CS261 Final Report

Team 29

Sylvester Cardorelle,**Tudor Cismarescu**, **Ollie Madelin**,
Josh Footman, **Joseph Parkins**, **Anil Kiral**

March 2017

Contents

1 Installation Guide/ System Requirements

1.1 Setup

1.1.1 Node

1.1.2 SQL

1.1.3 Angular

1.1.4 Npm Installation

1.2 Running

2 User Guide

2.1 Home Page

After completing the installation phase, the application will be ready to use. The landing page will look like the figure below.

Figure 1: Spikes Home Page

Welcome to SPIKES! From the homepage you can choose from three tabs at the top of the page; Analysis, Static Data File or Live Stream. The Static Data File tab is where to go when you have uploaded a file in the Analysis tab, here it will show you all the anomalies in the data. The Live Stream tab is identical to the Static Data File but is for live stream data instead.

2.2 Analysis tab

The Analysis tab is the starting point of the app, here you can upload either a static data file or enter an input stream source, along with stopping and starting the analysis. In order to select the other two tabs, analysis will have to have been initiated on this page.

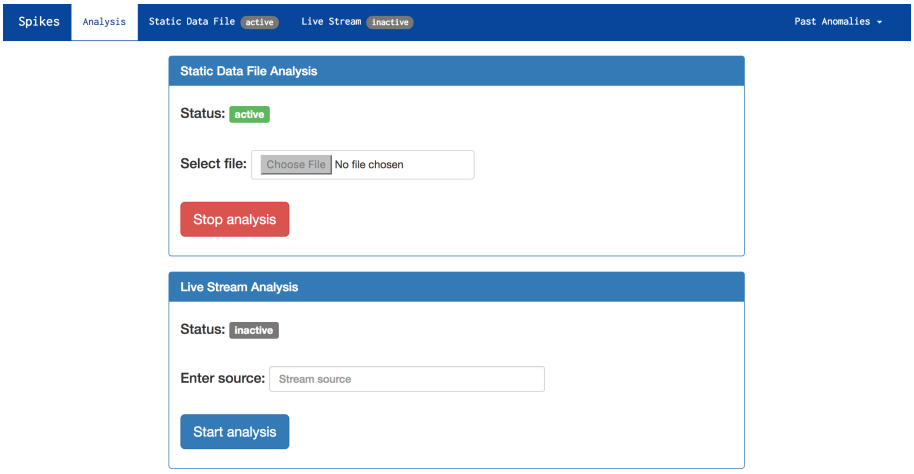

Figure 2: Analysis Tab

It consists of two controllers, one for static historical data and the other for a live stream data. Each controller has two options; upload and start analysis. There are no file size restrictions when uploading given sufficient hard disk storage. By commencing analysis, it will alert you that analysis is active, you will then be allowed to navigate to the corresponding tab.

2.3 Live Stream tab\Historical Data tab

When analysis is started, anomalies will start to be detected and generated in the respective tab, each anomaly that is detected will appear as an alert box on the page. Alerts contain only crucial data such as; Type of Anomaly, Symbol, Date and a drop down arrow that leads to more specific data. Volume spikes and price manipulation alerts will have a drill down button that generates a graph surronding displaying the suspicous data.

3 Research

After the analysis and design of the project, our software development team had developed a stronger idea of the technologies that would be used to design the system. This section further expands on the decisions and research that was made after the first deliverable that allowed us to finalise our choice of technologies.

3.1 Front-end

We decided as a whole group quite early that the whole application would be web-based rather than a desktop app. This would increase the accessibility of the application being cross-browser compatible and mobile friendly. If the client integrated a sufficient security system, the system could be accessed from anywhere with a secure internet connection.
The **AngularJS** framework was used to build the frontend as planned. We considered other JavaScript frameworks, such as **React**, however we felt that AngularJS was the better choice being a full-fledged MVC (Model-View-Controller) framework. As our system is a single page application (SPA) that will update dynamically, AngularJS became a natural choice for our developers.

After this decision was made between the Project Manager and Developers, our front-end developers took the initiative to intensively learn AngularJS using a multitude online resources. For example, Codecademy a programming teaching platform was used to learn AngularJS basics, whilst AngularJS Community pages were used to further expand their knowledge.

3.1.1 JavaScript Graph Library

Deciding on a suitable JavaScript Graph Library was crucial for this project as data visualisation was one of our major non-functional requirements. After extensive research we decided to use the **PlotlyJS** library for the following reasons:

- It generates responsive graphs that are able to scale dynamically on different platforms e.g. desktops, tablets
- PlotlyJS is known for use in financial analytics and supports a variety of graphs e.g. candlestick, boxplots
- Uses WebGL technology for graph rendering with a speed that surpasses many competitors
- Provides the option to export the rendered graphs as PNG files for further inspection if needed
- Allows the zooming into of graphs, allowing the user to further inspect specific areas of the graph

3.2 Back-end

3.2.1 Server-Side

Node.js was used as our interface between the frontend and the Java backend using the Express framework. This decision was made by the Back-end developers built on their exisiting skills with JavaScript. Node is extremely fast, compiling JavaScript code directly into machine code using Google's V8 engine, this means when facing large volumes of data the system will stay responsive due to the way Node handles concurrency. The modules and templates written in node can be easily reused throughout the serverside, thus reducing the size of our application and chance of bugs occuring.
Other popular server-side-scripts were considered such as Django and Ruby on Rails but given the unfamiliarity and time constraints, we discarded them because they did not fully utilise the current skills of the developers.

3.2.2 Database

Deciding whether to store stock data into a MySQL database became a frequently debated topic throughout the development of the application. If the system integrated a database ; storing the input stream into the database for a reasonable time period, it would allow us to select relevant data at will and improve the chance of the system finding potential perpetrators of any malicious anomalies.
Despite this, if the system relied on the database too heavily, a performance bottleneck would occur. Thus, making our system slower and less responsive. Our developer's countermeasure to

this was to cache the data within Java's abstract data structures and storing data for a certain amount of time until the oldest data would be useful in further algorithmic calculations.
Once the Back-end developers had come to a decision. It was then agreed between the whole group to run the system without a database, thus avoiding any bottlenecks and creating a multi-threaded application with minimal lag.

4 Design

4.1 User Interface

The main focus of the user interface was to provide a view so simplistic and natural that the client would not even consider using the provided manual. After sending the mock UI design to the client and received positive feedback, the front-end team decided to use the mockups as a guidline.
The color scheme of the user interface primarily uses a familiar but refreshing blue because it provides a contrast that does not strain the eye and draws the visual attention of the user. The font size used throughout the interface follows the universal usability principles such that the more important the information is the larger the font. Therefore, the larger text is scanned and smaller text is read more carefully. Also, the separation of functions onto individual tabs, prevents the user from being overwhelmed with information.

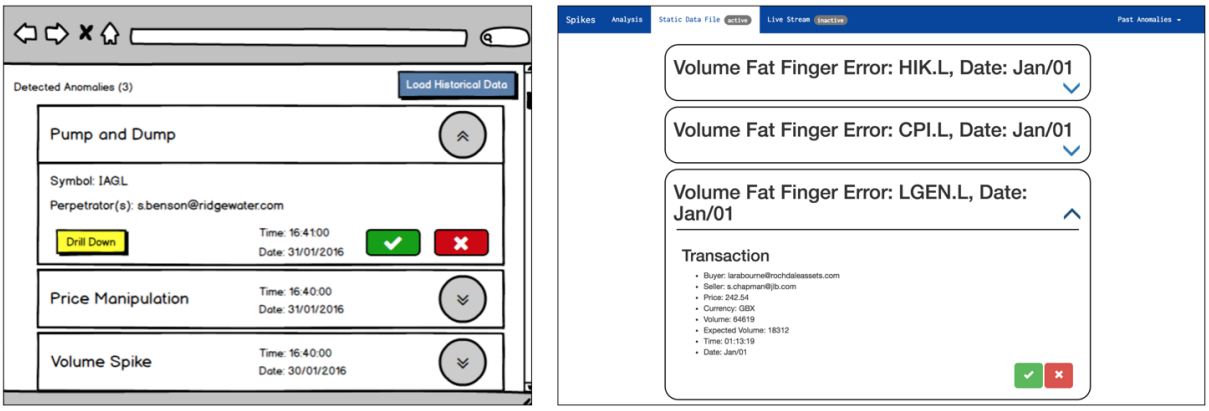

Figure 3: Evolution of user interface

4.2 Pseudocode

This section explores the type of algorithms that we used to analyse the input stream of data for anomalies.

4.2.1 Pump & Dump

Arguably, Pump & Dump was the most challenging type of price manipulation that we attempted to detect. After researching the characteristics of a Pump & Dump, our team concluded the following about Pump & Dump:

- A large batch of cancellation orders will occur before the dumping of stocks by a person or group (**Undetectable with current data set**)
- A steady increase of stock value from the average, when the pumping state begins (**Detectable**)
- A sudden drop in stock value when the dumping of stocks occur (**Detectable**)

Figure 4: Pump & Dump Example

By identifying the traits of a Pump & Dump, we were able to build the above model in fig:. We found it more intuitive to design an algorithm that would first detect a dumping state, then look back on historical data to detect whether a pumping state had occured before the drop in stock value. Below in fig: is the pseudocode that would later be transformed into Java code.

```
PumpAndDump{

        if( PriceAverageList.Size() < 60) {
            return null ; // As there is not enough data to run analysis on
        }
        else {
            dumpingState = false ; // Set flag
            //Check for a dumping state (large decrease in price)
            //Look at latest and previous price averages
            //Look at percentage decrease, if there is a large enough decrease then a dump has occured

             difference = PriceAverageList.Get(LatestPrice) -  PriceAverageList.Get(PreviousPrice) ;
             percentage_decrease = (difference/PriceAverageList.Get(PreviousPrice)) *100 ;

             if (percentage_decrease > -30){
                 dumpingState = true;
             }

             if (dumpingState) {
                 // If the dumping state is true, we then lookback on the 50 prices,
                 // before the drop to look for a pumping state
                 Pumping = 0 ;
                 for the last previous 50 stocks {
                   if(PriceAverageList.get(currentIndex) > PriceAverageList.get(currentIndex-1)){
                     Pumping++ ;
                   }
                 }

                 if (pumping >= 30) { // Then a Pump & Dump has occured
                   // Build an Anomaly object containing relevant details about the Anomaly

                   return Anomaly ;
                 }
             }

        }

}
```

Figure 5: Pump & Dump Pseudocode

4.2.2 Volume Spike

To detect volume spikes, the algorithm keeps track of the amount of stocks bought within a certain time interval and calculates a volume moving average (VMA) that will be used as a comparison for the consequent interval. Therefore, if the total stock volume in a period greatly surpasses that of the previous VMA by a user-controlled threshold. The program will flag a volume spike, as seen below in fig:. The threshold level would then adjust it's sensitivity to spikes depending on the user's feedback.

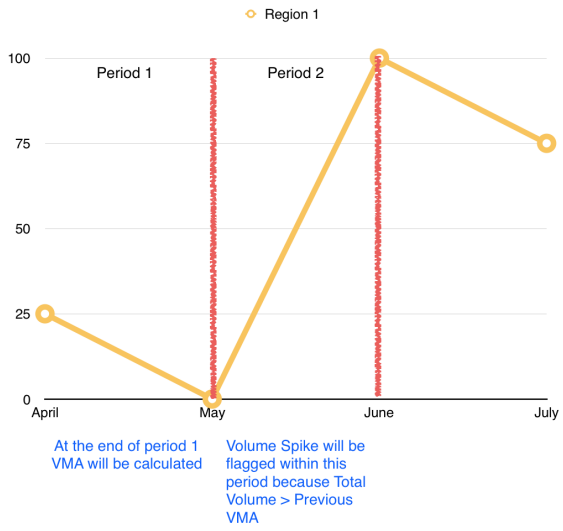

Figure 6: Volume Spike Example

```
Volume Spike {

    if (flag) { //if a volume spike has already occurred within the current time period
      return null;
    }
    else if ( currentPeriodVolume > limit) { //if amount of stocks sold has passed the dynamic limit
      //volume spike has occured
      flag = true;
      return anomaly; //send of the anomaly object containing all relevant information about the volume spike

    }
    else {
      flag = true;
      return null;
    }
}

```

Figure 7: Volume Spike Pseudocode

4.2.3 Fat Finger Errors

When detecting fat finger errors we decided to separate the error into two types:

- Fat Finger **Price Error**, which will detect fat finger errors related to the price of a stock transaction
- Fat Finger **Volume Error**, which will detect fat finger errors related to the size of a stock transaction

This is because it is possible for the Price or Size of a transaction to be extremely large compared to the average due to a typing fat finger error. The algorithm we developed to detect fat finger errors, uses orders of magnitude to determine whether the price or volume is extremely larger or smallerf than the average (in factors of ten).

```
FatFingerPriceError {
        if( ((stock.getPrice() > lastVma*10^n ) || ( stock.getPrice() < lastVma* 10^-n )) && (lastVma != 0) ) {
            // if the volume of a stock is extremely large or small then ff has occured

            //then there has been a price ff error
            //calculate severity of the ff error

            severity = (stock.getPrice() * 100) / lastVma;

            //send anomaly
            return anomaly;
        } else {
            return null;
        }
    }
}
```

Figure 8: Fat Finger Price Pseudocode

```
FatFingerVolumeError {
        if( ((stock.getSize() > lastVma*10^n ) || ( stock.getSize() < lastVma* 10^-n )) && (lastVma != 0) ) {
            // if the volume of a stock is extremely large or small then ff has occured

            //then there has been a price ff error
            //calculate severity of the ff error

            severity = (stock.getSize() * 100) / lastVma;

            //send anomaly
            return anomaly;
        } else {
            return null;
        }
    }
}
```

Figure 9: Fat Finger Volume Pseudocode

5 Implementation

5.1 Frontend

5.2 Backend

5.2.1 Parsing

6 Sprint Cycles

This section gives an overview on what each sprint cycle accomplished and how our system evolved using an agile approach.

6.1 Sprint Cycle 1

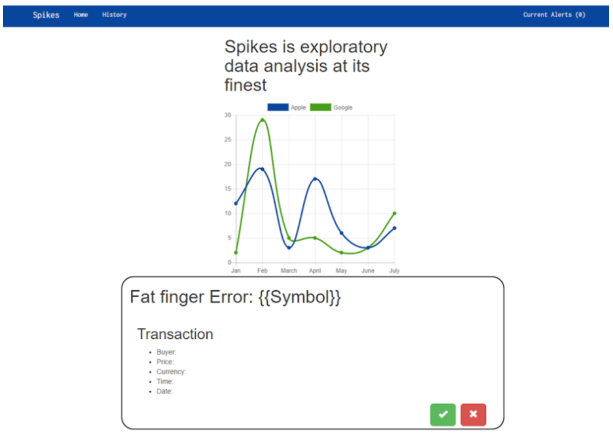

Figure 10: Initial UI skeleton

The first sprint cycle consisted of our back-end team designing the various algorithms while the front-end team worked on the UI skeleton simultaneously. We wanted to get the skeleton of the UI implemented as soon as possible, so that there was more available time for refinement. Initially, we had two pages; Home and History. The Home page would show all the anomalies and the History tab would have past anomalies that have since been removed from the Home page. The graph at the top of the page was created with Chart.js charting library and was used to give the team an idea what an example graph would look like. The anomalies would appear within the structures below the graph, where the user could either accept or reject the anomaly.

6.2 Sprint Cycle 2

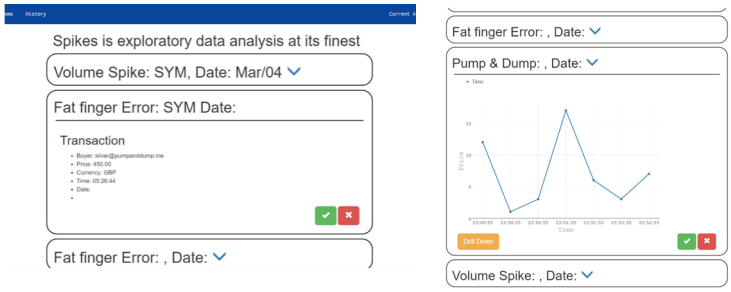

Figure 11: Development of alert boxes

In the second sprint cycle. We implemented the drop down arrow on the alert boxes so that design resembled the mockup more. We decided that the type of anomaly, symbol and date were the most important details regarding an anomaly, so these details were left outside of the drop down. It was also decided in the sprint cycle review meeting that Chart.js was insufficient to show the necessary data, instead Plotly.JS was used in its place as it provides much more features and an extensive range of graph types. The Drill Down button was implemented for volume spike and pump & dump, when clicked it would render a graph which would provide a better analysis of the anomaly.

6.3 Sprint Cycle 3

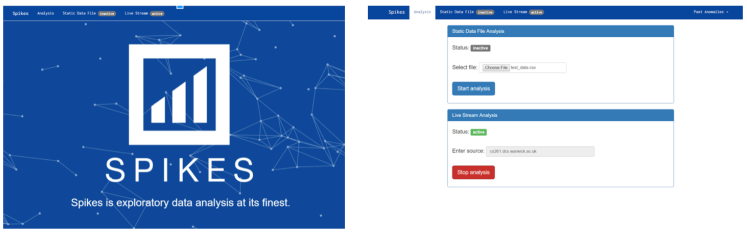

Figure 12: Introduction of Spikes 2.0

In the third sprint cycle, a major update to the UI occurred. A new homepage and logo were designed. We added three new tabs; Analysis, Static Data File and Live Stream. We also implemented the pseudocode for volume spike, the implementation required various adjustments as errors were realised throughout testing and eventually overcome. At this point, we were only able to test the algorithm with mock data collected in the Java back-end.

6.4 Sprint Cycle 4

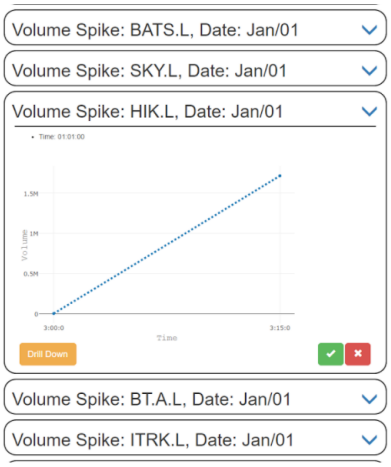

Figure 13: Volume spike detection fully functional

In the fourth sprint cycle we implemented the pseudocode for fat finger errors, initially it used simple bounds to detect an error, however after some research we decided to use orders of magnitude, however this wasnt working how we imagined it to do so. A realisation occurred in this sprint review meeting ; that two types of fat finger could possibly occur (Price/Volume). Our target for the next sprint cycle was to implement a fully-functional algorithm to detect Pump & Dump, with the system fully connected.

6.5 Sprint Cycle 5

In the final sprint cycle, we implemented the algorithm for Pump & Dump. This was most challenging for our team and although we had created the pseudocode in an earlier week we realised that this was not the optimal solution. After some discussion as a team we decided upon a solution that could work, however implementation proved to be just as difficult. After tweaking the sensitivity we managed to get the pump and dump detection to a state that all team members deemed suitable.

7 Testing

Throughout the development process, new test data was fabricated by each developer during the implementation of each feature. Therefore, we are confident that the majority of faults were uncovered due to the variety of test cases that the overall solution was subjected to.

7.1 Unit Testing

During the implementation of our solution, each feature was tested as it was developed to ensure correct functioning before integration. This strategy allowed the developers to carry out unit testing specifically designed for each minor component, as they were implemented, removing any bugs early on. The result of ongoing testing was a deep understanding of these individual components which helped the development team better understand and fix later issues.

7.1.1 Front-end Testing

In the interest of providing a responsive and interactive user interface (UI), the front-end HTML and CSS development team systematically tackled each activity that the user may have had to carry out using the application through the UI. Any errors related to interface feedback and the interaction between the user and the system were fixed at this stage. This provided a medium through which the development team could interact with the system and test further features.
Regarding the proper processing of flagged anomalies by the AngularJS front-end, the UI alert generation was tested using fabricated JSON packets of anomaly data. Once these packets were received, the testers observed the generated UI elements for any discrepancies and made any necessary modifications. During this process, an interface template to be used between the front-end and back-end was decided upon by the entire development team to provide consistency across software components.

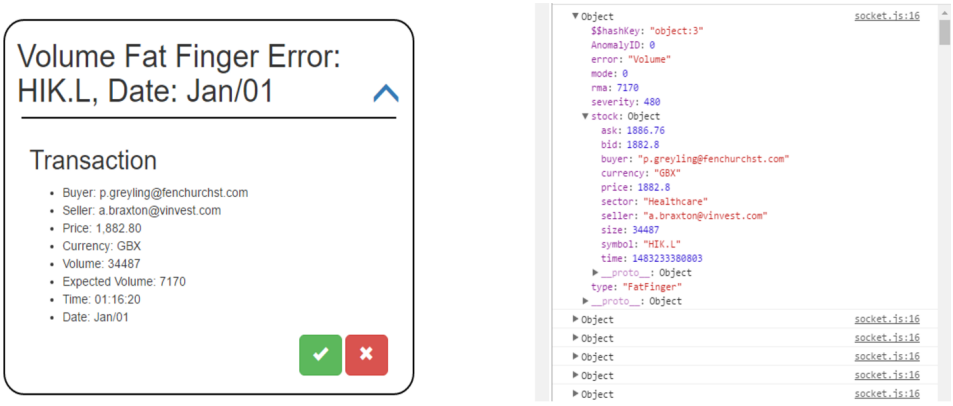

Figure 14: Testing of alert generation with mock data

7.1.2 Back-end Testing

All Java implementations of the analysis algorithms were subjected to intensive tests using the historical data provided, the live trade stream and fabricated trade data. The purpose of these tests was to verify that the algorithms were calculating reasonable values and flagging anomalous results correctly. Console reports (image num), printed during implementation of algorithms, were used to reveal the data processing done by the back-end.

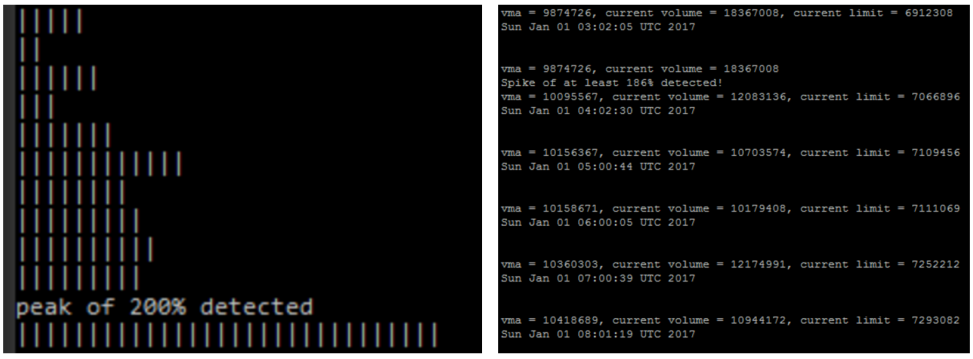

Figure 15: Testing of Java analysis algorithms

In addition, output data to be passed to the front-end was displayed to ensure that all interfaces established between developers were taken into consideration. Finally, all anomaly data being displayed had to be buffered by the NodeJS server before it was sent to the front-end. The development team observed data flowing into and out of the NodeJS server, and checked data stored by the server against expected formats and values. Removal of errors at this stage provided confidence in the transmission of data from the back-end processing to the front-end client view.

7.2 Integration Testing

Once each feature was completed, tested and debugged, it was integrated into the overall solution. Firstly, the node server was connected the Java back-end analyser using sockets and after this connection was fully tested. Secondly, the Node was connected to the Angular through socket.io, providing mock packets before the entire came together. Upon integration, tests were constructed and executed on the resulting, enhanced application. At this point, issues such as parsing errors were discovered by the front-end developers due to the mulititude of different templates generated by the angular and resolved before they could affect later versions. In this way, the team was able to ensure that any errors which appeared after the integration of a new component were strictly related to the component. This lead to easier debugging during each development iteration.

7.3 System Testing

Working versions of the application were tested using Black-Box Testing: fabricated data which contained patterns of real anomalies was fed into the software and resulting errors were checked for consistency with expectations. Additionally, working system versions were tested against the provided trade data files. The output of these tests was reviewed and checked against relevant parts of the input data to confirm that the anomalies were indeed relevant and the data displayed was correct. In addition to verifying correct detection of anomalies and transmission of data to the user, robustness and reliability were ensured by passing erroneous trade data to the analysis tool. This data included trades with missing fields or wrong types of data. The back-end development team set checks in place such that, even in such cases the system functioned correctly and the erroneous data did not affect later results.

7.4 User Acceptance Testing

In order to provide an application which was in line with expectations from the client, all features implemented were based on the initial Requirements Analysis, and Design and Implementation reports. According to the feedback received, our listed system requirements represented what is expected from the end product. Furthermore, the User Interface provided is based on the sketches delivered to Deutsche Bank, which received a positive reaction.

7.5 Non-Functional Testing

The application was also tested to satisfy non-functional qualities. The system performance was measured in terms of time taken to carry out processing and to display anomalies. This was done in the final sprint cycle. During an early period of development in which the system used SQL databases to store a large amount of data for eventual retrieval and analysis, we identified that database access tasks took a very long time. As a consequence, all other tasks would have to wait,

and the entire system was noticeably slower. To mitigate this issue, all database storage was made redundant and the database was completely removed from the process. Secondly, usability of the interface was tested by trials with users chosen randomly. The layout proved to be intuitive and easy to use, even for an inexperienced test subject. Moreover, all clickable areas were designed to be large and easy to use, and the colour scheme is suitable for colour blind users.

8 Project Management

8.1 Overview

8.2 Roles

9 Evaluation