Testing Report

COMPX341-A5 2020

Cheer, Alex 1344618

Contents

Testing Specifications

The system configuration the testing was performed on was as follows:

- AMD Ryzen 5 2600 processor @ 3.4Ghz
- 16GB RAM
- 256GB SSD
- Windows 10 Professional version 1903 x64
- Python 3.8.1
- Docker Engine v19.03.8
- Redis v6.0.5
- Flask
- JMeter 5.3
- AdoptOpenJDK 11.0.4+11

Test Results

The first round of tests had no CPU or memory restriction applied to the Docker container, and a 300ms thread delay in JMeter, showing an unrealistic but ideal environment.

The first scenario had a throughput of 114.9 requests per second, sending 6871 requests over a 60 second period. The spread of the response times was wide, with an average response time of 135ms, a median of 130ms and a peak at 410ms. This could be down to the fact that 2147483647 is also known as the largest 32bit integer, so as far as the application is concerned it's the largest value it can handle.

The 1-100 scenario helped back up this theory, as the average throughput for that test was 164 requests per second, with 9792 requests being sent over another 60 second period. The response time values back up this theory as well, with an average response time of 5ms, a median of 4ms, and a peak at 117ms, which is lower than the average to work out the previous scenario.

The final scenario, stress testing the primesStored value showed similar data to the 1-100 scenario. The throughput was 162.6 requests per second, with a total of 9709 requests sent over a 60second period. The response times barely fluctuated this time around, with an average of 7ms, median of 6ms and a peak of 178ms.

The second round of tests had a CPU limit of 0.1 and 128MB of RAM, with the same 300ms thread delay.

The first scenario had a throughput of 12.1 requests per second, sending 767 requests over a 60 second period. The spread of the response times was wide, with an average response time of 3736ms, a median of 3801ms and a peak at 4591ms. This shows the bottleneck the limits have created, which according to the Docker statistics was mainly a CPU bottleneck.

The 1-100 scenario helped back up this theory, as the average throughput for that test was 47.6 requests per second, with 2887 requests being sent over another 60 second period. The response time values back up this theory as well, with an average response time of 745ms, a median of 704ms, and a peak at 1293ms, which is still lower than the average to work out the previous scenario.

The final scenario, stress testing the primesStored value showed similar data to the 1-100 scenario. The throughput was 38.6 requests per second, with a total of 2348 requests sent over a 60second period. The response times were all over the place this time, ranging from 13ms to 1394ms at the peak. The average was 985ms, with a median of 996ms.

For the final round of tests with response time and throughput, I removed the CPU limit but left the RAM limit intact, to see whether it was a CPU bottleneck or a RAM bottleneck causing poor performance. I left the thread delay at 300ms again.

Something I noted during this round of tests, was that while calculating if the 2147483647 is prime or not, Docker reported it was using 100% of the VCPU, while the other tasks only use between 32-40% of the VCPU.

The first scenario had a throughput of 111.8 requests per second, sending 6690 requests over a 60 second period. The spread of the response times was wide, with an average response time of 147ms, a median of 133ms and a peak at 418ms.

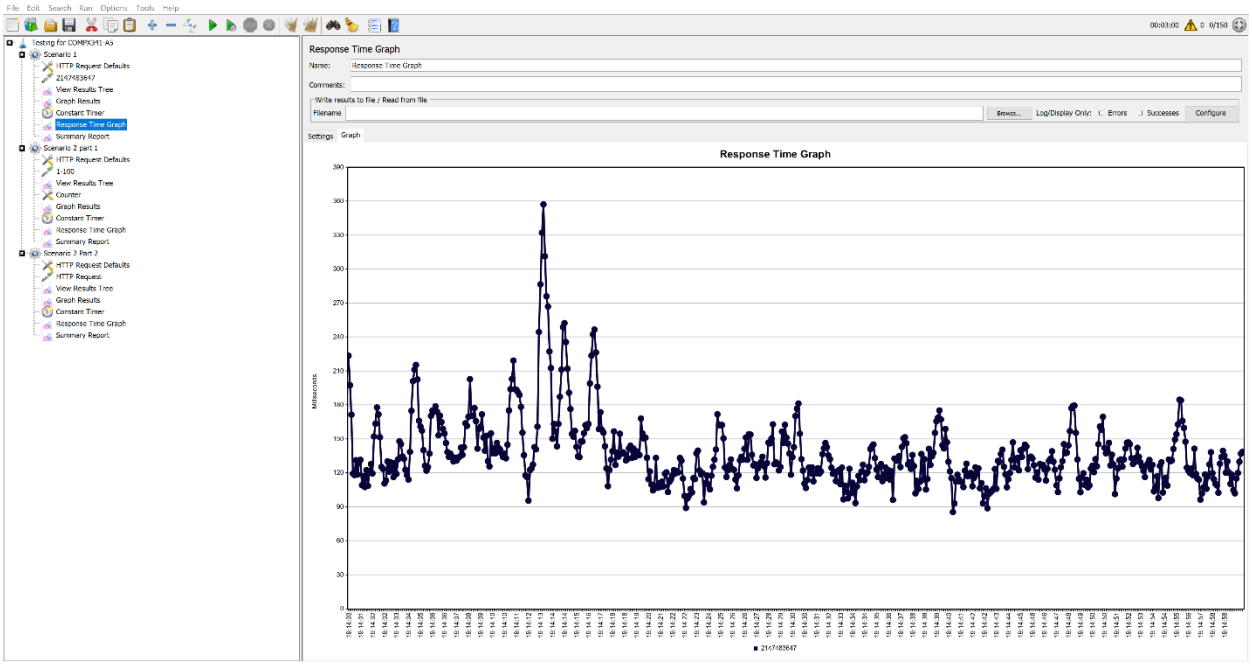
The 1-100 scenario helped back up this theory, as the average throughput for that test was 163.7 requests per second, with 9772 requests being sent over another 60 second period. The response time values back up this theory as well, with an

average response time of 6ms, a median of 5ms, and a peak at 111ms, which is lower than the average to work out the previous scenario.

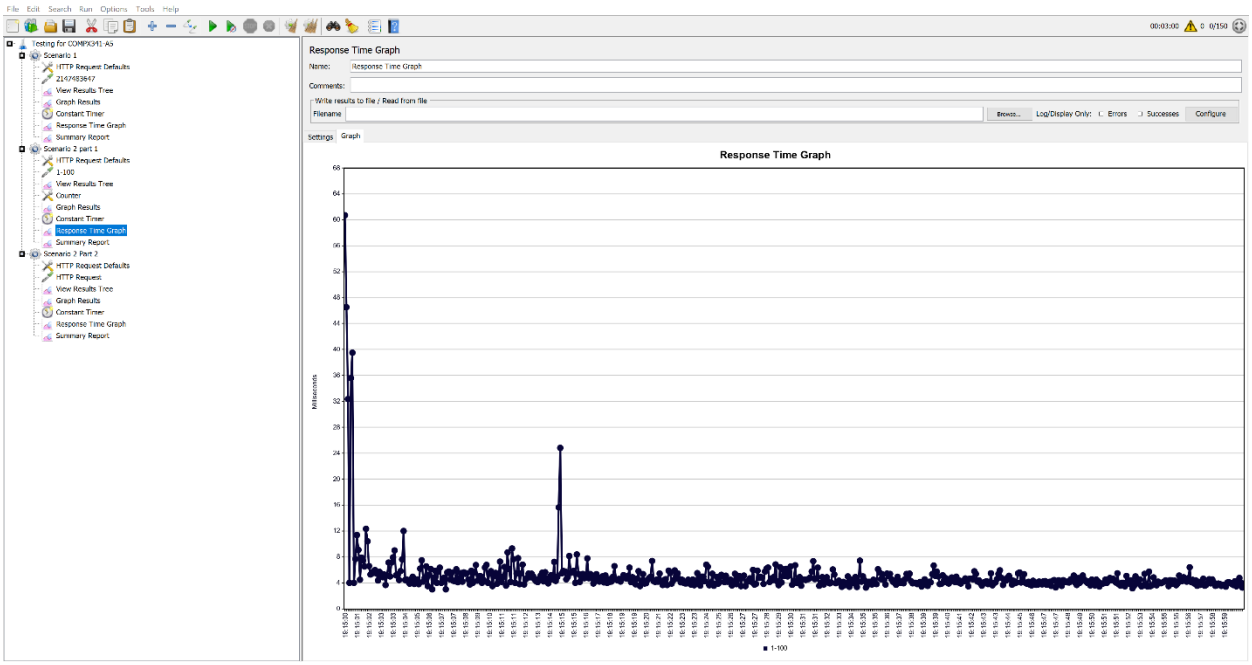
The final scenario, stress testing the primesStored value showed similar data to the 1-100 scenario. The throughput was 162.6 requests per second, with a total of 9709 requests sent over a 60second period. The response times barely fluctuated this time around, with an average of 7ms, median of 6ms and a peak of 178ms.

Graphs

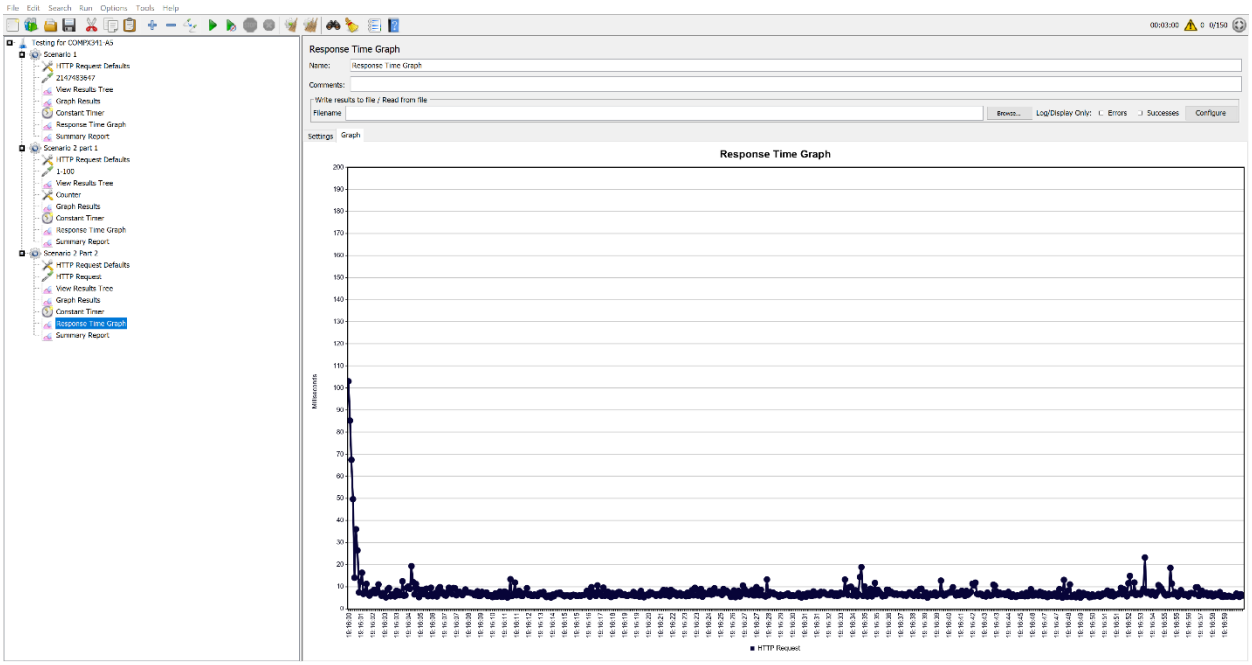
These graphs are all with no CPU or RAM limits, and 300ms thread delay



The response time graph for calculating if 2147483647 is a prime number

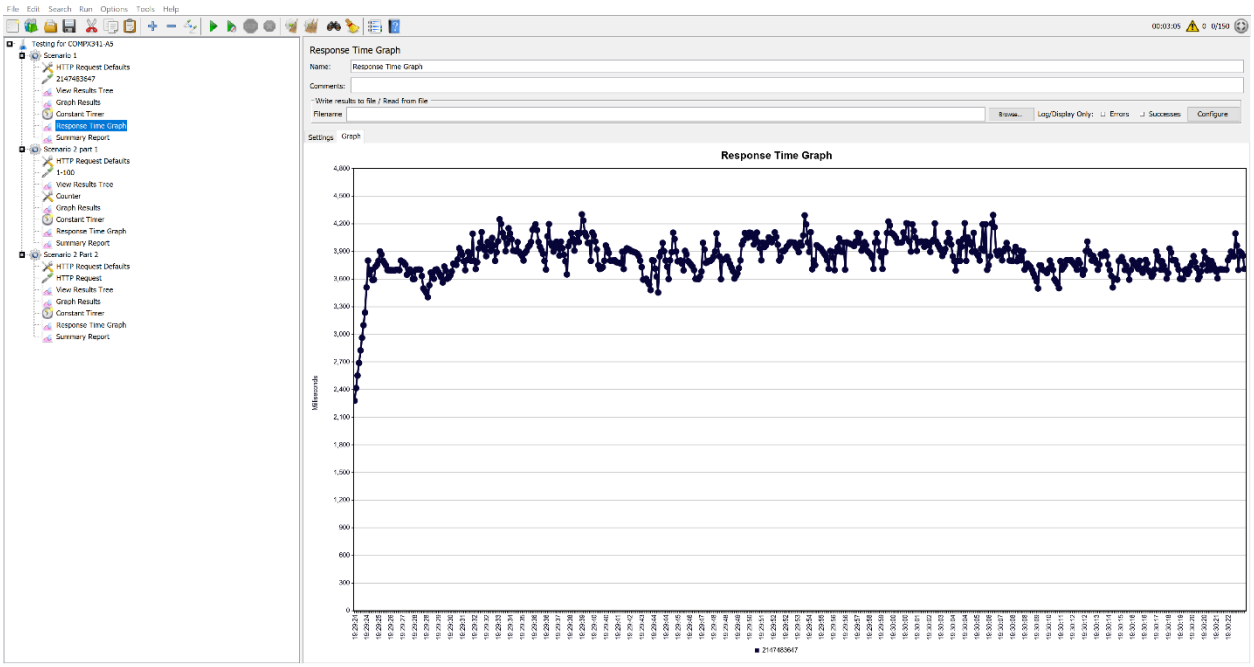


The response time graph for calculating which numbers from 1 to 100 are primes.

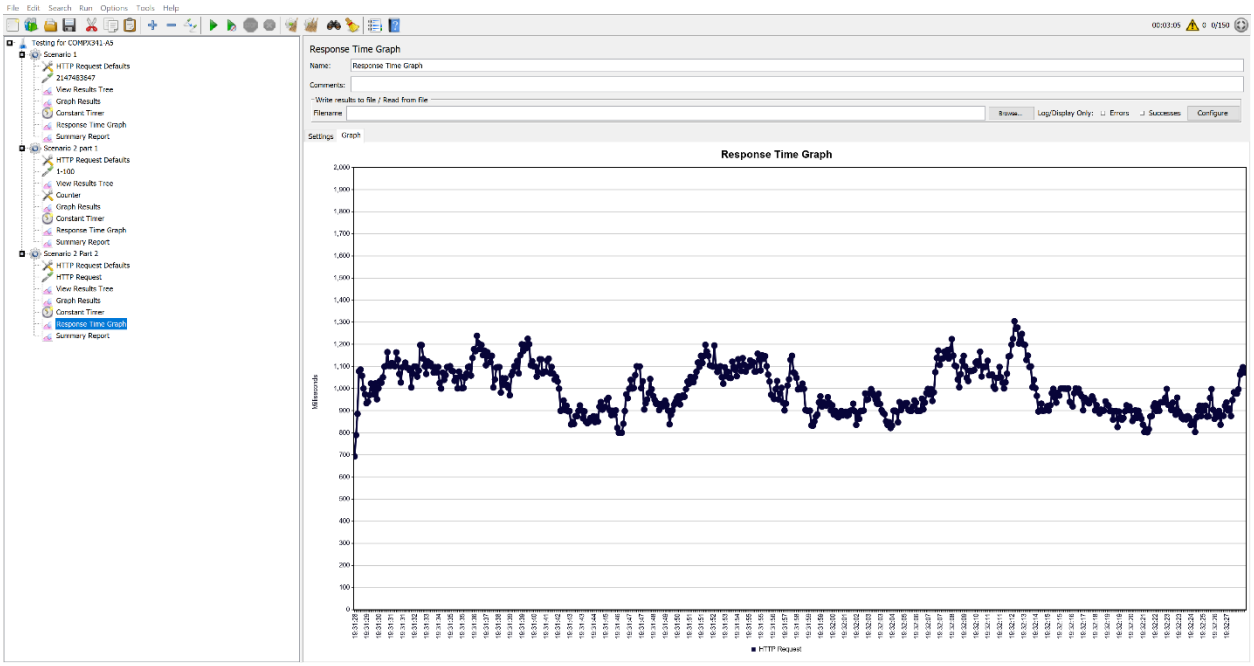


The response time graph for retrieving the stored primes from the Redis storage

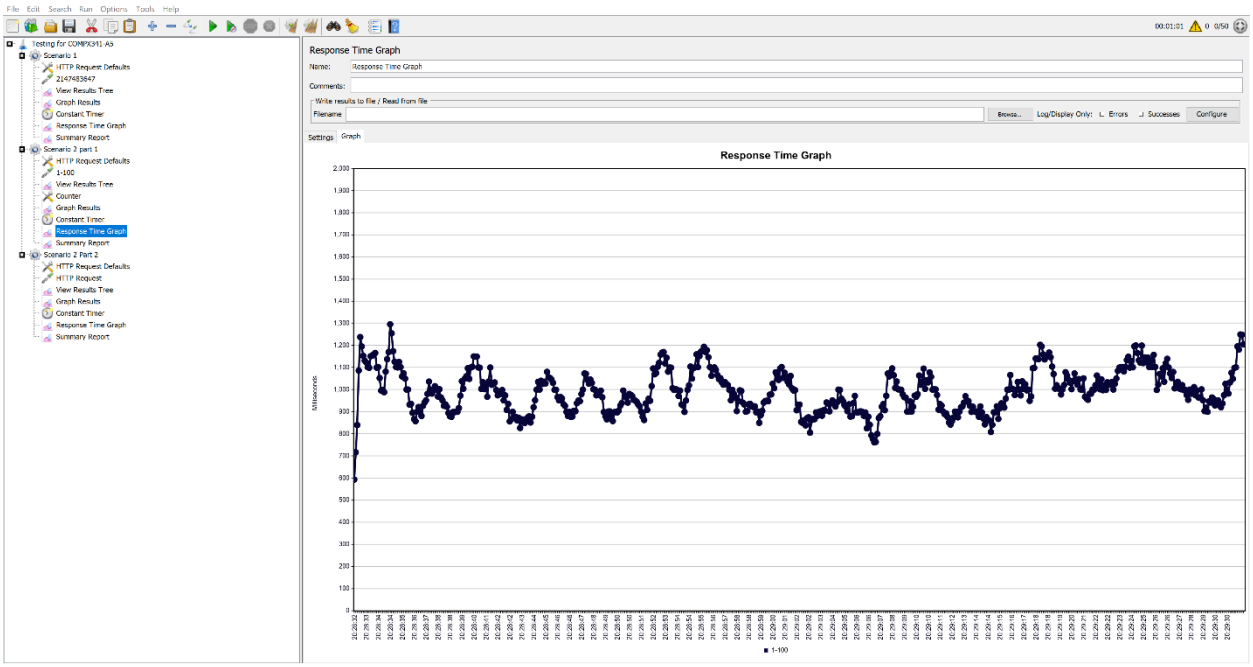
These tests are with the CPU and RAM limitations in place.



The response time graph for calculating if 2147483647 is a prime number

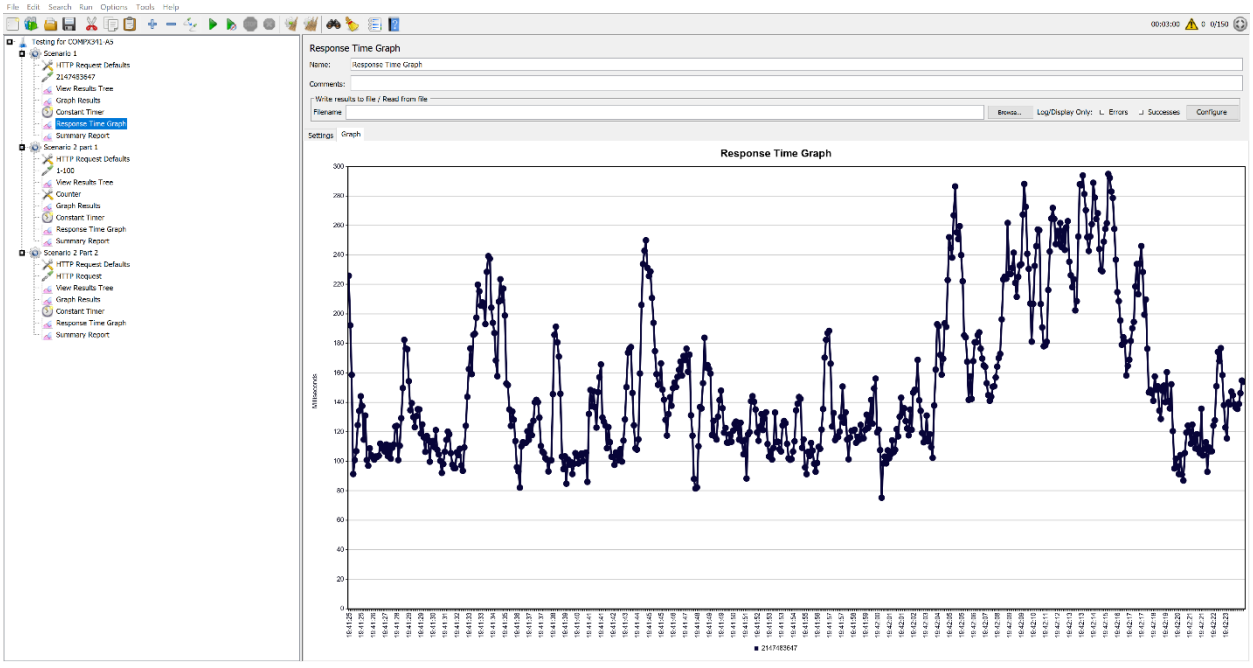


The response time graph for retrieving the stored primes from the Redis storage

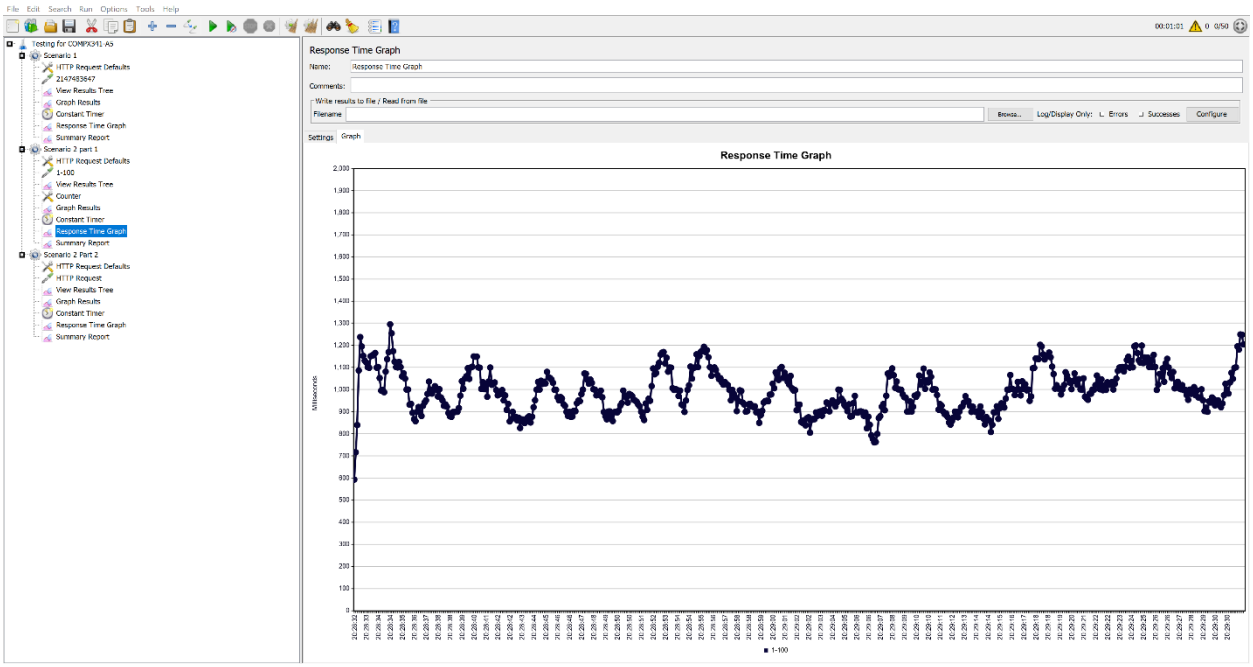


The response time graph for calculating which numbers from 1 to 100 are primes.

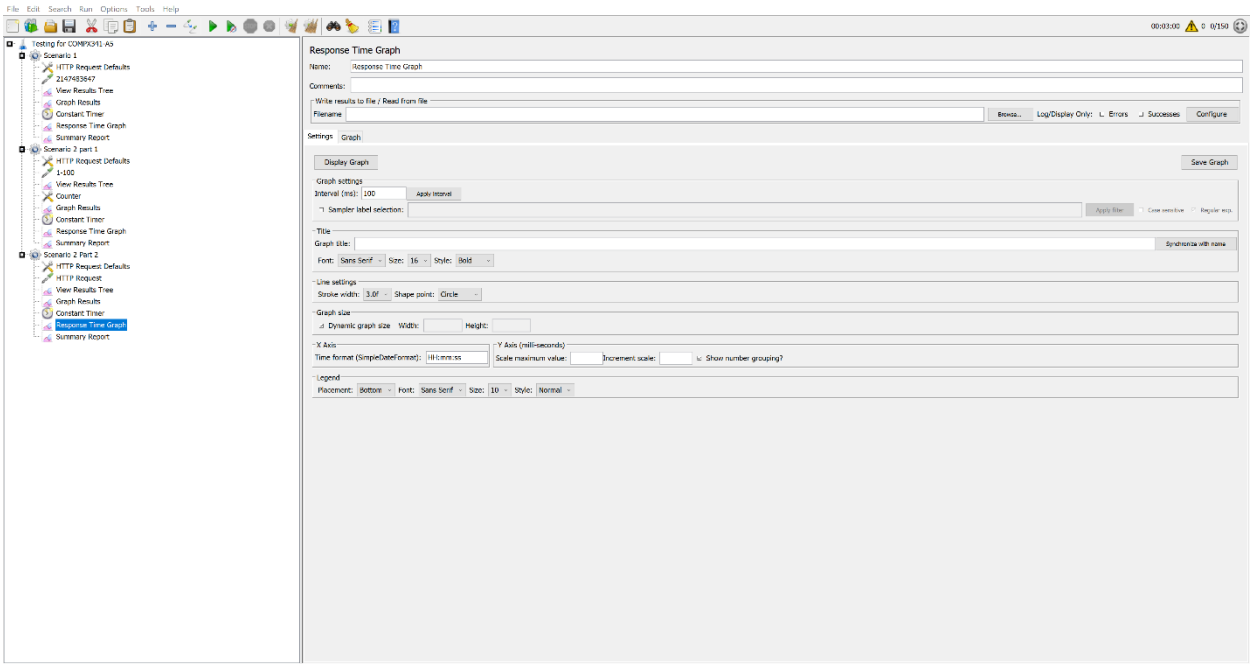
These tests are with no CPU limitation, but a RAM limitation.



The response time graph for calculating if 2147483647 is a prime number



The response time graph for calculating which numbers from 1 to 100 are primes.



The response time graph for retrieving the stored primes from the Redis storage

Manual testing:

If a number from 0-n, with n being an infinite number of possibilities is passed to isPrime, the API should return the correct output (either the number is a prime number, or it isn't.)

It should fail if the value is not between 0-n, such as negative numbers, or if a non-numeric character is sent to the API.

If primesStored returns either an empty cache value or the stored values, the API has returned the correct output.

If it is sent an argument it should not effect it, and it should ignore it and move on.

Code

All code is stored at https://github.com/nzgamer41/COMPX341-A5/ including JMeter testing files.