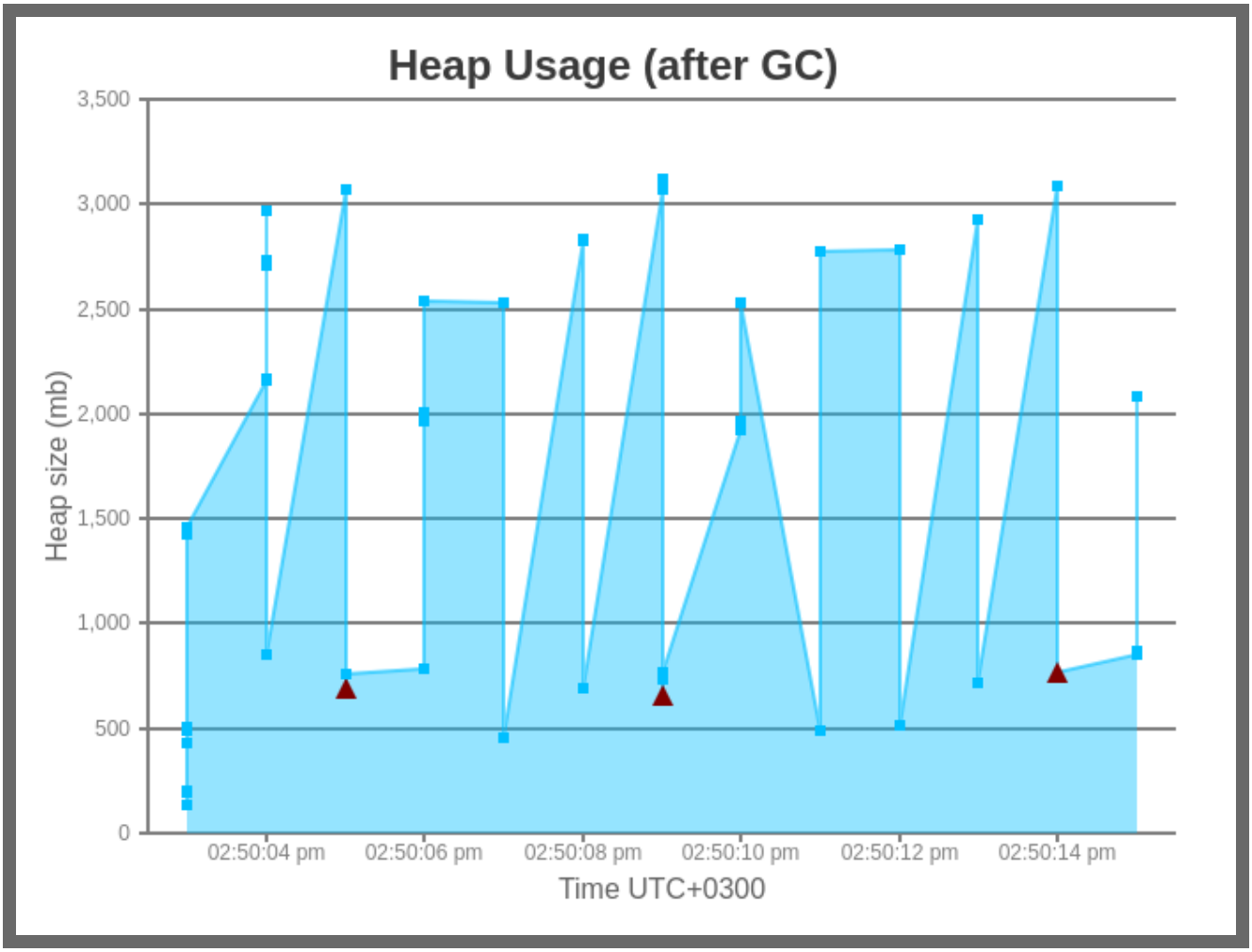
Duration (secs)	No. of GCs	Percentage
0 - 0.1	35	76.087%
0.1 - 0.2	9	95.652%

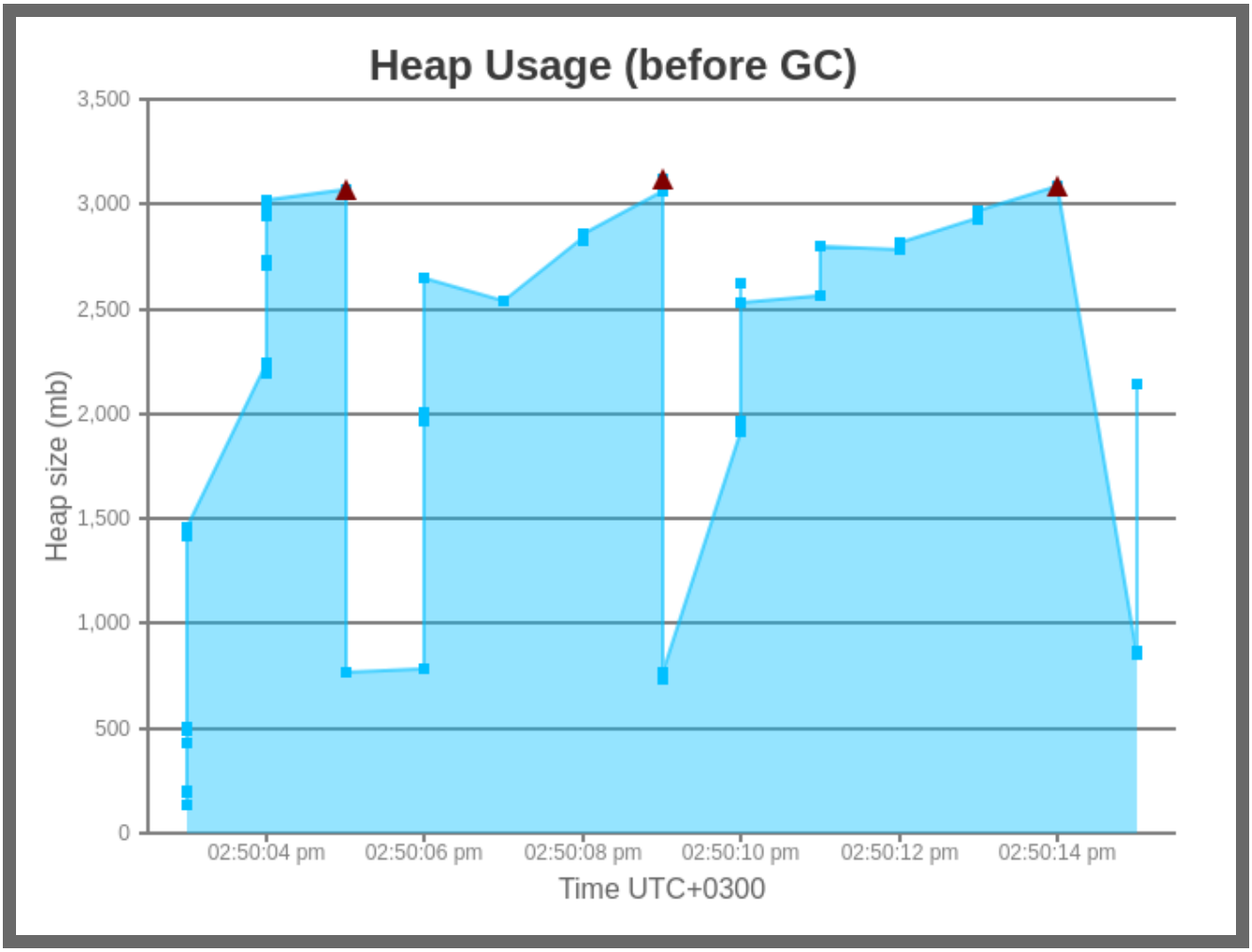
0.2 - 0.3	2	100.0%

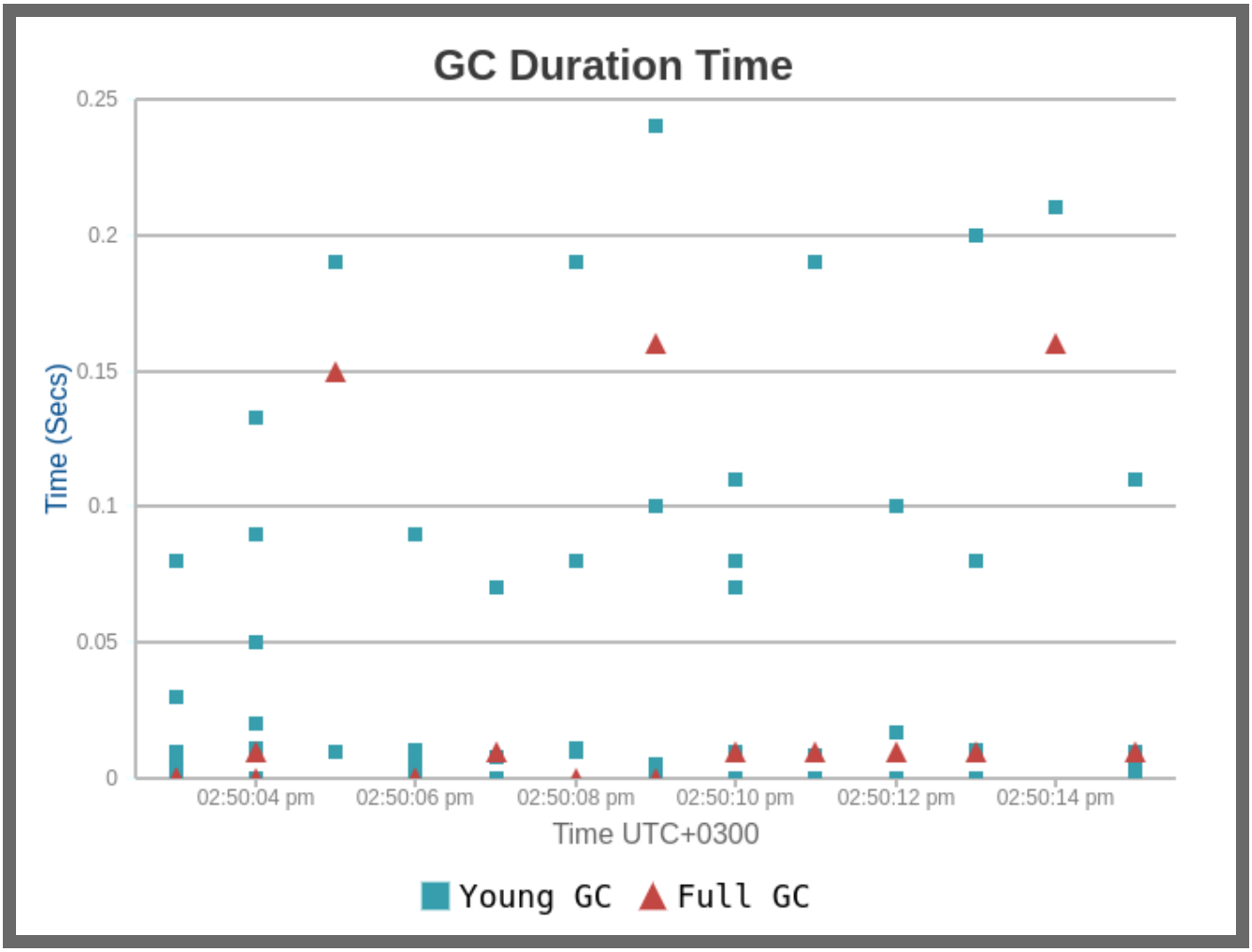


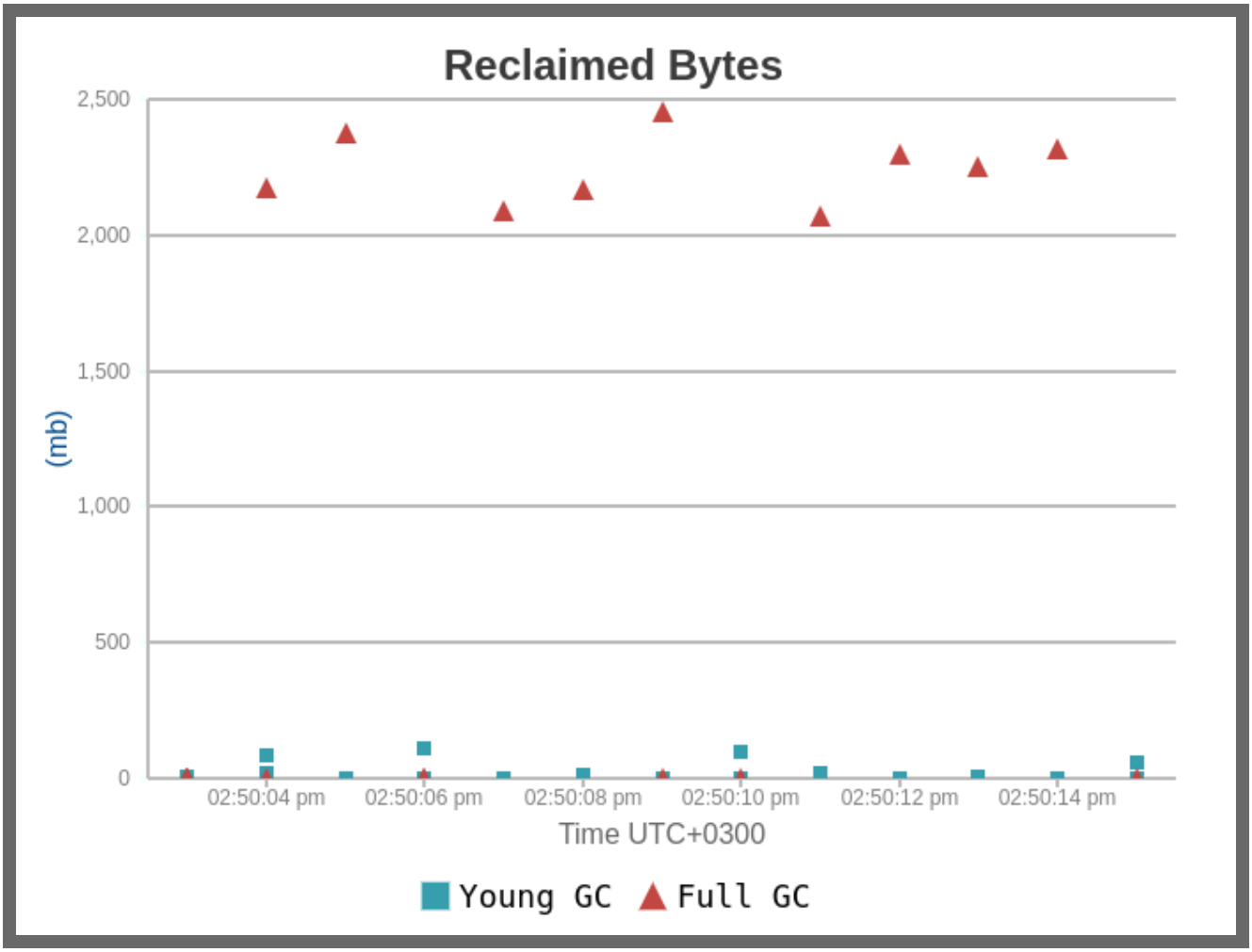
Interactive Graphs

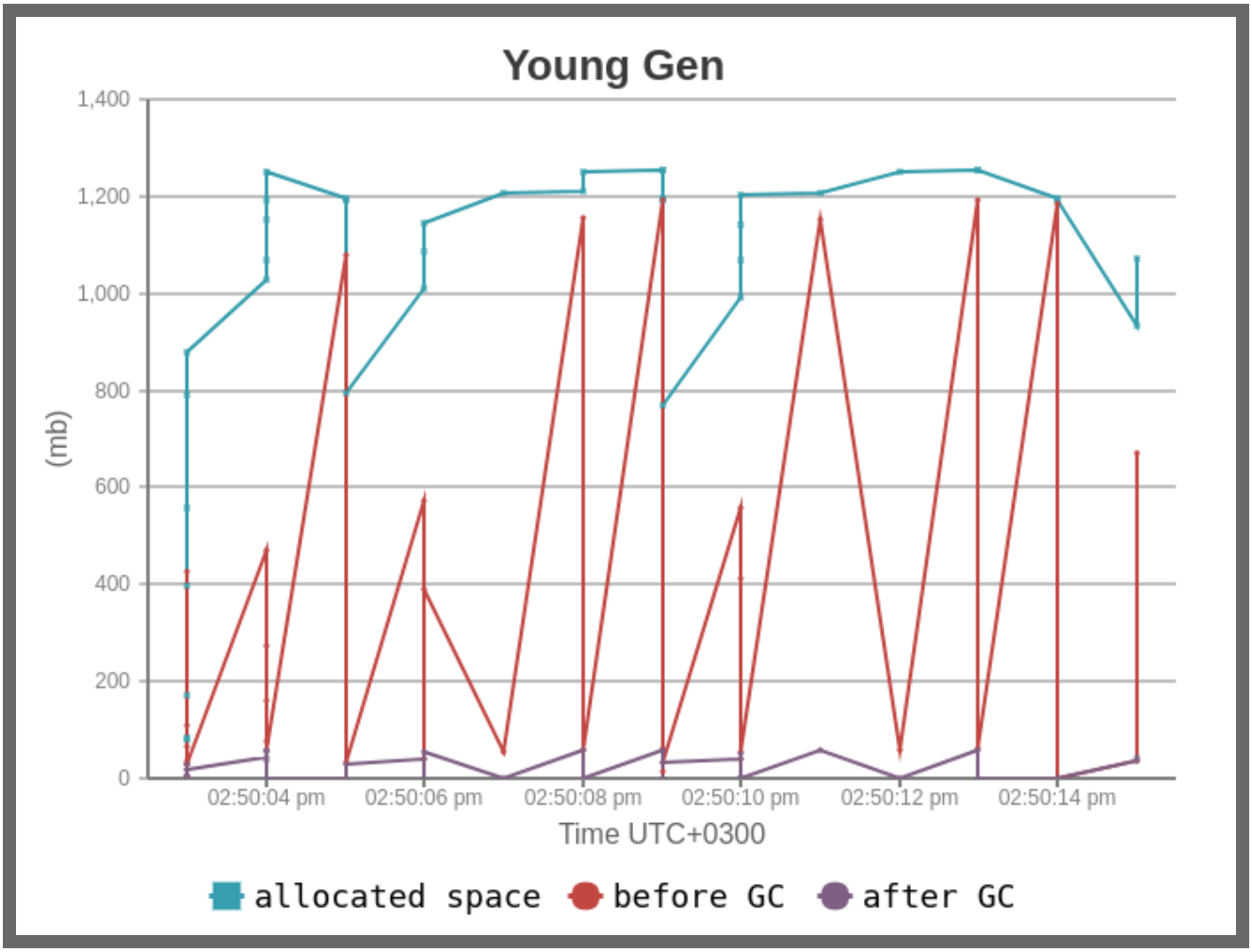
(All graphs are zoomable)

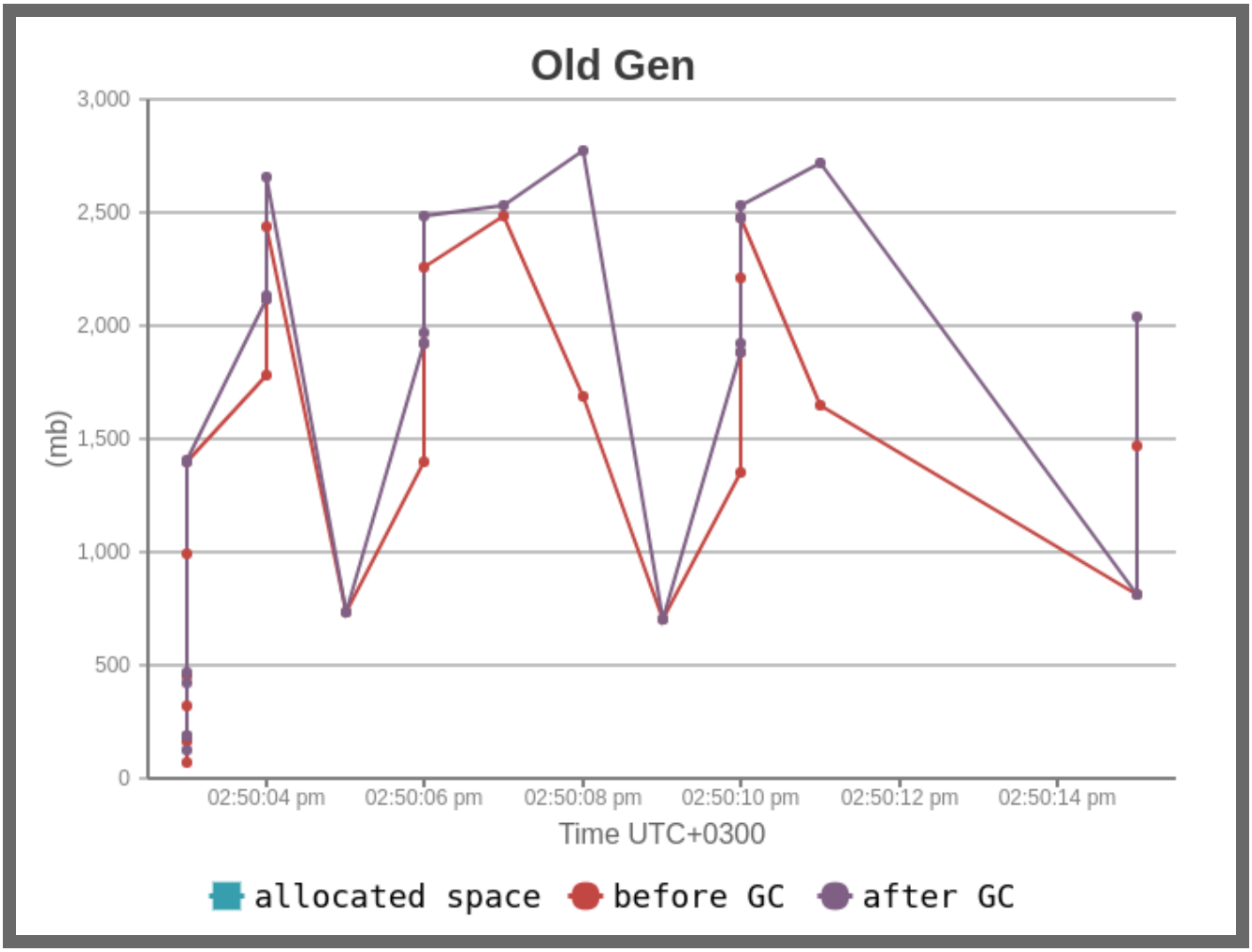


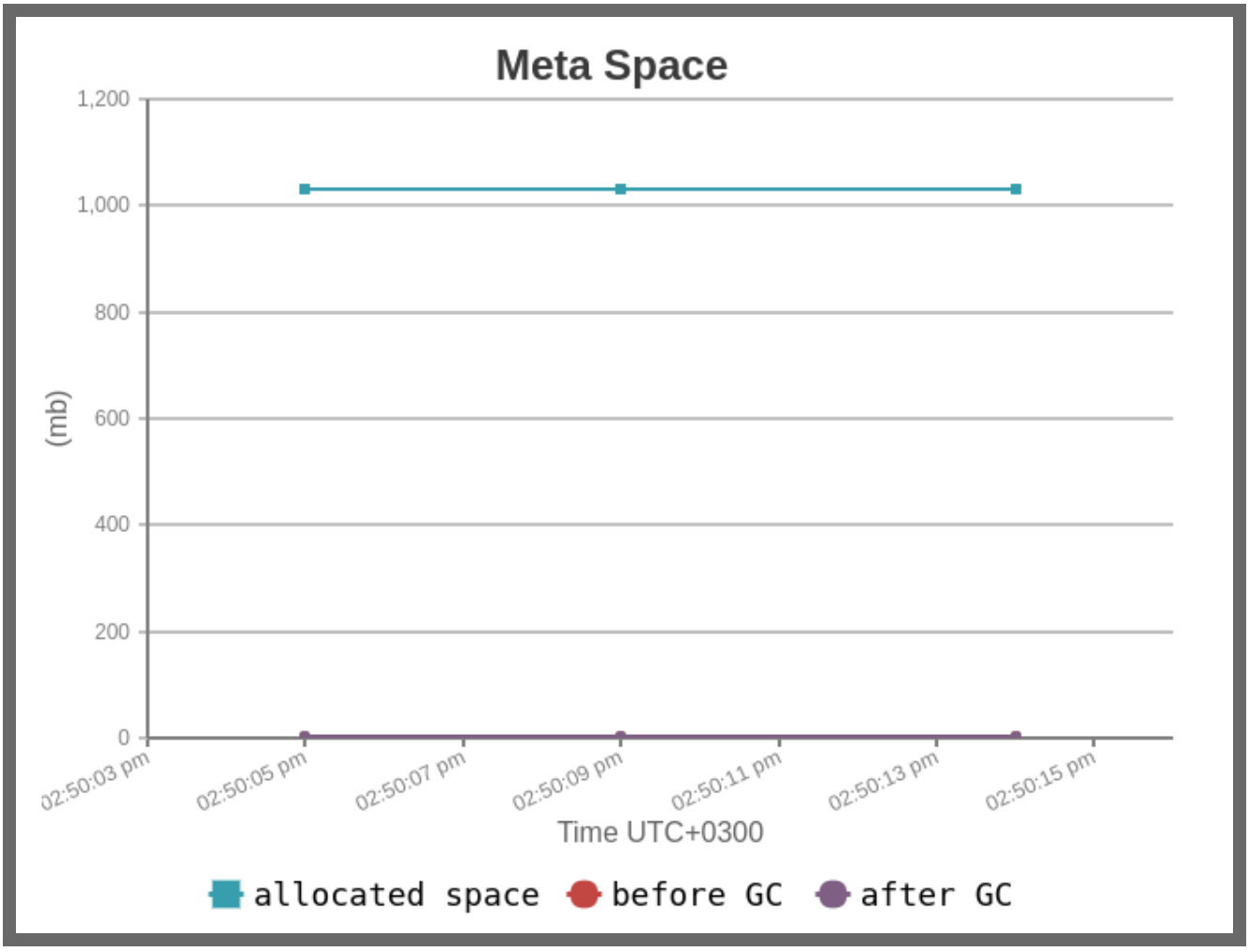

Heap Usage (before GC)
3,500
3,000
2,500
2,000
1,500
1,000
500
0
Heap size (mb)
02:50:04 pm
02:50:06 pm
02:50:08 pm
02:50:10 pm
02:50:12 pm
02:50:14 pm
Time UTC+0300

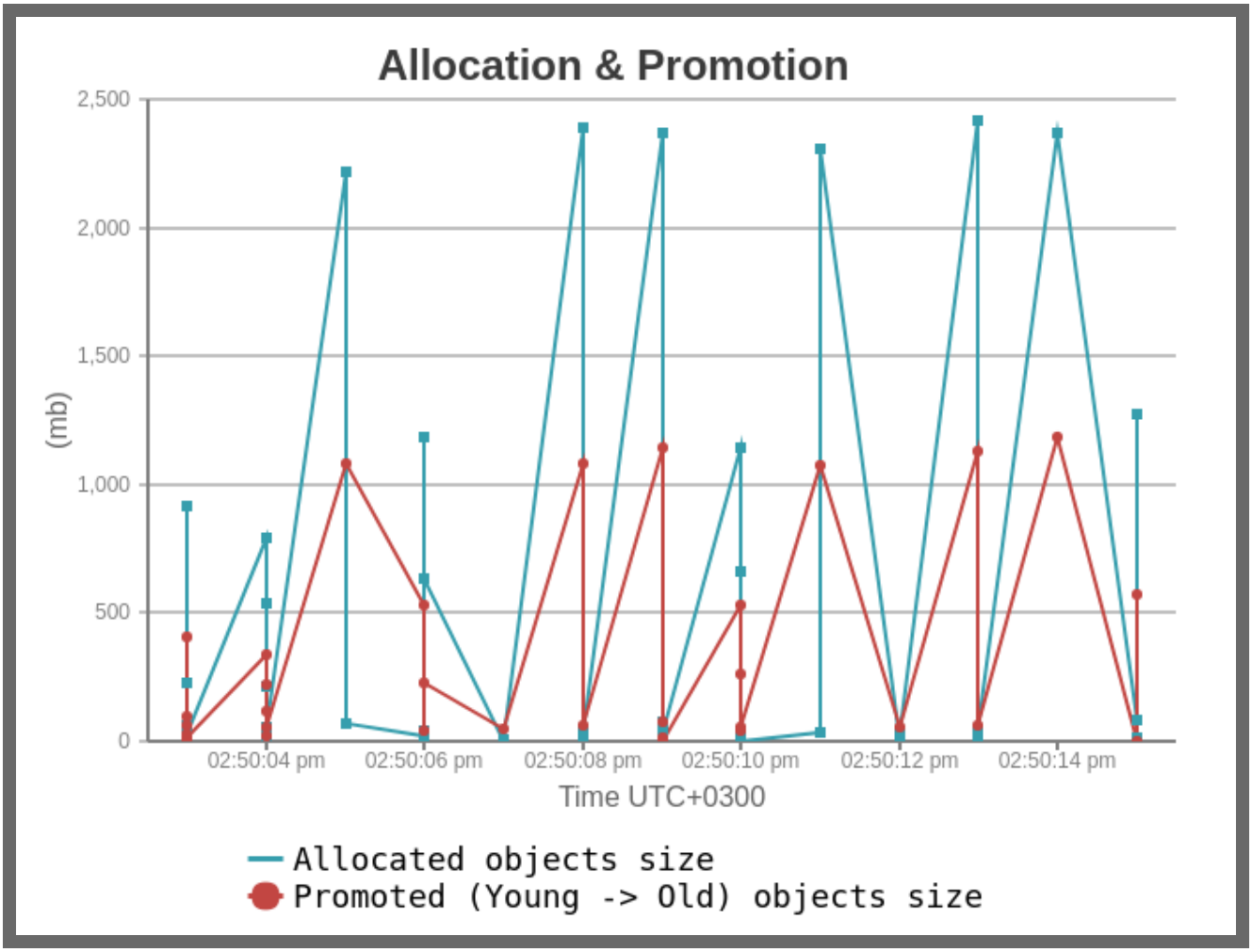

GC Duration Time
0.25
0.2
0.15
0.1
0.05
0
Time (Secs)
02:50:04 pm
02:50:06 pm
02:50:08 pm
02:50:10 pm
02:50:12 pm
02:50:14 pm
Time UTC+0300
Young GC
Full GC


Reclaimed Bytes
2,500
2,000
1,500
1,000
500
0
(mb)
02:50:04 pm
02:50:06 pm
02:50:08 pm
02:50:10 pm
02:50:12 pm
02:50:14 pm
Time UTC+0300
Young GC
Full GC

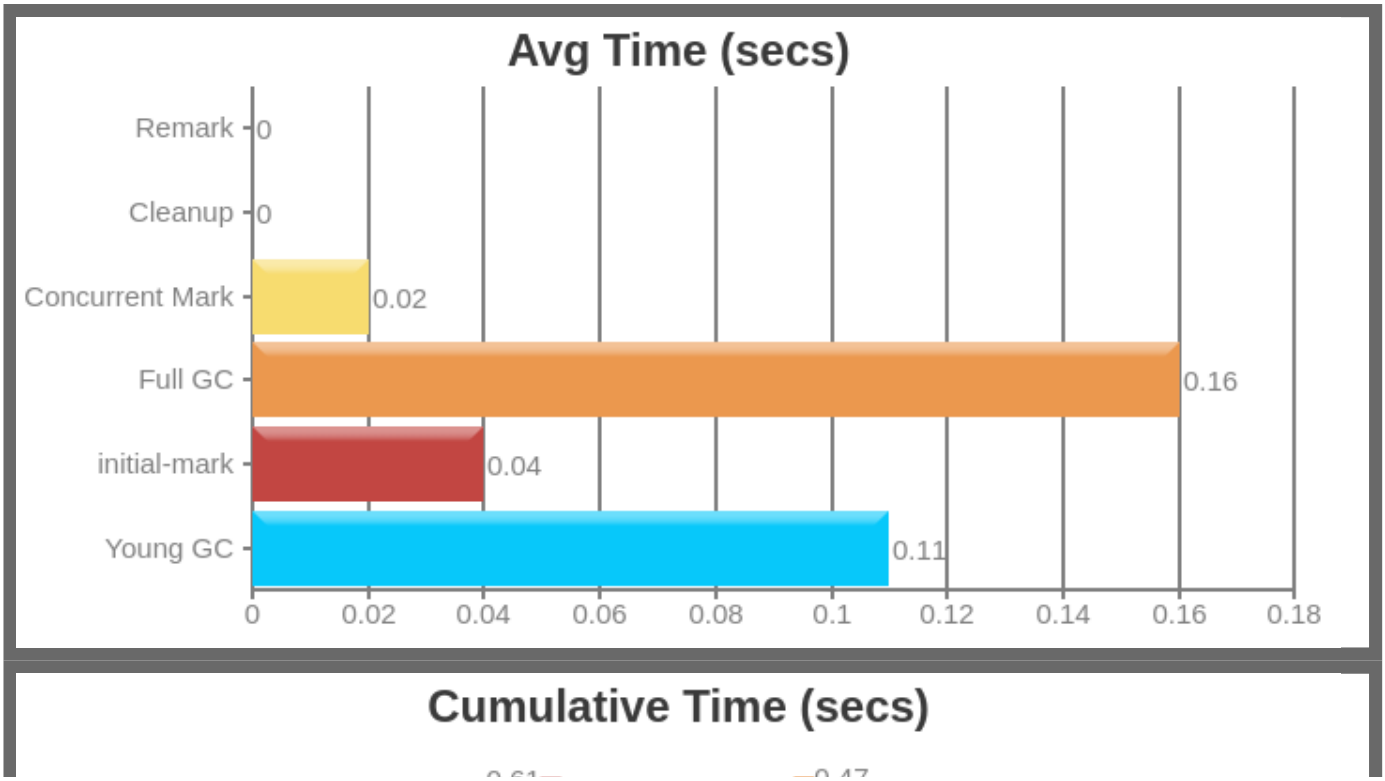

Young Gen
1,400
1,200
1,000
800
600
400
200
0
(mb)
02:50:04 pm
02:50:06 pm
02:50:08 pm
02:50:10 pm
02:50:12 pm
02:50:14 pm
Time UTC+0300
allocated space
before GC
after GC

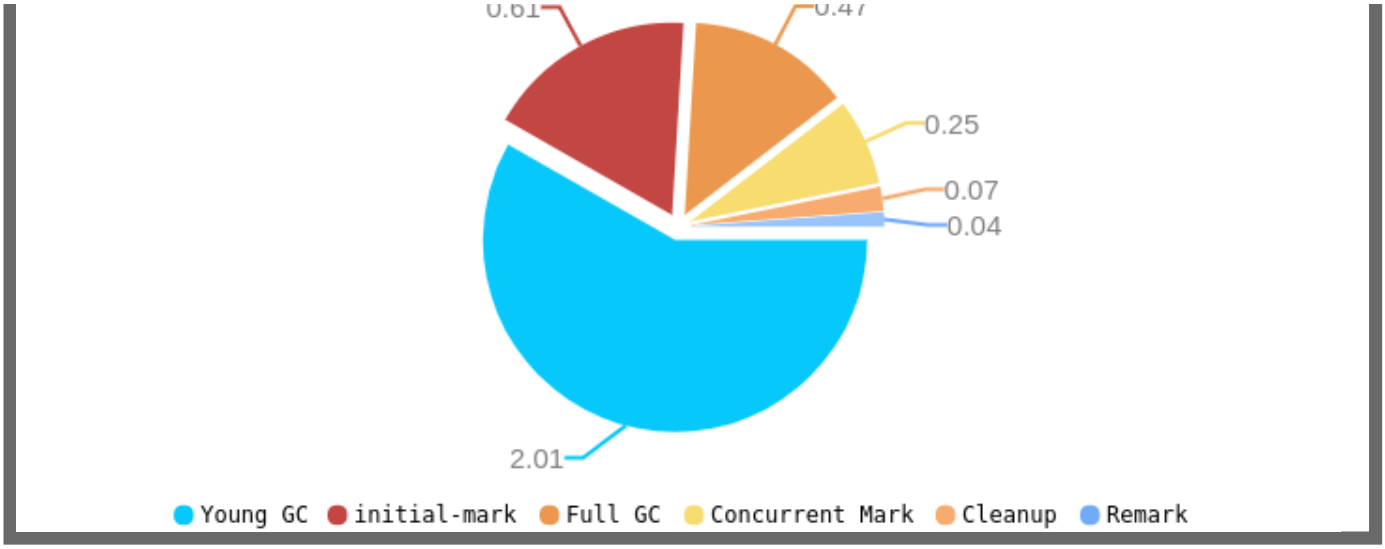

Old Gen
3,000
2,500
2,000
1,500
1,000
500
0
(mb)
02:50:04 pm
02:50:06 pm
02:50:08 pm
02:50:10 pm
02:50:12 pm
02:50:14 pm
Time UTC+0300
allocated space
before GC
after GC


Meta Space
1,200
1,000
800
600
400
200
0
(mb)
02:50:03 pm
02:50:05 pm
02:50:07 pm
02:50:09 pm
02:50:11 pm
02:50:13 pm
02:50:15 pm
Time UTC+0300
allocated space
before GC
after GC



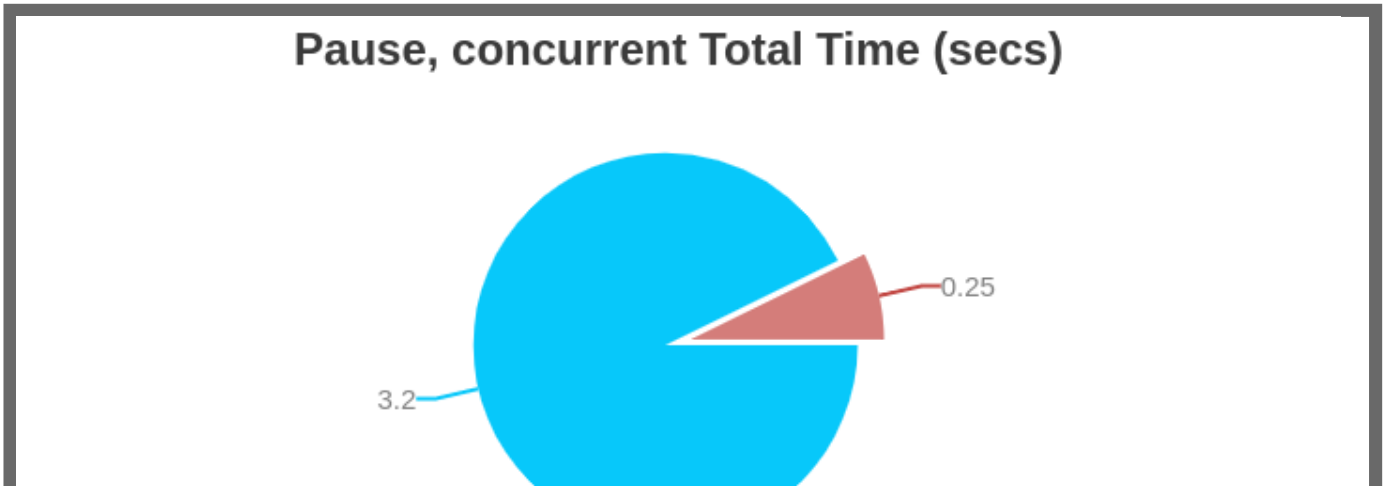
🔓 G1 Collection Phases Statistics

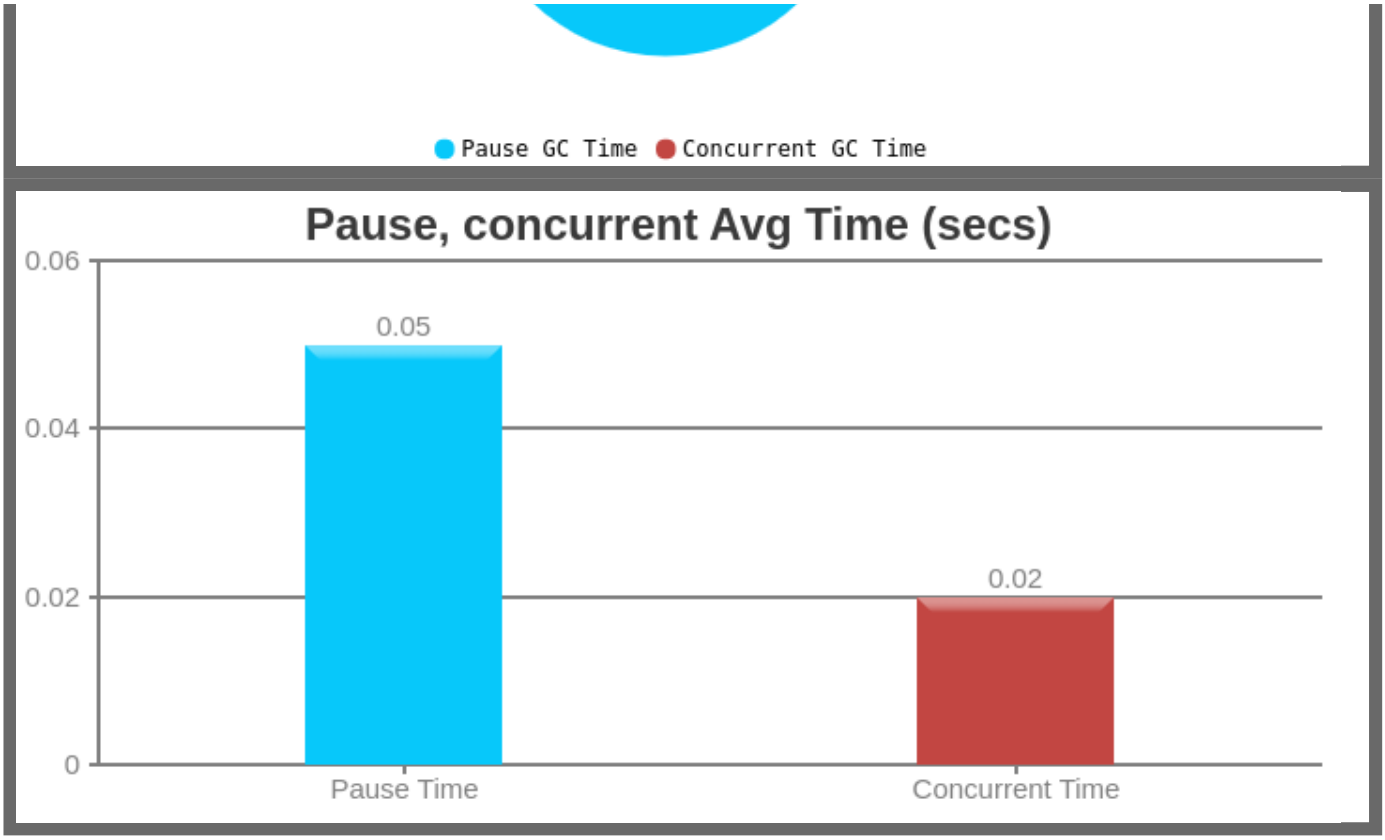




	Young GC	initial-mark	Full GC	Concurrent Mark	Cleanup	Remark	Total
Count	18	17	3	16	16	16	86
Total GC Time	2 sec 10 ms	610 ms	470 ms	247 ms	70 ms	40 ms	3 sec 447 ms
Avg GC Time	112 ms	36 ms	157 ms	15 ms	4 ms	3 ms	40 ms
Avg Time std dev	72 ms	36 ms	5 ms	31 ms	5 ms	4 ms	61 ms
Min/Max Time	0 / 240 ms	0 / 100 ms	0 / 160 ms	0 / 133 ms	0 / 10 ms	0 / 10 ms	0 / 240 ms
Avg Interval Time	727 ms	739 ms	4 sec 513 ms	788 ms	789 ms	788 ms	859 ms

G1 GC Time





Pause Time

Total Time	3 sec 200 ms
Avg Time	46 ms
Std Dev Time	64 ms
Min Time	0
Max Time	240 ms

Concurrent Time

Total Time	247 ms
Avg Time	15 ms
Std Dev Time	31 ms
Min Time	0
Max Time	133 ms

Object Stats

(These are perfect micro-metrics (https://blog.gceasy.io/2017/05/30/improving-your-performance-reports/) to include in your performance reports)

Total created bytes	22.11 gb
Total promoted bytes	10.39 gb
Avg creation rate	1.78 gb/sec
Avg promotion rate	858.64 mb/sec

Memory Leak

No major memory leaks.

(**Note:** there are 8 flavours of OutOfMemoryErrors (https://tier1app.files.wordpress.com/2014/12/outofmemoryerror2.pdf). 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

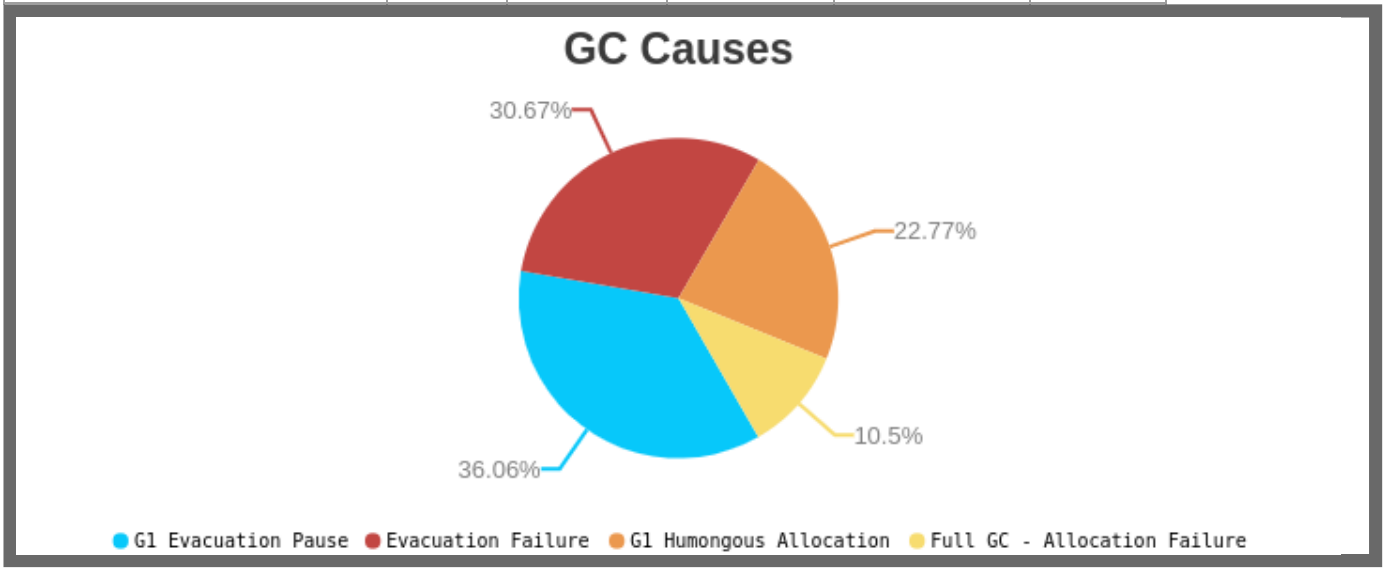
(To learn more about SafePoint duration, click here (https://blog.gceasy.io/2016/12/22/total-time-for-which-application-threads-were-stopped/))

Not Reported in the log.

GC Causes

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

Cause	Count	Avg Time	Max Time	Total Time	Time %
G1 Evacuation Pause	6	267 ms	233 ms	1 sec 601 ms	36.06%
Evacuation Failure	9	151 ms	233 ms	1 sec 362 ms	30.67%
G1 Humongous Allocation	20	51 ms	211 ms	1 sec 11 ms	22.77%
Full GC - Allocation Failure	3	155 ms	163 ms	466 ms	10.5%
Total	38	n/a	n/a	4 sec 440 ms	100.0%



Tenuring Summary

Not reported in the log.

Command Line Flags

-XX:G1MaxNewSizePercent=80 -XX:G1NewSizePercent=30 -XX:InitialHeapSize=260849920 -XX:MaxHeapSize=4173598720 -XX:+PrintGC -XX:+PrintGCDateStamps -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:SurvivorRatio=20 -XX:+UnlockExperimentalVMOptions -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseG1GC