COMPX341 Assignment 4 Report

# Introduction

This report is for Assignment 4 for COMPX341 Software Engineering Methodology, where we were ask to develop and stress-test using JMeter on a simple containerized application-server that implements 2 Http Restful API which are:

1. /isPrime/<number>

Decides if the input integer is prime and returns “<number> is prime” or “<number> is not prime”. If input integer is prime, add to redis object storage service

2. /primesStored

Returns a list with all the primes stored in the connected Redis service. Ensure that there are no duplicates of the stored prime numbers

I did this Assignment in the labs so that I didn’t need to deal with putty and WINSCP. I followed the Docker-Compose Tutorial and was able to successfully create the app without much trouble.

Afterwards, I modify the docker-compose.yml file to suit the assignment requirements where I set the port number to the last 4 digit of my student ID 6832. I will be using this port number as my server port for the rest of this assignment.

From here, I updated the docker-compose.yml file to what was specified in the assignment documents and from there, I begin to make the necessary changes to the app.py code so that it could perform the requirement.

# 

# Creating the App and Test cases

Before I could actually update the app to do the requirements, I had to make sure I understood the documentation for Flask, Redis and Python because I have never used any of these languages and images before. Once I have understood the documents, I begin to set out to begin coding.

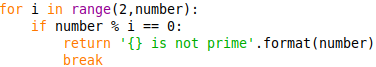
First of I had to create 2 methods where Flask needs to detect a change in the http request header and perform the correct python algorithm based on the inputs.

Image 1:

  
Image 1 shows an example of how that is done, where in the @app.route we call on the flask image so that when “/isPrime/<int:number>” is inputted into the browser header, it performs the following function.

The first function to create was the isPrime() function, the idea I had when it came to creating this function was just to take the number, check if its a number greater than 1. If it was 0 or 1, says that it is not a valid input because by definition, numbers 0 or 1 isn’t a prime number. From there if it the number was equal to 2 or more, we have a for loop and has a integer that increment after every division of the inputted number.

Image 2:



We use a modulus so that we have a easier time to check if the number is a prime number. If the number % integer ‘i’ which is incrementing with every loop equals to 0, it means it's not a prime number and we break out of the loop to end early. Of course, if the number is a prime number, we would then says that the number is a prime and add it to the redis storage.

The good thing about Redis is that it works similarly to a external database using keys, much like a hashtable. This makes it easy as we just need to create a key that all the prime numbers are associated to and just append it to the Redis storage Object.

Image 3:

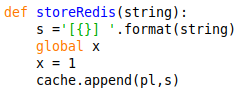
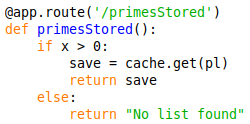


Image 3 shows how we add the found prime numbers to the redis storage object. First we cast the number from integer into strings because the redis append function takes a key and a string value. From there we format the string data to added square brackets around the number before we append it to Redis so that there is some clear distinction what the different numbers are.

From there we go on to make the primesStored() function that just outputs and displays the list of prime numbers stored.

Image 4:

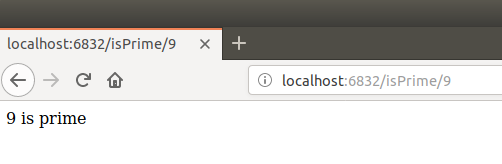


We just use the redis function get() to display the list of prime numbers that we saved based on the key that we provide which in this case is “pl”.

## Black and White box coverage testing

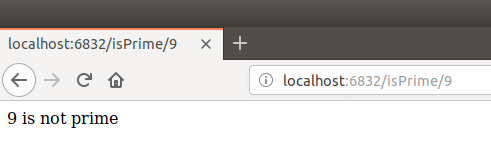
This is the section where I talk about how bad my initial code for my application is. While i was making the application, there were alot of syntax error and type cast error where I used the wrong syntax in my code resulting in error in my code not working. After I reached a point where my code was no longer calling out errors. I proceed to do black and white box coverage testing to find out bugs and fixed them.

One of the first notable test cases I did was that the isPrime() function was able to accurately identify a prime number. One of the first few problems I had was that although the function was identifying prime numbers that were even, it couldn’t identify prime numbers that were odd. An example I would give is that, I will give the input number of “9” and the function would return it as a prime number when it is clearly divisible by 3.

Image 5:  


This was because, we had a for loop where it will iterate through from number 2 to the input number to see if it was divisible by any number before it. If it wasn’t divisible, it will be identified as a prime number and added to the redis storage. The problem here was that the function was added the prime number immediately after it was tested to be divisible by 2, since most odd numbers aren’t divisible by 2, they would have been considered as a prime number before they were tested with other numbers. Hence we had to make sure that the numbers were only added if and only if, they have iterated all the numbers from 2 to the num and made sure that they were a prime number.

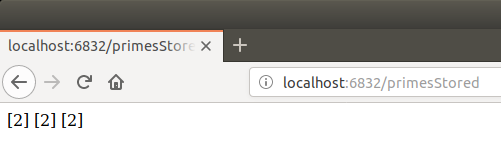
Image 6:



Of course, there was an issue when we are looking for a really big prime number such as 2147483647 it took a really long time for the function to identify it as a prime number because it needed to iterate from 2 to 2147483647 and divide it by that iterating number. So I had to hard code it in order to speed things up, where if the function finds 2147483647, it immediately knows its a prime number and adds it to the redis storage object.

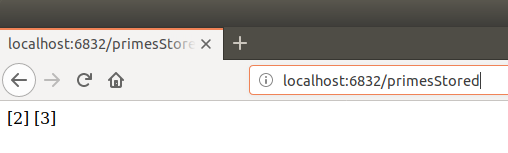
There was also another notable test cases where the primesStored() function was adding unnecessary duplicates into the redis storage object.

Image 7:



This was easily remedied by creating a global python list that stores all of the identified prime number and before adding any inputted number onto the redis list, we scan through the python list first to see if that particular number has already been added to the redis storage object.

Image 8:



Any other test cases done has been simplified to the chart below

|  |  |  |
| --- | --- | --- |
| Testing inputs | Expected Results | Explaination |
| 0 | Number must not be 0 or 1 | 0 is not a prime number |
| 1 | Number must not be 0 or 1 | 1 is not a prime number |

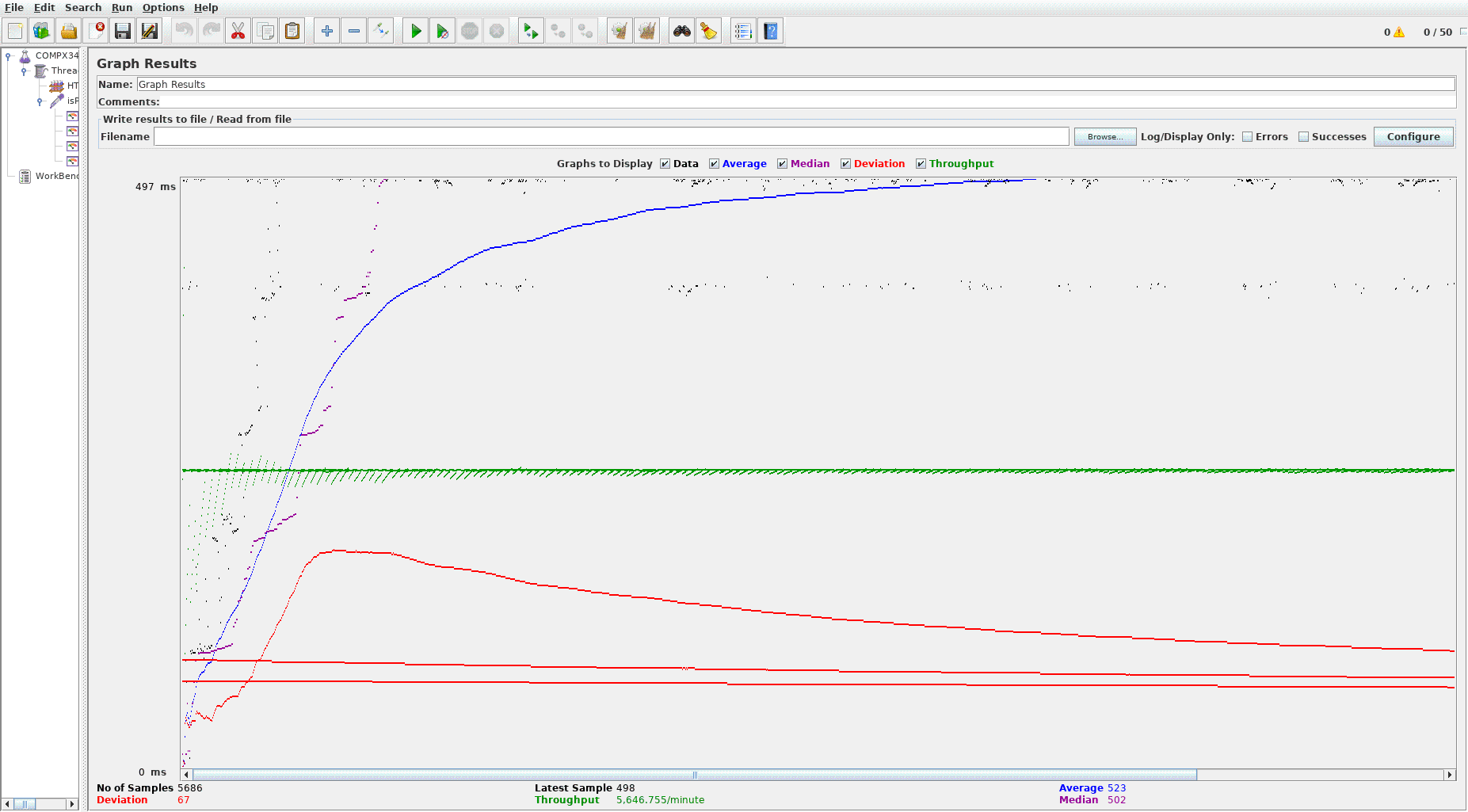
# Jmeter Stress Testing and Analysis

In this section, I will talk about the various stress test that I did to test my application based on the 2 scenarios provided, the changes in the CPU Share test as well as the Timer delay tests.

## Scenario 1

Scenario 1 is where we use 50 threads over a 60s time to repeatedly send a request with the number 2147483647 to decide if that number is prime and is invoking the application isPrime URI correctly. This is done by creating a thread group with 50 users and a scheduler of 60s and bascially just sending a HTTP request that invokes the isPrime() URI with the input value of 2147483647

1Graph.png:



If the graph values are a bit hard to read here, it might be easier to view the screenshot from Github, the image file is called “1Graph.png” and the link to the GitHub is here:

<https://github.com/Kamicandre/COMPX341_A4/blob/master/ScreenShots/1Graph.png>

Report Results:

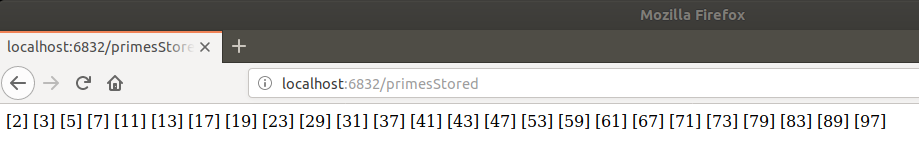


As you can see from the graph, initially the application seems to be able to process the requests fine before it starts being overloaded with the numerous number of request search for the same number. You can see the average slowly going up it seems to gradually even out to provide a more stable responses to the request. As you can see from the green line, the application is able to handle a large amount of request with not much difficulty. Since throughput signifies the number of request that can be made in a given period of time, it shows that the application is able to handle a lot of request per minute with the throughput being 5646.755 per minute. One hypothesis I have about this scenario is that because the application is receiving the same request over multiple times, it slowly adjust itself and is able to handle the load effectively.

## Scenario 2

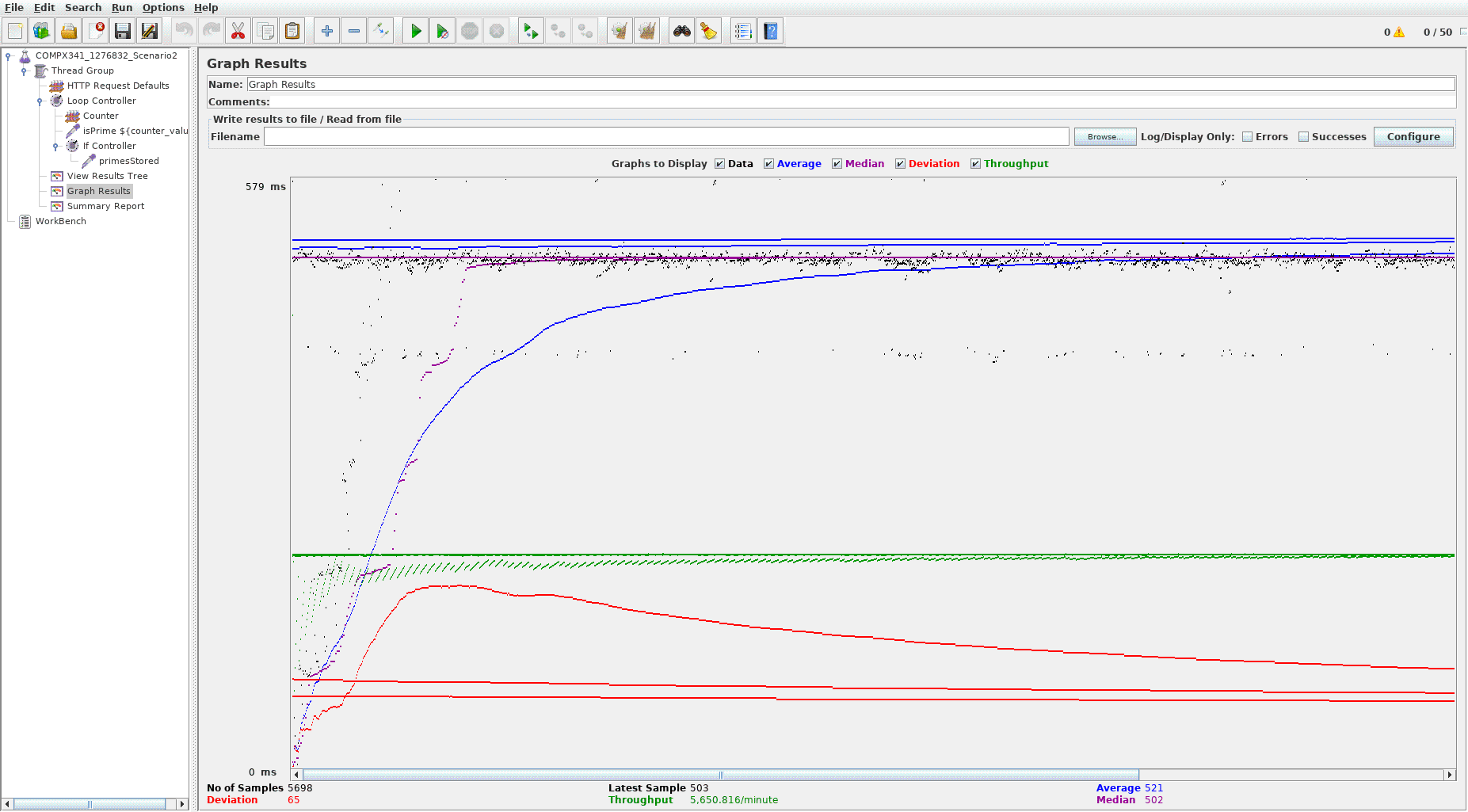
Scenario 2 is where I invoke the isPrime API for all numbers between 1 and 100; then it repeatedly invokes the primesStored URI of the app. I understood this is that Jmeter will have a counter going from 1 to a 100 and invoking the isPrime API for those number and once it has reached a 100, it will invoke the primesStored URI, and then it repeats itself again. This is done by creating a loop controller and a counter where the loop invokes the isPrime API with the counter incrementing from 1 to a 100 and inputting the counter value as the prime number we want to identify. Then we have a IF controller where if counter reaches a value of 100 it will invoke the primesStored() API so that it will display the list of prime numbers from 1 to a 100 like the image below

primesStored() function output:



2Graph.png

<https://github.com/Kamicandre/COMPX341_A4/blob/master/ScreenShots/2Graph.png>



Report Summary:

<https://github.com/Kamicandre/COMPX341_A4/blob/master/ScreenShots/2summary.csv>

For Scenario 2, it was really surprising to see a very similar result to Scenario 1, the graph looks almost identical. What surprise me most was that for Scenario 2, the throughput value is slightly higher than Scenario 1 with a value of 5650.816/min. This shows that as the number of request increase, the application is able to handle it well. Both scenario shows very similar results for the application which shows how well it can handle multiple request with different or same inputs.

For the next set of test, with regards to CPU and Timer Delay, I will be using Scenario 1 as the base test scenario, with differing CPU share values and different types of timer delays

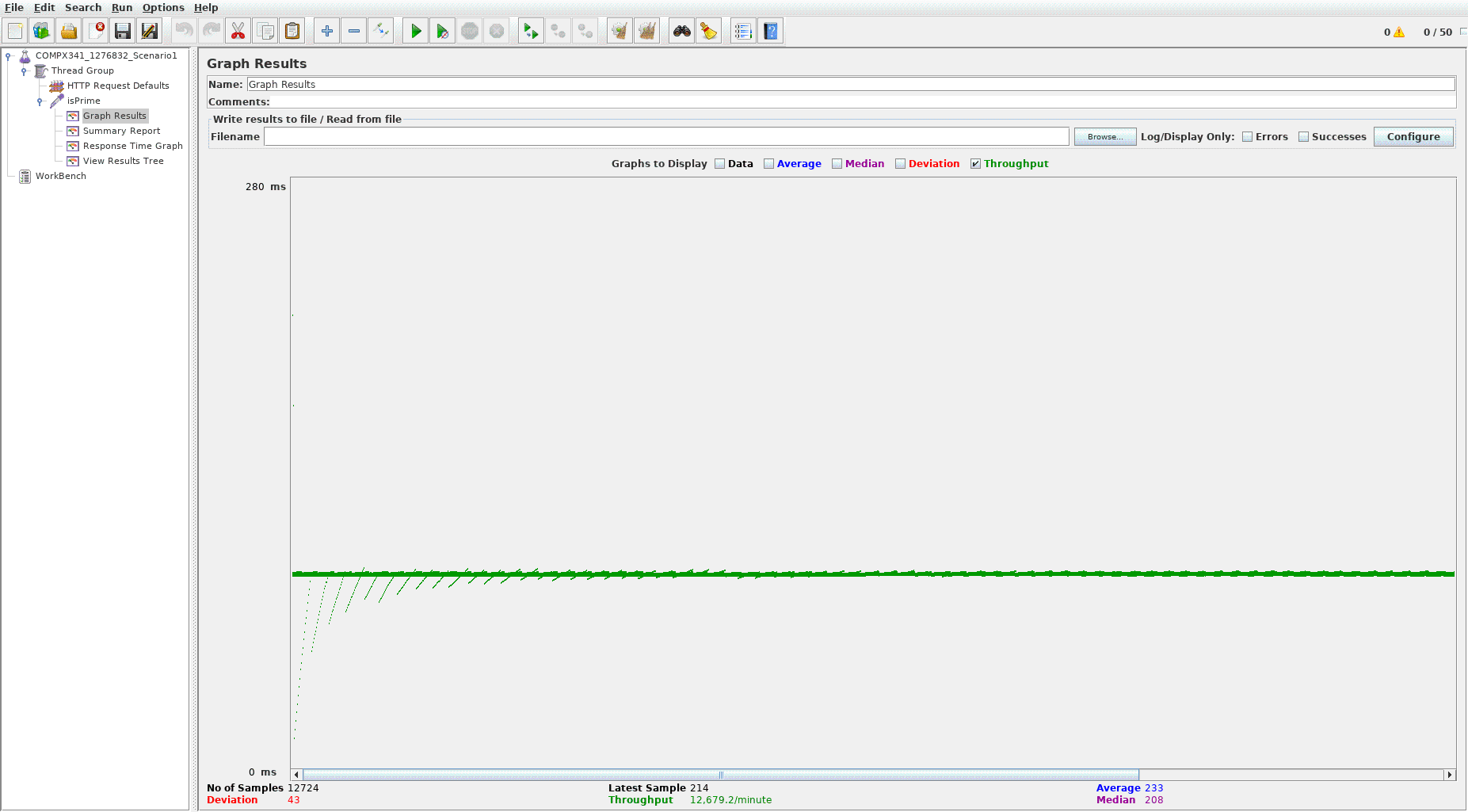
## CPU Tests

This is where I change the CPU share for the web container of the application in the docker-compose file and see if there were changes to the response time and throughput data of the Jmeter test for Scenario 1.

0.2 CPU Share:

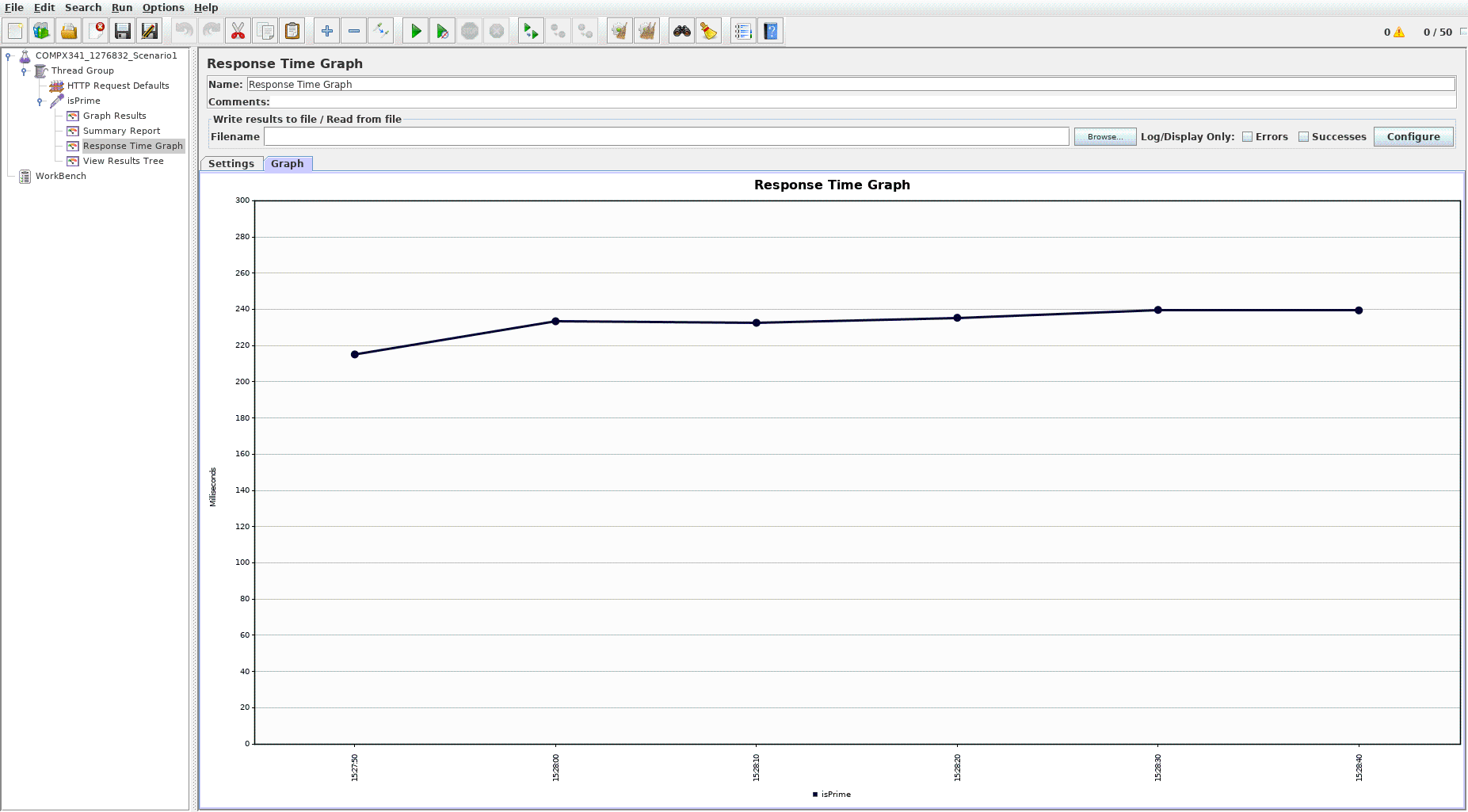
0.2Cpu Graph Results.png:

<https://github.com/Kamicandre/COMPX341_A4/blob/master/ScreenShots/0.2Cpu%20Graph%20Results.png>



0.2Response Time Graph.png:

<https://github.com/Kamicandre/COMPX341_A4/blob/master/ScreenShots/0.2Cpu%20Response%20Time%20Graph.png>

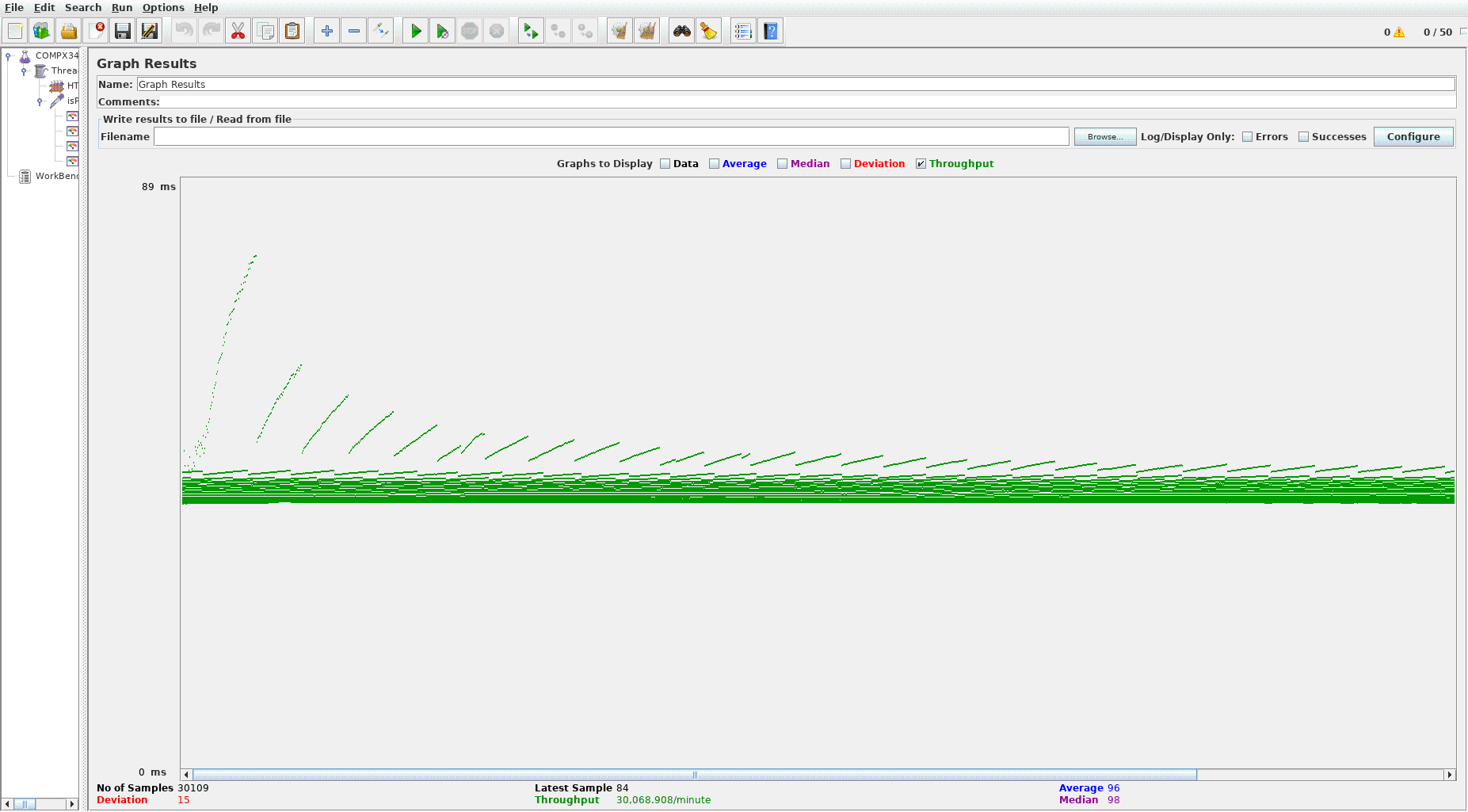


With the increase in CPU shares, the application throughput value increased to 12679.2/min showing that the application is able to handle a lot more requests in a minute. They also have a much faster response time, dropping to a response time of about 240ms which explains why the application is able to handle so much more request and hence the increase in the throughput data.

0.5CPU Share:

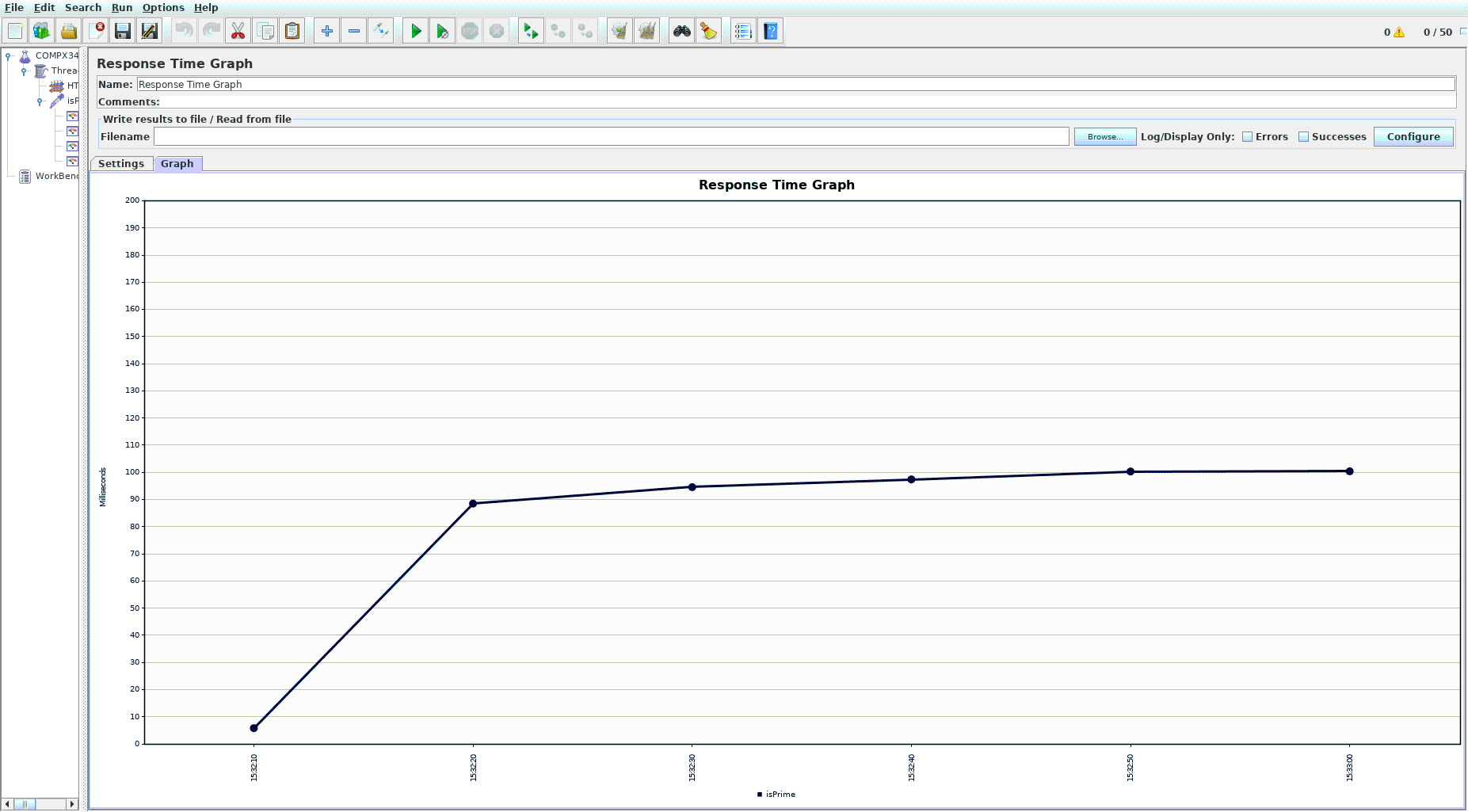
0.5Cpu Graph Results.png:

<https://github.com/Kamicandre/COMPX341_A4/blob/master/ScreenShots/0.5Cpu%20Graph%20Results.png>



0.5Cpu Response Time Graph.png:

<https://github.com/Kamicandre/COMPX341_A4/blob/master/ScreenShots/0.5Cpu%20Response%20Time%20Graph.png>

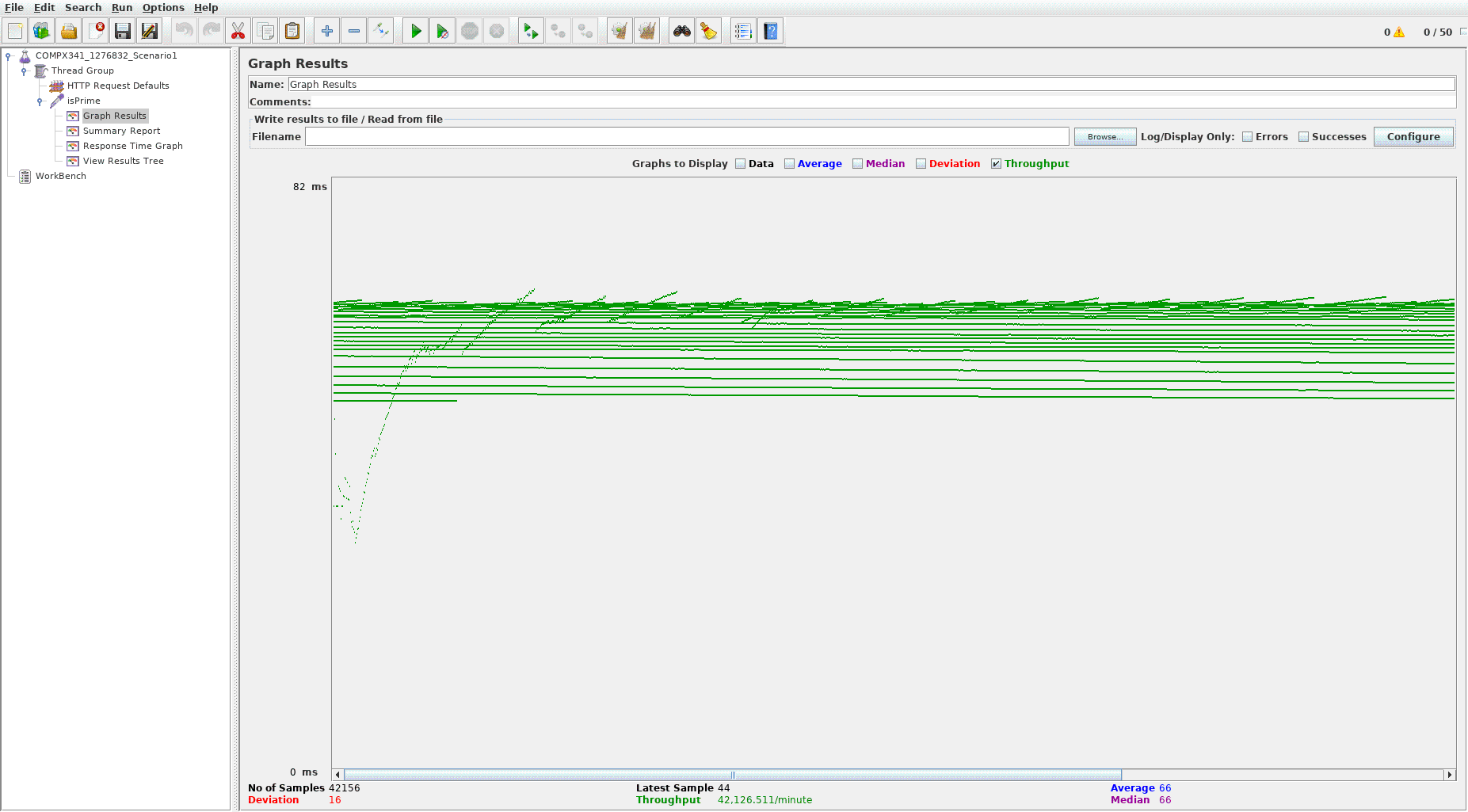


Similarly to 0.2CPU Share section, the throughput value increases to 30068.908/min, show that the application can handle more request with the increase in CPU shares. The response time graph also shows a decrease to a maximum of 100ms per request. With an initial response time of below 10ms, showing how quickly the application handle the inital request.

0.8CPU shares:

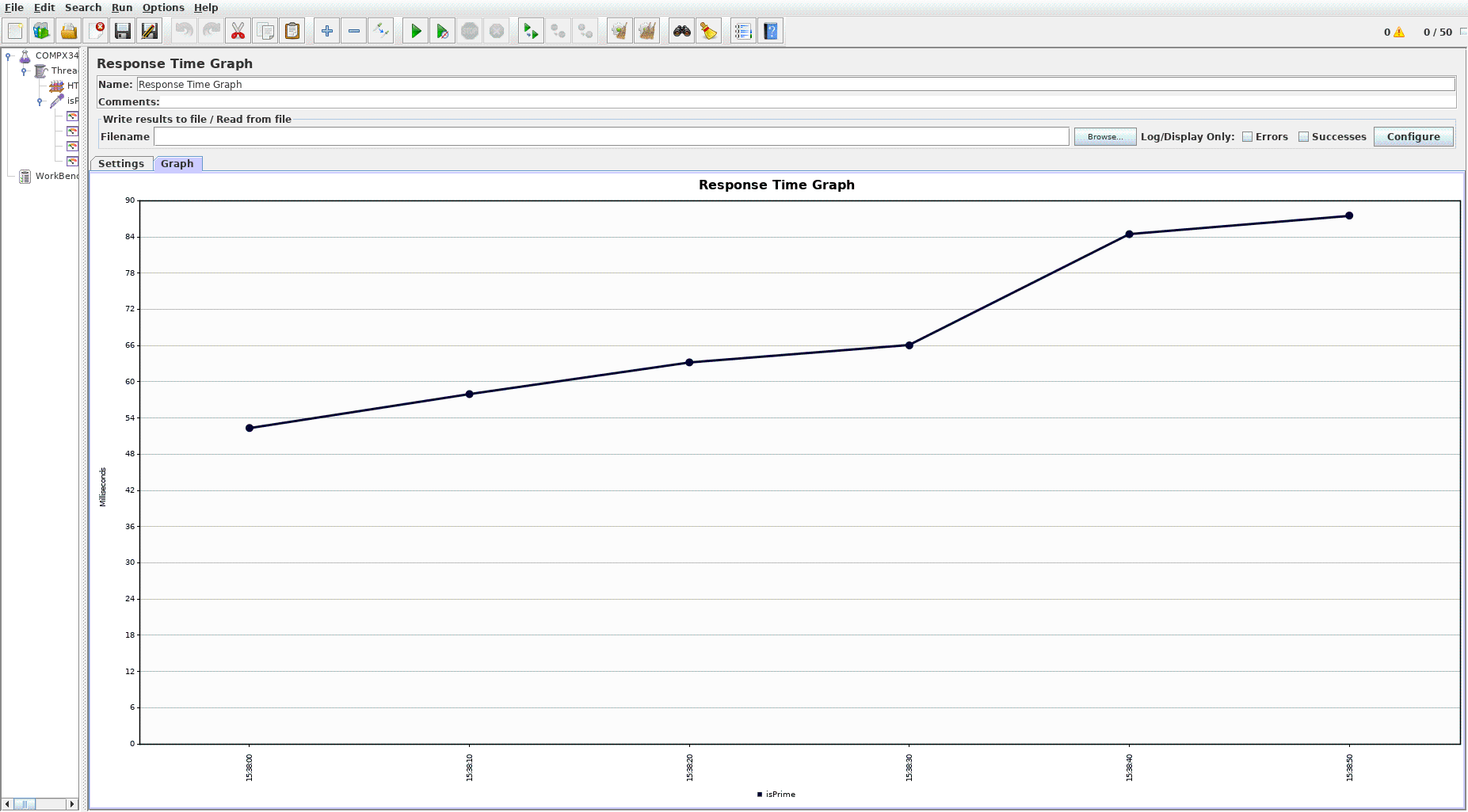
0.8Cpu Graph Results.png:

<https://github.com/Kamicandre/COMPX341_A4/blob/master/ScreenShots/0.8Cpu%20Graph%20Results.png>



0.8Cpu Response Time Graph.png:

<https://github.com/Kamicandre/COMPX341_A4/blob/master/ScreenShots/0.8Cpu%20Response%20Time%20Graph.png>



And finally, with 0.8 cpu share, the application is able to multiple request well with a throughput data of 42126.511/min and a response time of 90ms. Though on an interesting note, the 0.8CPU share response time for the initial requests is a lot higher than the response time for the initial in 0.5CPU share.

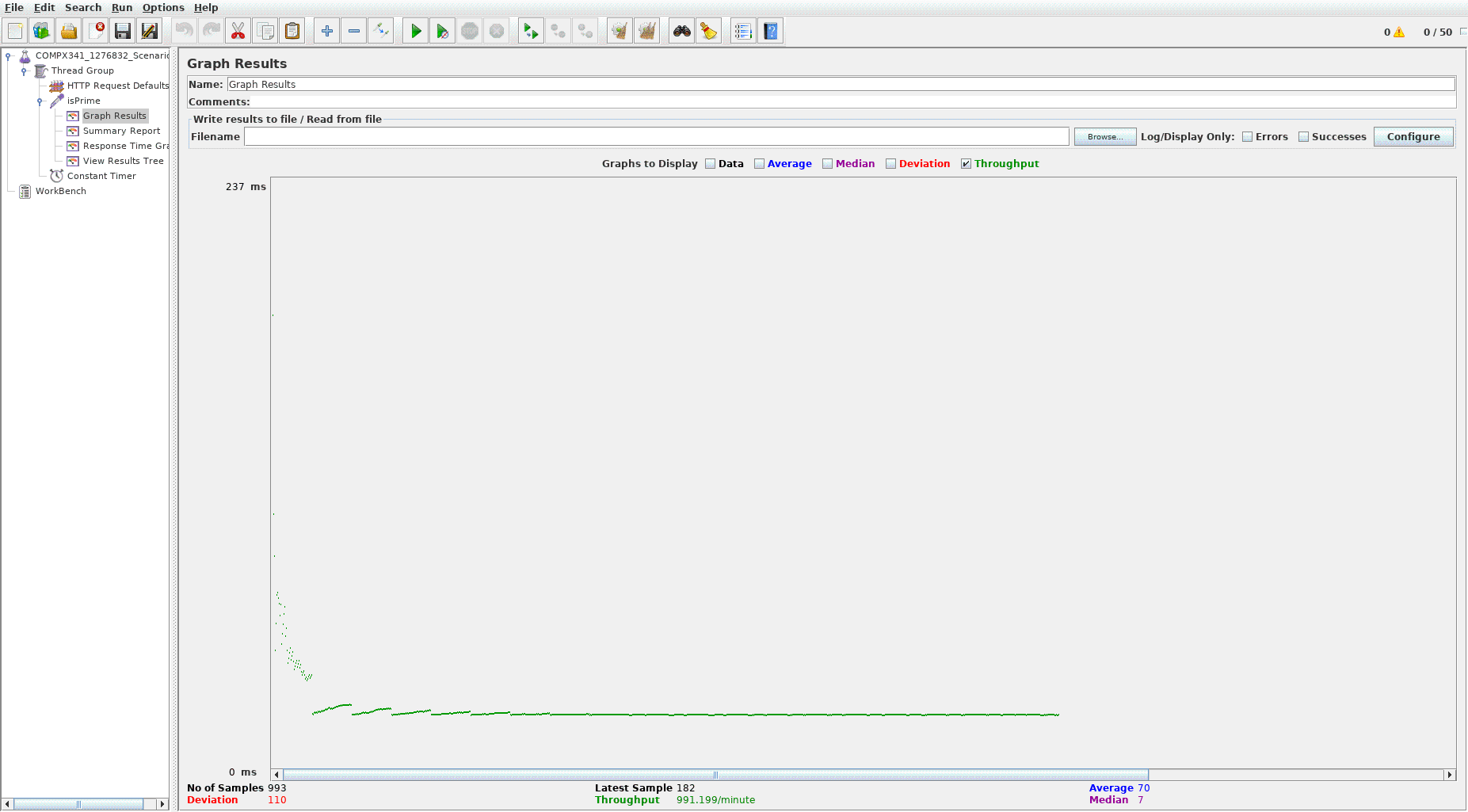
## Timer Delay Tests

This section is where i will show my test result for throughput data and response time with a normal cpu share of 0.1 and use scenario 1 as a stress testing with the addition of 3 different timers to pseudo simulate a user request timing or pacing.

Constant Delay Timer:

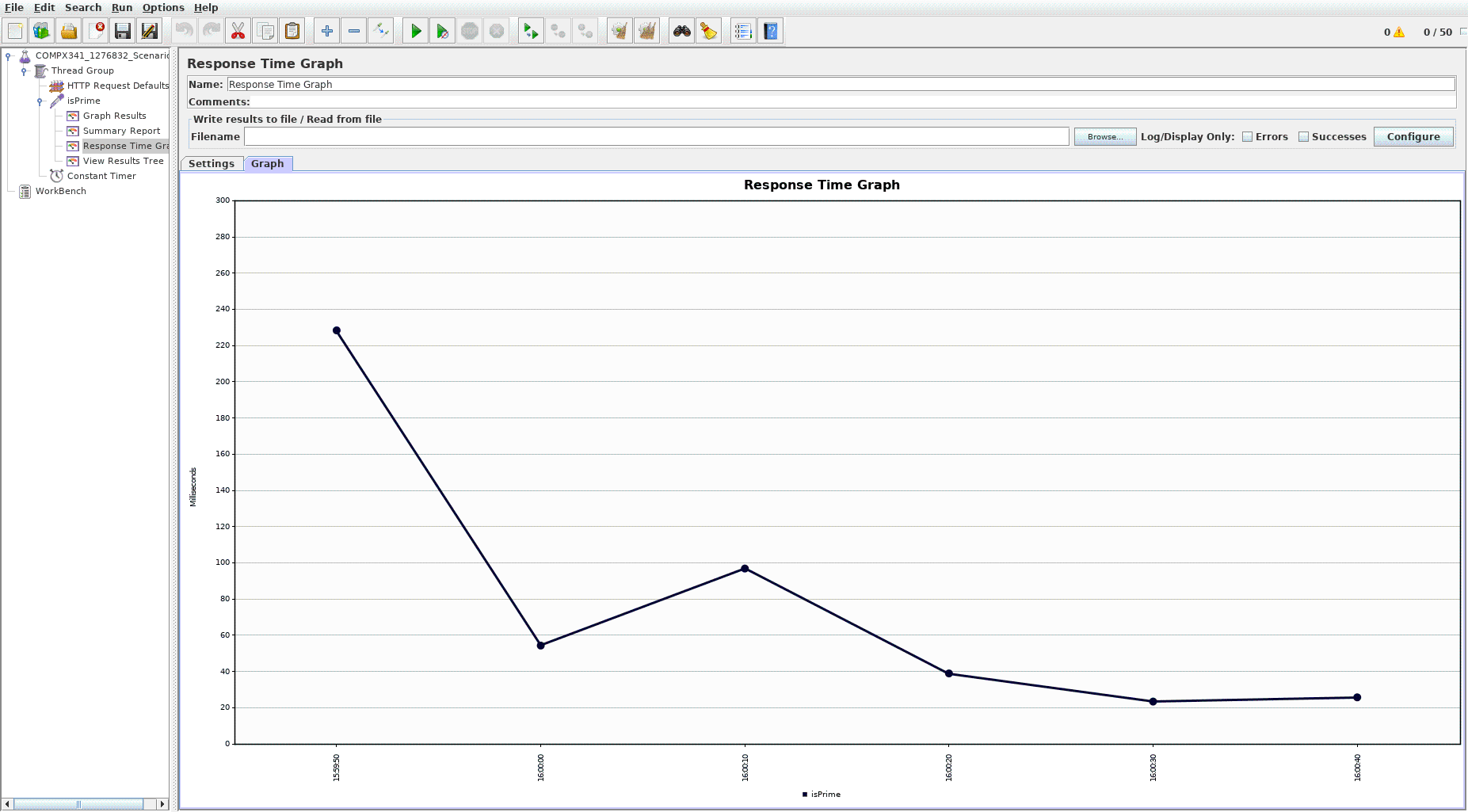
ConstantDelay Graph Results.png:

<https://github.com/Kamicandre/COMPX341_A4/blob/master/ScreenShots/ConstantDelay%20Graph%20Results.png>



ConstantDelay Graph Results.png:

<https://github.com/Kamicandre/COMPX341_A4/blob/master/ScreenShots/ConstantDelay%20Response%20Time%20Graph.png>

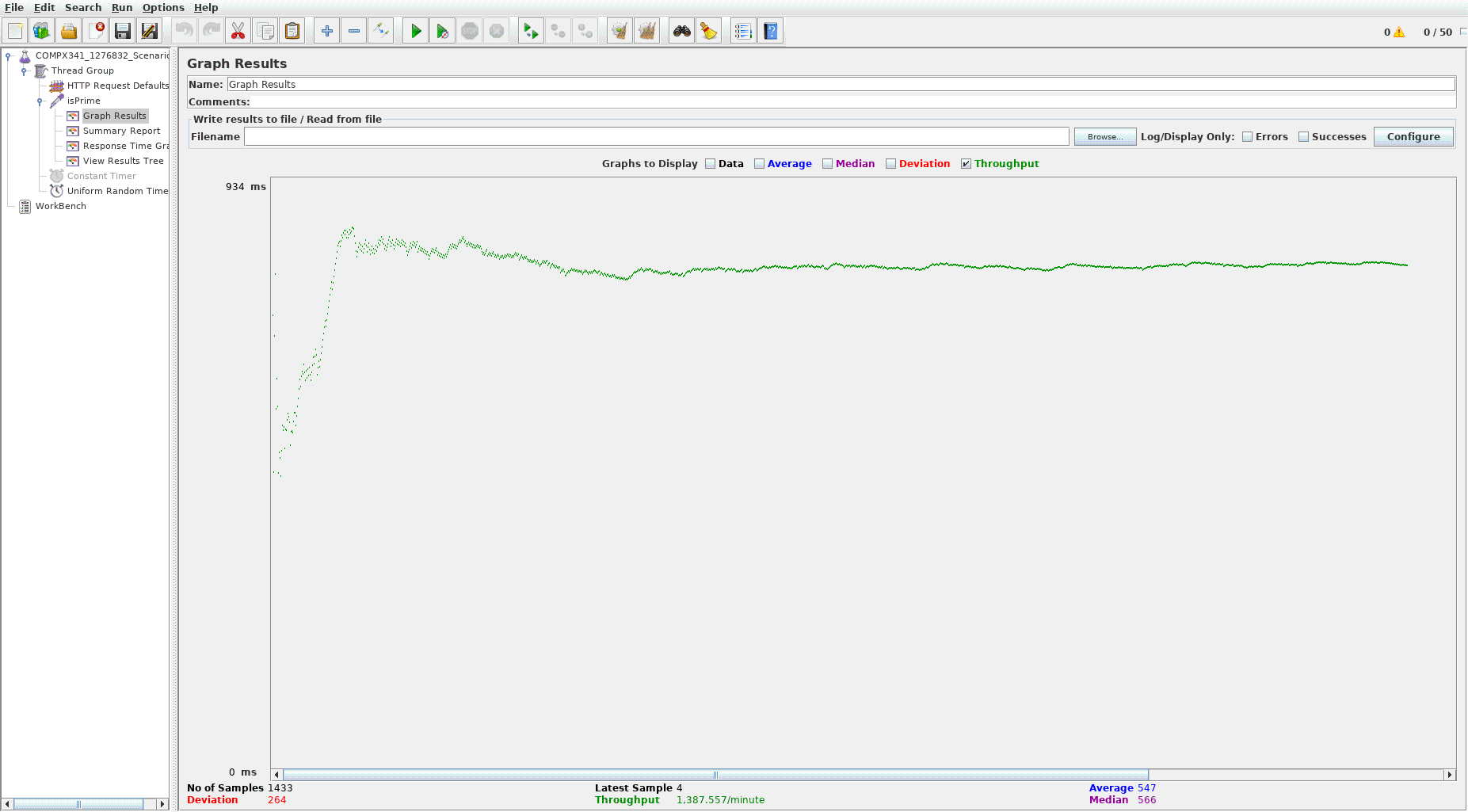


Throughput Data of 991.199/ min and a response time max of 230ms.

UniformRandom Timer:

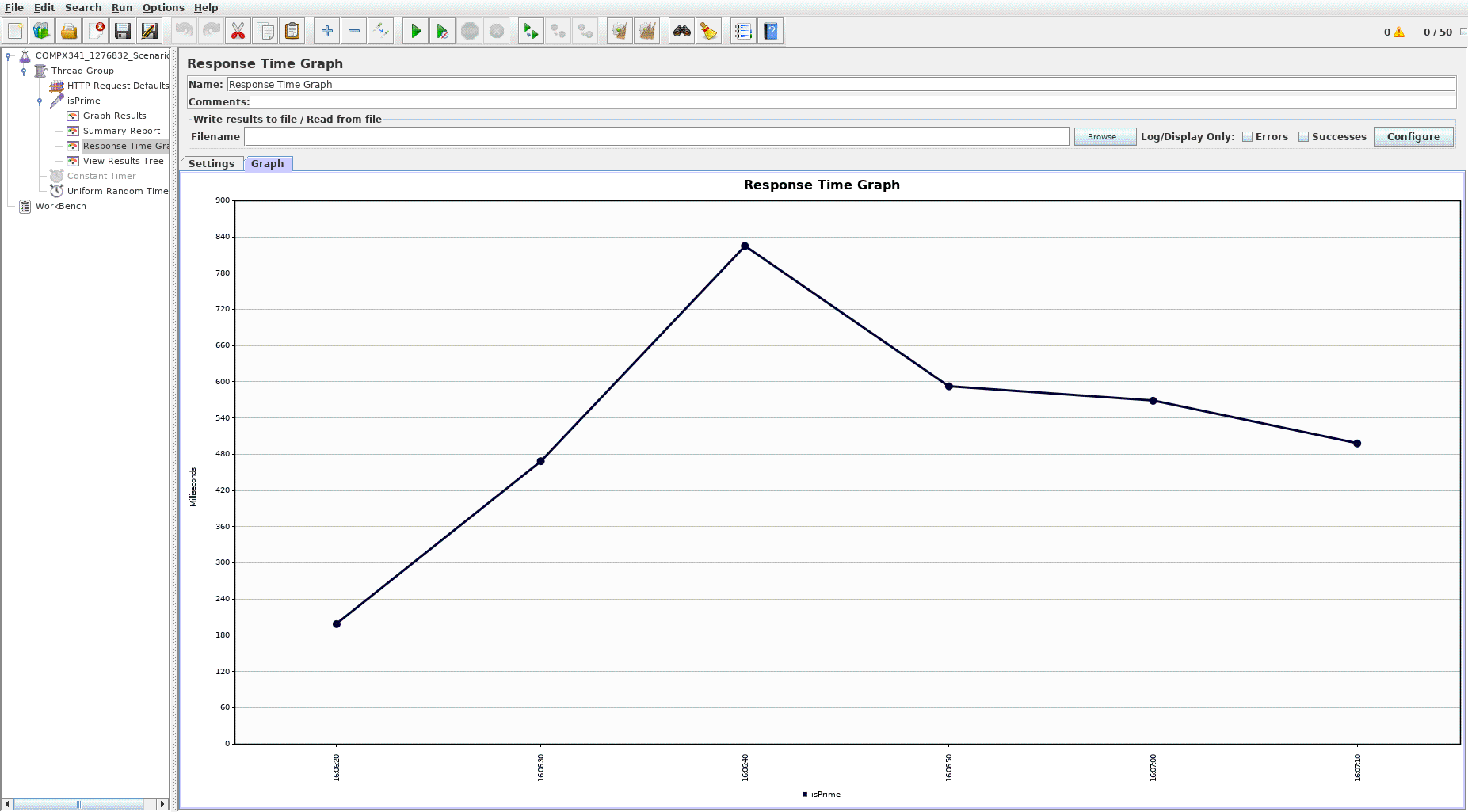
UniformRandom Graph Results.png:

<https://github.com/Kamicandre/COMPX341_A4/blob/master/ScreenShots/UniformRandom%20Graph%20Results.png>



UniformRandom Response Time Graph.png:

<https://github.com/Kamicandre/COMPX341_A4/blob/master/ScreenShots/UniformRandom%20Response%20Time%20Graph.png>

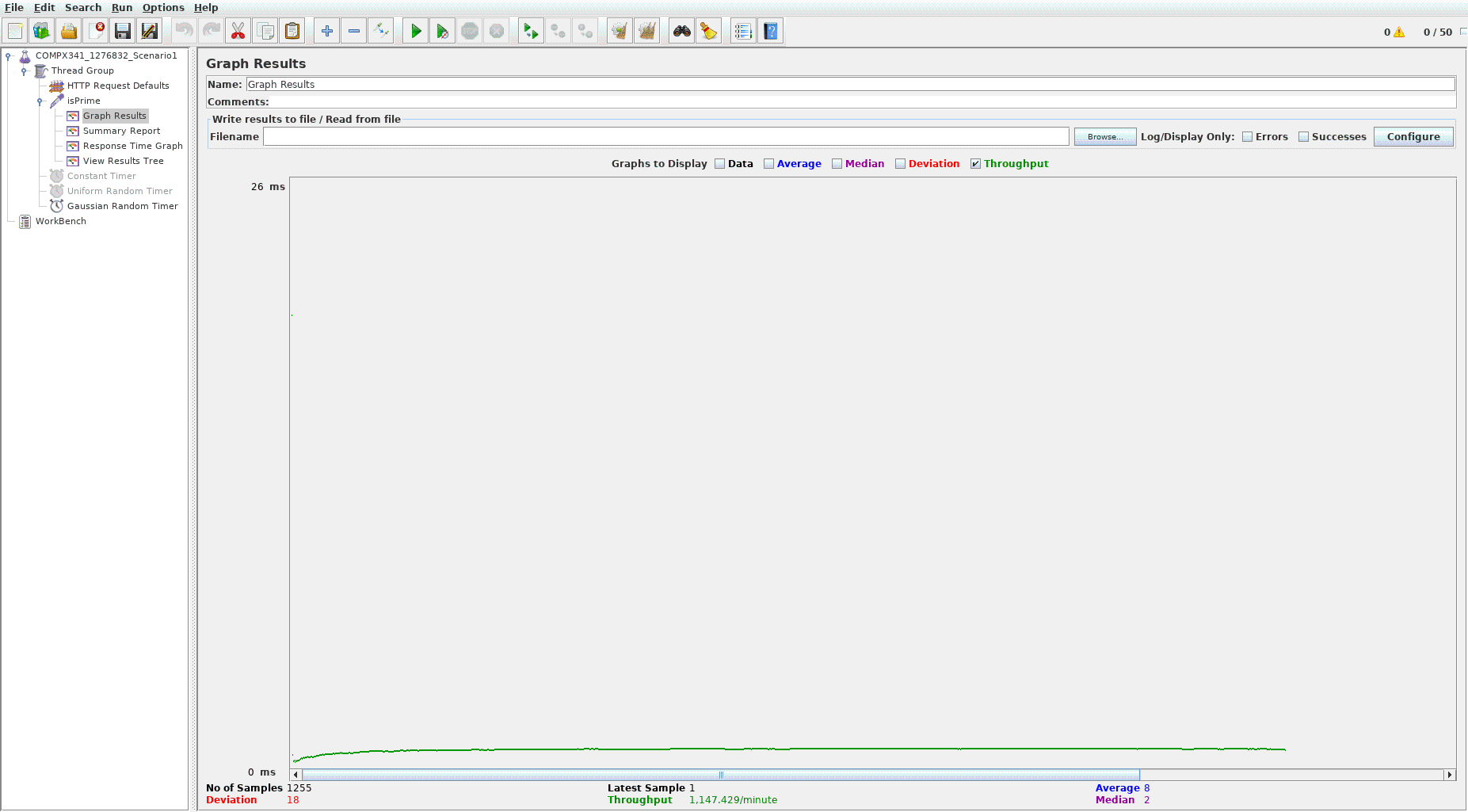


This timer delay was interesting because how high the response time is with a value of 840ms. The throughput data is 1387.557/min showing that the application can still handle it, just that it has a much higher response time.

Gaussian Random Timer

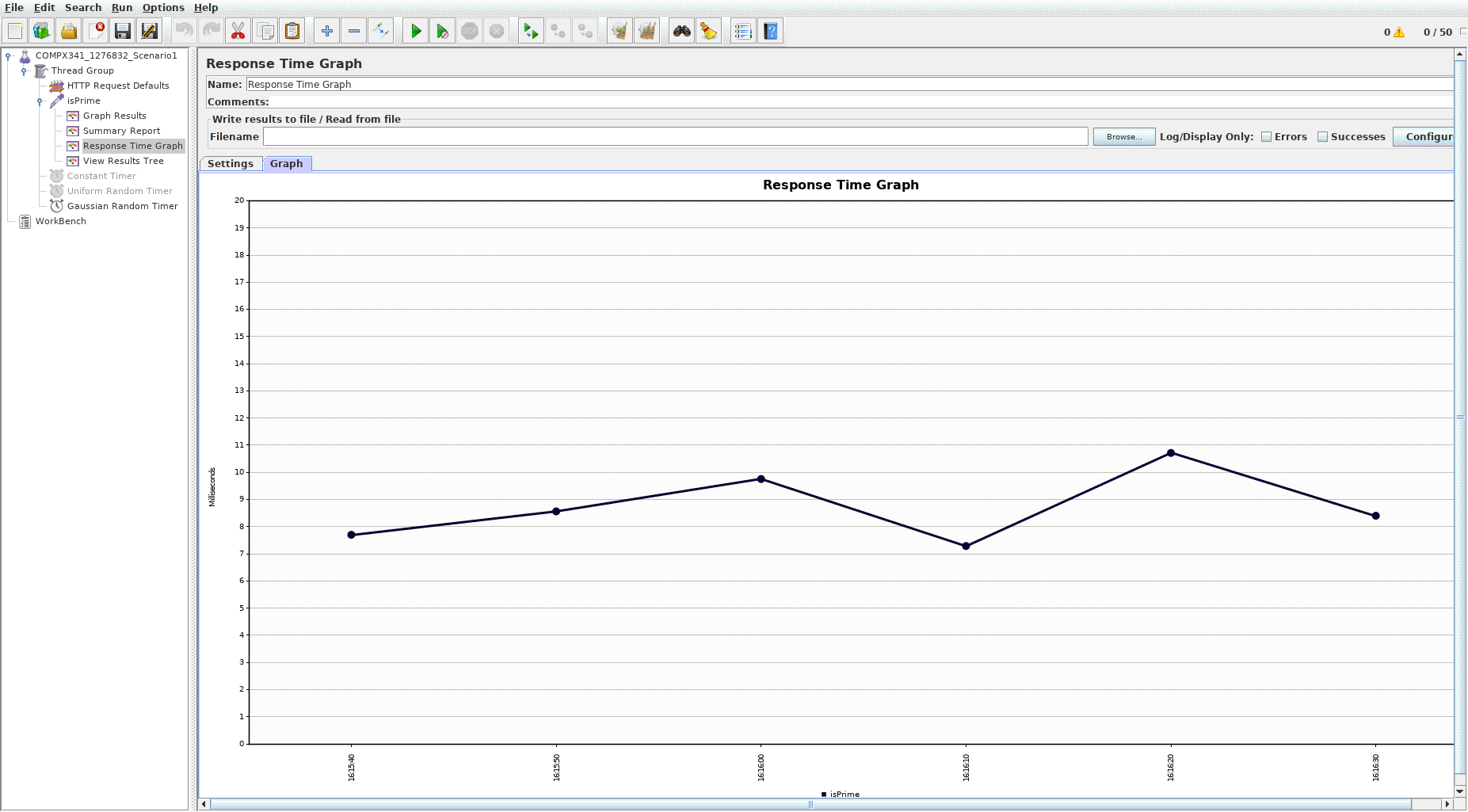
GaussianRandom Graph Results.png:

<https://github.com/Kamicandre/COMPX341_A4/blob/master/ScreenShots/GaussianRandom%20Graph%20Results.png>



GaussianRandom Response Time Graph.png:

<https://github.com/Kamicandre/COMPX341_A4/blob/master/ScreenShots/GaussianRandom%20Response%20Time%20Graph.png>



This one was interesting because of quick the response time is compared to the previous two delay timers. The throughput data is rough similar with a value of 1147.429/min and a really quick response time of 11ms.

# Conclusion

Overall this Assignment has been an interesting experience for me in terms of using docker to create containerized application-server. Also it is my first time coding in python so that was also another interesting experience. I can see how useful stress testing application such as JMeter can be useful for testing how well an application can handle large amount of request. This make me wonder how the hell is google able to handle that many requests and still operate quickly enough to make it seems like it can provide whatever request the user sent through their search engines. I do wish that I understood the results given from JMeter better so I could have better analyse how well or badly my application was doing. I only understood that throughput was how much transaction or request the application is able to handle and the response time is how quickly the request was process. This was a very interesting experience as well as a painful one because it show how inadequate I am in understanding JMeter. It was still interesting since it gave me my first time experience dealing with Docker, Docker-compose, Jmeter as well as python.

GitHub Repository Link: <https://github.com/Kamicandre/COMPX341_A4>