COMPX341 Assignment 4 Report

Contents

Github URL
Link to the public Github repository where code has been committed throughout the development process:
https://github.com/trebesh/COMPX341_Assignment4

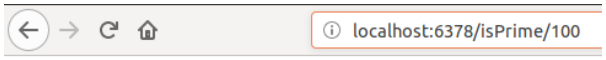
Software Specifications used

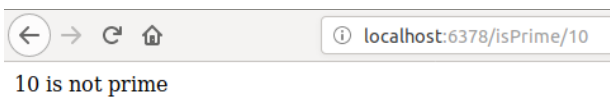
Python 3.6.7 (https://www.python.org/downloads/release/python-367/)
Redis, redis-py (https://redis-py.readthedocs.io/en/latest/)
Apache JMeter 2.13.21070723 (https://jmeter.apache.org/)

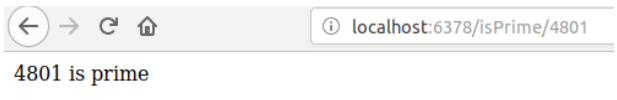
Manual Testing

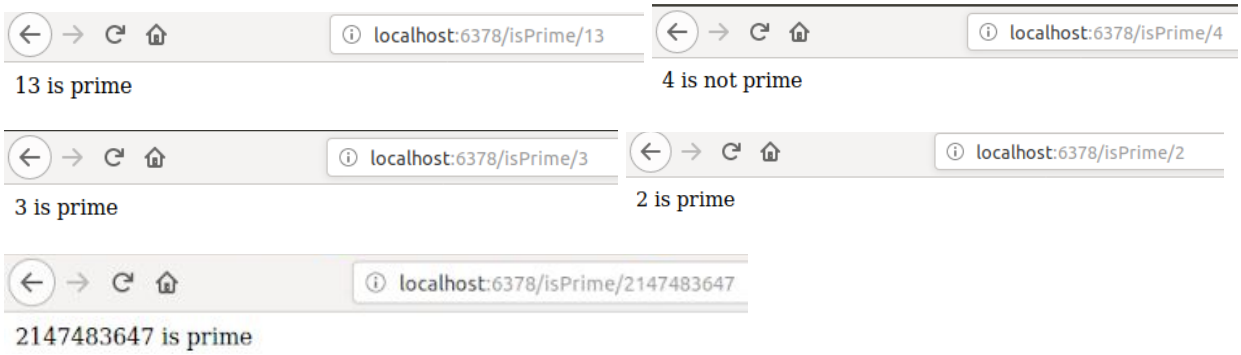
Testing the base page with no further URL specification after the port. Code was altered from the end-of-tutorial state to only display “Hello World!” and not the number of times the page had been visited.

Fulfilling REQ-1:
The page should display “<number> is prime” if a valid prime number is entered and “<number> is not prime” if a valid non-prime number is entered. Valid prime input include all prime numbers greater than 1 and less than the maximum capacity of an int type variable (theoretically unlimited by Python 3). Valid non-prime input include all whole integers positive and negative.

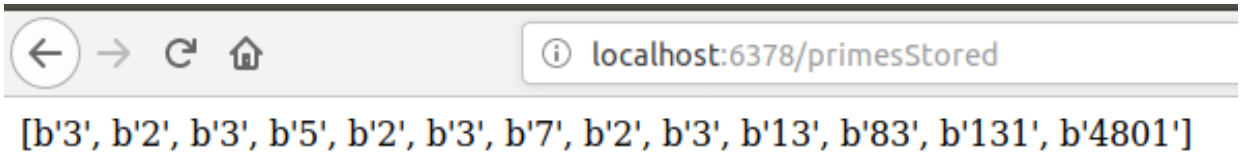




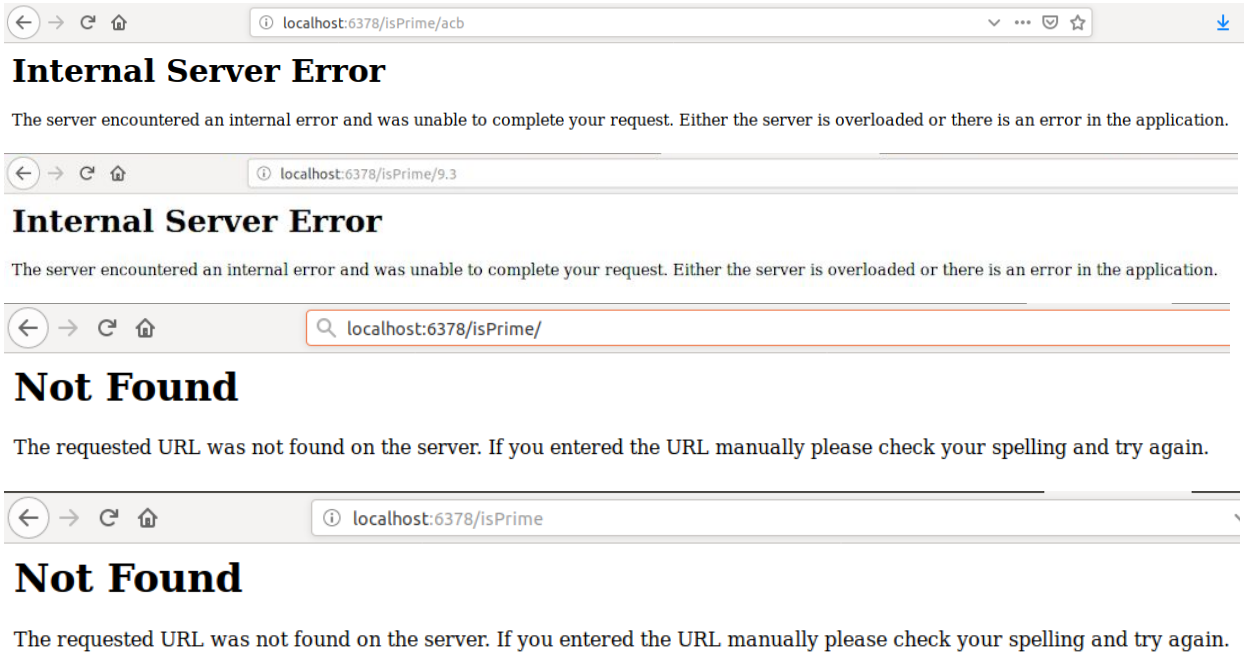




To test that the recognised prime numbers were being added to the list, primesStored was initially programmed to return all values in the redis list, including duplicates. After the above requests (and some other tested inputs) the expected list was displayed as follows:



For invalid inputs the page should return an error due to trying to perform math/integer operations on a value type other than integer. Invalid inputs includes unicode characters other than numbers, and decimal input. These cases should produce an Internal Server Error due to the python program attempting to perform integer operations on non-integer type input. In the cases of invalid input in the form of blank/null input or an incorrect/incomplete address a URL Not Found error should be generated and displayed.



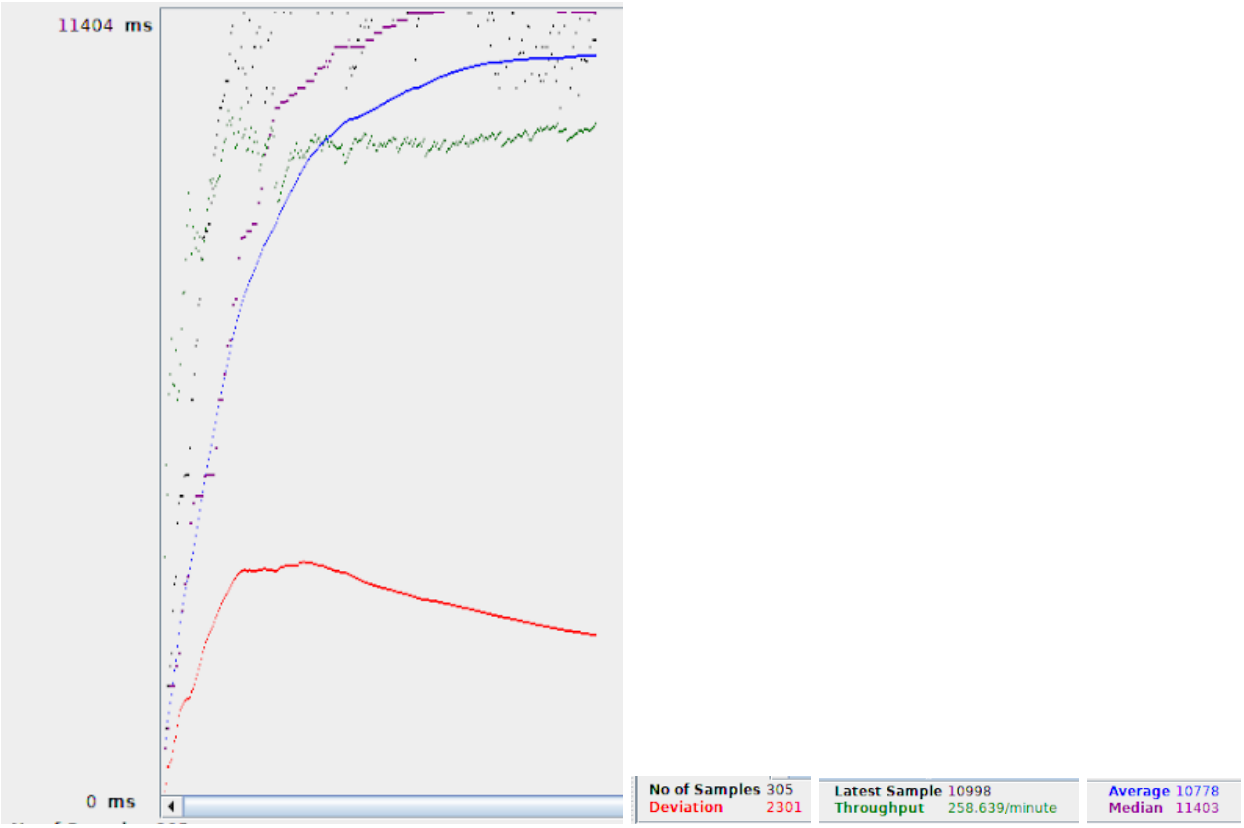
Fulfilling REQ-2:
If the primesStored address/request is correct then the app should return the list of the prime numbers stored in the redis database. Duplicate values are ignored after the first instance, hence the list is ordered first-in-first-out. After the above entries the new primes 17 and 23 (respectively) were added to the system to produce the following output from primesStored:

In order to speed up the primesStored response time, I added functionality to clear the list stored in redis and start it again (thus enabling lists to be shorter and so retrieved/processed faster) by following the path /clearPrimesStored.

Automated Stress Testing (using Apache JMeter)

Scenarios 1 and 2 were conducted with CPU limits of 0.1 (as in the initial tutorial) and both Ramp-up-period and start-up delay equal to 0. This enabled the two scenarios to be used to test the functionality of the different pages of the app in a way that could be use for comparison.

Scenario 1
Repeatedly invoking the apps' "isPrime/214783647" URL for 60 seconds using 50 threads produced the results graph below. Throughput of 258.239requests/minute indicates the app is running at a speed (even under this stress) that a human would not be likely to find too much complaint with. However this is not particularly fast in computational terms and this performance could be increased by allowing the app more CPU to use (as demonstrated in *Altered CPU Limits*).



Scenario 2

Using 50 threads to invoke isPrime URL for all integer values between 1 and 100, then repeatedly invoking primesStored for 60 seconds. To allow optimal processing/response time for retrieving the list this scenario was run immediately after performing a clear on the list. The results from testing this scenario are broken into two parts. Part one refers to the isPrime operation being carried out for all integers 1-100. Part two refers to the 60 seconds of requesting primesStored.

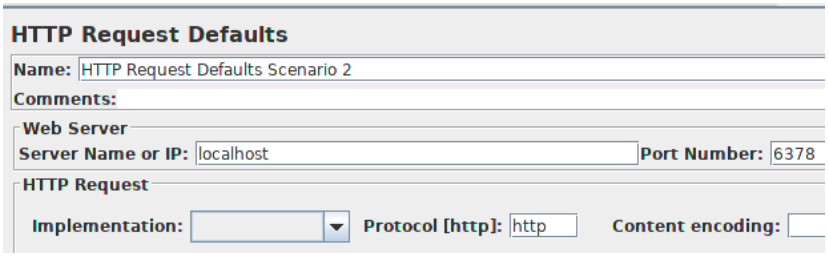
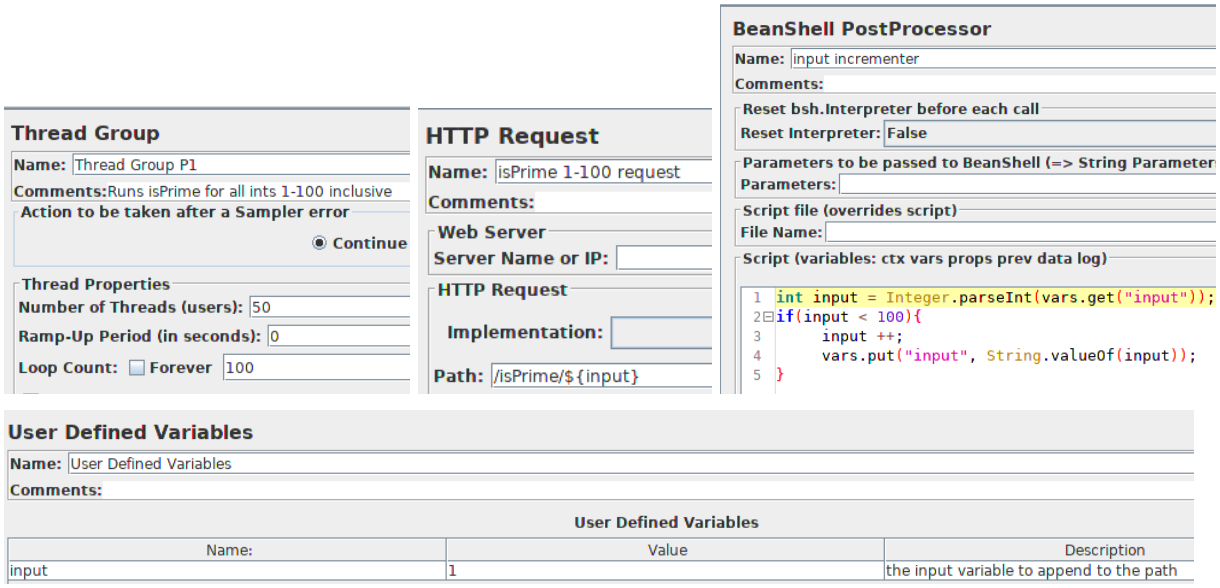


Figure: JMeter HTTP Request Defaults for Scenario 2

Part 1:

The pertinent JMeter settings used to pouce the test for part 1 are shown in the images below.



Figures above: JMeter settings for scenario 2 part 1

These settings produced the results graphed below. A throughput of 2163.035 requests/minute was measured with a mean average of 1377.

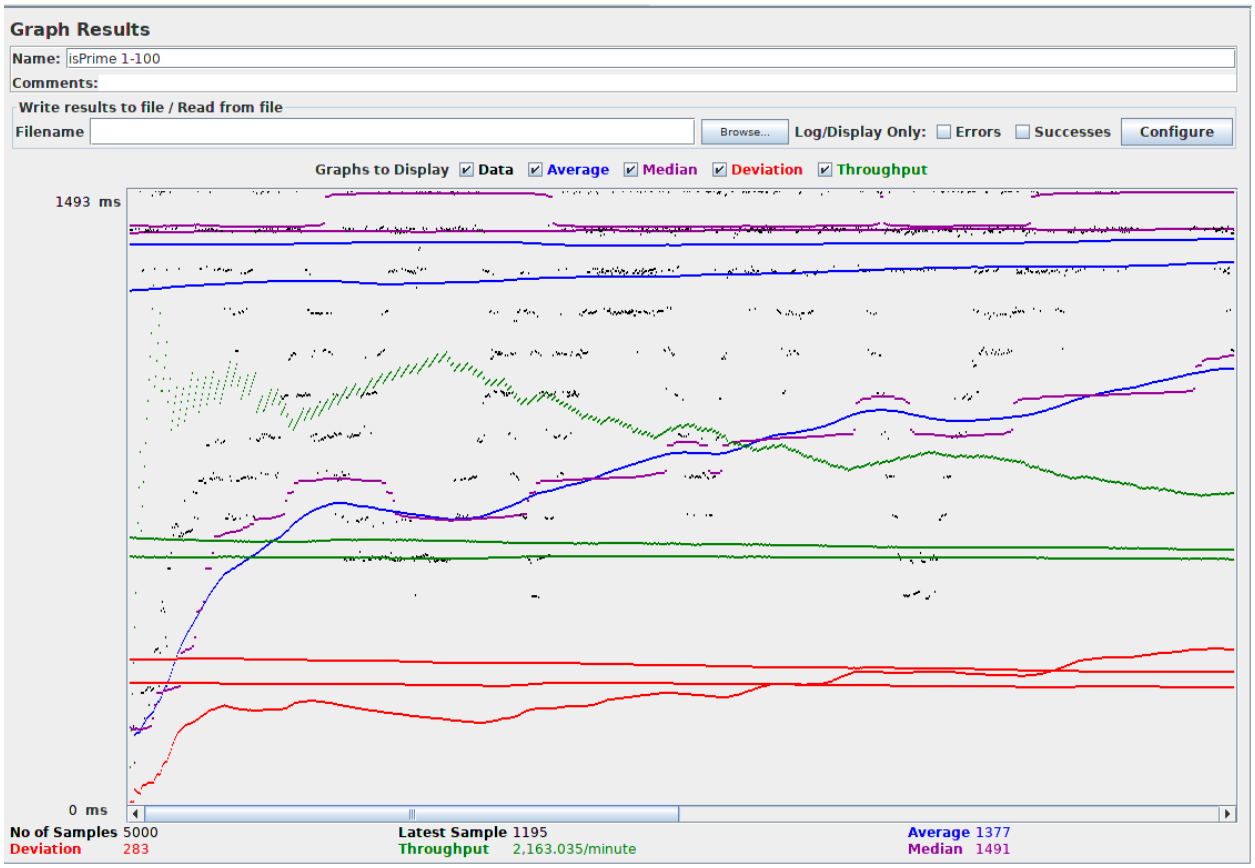
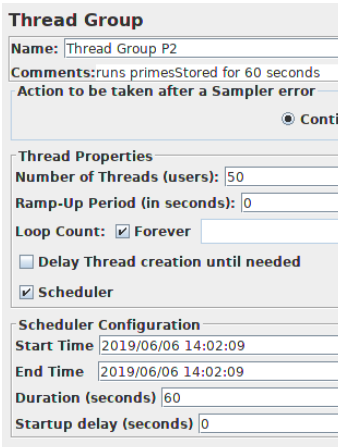
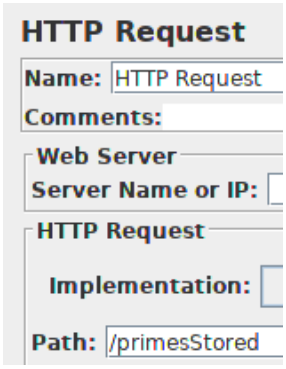


Figure: Results Graph for Scenario 2 part 1

Part 2:

The setting use to test the primesStored functionality are shown below. This test forces the program to sort through the redis list of primes (all 25 primes from 1-100) and discard duplicates (though thanks to running clearPrimesStored first this shouldn't occur and the page should function in optimum condition i.e. no duplicates in the list to ignore) before returning the list. However, as the results graph shows, the comparisons required to process the list for duplicates take up quite a bit of time and produce a throughput of only 7.813 requests/minute. This shows the process for retrieving the list of stored primes is quite slow so this could be an area to optimise if further development of this app were to occur.





Figures Left: JMeter HTTP settings for Scenario 2 part 2

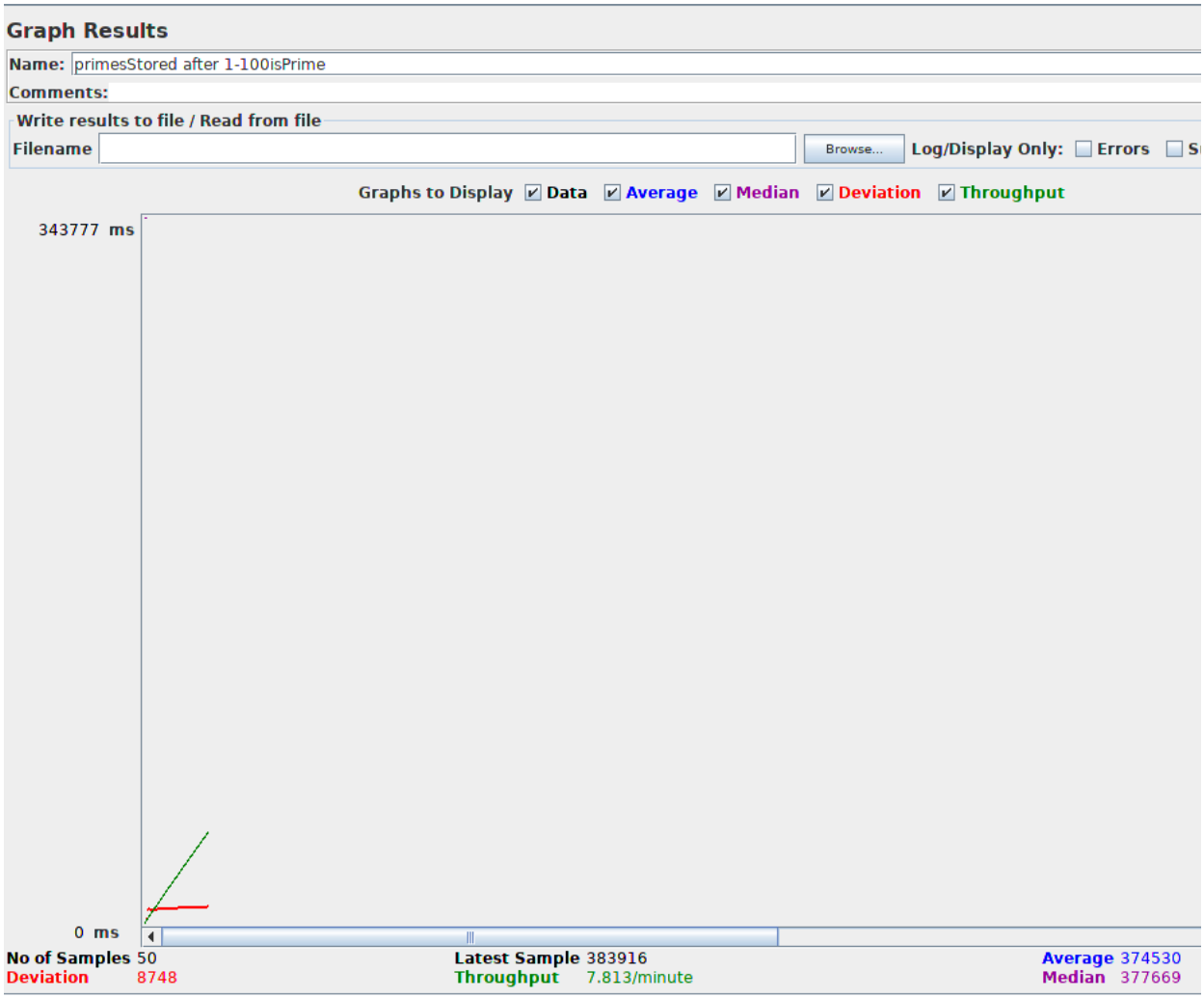


Figure above: Results graph for Scenario 2 part 2

Altered CPU limits

The amount of CPU available to the program is a main determinant of how quickly an app can run. Therefore altering this availability should definitively impact the performance (ou interest is in throughput and response time) of the app with more available CPU resulting in a faster program. This was tested with the app using *scenario 2* (to cover both isPrime and primesStored) and CPU limits of 0.05, 0.5 and 0.9 (altered in the docker-compose file). The results of these tests are shown below.

CPU = 0.05
For the CPU limit 0.05 the throughput of *part 1* was 892.464 requests/minute, a decrease by more than half for a halved CPU limit (implying the relationship between CPU limit and throughput is non-linear). Response times ranged from less than 2000ms for the first request to slightly over 4000ms, which, averaging out at ~3 seconds per response is rather slow for a web

app. It was decided that as this CPU limit negatively impacted the response times so much, *part 2* would not be tested - suffice to say that throughput would have been less than 7.813 requests per minute.

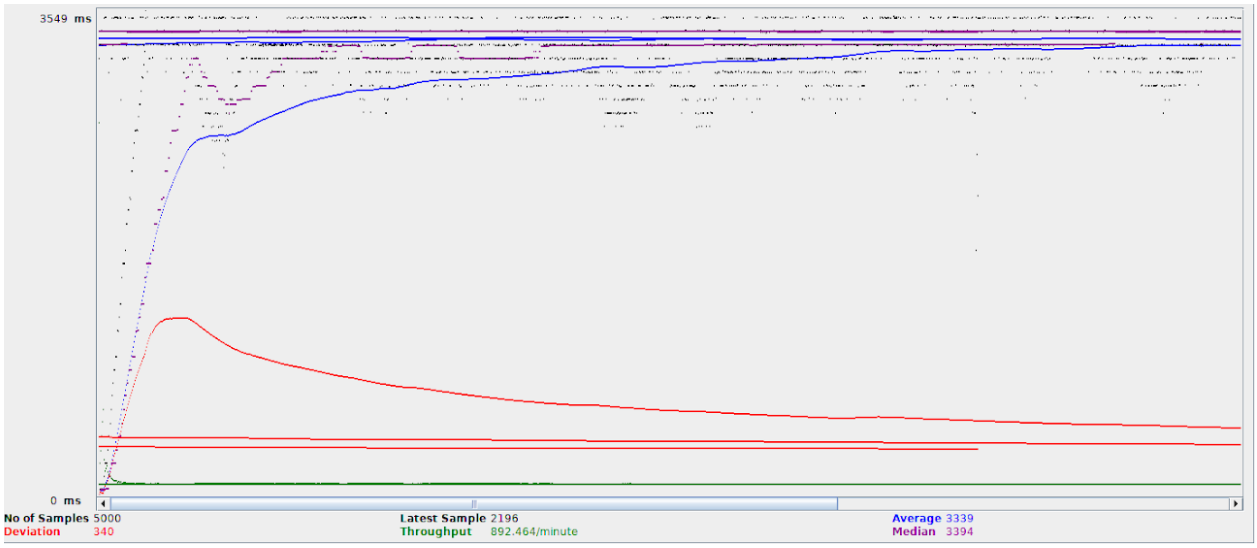


Figure above: Results Graph for CPU = 0.05 Scenario 2 part 1

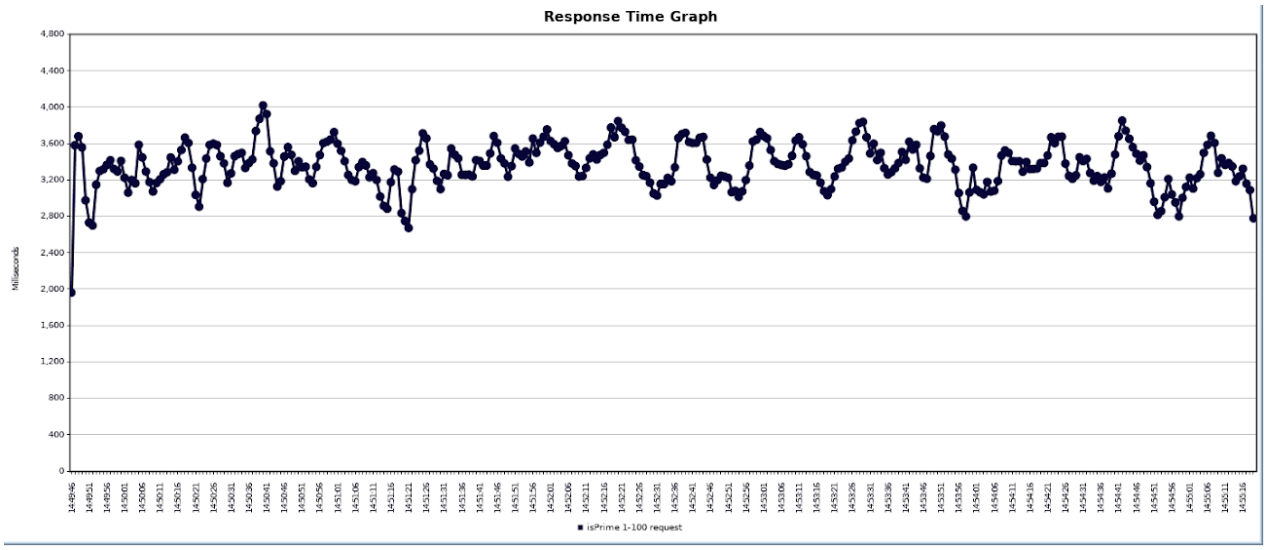


Figure above: Response Times Graph for CPU = 0.05 Scenario 2 part 1

CPU = 0.5

With a CPU limit of 0.5 (an 5* increase from the initial 0.1) the *part 1* throughput increased almost 6* to 12.787 requests/minute, and the mean average showed a drastic decrease to 228. The response times showed more variation, ranging between less than 100ms and slightly over 4000ms with the majority or response times below 2000ms. For *part 2* the throughput increased almost 5* (compared to a CPU limit of 0.1) to 38.087. There wasn't enough data to generate a response time graph for *part 2* of this CPU limit but it would be expected that the majority (and average) of response times would be greater than that for the CPU limit 0.1.

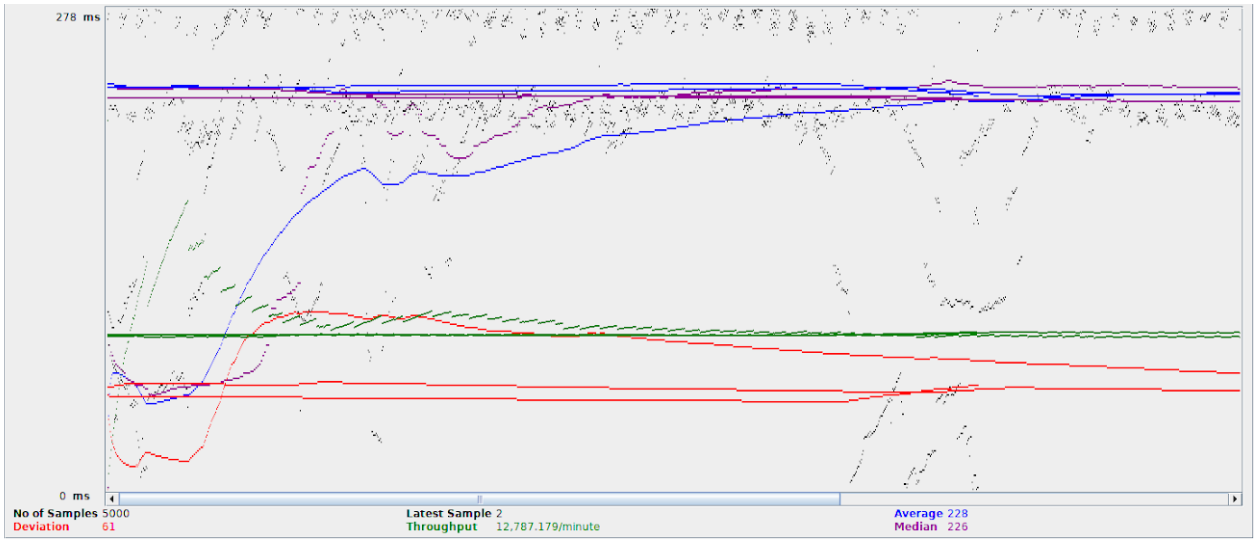


Figure above: Results Graph for CPU = 0.5 Scenario 2 part 1

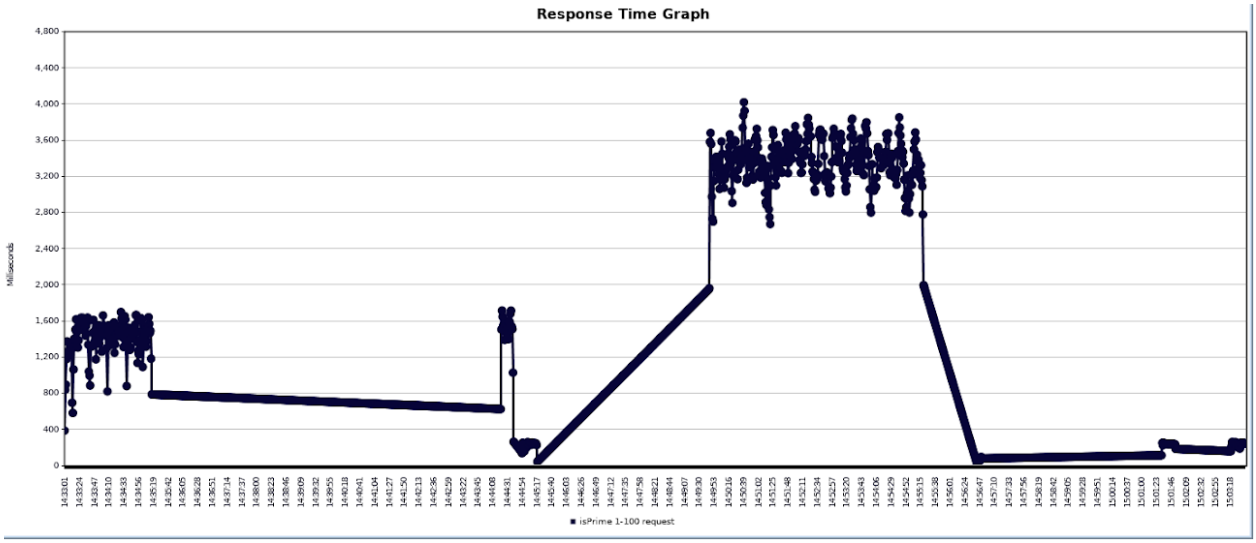


Figure above: Response Times Graph for CPU = 0.5 Scenario 2 part 1

Figure above: Results Graph for CPU = 0.5 Scenario 2 part 2

CPU = 0.9

The CPU limit of 0.9 produced a throughput of 39,328.789 (over 16* greater than the CPU 0.1 base case) for *part 1* with an average of just 71. Response times were similar to those of the 0.5 CPU limit, ranging from <100 to just over 4000 as can be observe on the graphs. This could indicate that the CPU is no longer the limiting factor on response times, however the drastic increase in throughput is counterintuitive to this. For *part 2* a throughput of 51.079 responses/minute was achieved over the minute long testing window. This produced sufficient data to generate a response time graph for *part 2* which shows largely constant response times of approx. 58,000ms with very little variation. For the primesStored function to be taking 58 seconds (not much quicker than an observed response time for CPU = 0.1) to produce a response running with a CPU limit of 0.9 indicates that the limiting factor here is the function itself and hence optimization options should be explored.

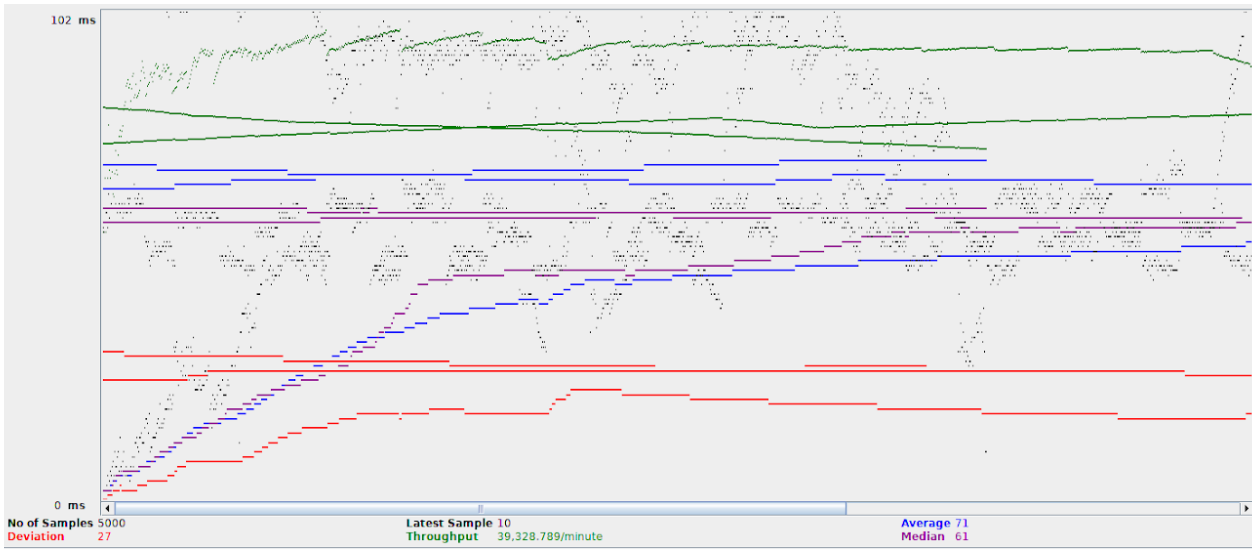


Figure above: Results Graph for CPU = 0.9 Scenario 2 part 1

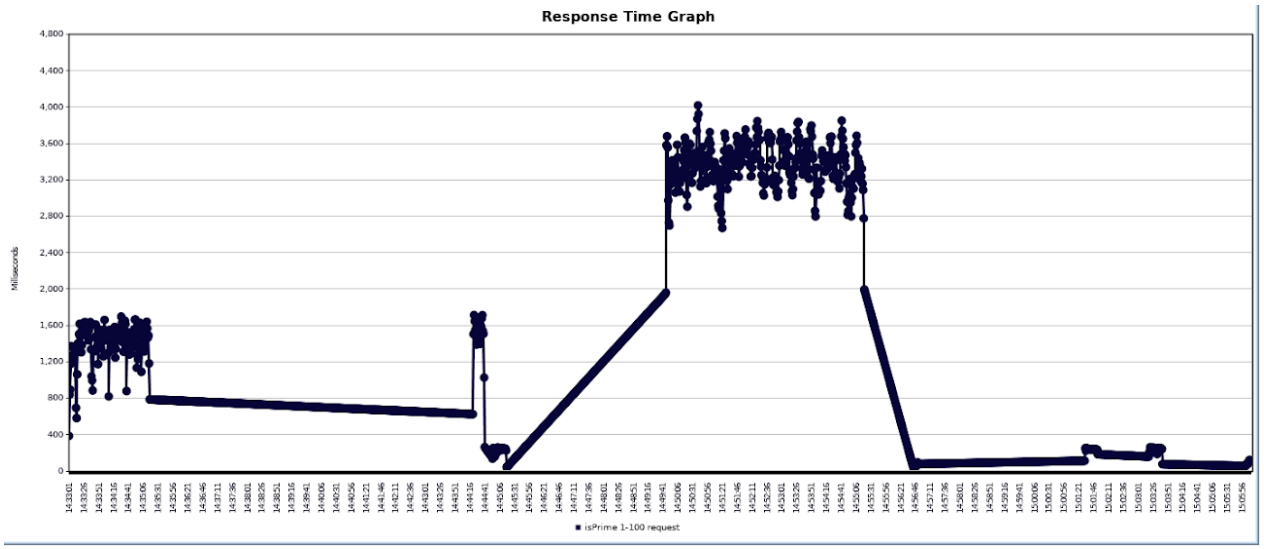


Figure above: Response Time Graph for CPU = 0.9 Scenario 2 part 1

Figure above: Results Graph for CPU = 0.9 Scenario 2 part 2

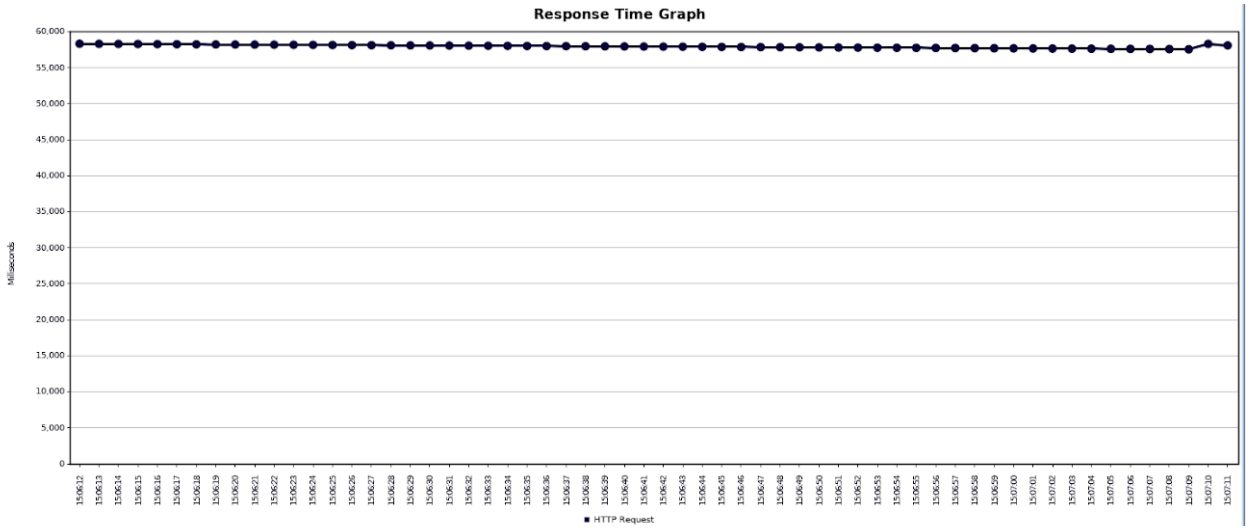


Figure above: Response Time Graph for CPU = 0.9 Scenario 2 part 2

Altered timer delays

The tests discussed below were performed with a CPU limit of 0.9 to maximise data collected. The initial ramp-up time used in all above test cases was 0s (i.e. start all threads immediately) The following tests alter this Ramp-up period to examine the effect of different request spacing (temporally) on throughput and response times. Once again the *scenario 2* test process was used.

Ramp-up = 5s

A ramp-up period of 5 seconds decreased the throughput of *part 1* by ~10000requests/minute to 29603.316requests/minute and decreased the range of response times by over 1000 to range between 0 and just shy of 3600ms. The same negative effect on throughput was observed on *part 2* with a decrease to 47.267 requests/minute. The ramp-up change had an interesting effect on the response times of *part 2* with responses to late requests (sequentially) being faster than to earlier requests.

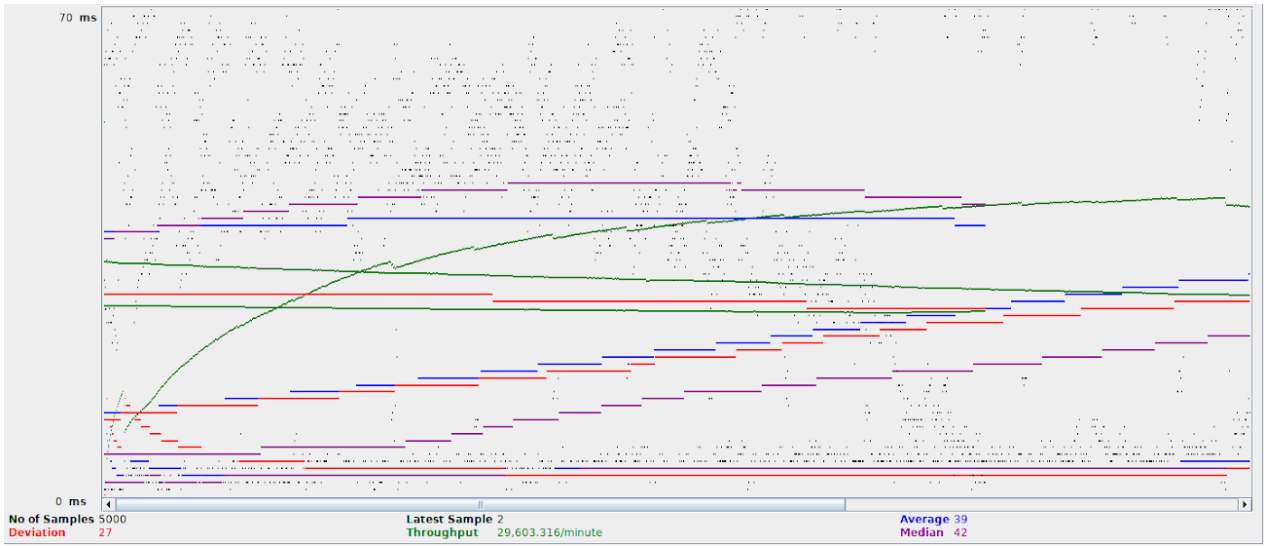


Figure above: Results Graph for CPU = 0.9 Scenario 2 part 1 Ramp-up = 5s

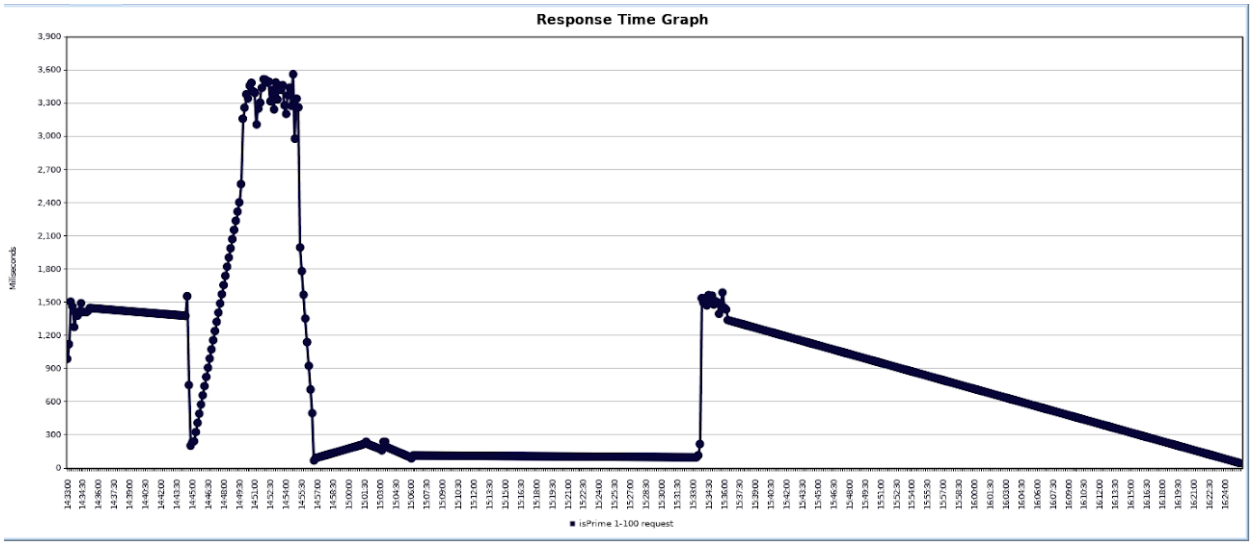


Figure above: Response Times Graph for CPU = 0.9 Scenario 2 part 1 Ramp-up = 5s

Figure above: Results Graph for CPU = 0.9 Scenario 2 part 2 Ramp-up = 5s

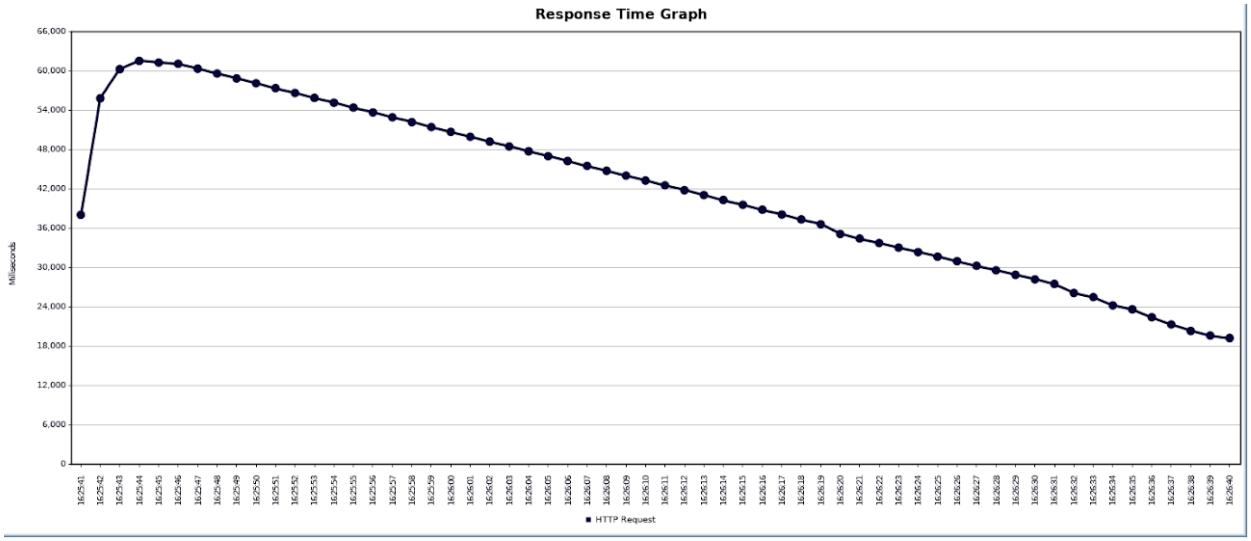


Figure above: Response Time Graph for CPU = 0.9 Scenario 2 part 2 Ramp-up = 5s

Ramp-up = 10s

The ramp-up period of 10 seconds decreased the throughput of *part 1* by a further ~4000requests/minute to 25764.342requests/minute, however there was little to no difference in the response times. Interestingly, for *part 2* the increased Ramp-up time led to an increase in throughput (over the 5s ramp-up, still below the ramp-up of 0s). The ramp-up change had an drastic effect on the response time graph, with times ranging from 0 to ~375000ms.

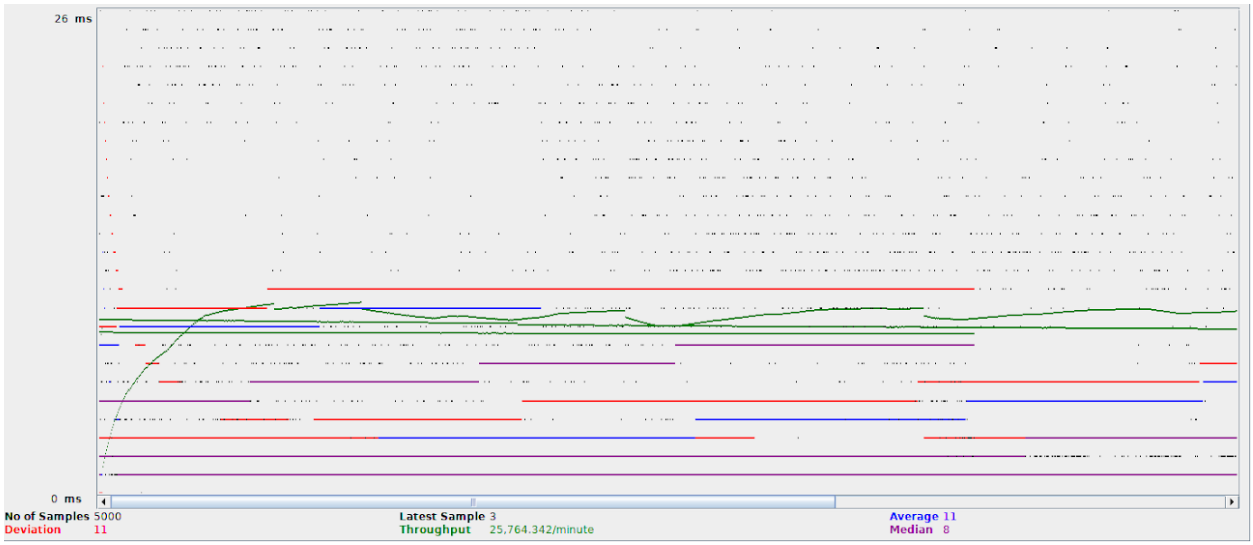


Figure above: Results Graph for CPU = 0.9 Scenario 2 part 1 Ramp-up = 10s

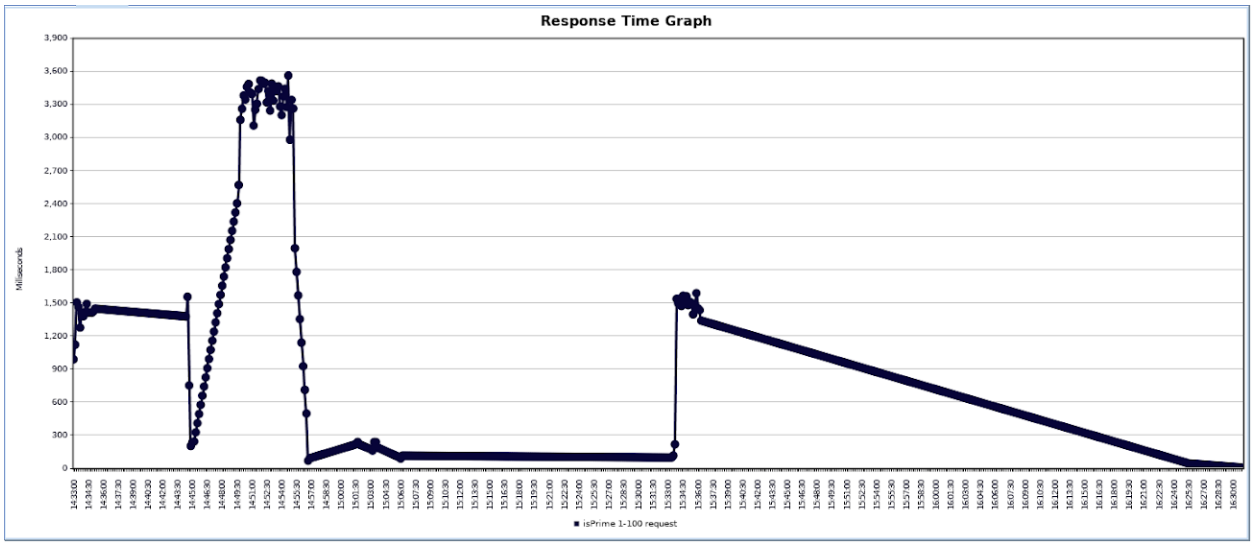


Figure above: Response time Graph for CPU = 0.9 Scenario 2 part 1 Ramp-up = 10s

Figure above: Results Graph for CPU = 0.9 Scenario 2 part 2 Ramp-up = 10s

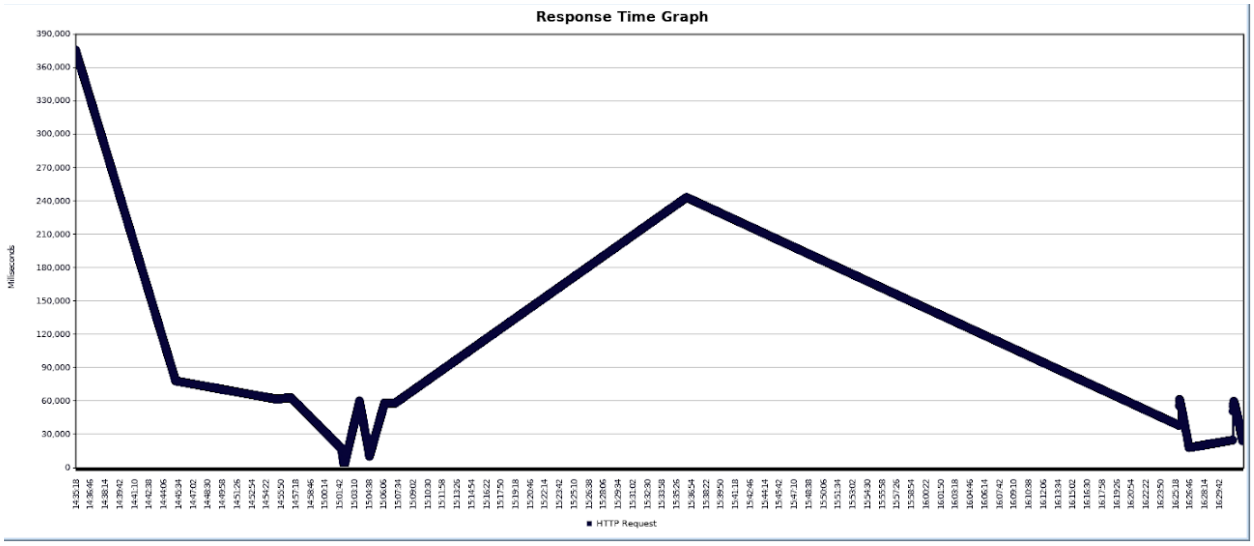


Figure above: Response Time Graph for CPU = 0.9 Scenario 2 part 2 Ramp-up = 10s

Ramp-up = 60s

Increasing the Ramp-up time to a full minute resulted in a *part 1* throughput of 5064.402requests/minute with a mean average of just 2. The response times showed an increase in range over the other ramp-up tests, returning to the range shown by the CPU = 0.9 and ramp-up = 0 of between 0 and just over 4000ms. Throughput for *part 2* dropped lower than any other CPU = 0.9 test at 46.937requests/minute and generated a roughly parabolic response time curve with values ranging between 0 and just over 50,000.

Figure above: Results Graph for CPU = 0.9 Scenario 2 part 1 Ramp-up = 60s

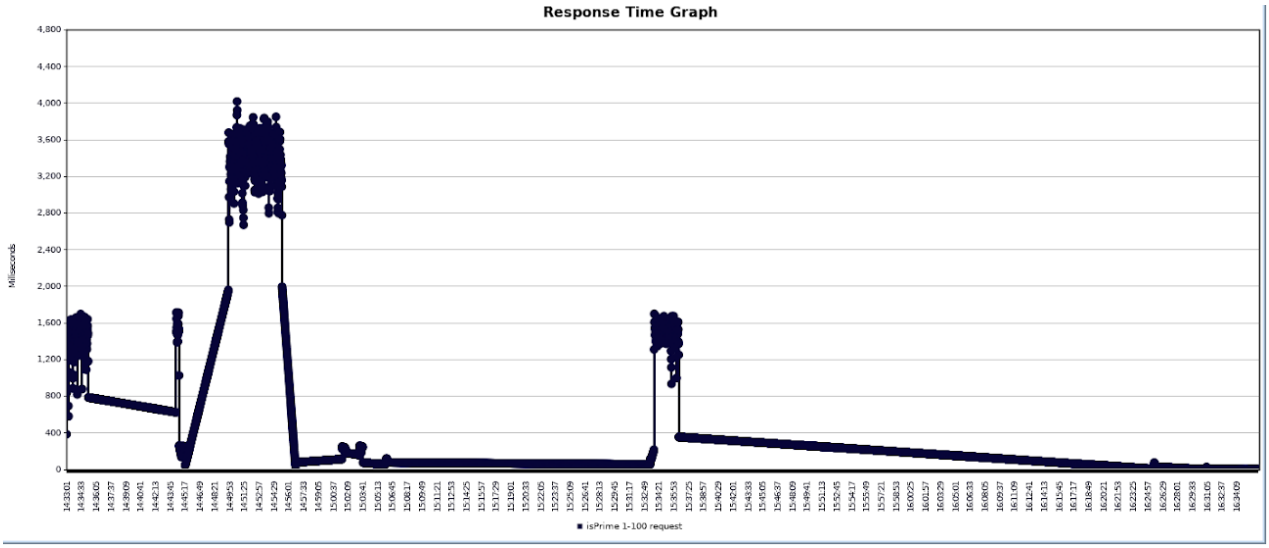


Figure above: Response Time Graph for CPU = 0.9 Scenario 2 part 1 Ramp-up = 60s

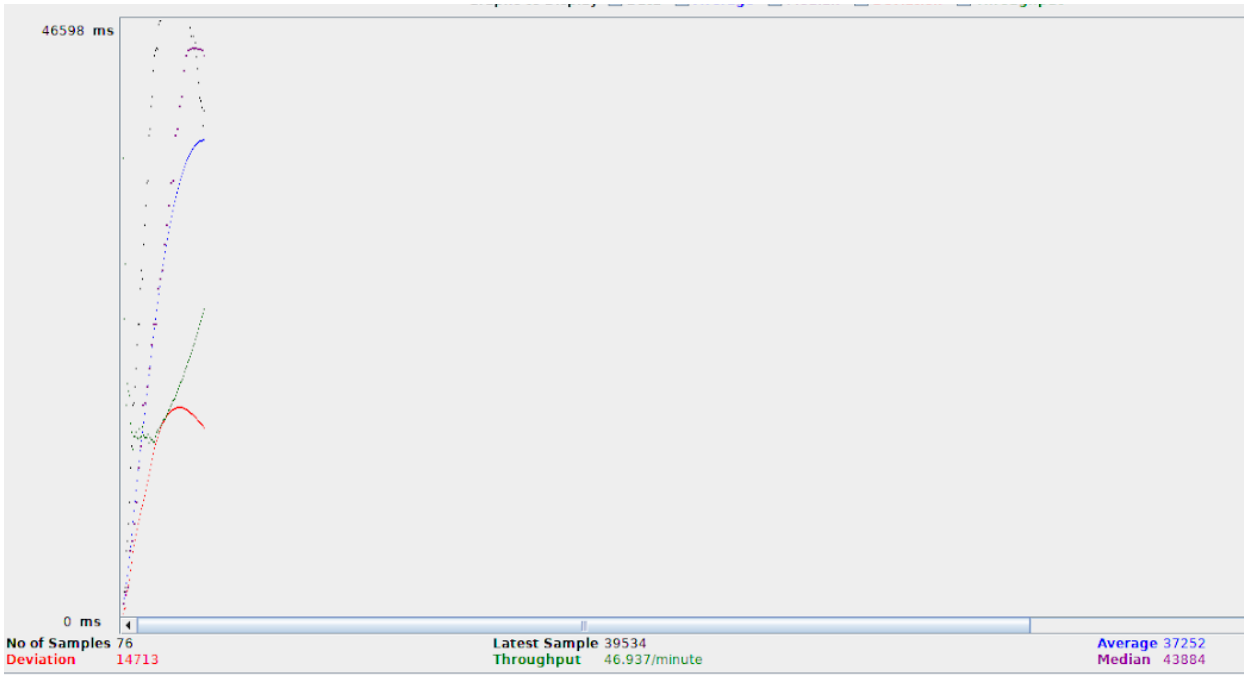


Figure above: Results Graph for CPU = 0.9 Scenario 2 part 2 Ramp-up = 60s

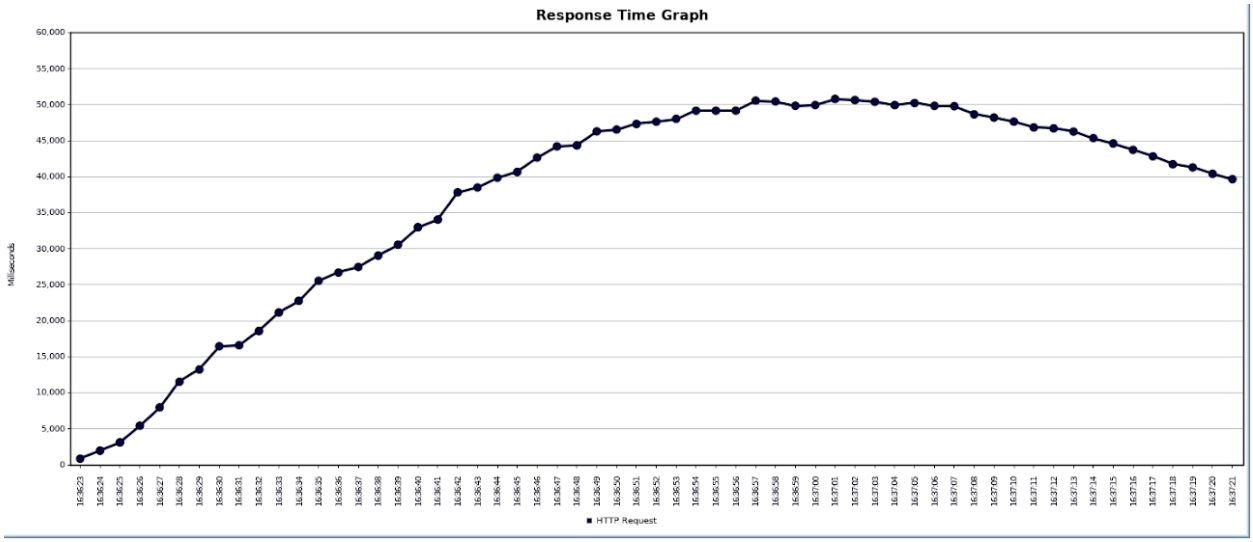


Figure above: Response Time Graph for CPU = 0.9 Scenario 2 part 2 Ramp-up = 60s