# ENSF 609/610
# Team Design Project in Software Engineering I & II
# Midterm Report



**Group Members:**
Deylin Yiao, 10129370
Kody Kou, 30002709
Zach Frena, 10150373

# Table of Contents

## Abstract

The energy-production industry requires consistent monitoring of operations and facilities to optimize efficiency and prevent profit-loss, mechanical failures, or technical delays. Machine learning platforms further enhances the monitoring by predicting and notifying operators of any real-life events that could cause upsets during operation. An existing set of machine learning tools have been developed by IBM to assist clients in generating warnings and notifying them within reasonable time to take preventative action. However, the current warning and upset recordings currently cannot be tracked in terms of performance as there are no tools to convert the recordings into a meaningful visual representation of the ML performances.

This capstone project comprises of a responsive web application that the internal Data Science team can use to gain insight and clarity on the health of the machine learning model system. Provided to the end-user are a variety of key performance metrics (KPIs) and statistical values directly related to the machine learning model outputs (e.g. precision, recall, F1-score, etc.). In addition, charts, graphs, tables, and visual indicators will assist the Data Science team in making better and more well-informed decisions to provide value to their clients.


## Introduction

Asset operators in the energy-production industry are always striving to find the balance between minimizing operational costs and maximizing efficiency, reliability, and production. Avoiding unplanned shutdowns, reducing downtime, and improving reliability is extremely important in these high-risk environments where small changes can have a massive impact. Effective maintenance scheduling and load forecasting is key to success for the operations team. To be the most effective, asset operators need to be aware of the facility conditions 24 hours/day for 365 days/year. In addition to this, the complexity required to optimize these systems drastically increases as the processes are continuously interconnected site wide. Humans are unable to ingest, analyze, and act on these datapoints due to the sheer volume and complexity, resulting in missed opportunities.

Digital platforms and monitoring systems play a huge role in helping asset owners pinpoint complex problems and take quick action to prevent unfavourable outcomes. Utilizing digital recording technologies, data science, and machine learning techniques, the overall costs associated with asset monitoring are also reduced. There is also the opportunity to collect production data and generate valuable insights in the long-term. These insights can be used to positively impact asset performance, improve quality, and increase reliability— Therefore, summarizing and presenting this real-time data visually using easy-to-understand metrics ensures that the end-users can make informed decisions that will dramatically impact business performance.

Due to the dependence and trust that the operation teams have on these digital platforms, it is imperative to continuously validate and optimize the underlying models used for decision making. Using the right digital tools can ensure that the prediction/modelling platform achieves its intended purpose with a high level of accuracy.

## Motivation, merits, significance, scope

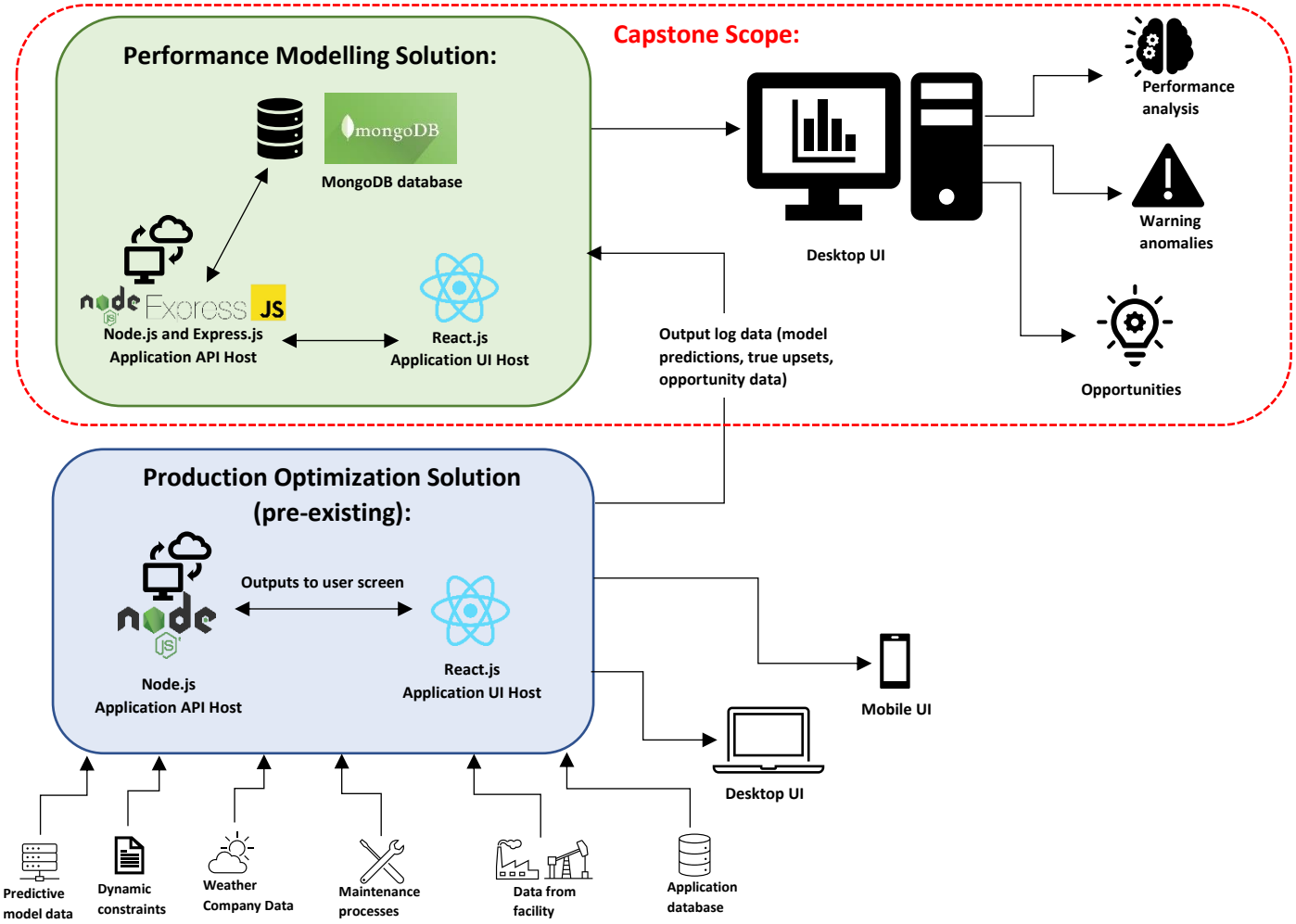
IBM has developed a machine learning platform alerts the operators when oil and gas production issues arise, such as when various on-site equipment reaches capacity limits or goes offline. The objective of the capstone project was to develop a dashboard to display the key performance indicators (KPIs) of machine learning models used in this pre-existing observation platform. This monitoring system aids the data science team in visualizing the performance and trends for the machine learning models that are deployed within an AI solution. The developed monitoring dashboard developed also determines if any flag anomalies are present within the currently connected system of assets. Common flag anomalies include excessive flagging or the absence of flags within a pre-specified time-period and can be quickly identified to assist the internal team in uncovering potential issues.

This machine learning KPI dashboard serves as another source of valuable insight that the IBM data science team will hopefully implement into their full technology suite.

In terms of scope, it was decided early on that the project has 3 major components of delivery:

1) Alert on upcoming risks and pin-point areas of immediate next steps using high-impact visualizations.
2) Identify and display performance metrics surrounding model predictions vs. actual asset upsets.
3) Provide insight on opportunities that can improve asset integrity and potentially model performance.

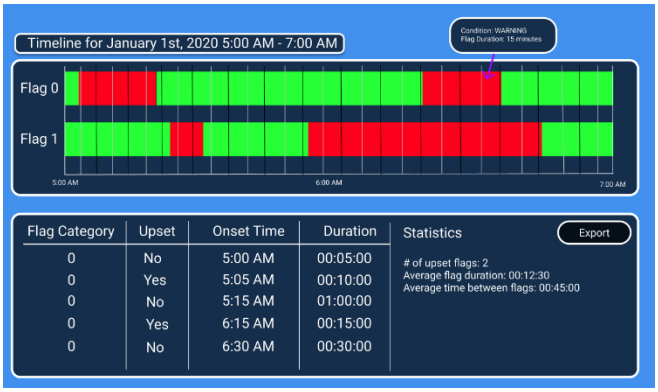
## Conceptual Diagrams

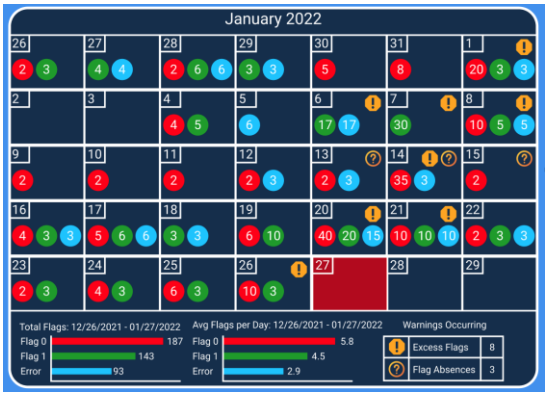


# Background

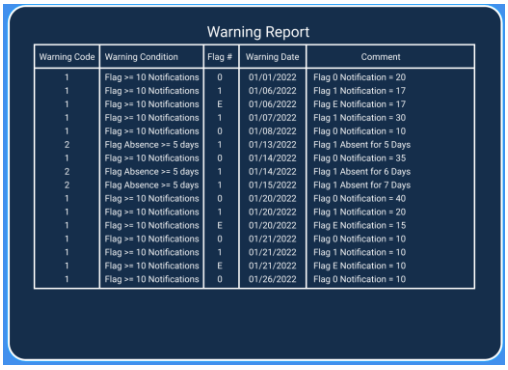
## Overview of what we have learned:

As with most iterative and incremental design processes, we began our dashboard visualization by creating wireframes before any undertaking any actual software development. Our team scheduled stakeholder and SME informational sessions to understand the use-cases, pain-points, and requirements for the proposed solution. We started with building low-fidelity mock-ups using Figma to help map out the shell of the interface and play around with the basic information architecture:

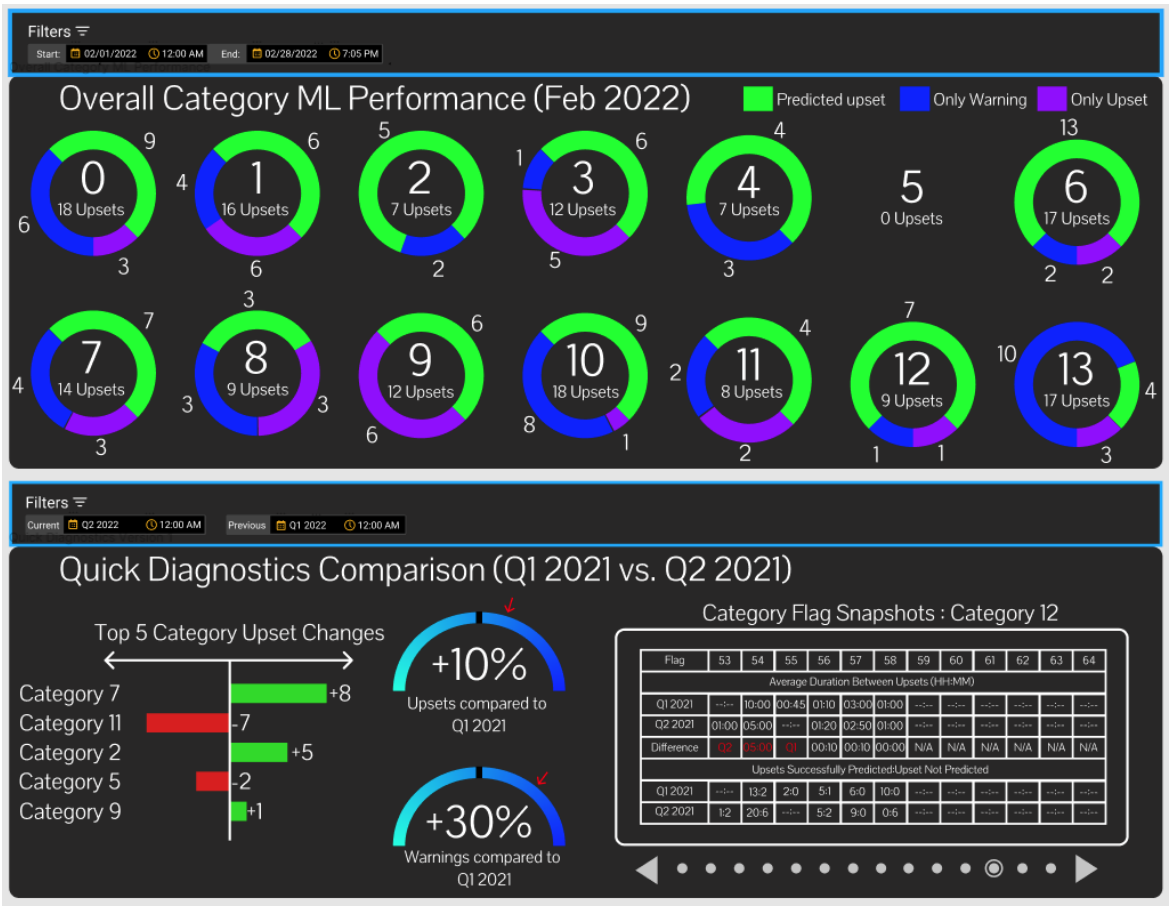




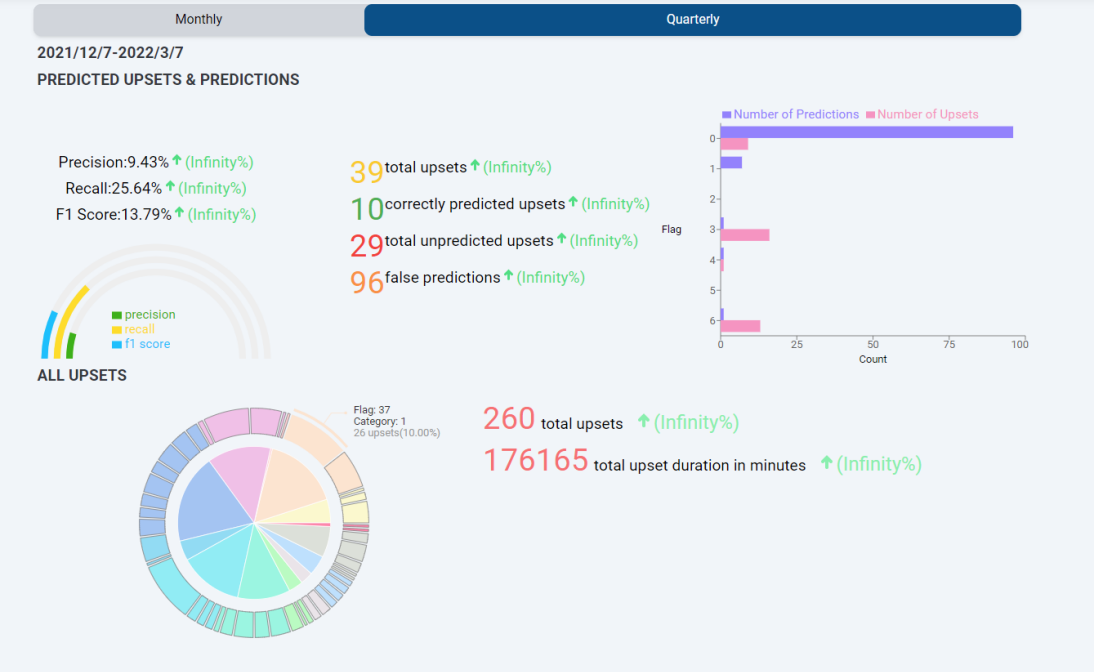


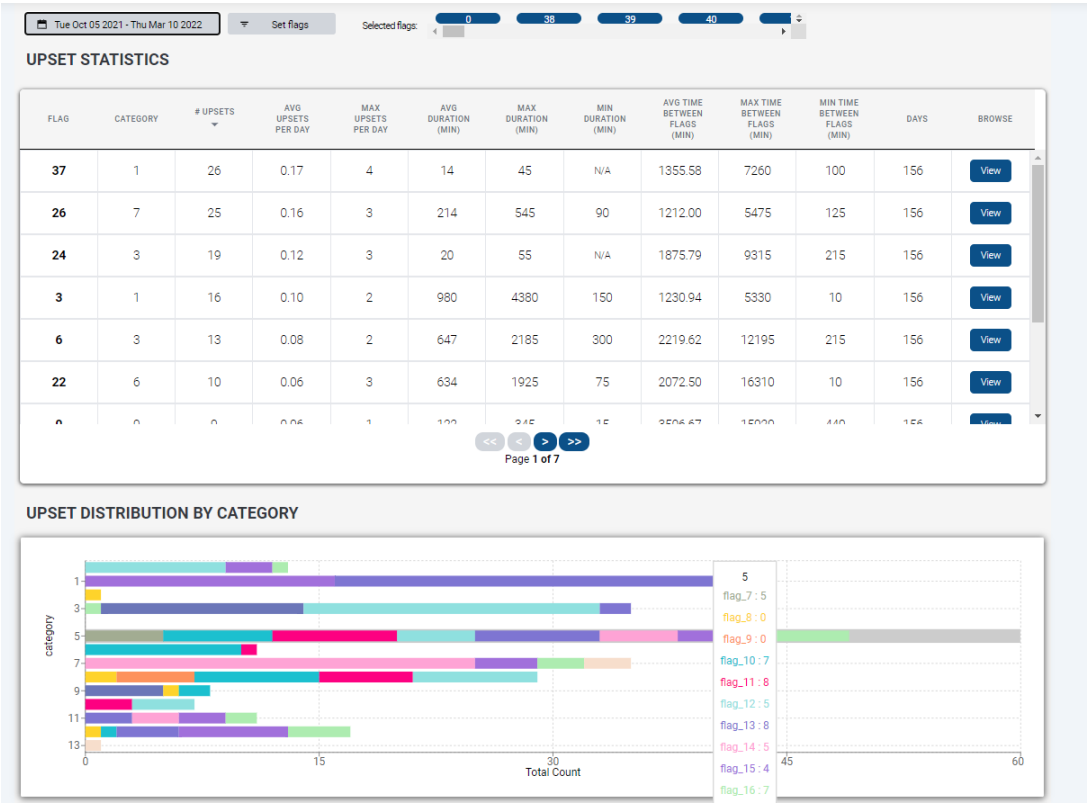

Once the low-fidelity mock-ups were complete, additional feedback sessions were scheduled with the data science team to identify the key metrics destined for the final visualization. The second iteration of the Figma wireframes (high-fidelity) were designed and created to act as a more complete representation of the final solution:


Filters
Start: 02/01/2022 12:00 AM
End: 02/28/2022 7:05 PM
Overall Category ML Performance (Feb 2022)
Predicted upset
Only Warning
Only Upset
0 18 Upsets
1 16 Upsets
2 7 Upsets
3 12 Upsets
4 7 Upsets
5 0 Upsets
6 17 Upsets
7 14 Upsets
8 9 Upsets
9 12 Upsets
10 18 Upsets
11 8 Upsets
12 9 Upsets
13 17 Upsets
Filters
Current Q2 2022 12:00 AM
Previous Q1 2022 12:00 AM
Quick Diagnostics Comparison (Q1 2021 vs. Q2 2021)
Top 5 Category Upset Changes
Category 7 +8
Category 11 -7
Category 2 +5
Category 5 -2
Category 9 +1
+10%
Upsets compared to Q1 2021
+30%
Warnings compared to Q1 2021
Category Flag Snapshots : Category 12
Flag 53 54 55 56 57 58 59 60 61 62 63 64
Average Duration Between Upsets (HH:MM)
Q1 2021 --:-- 10:00 00:45 01:10 03:00 01:00 --:-- --:-- --:-- --:-- --:-- --:--
Q2 2021 01:00 05:00 --:-- 01:20 02:50 01:00 --:-- --:-- --:-- --:-- --:-- --:--
Difference Q2 05:00 Q1 00:10 00:10 00:00 N/A N/A N/A N/A N/A N/A
Upsets Successfully Predicted:Upset Not Predicted
Q1 2021 --:-- 13:2 2:0 5:1 6:0 10:0 --:-- --:-- --:-- --:-- --:-- --:--
Q2 2021 1:2 20:6 --:-- 5:2 9:0 0:6 --:-- --:-- --:-- --:-- --:-- --:--


Filters
Category: Category 0
Flags: Flag 39
Start: 02/01/2022 12:45 AM
End: 02/28/2022 7:05 PM
Category ML Performance (Feb 2022)
Predicted Upset
Only Warning
Only Upset
0 18 Upsets
Upset Compositions
Flag 0
Flag 38
Flag 39
Flag 40
Average Warning Time before Upset
Upset / Warning Ratio
Upsets
Warnings
39
38
0
40
Category Quick Diagnostics - Category 0
Predicted Upset
Only Warning
Only Upset
Select time frame: Current Month Current Quarter Custom
Select comparing time frame: Last Month Last Quarter Custom
+12
More upsets than previous month (18 vs. 6)
-6mins
Less minutes between upsets (50mins vs. 44mins)
+4
More only warning errors occured (6 vs. 2)
-13d4h3m
Previous month had longer max absence time duration (7d4h2m vs. 20d8h5m)
+25%
Increase of predicted warning / total (0.5 vs. 0.4)
+0.2 u/d
More upsets/day than previous month (1.2 u/d vs. 1.0 u/d)
+3
More only upset errors occurred (3 vs. 0)
39 vs. 38
Flag with the most upsets for each time frame (39 - 9 upsets vs. 38 - 4 upsets)

After receiving positive feedback from the SMEs and data scientists, we began the front-end development process using JavaScript in a React framework. The current state of the solution in the development environment is shown below:



Prior to this capstone project, our group was slightly familiar with building JavaScript-based applications using a React framework. However we had only had prior experience with SQL databases, yet we chose to implement our database in NOSQL (MongoDB specifically) because our incoming data was in JSON format. In our previous coursework we had used Spring Boot (in Java) for our backend framework, but we decided to use Express.js to match IBM's existing framework for their solution. Recharts is a React-based charting library that we used to build our visualizations (charts, graphs, and tables). None of us had experience with Recharts prior to this project so we've all needed to read into the documentation so that we could utilize the tools effectively.

# Methodology

## Define challenges and discuss solutions in detail

One of the main challenges we originally faced (at least in the initial stages) was the lack of abundant and long-term data. In addition, due to client confidentiality, extra data-anonymization steps were required for the production data before it could be utilized by our team. There was only a small amount of data (less that 2 months worth) that we were able to use as an input for our dashboard/monitoring solution. We had to be careful with our design, as this posed the risk that the created dashboard would only display a sub-section of the total issues that may be present in the long-term dataset. These concerns were mitigated by having frequent and consistent meetings with the data science team to gain feedback on which features, and model metrics are necessary. It was also determined that if the data for a particular situation doesn't exist during the development process, mock data can be generated to take its place.

On the frontend side of the application, our team needed to get familiar with the Recharts library quickly in to create the data visualizations. Recharts is relatively intuitive and easy to work with, but there are only a few examples of its usage online, so we had to heavily rely on the documentation. The charting components in ReCharts are highly customizable and were able to meet our requirements, however with the lack of implemented examples it was difficult at first to achieve our design goals.

Most of the challenges we faced in this project emerged during the backend development process, as it was much more complex than the frontend and we were less familiar with the data ingestion process. Firstly, the input data comprised primarily of nested JSON objects-- it took us a while to parse through this data effectively so that the relevant information could be extracted and formatted. While the upset or warning recordings objects had copious amount of information stored, ultimately the decision was made to select five crucial parameters required to group warnings/upsets and calculate statistics: the flag name, the category the flag belonged to, status of the flag (I.e., warning/upset/normal status), the time the warning/upset was created at, and any errors that occurred during a recording. We had originally pulled the pertinent data into arrays to perform statistical analysis, however the loading times were in the time range of 20-30 seconds (far from acceptable from a real-time dashboarding perspective). To solve this issue, we re-evaluated how we ingested the data and opted instead for an aggregation query, which allowed us to perform the grouping operations (sums, averages, counts, etc.) on the data *as we were pulling it* from the NoSQL database. This resulted in API calls that were much more acceptable (i.e. 2 to 3 seconds, or a 90% decrease in response time). The next challenge we faced was determining which upsets corresponded with the associated prediction. The entire point of the original monitoring system was to identify true positives (i.e. where every upset was predicted correctly), however with all machine learning algorithms there is always a chance of a false positive (i.e.

a failure was predicted when no upset occurred) or a false negative (i.e. an upset occurred with no prediction/warning). At first, this time series data was difficult to work with, and this complexity was compounded when different warning flags were assigned different prediction windows for their corresponding real-life upsets.

Most of the samples in the upset and warning datasets were representative of five-minute intervals, which added into the complexity of determining the duration of a single upset or warning. Warnings or upsets were recorded when a threshold is achieved and proceeds to alert the system an upset or warning has occurred. However, the upset/warning recorded had no predefined grouping splitting up warning/upset recordings to associate to a specific warning or upset. To group warning and upsets recordings as a single entity, flag warning or upsets were compared from the current recording timeframe to the next recording timeframe. For example, if the difference between the current timeframe and the next timeframe is a 5 min duration (i.e. no breaks between each five-minute interval recording) then it is determined that both recordings are tracking the same warning or upset, as a single warning or upset can be triggered for hours.

After individual warnings and upsets were determined for each flag, specifications of acceptable windows were given to determine if a warning was sufficiently issued ahead of time before an upset occurred. The main purpose of the warning triggers is tied to machine learning models that will alert the possibility of an upset for the flag. The process of determining if a flag's upset was successfully predicted or not was to compare the compare each upset to the warnings occurring within the same timeframe. Each flag has varying acceptable windows for warnings. For example, flag 0 has an acceptable window of 30 minutes before upset occurrence, whereas flag 1 has an acceptable window of 12 hours before upset occurrence. If a warning does occur within the upset's acceptable timeframe, then the upset will be marked as a predicted upset. However, upsets occurring outside of any upset windows would be designated as a false alarm, while upsets without warnings are labelled as unpredicted upsets. We would further use these designations later to calculate and determine the machine learning performance metrics, such as precision, recall and f1-scores.

Additional statistics were determined from the warnings and upsets for each flag. As a scope of the project, the warnings and upsets can be further analyzed to determine other statistical metrics, such as total count, average duration of a warning/upset, frequency of warning/upset occurrence, etc. The endpoint to display these statistics were generated and displayed as tables separately for upsets and warnings.

Finally, the backend architecture was also heavily deliberated upon during the development process. Since the dashboard requires such a large variety of data, numerous endpoints were required to provide a comprehensive dataset that could be used for visualization purposes. Keeping all these endpoints organized in the backend was imperative, so we chose to modularize as many components and functions as possible for high maintainability, improved readability, and software reuse.

## Design Documents, Solution Details, etc.

The four key technologies that make up our stack are MongoDB (acting as our NOSQL database), Express.js (the back-end web application framework), React.js (client-side front-end JavaScript framework), and Node.js (the JavaScript web server). This is well-known as a MERN stack and is

commonly used to create a 3-tiered architecture for web applications. In addition, we have also implemented libraries like Recharts to build the visualizations and dashboards for the end-users.

We made the decision to implement the NOSQL database format for several reasons. Firstly, the machine learning platform that monitors the physical asset and generates predictions provides a large amount of data in JSON format and each file set often differs from the next. The large amount of flexibility that NoSQL databases offer was greatly needed to manage the ever-changing format of the input data. Secondly, NoSQL databases have the ability to scale very effectively, which is something we will definitely need to take advantage of as the input JSON data we receive is on the magnitude of 0.5-1.5GBs, containing thousands or even millions of lines of code.

## Backend Design

In the initial stages of backend development, a group decision was made to ensure that similar technologies to the pre-existing Production Optimization Solution application were utilized. Since the backend of the pre-existing application was developed on Node.js, we have followed suit. Out of the different backend frameworks available, Express.js was the framework of choice.

Express.js makes Node.js REST API development quick and easy, all group members were able to pick up Express easily without experience. Developing the backend on Node and Express also allows our group to utilize the vast number of libraries available to Node.

Since we have selected MongoDB the database of choice, we set up data schemas and modeled our application data with the assistance of the Mongoose library, which is an Object Data Modeling (ODM) library that includes validation, type casting and other functionalities to enforce a specific schema at the application layer.

The backend REST API is developed with the MVC architecture in mind, where the models are the Mongoose schemas, while the controllers are the business logic associated with processing data objects. Some other libraries such as passport.js, helmet.js, will be utilized to add additional functionalities such as authentication and network safety.

While developing the methods for the controllers in the backend, we have kept in mind the large amount of data that requires processing and avoided all methods that have a time complexity of $O(n^2)$ to greatly reduce processing times.

## Frontend Design

We have decided to develop the frontend in React due to all group members having some experience and because the pre-existing application was developed in React as well. On top of this, React allows us to build reusable components and utilizes a virtual DOM which improves performance. The frontend is developed with modularity in mind, meaning that components are also developed in a way that allows them to be reused and reordered easily. Multiple asynchronous calls are made by individual components instead of one large API call on each page to improve program performance and speed, this also provides the program with some resistance to errors (i.e. if one API call fails, the rest can still succeed).

Some of the libraries we have utilized during the process of frontend development include:

- TailwindCSS: Tailwind is our CSS framework of choice mainly due to how easy it is to work with since it is the only CSS framework which has intelliSense support on VSCODE. Tailwind is also highly customizable and allows users to create custom stylistic classes and parameters based on classes provided by tailwind.
- React Query: We used React query to manage the status of API calls, React query helps fetch, cache, and update data in React and therefore mitigates the necessity of reducers such as Redux. It helps to ensure the number of calls made is minimized and calls are only made when necessary (such as when context is changed). Since react query also helps monitor the status of API calls, it also eases the process of developing loading screens.
- React Table: React table was used to develop different tables in the application. Initially, all tables were made from scratch in jsx, however, implementing additional functionality such as sorting, scrolling, and paging turned out to be messy and made too many API calls. Thankfully React Table has sorting and paging functionality built in and is easily accessible. By using both React query and React table in conjunction, the number of redundant API calls is minimized, and data is cached properly.
- Recharts: Most of the plots in our application are developed through Recharts because of the highly customizable nature of the graphing library. This allows each and every component of each graph to be easily customizable.

# Evaluation of work thus far

## Benchmarks

Strictly speaking, because this dashboarding system was built completely from scratch, there aren't many areas where process optimization is available or appropriate. One area that we were able to enhance was the API call response time as data was pulled from the NoSQL database to the frontend. As previously mentioned, we had originally attempted to pull data from the JSON format into JavaScript arrays to perform statistical methods on them, however this took 25-30 seconds if the full dataset was queried. Instead, we utilized MongoDB's aggregation query which allowed us to perform grouping methods like sum(), count(), average(), etc. on the data simultaneously with the API calling mechanism. This resulted in API response times 90% faster than previously evaluated, bringing the database-to-visualization time down to 2-3 seconds.

Achieving functional-based requirements became the main method of benchmarking progress as we continued throughout the semester. The SMEs and Data Scientists at IBM were consulted on a weekly basis and were able to provide insight on what features and components they had in mind. The table below shows a list of the requested features and the corresponding completion status.

| Parent Component | Child Component | Status |
|---|---|---|
| **Landing Page** | ML model metrics (overview) | Completed |
| | Predicted upset statistics (overview) | Completed |
| | Upset statistics (overview) | Completed |
| | Detailed upset statistics | Completed |

| | | |
|---|---|---|
| **Upset Page (overview)** | Categorized upset distribution | Completed |
| | Total upset count | Completed |
| **Upset Page (individual flag)** | Detailed upset statistics | In-progress |
| **Prediction Page (overview)** | Detailed upset statistics | Completed |
| **Prediction Page (individual flag)** | Metrics/scores | Completed |
| | Prediction statistics (KPIs) | Completed |
| | Confusion matrix | Completed |
| | Daily Upset and Prediction counts | Completed |
| **Opportunity Page** | Opportunity counts and metrics | In-progress |

The main difficulties that were encountered in the backend was to filter the copious JSON files uploaded into the MongoDB servers and determine useful information to be used to create visual representations of the machine learning model performances for IBM monitor team. The benchmark for the backend is to filter the information and generate endpoints for the frontend implementation. As such we have divided the endpoints into three categories: upsets (physical events that occurred for the individual flag), warnings (predictions of the occurrence of upsets for flags), and machine learning performance which combines elements from both upsets and warnings. The following table establishes the endpoints generated, the status of the endpoint generation, and status of frontend implementation.

| Endpoint | Purpose of Endpoint | Endpoint Status | Frontend Implementation Status |
|---|---|---|---|
| **Upsets (Status Routes)** | | | |
| getAllUpsets | Returns an array of all flag objects with UPSET status | Completed | Not in use* |
| getFlagList | Returns an array of all flag objects containing only flag name parameter with UPSET status | Completed | Not in use* |
| getFlagComposition | Groups flag upset recordings into upset entities and generate individual and total statistics for each selected flag and given timeframe | Completed | Implemented |
| getStackedBarChart | Derived from getFlagComposition endpoint, formats the upset statistics into an acceptable format for ReChart stacked bar plot | Completed | Implemented |
| getStatistics | Derived from getFlagComposition endpoint, formats the upset statistics into an acceptable format for React Table | Completed | Implemented |
| getFlagBarChart | Derived from getFlagComposition endpoint, formats the upset statistics into an acceptable format for ReChart individual bar plot | Completed | Implemented |

| Warnings (Prediction Routes) | | | |
|---|---|---|---|
| getAllPredictions | Returns an array of all flag objects with WARNING status | Completed | Not in use* |
| getFlagList | Returns an array of all flag objects containing only flag name parameter with UPSET status | Completed | Not in use* |
| getFlagComposition | Groups flag warning recordings into upset entities and generate individual and total statistics for each selected flag and given timeframe | Completed | Implemented |
| getFlagStatistics | Derived from getFlagComposition endpoint, formats the warning statistics into an acceptable format for React Table | Completed | Implemented |
| getErrors | Returns an array of all flag objects with an error message | Completed | In-progress |
| **Machine Learning Performances (Performance Routes)** | | | |
| getBothComposition | Combines the getFlagComposition endpoints from status and prediction routes | Completed | Implemented |
| getStatistics | Derived from getBothComposition endpoint, generates upset/warning statistics and formats into an acceptable format for | Completed | Implemented |
| getColourBandStatistics | Derived from getBothComposition endpoint, formats performance statistics to be put into a colour band time chart | Completed | In-progress |

*All endpoints with the frontend status of not-in-use were generated as a baseline and for testing purposes.

## Test Plan/Results

In terms of endpoint development, the endpoints were tested using Postman to verify the validity of the endpoints, testing the ability to select certain flags and timeframe for certain endpoints while recording the average response time for each endpoint. The average response times are based on queries for all flags, and within a timeframe with all available data (26 days). Any large discrepancies in our response time between upsets, warnings and performances is likely due to the availability of flags for each route. Currently, there are 71 upset flags, 11 warning flags and 11 performance flags. The additional number of flags requires more data filtering for the upset endpoints and therefore requiring a longer response time. The output of the endpoints is a major factor in increasing the response time, as evident for both upsets and warnings in terms of retrieving all upsets/warnings and obtaining the list of unique flags with the warning/upset status. Both methods follow a similar query, the key difference is getFlagList method only returns the flag list and therefore takes less time in terms of data generation and exporting.

| Endpoint | Is Filterable by Flags & Date? | Average Response Time (ms) |
|---|---|---|
| **Upsets (Status Routes)** | | |
| getAllUpsets | No | 600 |
| getFlagList | No | 190 |
| getFlagComposition | Yes | 2,300 |
| getStackedBarChart | Yes | 2,260 |
| getStatistics | Yes | 2,300 |
| getFlagBarChart | Yes | 2,300 |
| **Warnings (Prediction Routes)** | | |
| getAllPredictions | No | 35 |
| getFlagList | No | 15 |
| getFlagComposition | Yes | 30 |
| getFlagStatistics | Yes | 30 |
| getErrors | Yes | 25 |
| **Machine Learning Performances (Performance Routes)** | | |
| getBothComposition | Yes | 300 |
| getStatistics | Yes | 300 |
| getColourBandStatistics | Yes | 300 |

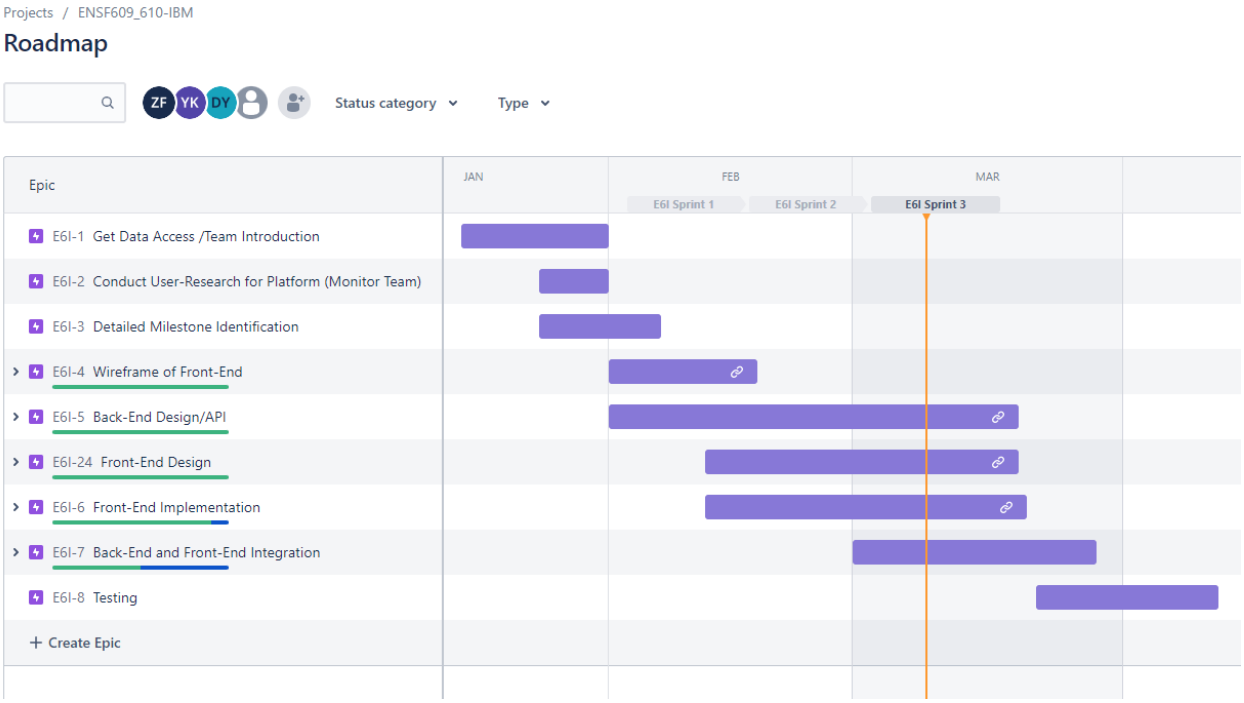
### Recommendations going forward

In terms of code development, with the future addition of opportunity dataset, it is more pertinent the routes and controllers remain well sorted to prevent confusion and inserting all functions into a single JavaScript file. As some of the endpoints are derivations of other endpoints, the fundamentals of the base endpoints may be converted into a more generic method to be used by other endpoints to clean up the code and promote code reuse. Further analysis into aggregation queries of MongoDB or other solutions may be necessary to improve backend response time.

## Plans going forward

### Gantt Chart (remaining deliverables and timeline)

Alongside weekly meetings with the IBM team (and daily meetings amongst ourselves), we have used Jira to maintain an agile software development process. 2-week sprints were used to break up the development cycles into manageable groups of tasks in our application backlog.




## Adjustments to original scope

As of writing, most objectives set out by both IBM and the U of C team have been met. A KPI dashboard containing warning and upset statistics for each individual flag has been implemented and require a few additional graphs (discussed below) to fully cover the initial scope. Progress of fulfilling the scope has been closely monitored, as the U of C team maintains weekly meetings with the IBM team. Both teams have agreed as an extension to the original scope, opportunity dataset will be introduced to parse through and generate statistics and visualization of the opportunity prediction performance. Other considerations discussed was to implement authorization or deploy the software onto a cloud hosting service/platform, with the latter requiring more discussion between teams. Further testing is required as the initially given data were purged of parameters irrelevant to the KPI dashboard but contains a significant amount of data and can cause the input files sizes to range in the megabytes or gigabytes.

## End results and final product discussion

Since most of the scope requirements originally outlined for the projects have been satisfied, IBM has provided an additional dataset for performance enhancing opportunities (also generated by the machine learning model) with the goal to add an additional set of visualizations to the monitoring system. The opportunity dataset is different compared to the upset and warning dataset and the flags within opportunity do not correspond, unlike the warning and upset dataset. As such, discussions with the IBM monitor team would be necessary to gain an understanding of the new set. As of current discussion, statistics for the opportunity dataset will be generated and displayed in a similar manner to the previous datasets.

Additional plots that could be generated for the warning and upset dataset would be determining the number of error recordings per day and on a flag basis, as well as a color band plot depicting the warning, upset and normal periods. Earlier discussion for the colour band time graphs would suggest daily or weekly scale would be beneficial for the monitor team. Error messages do occur for the

warning/upset recording and usually pertains to API request errors, and therefore essential for IBM to determine if warnings/upsets are adequately recorded.

Further testing of the file size input has been discussed in the previous section but will be implemented as a part of the original scope.

# Reflection and Conclusion

## What approaches have worked and what hasn't worked?

One successful approach adopted was the consistent communication within the U of C team and committing to weekly meetings with the IBM team. Initially, bi-weekly meetings were set to meet between teams and gaining constructive feedback on the dashboard progression. While the IBM team can be easily reached via email or text messages, the conversion from bi-weekly to weekly meetings promoted progression as the IBM team were more regularly updated, and the UofC team members were held responsible to showcase progression or discuss concerns or questions on a more frequent basis. Also, since the capstone project only last a duration of 3 months, the bi-weekly meetings would have been inefficient and could lead to delays or mismatched expectations between teams.

## What are the most profound lessons learned thus far?

In terms full stack development, the one source of a significant learning was from brute-forcing the flag filtering required for statistics generation. Without consoling the MongoDB documentation or other team members, the flags were initially filtered inefficiently through the combination of for-loops and if-else statements. As such, filtering all flags over a monthly timeframe led to wait times being half a minute long, until further discussion with other group member and consoling through the documentation led to a more efficient query and response time was cut down significantly. The most important lesson that was learned from this experience was to have regular communications amongst group members and expressing concerns and questions in a relatively quick manner to prevent delays from occurring.

As the team generated more endpoints and components requiring integration, it became very important to have a clean and well-sorted file structure was required to prevent excess function cluttering and file location confusion. The endpoint development initially had routes and supporting functions all within the same JavaScript file. The endpoints have since been divided into two files, one to store the route selection and another to store the supporting functions the route requires. By decluttering and organizing code, further analysis can be done to determine if certain parts of the supporting functions for routes can be reused. For example, a flag list generation was required by most routes developed and the supporting function was placed into a generic controller file to promote code reuse and minimize cluttering.

As the MERN tech stack contained new frameworks and library never used by the UofC team had caused a steep learning curve, the team had to be quick to adapt. Although quick adoption can easily be attained from watching YouTube tutorials or through paid courses, through daily communication between team members we efficiently pooled our knowledge together by discussing developed solutions or expressing different aspects of current issues.

## Are you making any changes to your process going forward?

The current workflow and weekly progression check-in have work well for both teams. Currently, there are no changes to the workflow for both teams, as it has allowed for sufficient feedback and progression reporting within the UofC team and between teams. While the Jira board has help keep the team developing the dashboard in an agile manner, more frequent task creation and reporting would be beneficial in testing and finalizing the dashboard as the end-of-term approaches.