Rory Smith V00891516
Oliver Lewis V00877996
Janhavi Dudhat V00870135

Group 3 - Pygang
CSC468 Project Documentation

January 31st, 2021

Requirements

Functional Requirements

- The system allows users to
    - View their account.
    - Add money to their account.
    - Get a stock quote.
    - Buy a number of shares in a stock.
    - Sell a number of shares in a stock they own.
    - Set an automated buy or sell point for a stock.
    - Review their complete list of transactions.
    - Cancel a specified transaction prior to it being committed.
    - Commit a transaction.
- The system provides a complete transaction history for a specific user, including cryptokeys supplied by the quote server on a stock quote..
- The system provides a complete history of every system transaction for all users.
- Stocks are bought and sold in terms of the nearest whole number of shares that can be bought for the given amount of money specified.
- The system does not update account balances until the user commits the transaction.
- Users have one minute to commit a transaction before the stock price is voided.
- Once a commit has timed out and the stock price is voided, the new stock price is displayed to the user for them to commit.
- An automated buy or sell point for a stock will not result in negative account balances when triggered.
- The user is notified when commands successfully execute.
- The user is given an error message when a command fails to execute.
- If a commit takes more than 15 seconds then the corresponding buy or sell command fails, and the user must submit the buy or sell command again.

Performance Requirements

- Every transaction within the system is logged with a record of the transaction, timestamps of the transaction, processing time, and all account state changes within the system.
- A commit takes no longer than 15 seconds to confirm the transaction and be updated within the system.
- The system handles the final workload without faults or errors.
- Error messages sent to the users do not reveal the underlying technologies or architecture.
- User commands are sanitized before being processed.

Design Constraints

- A web application that provides users with a UI for interfacing with the system's backend.
- Docker Container technologies are used.
- The system interfaces with the Legacy Coder Server to receive stock quotes.
- The system's architecture is distributed.
- The server code is run on the Linux Lab B203 machines.
- The system makes all logs available as an ASCII text file.

Development Platform

- Linux KVM running Ubuntu 18.04

Architecture

Allowing for scalability was the main focus for many of the decisions regarding architecture. We used the provided example architecture as a starting point and made modifications that allowed it to be scaled. As seen in Figure 1. The core idea of having separate containers handle HTTP requests from the clients and a backend transaction server handle the commands is still there.

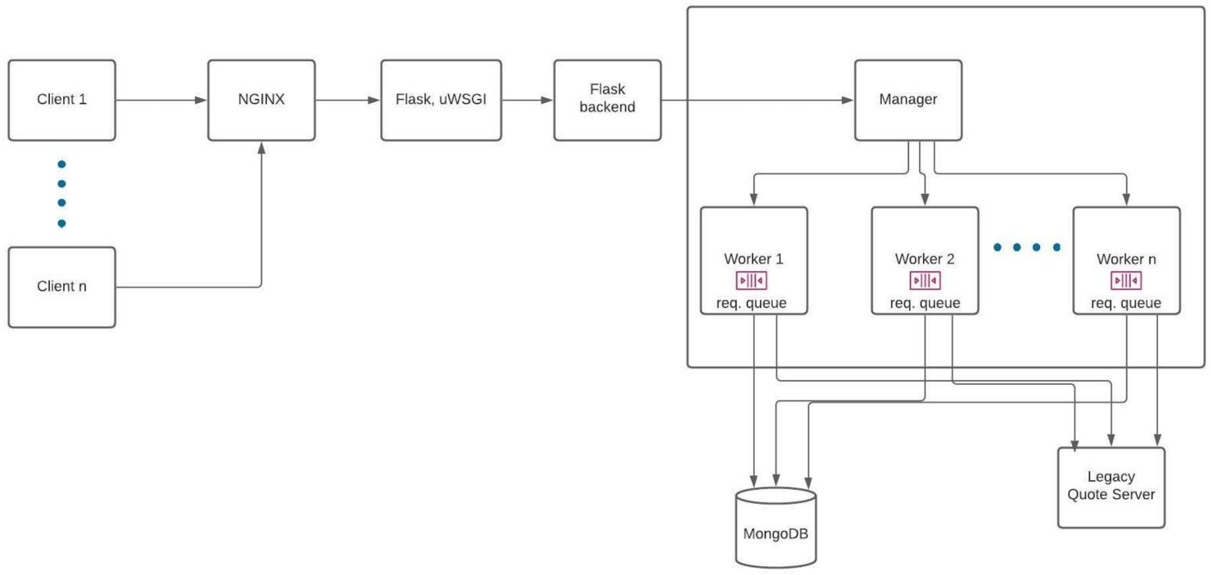

Figure 1. Project Schema

Added to the frontend of this design is NGINX which acts as a reverse server proxy and load balancer. This is a standard design decision for many servers and allows for great scalability if upgraded to the premium version. For our uses however, the free version provides sufficient scalability as it's unlikely the system will need more than one container to handle HTTP requests. uWSGI was chosen to manage the flask application as it is widely used, especially when paired with NGINX. Since Flask is a very lightweight application framework these managing frameworks are needed to provide any level of scalability and security for the front end.

Interacting with the database and legacy quote server takes time and the requirement that each user's commands are executed synchronously remove many possibilities for concurrency within one process without complex data management for many users. To solve this scaling issue, multiple containers which can independently handle requests can be spun up as needed by the manager container. The manager handles load balancing, docker handling, and reading of the workload files. Frameworks such as Kubernetes could replace the custom manager container but configuring Kubernetes for our specific use case is thought to be more difficult than building a custom load balancer from scratch. Each worker container can access the Legacy Server and the MongoDB database at any time and it's up to each to handle this concurrency. A Redis cache may be added to replace some functionality the database would handle if scaling becomes a problem for it.

Tools and Libraries

Pygang has considered many various frameworks for implementing the first iteration of the day trading software. There are many different ways to implement each part of the software with many of the frameworks/libraries sharing similar pros and cons. This led to some of the choices coming down to reputation among the communities using them and the experience each team member had with them.

Docker

Docker was specified as a requirement from the client but would have been chosen regardless. Grouping frameworks and components into containers allows for scalability and reduces complexity of the software overall. Managing the initialization of the system is handled by docker-compose with worker containers handled by a python3 script using the Docker python library. Kubernetes was heavenly considered since its far more powerful than pure docker-compose but its heavy learning curve due to its numerous features and its use only in large professional applications made it out of the scope of the project.

Frontend

The client facing front end of any website software has possibly the most oversaturated software selection of any development area. With the main focus of the software being to provide the best possible scalability and reduce the latency of each command sent to the backend with little need for a highly dynamic frontend, a simple static framework was all that was needed to handle client requests. For this reason, Flask was chosen to handle the requests of each client with uWSGI handling HTTP and vertical scaling of each flask application. NGINX was chosen as a reverse server proxy and load balancer for its common use among many popular websites and its easy setup. Including uWSGI and NGINX for handling HTTP requests provide more power and scalability than the project requires as all load is sent from workload files but their simple setup and small overhead allow for easy scaling in the future should the need arrive.

Backend

The backend of the project is where scalability is mostly needed. Since there is little processing needed for each incoming command and the bottleneck of the system being waiting for the database and legacy quote server, python3 being run inside a docker container was chosen to handle the bulk of the application. If there was a need for extensive processing, a different language could have been chosen, but even if the demand was increased, a binding to a C module could be implemented which would most likely handle this.

Each python script will make heavy use of both threading and asyncio to reduce the time wasted waiting for the legacy server and database.

Database

This is a medium sized project that requires the system to be highly scalable. Between performance and latency, high performance is the priority. Along with this, the primary key (i.e. userid) is known for the data that will be stored within the system. Our access pattern is also defined since the number of user commands are provided to us already and are limited. A NoSQL database would be the right choice for such a system, especially keeping in mind the scalability and performance aspect of the required system. MongoDB is a NoSQL database that can provide us with fast and cheap reads and it also supports scalability. Additionally, it is easy to use; it can be replicated for contingency purposes and to provide high system availability. These make MongoDB an appropriate choice for the required system.

MongoDB will be run as a local instance since that would deliver faster results than on cloud and it will be run in it’s own container.

Project Plan

The following table outlines critical tasks required to have end-to-end functionality of the system. Therefore, all tasks must be completed before February 7th in order to pass the single user workload file (see Table 2 below for all milestones).

Table 1: Preliminary project tasks and assignment for single user workload milestone

Tasks and Subtasks		Team Member(s)
Setting up the MongoDB container		Oliver
Creating the worker container	Needs to handle incoming commands	Oliver
	Needs to interface with the quote server	Janhavi
	Needs to interface with the MongoDB instance	Rory
Creating the manager container	Creating a module for handling the workload files	Rory, Oliver, Janhavi
Creating the module to handle logging		Oliver
Front-end	Creating a user interface for performing day trading operations	Janhavi
	Setup NGINX and uWSGI in their containers	Rory
Performance verification	Testing and analysing the performance of the system	Rory, Janhavi, Oliver

The specifics on what needs to be done further along in the project will be dependent upon the performance of the system as the number of users increases. This may include building redundancy into the database, implementing a caching system, load balancing, and message queues. All team members will take part in the testing process as well as writing documentation.

Milestone Deliverables

The following table outlines the due dates of key project deliverables.

Table 2: Project Milestones

Date	Deliverable
February 7th, 2021	● Verified execution of 1 user workload file
February 14th, 2021	● Verified execution of 10 user workload file
February 21st, 2021	● Verified execution of 45 user workload file
February 28th, 2021	● Verified execution of 100 user workload file
March 14th, 2021	● Verified execution of 1000 user workload file
March 27th, 2021	● Verified execution of final workload file
April 3rd, 2021	● Group project presentations
April 5th, 2021	● Final project ● Submission of project source code