DSCI 644 Design Document

Rochester Institute of Technology

Michael Kogan
Michael Kitching
Akanksha Arora
Mohammadreza Shojaei Kol Kachi
Muhammad Fazalul Rahman

Oct 16, 2020

Table of Contents

Introduction

The purpose of this document is to summarize the results of the work done during the elaboration phase of this project. Over the past few weeks, we revisited and broadened the requirements, developed a system design, and created implementation, test, and support plans. For a detailed introduction to the project please see our proposal document.

Requirements Revisited

One of the shortcomings we noticed with our previous document was that our requirements were developed solely around the model, and not around the system as a whole. As such, it was necessary to revisit the requirements analysis and broaden the specifications to include requirements pertaining to the hosting system. The extended list of requirements is presented in the table below. As before, there are three types of requirements: functional performance, and implementation.

Requirement	Title	Description
FR1	Enhanced Model	The provided model needs to be re - visited and enhanced to meet the performance requirements.
FR2	Model Input	The model shall receive as input only a text review. No other input variables should be considered.
FR3	Model Output	When given a text review, the model will output the probability that the review is positive. A review is considered positive if the associated rating is > 0.5.
FR4	Model Evaluation	The model will be evaluated using 0 - 1 loss on the test set.
FR5	Data Set	The model that will be used in production must be trained on the entirety of the provided dataset: “AppReview.csv”.
FR6	Software Hosting System	As part of the solution, a system will be provided to host the model on the client’s internal company network and provide controlled access.
FR7	Model Access (Frontend)	The model will be accessed through a web portal, which will serve as the frontend of the hosting system. This component will accept user inputs, convert the to the appropriate format, interface the model, and represent the results as required.

FR8	Single/Multiple User Input	The user will be able to either enter a single text review and obtain a predicted sentiment, or upload a CSV file containing many reviews, and download a corresponding file with associated predicted sentiments.
FR9	File Output Type	When the user uploads a CSV file as per FR8, the corresponding output file will also be a CSV file of ones and zeroes, representing positive and negative reviews.
FR10	User Authentication	Each user accessing the portal should be authenticated against the company's Active Directory. This will be done using Windows Integrated Authentication.
FR11	Model Hosting (Backend)	The model will be hosted on a backend server. The only task of this component is to respond to queries, replying with one or more predicted sentiments. As per FR7, appropriate presentation of these outputs will be handled by the frontend.
FR12	Model Interface	The model will be interfaced via a custom developed RESTful API. This will allow the client to easily access the model programmatically if required.
FR13	System Reproducibility	The system should be easily reproducible so that it can be locally staged at the various branches the client's company has. This will ensure that all associated traffic is kept local.
FR14	System Support	The system will be updated on a monthly basis as required, and will be managed according to the support plan presented in this document.
FR15	Project Website	A website needs to be created that will host the project and act as a gateway to all deliverables. This is separate from the frontend web portal used to interface with the model.
FR16	Project Website Content	Project details, including the project's purpose, requirements, goals, and architecture, should be included on the website. In addition, all project documentation and the Azure ML Studio model need to be linked.
FR17	Project Documentation	Each sprint is to be capped by a report which will include all deliverables associated with that sprint. The deliverables are listed in the "Team Project Instructions" document. These documents will be made available on the project website.

PR1	Prediction Latency	A single prediction should take no more than 0.1 seconds.
PR2	Model Accuracy	The model should achieve test error of less than 10%.
PR3	Number of Users	Although the anticipated number of users is small as this is a specialized application, each deployment should be able to comfortably handle up to 10 simultaneous users.
PR4	Availability	The system should be highly - available, and should be down for no more than one day every month to allow for regular maintenance (as per FR14).
ImpR1	GitHub	The project website must be hosted on GitHub.
ImpR2	Trello	The project must be managed through Trello.
ImpR3	Azure ML Studio	The model must be initially implemented in Azure ML Studio for demonstration purposes, before being integrated into the wider system.
ImpR4	Project Lifecycle	The project is to follow the OpenUP process and be broken down into four sprints: Initiation, Elaboration, Construction, and Transition.

Design and Architecture

The system design based on the expanded specifications listed above is conveyed in this section. We will first describe the hosting system in detail, and then proceed to explain our approach for coming up with what we believe is the best possible model for this problem.

Software Hosting System

This subsection describes the design and architecture of the system that will host the model and web portal. The decision was made to follow a client - server architecture with a frontend and a backend server to host the web portal and model respectively. These will be deployed on the client company's internal network in the same security zone as other company servers. Users on the client company's network will then be able to access the web portal as they would any other internal web application.

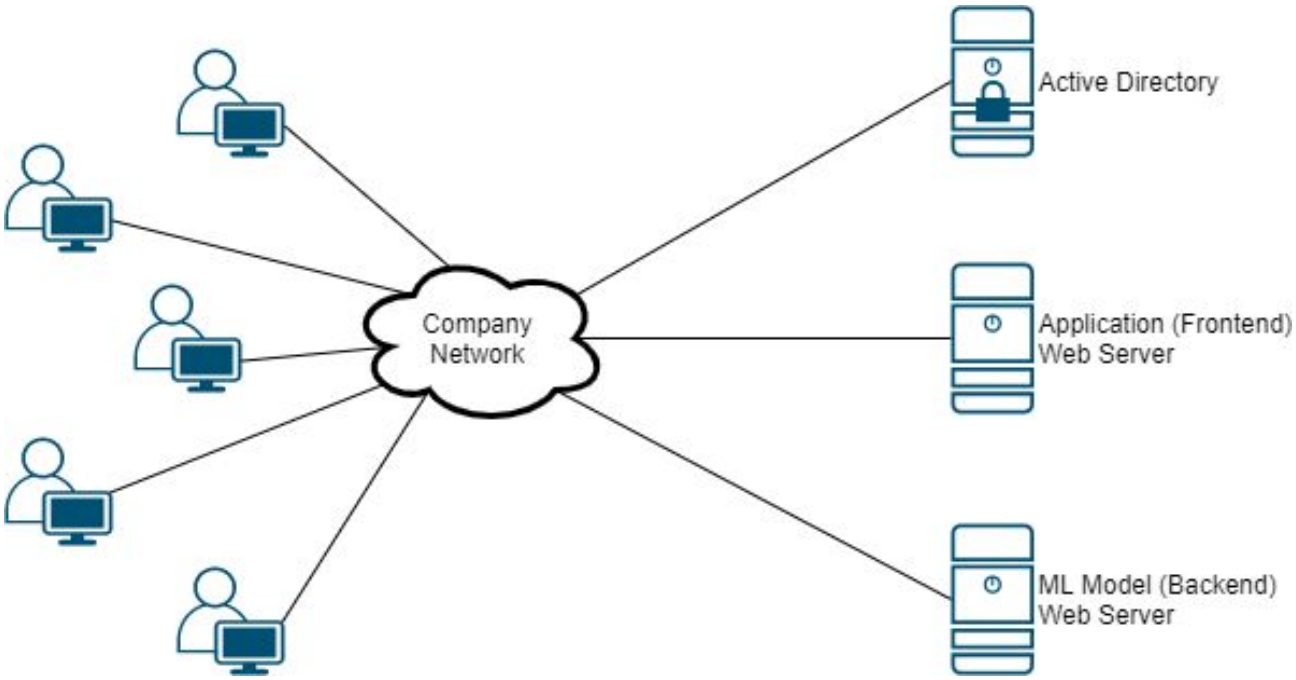

Figure 1 - Deployment Architecture

The software will be distributed across the frontend and the backend. The frontend will take care of user authentication, data representation, and sending queries to the backend. It will provide avenues for user input and will represent the results in an appropriate manner, as defined in FR8 and FR9. The backend will be responsible for receiving queries, generating predictions, and replying with the results. It is essentially a wrapper for the model, allowing it to be interfaced with HTTP requests. The backend will be designed with extensibility in mind.

When a user navigates to the web portal, they will be challenged to present their credentials. The web browser will automatically provide the same credentials that were used to log in to the company network (through the domain controller). The web portal will check these against the local active directory, and should there be a mismatch the user will be asked to re-enter their domain credentials. A successful authentication is illustrated in the following diagram.

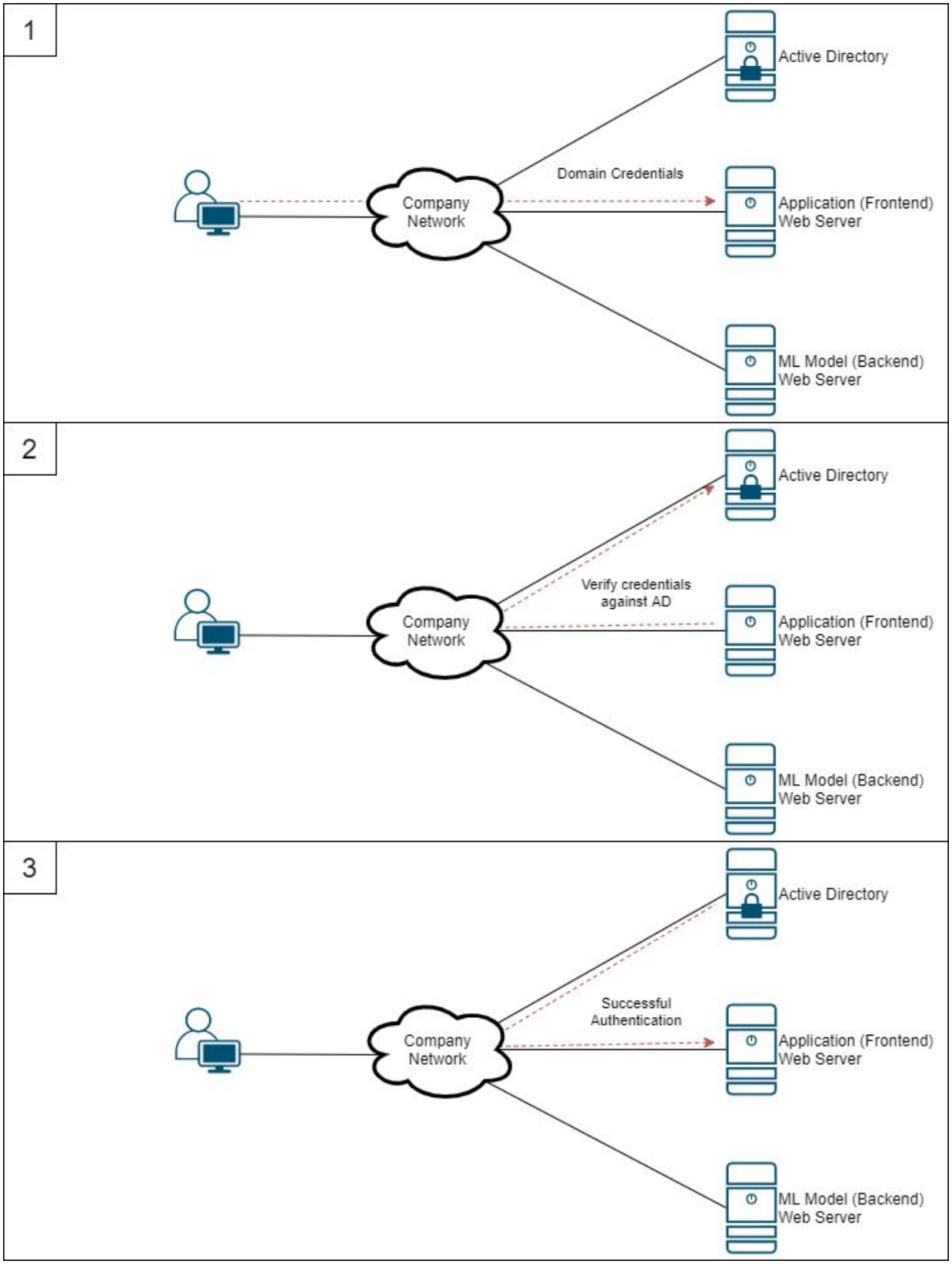

Figure 2 - User Authentication

Communication between the frontend and backend will occur using HTTP commands (via the API), namely the POST and GET commands. The frontend will send the input data as part of a POST request to the backend, which will in turn generate predictions and store them, sending a reply with the HTTP 201 status code, who's message body includes the location of the newly created predictions. The frontend will then be able to retrieve these predictions using a GET request.

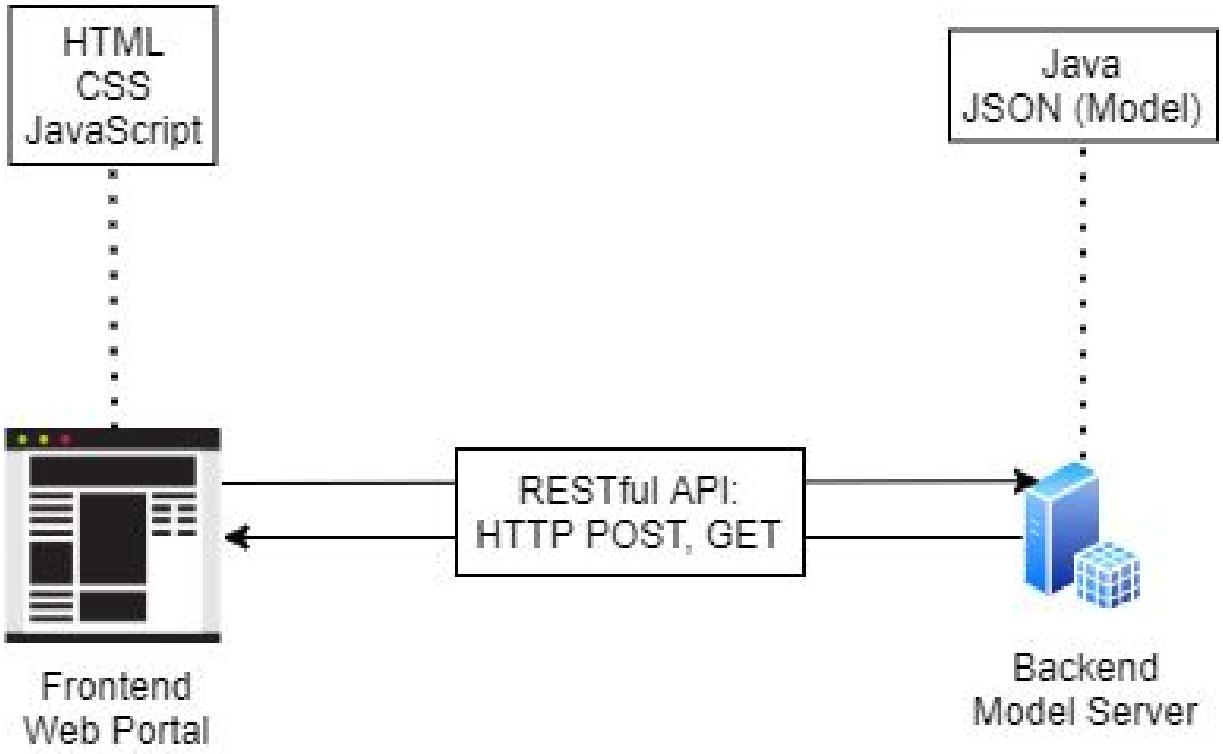

Figure 3 - High Level Software Architecture

The frontend software will consist of a web page with a javascript component to send and receive data and to appropriately represent the model's predictions, either by displaying it in the webpage (in case of a single query) or by generating a CSV file with ones and zeroes representing positive and negative predictions.

The backend software consists of the models and the wrapper to present these models as an HTTP server, and will be written in Java. The models themselves are trained and the final parameters are stored in json files on the backend server, which can be read from the disk on instantiation. The software architecture is shown in the following class diagram.

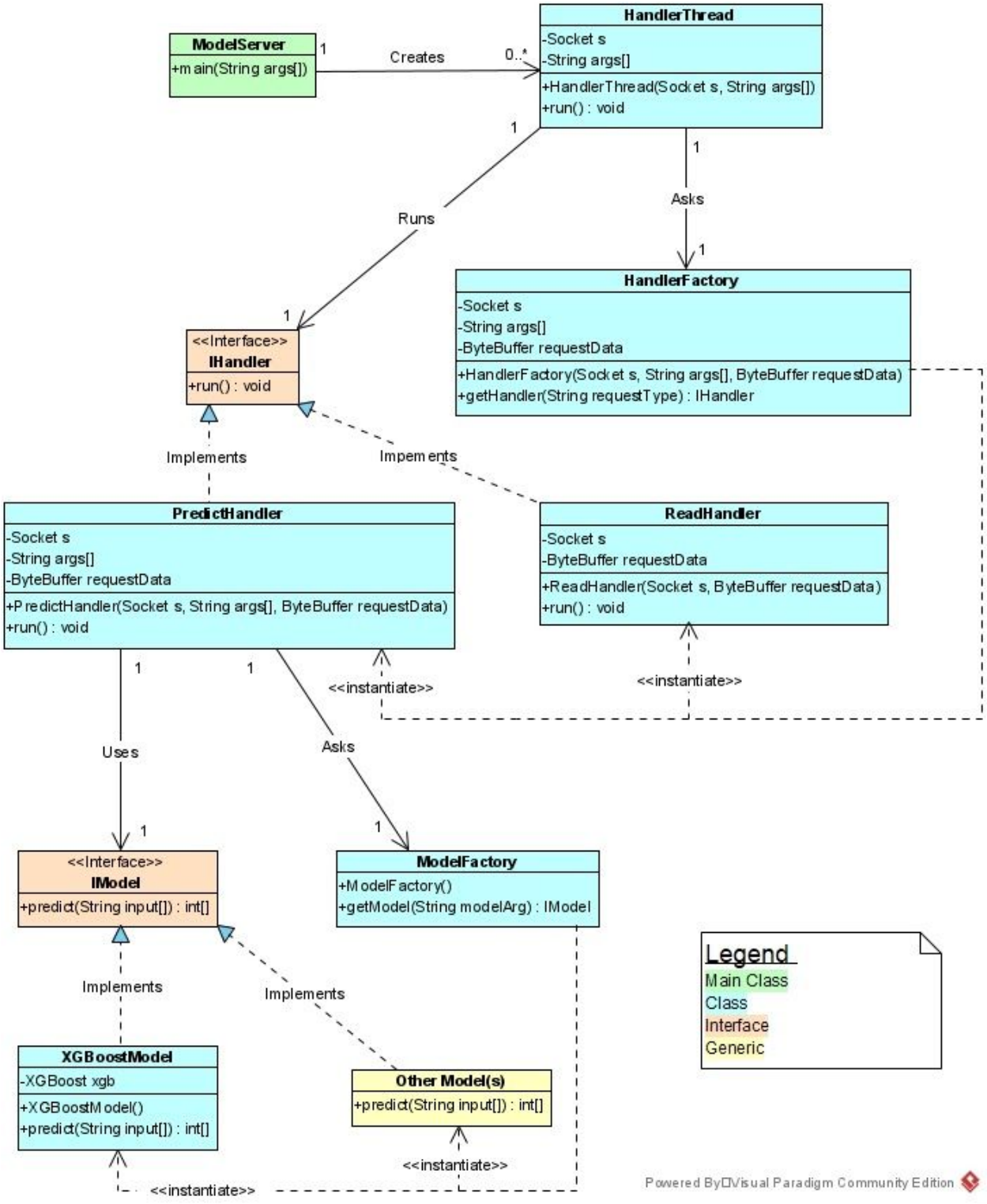

Figure 4 - Backend Class Diagram

In the following paragraphs we will describe each class in detail and provide sequence diagrams for some of the operations. Please note that these sequence diagrams are not complete in that they may not show interaction with base Java classes. The main purpose is to communicate how components of this software interact.

The ModelServer (main) class is responsible for handling the program flow. On startup, it enters an infinite loop, listening for connections on port 80. Every time a connection is made, it instantiates a new HandlerThread object, passing in the socket on which the connection was established and the command line arguments. It then calls run on the HandlerThread object, spooling up a new thread to handle the request so that it can go back to listening for connections.

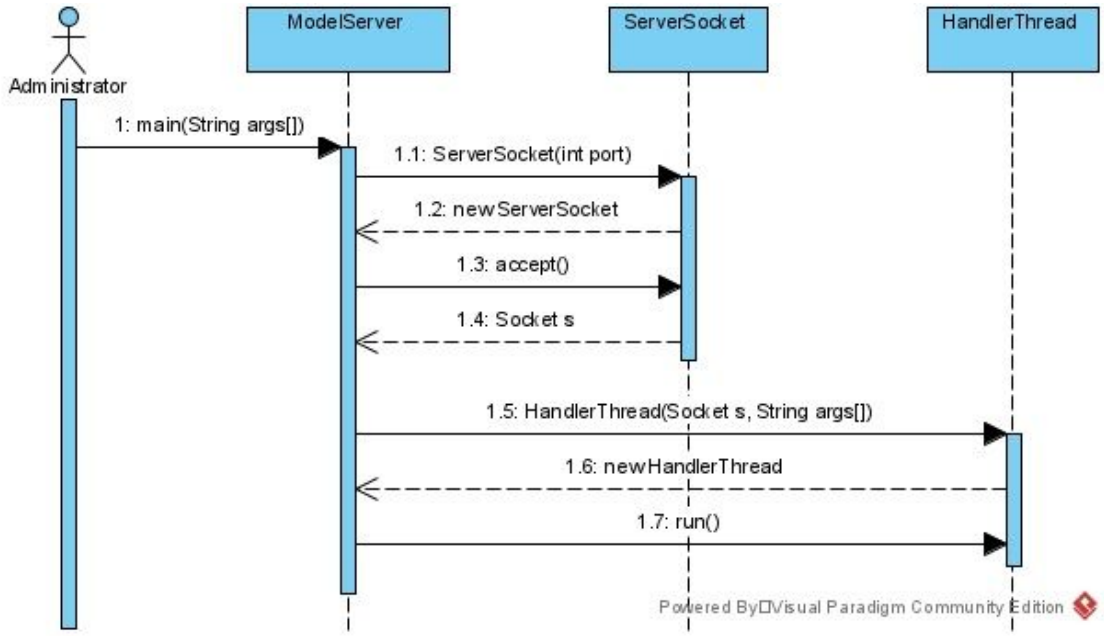

Figure 5 - Main Class Sequence Diagram

The HandlerThread class handles the request. On instantiation, it is passed the socket, which will be used to communicate with the client, and the command line arguments provided on server startup, which will be used to determine the model type. In its run method, it will receive the request from the client and parse it to determine the request type and split out the message body. It will then instantiate a HandlerFactory and ask it to provide a handler. It will then call the run method of the handler it received, which will complete the interaction with the client.

The HandlerFactory class is used to obtain the correct type of handler based on the type of HTTP request. On instantiation, it is passed the socket, command line arguments, and the request data, which it will in turn pass to the handlers it creates. When the HandlerThread wants to obtain a handler, it will call the getHandler() method, passing in the request type as a string.

The HandlerFactory will use the request type to determine the correct type of handler, instantiate it with the required parameters, and pass it back to the HandlerThread.

The IHandler interface is used to provide a means for the HandlerThread to talk to the different types of handlers that the HandlerFactory creates. There are currently two types of handlers: the PredictHandler, which handles POST requests, and the ReadHandler, which handles GET requests.

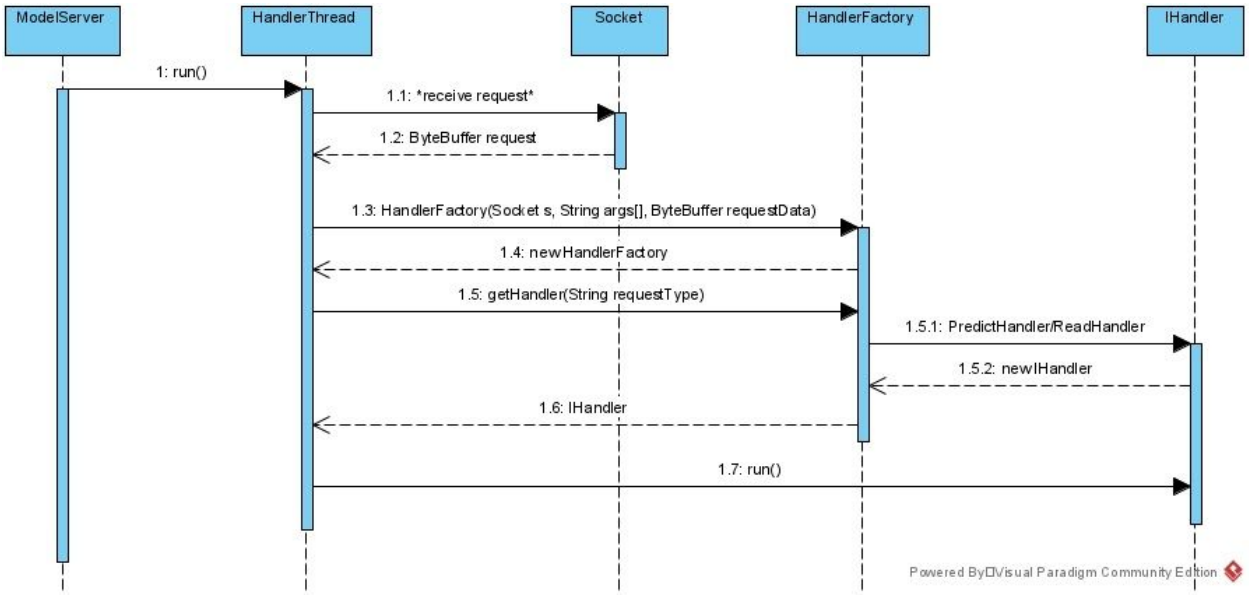

Figure 6 - Thread Handler Sequence Diagram

When the PredictHandler class is instantiated, it is passed the socket, the command line arguments, and the request data from the HandlerFactory. When it's run method is called, it first instantiates a ModelFactory, which it will use to obtain the model. It then extracts the model type argument from the command line arguments and calls the getModel() function in the ModelFactory class, passing in the model type argument. The ModelFactory will create the appropriate model and return it to the PredictHandler. It will then format the request data as an array of Strings and pass it to the model, which will generate the corresponding predictions. It will store these predictions in a newly - created location on the server and generate a reply to the client, which will contain the location of the predictions.

The ModelFactory class is used to obtain the correct model. When the server is first launched, there is a required command line argument which will determine the model type. This argument is passed to the ModelFactory when the getModel() command is called. It will use this argument to determine which model to load from the disk. As mentioned, models are pre - trained and the parameters are stored in a JSON file, meaning the model does not have to be re - trained, it is simply loaded. It will pass this model back to the PredictHandler class so that it can generate predictions.

The IModel interface is there to ensure that the PredictHandler can use different models seamlessly. It ensures that every model has an appropriate predict method which will be called by the PredictHandler in its run method.

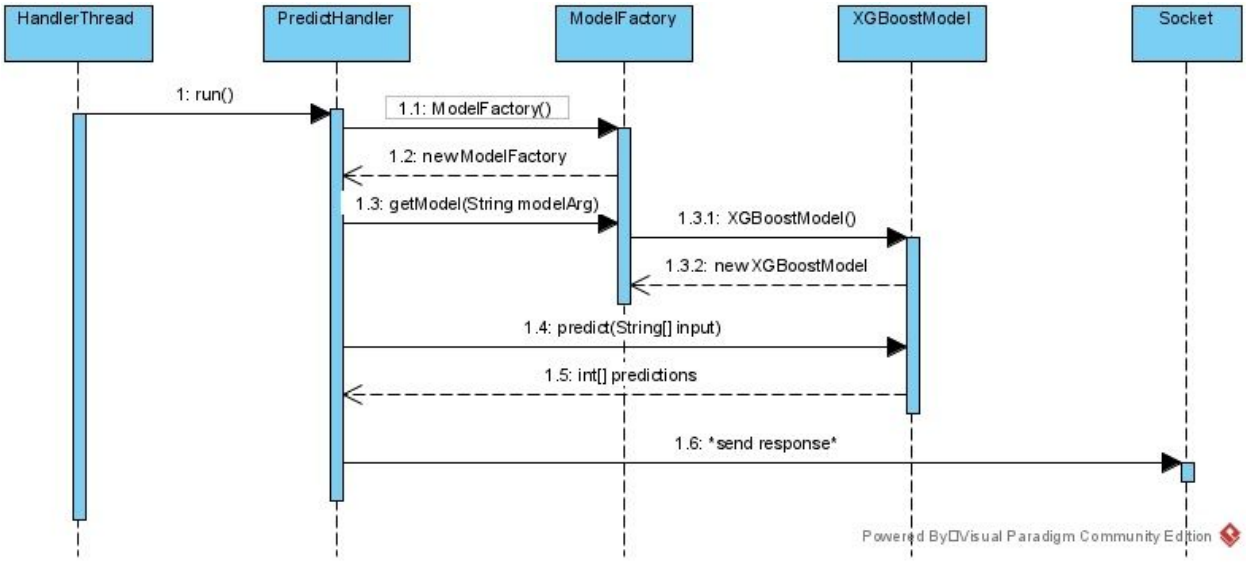

Figure 7 - Predict Handler Sequence Diagram

The only concrete model currently implemented is the XGBoost model, it was chosen because it gave the best performance. Each model will be unique in its implementation, but will contain the predict method prescribed by the IModel interface.

When the ReadHandler class is instantiated, it is passed the socket and the request data. When the run method is called, it will extract the resource location from the request data, read it, and reply to the client with the request resource (the generated predictions). The sequence diagram would look like a simplified version of the PredictHandler diagram above.

This software uses the strategy and factory patterns to ensure that the code remains as extensible as possible, ideally making future evolutions of the software seamless. The two anticipated areas of change are:

1. Responding to new types of requests
2. Including different models (potentially for each request)

As such, both of these tasks were defined as interfaces and made as abstract as possible. To handle new types of requests, one simply has to write another handler class which implements the IHandler interface and modify the HandlerFactory to include the logic which would deal with the creation of this new type of handler. To provide additional models, one would have to store the parameters as a JSON file on the backend, create a new class implementing the IModel interface, and modify the ModelFactory code to handle the new creation logic. On server startup, the correct model has to be specified as a command line argument.

What this architecture allows us to do in the future is to specify the model not on server startup, but as part of the request. This would mean each request to the server can ask to use a different type of model, which would make the system much more flexible.

User Interface

The user interface will look as follows. Please keep in mind that the different colours are there only to show different types of content on the UI.

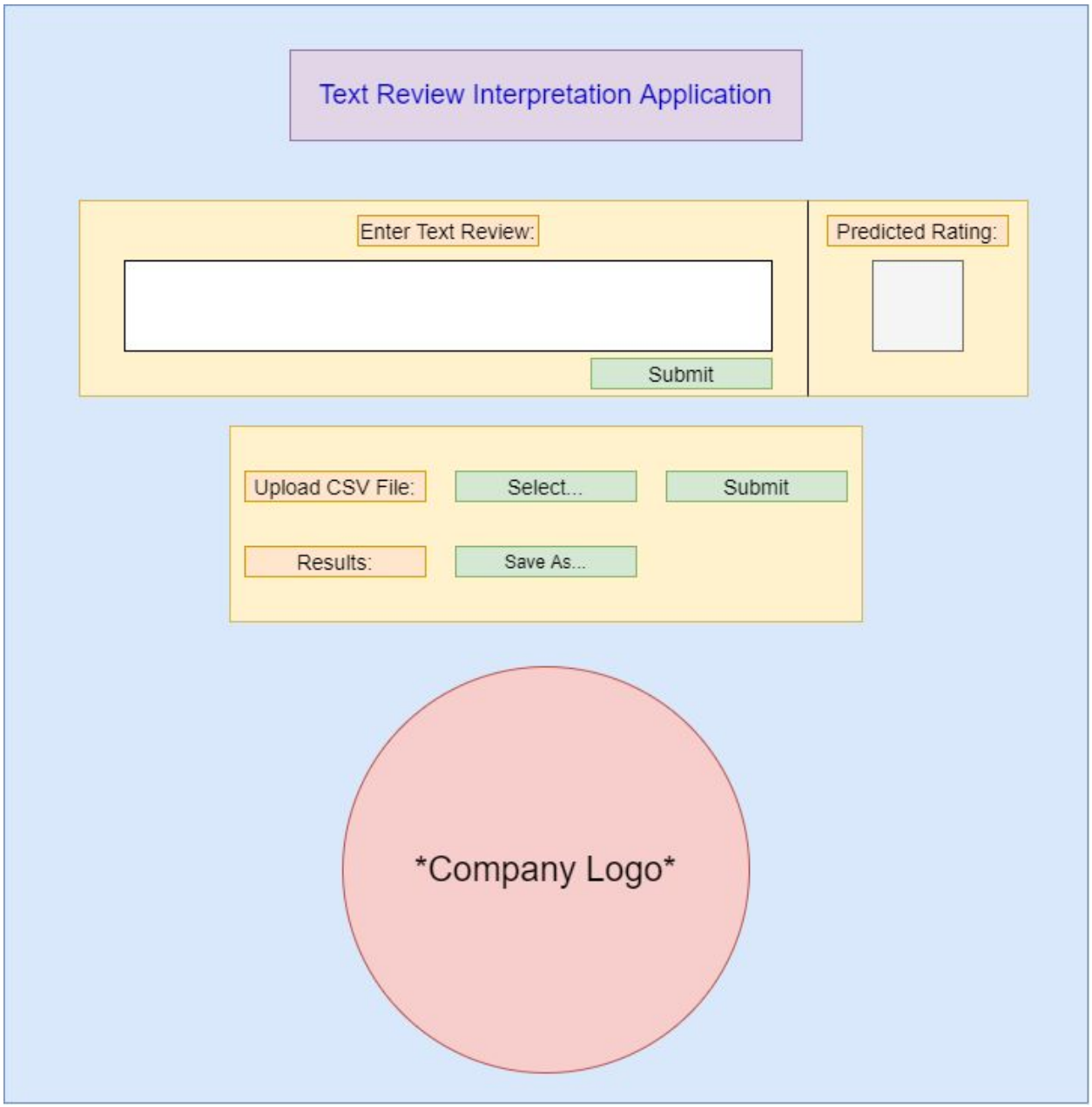

Figure 8 - User Interface Sketch

Although this is a rather simplistic view and can be improved upon later, it includes all necessary elements to make this a working system. The UI will be further refined through consultation with the client.

Machine Learning Model

We chose to consider this problem as a classic sentiment analysis problem, where the goal is to predict whether each review is positive or negative. The original dataset was transformed as follows: each rating was converted to either 0 or 1 by rounding ratings that are greater than 0.5 to 1 and less than or equal to 0.5 to 0. To decide on the best model, we are considering three key areas: text pre - processing, text representation, and the type of model used. We are also considering novel approaches which do not require pre-processing, namely VADER.

Text pre - processing refers to all the steps taken to modify the original text. These include removing stopwords, capitalizing/uncapitalizing all letters, removing special characters, and other operations. The pre - processing used in the initial model was in our opinion too aggressive, and removed too much context for the text to remain useful. In some cases, we shortened down the list of stopwords to allow for negations and other words which could indicate a negative review. We also considered adding to the list of stopwords the words “full” and “review” as they appeared at the end of every review.

Text representation or text analysis refers to how a given text is transformed to a feature vector on which a model can be trained. The initial model used Latent Dirichlet Analysis, which creates a set of topics and represents each text as a vector of percentages (adding up to 1) corresponding to how related the text is to each of the created topics. Another approach is to use N-Grams, which are a contiguous sequence of N items from a text. Essentially, this means tokenizing a text and creating a vector where each element corresponds to the number of times an N-Gram appears in that sentence.

The model itself refers to the machine learning approach used to build the classifier. There are many famous approaches, including neural networks, generalized linear models, ensemble methods (random forest, boosting, etc), and trees. We are considering several different models to determine which one works best in combination with the aforementioned operations. There are also novel approaches which do not require the traditional pre - processing and text representation steps outlined above.

The approach we are considering that does not require pre-processing and text representation is VADER (Valence Aware Dictionary and sEntiment Reasoner), which is a lexicon and rule-based sentiment analysis tool that is specifically attuned to sentiments expressed in social media developed with MIT license. Using the tool designed for social media, we are considering its use for analyzing the sentiment of user reviews, which may have similar characteristics. VADER uses a compound score, which gives a score for each word in the lexicon and is then normalized between -1 (extremely negative) and 1 (extremely positive). This is the useful metric

for sentiment analysis of smaller word groupings such as sentences - perfect for our analysis of reviews.

The compound scores noted above generally have a positive, negative and neutral threshold for scoring. In our analysis, we are removing the neutral threshold to ensure we come up with a 0/1 score similar to our other models. This increases the accuracy of the model, as we were not considering the neutral reviews, rather just positive or negative.

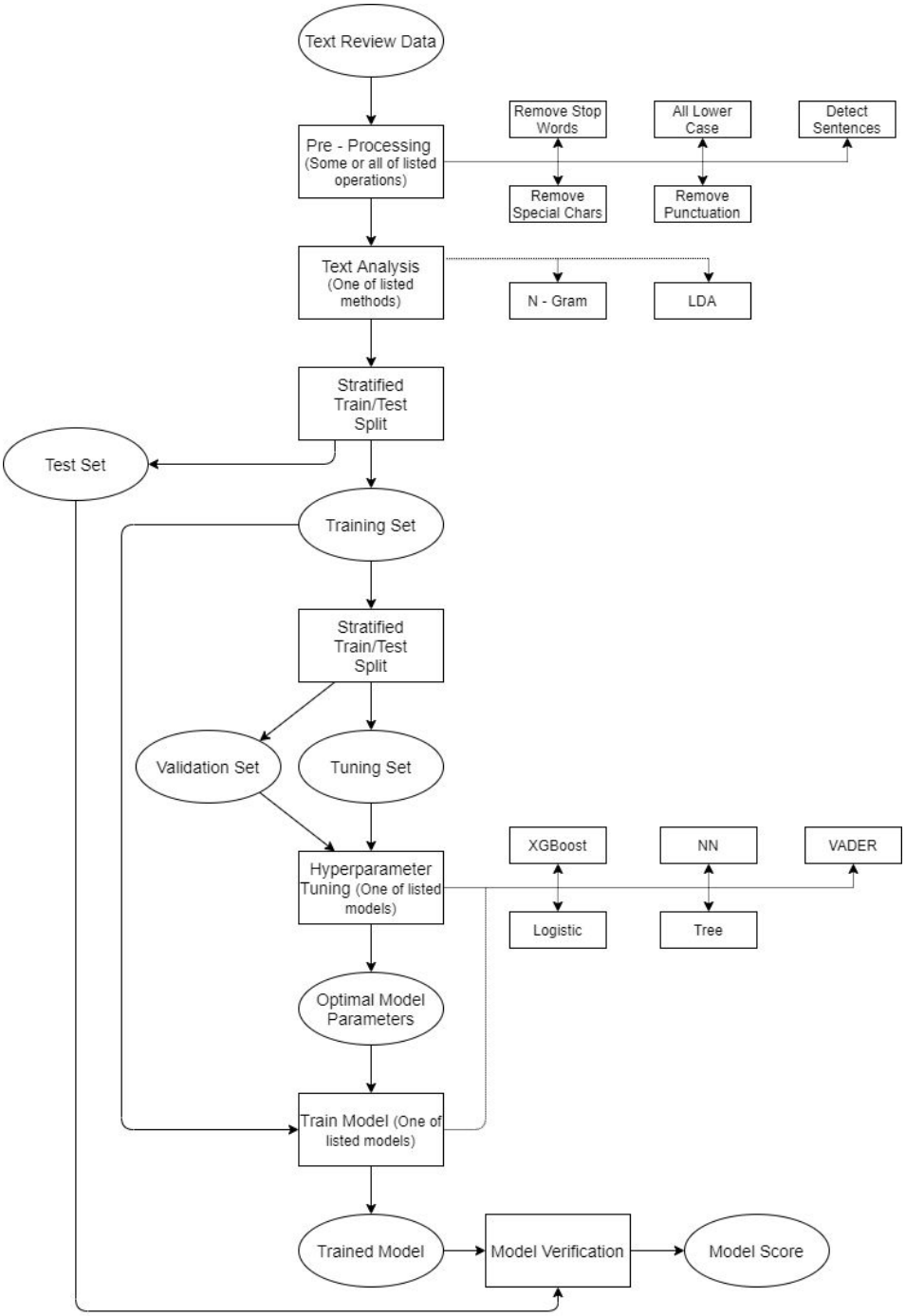

Figure 9 - Machine Learning Pipeline

The flow chart above shows the machine learning pipeline we developed to determine the best model. It starts with the text reviews, which then go through pre - processing steps and are represented either as N - Grams or through Latent Dirichlet Analysis. We then use a stratified train/test split to generate the training and test sets, and again use a stratified train/test split on just the training set to generate a tuning and validation set, which are used to tune model hyperparameters. Once the optimal hyperparameters are chosen, the model is trained on the entirety of the training set and validated against the test set to come up with a final accuracy score. If an approach does not require a given step, it is simply skipped.

Implementation and Test Plan

We see the implementation and testing of this product occuring in several phases. The first step is to finalize the model. In order to speed up implementation and testing of the model itself, it will initially be built out in Microsoft's Azure ML Studio, which lends itself very well to the kind of rapid prototyping required for training machine learning models. Once the model is finalized in Azure, it will be tested to ensure that all model - related requirements are met. These include FR1 to FR5 and PR1 to PR2. The use of Azure ML studio is actually mandated by the client as per ImpR3.

Once the model has been finalized, it will be exported into a file to be later read in by the backend software. At this point, implementation of the hosting software will commence. Initially, a test lab will be built out to include a frontend server, and backend server, and an active directory, as well as a single commercial - scale router to simulate a small enterprise network. The software will then be developed, deployed on the test build, and internal (alpha) testing will be conducted to ensure that the remainder of the requirements are met.

Following the completion of internal testing, news servers will be procured and staged with the finalized software. The original test build will remain untouched, so that future iterations of the software are not tested in production. They will then be deployed to a single site of the client's choosing and a set of employees will be identified for client (beta) testing. The beta testing will be managed by a quality assurance engineer from our team. Once beta testing is complete, full scale roll - out can occur at each identified client site. Although the build is replicable, small integration steps such as IP addressing and DNS integration will have to occur at each site separately.

Support Plan

The deployment will be supported by our team for the duration of the support contract. All requested changes will be tested on our internal test build before undergoing a phased roll - out into production. There are several types of changes we see happening on a regular basis:

1. Retraining of the model: the model will be retrained and exported as usual. The file containing the old model will then be backed up and replaced with the new file. The server does not need to be restarted as the model is read on each request, and all new requests will simply use the model read from the new file.
2. Addition of a new model: should an entirely new model be required, it will be trained and exported as usual. A new class extending the IModel interface will be created, and the logic in the ModelFactory class will be modified to accommodate this new model. The server will then need to be restarted with the new model passed in as the “model type” command line parameter.

There may also be more large - scale changes to include the addition of new functionality. Some changes, such as the addition of new request types or allowing for the use of a different model with each request are simpler since the software was designed with those potential changes in mind. Others may be more complex and may require a re - design of the software.

The hardware will be cycled as it approaches end - of - life and original equipment manufacturer support is no longer available.

Other Elaboration Phase Deliverables

The other deliverables as part of this phase include:

1. Github Repository - has been created and an invite was sent to our professor.
2. Project Management Framework - a Trello board has been created and is being used to manage the project, an invite was sent to our professor.
3. Project webpage - the webpage has been created and is being hosted on Github.