Load Testing

Bedir Asici

Contents

Introduction

Load testing is an important part of the Software Development Lifecycle. It allows for the simulation of real life scenarios. Before deployment load testing allows engineers to gauge the performance and durability of the product under various stressors such as amount of users or time limits. It allows for the product to be resource and financially efficient.

For this project a simple web server which calculates the factorial result of a passed in integer variable was used. The web server kept record of entered numbers and resulted in a ram based storage system known as redis. The values that were stored were queryable with the "calculationsStored" function which returned a list of the numbers and resulting factorials.

Jmeter was used to create test scenarios that would query the web server for both the factorial function and the calculationsStored function. Vagrant was used as a server system. Vagrant allowed for a host and guest virtual user system that could have multiple guests at the same time. Docker was used to launch the python application as a webserver on top of the vagrant virtual machine. Docker was an important piece as it had a cpu limit that was adjustable. The cpu limits used for this assignment were 1.0, 0.9 and 0.8.

Python code completion

For the first requirement which requested the allowance of negative integers to be processed in addition to positive integers. The signed parameter within @app.route was utilised. Setting the value to signed = True allows for negative values to be accepted without causing a system error. From there if the number was negative, it will be picked up by an if statement prompting for a correct input from the user. If the input was correct then it will be stored in redis. As a key and value pair. With the user input number being the key and a string concatenation of the input number and resultant factorial being the stored value.

```
@app.route('/factorial/<int(signed=True):number>')
```

For the second requirement which requested a list of all stored values to be returned, the cache.keys() function of redis.py was used. This function returns a list of all keys that are stored within redis. Using a for loop, each key was iterated over, within each iteration the stored value was appended to a string that started empty. After all the keys in the redis cache were iterated over, the string containing the list is returned. The appending was completed using spaces as opposed to "\n" characters as this would save a decent amount of byte transmission when returning. If the output is piped into another program it will be equally simple to implement a reading function based around spaces as new line characters. Thus the conscious decision was made to save time and processing power.

Jmeter tests

Jmeter was used to write tests in jmx files, which were then used to test the web server. Using the jmeter GUI thread pools of the correct amount within the correct time limits were created. Within each thread pool HTTP requests were attached. Within the HTTP request a constant timer was attached, for the three timer values: 0ms, 100ms, 200ms were chosen. Within scenario two, the population of the redis cache was a requirement, for this a counter variable was attached to the http request that would increment after each request. Finally a summary report was attached as a listener to the request that would log all findings in a .csv file format. The summary report has an error <sentBytes> field that jmeter does not recognize, this was manually removed from the test file manually after each jmx file was created.

Finally, running GUI mode in vagrant was not permitted, as a result of this all tests were created outside the vagrant machine. The created jmx files were copied over to the same directory which ran vagrant. Within vagrant, jmeter was installed using the command line. Afterwards jmeter could be run in terminal mode.

Terminal Jmeter command example: sudo jmeter -n -t calcStored_100_delay.jmx

Collecting Results

One of the first problems encountered was the very high elapsed time of requests. This was due to the default cpu limit of the docker being 0.1. This was problematic for 2 reasons, the first being that it was inefficient at testing the web server as a group of tests would take minutes requiring a large amount of human supervision. The second reason was that it wasn't an accurate representation of the stress that the server would experience if it was deployed. Hence different cpu limits were tested, cpu limits below 0.7 were too slow. Because of this cpu limits of 0.8, 0.9, and 1.0 were chosen.

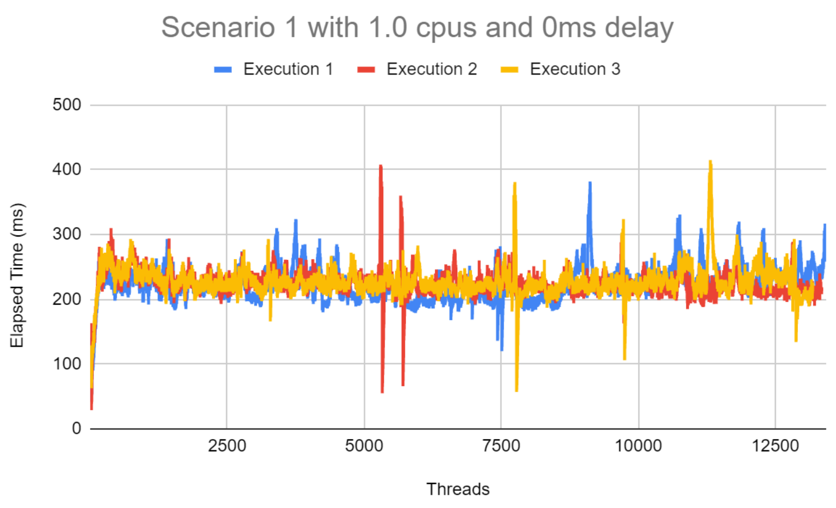
-Scenario 2 specific issues
Various methods of storing and returning the calculations were experimented with. Including lists sets and key value pairs. One encountered issue was an empty return; this had happened after swapping data storage types, clearing the cache using .flushall() seemed to have fixed this issue. Testing redis within the terminal did not have this issue, the issue wass assumedly resultant from either name collision or at that point there was a large buildup of data within the cache and the memory limit was reached, using flush all fixed this issue.

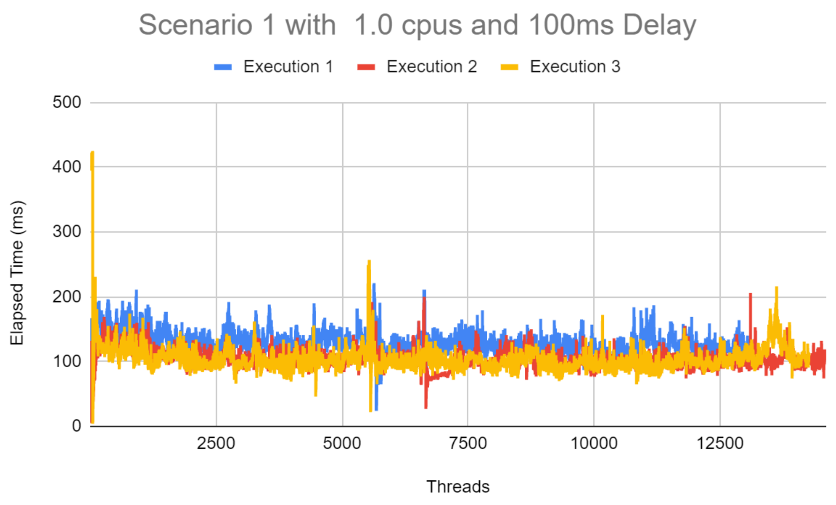
To further optimise the testing process, populating redis was made into its own jmx file. This allowed for the population to run independently from the querying. This made it so that for every one test that populated redis 9 tests that queried could be run. The only other instance that required repopulating the table was when the docker was halted when changing the cpu values.

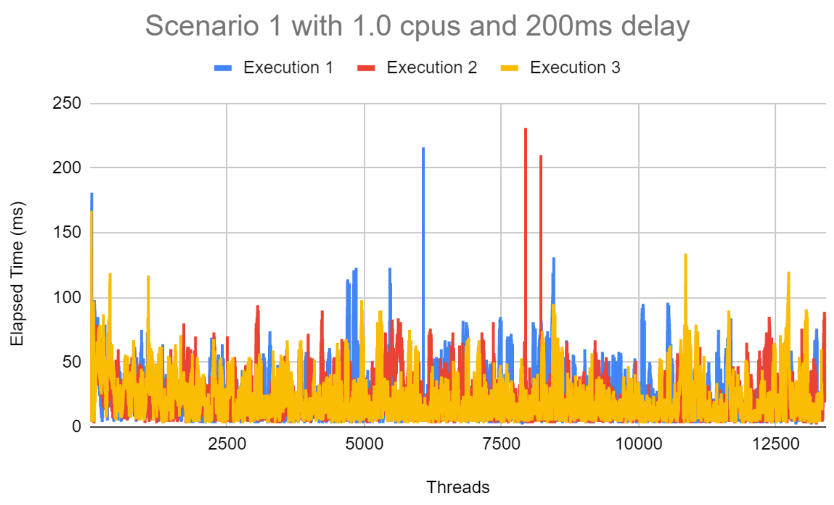
-Further memory issues
During Scenario 2, a small amount of requests would return with error code 500, after which manually using curl to query calculationsStored would not return a result. This was due to memory limit issues. To account for this various memory limits were tested at all stages. The java virtual machine heap memory was increased from the default to 4GB, the values 1GB and 2GB were tested but they still had errors. The vb memory variable in the vagrant file was increased to 8GB, this allowed for a maximum storage of 8GB. To make use of this the memory limits within docker for both web and redis were increased to 4GB each.

Findings and Discussion

Scenario 1 with 1.0 cpus

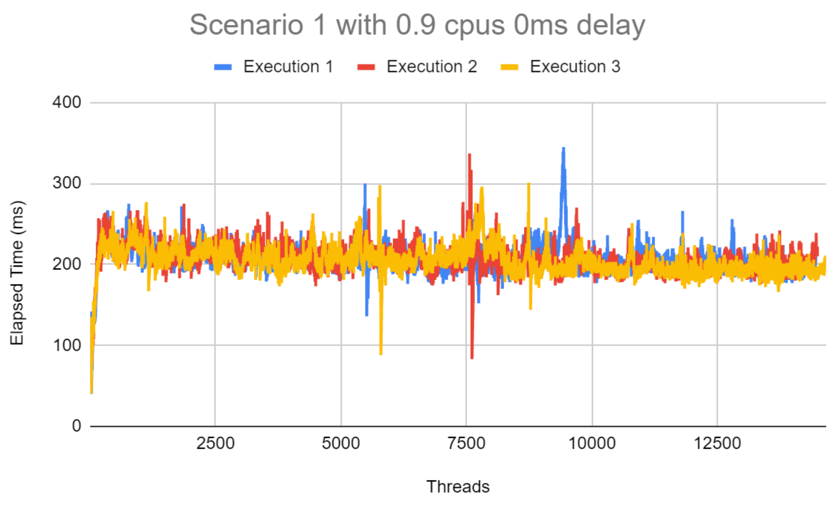


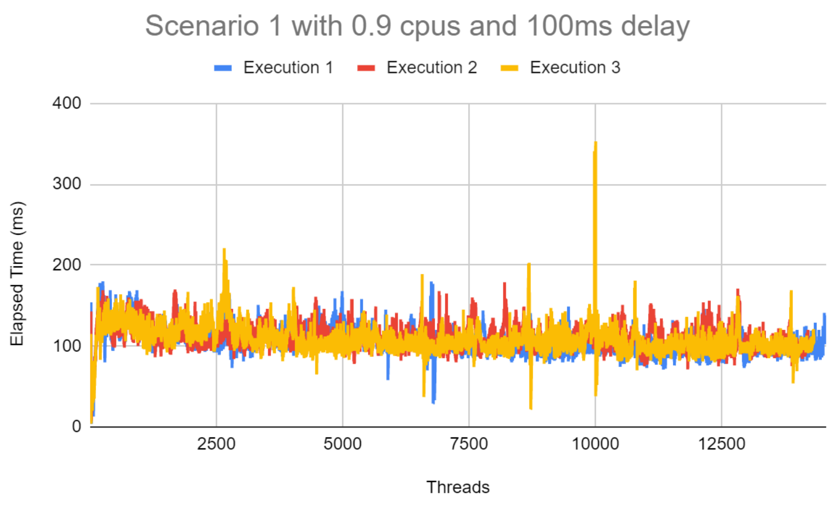


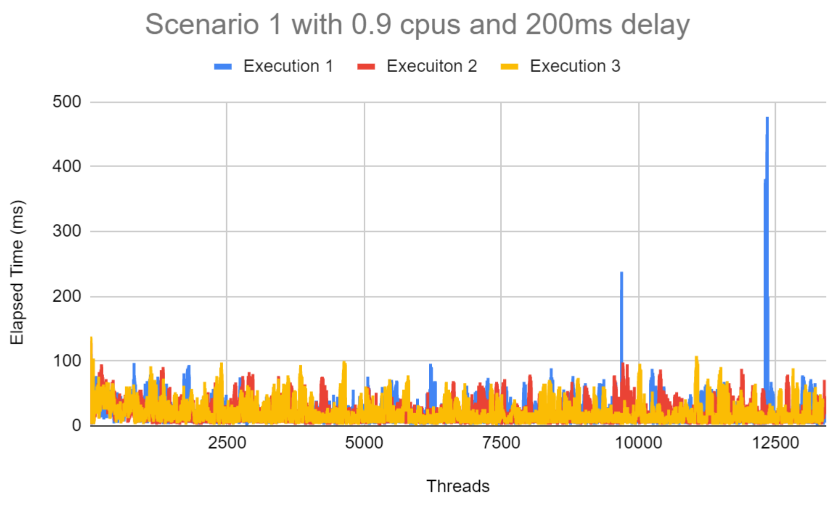


Delay	Average Elapsed Time (ms)	Total Threads	Throughput (threads/second)	Average Threads in System (L)
0ms	223.45	13428	223.80	50.0
100ms	110.64	14608	243.50	27
200ms	22.37	13419	223.65	5

Scenario 1 with 0.9 cpus

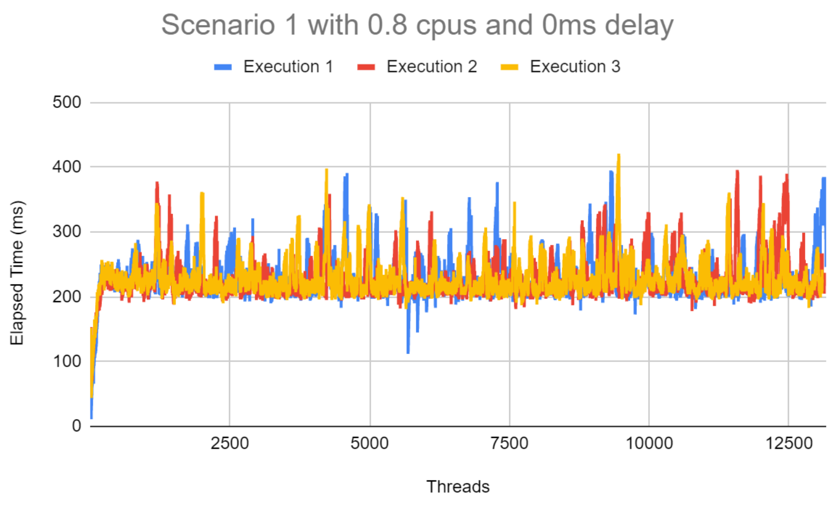


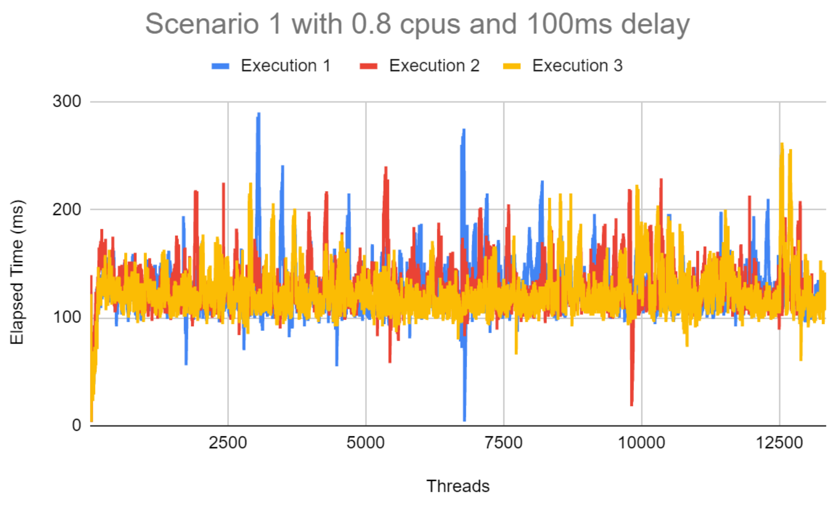


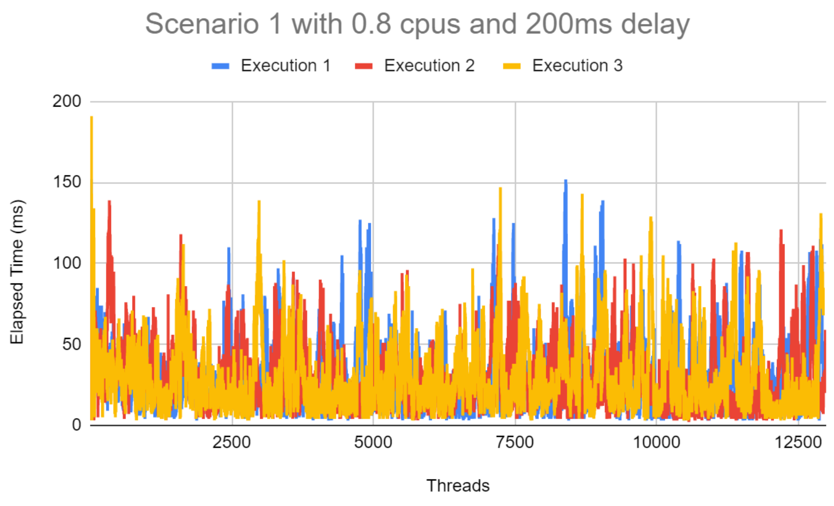


Delay	Average Elapsed Time (ms)	Total Threads	Throughput (threads/second)	Average Threads in System (L)
0ms	204.60	14671	244.52	50
100ms	106.62	14570	242.83	26
200ms	21.41	13426	223.77	5

Scenario 1 with 0.8 cpus







Delay	Average Elapsed Time (ms)	Total Threads	Throughput (threads/second)	Average Threads in System (L)
0ms	226.72	13166	219.43	50
100ms	124.66	13351	222.51	28
200ms	29.24	12996	216.60	6

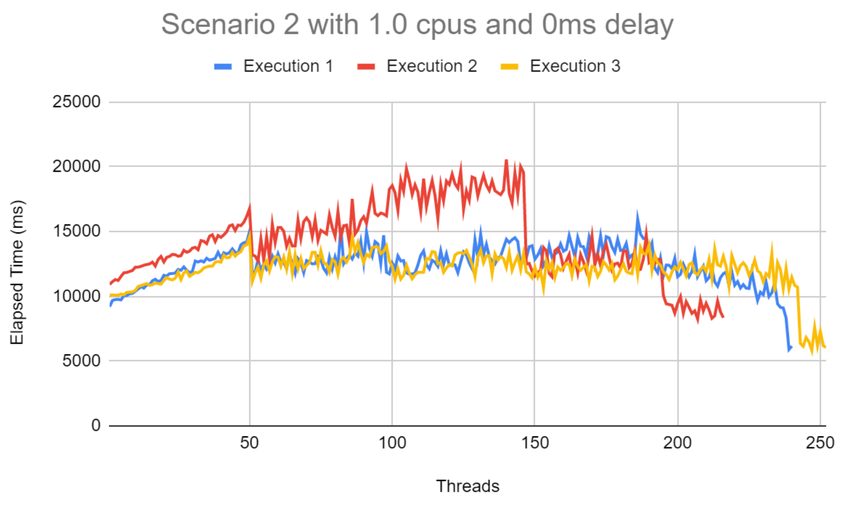
Comparison within scenario 1

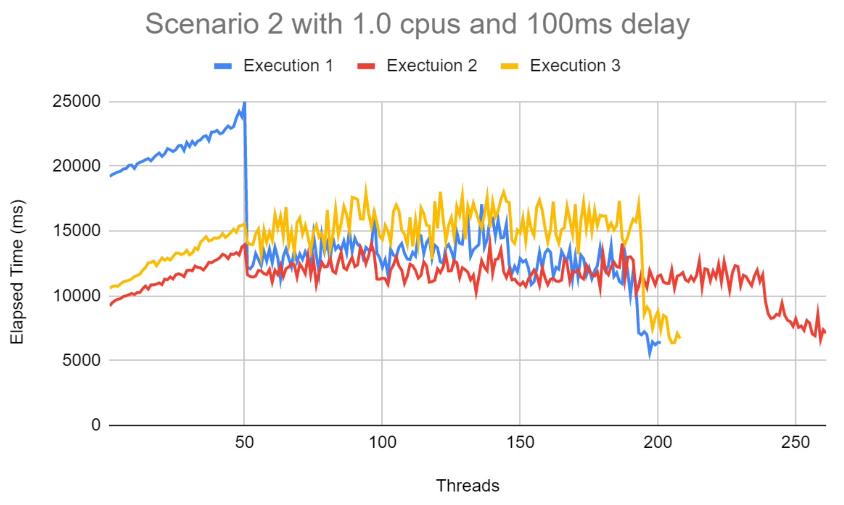
The main observed difference between the tests was the correlation between increased delay in queue wait time and the decrease in elapsed time. This means that forcing requests to wait a fixed time in queue before allowing them to be processed reduces the time required to be processed. The main reason for this would be the reduction in competition between threads for resources of the system. Doubling the delay from 100ms to 200ms seems to have reduced the average elapsed time by 5 fold.

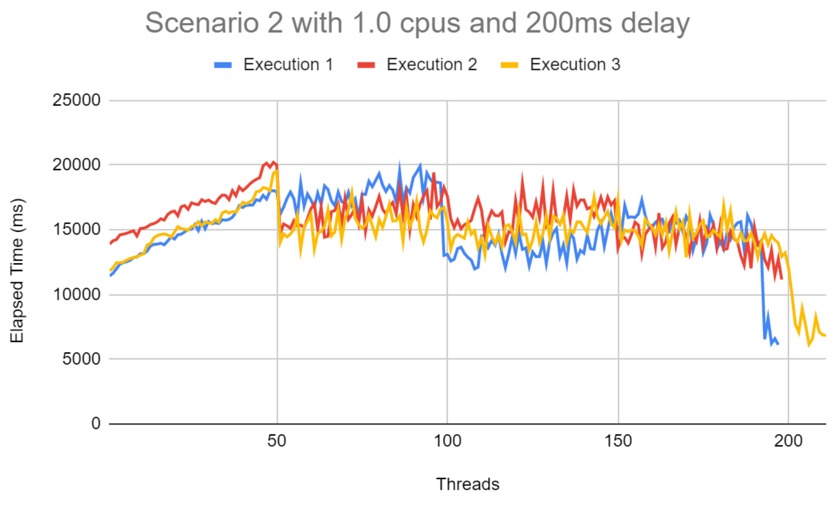
Using Little's law $L = \lambda W$ where average number of threads in the system at any point is equal to the average elapsed time multiplied by the throughput data (amount of total threads in system over run time). The results of the tests were compared. The average amount of requests being processed at any time during the execution shows that processing more requests at one time made it so that each request took longer to process. And conversely processing fewer threads made it so that each request was completed much faster.

Comparison between cpu differences within the tested levels did not show significant difference. This was due to the simplicity of the function being performed by the request.This is apparent as for approximately half of the test data the latency was exactly the same as the elapsed time. This is a significant observation as latency is the time recorded when receiving the first byte of information resulting from a request and elapsed time is the time to receive the last byte of information. These two variables being equal means the information exchange was done in a speed which makes it so that it isn't at a threshold for it to be affected when the amount of cpus changes, e.g. if the entire exchange was completed in one datagram packet exchange being able transfer 2 datagram packets at once will not affect the speed in this case.

Scenario 2 with 1.0 cpus

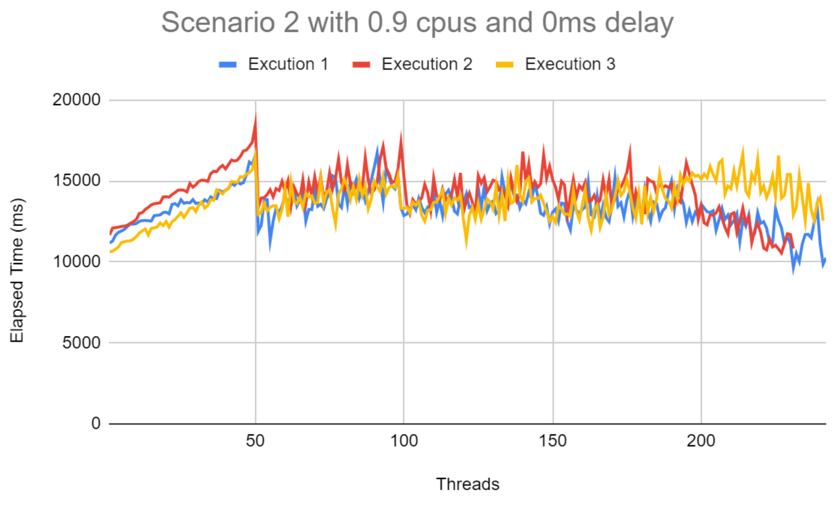


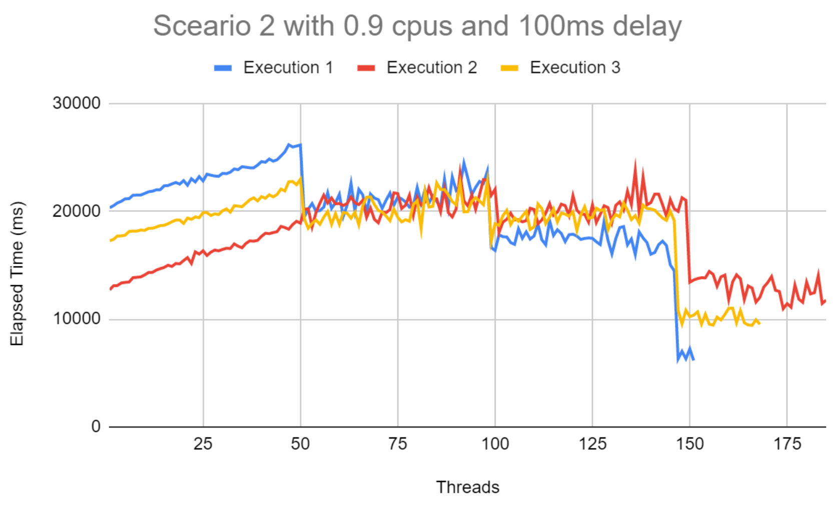


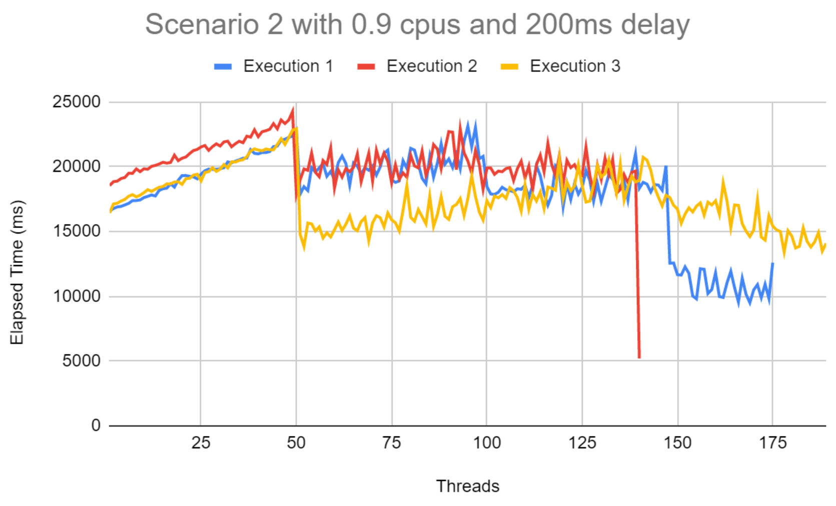


Delay	Average Elapsed Time (ms)	Total Threads	Throughput (threads/second)	Average Threads in System (L)
0ms	12934.75	236	3.93	51
100ms	13610.77	223	3.72	51
200ms	15270.15	202	3.37	51

Scenario 2 with 0.9 cpus

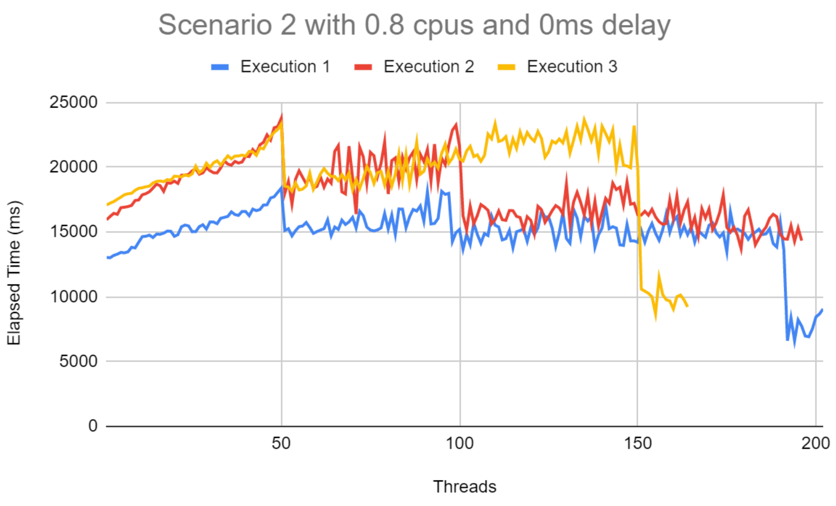


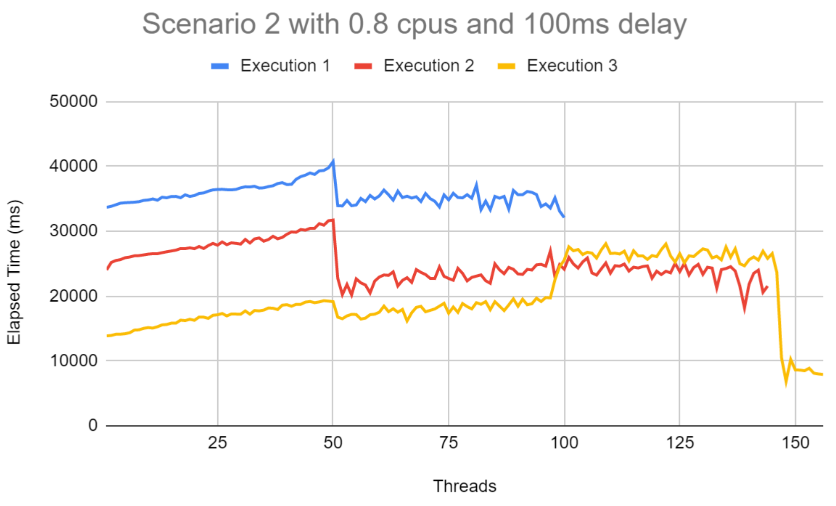


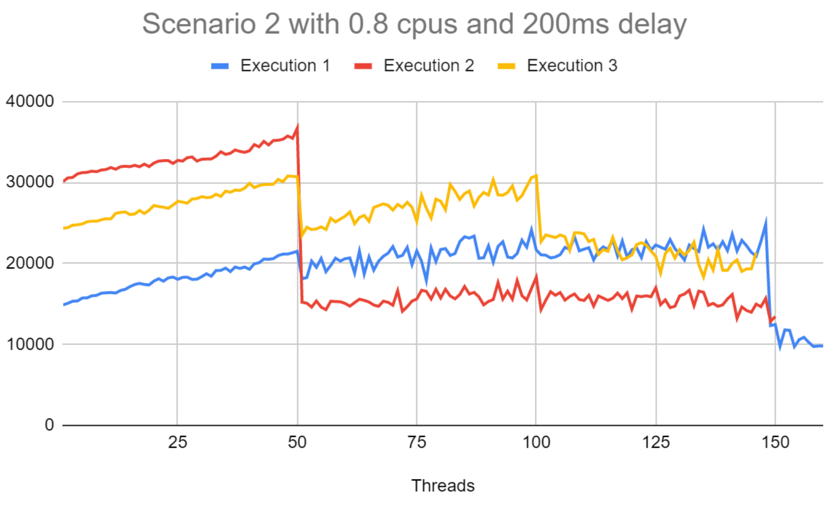


Delay	Average Elapsed Time (ms)	Total Threads	Throughput (threads/second)	Average Threads in System (L)
0ms	13782.57	238	3.97	55
100ms	18843.71	168	2.8	53
200ms	18612.60	168	2.8	52

Scenario 2 with 0.8 cpus







Delay	Average Elapsed Time (ms)	Total Threads	Throughput (threads/second)	Average Threads in System (L)
0ms	17468.14	187	3.12	55
100ms	26784.48	133	2.22	60
200ms	22135.59	152	2.53	56

Comparison within scenario 2

Within scenario 2 the average wait time per request is quite high, with the lowest elapsed time being 12934.75ms and the highest 17468.14ms with 0ms delay. Because of these large values required to fully process the data within the cpu, the effect of cpu limits is much higher than that of queue time delays. Because of this, forcing requests to wait extra time to be processed in the queue is not making a large difference. A request that is being processed at one time does not finish in time to pick up the next request that has just finished waiting.

Cpus can carry out a process called parallel processing where they can break up a serial process (one that must be run start to finish) then run the pieces at the same time and combine outputs to get the correct result while spending less time. This is achieved within the cores of a cpu. Docker has a feature which allows the modification of cores being used. By specifying a limit in the ".yml" a portion of the system's cpus (cores) can be used. As a result of experimenting with varying cpu limits it is apparent in scenario 2 that breaking up the large request into small processes and running parallel is decreased average elapsed time.

Decreasing the cpus from 1 to 0.9 increased the average elapsed time from 12934.75 to 13782.57 this is an increase of 6.6%. Further decreasing the cpus to 0.8 increased the elapsed time to 17468.14 this is an increase of 35% from using 1.0 cpus.

Average threads in system for scenario 2 were calculated to be above 50, this is an interesting finding as Jmeter was used to send threads in groups of 50. The main causes for this could be the fact that the very last request does not finish within the 60 seconds and in fact goes overtime. If we test this for the most apparent example of 0.8 cpus with a delay of 100 ms. We find that (133 /(60+26.8) * 26784.48)/1000= 41. The new calculation is the total threads divided by the potential runtime where 60 + 26.8 is the targeted runtime plus room for one extra request to finish. Using these new inputs for little's law we find that even the highest unexpected thread's value falls into the expected range.

Conclusion

It was shown that principles of little's law and queue theory are present in the scenario experiments that were carried out. Longer queue times correlated to a faster response at a lower throughput. Whereas for larger requests increasing queue time did not have as large an impact. Increasing cpu limits was shown to reduce processing time especially in larger requests where splitting up load into pieces was beneficial.