Project Software Documentation

End-To-End Development and Analysis of a Production Day Trading System

SENG 468

University of Victoria

Alwien Dippenaar
V00849850
B.Sc Computer Science
alwiend@uvic.ca

Braydon Berthelet
V00741364
B.Eng Software Engineering
braydonb@uvic.ca

Professor: Stephen W. Neville

1 Introduction

1.1 Purpose

This software document describes an in-depth specification of the BRALCA stock day trading system. The topics of discussion throughout this document will focus on the following:

- Full documentation of architecture including complete analysis of design choices.
- Full documentation of test plans, testing results, and test analyses.
- Full documentation of work effort required to build the prototype, including weekly individual log book entries signed by each member of the design team.
- Complete capacity planning and transaction time documentation, including experimental results and extrapolations.
- Full security analysis.
- Full project planning and execution documentation for prototype development effort.
- Full analysis of system capacity and capacity planning documentation.

With all these topics to be covered in this document, it should give a good general overview of the work effort and capabilities of the software system that was created.

1.2 Project Scope

The BRALCA stock day trading software system is designed to provide efficient and accurate interaction with the stock market for users to buy and sell stocks. The software system will include the accuracy of stocks provided by the stock quote server and serve as an independent broker to help minimize additional costs incurred with interacting with the market, such as quote cost and brokerage charges. The software is to support an expected minimum of 1000 concurrent users with minimal transaction processing time.

1.3 Glossary of Terms

Term	Description
AWS	Amazon Web Service
EC2	Elastic Compute Container running on AWS
TPS	Transaction per second
WS	Web Server
AS	Audit Server
DB	Database Server
TS	Transaction Server

SP	Stored procedure

1.4 Overview

This software documentation specification contains and describes all of BRALCA’s software systems features, command use cases, testing metrics, and performance results throughout the development of the prototype. Section 2 contains an overall description of the BRALCA stock day trading system. Section 3 covers the full testing details that were implemented to find ways to improve the overall system. Section 4 details the entire implementation phase of the project broken down into planning and execution phases. Section 5 and 6 covers the performance and capacity of the system that was achieved and how it was improved throughout the development lifecycle. Section 7 clarifies the software quality attributes that were implemented to create a safe and secure system.

2 Overall Description

2.1 Product Features

The features that are included in the BRALCA stock trading software are as follows:

- **Add**
  - Users are able to deposit funds into their account for the use of buying stock.
- **Quote**
  - Users are able to send a request to view how much a stock currently costs. This quoted price is valid for a 60 second timespan.
- **Buy**
  - This command will allow users to buy a given stock for an amount that is equal to the total worth of an amount of stocks.
- **Commit Buy**
  - Users are able to commit the most recent buy command executed, this will purchase the stock that the user requested to buy and place it into their account.
- **Cancel Buy**
  - The most recent buy command executed will be canceled upon execution of this command, the funds set aside for the stock purchase will be returned into the users account.
- **Sell**
  - This command will prepare to sell a given stock, the amount specified is the total worth of stocks that the user would like to sell.
- **Commit Sell**
  - The most recent sell command that was executed will have the amount of stock that was prepared for sale be placed into the users funds.
- **Cancel Sell**
  - The most recent sell command that the user executed will have the amount of stock returned to the user upon completion of this command.
- **Set Buy Amount**
  - This command will set an amount to buy a given stock when the current amount is less than or equal to the amount provided with the Set Buy Trigger command.
- **Cancel Set Buy**
  - The amount that was set for a given stock will be canceled, this command can be performed before or after the Set Buy Trigger command. The amount that was specified will be returned to the users account.
- **Set Buy Trigger**
  - This will set the trigger point to buy a stock at its current price, the trigger will check a stocks price everytime the quoted price becomes invalid.
- **Set Sell Amount**
  - This command will set an amount to sell a given stock that the user owns when the price of that stock is equal or greater than the price that is set with the Set Sell Trigger command.
- **Cancel Set Sell**

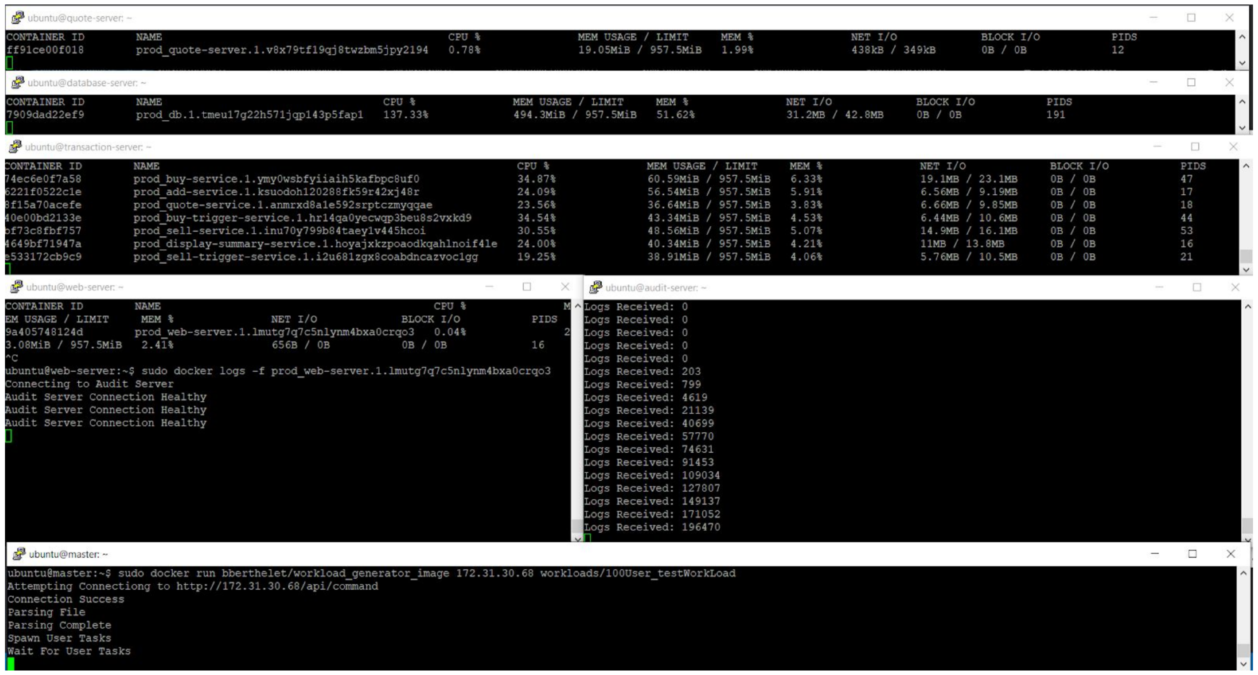
    - This command will return the users stock amount that was set for that given stock. This command can be executed successfully after either Set Sell Amount or Set Sell Trigger.
- **Set Sell Trigger**
    - This will set the trigger point to sell a stock at its current price, the trigger will check the price of the stock everytime it becomes invalid by creating a new quote request.
- **Dumplog**
    - This command will print out the given users transaction history.
- **Dumplog (Supervisor Use Only)**
    - This command may only be executed by a supervisor in order to print out the entire transaction history that has occured in the system from launch until the moment of executing the command. The results are saved to a file that is specified from the input.
- **Display Summary**
    - This command will display the users account with their funds and all their owned stock, as well as their transaction history, and all set buy/sell triggers along with their parameters.

2.2 Operating Environment

The BRALCA stock day trading system interaction is done through a web user interface (UI) which runs in an internet browser, this UI will work on any computer that has an internet connection regardless of operating system that the computer may be running. The software itself will be deployed using docker on a distributed network of linux machines.

2.3 Architecture Design and Implementation Constraints

The architecture for the stock day trading system consists of a microservice oriented transaction server, web server, audit server, MySQL database, and a redis cache. The server was designed this way to allow for individual services to be monitored separately. By splitting up the services it lent itself to easily being able to find bottlenecks.



```
CONTAINER ID        NAME                                        CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O           PIDS
ff91ce00f018        prod_quote-server.1.v8x79tf19qj8twzbm5jpy2194   0.78%               19.05MiB / 957.5MiB   1.99%               438kB / 349kB       0B / 0B             12
```

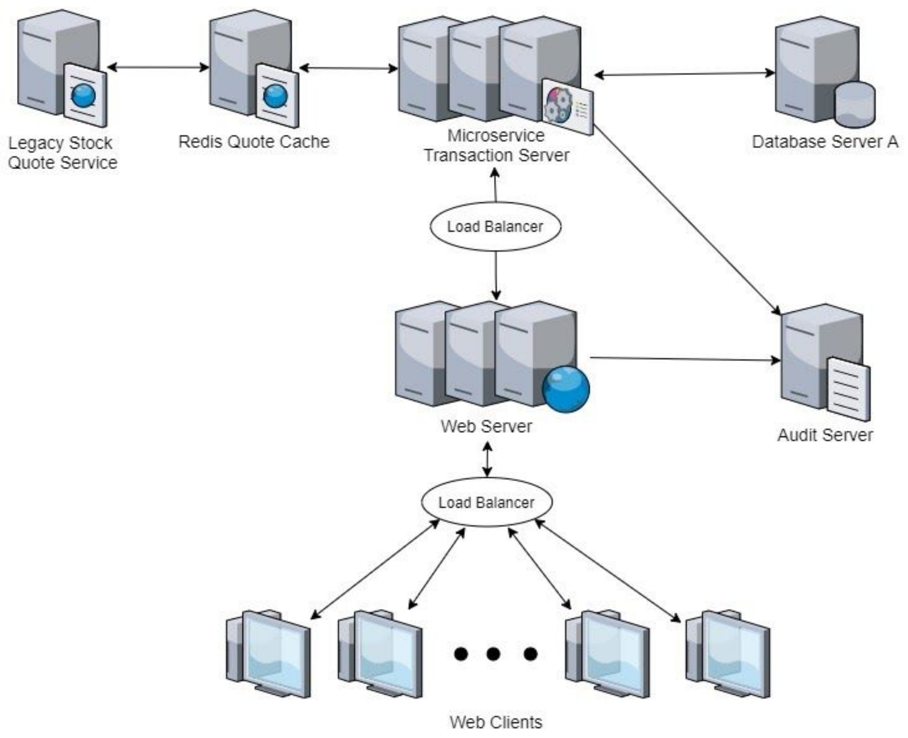
```
CONTAINER ID        NAME                                   CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O           PIDS
7909dad22ef9        prod_db.1.tmeu17g22h571jqp143p5fap1    137.33%             494.3MiB / 957.5MiB   51.62%              31.2MB / 42.8MB     0B / 0B             191
```

```
CONTAINER ID        NAME                                                  CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O           PIDS
74ec6e0f7a58        prod_buy-service.1.ymy0wsbfyiiaih5kafbpc8uf0          34.87%              60.59MiB / 957.5MiB   6.33%               19.1MB / 23.1MB     0B / 0B             47
6221f0522c1e        prod_add-service.1.ksuodoh120288fk59r42xj48r          24.09%              56.54MiB / 957.5MiB   5.91%               6.56MB / 9.19MB     0B / 0B             17
8f15a70acefe        prod_quote-service.1.anmrxd8a1e592srptczmyqqae        23.56%              36.64MiB / 957.5MiB   3.83%               6.66MB / 9.85MB     0B / 0B             18
40e00bd2133e        prod_buy-trigger-service.1.hr14qa0yecwqp3beu8s2vxkd9  34.54%              43.34MiB / 957.5MiB   4.53%               6.44MB / 10.6MB     0B / 0B             44
bf73c8fbf757        prod_sell-service.1.inu70y799b84taey1v445hcoi         30.55%              48.56MiB / 957.5MiB   5.07%               14.9MB / 16.1MB     0B / 0B             53
4649bf71947a        prod_display-summary-service.1.hoyajxkzpoaodkqahlnoif4le   24.00%              40.34MiB / 957.5MiB   4.21%               11MB / 13.8MB       0B / 0B             16
e533172cb9c9        prod_sell-trigger-service.1.i2u681zgx8coabdncazvoc1gg   19.25%              38.91MiB / 957.5MiB   4.06%               5.76MB / 10.5MB     0B / 0B             21
```

```
CONTAINER ID        NAME                                          CPU %               M
EM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O           PIDS
9a405748124d        prod_web-server.1.lmutg7q7c5nlynm4bxa0crqo3   0.04%               2
3.08MiB / 957.5MiB   2.41%               656B / 0B           0B / 0B             16
^C
ubuntu@web-server:~$ sudo docker logs -f prod_web-server.1.lmutg7q7c5nlynm4bxa0crqo3
Connecting to Audit Server
Audit Server Connection Healthy
Audit Server Connection Healthy
Audit Server Connection Healthy
```

```
Logs Received: 0
Logs Received: 0
Logs Received: 0
Logs Received: 0
Logs Received: 0
Logs Received: 203
Logs Received: 799
Logs Received: 4619
Logs Received: 21139
Logs Received: 40699
Logs Received: 57770
Logs Received: 74631
Logs Received: 91453
Logs Received: 109034
Logs Received: 127807
Logs Received: 149137
Logs Received: 171052
Logs Received: 196470
```

```
ubuntu@master:~$ sudo docker run bberthelet/workload_generator_image 172.31.30.68 workloads/100User_testWorkLoad
Attempting Connectiong to http://172.31.30.68/api/command
Connection Success
Parsing File
Parsing Complete
Spawn User Tasks
Wait For User Tasks
```



The Transaction Microservice Server is comprised of the following:

- **Add Service -** Creates or finds a user then inserts funds into their account that is stored in the database server
- **Buy Service -** Responsible for the BUY, CANCEL_BUY, and COMMIT_BUY commands. Interacts with the Quote Service in order to get correct stock quotes.
- **Buy Trigger Service -** Responsible for SET_BUY_AMOUNT, CANCEL_SET_BUY, and SET_BUY_TRIGGER commands. Once SET_BUY_TRIGGER is executed, a trigger for a given stock will be created or a users' transaction details are appended to its list of users who also have a trigger set for that stock. Will request a quote price from the Quote Service and execute the trigger for all users who's trigger amount applies for the sale.
- **Display Summary Service -** Will retrieve a users' account details from the database server and transaction history that is stored on the audit server.
- **Quote Service -** This service will get a request from either a user or another service for a given stock, it will first look for the quote in the redis quote service cache, it will be stored here if it is still valid. If not then it will request a new quote from the legacy quote server and store that result in the cache and send the price to whomever asked for it.
- **Sell Service -** Responsible for the SELL, CANCEL_SELL, and COMMIT_SELL commands. This service will request a quote from the quote service which will be used for selling stocks.
- **Sell Trigger Service -** Responsible for SET_SELL_AMOUNT, CANCEL_SET_SELL, and SET_SELL_TRIGGER. When the command SET_SELL_TRIGGER is executed, this will create or find a timer that checks the price of a stock whenever it becomes expired, for any user who's trigger price applies for a sale to happen it will commit those transactions that apply.

There were some implementation constraints to developing the overall system which included quote and transaction time to live, and buy/sell triggers are set and checked for a given stock and not for per user.

There are two different timers to be wary of in the system, whenever a quote is retrieved from the legacy quote service, that quote then has a time to live of 60 seconds from when retrieved, after this 60 seconds then the quote is considered not usable. Another timer is for buying and selling stocks, from the time a BUY or SELL command is executed, a 60 second timer starts for that transaction. In this time period the user must either cancel or commit the transaction, if the timer exceeds 60 seconds then the resources that were reserved for this sale are returned to the user. These two timers are independent of each other as if the quote is valid from the time the BUY/SELL command is executed then the transaction is valid even if in that time the quote becomes expired.

When the final command in the sequence for creating a trigger is executed, that command being either set buy/sell amount. A timer is created for the given stock and will request a new quote price once it has become invalid. If another user creates a trigger for the same stock then timer will simply add that users' transaction to the stock timer object. In the case if the trigger amount is correct to complete the transaction then they will be removed from the stock timer object. Finally, a stock timer is only valid if one or more users have set a trigger for that stock, in the case it becomes invalid there is no need to request new stock quotes anymore.

2.4 Assumptions and dependencies

Some basic assumptions that were made during the implementation of the system is that a quotes' symbol is uppercase alphabetic characters with a length of 1-3 characters. We assumed that the user could input anything into the GUI input box, thus the input must be preprocessed and only allow the user to input a certain format of commands. Checking all of the input is essential as to not allow the user to accidentally bring down the system with some characters or symbols that are invalid, this also allows some security against outsider attacks.

Another assumption made is using the 3rd party software Redis for caching in our system. The assumption is that Redis will be capable of handling the data that our system feeds it. If Redis were unable to handle the traffic then that would cause the service fees for retrieving quotes to increase as the system would now have to go directly to the legacy quote server and incur the cost associated with that action.

3 Testing Analysis

3.1 Test Plans

The testing method used for all tests was emulated testing, for this we had a working system and had a workload generator which would give commands concurrently to simulate a real world environment. The workload generator capable of sending <number of tps> was implemented to send user commands to the web server in order to both test single components of the system, or the system as a whole. Testing single components was intended to discover bottlenecks or underperforming services as this allowed us to see the total execution time and how that service dealt with any specific use case.

The test plans included in this report will include the workload files provided, as well as custom workload files that target a specific component within the transaction server and raw connection limit to the web and audit servers.

3.2 Test Environments

The tests were run in 2 different environments: a personal computer and an EC2 distributed cluster. With docker CPU % is capable of going up to vCPU*100% total, this means a single container can theoretically be above 100% CPU usage.

3.2.1 Personal Computer

OS: Windows 10 Pro
Processor: Intel Core i7-9750H CPU @ 2.60GHz
vCPUs: 12
RAM: 16GB
SSD: Yes

3.2.2 EC2(t3.micro) Distributed Cluster

OS: Ubuntu Server 18.04
Processor: Intel Xeon Platinum 8000 series
vCPUs: 2
RAM: 1GB
SSD: Yes

3.3 Testing Results

<Metrics from our testing like execution time, utilization of each container etc..>

3.3.1 Personal Computer

3.3.1.1 10 User Workload

	Threaded	Async	No Wait Logging	Quote Cache & DB SP	Streaming Services
Avg. TPS	39.117	61.585	208.611	329.261	476.191
WS Avg. Util.	10%	20%	80%	150%	110%
AS Avg. Util.	10%	15%	40%	80%	50%
DB Avg. Util.	15%	15%	35%	100%	80%
TS Avg. Util.	5%	10%	30%	40%	120%

3.3.1.1 45 User Workload

	Threaded	Async	No Wait Logging	DB Stored Procedures	Streaming Services
Avg. TPS	48.681	61.653	199.033	567.215	1,436.162
WS Avg. Util.	20%	40%	160%	200%	160%
AS Avg. Util.	15%	35%	95%	70%	115%
DB Avg. Util.	30%	50%	60%	65%	185%
TS Avg. Util.	10%	20%	70%	60%	300%

3.3.1.1 100 User Workload

	Threaded	Async	No Wait Logging	DB Stored Procedures	Streaming Services
Avg. TPS	---	---	159.253	211.795	1,770.388
WS Avg. Util.	---	---	70%	80%	180%
AS Avg. Util.	---	---	40%	40%	105%
DB Avg. Util.	---	---	40%	90%	250%
TS Avg. Util.	---	---	25%	70%	430%

3.3.2 EC2(t3.micro) Distributed Cluster

3.3.2.1 10 User Workload

	Threaded	Async	No Wait Logging	Quote Cache & DB SP	Streaming Services
Avg. TPS	46.649	62.709	147.390	158.730	845.120
WS Avg. Util.	50%	70%	80%	130%	90%
AS Avg. Util.	75%	80%	80%	120%	70%
DB Avg. Util.	20%	30%	30%	40%	80%
TS Avg. Util.	10%	20%	20%	35%	120%

3.3.2.1 45 User Workload

	Threaded	Async	No Wait Logging	DB Stored Procedures	Streaming Services
Avg. TPS	42.353	45.045	113.636	188.679	657.895
WS Avg. Util.	40%	55%	70%	120%	100%
AS Avg. Util.	50%	60%	80%	100%	110%
DB Avg. Util.	10%	10%	20%	60%	140%
TS Avg. Util.	40%	40%	70%	100%	160%

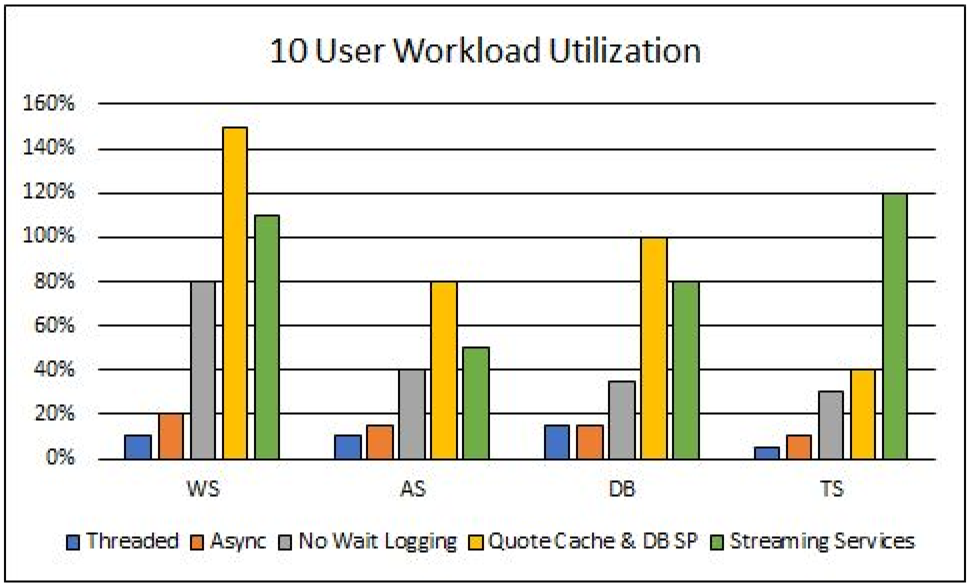
3.3.2.1 100 User Workload

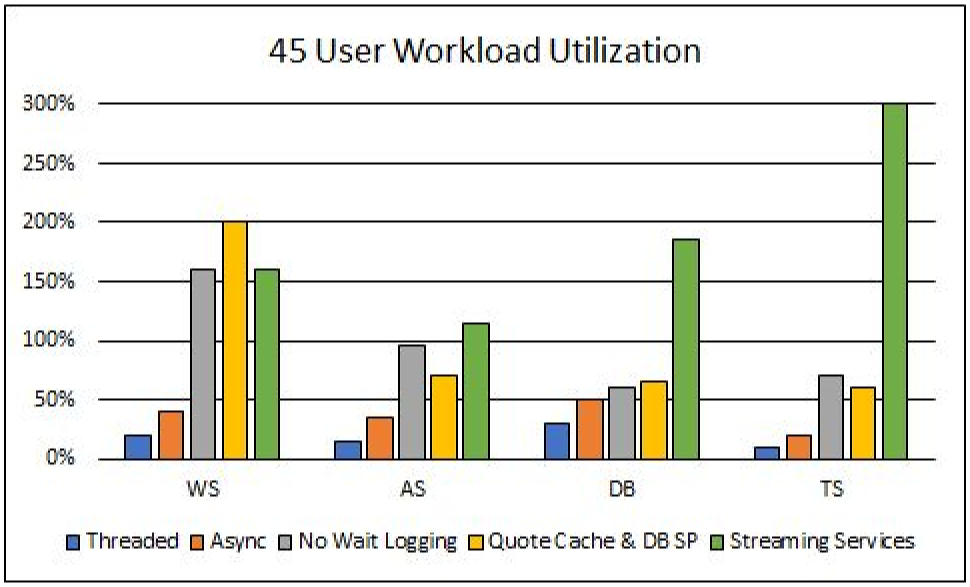
	Threaded	Async	No Wait Logging	DB Stored Procedures	Streaming Services
Avg. TPS	---	---	128.691	146.412	1374.023
WS Avg. Util.	---	---	35%	100%	110%
AS Avg. Util.	---	---	20%	90%	100%
DB Avg. Util.	---	---	20%	40%	130%
TS Avg. Util.	---	---	60%	75%	200%

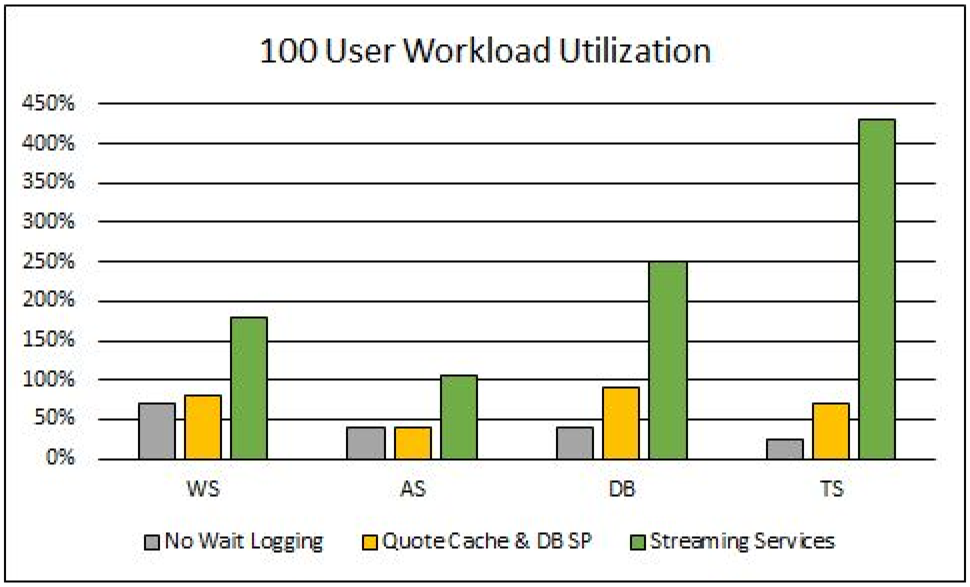
3.4 Test Analysis

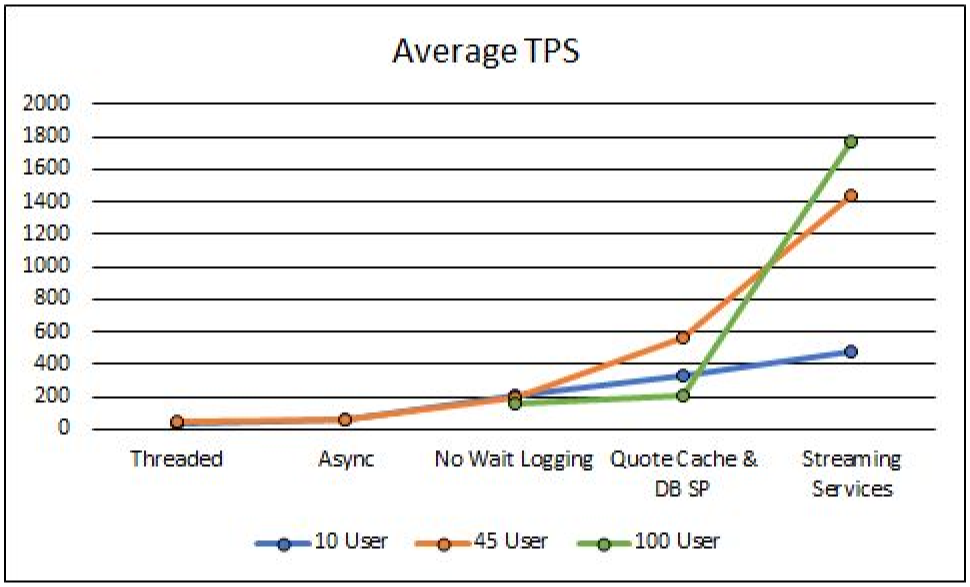
This section will include visuals of the test results found above and a short synopsis describing the results as the performance increase and capacity will be discussed in greater detail in sections 5 and 6.

3.4.1 Personal Computer



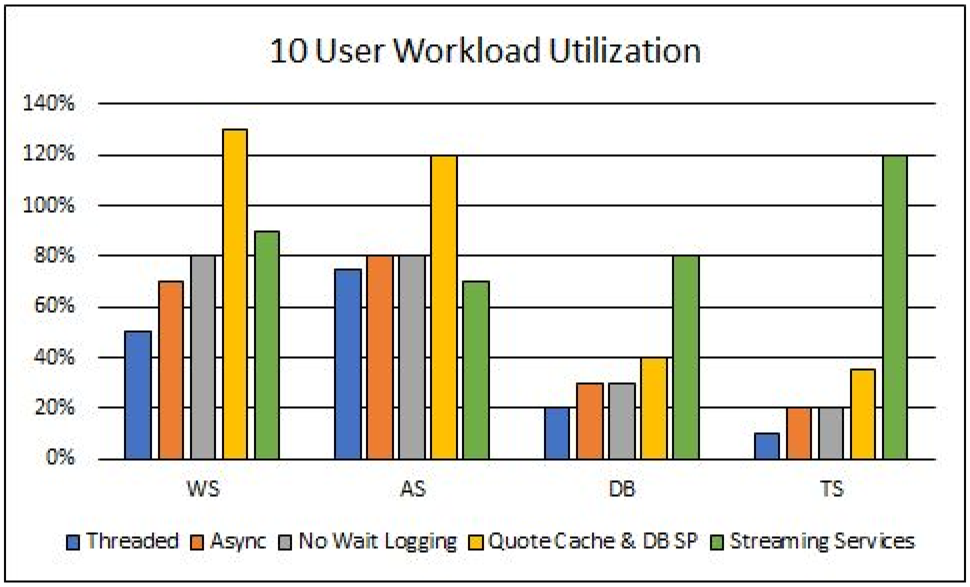


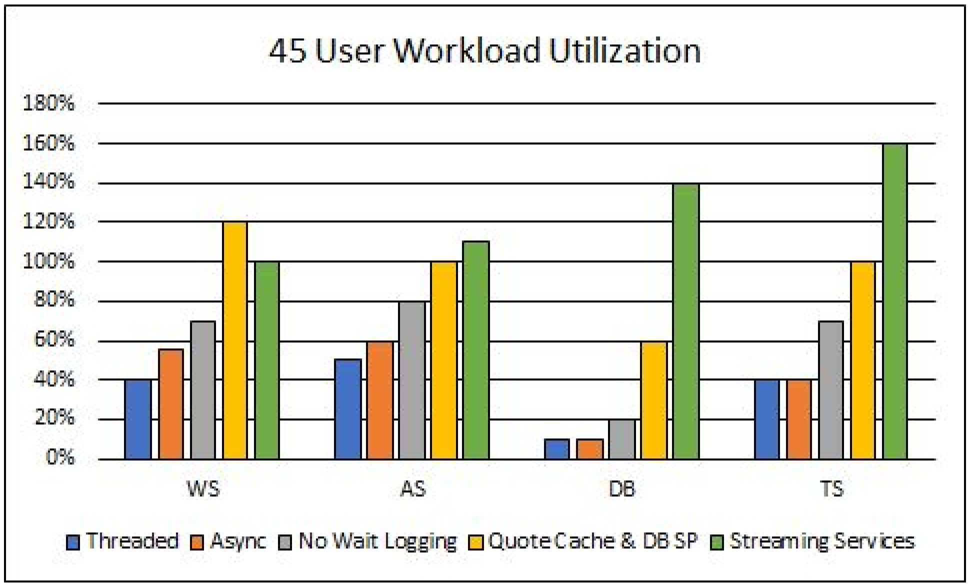


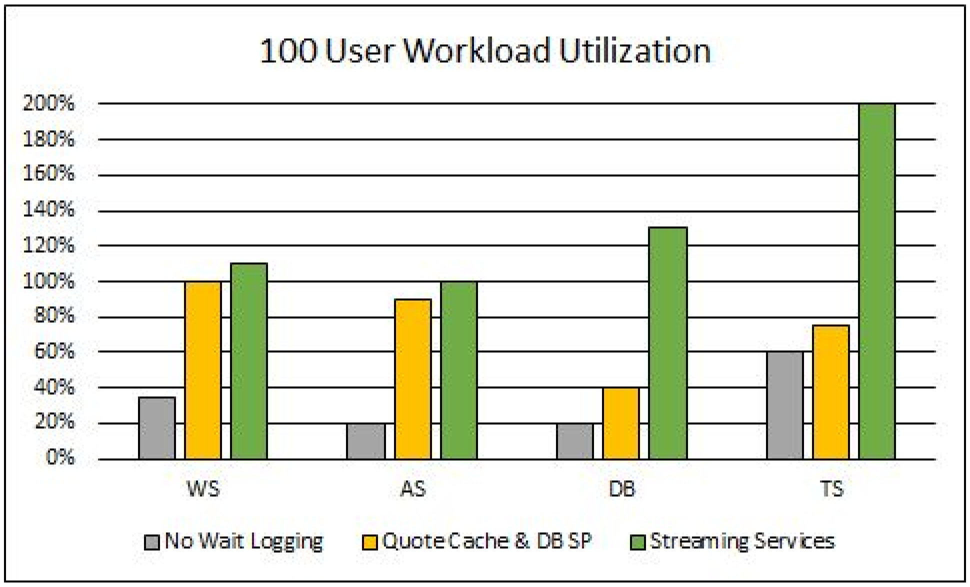


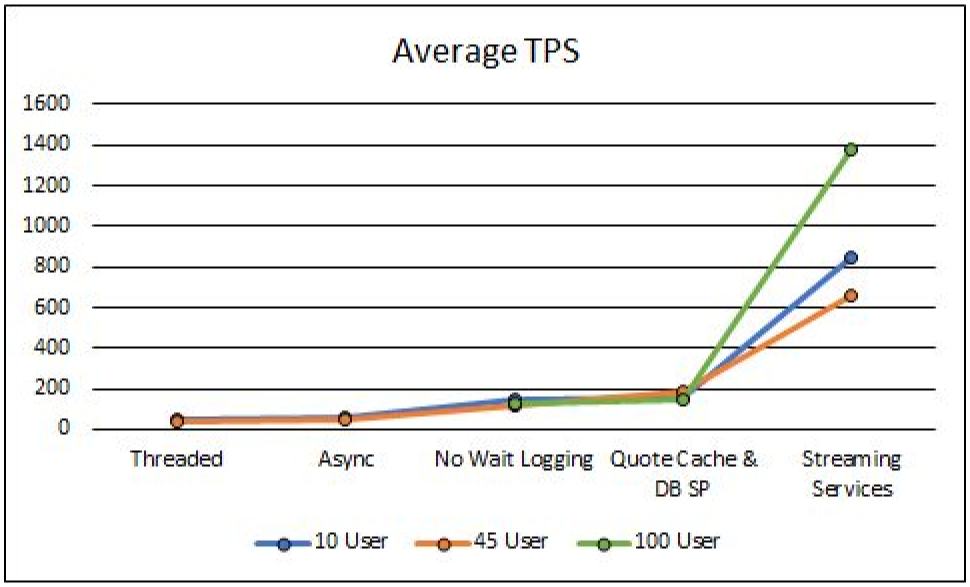
From the workload graphs above, it can be seen that the transaction server was performing minimally up until streaming services were introduced and with it being a key component in the system. It ultimately was instrumental in the large improvement of TPS from quote caching and database stored procedures to streaming services. Overall that improvement for the 3 user workloads is on average a 311.24% increase in the TPS.

3.4.2 EC2(t3.micro) Distributed Cluster





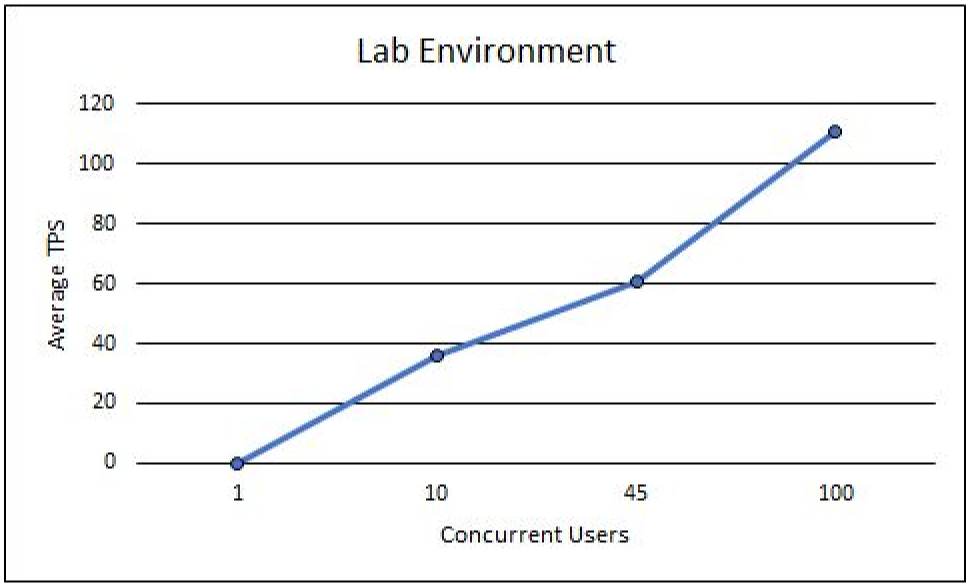




It is evident that the database server and transaction server are both under utilized and are a bottleneck in the system, it is only until streaming services are introduced to improve their utilization and speed up the system. This speed up is also extremely evident in the average TPS of the system from quote caching and database stored procedures to streaming services. The overall average TPS increase for this jump in all 3 user workloads is 506.52%.

3.4.3 Lab Environment

Users	Quotes	Cost	TPS	Date
1	32	1.6	0.000	2020-02-11 15:09:36
10	3175	158.75	35.948	2020-02-11 15:19:04
45	3160	158	54.248	2020-02-13 18:06:31
45	3160	158	60.617	2020-02-13 18:29:46
100	33121	1656.05	111.048	2020-02-27 17:56:35



As we were unable to test in the lab environment without the streaming services upgrade due to the coronavirus. We came to the conclusion that the EC2 distributed cluster behaved in a similar and realistic manner as the lab environment does. This means that the large increase in transactions per second will be similar in the lab.

4 Prototype Development

4.1 Planning

The development for the system started to create a strong foundation that we were able to build upon and create features that did not involve rewriting large portions of code. This was to create a basic but functioning web server, audit server, complete microservice transaction server, database, and workload generator. Each microservice extended a base service that handled all communication with the web server as well as an audit writer that handled all communication to and from the audit server. This first iteration was multi-threaded.

There are five main improvements that are to be implemented into the server

- Asynchronous communication between all servers
- Remove any delays associated with logging
- Reduce database connections
- Quote server cache
- Streaming communication between all servers

4.2 Execution

Each improvement was implemented over a one to two week period with the final improvement ongoing.

Asynchronous communication was developed by utilizing C# async/await feature. The workload generator was the first area that had asynchronous communication, then the web server. The web server was individually stress tested to validate the performance gain. Then async was implemented on the audit server and writer. The last stage of this improvement was implementing async on the services within the transaction server.

The web server and transaction microservices were being bottlenecked by the audit writer as it would wait for logs to successfully write. There were different attempts taken to reduce this and in the end the audit writer would handle logs without needing the connected server to wait for a response, this reduced load on all servers.

The next big change was reducing connections to the database server. Most services required multiple connections to the database to perform one command, this resulted in poor performance due to io operations to the database and waiting on free connections. To reduce connections it was decided to create stored procedures to handle all complex operations to the database. This reduced each service to a single connection per command. The side effect was the database was now under more stress.

The last major bottleneck that was addressed was server connections and a quote server cache. The cache was handled on the quote service; all services that required a quote would now request one from the quote service instead of the quote server directly. The first streaming connection that was setup was for the audit server. Communication data was changed from XML to MessagePacks capable of asynchronous operations. Next all other server connections also implemented async MessagePack communication and all user connected services were cached on the web server to allow it to be reused. This last change drastically improved performance allowing for servers to be fully utilized.

4.3 Work Effort

As the software system was completed with a group with two members, it is trivial to see who completed which portions of the project with some components being worked on in unison.

4.4 Weekly Logbook Entries

First Name	Last Name	Activity	Time Spent	Submit Time
Alwien	Dippenaar	sql transactions for triggers	1.500	2020-03-20 18:35:27
Alwien	Dippenaar	documentation	3.000	2020-03-20 18:35:10
Braydon	Berthelet	debugging	15.000	2020-03-13 21:03:45
Alwien	Dippenaar	Distribute system	3.500	2020-03-10 17:54:48
Braydon	Berthelet	Distribute system	3.500	2020-03-10 17:50:00
Alwien	Dippenaar	database optimizations	8.000	2020-03-10 14:38:09
Braydon	Berthelet	Database optimizations	15.000	2020-03-10 14:37:13
Braydon	Berthelet	No work week of March 2	0.000	2020-03-10 14:36:41
Braydon	Berthelet	Full async week of Feb 24	20.000	2020-03-10 14:36:16
Braydon	Berthelet	Sick week of Feb 17	0.000	2020-03-10 14:35:31
Alwien	Dippenaar	Implement caching	9.000	2020-02-21 19:52:47
Braydon	Berthelet	async workload generator and web server	10.000	2020-02-13 18:08:45
Alwien	Dippenaar	Study for another class midterm	0.000	2020-02-13 18:08:27
Braydon	Berthelet	Merge conflicts, bug fixes, buy trigger, db schema fixes, mock quote server result change	11.000	2020-02-07 21:35:58
Alwien	Dippenaar	testing/debugging	3.000	2020-02-07 21:35:41
Alwien	Dippenaar	finished buy command suite	1.000	2020-02-07 21:35:12

Alwien	Dippenaar	implemented basic display service	2.000	2020-02-07 21:34:49
Alwien	Dippenaar	implemented sell command suite	5.000	2020-02-07 21:34:24
Braydon	Berthelet	Workload Generator	4.000	2020-02-02 23:24:14
Braydon	Berthelet	Finished Audit Server conform	3.000	2020-02-02 23:23:43
Braydon	Berthelet	Audit server conform	5.000	2020-02-01 22:15:40
Braydon	Berthelet	Audit server and schema conforming	12.000	2020-01-31 19:38:37
Alwien	Dippenaar	meeting	2.000	2020-01-31 19:38:53
Alwien	Dippenaar	Code Buy, buy commit, and buy cancel commands	10.000	2020-01-31 19:38:38
Braydon	Berthelet	meetings	2.000	2020-01-31 19:37:19
Braydon	Berthelet	Mock quote server, quote service, web server, docker setup, database, github setup	15.000	2020-01-28 15:14:50

5 System Performance

To increase the performance of the overall system, as stated in section 4, five main improvements were:

- Asynchronous communication between all servers
- Remove any delays associated with logging
- Reduce database connections
- Quote server cache
- Streaming communication between all servers

All changes to the system did improve the overall performance of the system, however some improvements lead to much larger jumps in the performance. The measure of performance used to analyze the system is transactions per second and the cpu utilization.

5.1 Experimental Results

In the early stages of testing the system, it was clear that the main bottlenecks in the system were the transaction server and the database server. Some additional testing was conducted on the individual microservices that made up the transaction server to find the bottlenecks within. The largest improvement with the microservices was converting the backend logic into database stored procedures as in the first iteration of the project the backend naively interacted with the database opening and closing connections multiple times per one command executed. As seen from the testing results in section 3, the jump from threaded to quote cache & database stored procedures easily doubles the utilization with this upgrade. Not only did this help improve the transaction server but the database server did not have to deal with the many requests it received from many processes seeking simple 1 result queries. This however puts more strain on the database server when heavier loads are processed. The initial distributed system had the database server on the same system as the transaction server for locality but was needed to be shifted to its own system after the streaming services upgrade.

Removing any delays for actions associated with logging to the audit was an improvement that increased the performance of all the servers with those mainly being the web and audit server. The web and transaction servers would be waiting for the audit server to successfully write logs, this wait time slowed down the system in a cascading manner as the system may log multiple times per command executed. This would add up and eventually create timeouts, however the logs would still successfully write and the system would finish executing the workloads. But because many services and servers would wait it reduced the potential throughput and load that could be handled.

Caching quotes with an in-memory cache not only increased performance but also saved the company money in the cost incurred from requesting a quote from the legacy quote server. Before caching the transaction services would receive a fresh quote price every time the system needed to know the price, but with the price came the wait time for internet latency. This wait time would build up and if the average wait time was 0.01 second for example the 100 user workload requested a quote 33121 times. This would add an unnecessary 5.5 second wait time for running that workload. Additionally, when a quote is requested the system checks to see if that quote is currently being requested which avoids race conditions of multiple quotes being requested at the same time.

Finally, the improvement which provided exponential increases in performance is introducing streaming services. The utilization of servers that relied heavily on communicating with multiple other servers saw a large spike, that mainly being the transaction server. The transaction server communicates with every other server, and most of the time multiple per command that was executed. As seen from the test results in section 3, the utilization increased on average 264% and TPS increased on average 408.5% for both instances of testing done on the personal computer and EC2 distributed cluster.

There are still several improvements that can be added to the system to increase the throughput. The system is currently distributed but it is not replicated. A load balancer can be set up to allow for services to be spun up in parallel. Before being able to set up load balancing the caching strategy would need to be changed to a distributed system. This would reduce overall load on single components allowing a higher throughput. Additionally, a distributed cache could be set up for recent data returned from the database.

5.2 Experimental Extrapolations

The latest performance log submitted for validation that was run in the labs using the no-wait-logging code. It can be seen that the performance of the t3.micro EC2 cluster closely matches the performance in the lab. The cluster performed slightly better but this is most likely due to running a local quote server.

Using the linear extrapolation equation: $y(x) = y(1) + \frac{x - x(1)}{x(2) - x(1)} * [y(2) - y(1)]$, we can predict the performance of the 1000 user workload using the results from the 45 and 100 user TPS with all upgrades implemented.

$y(1000) = 1374.023 + \frac{1000 - 100}{45 - 100} * [647.895 - 1374.023] = 13,256$ TPS theoretically. This value may not be as accurate as the data is not linear and machine resources will come into play which will play a significant effect on the performance.

6 System Capacity

6.1 Capacity Planning Analysis

At the start of the project, the guideline for the project was to create a system that must be capable of handling a minimum of 1000 concurrent users. To give users the feeling of instantaneous response, transactions must be executed and completed within 0.1 seconds from submission. Studies have shown that 1 second response time keeps a users' flow of thought to be seamless, and any time between 1-10 seconds keeps the users' attention but feels to be at the mercy of the computer. And after 10 seconds waiting for a webpage to respond, their attention goes elsewhere and then potentially will lose a customer forever. It is for this reason to keep response time of the system to be minimal without exceeding 1 second. To be able to maintain a response time of 0.1 to 1 second for the 100 workload which requires handling 100,000 commands will require the system to maintain a minimum of 1,000 transactions per second.

6.2 System Capacity Analysis

6.2.1 Personal Computer TPS

	10 User TPS	45 User TPS	100 User TPS
Threaded	39.117	48.681	---
Async	61.585	61.653	---
No Wait Logging	208.611	199.033	159.253
Quote Cache & DB SP	329.261	567.215	211.795
Streaming Services	476.191	1436.162	1770.388

The peak throughput of the system occurred when handling 100 concurrent users, the transactions per second are within the range of response time needed to sustain a system that offers a seamless user experience.

6.2.2 EC2(t3.micro) Distributed Cluster

	10 User TPS	45 User TPS	100 User TPS
Threaded	46.649	42.353	--
Async	62.709	45.045	--
No Wait Logging	147.39	113.636	128.412

Quote Cache & DB SP	158.73	188.679	146.412
Streaming Services	845.12	657.895	1374.023

6.2.3 Lab Environment Pre Streaming Services

Users	Quotes	Cost	TPS	Date
1	32	1.6	0.000	2020-02-11 15:09:36
10	3175	158.75	35.948	2020-02-11 15:19:04
45	3160	158	54.248	2020-02-13 18:06:31
45	3160	158	60.617	2020-02-13 18:29:46
100	33121	1656.05	111.048	2020-02-27 17:56:35

The EC2 distributed cluster and lab demonstrated similar results within reason, and demonstrated acceptable response times after the upgrades were implemented. Therefore we can conclude that the results would be similar to that of the EC2 distributed cluster and the capacity of the system would be acceptable.

7 Quality Assurance Analysis

7.1 Safety Requirements

The possible damage that could arise with a stock trading system has the potential to impact a users' life and their financial security. Any discrepancies with managing their account funds or inaccuracy of stock transactions can negatively impact both the user and BRALCA such as a loss of funds or a lawsuit for mismanaging a users account information. In order to correctly implement the system some policies had to be put into place. In order to guarantee accuracy of stock interactions, the price of the stock that has been quoted for a buy or sell action is valid for a total of 60 seconds. After a buy or sell action has been executed the user has 60 seconds to either commit or cancel the action, after this 60 seconds has elapsed the transaction will become expired and the users funds or stock will be returned to their account for use in future transactions. When a user performs an action that either interacts with their fund or stock balance, an amount equivalent to their request must be reserved from their account so as to not let the user complete a set of actions they do not possess the correct amount of funds or stocks to execute.

7.2 Security Requirements

Having the system be secure is vital as any system that deals with finances must be secure in order to not have any malicious actors steal money. The input that is retrieved from users is forced to be in a specific format and then the contents of items in that string are checked for any invalid input, this is otherwise known as sanitizing input. Sanitizing input is a good first contact to battle against cross-site scripting and SQL injection attacks. Users are unable to interact with an account other than their own to combat users trying to manipulate the system and gain another users funds.

7.3 Software Quality Attributes

Since the software system uses a microtransaction architecture for its back-end transaction server, if one of the services happens to go down then it does not affect the other components of the transaction server as they are independent from one another. With this in mind then simply creating another instance of a service or load balancing could help the system to achieve greater performance overall.

The simple web GUI allowed users to interact with the system given they are knowledgeable of the proper order of events needed to successfully buy and sell stocks. As all input and output is seen on a single web page, finding details on quotes or account information requires only inputting the command and a single mouse click, effectively speeding up the workflow of the user.