Algorithmic Trading Application Design

Document

Jacob Berwick, Francisco Bolanos, Hae Won Cho, Dakota Cleeves, Alec Hayden, Matthew Jordan, Jose Ramos, Shile Song

CS10
CS 461: Senior Capstone Fall 2019
Oregon State University

Abstract

We have designed our application as a microservice architecture with 8 separate components. Those 8 components are: React frontend which is the portion of the application the user interacts with and will send the user data to the backend. ASP.net Core Backend which is responsible for queuing and sending requests between the frontend and backend. Stock Market Data fetcher which will retrieve stock market data whenever we need to analyze data to create a portfolio. Trade Execution Module which will place market orders to purchase assets when a user accepts a portfolio or a portfolio change is made. Portfolio Management Module which keeps track of active, potential, and suggested portfolios. User Data Management Module which consists of a database that will store all user information. Stock Portfolio Generator which is the module that will analyze data and generate portfolios for the user. Risk Management Module which works in conjunction with the portfolio generator to determine how to minimize and handle risk.
CONTENTS

1 Introduction .................................................. 4
   1.1 Purpose ................................................... 4
   1.2 Scope ....................................................... 4
   1.3 Overview .................................................. 4
   1.4 References ................................................ 4

References ....................................................... 4

2 Design Concerns .............................................. 4

3 General Architecture ......................................... 5
   3.1 Design ....................................................... 5
   3.2 Rationale ................................................... 5

4 Individual Components ....................................... 5
   4.1 React Frontend ............................................ 5
      4.1.1 Design ................................................. 5
      4.1.2 React Framework ..................................... 6
      4.1.3 Learnability ......................................... 6
   4.2 Efficiency ................................................ 7
      4.2.1 Memorability ......................................... 7
   4.3 ASP.net Core Backend ................................... 7
      4.3.1 Design ................................................. 7
      4.3.2 Rationale ............................................. 8
   4.4 Stock Market Data Fetcher ............................... 8
      4.4.1 Design ................................................. 8
      4.4.2 Rationale ............................................. 8
   4.5 Trade Execution Module ................................. 9
      4.5.1 Design ................................................. 9
      4.5.2 Logistics Viewpoint ................................. 9
      4.5.3 Rationale ............................................. 9
   4.6 Portfolio Management Module ......................... 9
      4.6.1 Design ................................................. 9
      4.6.2 Rationale ............................................. 10
   4.7 User Data Management Module ...................... 10
      4.7.1 Design ................................................. 10
      4.7.2 Rationale ............................................. 12
   4.8 Stock Portfolio Generator ......................... 12
      4.8.1 Architectural Design ............................... 12
      4.8.2 Algorithmic Design ............................... 12
1 INTRODUCTION

1.1 Purpose

The purpose of this Design Document is to provide a guide of the architecture and microservices that make up the Algorithmic Trading Application. This document will be used by developers and stakeholders, like Levrum Data Technologies, to know how the application should be developed.

1.2 Scope

The Algorithmic Trading Application will be a web-based application that will create investment portfolios for users, containing stocks, options, and equities. The investment portfolio created will be dependent on the user’s risk tolerance and their account balance. The portfolio will be monitored and changed based on risk management and dynamic strategies. The intended users for this application are people nearing retirement or already retired.

1.3 Overview

After the introduction, this document will outline the primary concerns for the design of the application. Then the detailed descriptions of the general architecture and individual services will be presented along with their rationales. Lastly, there will be a proposed timeline for development.

1.4 References

https://www.codecademy.com/articles/what-is-rdbms-sql

https://www.tutorialspoint.com/sql/sql-primary-key.html

https://www.techopedia.com/definition/7272/foreign-key


https://www.nngroup.com/articles/usability-101-introduction-to-usability/

https://www.codecademy.com/articles/react-virtual-dom


2 DESIGN CONCERNS

There are three primary concerns for the design of the Algorithmic Trading Application. The first is meeting the usability needs of the elderly which are the target users for this application. The second concern is speed. Trading orders need to be quickly placed given that volatile markets have rapidly changing prices. The third concern is cost. This application will involve querying large amounts of data which can become costly.
3 General Architecture

3.1 Design

Fig. 1: This is the microservice architecture for the Algorithmic Trading Application.

The application will be web-based and consist of a React frontend with an ASP.NET Core backend. The backend and frontend will communicate with a REST API. Other than the backend and frontend, there are six other modules to be connected. A microservice architecture will be used to connect every component. These components are the stock market data fetcher, trade execution module, portfolio manager, user data manager, portfolio generator, and the risk management module. The backend will communicate with these other components through RabbitMQ topics, directly or indirectly. In figure 1, each black arrow between two components represents communication via RabbitMQ.

3.2 Rationale

The primary reason for using a microservice architecture is so that each component can communicate with each other quickly and only when necessary. Not every component needs to communicate with each other, so it is better for them to be decoupled instead of layered on top of each other. Components will only subscribe to the topics they need so that communication is free, effective, and multi-directional. Also, with a large development team it will be quicker to work on a microservice architecture because different components can be worked on simultaneously.

4 Individual Components

4.1 React Frontend

4.1.1 Design

We will be building our user interface using a React framework. When developing our user interface we are going to be focusing on Jakob Nielsen’s Usability standard. The first component is learnability, which is how easy it is for users to
accomplish basic tasks when they first come across the design. The second component is efficiency which covers how quickly users can perform the tasks they want to perform. The third component is memorability, which is whether or not a user is able to reestablish proficiency in using a user interface, after not having used it for some time. The fourth component is errors, which covers how many errors are present and how severe they are, and whether the user is able to recover from them. The last component is satisfaction which is basically whether or not users enjoy using your design.

4.1.2 React Framework

Our interface needs to be quick and be able to handle thousands of requests at one time to ensure our users have a good overall performance. We get this from the React framework. React uses a Virtual Document Object Model (DOM). The DOM is a tree like structure that allows scripts to interact with the content and structure of the webpage and update them. The virtual DOM only updates the specific element that has changed by comparing it to an old snap of the HTML code and figuring out what has changed from the previous version. A real DOM would work down the entire DOM tree and change/update everything until it reaches the element that was specified and changes that, but this would be inefficient. This will directly affect our accounts page because we will be updating a small portion of that page by whichever module is chosen on the left side panel.

4.1.3 Learnability

To make sure our UI is simple and easy to use we will minimize the amount of pages the user must visit to accomplish their tasks. Our homepage will have an area for the user to get started and it will have a step by step guide on how to start an account and start investing. The account settings page will be where the user can change any information about their account after initial setup. The Portfolio Module page will allow the user to see everything from their finance history, portfolio management, and a guidance module. The figures below represents a mock of the homepage (Fig 2) and the accounts page (Fig 3) of the user interface.

Fig. 2: This is mockup a of a proposed Homepage
4.2 Efficiency

Our target group is older so we want to make sure our application is efficient, to ensure this we are aiming to keep the interface simple so the user can easily maneuver around quickly and with minimal clicks. Users will probably want to login, manage something with their portfolio, and log out to make this easy upon a successful login the user will go to their portfolio information page where they will be able to see their finance history, manage their portfolio and use the guidance module.

4.2.1 Memorability

To keep our user interface memorable we made sure to keep it simple so there is not much to learn and is easy to remember. By grouping many modules within the accounts page where the user will be spending most of their time, they will quickly get used to the different modules of the application and retain where to find them and how to use the modules. We will make sure to keep consistent styles across all pages to assist in retaining how to use the application.

4.3 ASP.net Core Backend

4.3.1 Design

In order to connect the backend services with the frontend user interface, we will use an ASP.net Core backend. This backend module will be programmed in C# and will employ a message queueing protocol with RabbitMQ. This backend service will basically act as a bridge for the React frontend and all the microservices. The frontend and backend will communicate through a REST API while the backend communicates with the other services through RabbitMQ topics. This API will be used to give the frontend an interface to make requests to other services. Although this module is the bridge between frontend and backend, it does not communicate with every service directly. It only communicates directly with the user data management module, the stock market data fetcher, and the portfolio management module. Anytime one of these modules needs to send or receive data with the frontend it will have to go through the ASP.net
Core backend service. RabbitMQ will help in the queueing of these messages by employing the Advanced Message Queuing Protocol so that no request goes unnoticed or unordered.

The specific requests that the ASP.net core backend will deal with are the following: when a user requests a portfolio this module will tell the other services to start analyzing market data to create potential portfolios and once the portfolios are created, this module will send them to the frontend to be presented to the user; when the user accepts a presented portfolio, this backend module will message the portfolio management module to start the process of entering portfolio positions; any decision the user makes and weekly data will be sent to the user data management module for storage.

4.3.2 Rationale

The main purpose for the ASP.net Core backend module is to present the React application with an API for interacting with the other services. Also, the frontend won’t have to delegate tasks because that is the backend’s job. It would be possible to build the application without this module, but the message queuing would be very messy and each microservice would have to have their own APIs. We are using RabbitMQ as our message broker because it can communicate with many different types of technologies. This is great for our multiple different services, especially if something needs to be changed or added later. Also, RabbitMQ guarantees message delivery even if something crashes which will help in debugging as well as guarantee a consistent database for our user data.

4.4 Stock Market Data Fetcher

4.4.1 Design

The Stock Market Data Fetcher is one of the modules that the microservice architecture will connect with other components. The communication between the modules will take place through RabbitMQ topics. The only services planned to use the Stock Market Data Fetcher are the Portfolio Management Module, Stock Portfolio Generator, and the ASP.NET Core Backend Service. These services can request that Stock Market Data be fetched and published to the RabbitMQ topic by sending a Request message on that topic. Every time market data is requested from other services, the Stock Market Data Fetcher will get market data from one of our sources by calling API(s) with an API key(s). The Stock Market Data Fetcher should respond to requests by publishing stock Market Data on a RabbitMQ topic that other services can listen to, but it isn’t sending messages to any particular services. The published data can be subscribed by the Stock Portfolio Generator for analysis, the backend so it can be relayed to the user, or the Portfolio Manager Module for an update.

4.4.2 Rationale

We have decided to fetch stock market data every time users asks for them for now. Whether or not we use a database to cache stock market data depends on the cost and availability of historical data. We will weigh the cost of developing the database to cache the data and the cost of getting the data repeatedly. For now, the application will fetch the data from the source every time, but when the cost of fetching data for every request gets more expensive than the cost of caching data from a database, we can use the RabbitMQ topic as a cache for the stock market data by increasing the length of the topic queue. If the Stock Market Data Fetcher stores data in the queue, services that use the Stock Market Data Fetcher can first search the data they need. This will prevent unnecessary API requests and speed up the application.
4.5 Trade Execution Module

4.5.1 Design

The Trade Execution Module is only connected to the Portfolio Manager Module. The communication between these services will be done through RabbitMQ topics. The Portfolio Management Module will send requests to the Trade Execution Module when orders need to be made to buy or sell assets for a user portfolio. The Trade Execution Module will respond with a confirmation or status of the trade. Internally, the Trade Execution Module will have two main elements. The first element is Quantconnect’s lean engine for connecting the application with a broker to make live trades. The last component is a database of ongoing trades submitted by Quantconnect ordered by user.

![Diagram showing the flow of a request through the Trade Execution Module](image)

Fig. 4: This shows the flow of a request through the Trade Execution Module

4.5.2 Logistics Viewpoint

The following are supported brokers by Quantconnect: Interactive Brokers, OANDA, Coinbase Pro, and FXCM. A subscription and/or active account needs to be created in one or more of these accounts ahead of time.

4.5.3 Rationale

It is important to have the Trade Execution Module as a standalone service to ensure re-usability of the module. RabbitMQ has a variety of reliable protocols for efficient communication between two modules like the Trade Execution and Portfolio Manager module. This is important in live trading where slow or dropped trade request are undesirable. The trade log database allows for the module to remember submitted trades so that it does not wait on a reply from the broker before submitting another trade. It makes the module more efficient. Quantconnect’s Lean engine is free and open source. This reduces the cost of managing the application and allows for flexibility in the trading process in Quantconnect’s engine.

4.6 Portfolio Management Module

4.6.1 Design

The Portfolio Management Module is used to orchestrate the acquisition of assets and manage active portfolios. Once a user has accepted a suggested portfolio this module will send a request to the Trade Execution Module to enter the positions. Then, it will receive updates from the Stock Market Data Fetcher and compare this data with the User Data Management Module to decide whether an order needs to be placed based on user defined risk tolerance and the Risk Management Module strategy. This module will also record all of these decisions and updates to present to the user on the frontend. Some of these decisions and data will also be sent to the User Data Management Module for recording in the database.
Fig. 5: An example of what data is transferred to the Portfolio Management Module. The Portfolio Management Module transmits the accepted data to the processor modules.

4.6.2 Rationale

The Portfolio Management Module is the center point for the application because it is the service that connects all the other modules. This module is directly connected to all other services using RabbitMQ for free and effective communication. This module could be coupled with the ASP.net Core backend service, but that would create a messy message queue. Also, if we ever need to refactor the application to use a different kind of user interface then this module will not need to change.

4.7 User Data Management Module

4.7.1 Design

To store the various information about our users, we will be using a relational database. Relational databases are databases that care about the relationships between the data stored inside. Often times, relational databases are broken into tables. Instead of having one extremely large database with many attributes, we can split the database into many tables, each of which holds specific entities and attributes. For example, we will most likely have an address table which simply holds the addresses of our users. However, we need to a way to tie users to their addresses, so a relational database allows us to create a relationship between a user and their address. Having this structure allows us to create an organized and easily accessible database.

To administer and handle the relational database, we need a Relational Database Management System (RDBMS). For this, we will be using Structured Query Language (SQL). SQL is the standard language that allows us to manipulate, interact, and communicate with our relational database. There are many RDBMS that are based off of SQL. For this project, we will be using MySQL because it is open source, and its ease of use.

Before we create our database, we will create an Entity-Relationship Diagram (ERD) and a database schema. The ERD is a high level way of visualizing the data in a database. ER models usually just contain entity and relationships. Entities are the things in the tables, the actual objects themselves. The relationships describe which tables are related and how they are related. For example, we are guaranteed to have both a user table and an address table. The entities that make up the user table are users. The entities that make up the address table are addresses. The ERD would also show a relationship between users and addresses because during implementation, we want to be able to tie users to
an address. The type of relationship between the user table and the address table will most likely be a many-to-many relationship in which one user can be tied to multiple addresses, and one address can be tied to multiple users.

Fig. 6: An example of an ERD with two tables, Users and Addresses. Here the two tables have a many-to-many relationship. That is, each user must have at least one address, and one address must be tied to at least one user. We will be creating an ERD which contains every table and their relationships to other tables.

As for schema, they are similar to ERDs but they are not as abstract. Schema get into the details by outlining what the names of the tables will be. They also list the attributes that the table will contain. The attributes are the traits that make up the entities in a table. For example, an address is an entity that will be stored in a table in our database. The address will not be one really long string, but instead, will be composed of many attributes. Possible attributes include: house number, street name, street type, city, etc. Not only do we list the attributes, but we also formally name them. Lastly, one of the more important features of schema is that they show both primary keys and foreign keys. A primary key is a field or set of fields, that can be used to identify a unique row in a database. That is, a primary key should find one and only one row in a database. The primary key should only consist of the smallest number of attributes needed to retrieve a single, unique row. For example, in our users table we will have a User ID attribute. That User ID will be the primary key because it will always retrieve a single, unique row since User IDs have to be unique. Foreign keys are important because they are what actually links two tables and provide that relationship. The foreign key is created by placing the primary key of one table, inside another table. For example, to create the relationship between the users and addresses tables, we would take the primary key of the users table, which is User ID column, and place it inside of the addresses table. Therefore, the User ID column in the addresses table acts as the foreign key which ties an address to a user. For now, we don’t have complete information to be able to completely draft our ERD and schema. Closer to the beginning of development, our team will hold a development session in which we will create the blueprint for our database that holds the user’s personal information.

Fig. 7: This is an example schema showing the relationship between the Users table and the Addresses table, and the associative table between them, as it is a many-to-many relationship. The schema shows which attributes will make up each table, what the primary keys are, and which foreign keys exist.
Once we have the ERD and schema figured out and completed, it will be time to actually go ahead and create our database. The first thing we will want to do is create the tables we want to use. As stated previously, we will be using MySQL as our RDBMS. We will write the queries that create the tables. These queries contain the tables names, the attributes within the table, the attribute’s names, types, constraints, and other defining traits. The create table queries also outline primary keys and foreign keys.

Once we have created the tables, we will need to create the queries which will allow us to insert rows into tables, update rows in tables, and delete rows in tables. These queries will be made for every table in the database, allowing them to be manipulated and edited. The queries will use the SELECT, INSERT, UPDATE, and DELETE operators to perform operations on the tables.

4.7.2 Rationale

We chose to implement a relational database to store our user information because we care about the relationships between the various pieces of information about our users. NoSQL, or an unstructured and non-relational database is better suited for large amounts of data as well as distributed data, where the relationships in the data do not matter, which is not suitable for our project.

We chose MySQL as our RDBMS because it is an all-around solid RBDMS. First of all, MySQL is extremely easy to learn and create queries that perform operations on our database. It is open source and well-documented. Support for MySQL is also easy to find. Another advantage of MySQL is that it provides strong data security and is very reliable, with great performance and uptime.

4.8 Stock Portfolio Generator

The Stock Portfolio Generator is where the actual portfolios are generated from various equities such as stocks, options, and bonds.

4.8.1 Architectural Design

As with the rest of the components, the Stock Portfolio Generator will communicate with the other modules via RabbitMQ protocols. It connects mainly with three other modules, the Stock Market Data Fetcher, Portfolio Manager Module, and the Risk Management Module. The Stock Market Data Fetcher will be crucial for this module as the underlying models within the generator will need access to both historical data to train the models, and real time data to both generate new portfolios and rebalance existing ones. Additionally, models should be constantly updating and retraining as time goes on to keep the models performing well with changes in the market. The Portfolio Manager Module is where the portfolio requests will come from and the portfolios are returned to. When requesting a portfolio, the Portfolio Manager Module will pass along the relevant user data such as risk tolerance, account balance, etc. Based off these inputs the generator will determine which model to use when allocating a portfolio. After generating a portfolio, it will be passed to the Risk Management Module for risk assessment before it is returned to the Portfolio Manager Module. Depending on the results, the generator will either return the verified portfolio, or adjust or regenerate a new portfolio and send it back to the Risk Management Module until verification.

4.8.2 Algorithmic Design

Many different models using various techniques will be used to generate portfolios for users. While our final strategy is yet to be determined, our research has indicated promising methods to use in testing. Genetic Algorithms will likely
be used to select features for the various models used. However, this alone will not be able to generate a portfolio and models to predict future stock prices, risk, etc. Some promising algorithms to achieve this are support vector regression, recurrent neural networks using LSTMs, and reinforcement learning. These models will have their input features optimized by genetic algorithms before final model evaluation. When it comes to training, the models will be trained on a period of historical data with a validation set of more recent data that the model was not trained on. After the models have been evaluated and the top performers are selected, we can use them to analyze a set of equities based on user data. Based on the results the equities will be ranked and a portfolio will be generated proportionate to the rankings.

4.8.3 Rationale
Due to the nature of the microservice architecture it is imperative to be able to test the module independently of the others. Since the modules will communicate through RabbitMQ, we can test the Stock Portfolio Generator through API calls and validate the output. Additionally, real time stock data from the Stock Market Data Fetcher will be needed in order to make sure the models have the most up-to-date information when making decisions. The underlying models will need to be analyzed often by the Risk Management Module to ensure they meet user requirements.

In an application such as the stock market, there are hundreds if not thousands of indicators, financial statistics, and fundamental variables. For example, one could look at Price-to-Earnings Ratio, Earnings per Share, Liquidity Ratio, Volatility, among many more. Genetic Algorithms can allow us to determine which model inputs will result in the best performance without manually testing each possibility. Genetic algorithm feature selection has shown to greatly improve performance compared to the benchmark. Support vector regression in particular has been shown to be effective using optimized features from genetic algorithms. Recurrent neural networks using LSTMs have the advantage of LSTMs being able to remember long term trends better than other models. They especially excel with time series data and thus are great for the stock market. Additionally, they have the ability to take in various lengths of time as input data and also output various lengths of time. Thus, they can be used for different time frames and predict values at various times in the future. As for reinforcement learning, it also has advantages when applied to the stock market. It learns how to act in different situations and thus can make decisions in line with what a human may do. Rather than just predicting prices, reinforcement learning can learn when to buy, sell, hold, etc. Additionally, since the reward function can be tied directly to portfolio value or risk, it can also learn when to divest if it believes it will limit loss.

4.9 Risk Management Module
The Risk Management Module is the decision maker on how to manage risk in portfolios and trades.

4.9.1 Architectural Design
The Risk Management Module, in the scope of this project’s architecture, only interacts with the Portfolio Manager Module and Stock Portfolio Generator. These modules use RabbitMQ’s software and protocol(s) to allow the modules to communicate with the Risk Management Module. Communication here is bidirectional among the modules.

4.9.2 Algorithmic Design
The following is the overview design of the algorithm for this module:

Inputs: Requests from Portfolio Manager and/or Stock Portfolio Generator Modules.
**Outputs**: A decision of actions to take like buy, sell, hold of data set based on risk strategies.

**Main Functions**:

*Receive_Request(connection)*
Input: connection from which to look for requests
Output: data from request and what module requested the analysis
Description: This function will check for requests from the Portfolio Manager or Stock Portfolio Generator Module and initialize the risk analysis.

*Risk_Assessment(request)*
Input: request information
Output: information on the cumulative risk analysis done
Description: This module will call on the various Risk_Strategy functions to evaluate risk based on the request.

*Risk_Strategy1(data)*
Input: data extracted from the request
Output: risk analysis results

*Risk_Strategy2(data)*
Input: data extracted from the request
Output: risk analysis results

*Make_Decision(risk_results, request)*
Input: risk results from Risk_Assessment and request information
Output: A list of recommendations
Description: This function will use the risk results to make decisions based on the requirements of the original request.

*Send_Results (decisions, connections)*
Input: list of decisions and original connection and request
Output: success or failure on sending the information to the other module
Description: Will send the result of this module’s recommended actions

4.9.3 **Stakeholder Concerns**

Risk management in trading is very important in this project and thus a big concern for our client may be how can we make sure our Risk Management Module is effective and reliable. By making this its own module we can easily unit test this module for efficiency and reliability.

4.9.4 **Strategy Design**

The risk management strategies will be implemented as independent functions inside the module. As shown in the algorithmic viewpoint. Examples of risk management that could be included in the module are stop-loss, take-profit, risk of ruin, diversification, hedging, and expected returns.
4.9.5 Rationale

This design of the module is important because it can be implemented and tested independently from the other modules. Making it easy to develop and test this module’s functionality. RabbitMQ is what will be connecting the modules once it is ready to be integrated. The strategy design for this module also allows for strategies to be independently developed and for new strategies to be easily added in the future.

5 Timeline

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ASP.net Core Backend</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stock Market Data Fetcher</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stock Portfolio Generator</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trade Execution Module</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Risk Management Module</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Portfolio Manager Module</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User Data Management Module</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>React Frontend</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 8: Gantt Chart: Timeline of Project Tasks

6 Conclusion

The Algorithmic Trading Application will use a microservice architecture to connect 8 separate services. Each service has a single purpose and only communicates to others when necessary. The end result is a complete algorithmic trading application with a friendly user interface that will be complete in May 2020. This Design Document intends to communicate exactly what will be developed and will change as necessary.

7 Glossary

ASP.NET: Open-source framework for building web applications in .NET and c#, page 7

Backend: The portion of an application that handles storing and manipulating data received from the front end, page 7
Efficiency: Refers to how quick a user is able perform the actions that they want to perform, page 7

Entity-Relationship Diagram(ERD): Diagram representing a database’s tables and the relationships between them, page 10

Frontend: The portion of an application the user will interact with, page 5

Genetic Algorithm: Metaheuristic machine learning algorithm bases on the evolutionary principle of natural selection, page 12

Learnability: The degree to which a user is able to quickly learn how to use an application, page 6

Memorability: The ability of a user to remember how to use an application, after time away from it, page 7

Microservice Architecture: Software development technique that separates an application into separate, independent services, page 5

Quandl: Marketplace for financial, economic, and alternative data, page 8

Quantconnect: Open-source, cloud-based algorithmic trading platform, page 8

RabbitMQ: Open source message broker software, page 5

React: Open-source javascript library for developing user interfaces, page 5

Reinforcement Learning: Aspect of machine learning that focuses on the actions the agents take, page 13

Relational Database Management System(RDBMS): System that allows for the administration and interaction of a relational database, page 10

Relational Database: Database system that is broken up into tables with explicit relationships between tables, page 10

Representational State Transfer(REST API): Software architectural style, page 5

Schema: Diagram representing the attributes that make up the tables, as well as the primary and foreign keys inside, page 11

Structured Query Language(SQL): Standard language used to manipulate a relational database, page 10

Usability: The degree to which a user can successfully use an application to perform their intended goal, page 6