

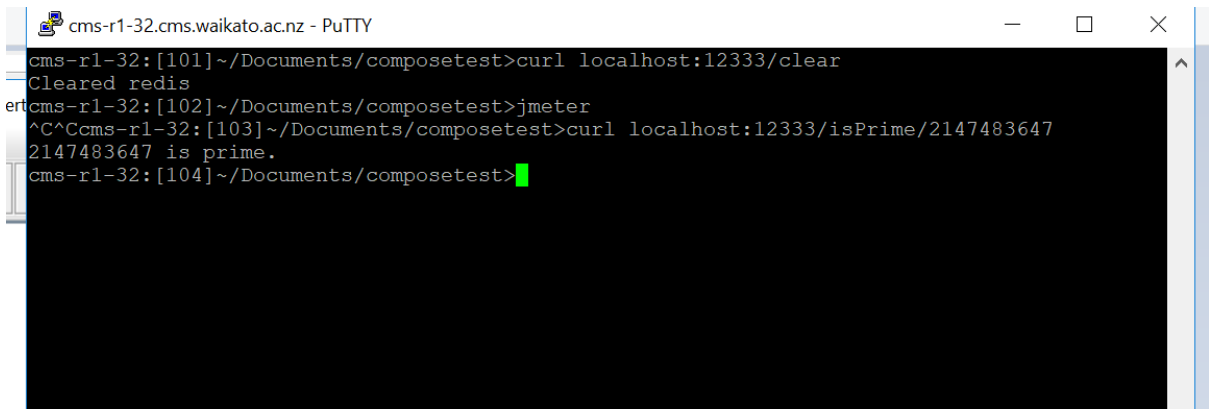
## GITHUB LINK:

## Testing

Edge case testing

Test by inputting the numbers: a, -1, 1, 2, 12, 199 into localhost:12333/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



```
cms-r1-32.cms.waikato.ac.nz - PuTTY
cms-r1-32:[101]~/Documents/composetest>curl localhost:12333/clear
Cleared redis
cms-r1-32:[102]~/Documents/composetest>jmeter
^C^Ccms-r1-32:[103]~/Documents/composetest>curl localhost:12333/isPrime/2147483647
2147483647 is prime.
cms-r1-32:[104]~/Documents/composetest>
```

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

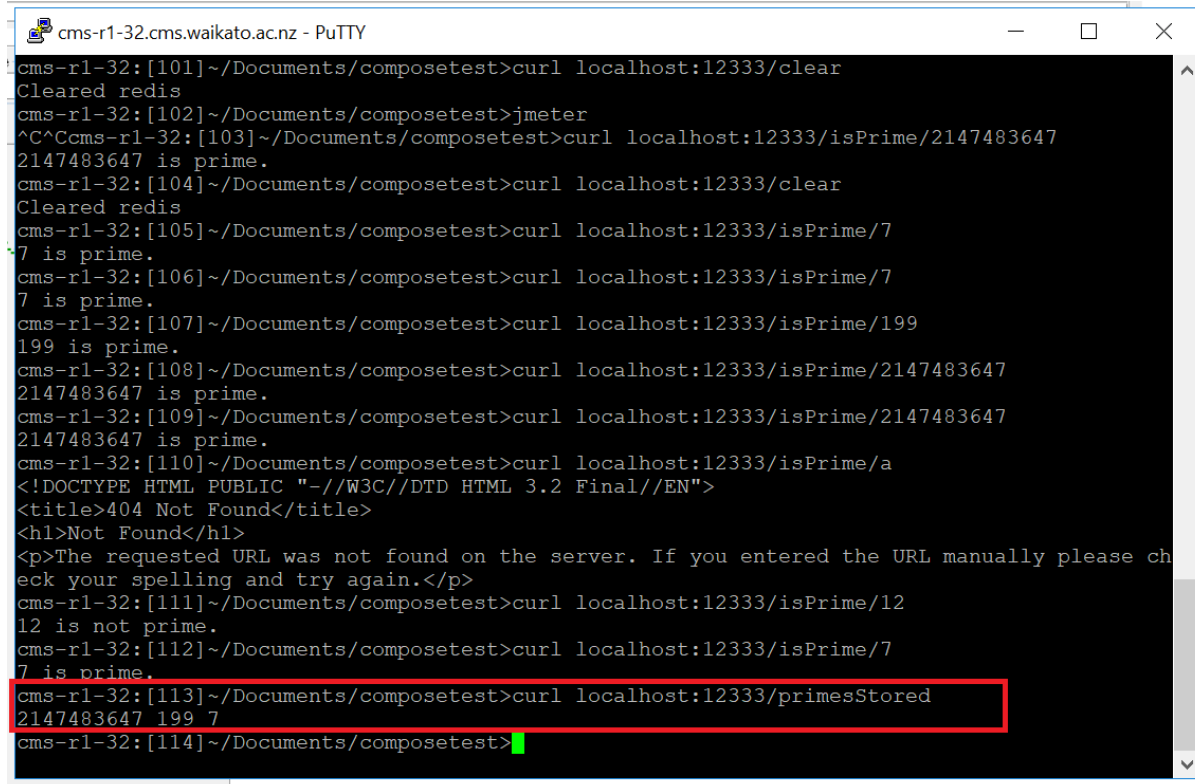
localhost:12333/isPrime/7

Finally I will call localhost:12333/primesStored

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

Like this: 2147483647 199 7

Here's the result of my testing



```
cms-r1-32:[101]~/Documents/composetest>curl localhost:12333/clear
Cleared redis
cms-r1-32:[102]~/Documents/composetest>jmeter
^C^Ccms-r1-32:[103]~/Documents/composetest>curl localhost:12333/isPrime/2147483647
2147483647 is prime.
cms-r1-32:[104]~/Documents/composetest>curl localhost:12333/clear
Cleared redis
cms-r1-32:[105]~/Documents/composetest>curl localhost:12333/isPrime/7
7 is prime.
cms-r1-32:[106]~/Documents/composetest>curl localhost:12333/isPrime/7
7 is prime.
cms-r1-32:[107]~/Documents/composetest>curl localhost:12333/isPrime/199
199 is prime.
cms-r1-32:[108]~/Documents/composetest>curl localhost:12333/isPrime/2147483647
2147483647 is prime.
cms-r1-32:[109]~/Documents/composetest>curl localhost:12333/isPrime/2147483647
2147483647 is prime.
cms-r1-32:[110]~/Documents/composetest>curl localhost:12333/isPrime/a
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server. If you entered the URL manually please check your spelling and try again.</p>
cms-r1-32:[111]~/Documents/composetest>curl localhost:12333/isPrime/12
12 is not prime.
cms-r1-32:[112]~/Documents/composetest>curl localhost:12333/isPrime/7
7 is prime.
cms-r1-32:[113]~/Documents/composetest>curl localhost:12333/primesStored
2147483647 199 7
cms-r1-32:[114]~/Documents/composetest>
```

Output is what I expected.

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

## Stress test results

CPU pinned 0.1

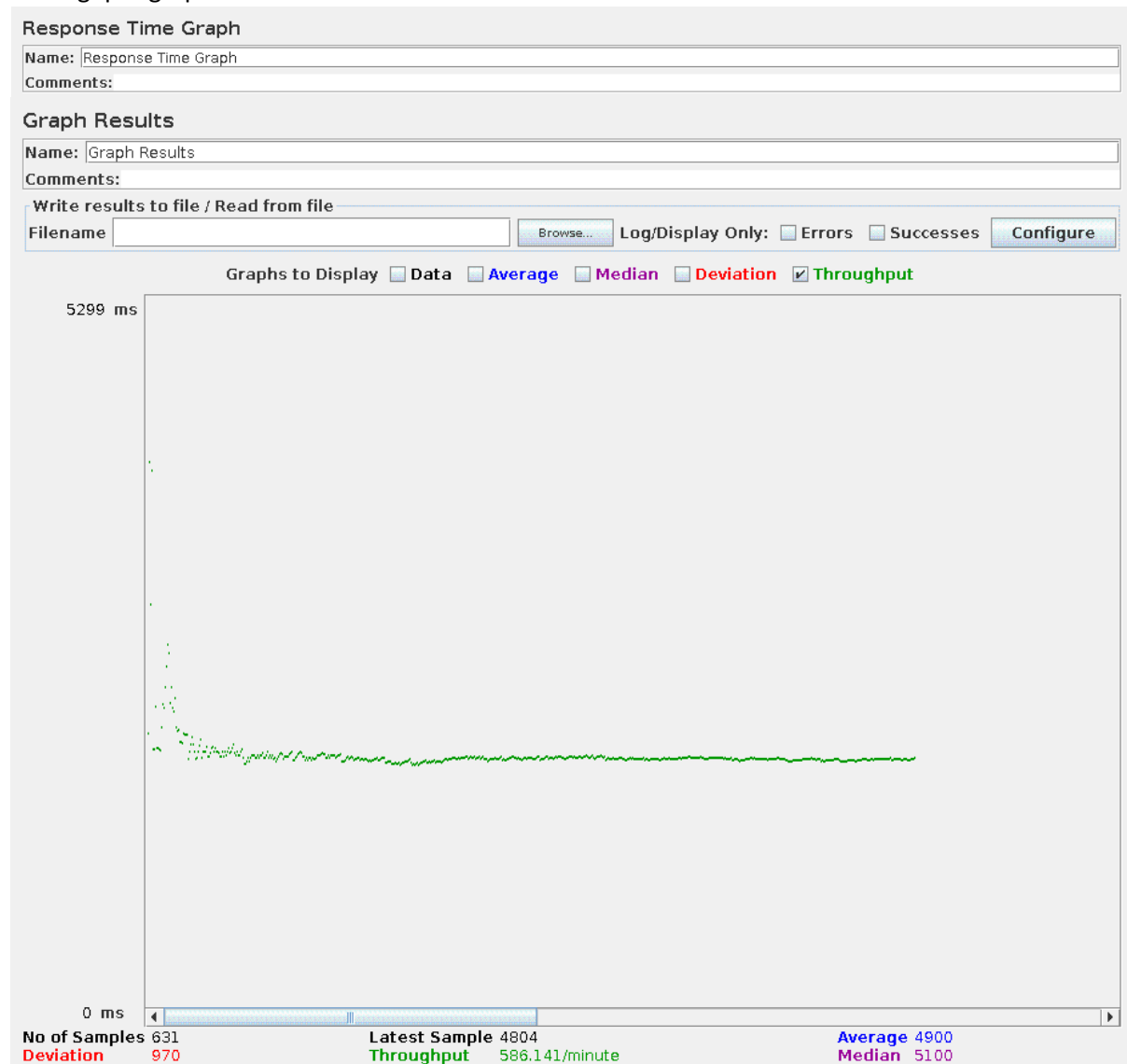
All stress tests showcased in this report were preformed with a ramp up timer of 1 second and using 50 threads.

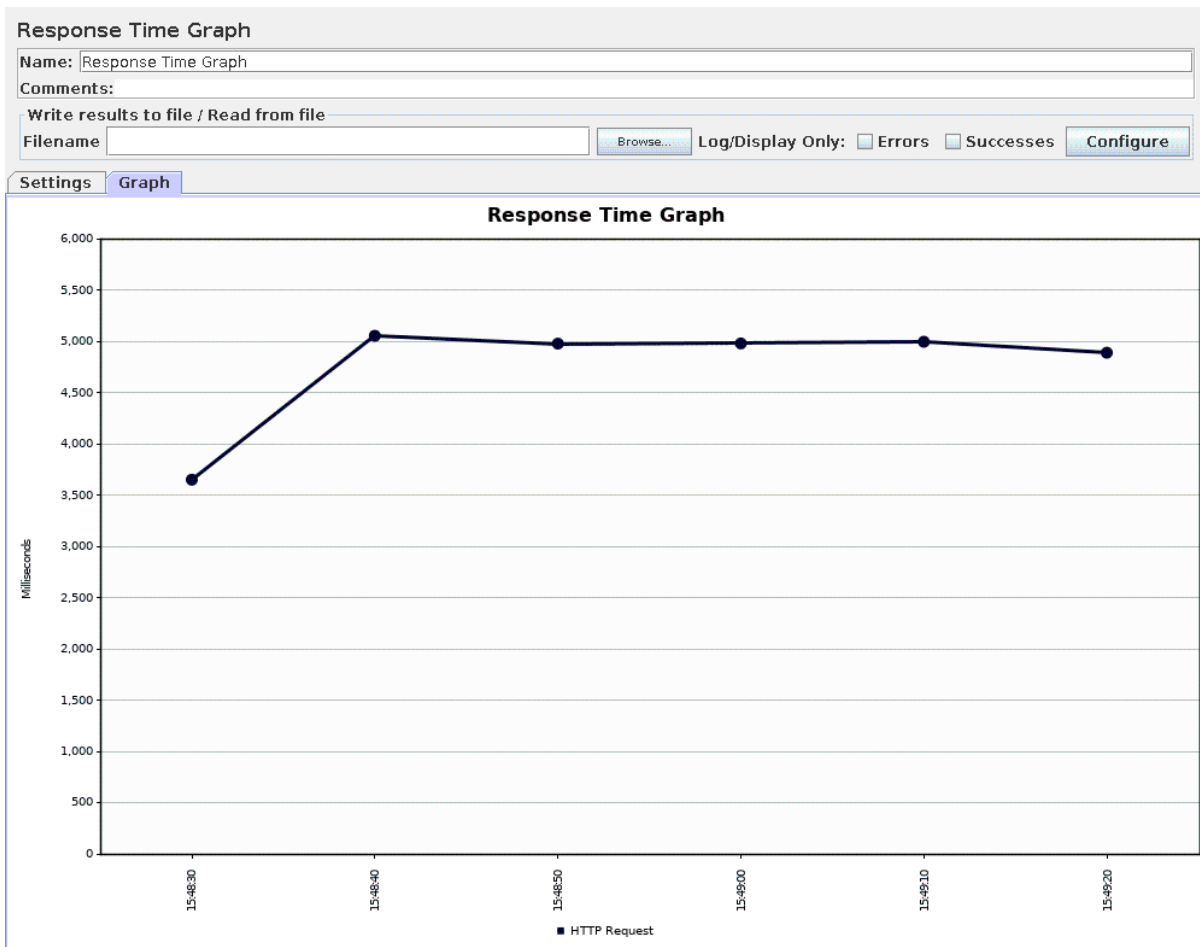
### Scenario 1

When I was using a different algorithm to determine if a number was prime my server was struggling, taking a long time to respond to the requests. After updating the algorithm with a much faster one I adapted from [www.geeksforgeeks.org/analysis-different-methods-find-prime-number-python/](http://www.geeksforgeeks.org/analysis-different-methods-find-prime-number-python/) it had no problems quickly resolving the requests. I retested this updated code using my

unit tests described above and it passed. I then preformed all the following stress tests using the updated isPrime function.

### Throughput graph

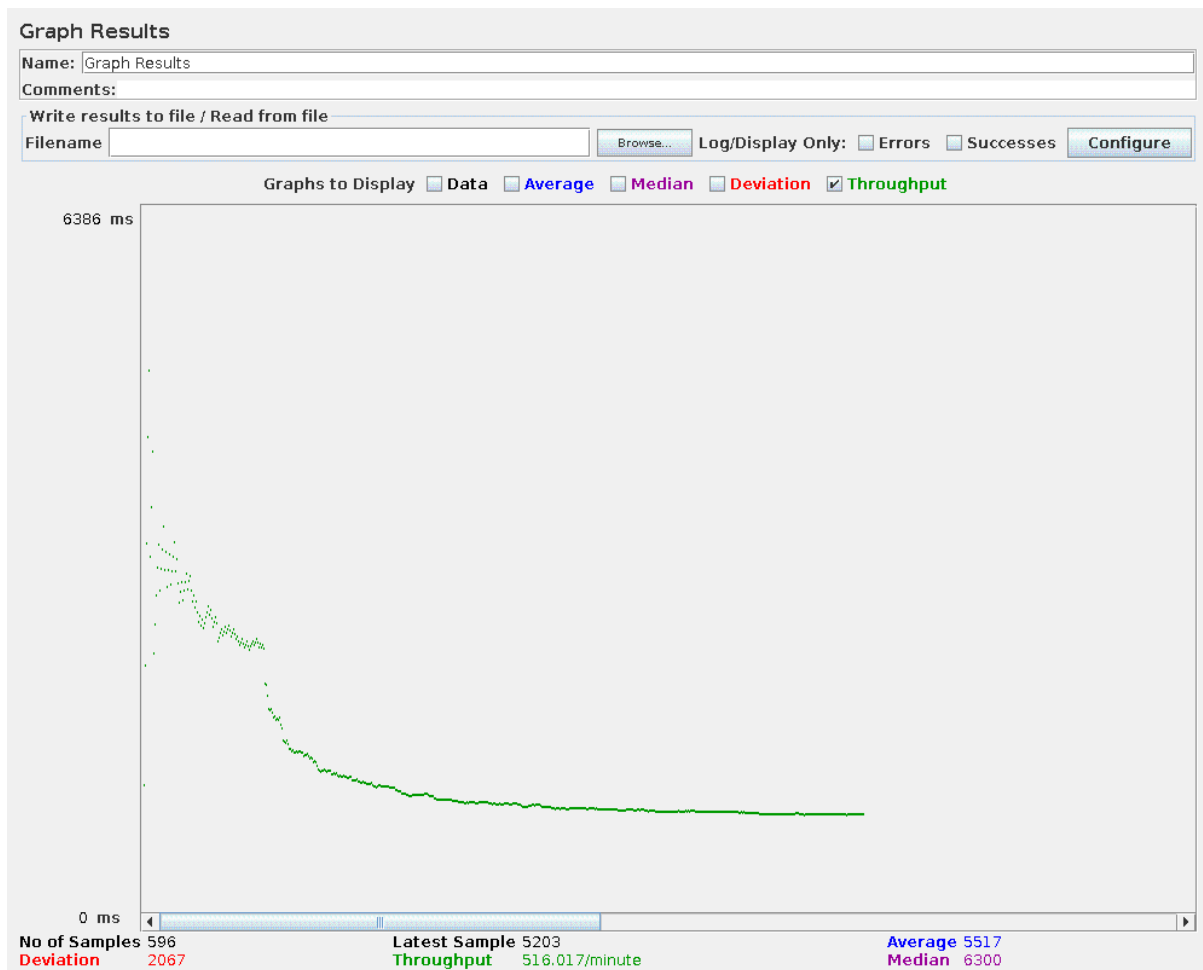


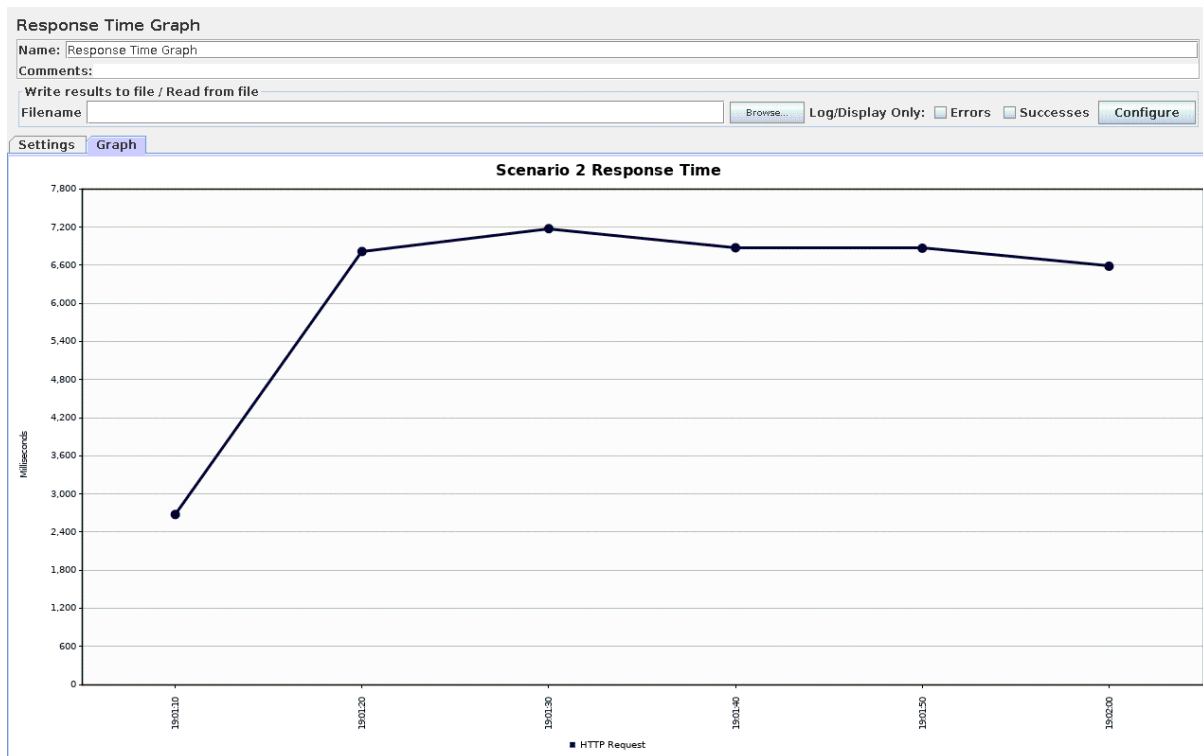


As shown in the graph above throughput stayed consistent through the stress test, only at the start was there much variability in throughput. Response time remained also remained consistent which was expected as we are invoking the same function over and over. Curiously both response time and throughput recorded slightly better results at the start of the stress test, I hypothesis as to why this is in my conclusion.

## Scenario 2 Stress Test

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.



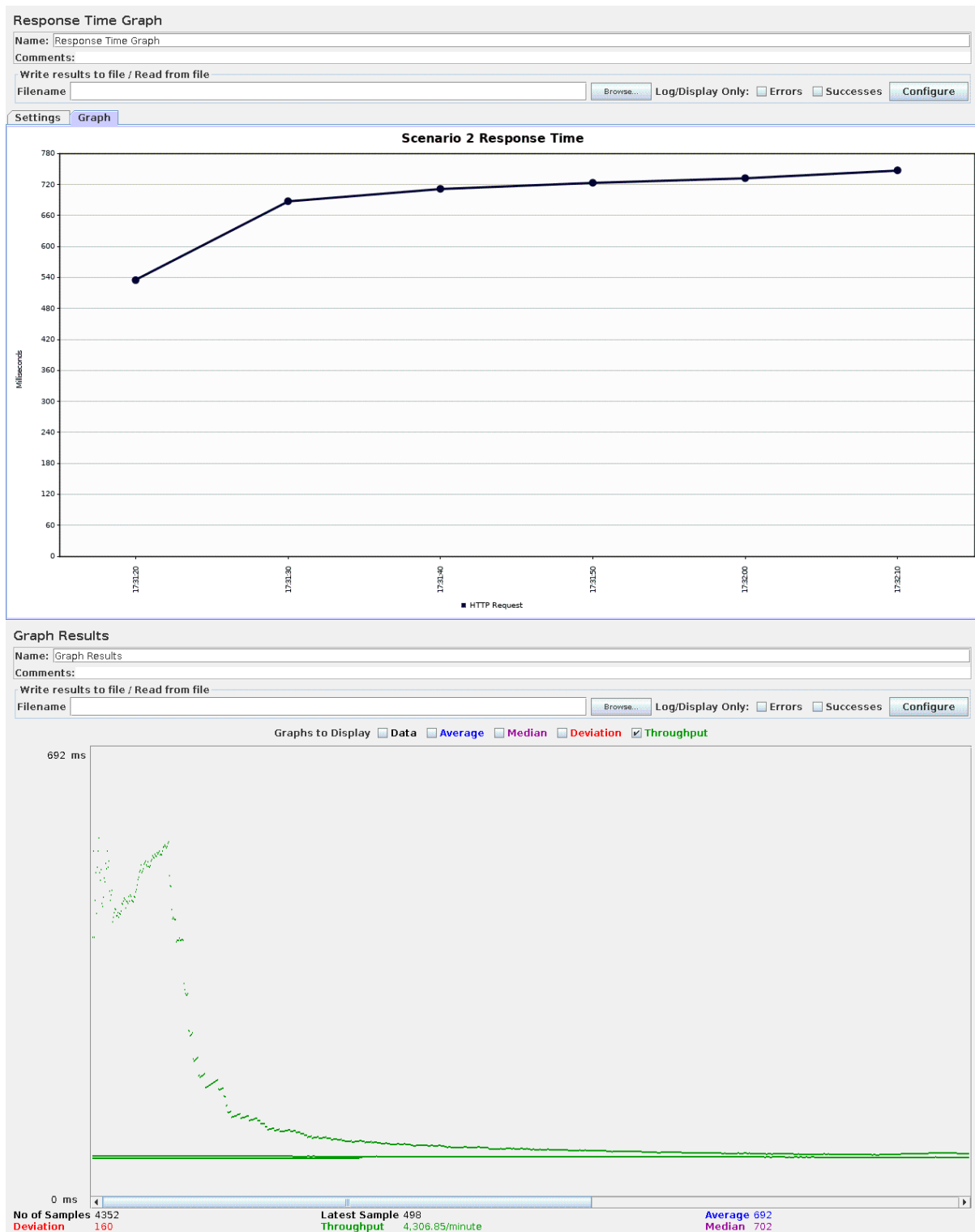


At the start the response time is low similar to what occurred during scenario 1 testing. Once we switch over to the new URI response times increase as the primesStored function uses more processing power to compute a result. This is mirrored in the throughput results; at the start throughput is high 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. The increased amount of time it took to execute the primesStored function led to an increasingly long queue, which meant that the response time jumped significantly as the requests spent longer sitting in the queue waiting to be served.

## 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.

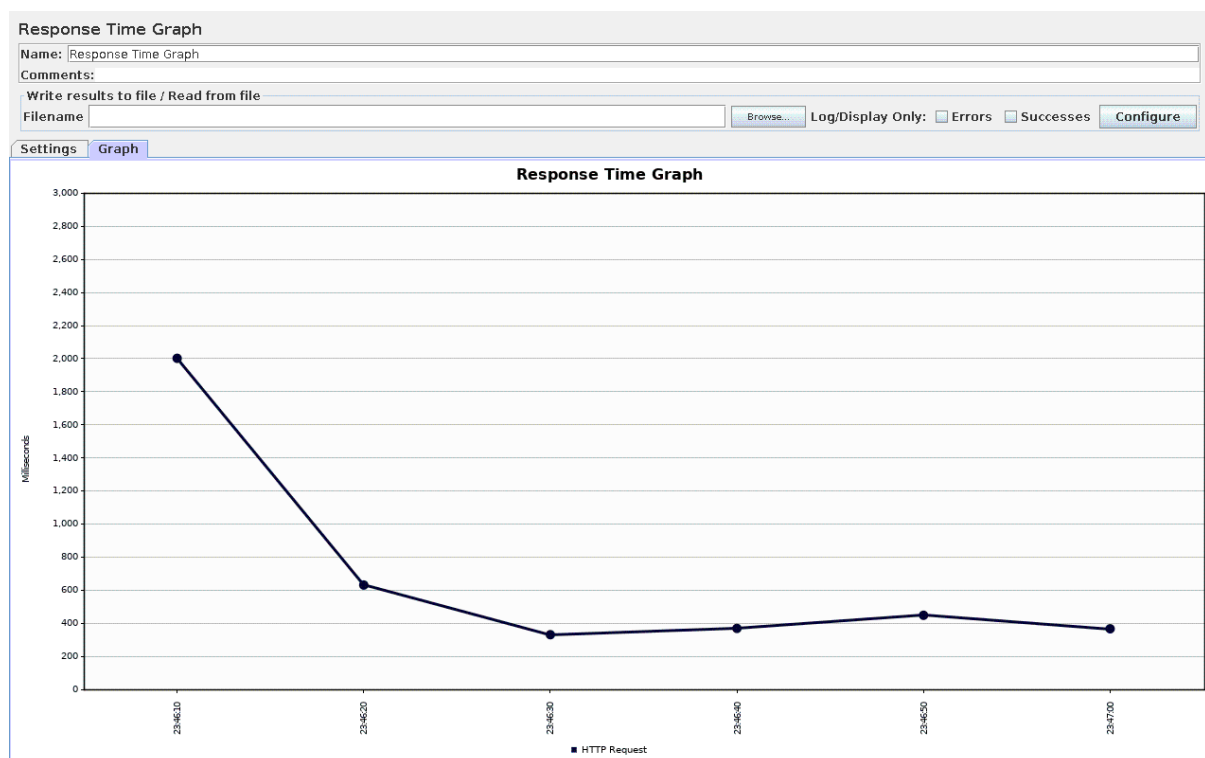
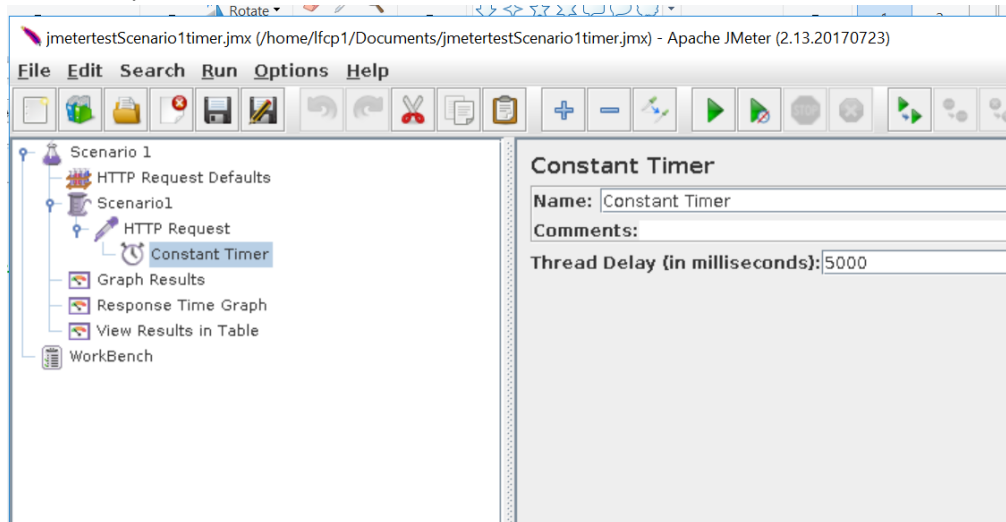
## Unrestricted CPU stress test



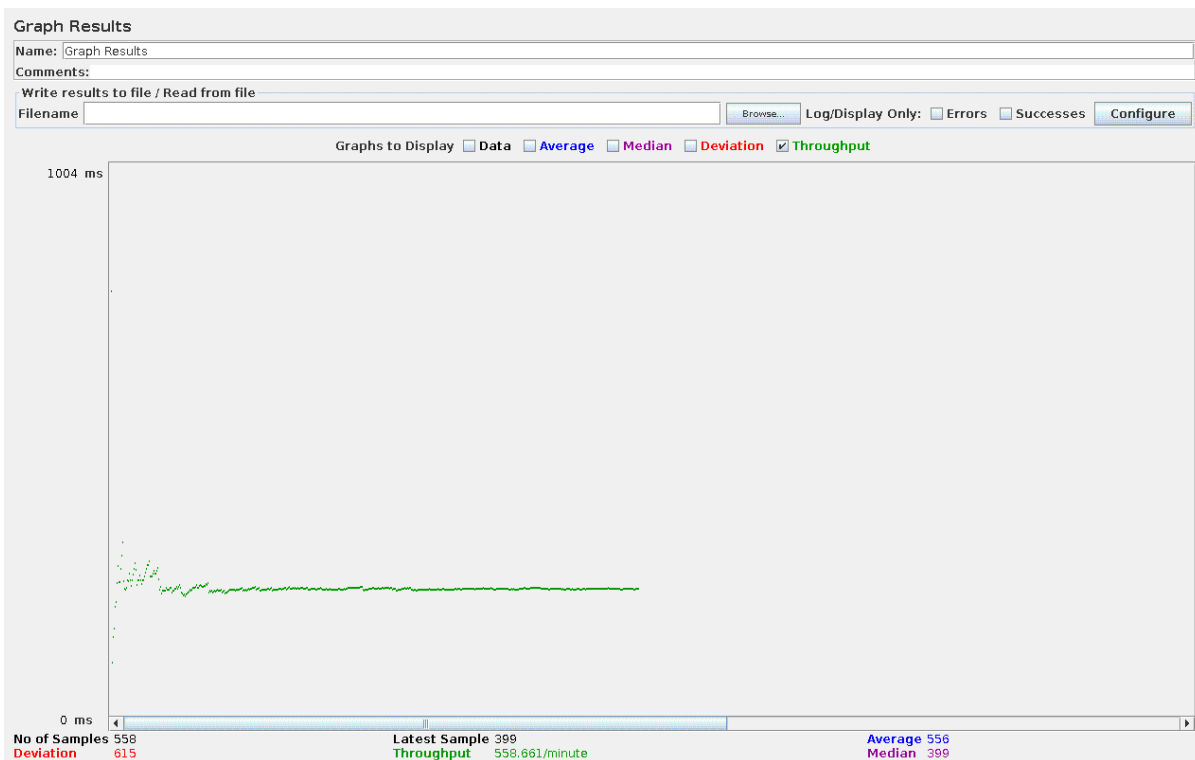
Throughput increased dramatically, almost by a factor of 10 compared to the restricted CPU stress test documented above. Response time also decreased by about a factor of 10. The functions were taking far less time execute with the increased CPU processing available. This roughly factor of 10 change 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. This allows me to easily compare results with the first Stress test documented in this report, as I am now using the same settings and the only difference between the first stress test's test plan and this one is the addition of the timer.







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

By looking at the throughput and response times data from the various stress tests that I have done, a pattern begins to emerge. At the start of the stress test when there isn't a queue, requests are preformed quickly, limited only by the processing power of the CPU. As more threads are enabled and consequently more requests are received the response time increases. These new requests are now sitting in a rapidly growing queue of requests.

Throughput is more variable in the beginning it quickly begins to stabilise as we settle upon the maximum output of the server. Increasing the CPU processing power available increases throughput by being able to respond to clients quicker this in turn leads to a shorter queue. Explained using little law

$$L = \lambda W$$

**L:** Average number of requests in the system

$\lambda$ : Average arrival rate of requests

$W$ : Average time in the system

$W$  is reduced

$\lambda$  Stays the same

So  $L$  (average time in system) is reduced.

Implementing a timer also reduces  $L$  but this time instead of  $W$  being reduced the average arrival rate of requests ( $\lambda$ ) is reduced while  $W$  stays the same.

Since our server is being hosted by the same machine that our JMeter http requests are coming from, latency is very low. This means that when we increase the processing power response times show as much of an increase as throughput. If for example we were performing these tests on a server that is connected to us through the internet rather than being on the same machine I would expect the relationship between response times and CPU processing power to be less as more of the response time would be devoted to the routing of the request to the server.