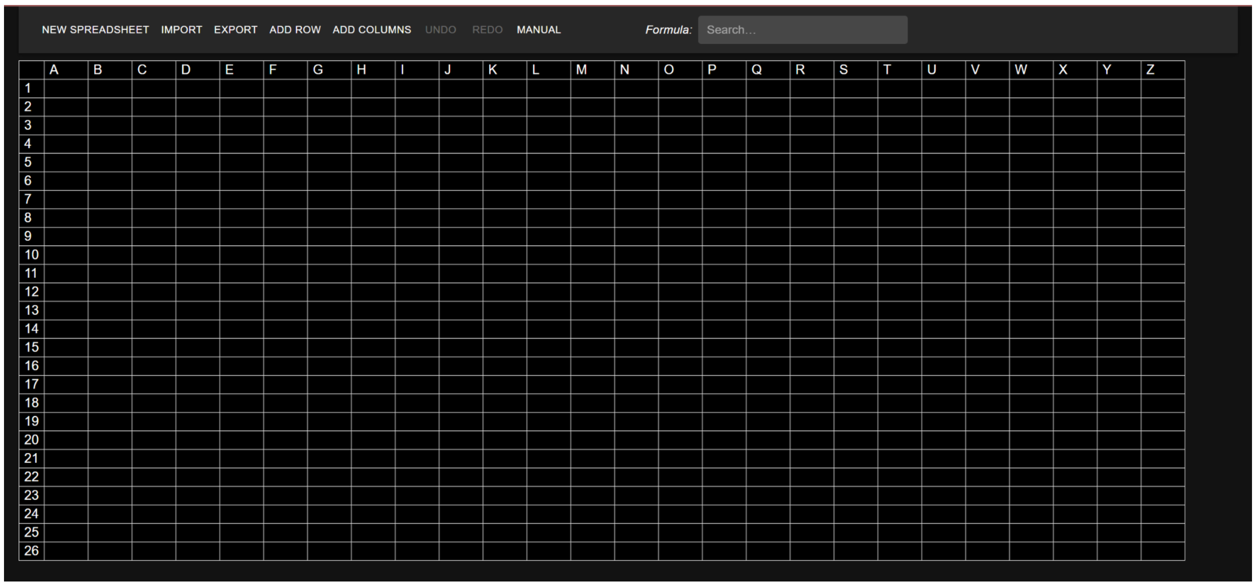
Spreadsheet Application

Team 108

Jonah Nidorf, Peter Benson, Timothy Porter

Summary of system functionality:

Our spreadsheet application offers a wide set of functionality for data manipulation and analysis. We have implemented all of the required features from the project spec, in addition to three exciting new features that improve the quality of life for the user and provide them with even more useful tools for data analysis:



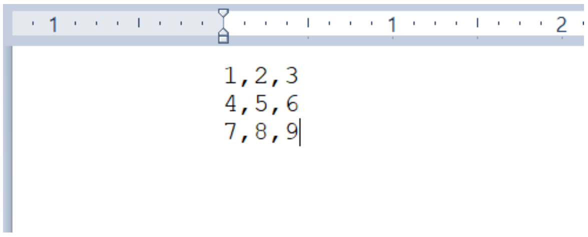
Additional Feature #1: CSV Import/Export - Our app allows for users to import data from local CSV files on their computer, as well as export the data from an active spreadsheet to a CSV file on their hard drive.

To import from a local CSV, click the ‘import’ button on the taskbar at the top of the spreadsheet. This will prompt you to locate a CSV file using your computer’s file explorer. After selecting the file you want imported, you must then press the ‘upload’ button on the taskbar in order to push the imported CSV’s data into the active spreadsheet.

To export your active spreadsheet to a local CSV file, click on the ‘export’ button on the taskbar at the top of the spreadsheet. This will open a window of your computer’s file explorer where you can name and save the spreadsheet to any location on your hard drive. Make sure to save often!

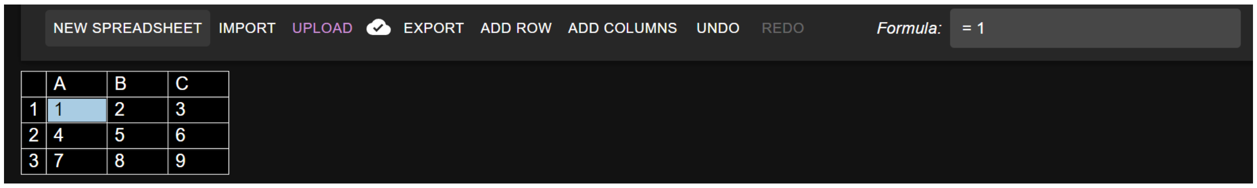
When importing a file, the contents of the CSV will be automatically parsed to fit the syntax and structure of our spreadsheet app.

For example, a CSV on your hard drive might contain the following data:



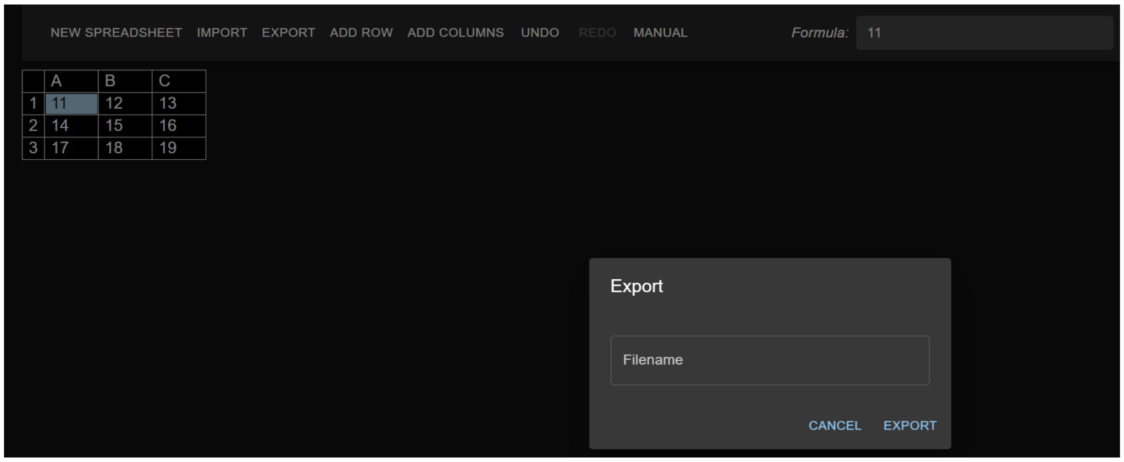
```
1,2,3
4,5,6
7,8,9
```

Once imported and uploaded, it will be parsed as follows into our application:

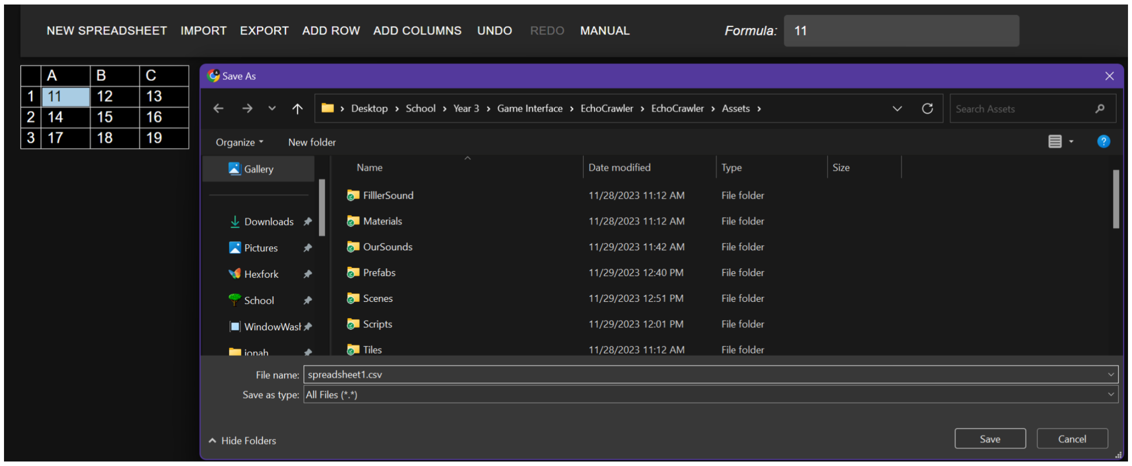


	A	B	C
1	1	2	3
2	4	5	6
3	7	8	9

Then, after making edits it can be renamed and saved in the following manner:



	A	B	C
1	11	12	13
2	14	15	16
3	17	18	19

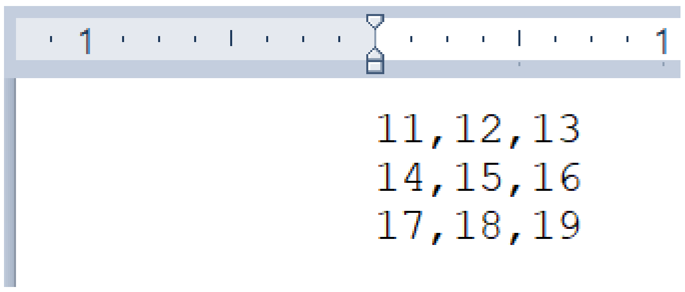


	A	B	C
1	11	12	13
2	14	15	16
3	17	18	19

Name	Date modified	Type	Size
FilllerSound	11/28/2023 11:12 AM	File folder	
Materials	11/28/2023 11:12 AM	File folder	
OurSounds	11/29/2023 11:42 AM	File folder	
Prefabs	11/29/2023 12:40 PM	File folder	
Scenes	11/29/2023 12:51 PM	File folder	
Scripts	11/29/2023 12:01 PM	File folder	
Tiles	11/28/2023 11:12 AM	File folder	

File name: spreadsheet1.csv
Save as type: All Files (*.*)

This will save the CSV file on your hard drive which now look like this:



```
11,12,13
14,15,16
17,18,19
```

We found this feature to be essential for the use of a spreadsheet application, as it allows the user to save their spreadsheet in-between sessions, and re-upload the saved file so they can pick it up for later use. Without this feature, or perhaps a similar cloud-saving capability, a spreadsheet application would be rendered pretty useless, since all changes would be lost every time the user leaves the app.

Additional Feature #2: Undo/Redo changes - Our application provides the user with UI capabilities to ‘undo’ changes made to an active spreadsheet and its cells, as well as to ‘redo’ any changes that have been previously ‘undone’.

The ‘undo’ and ‘redo’ buttons on the taskbar at the top of the spreadsheet control this feature. If either the undo or redo button is grayed-out, that means there are no valid changes to be ‘undone’ or ‘redone’, respectively. The UI will prevent the user from clicking on these buttons to prevent errors when trying to undo/redo changes when no changes are present.

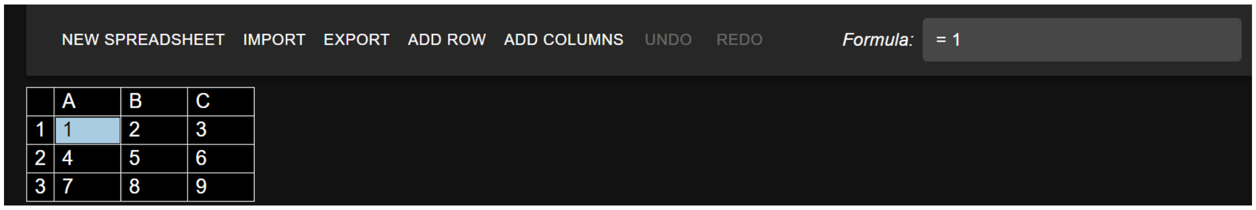
If the ‘undo’ button is white, meaning it is clickable, that indicates that there has been a change made to the spreadsheet that can be reversed. By clicking the button a single time, a single change will be reversed and the spreadsheet returned to its state prior to the change. You may press the ‘undo’ button multiple times to undo multiple changes. Once there are no more changes to the spreadsheet that can be ‘undone’, the button will once again become grayed-out and unclickable.

The ‘redo’ button will become white and clickable AFTER an ‘undo’ action is performed, and is the most recent action. Pressing it would revert the ‘undo’ action, and return the spreadsheet to its original state before the user pressed ‘undo’. If a non-’undo’ action is subsequently performed, then the ‘redo’ button will once again become deactivated since the most recent change is no longer an ‘undo’ action.

Examples of actions that can be ‘undone’ include, but are not limited to, adding/removing a row/column, editing the contents of a cell, uploading a CSV file, and entering a formula/expression.

Below is a visual example of a spreadsheet in the application at four successive states (take notice of the ‘undo’ and ‘redo’ buttons at each state) -

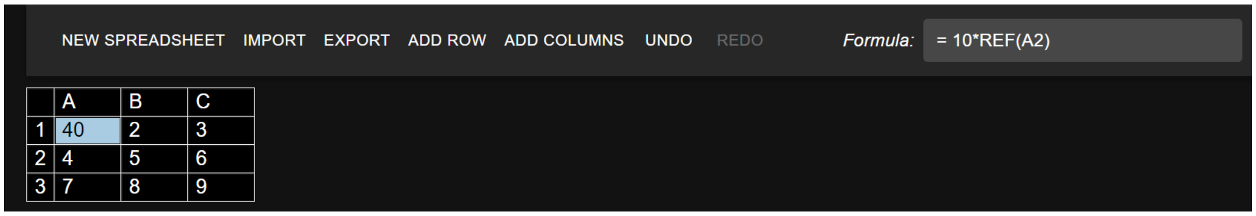
1) A new spreadsheet prior to any edits:



Formula: = 1

	A	B	C
1	1	2	3
2	4	5	6
3	7	8	9

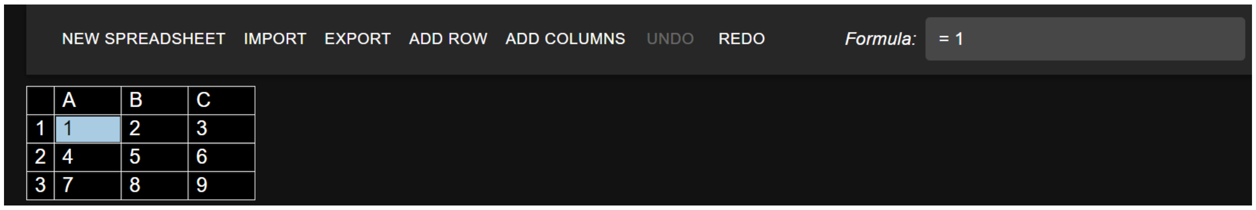
2) The spreadsheet after an edit is made:



Formula: = 10*REF(A2)

	A	B	C
1	40	2	3
2	4	5	6
3	7	8	9

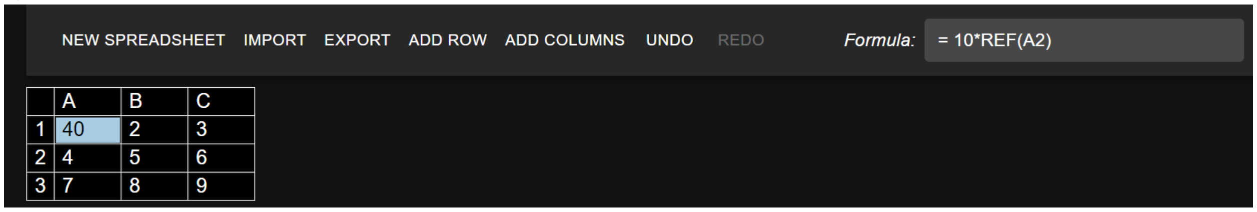
3) The spreadsheet after the edit is 'undone':



Formula: = 1

	A	B	C
1	1	2	3
2	4	5	6
3	7	8	9

4) The spreadsheet after the 'undo' is 'redone':



Formula: = 10*REF(A2)

	A	B	C
1	40	2	3
2	4	5	6
3	7	8	9

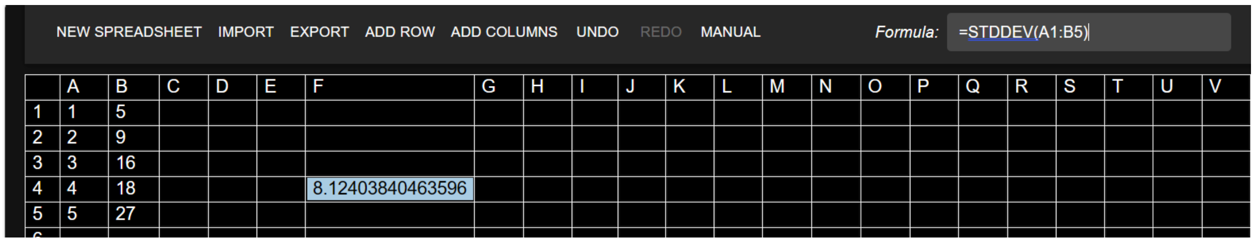
This feature greatly adds to the quality-of-life of our application. As human beings, we all make mistakes, especially when it comes to hyper-specific and mathematical actions like spreadsheet creation and data analysis. The ability to quickly and reliably revert your mistakes is very helpful and saves the user from the major headache that would be manually going in and changing back every cell to its original value after making a slew of erroneous edits.

*Additional Feature #3***:** *Statistical Analysis* - Lastly, our application gives the user a set of features/formulas that allow for more advanced statistical analysis within a spreadsheet as compared to the base functionality which consists only of formulas and range-based expressions.

Many spreadsheet users are using sheets to take in data they then want to do statistical analysis of. To assist with this we have included three additional functions in the application, STDDEV to read the standard deviation of a range of values, ZTEST to get the z-value of a number based on a sample of population data, and REG, to calculate a linear regression on a given data set.

These functions work the same as any other range expression, with the exception of ZTEST which requires a second function argument after a comma, formatted ZTEST(A1:B2, x) where x is the value we want to perform a z test on. These tools make statistical modeling far easier as it can all be done in the spreadsheet itself without using external tools.

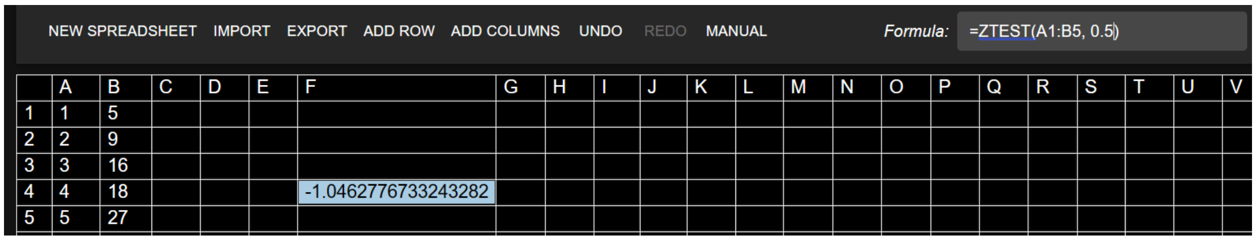
Standard deviation function



Formula: =STDDEV(A1:B5)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1	1	5																				
2	2	9																				
3	3	16																				
4	4	18				8.12403840463596																
5	5	27																				

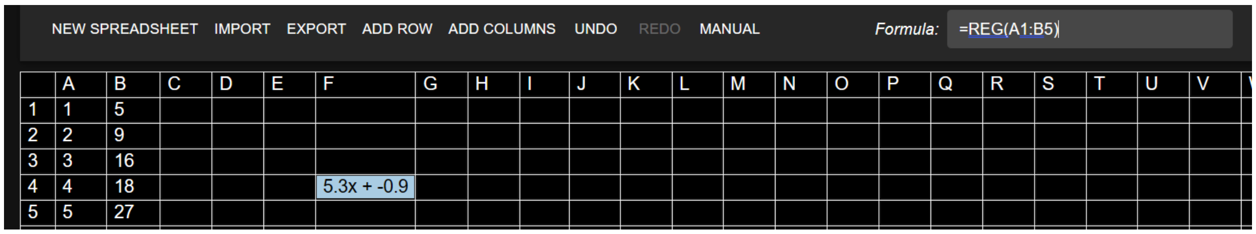
Z-Test function



Formula: =ZTEST(A1:B5, 0.5)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1	1	5																				
2	2	9																				
3	3	16																				
4	4	18				-1.0462776733243282																
5	5	27																				

Regression modeling



Formula: =REG(A1:B5)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1	1	5																				
2	2	9																				
3	3	16																				
4	4	18				5.3x + -0.9																
5	5	27																				

High-Level Architecture:

At a high-level, our system is built of a few important components. In the backend, the most important classes are Spreadsheet and Cell. Most of our logic is contained in these two classes, and the subclasses they interact with.

The Spreadsheet class, also known as the Spreadsheet model, handles core system functionality such as creating new spreadsheets, inserting and removing rows and columns, and keeping track of all actions so that the user can undo/redo their changes. These user actions are stored as ISheetAction objects, which are concretely known as ColumnAction and RowAction, and contain the information necessary to undo and redo any change made to the spreadsheet.

Stored within the above-mentioned spreadsheet model is a 2D array of Cells. The Cell class contains the logic that handles all arithmetic, formula parsing, cell referencing, range based expressions, and statistical analysis. The Cell class utilizes custom literal objects, namely the StringLiteral, NumberLiteral, and ErrorLiteral classes.

The UI consists of two high-level components: the TopBar and the Table. These are connected in the App.tsx file, which enables them to share the underlying spreadsheet and the currently selected Cell. In the Top Bar, each button connects to the Spreadsheet and the higher-level states, such that when a button is pressed, it updates the spreadsheet and the display. Essentially, we pass high-level state variables and functions down through our props so the low-level components can access them. The Formula is implemented similarly, as the currently displayed formula is treated as a high-level state.

The Table class renders the Spreadsheet data row by row. Firstly, the column headers are DeleteSliceButtons, which are components that can be used for deleting either rows or columns. Each row consists of a row header, which is a DeleteSliceButton, and a row of DisplayCells. DisplayCell is essentially a wrapper for the Cell class that also enables interactivity through its onClick methods. When a cell is clicked, DisplayCell passes this information up to the higher-level states, where it is used to render and change the appropriate formula. Furthermore, the DisplayCell handles the cell highlighting when a cell is clicked.

Tracking Cell Dependencies:

Each spreadsheet cell is initialized with two cell sets, an “upstream” set and a “downstream” set, both initially empty. The “upstream” set denotes cells upon which the value of this cell depends. Conversely the “downstream” set denotes the cells whose value depends upon this cell. Whenever a cell’s equation is evaluated and we come across a reference to another cell, we insert the current cell into the referenced cell’s downstream, and we insert the referenced cell into the current cell’s upstream.

This allows us to do two important things, firstly, whenever a cell’s value is changed, we can easily loop through everything in that cell’s downstream, and recursively update those cells until there is no more downstream. Secondly, whenever we need to know if a cell is dependent on another (to prevent things such as circular references) we can simply check if the dependant cell has the relevant cell in it’s upstream, but recursively seeing in any of the cell's in it's upstream are the cell we're looking for.

System Evolution:

Throughout the development cycle, our design has changed significantly. The current implementation draws from our initial phase B design plan, but makes major changes in system architecture. For example, in our Phase B plan, we included an abstract Dataset interface with concrete classes for Row and Column, but in our current design, we have instead opted to simplify things and just use a 2D array of Cells in our spreadsheet model. We decided to make this change after trying to use separate arrays of rows and columns, as planned, and realizing that there can be dependency issues when inserting/removing rows/columns and it was much more streamlined to leave this idea behind.

Another departure from our original plan is that we steered away from implementing plots/tables as an additional feature. We found during the implementation phase that we may have bit off more than we can chew by planning for 4 major additional features. This could have been more achievable if we had 4 team members, such that each person could cover 1 of the 4 planned features, but at a certain point we agreed that our 3 features added plenty of functionality to fulfill the requirements of the application spec.

Development Process:

Techniques and Methods:

During this project, we used the Sprint development method. Each meeting consisted of us sharing our work from the past week and resolving any questions about it. Then, we discussed which features we should tackle next, and assigned them to each person. We also formed tentative plans for the following week, so we had a longer-term plan and could budget our work appropriately.

Libraries Utilized:

We used the MaterialUI library for UI templates for things such as buttons, pop-up dialogs, and the top bar. This helped us achieve a more polished look and feel for our web interface.

Team Collaboration:

Our team consisted of three team members. During this project, one team member worked on the low-level details of formula parsing. Another team member implemented the front-end user interface using React and MaterialUI. The final team member handled the higher-level details of the Spreadsheet implementation, and also took the lead on the written and documentation portions of the project such as the weekly progress reports.

This division of labor helped enforce design patterns we learned in class – it was very useful to be able to call a method that we hadn't written ourselves and rely on its output. Furthermore, it integrated well with our respective strengths – the team member using React had some experience with React, and the other two team members had more experience with back-end software engineering.

It was feasible to implement all requirements despite us being a smaller team than the rest of the class, but a fourth team member would have been helpful at certain points for doing things like writing tests, fixing the bugs as they come up, and an overall broader division of labor.

Lessons Learned / Experience Gained / Problems Encountered:

We learned a lot about version control and developing software that meets specific requirements. During the project, we utilized multiple branches for each of our individual changes, as well as branches focused on entirely new features or approaches. For example, we created a new branch when we transitioned from a purely HTML-based UI to a UI using templates from MaterialUI.

Most of the challenges we faced had to do with issues relating to multiple pieces of the application. For example, we migrated from a column labeling system that went from A to Z to one that could go past Z with labels such as AA and AB. This required changes to the underlying logic of cell referencing, as well as the front-end formatting of the additional columns. To overcome this challenge, we agreed on a common way to represent this on both ends, and each of us ensured that our section was compatible with the larger application.

We have included the installation and user guide in the README included with the implementation. Additionally there is a manual on the web application for further usage guidance.