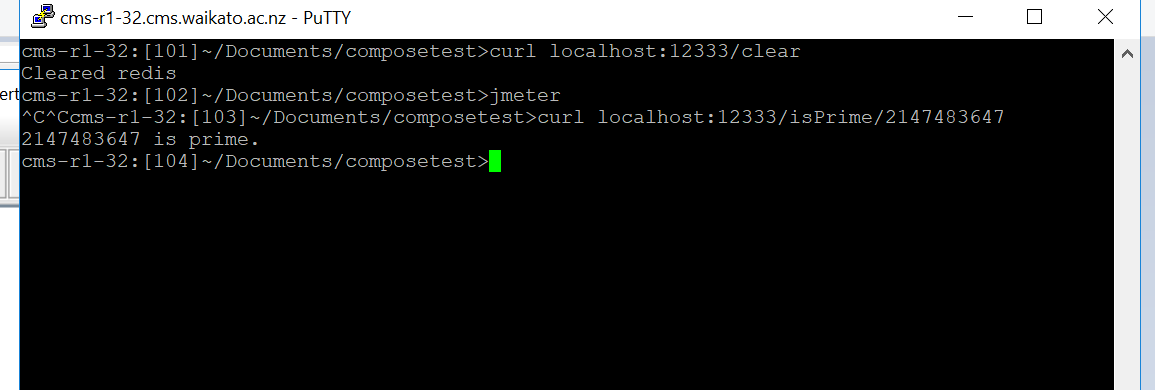
**Testing**

Edge case testing

Test by inputting the numbers: a, -1 , 1 ,2,12 , 199 into localhost:123333/isPrime/<number>

|  |  |  |
| --- | --- | --- |
| Input | Expected Output | Actual Output |
| a | Server error | Server error |
| -1 | Server error | Server error |
| 2 | Prime | Prime |
| 12 | Not prime | Not prime |
| 199 | Prime | Prime |
| 2147483647 | Prime | Prime |



Result: passed all testing correctly. When I entered -1 and a I got a 404 not found error which is fine because neither of those are valid URIs.

Testing primeStored

I’m using white box edge case testing to test the primesStored API.

I’m using the isPrime function to add the prime numbers to the redis list. Then I’m calling the primesStored API and seeing if the output matches my expected output.

localhost:12333/isPrime/7

localhost:12333/isPrime/7

localhost:12333/isPrime/199

localhost:12333/isPrime /2147483647

localhost:12333/isPrime /2147483647

localhost:12333/isPrime/a

localhost:12333/isPrime/12

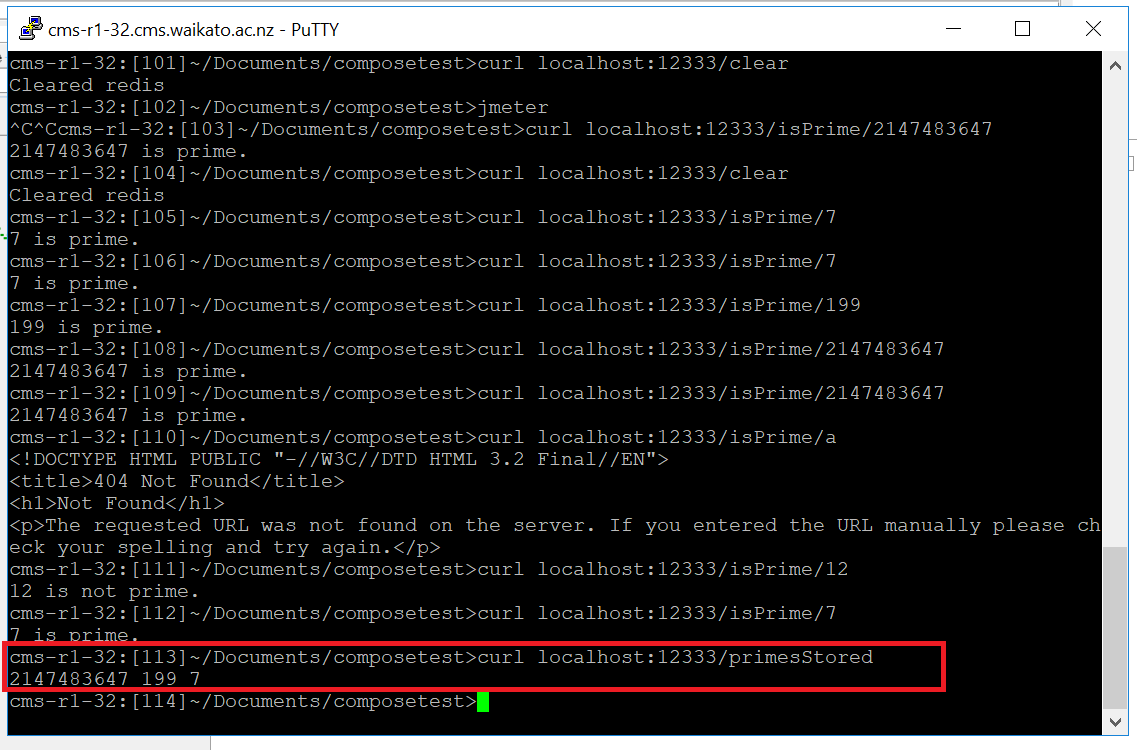
localhost:12333/isPrime/7

Finally I will call localhost:12333/primesStored

My output should be a list of the prime numbers with no duplicates

Like this: 214748367 199 7

Here’s the result of my testing



Output is what I expected.  
The program passed all its unit tests so I can move on to the stress testing.

Stress test results

* 1. Vcpu

Scenario 1

Webserver handled the scenario one stress test easily. Before when I was using a different algorithm to determine if a number was prime it was struggling, taking a long time to respond to the requests. After updating the algorithm with a much faster one I got from www.geeksforgeeks.org/analysis-different-methods-find-prime-number-python/ it had no problems quickly resolving the requests. I then retested this updated code using my unit tests described above. All stress tests were preformed with a ramp up timer of 1 second and using 50 threads

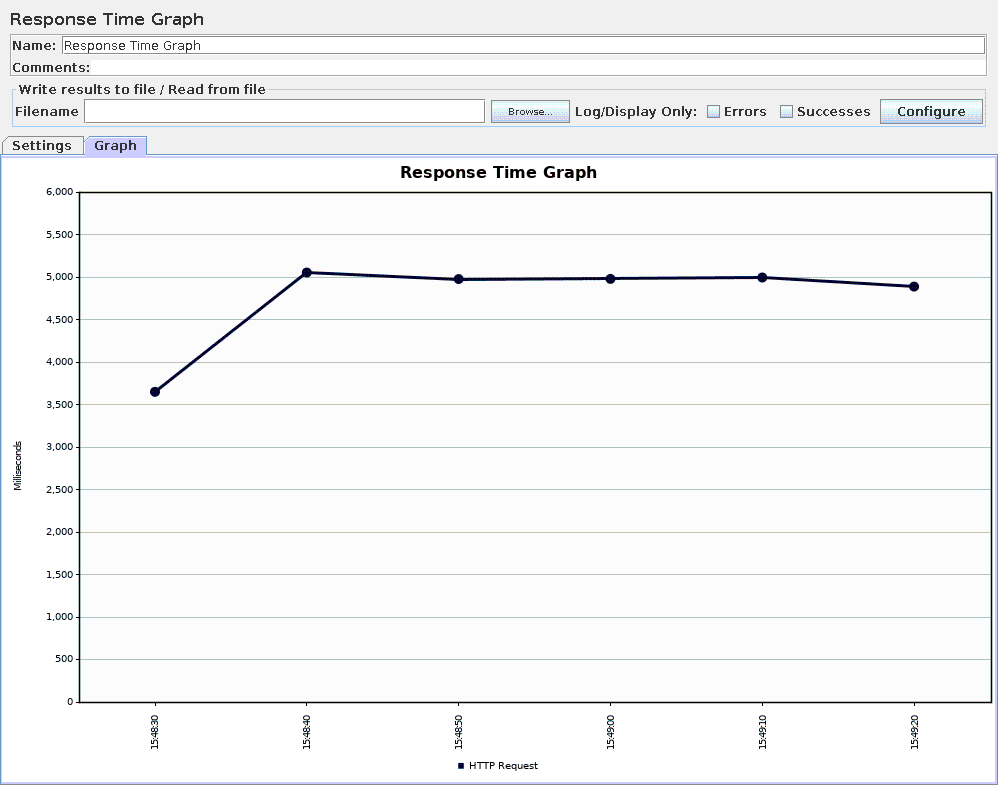
Throughput graph

Figure 1-response time Scenario 1 CPU = 0.1

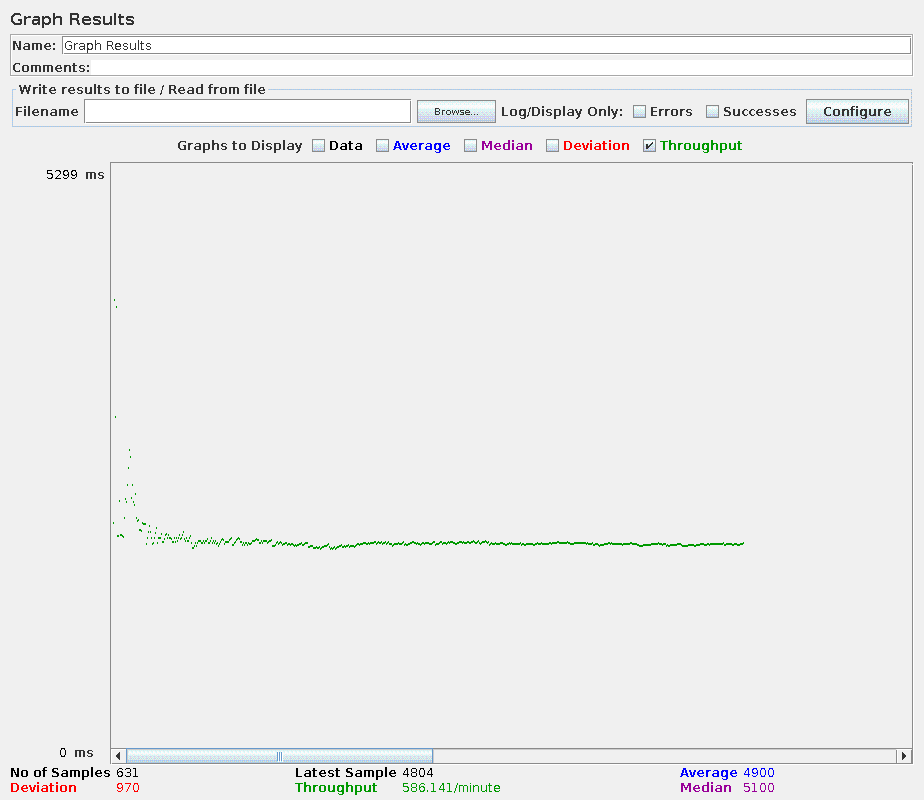
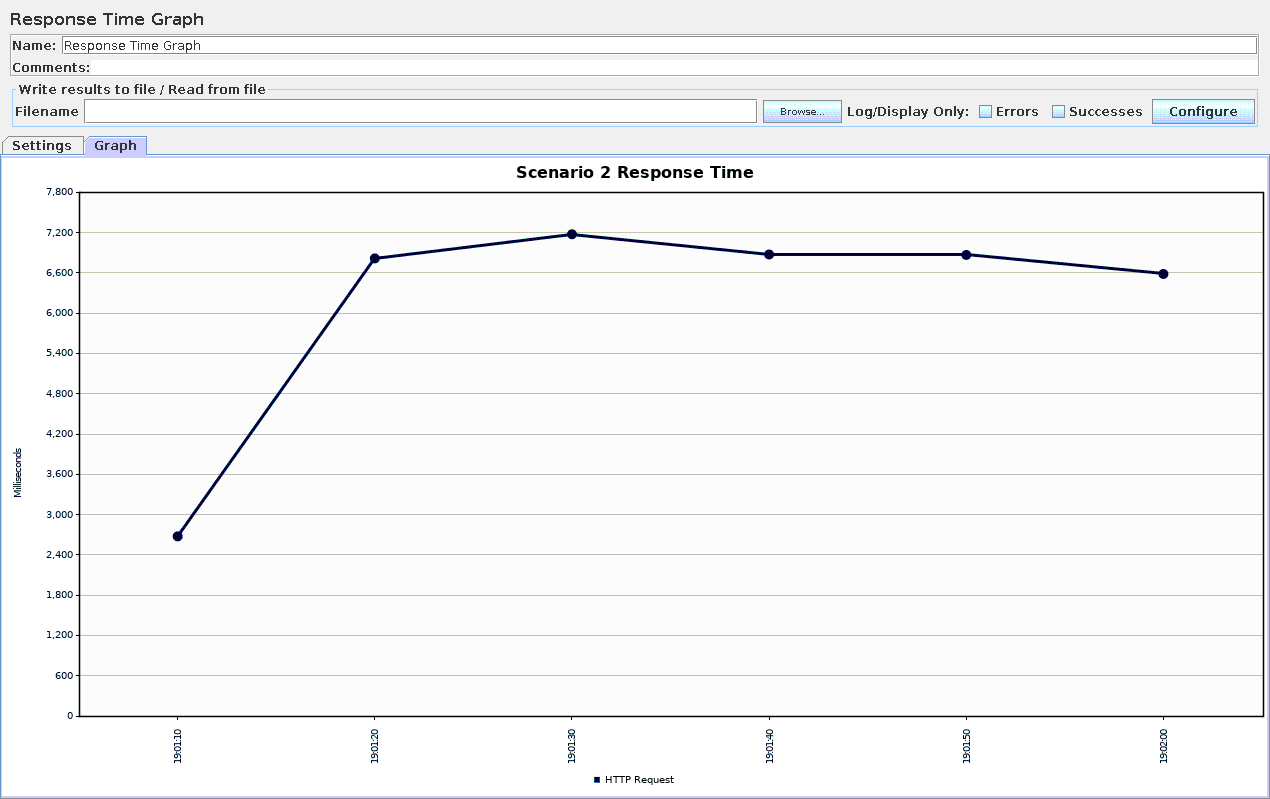
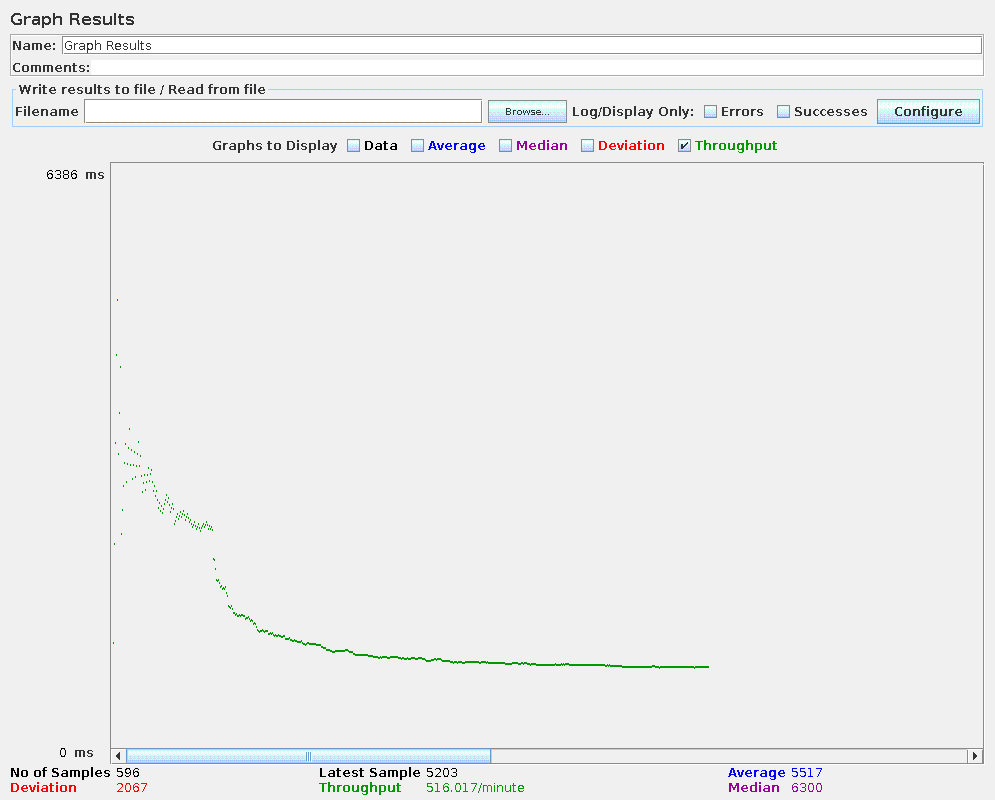


Figure 2-throughput scenario 1 CPU 0.1

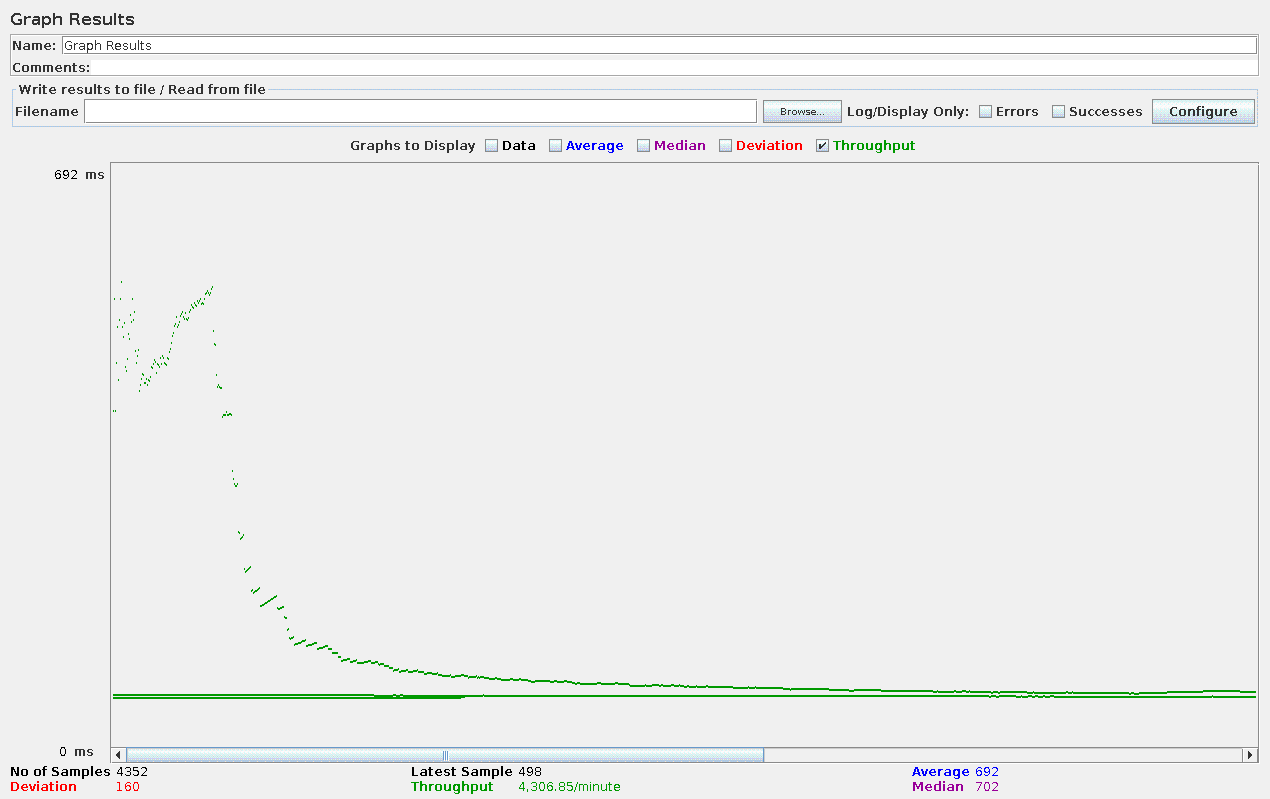
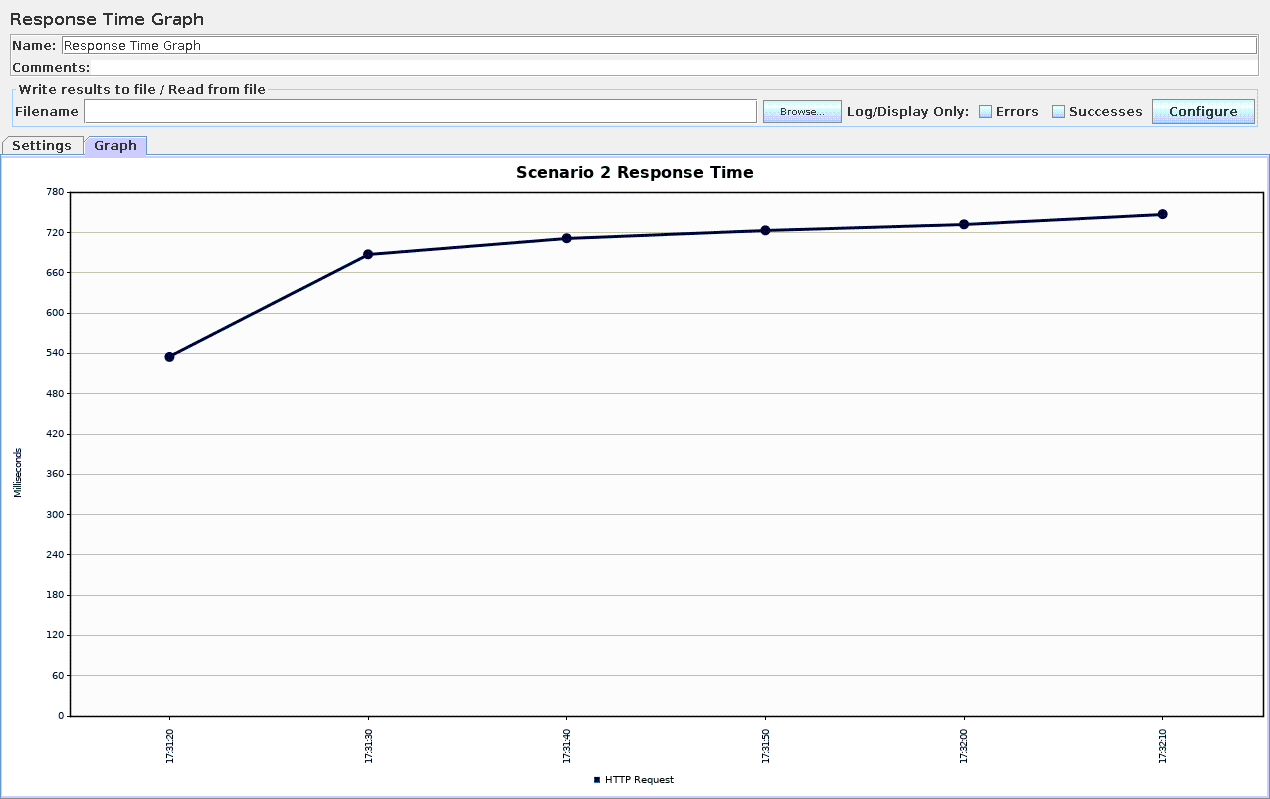
As shown in the graph above throughput stayed consistent through the stress test, only at the start was there much variability in throughput. Response time remained also remained consistent which was expected as we are invoking the same function over and over. Curiously both response time and throughput recorded better results at the start of the stress test. This may be due to the ramp up time for the threads.

Scenario 2



The second scenario was more complicated. Firstly the isPrime function was invoked for the numbers 0 to 100, then the primesStored URI was repeatedly invoked for 60 seconds. At the start the response time is low similar to what occurred during scenario 1 testing then once we switch over to the new URI response times increase as the primesStored function uses more processing power to compute a result. This is mirrored in the throughput results; at the start throughput is high and inconsistent. When the first type of http requests ends and is replaced by the more processor intensive primesStored request throughput slows down significantly until it stabilised at around 500 requests per minute.

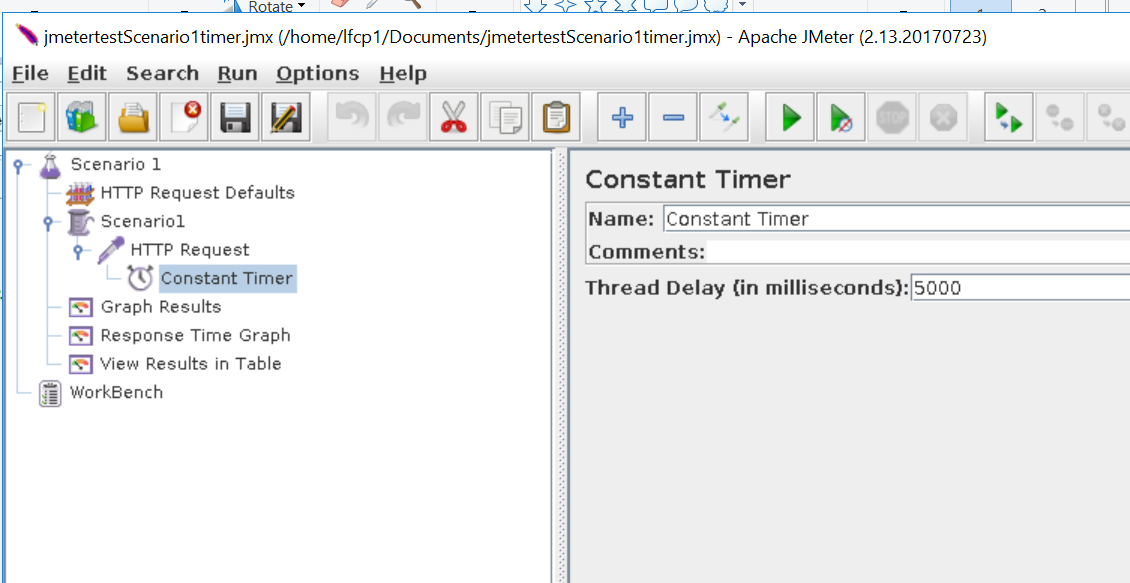
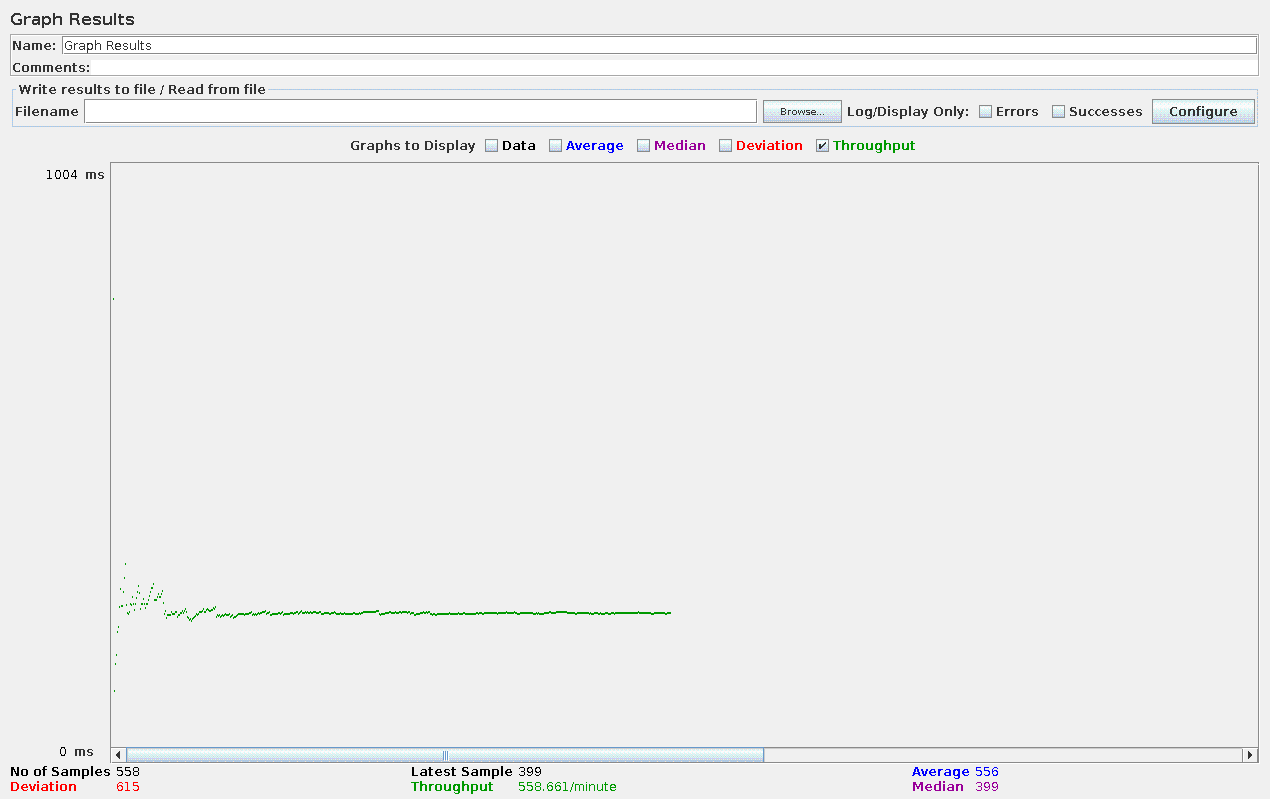
Changing the CPU utilisation

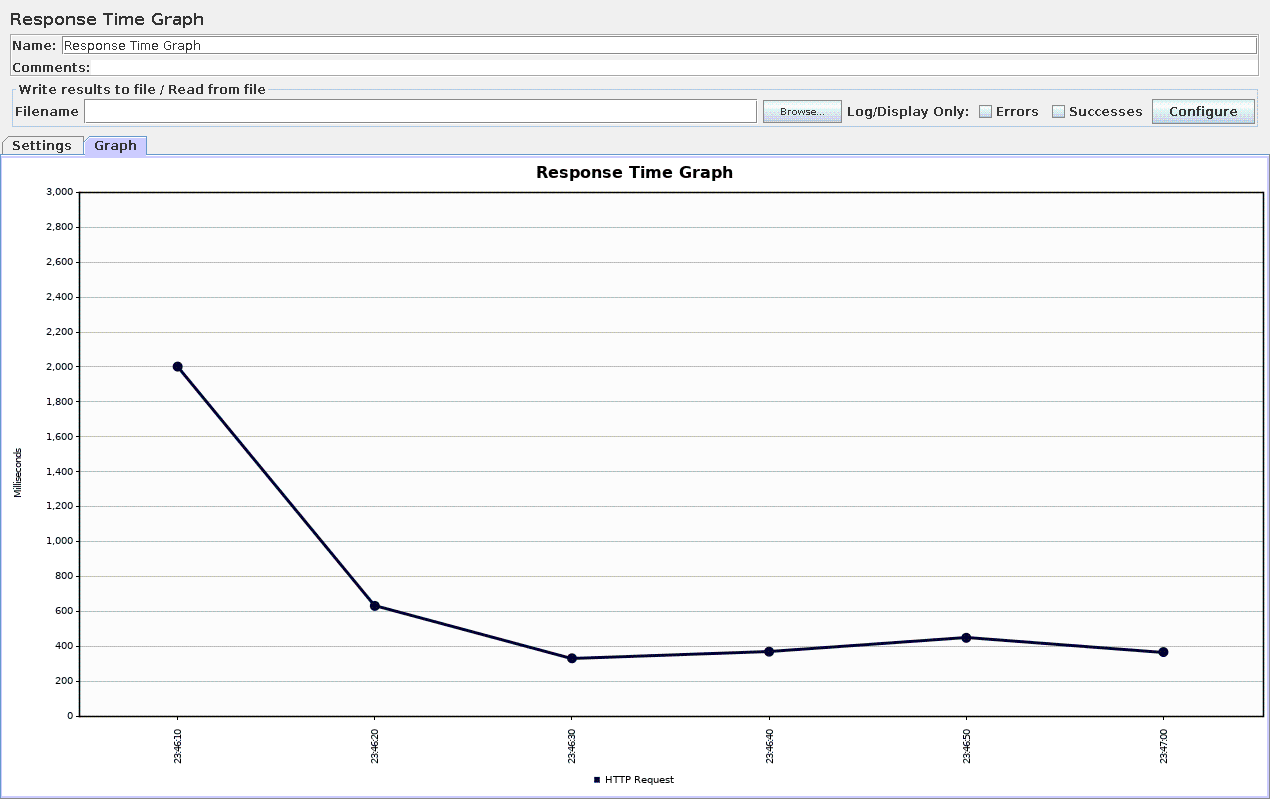
I played around with setting the CPU restriction at different levels 0.5, 0.8 etc and trying out stress tests. The most interesting result was when the CPU was unrestricted and scenario 2 was used as the test plan. The graphs below show the throughput of and the response time during this stress test. 

Throughput increased dramatically, almost by a factor of 10 compared to the restricted CPU stress test documented above. Response time also decreased by about a factor of 10. The functions were taking far less time execute with the increased CPU processing available. This roughly factor of 10 change makes sense when we consider in the previous tests the CPU was pinned at 10% of its capability while now we are able to exploit 100% of its processing power. The decreased response time means that the http requests are spending less time sitting in a queue, waiting for the requests ahead of them to be processed.

CPU pinned at 0.1 and use of a 5000ms timer

For the last stress test shown on this report, I added a 5000ms timer which would execute a 5000ms pause between each request. The CPU utilisation was changed back to 0.1 in the docker-compose.yml file.





Throughput is similar to the scenario one stress test without the timer. This suggests that the server is working flat out and always has requests in its queue. Even by increasing the time between requests the server is not idle at anytime during the stress test. The real change comes in response times now that there are less requests being sent, the queue that the requests sit in before being processed by the server is much shorter so they spend less time waiting to be processed before being returned.

Conclusion