Gatheo Studios©

UM Bus Transportation Tracking Application
Full Project Portfolio

Gabriel Ruiz, Matthew Class, Matteo De Angelis

Version 1.4

Revision History

Date	Version	Description	Author
07/Dec/22	1.0	General Outline; Combining assignments; Feedback revisions	Matthew Class
09/Dec/22	1.1	More combining assignments; edited diagrams	Matthew Class
10/Dec/22	1.2	Added cloud architecture diagram, database diagrams, and new database section (11.3)	Matthew Class
11/Dec/22	1.3	Modified App UI, updated pictures respectively, and added more description text	Matthew Class
13/Dec/22	1.4	Added Front-End Technologies, Back-end Technologies, Database Products, and DevOps Tools Added WBS, Project Timeline Plan, Project Cost Plan	Gabriel Ruiz, Matteo De Angelis

Table of Contents

Full Project Portfolio

1. Introduction

The UM Bus Transportation Tracking Application is a mobile app that will better communicate the buses arrival time, departure times, live tracking of location of buses, and the fullness of buses. The user interface should be easily understood, clearly communicating the data to the user. The goal is to improve upon the already existed bus tracking system for a better overall experience. For both the bus riders and bus drivers.

1.1 Purpose

The purpose of this project is to improve the quality of the UM bus system. At the moment, the information provided by the current application is limited. However, this project serves to improve it by adding more features. It will provide the user with an easier experience as well as provide important information for UM.

1.2 Scope

This project's scope pertains to improving the current bus tracking system for better communication. It focuses on the University of Miami, its bus system, and the people that ride the bus.

1.3 Definitions, Acronyms, and Abbreviations

- UM – University of Miami
- GPS – Global Positioning System
- API – Application Programming Interface
- JSON – JavaScript Object Notation
- USB – Universal Serial Bus
- GUI – Graphic User Interface
- REST – Representational State Transfer

1.4 References

- ITEM 01-Sample_SRS
- ITEM 02-Sample_UseCaseModel

1.5 Overview

The following sections will provide more information on the functionality, performance and requirements of this project. Provided in these sections are the general constraints we believe will better the usability for the application. Lastly, the final section provides a user case model as well as all the actors for the project. This describes the typical use case of both the end user and the administrators. These sections provide a general use for the application as well as an understanding of all interactions to and from the application.

2. Overall Description

The proposed project is a major improvement to the UM bus system. At the moment, users have access to limited information about the arrival of buses to any given stop. This project aims to provide the students and faculty members of the university with more information that will improve the user experience. Utilizing weight and GPS sensors, users will be provided with the real time location of any bus as well as the number of riders on that bus.

Assumptions for the project:

- The application will be used as a web app since this will be easier for users to simply utilize any device and browser.
- Any student or faculty member of UM will be given access to the application through CaneLink.
- UM will serve an administrator to add or remove buses to the service.

3. Specific Requirements

3.1 Functionality

3.1.1 Display Information

Users will be able to choose the particular stop that they wish to view estimated time. Additionally, the bus's information including real time GPS location and number of riders will be displayed.

3.1.2 Add and Remove Buses

Administrators will be able to add and remove any bus to both the system and the database. This will be only visible to the administrators, and once the bus is added, users will be able to receive all information pertaining to that newly added bus.

3.2 Usability

3.2.1 Training Time

Training time for a user must be zero seconds. As we aim to provide information easily. The user must be able to understand what is on his or her screen with just a glance.

3.2.2 Response Time

As we want the application to update the buses' location in real time, we are aiming for a worst-case scenario of 500ms to update the information.

3.2.3 Communication

An API must be established for communication between the buses and the user end of the application. This API will request data from the buses, and then communicate the concise information needed to the user.

3.3 Reliability

3.3.1 Availability

The application should follow the normal UM bus services hours, and any maintenance must be done outside of these hours. Degraded mode will also occur during these hours as the application will only have to display limited information.

3.3.2 Mean Time Between Failures

Mean Time Between Failures should ideally be day since the application should work all day, and then be maintained or fixed outside of working hours.

3.3.3 Mean Time to Repair

Mean Time to Repair should be limited to at most 1 hour. Due to the buses' busy and frequent schedule, the time must be kept to a minimum to improve reliability.

3.3.4 Accuracy

The application needs to be accurate in two regards, any bus's position and estimated arrival, as well as the number of people on any given bus. These are the two main aspects of the application that require providing a useful service.

3.3.5 Maximum Bugs or Defect Rate

Bugs should be kept to a minimum; therefore, we will follow the typical commercial average of 20-30 bugs per thousand lines of code.

3.3.6 Bugs or Defect Rate

Major bugs include erroneous estimated times, bus position and amount of people on bus that can come from sending the wrong packet information. Minor bugs can include an item not being displayed in the correct location, as well as the program getting "stuck on loading".

3.4 Performance

3.4.1 Response Time

Response time for the application should be not more than 500ms, as the buses' positions must be updated frequently. However, the amount of people on a given bus does not have to continuously be updated as we can compute this number in any stop, and thus the response time will not be as heavily impacted as the location.

3.4.2 Transitions per second

Our estimated transactions per second will be two. This is because we want the location of the bus and the estimated arrival to be updated every 500ms at the latest.

3.4.3 Capacity

The capacity for the project will be any UM student or faculty member. Given that this application will provide general information. Users will not require an account, and at any given time, any user can easily access the information provided.

3.4.4 Degradation mode

Degradation mode will happen when the database containing each bus's unique key fails. In this instance, the application will still be able to provide information on a bus's arrival and capacity, however, the user will not be able to see the bus's location in real time.

3.4.5 Communication Layer

This application relies heavily on the use of a communication layer. Buses must be able to communicate with a server that will then relay the information to the user. As well as a relatively small database, this will contain information, such as serial numbers, to distinguish any bus.

3.5 Supportability

3.5.1 Supportability Requirement One

Due to the nature of the application, we understand that a web app would suffice for displaying information to the user. Therefore, Typescript will be our language of choice as we can build both the API as well as the front-end for the application. In addition, our naming convention will be camelCase as it is easy to follow and makes the code more readable.

3.6 Design Constraints

3.6.1 Design Constraint One

As stated previously, our software language of choice will be TypeScript. Additionally, we will follow Scrum for our project management as this will ensure that new features are added and maintained more consistently. Our main development tool will be Visual Studio Code, since this provides all the functionality we require. The other two main requirements are the library ReactJs for front-end development as well as the runtime environment for developing and running our API. Additionally, we also require a hosting platform, which we will utilize Heroku.

3.7 On-line User Documentation and Help System Requirements

- A full tutorial walkthrough of the app will be provided online to anyone that may need help navigating the app.
- Additionally, there will be an available help page in the app to give quick tips and advice on navigating the app.
- Another full tutorial for admin privileges (i.e., bus driver users) to walkthrough any of those features.

3.8 Purchased Components

Purchased components for the application will include a miniature computer with GPS and weight sensors that will connect to the API to update the information on any bus. As well as the continuous cost of deploying the API on Heroku.

3.9 Interfaces

3.9.1 User Interfaces

The user interface will be kept to a minimum as we only wish to provide information at a glance. Additionally, users will simply have to provide the stop him or her are currently located to determine the estimated arrival time.

3.9.2 Hardware Interfaces

Since the buses will have to communicate to the API wirelessly, each bus will contain their own hardware interface. This will be composed of a GPS locater, weight sensors to determine the amount of people on the bus, and finally, the microcomputer that will relay the information to the API. These components will be connected through USB.

3.9.3 Software Interfaces

Since we want the information to be displayed neatly, we will take advantage of the fact that most people have phones with a GUI already. The application will have a front-end as a web application since this will prove to be more convenient for users, as they can simply use any browser.

3.9.4 Communications Interfaces

Communication will be handled by the API. This will be a RESTful API as this will serve as the communication between the buses and the web application front-end.

3.10 Licensing Requirements

Any software/hardware devices used that require licensing will receive the right documentation to be able to use it.

3.11 Legal, Copyright, and Other Notices

- All bus data is purely an estimation at times. None of the data listed is truly live so some timings and data may be off slightly.
- Any trademark, product warranties, disclaimers, and/or copyright of companies/products used will also be provided.

3.12 Applicable Standards

Industry standards will apply where applicable.

4. Use-Case Model for the System

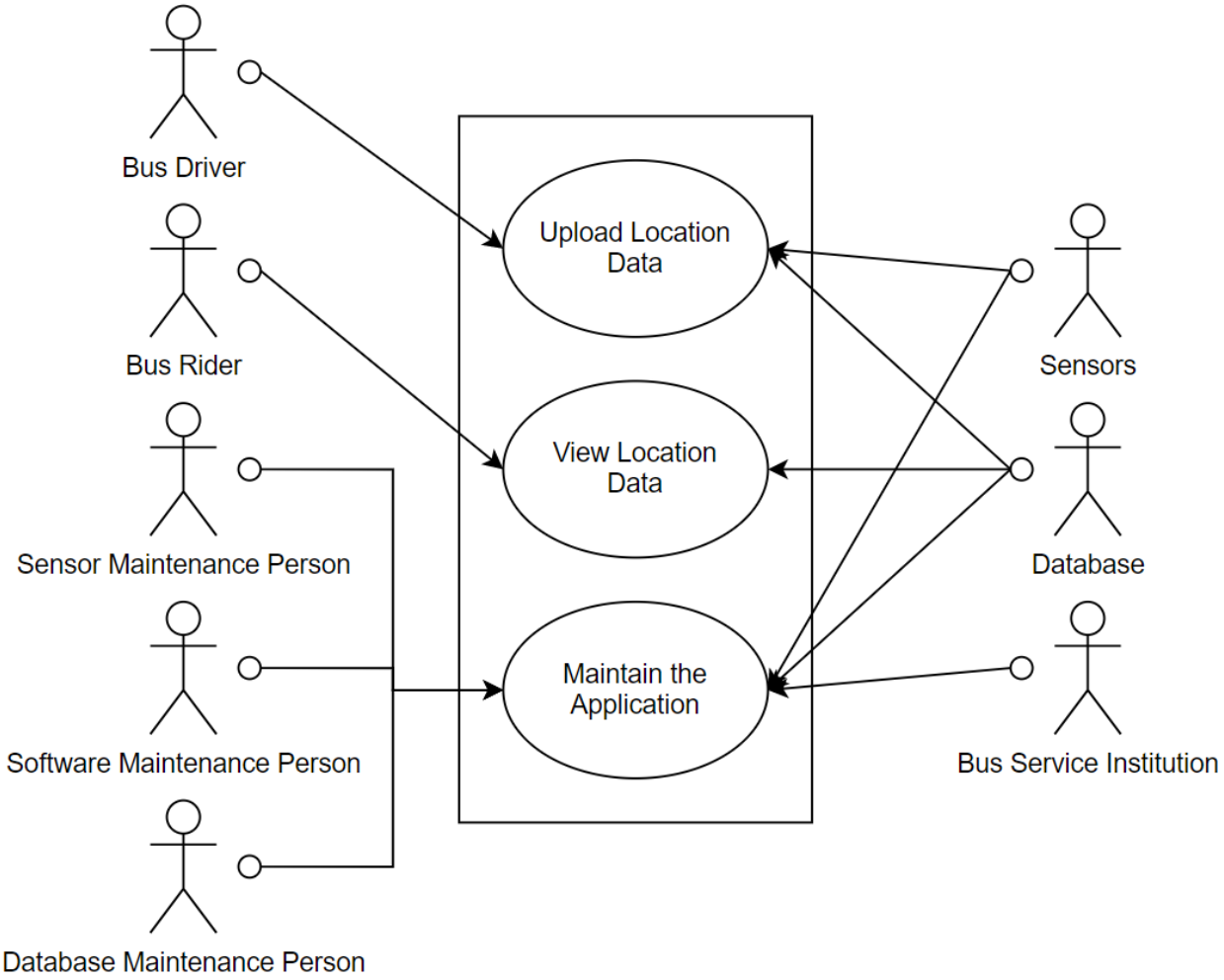

Figure 1: Use-Case Model for the System

5. List of Actors

5.1 Bus Driver

This actor represents whoever is operating the bus. They will be interacting with the system by manually inputting data such as break time and unexpected delays/occurrences.

5.2 Bus Rider

This actor represents whoever is riding the bus. They will be interacting with the app to know about the data about the bus.

5.3 Sensor Maintenance Person

This actor is responsible for maintaining the sensors used to track the buses. Whether that requires a repair or replacement.

5.4 Software Maintenance Person

This actor is responsible for maintaining the application and any other software involved. If any bugs or problems occur in the software, they will need to fix them as soon as possible to ensure everything runs smoothly.

5.5 Database Maintenance Person

This actor is responsible for maintaining the database. Any problems that occur with storing or retrieving data with the database should be handled by them.

5.6 Sensors

This actor represents all sensors used to determine bus information. The most prominent one being the GPS system on board every bus.

5.7 Database

This actor represents the database that stores all wanted information. This includes bus information and any user information as well.

5.8 Bus Service Institution

This actor represents the institution that provides the services to the application. In this case it is the University of Miami.

6. List of Use-Cases

6.1 Upload Location Data

Bus sensors, bus driver, and database are the actors in this use case. This use case will include the upload of all information regarding the bus to the database for both storage and for user access.

6.2 View Location Data

Database and bus rider are the actors in this use case. In this use case, the database will be used to pull the current bus data so that the bus rider may view the data.

6.3 Maintain the Application

Sensor maintenance person, software maintenance person, database maintenance person, sensors, database, and bus service institution. This use case will ensure all required systems stay fully operational.

7. Use-Case: Upload Location Data

7.1 Brief Description

This use case will be gathering data for the bus through the bus driver giving data/feedback and the sensors themselves.

7.2 Actor Brief Descriptions

7.2.1 Bus Driver

This actor represents whoever is operating the bus. They will be interacting with the system by manually inputting data such as break time and unexpected delays/occurrences.

7.2.2 Sensors

This actor represents all sensors used to determine bus information. The most prominent one being the GPS system on board every bus.

7.2.3 Database

This actor represents the database that stores all wanted information. This includes bus information and any user information as well.

7.3 Preconditions

The phone has the app downloaded

There is an active network for the mobile app to connect to

7.4 Basic Flow of Events

1. The user opens the bus riding app.
2. The app will display all information about all bus routes.
3. The user will be able to select a particular route.

7.5 Alternate Flows

7.5.1 Specific Bus Information

1. If in step3 the user selects a particular route, the app will request to select a bus route.
2. The user is prompted to select a stop at the given bus route.
3. The information about the next upcoming bus is displayed.

7.6 Key Scenarios

7.6.1 The user would like information on a given bus

1. The user is prompted to select a bus route.
2. The user is prompted to select the bus stop where they are located.
3. Information about the next upcoming bus is displayed good.

7.7 Post Conditions

7.7.1 Successful Completion

Data is successfully sent from sensors and bus rider to both the bus rider and database for storage.

7.7.2 Failure Completion

Error with sending data and error message logged for examination later.

7.8 Special Requirements

[SpReq: WC-1] The system should be sending data every 500ms.

[SpReq: WC-2] Not all data received will be given to the users. Some data will be exclusively sent to the database for internal uses.

[SpReq: WC-3] App will be allowed to run minimally in background for most accurate and fastest data retrieval. If not allowed, slight delay may occur.

7.9 Sequence Diagram for the Basic Flow

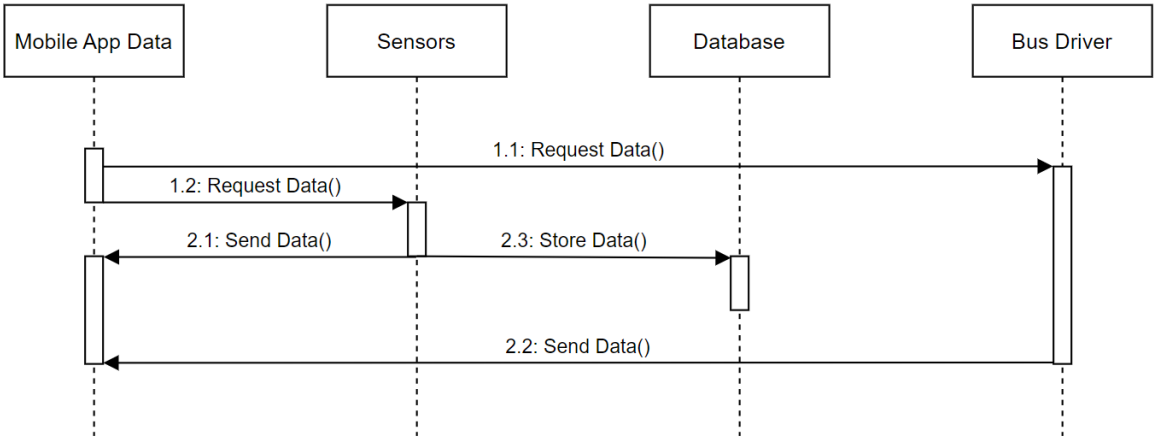

Figure 2: Basic flow of “Upload Location Data”

8. Use-Case: View Location Data

8.1 Brief Description

This use case will be the application user reading the data made available to them, and the database reading data given to it.

8.2 Actor Brief Descriptions

8.2.1 Bus Driver

This actor represents whoever is riding the bus. They will be interacting with the app to know about the data about the bus.

8.2.2 Database

This actor represents the database that stores all wanted information. This includes bus information and any user information as well.

8.3 Preconditions

The phone has the app downloaded

There is an active network for the mobile app to connect to

Data is available to be read (no error in sending/receiving data)

8.4 Basic Flow of Events

1. User opens app.
2. App establishes connection to server.
4. Server responds with connection confirmation, and awaits requests.
3. App sends request to display all bus routes information.
4. Server responds with appropriate information.
5. The user selects the bus route and stop, he or she wishes to view.
6. App sends request to server with bus route and stop flags.
7. Server responds with information for that specific stop.
8. App closes or goes idle.
9. Server removes the connection.

8.5 Post Conditions

8.5.1 Successful Completion

The bus rider receives full live data and is able to read it regarding the buses and the database receives the data for storage as well.

8.5.2 Failure Completion

Error message sent to user along with to system to log for examination. Will then keep requesting data live until successful completion.

8.6 Special Requirements

[SpReq: WC-1] Data should be given in an easily readable format. Timings should be streamlined and simplified.

[SpReq: WC-2] If allowed to run in background, main data will already be available to app. If not allowed, longer and more communication will be required to obtain the live data.

8.7 Sequence Diagram for the Basic Flow

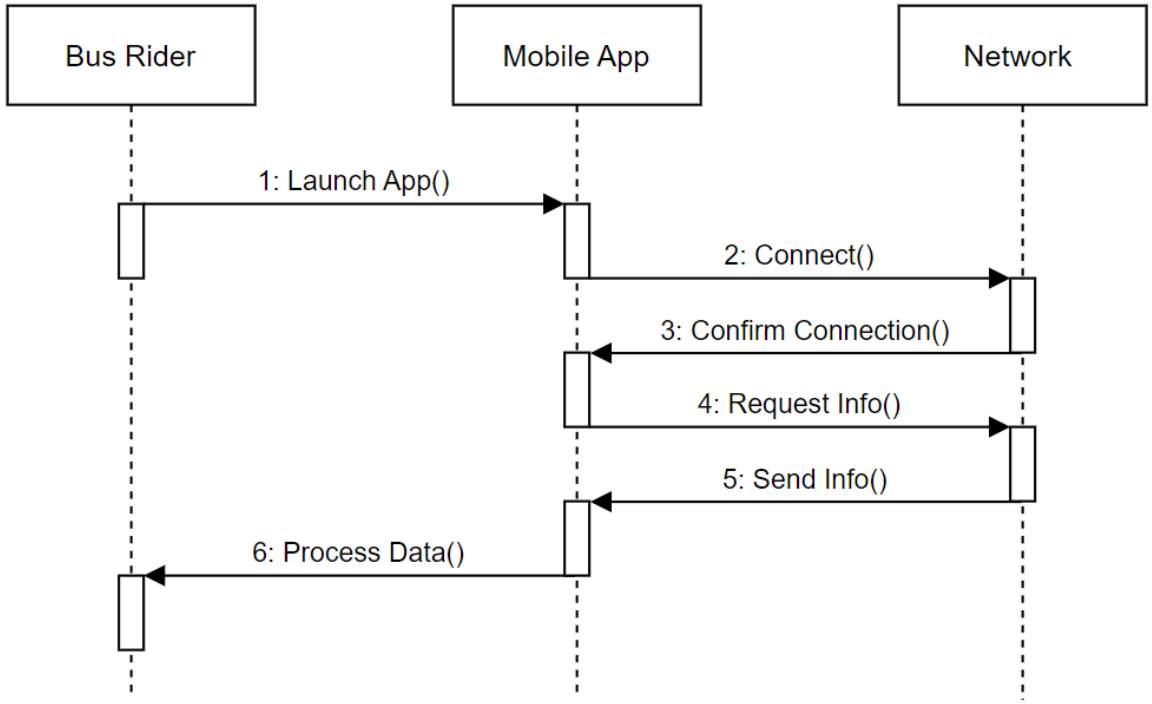

Figure 3: Basic Flow of "View Location Data"

9. Conceptual Architectures

The current design for the UM Bus Application will be utilizing two architectures: the pipe filter architecture and the client server architecture. The pipe filter architecture will allow for a constant stream of data while the client server architecture favors the design of the users connecting to the server for accessing the wanted data.

9.1 Pipe Filter Architecture

Pipe filter architecture is a synchronous system allowing for a constant stream of data to be sent over the network. Pipe filter architecture will be integral to the use case of uploading the bus data. A synchronous system is an important feature for this system as it relies on constant updates on the buses' data. Both the live tracking of the bus and an accurate estimate of arrival times is integral to the system to provide the most accurate data to the users. As stated in the team's requirement specification document. The worst case the team is aiming for 500 milliseconds. Preferably though, the system will be faster than this requiring the constant flow of data. Figure 4 below shows a sample diagram of the pipe filter communication for the project. Here, GPS data from the buses is transmitted through a pipe (the arrows), it will then arrive in a filter. The filters will process the data and do any necessary computations to the data necessary. This process of sending from pipe to filter can be repeated as much as it is needed. Finally, the data will be sent through a final pipe to the data sink. The data sink is the target data, which is what the user is requesting [1].

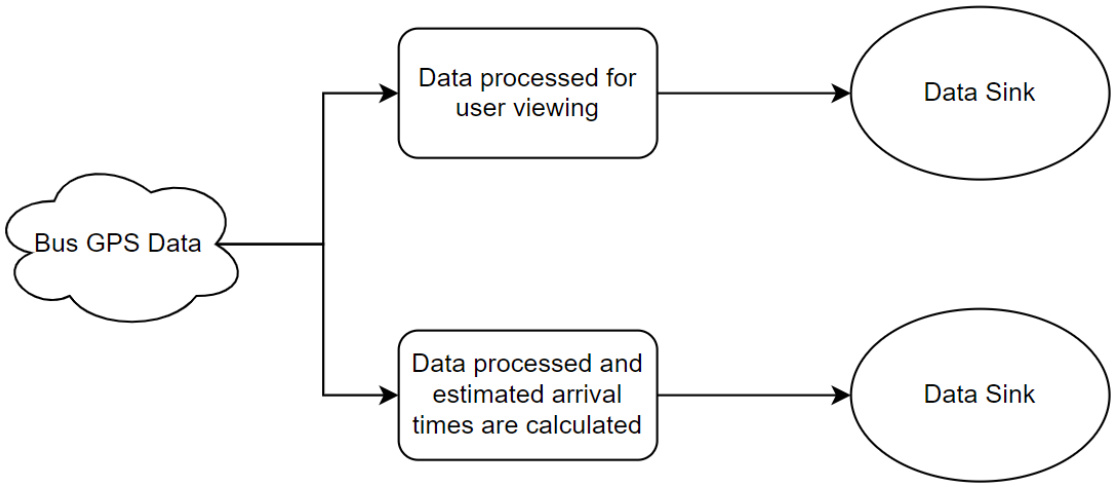

Figure 4: Pipe Filter Architecture Diagram

9.2 Client Server Architecture

Client server architecture is a layered architecture comprising of the client (the users on their phones) and the server. Client server architecture will be important to the use case of the user viewing the bus data. This architecture is applicable as once the user launches the app; they will send a request to the server for connection. Once connected, they will then send a request for the live map bus information. If the user wants to see the estimated arrival time to their stop, they will have to send another request to the server to retrieve that information. They can then keep sending request to the server for any other information they will want to access. They will then be able to end their session by exiting and closing the app. The app will only do a timed-out disconnect after the app is not used for at least 10 minutes. The UM Bus App will be a thin client as it requires a connection to the server to receive the bus information. The data will not be stored on the phone for both security reasons and to decrease load on any mobile phone. Figure 5 below shows a sample diagram of the communication with this architecture. The communication shown for "client 2" is how it would look for any other client.

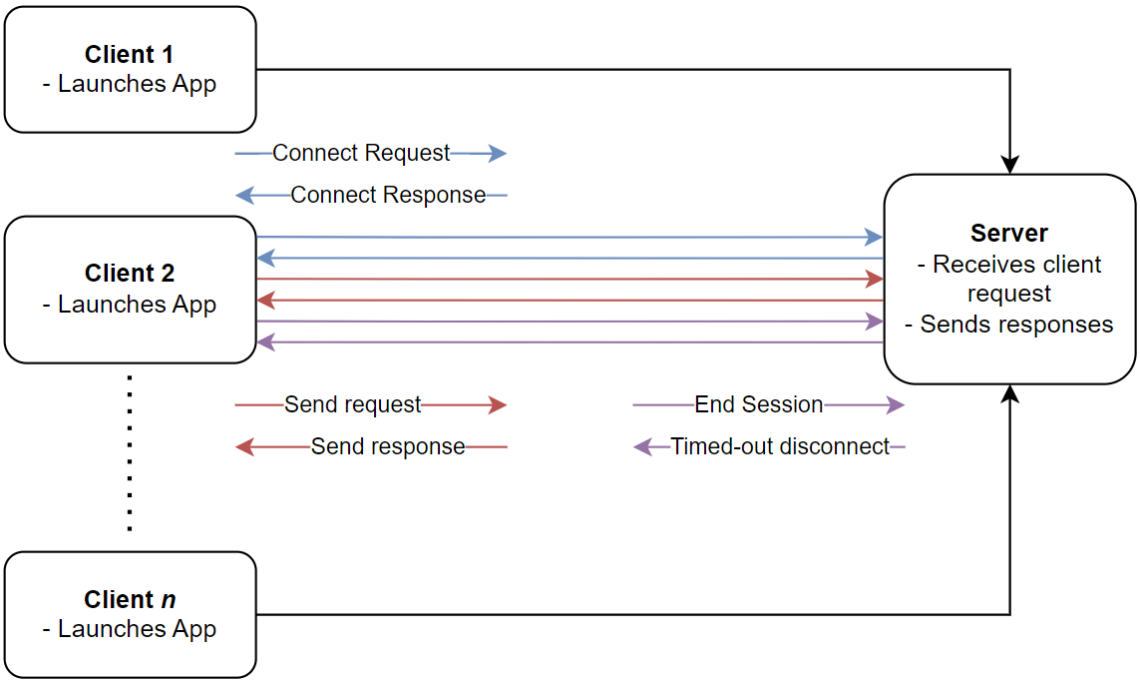

Figure 5: Client Server Architecture Diagram

10. N-Tier Cloud Architecture

The current system design for the application will be supported by an N-tier cloud architecture. N-tier cloud architecture uses the layered conceptual architecture. This cloud architecture is simple allowing for straightforward integration. The team's current system integrates into this well as this cloud architecture is good for web application framework. These will make it easier to maintain, write, and scale the web application for this project [2]. N-tier cloud architecture can also take advantage of making remote service calls. This will be important for users to login with their UM credentials. The login will bring the user to the web to use the university's login system instead of the team making a completely new system. The UM Bus Application will use an open layer architecture. As the system needs to be transmitting the data as fast as possible to keep a real-time update of information for the user, open layer will be better for that. Closed layer can run into issues of have to loiter through useless layers just to pass data to the layer that needs it. Open layer architecture will allow the data to be able to immediately get to the layer that needs it. The only concerns with using this architecture are its monolithic nature, security risks, and the dependencies that can occur. Dependencies are caused by wasted layers and the system requiring certain requirements to pass the data along. Using open layer architecture should help with this slightly though. Dependencies and security risks can be avoided though when good design is kept in mind. The monolithic nature of this system is a harder challenge to overcome as it is the nature of this type of architecture. This will have to be maintained and reviewed through the projects development operations (dev-ops) stage and deployment stage to ensure a system upgrade is not required. As shown in figure 6 below, this is a general design of our system. Our system design gets hazy once in the "middle tiers" as this is unknown at the time how many tiers and their task we will have. We will have the client, a web application firewall (WAF), a web tier, messaging to middle tiers, and a data tier/database at the end though.

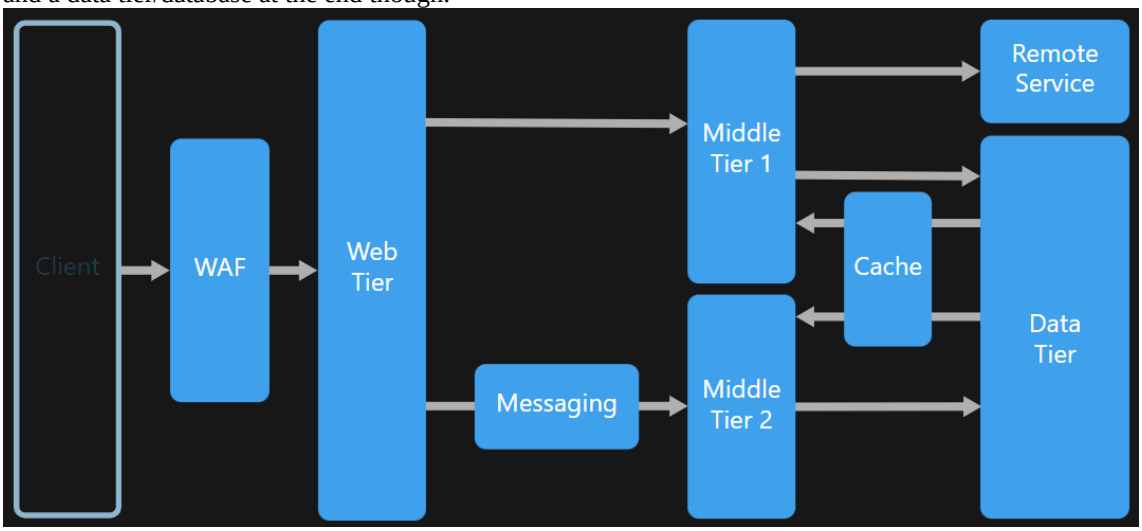

Figure 6: N-tier Cloud Architecture

11. Database Architecture

11.1 Transaction Data for Application

The transactional data will correspond to the "pinging" of the buses. This will be used whenever a rider wishes to receive information about the bus. The server will automatically detect the buses' location and calculate estimated time of arrival, and send the information to the driver. Another transactional data will be upkeep on the service. Information for the drivers, such as which bus to take and what their hours will be, will be provided separately once the driver logs on. Additionally, information of the upkeep of the buses will also be kept. Data on the buses fuel tank, mechanical services, crashes, and other similar situations (flat tire, door not opening, etc...) will also be reported by the driver as a transaction. Finally, user reviews and statistics about the service, how many riders, for how long, busier times, will be kept as a transactional data to be later analyzed for optimization of the application. As all the data revolves around timings. The time series database would be best for this. The time series database supports a high number of writes as data is collected in real

time from several sources. The sensors on the many buses can all transmit their data along with their timings all at the same time to the database. It will then organize it by the time. This will make it easy to calculate estimate bus timings. The data can then be analyzed later to further improve bus run times. This system also will do deletes done in bulk operations. Once the wanted data has been transferred to another database for analyzing, the database can be wiped to avoid running out of memory as the data size in this database architecture can grow rapidly. Figure 7 below shows a current design with sample data for the transaction database. The "location" column is currently just having *data* as we have not decided what data will be. Though the current though is to use latitude and longitude for the location. More research must be done on the GPS sensors to decide what this will look like.

Time	Location	Fuel	Notes
2022-10-27T12:10:00Z	*data*	90%	N/A
2022-10-27T12:10:05Z	*data*	90%	N/A
⋮	⋮	⋮	⋮
2022-12-01T15:30:45Z	*data*	45%	Going on break

Figure 7: Sample Data for Transaction Data

11.2 User Data for Application

The user data will consist of the two main profiles. One will be the student's and faculty's information which will be provided via CaneLink, and the second will be the information of the buses themselves. The application will use the same profile information, account email, picture, and C number, to establish the accounts for the riders. A separate profile will be created for the drivers of the buses where we will store the data on the drivers. Name, picture, position, and credentials will all be added to the driver's profile. Finally, we will also require static data that corelates to the buses themselves, such as serial number, date of purchase, and warranty information. The user data information is very relational to each other and are all integrated and connected with each other. This lends themselves to be a relational database management system (RDBMS). This will allow for intuitive connections between users, faculty, and the bus information. The team wants the user data database to be able to be updated anytime user information changes or to clear the system after a user has left the university. This database is also good as all students will have the same information to input, faculty will have the same information to input, and finally the bus information will have the same data to input. There is strong relationships and consistencies throughout the same tables. Figure 8 below shows a current design for user data. If needed, we will make our own database storing student/faculty information. Though, we intend to partner with the University of Miami and have access to their databases instead.

Username	C Number	email	Notes
jba2097	C52618549	jba2097@miami.edu	N/A
asd668	C12345678	asd668@miami.edu	N/A
⋮	⋮	⋮	⋮
qwe098	C09876543	qwe098@miami.edu	N/A

Figure 8: Sample Data for User Data

11.3 User App Information

The app itself will have application data that changes from user to user that should be saved. This data would include their settings, most recent route selected before closing the app, etc. This data needs to be opened quickly as loading the user preferences and profile management. For this reason, a key/value storage database would be perfect for this. Key/value databases are highly optimized for lookups as its design is akin to a large hash table. Each entry will have a unique key to quickly access the info needed. Figure 9 below shows sample data of the current design mockup for the user app information.

Name	Value	Size	Priority
Settings	ADkE8912jfMflela	20	Medium
Last_Used_Route(s)	DkMEo0128DKEk	10	Medium
⋮	⋮	⋮	⋮
Login_Info	lEf912DHEN193kfl	29	High

Figure 9: Sample Data for User App Information

12. UX Design

Figure 10 below is the UX mockup using Miro. This shows the general flow of bus app, and how the users will navigate it.

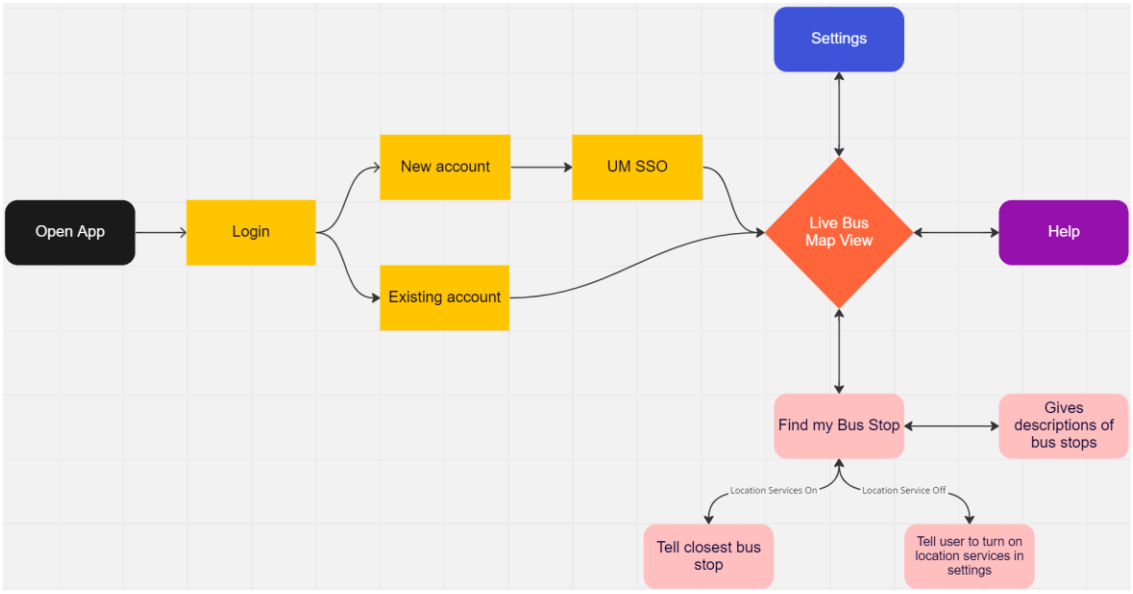

Figure 10: UX Mockup

13. UI Design

13.1 General UI Mockup Design

Figure 11 below shows a preliminary mockup of the UI with the general flow between pages. This was used as the foundation to start building the first prototype of the app.

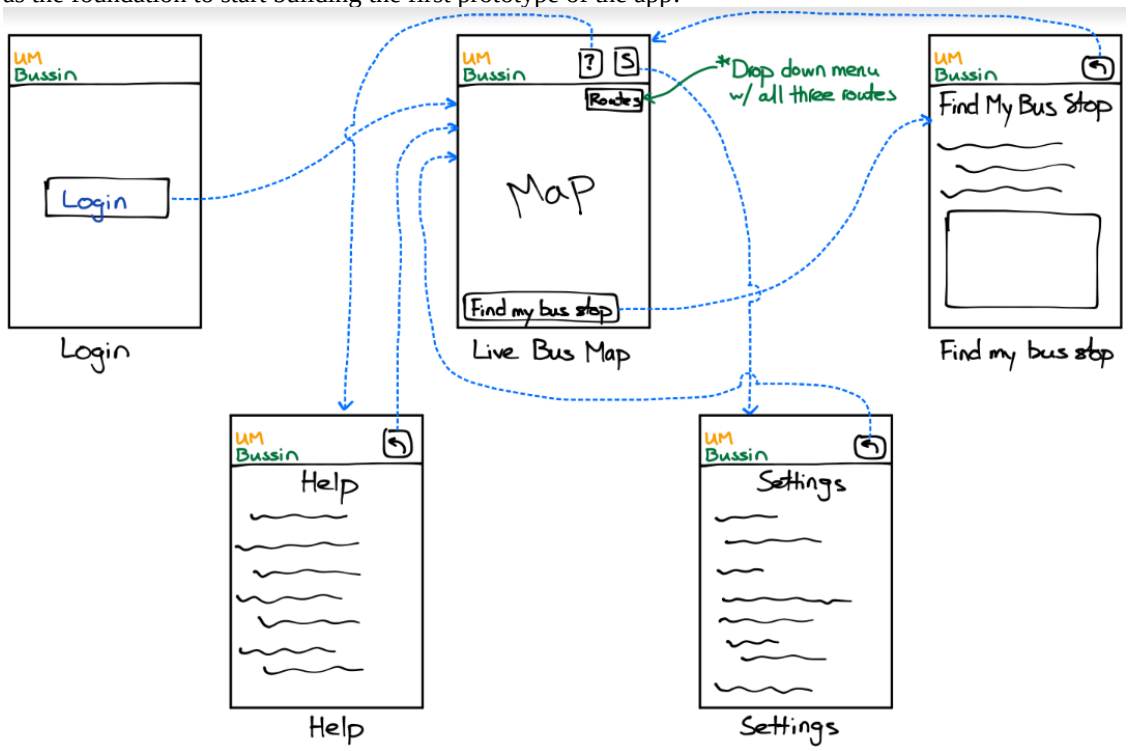

Figure 11: UI General Mockup

13.2 UI Prototype

Figures 12, 13, and 14 below are all the app pages made using Figma. Figures 12 and 13 shows all the

different app pages made. Figure 12 shows the first three pages. The “Live Bus Map” page will be the main page used by users. This map will display any selected bus routes. . The “All Routes” button will turn into a drop-down menu showing all available bus routes to select. The user will also be able to access all other pages here including Find My Bus Stop, Help, and Settings pages. The “Find My Bus Stop” will help the user find a bus stop if they need help. This can be done manually through the three drop down menus or through using GPS (if the user allows the app to access their location). Once a bus stop is selected, it will bring the user back to the map zoomed into the specific bus stop. They will then be given the option to go into street-view to see the surrounding area up close if they would like. Figure 13 has the final two pages of the app. The “Help” page will allow users to see a Q&A of any questions they may have. We have put some questions that will be appearing in the app there. The “Settings” page will have any settings the user can edit there. Currently, we can change the colors of the route. A logout button and feedback button will also be found at the bottom of the page. A note has been put at the bottom of these two pages that more help questions and settings will be added as development of the app continues. Finally, figure 14 shows the alternate page looks for both the “Live Bus Map” and “Settings” page. The left photo on this figure shows what the page will look like when the “All Routes” button is pressed. The empty white squares shown will be able to be selected to show the respective bus route. The right photo shows what the page will look like when the “Feedback” button is pressed. This displays a pop-up window where the user will be able to type in whatever feedback they would like. If at any point they want to cancel this, the “X” button may be pressed to close the pop-up window.

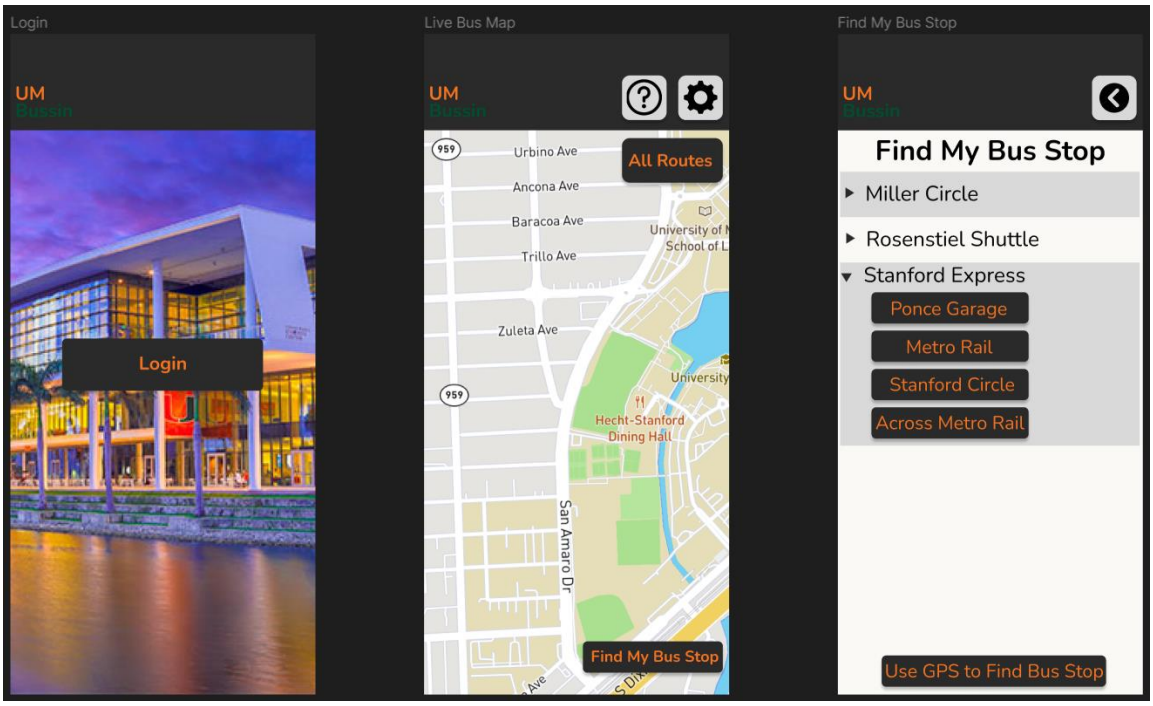

Figure 12: “Login”, “Live Bus Map”, and “Find My Bus Stop” App Pages

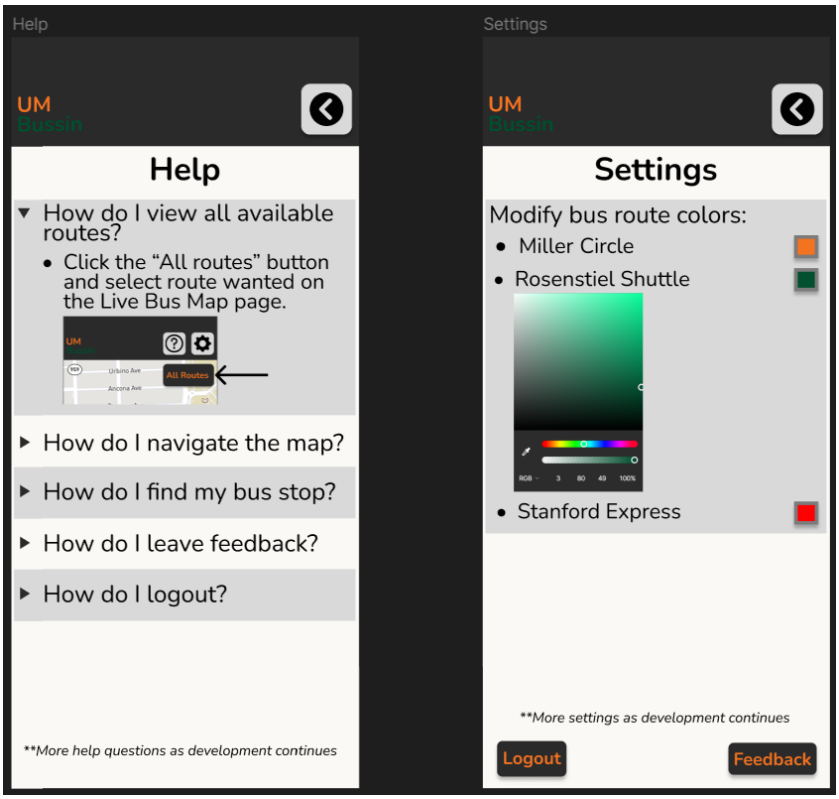

Figure 13: "Help" and "Settings" App Pages

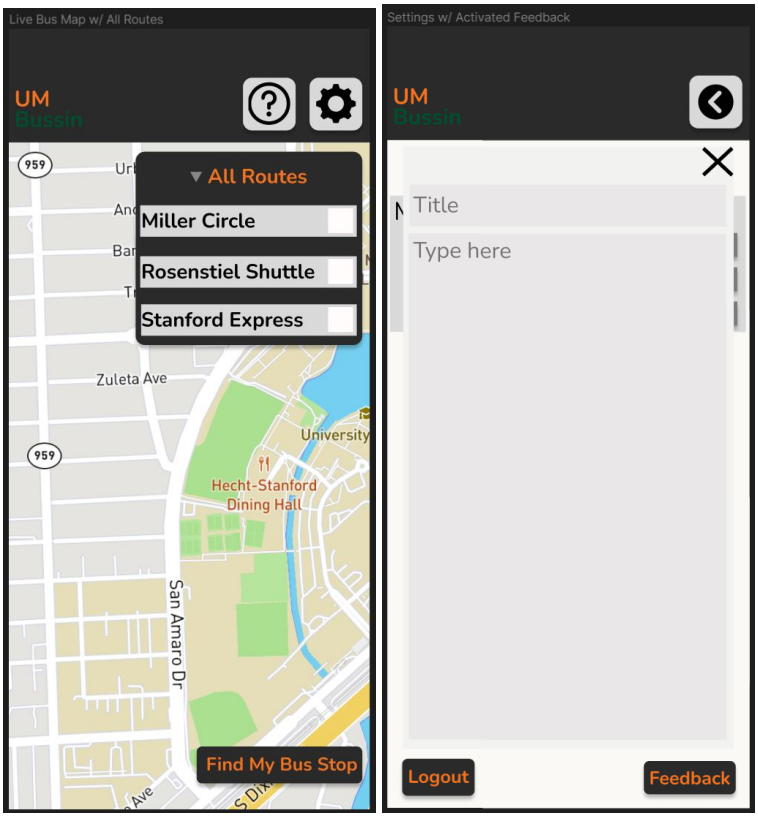

Figure 14: "All Routes" and "Feedback" Button Pressed

Figure 15 below shows the full flow the app. A software called Figma was used to develop this prototype. A full app demo is also available through the Figma app allowing for a walkthrough of what the app would feel like. The different arrows show that when that button is pressed, the user will be brought to the page it is pointing to. The “Login” page will lead the user to the “Live Bus Map” page. Here the “All Routes” button can be selected to create a drop-down menu. From here, they can access the “Find My Bus Stop”, “Help”, and “Settings” page. These three pages all have a back button found in the top right that will bring the user back to the “Live Bus Map” page. The “Find My Bus Stop” buttons will all lead the user back to the “Live Bus Map” page except it will bring them, zoomed in, on the selected bus stop. The “Settings” page houses two additional buttons found at the bottom. The “Logout” button will lead the user back to the “Login” page, and the “Feedback” button will provide the user with a full screen pop-up window with a text box allowing them to send in feedback.

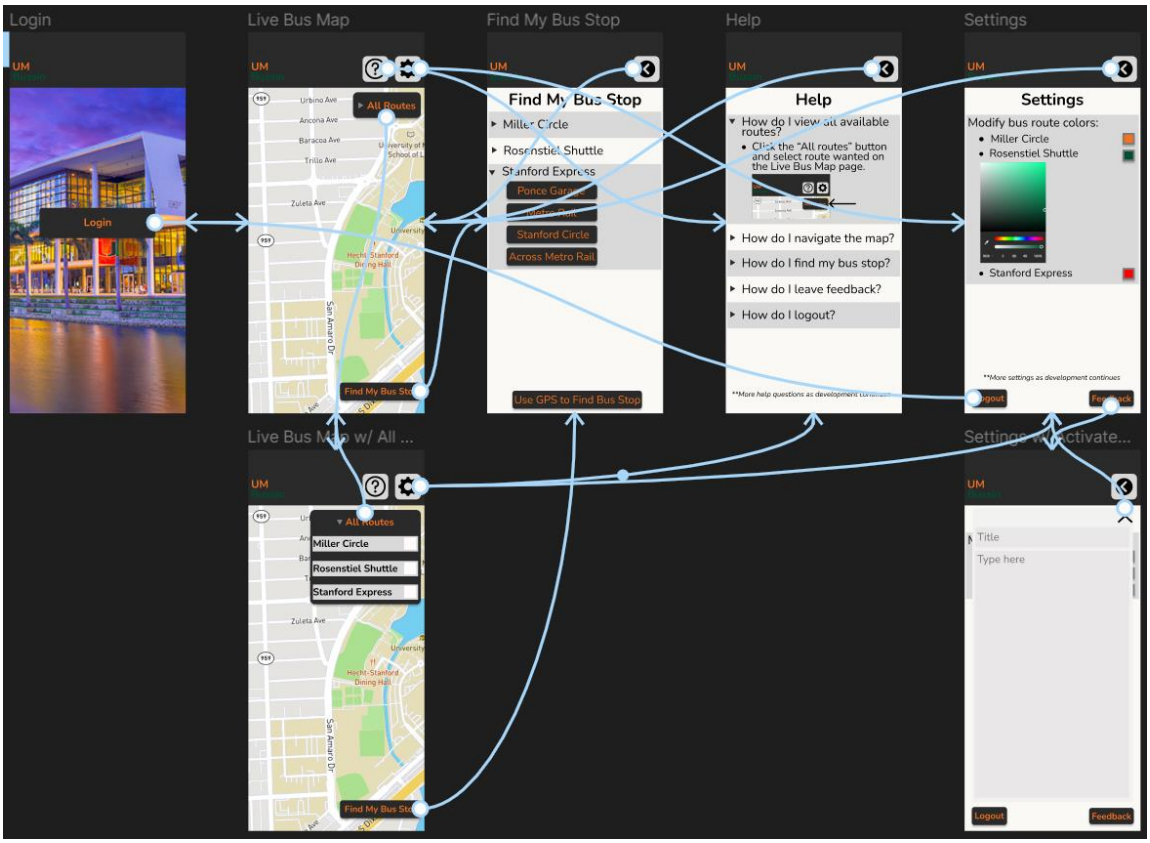

Figure 15: App Use Flow

14. Tech Stack

14.1 Front End Technologies

The front end will be done by using TypeScript and React Native. Our two reasons are for easy connection to the API, as well as cross platform capabilities. Since we want the app to be used by all students, creating only an iOS or Android app seemed like a waste, and while making native apps for each platform will create a performance boost, it is not the kind of boost we need as our app does not require any kind of complex computation or rendering. Secondly, since React Native apps have two threads, a main and a js thread, it will be simple to handle requests and responses from our ExpressJS back end.

14.2 Back End Technologies

As it was hinted earlier, the back end for this project will be done by using an API with ExpressJS. With ExpressJS we can handle requests from the app and create responses in a JSON format. Also, ExpressJS can utilize prisma to create queries for our database, format the data, and send it to the app as a response.

14.3 Database Products

For our database, we will be using an SQL database as this fits our needs more. Since we will not be handling many transactions, but instead relational data such as which bus goes on what routes and which drivers drive which bus, we have opted to use an SQL database that will store the information the API requires immediately. Additionally, we will have a time series database that will hold observational data. In the time series data, we will keep track of how users interact with the app as well as the bus system to optimize anything if we see that there could be improvement. The data here will be amount of people in a bus, amount of people waiting for the bus, and which routes and stops are more congested and at what time.

14.4 DevOps Tools

Our DevOps tools are defined by the nature of the project, starting with testing. Since we are using TypeScript for both front-end and back-end, we can use Jest for unit testing. Then for version control, we will be using Git and GitHub, as well as CircleCI for pipelining since this is easily implemented with GitHub. For our database, we plan on using Oracle Database as this will provide all the functionality we need for our SQL database. Lastly, we will use the Google Cloud Platform to host both the staging and production environments for the API, and we will have the app available on both the Google Play Store and the Apple App Store with its unique app id, where they are also stored in order for users to download.

15. Work Breakdown Structure (WBS)

Figure 16: WBS Layout

16. Project Timeline Plan

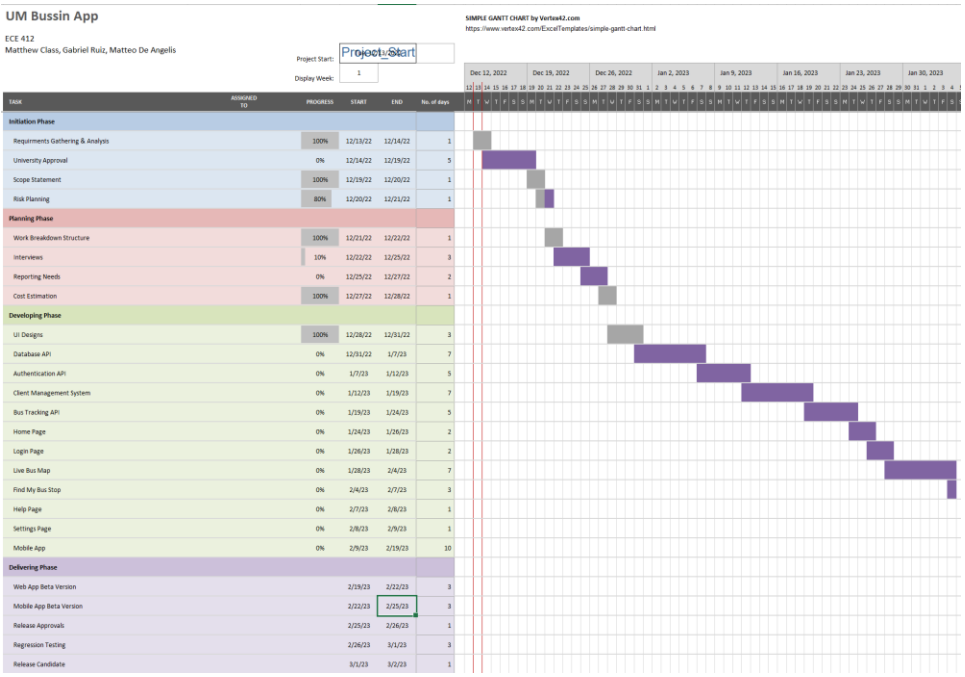

Figure 17: Full Project Timeline

17. Project Cost Plan

UM Bussin App ECE 412

Project Lead: Matthew Class, Gabriel Ruiz, Matteo De Angelis
Start Date: 12/12/2022

		BUDGET	ACTUAL	Under(Over)
Total		$ 4,125.00	$ -	$ 4,125.00

WBS	Task	Subscriptions Year	Subscriptions Rate	Materials Units	Materials $/Unit	Fixed Costs	Budget	Actual	Under(Over)
1	**Software Costs**						$ 1,125.00	$ -	$ 1,125.00
1.1	Application Software User Licences		$125.00			$125.00	125.00		125.00
1.2	Database User Licences		$0.00			$0.00	-		-
1.3	Operating System		$0.00			$0.00	-		-
1.4	Additional Security Applications		$1,000.00			$1,000.00	1,000.00		1,000.00
2	**Hardware Costs**						$ 3,000.00	$ -	$ 3,000.00
2.1	Servers			2.0	$1,000.00		2,000.00		2,000.00
2.2	Emitters			10.0	$100.00		1,000.00		1,000.00
2.3	Storage			0.0			-		-
3	**Network Costs**						$ -	$ -	$ -
3.1	Routers			0.0			-		-
3.2	Wireless			0.0			-		-
3.3	Switching Devices			0.0			-		-

Figure 18: Full Project Cost