Final Project Document

Jacob Hammond

Johns Hopkins University

EN.525.743

12/11/2023

Contents

1. Project Description

Commuters that utilize public transit recognize how important it is to accurately estimate travel times. A slight miscalculation can completely comprise what was supposed to be a quick and efficient commute and lead to delays and consequences such as missing appointments or being late to work. It is common for public transit providers to post schedules for buses and subways, but many riders will testify that these schedules are rarely accurate and seldom even followed. Delays, traffic, and a variety of factors can easily shake up the planned schedule and have a butterfly effect causing the entire days' worth of departure and arrivals to inch later and later. Many riders attempt to avoid this headache by using third-party map applications, which attempt to fetch real-time transit data and estimate the most efficient route – but these depersonalized tools are also often inaccurate and are usually suited for providing an estimate based on historical trips giving users only a best guess of what route they should take.

This project proposes an embedded system that can accurately calculate commute information in real time for a single user that is personalized and programmed to handle real-time calculations of the most used routes. The system would provide a level of granularity lacking from other tools by targeting specific routes to provide actual optimization of those routes, rather than estimations or best guesses. The system would calculate departure times and notify the user in near real-time if there are changes, delays, or better routes available without the user having to view schedules, or manually refresh a myriad of transit pages.

1.1. Problem Statement

Commuters that deal with this problem have likely recognized that one of the only methods to solve this problem is to forgo third-party map applications and instead reference multiple public transit pages to determine the following:

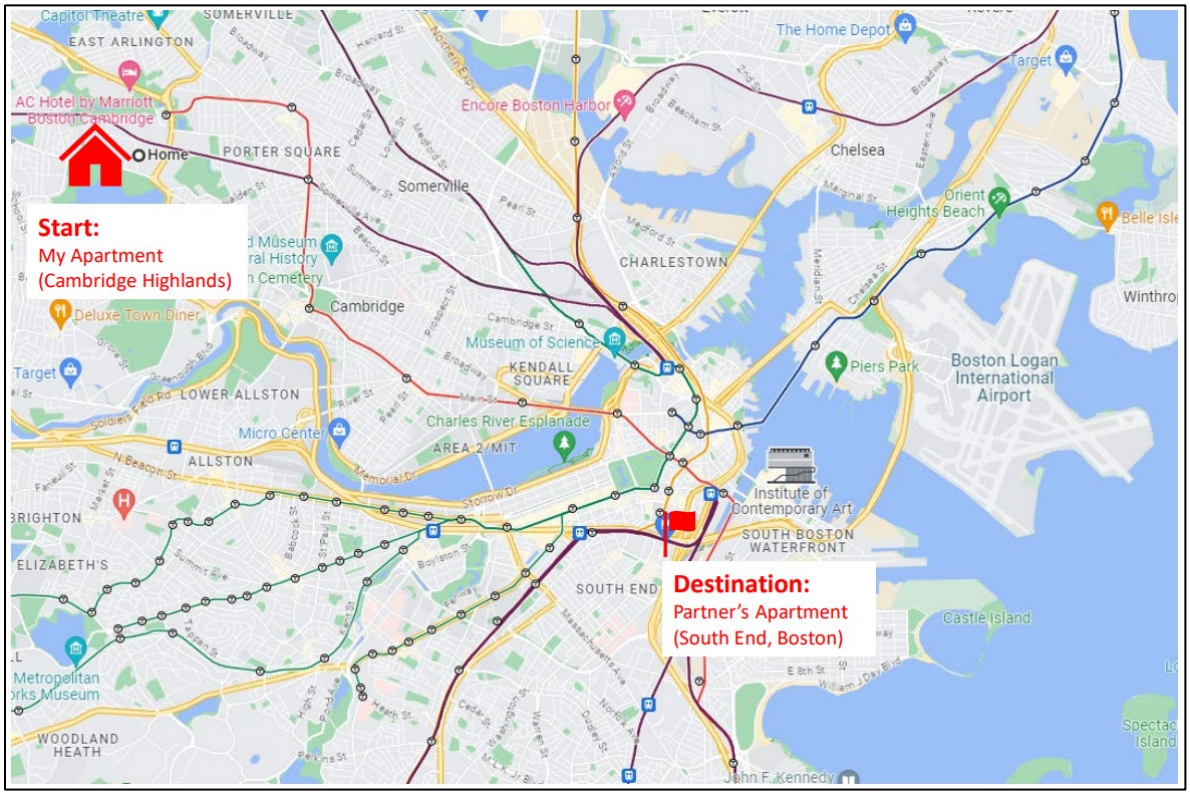
- When does the next bus/train arrive?
- Will the bus/train make the connection at a subsequent changeover within reasonable window?
- If so, what exact time should one walk to the nearest bus stop/subway station and minimize waiting without missing the departure?
- If not, what alternate routes are available, and ask the questions once again for the next route.

Finally, after reviewing all possible routes, it is still up to the commuter to manually determine the combination of all variables to determine which route is optimal with respect to the fastest estimated

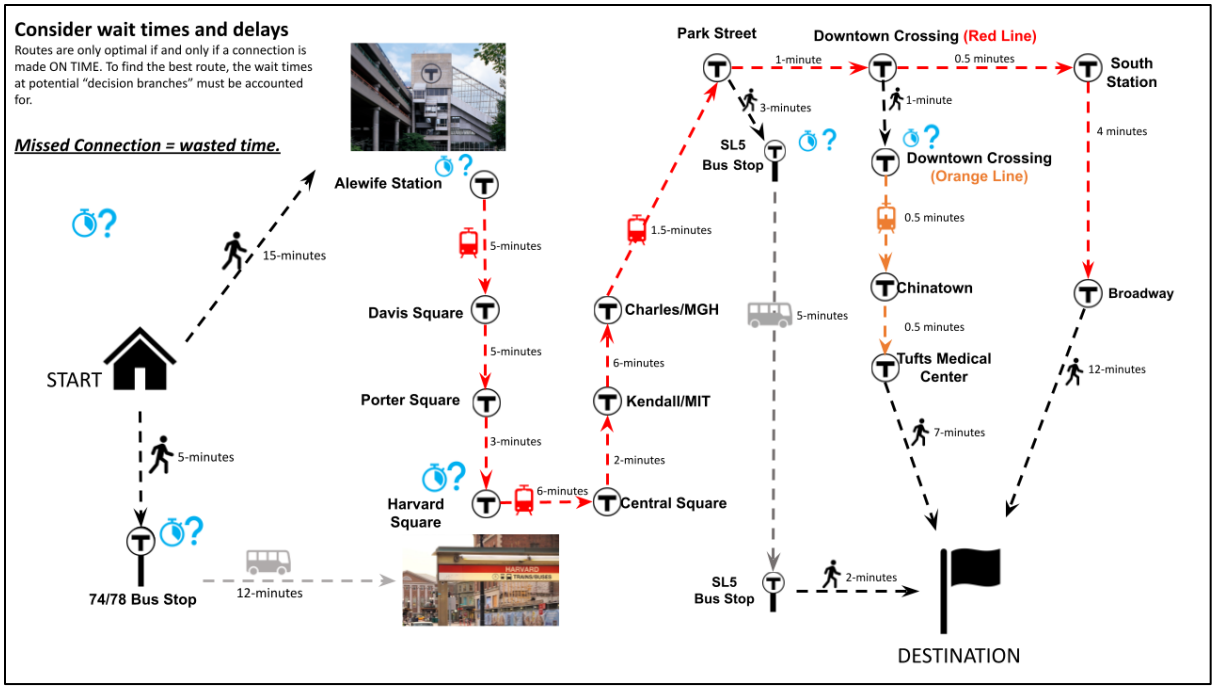
arrival or shortest transit time. This process is cumbersome and takes several minutes to flip through pages and map our routes and then compare them. Once a commuter has decided on a route, the best they can do is hope that there are no delays or changes before departure unless the process is repeated at regular intervals to account for delays or speedups of the multiple modes considered in the respective routes.

1.2. Case Study

To demonstrate the problem, the following case study is provided to showcase an example starting location and destination in Metro Boston utilizing the Massachusetts Bay Transportation Authority (MBTA) transit system.



This example scenario has six possible transit routes that combine multiple modes of transportation including several options for walking, taking the bus, and taking the subway.



It is clear from this case study how this problem can become somewhat of a "traveling salesman problem." The traveling salesman problem is a popular combinatorial optimization problem that asks for the shortest route for a salesman that needs to visit multiple cities exactly once. This problem is computationally hard because the number of possible solutions grows exponentially with the number of cities (Paschos, 2014). This transit case is similar in that the "cities" in this situation can be represented by the various subway stations, bus stops, and walking routes. In this case, one might consider what this means for the six possible routes shown – the problem must consider the following variables and their respective departure, arrival, transit time, and delays:

- 5 wait times between modes/changeovers.
- 4 possible bus routes.
- 2 subway lines with variable departures.
- 14 different subway stations.

With a total of 25 variables to consider, this problem becomes very difficult to compute manually or even computationally. To find the most efficient route, one must consider all possible routes, calculating all possible wait times and departure times, find the total travel time, then compare that result with all other possible routes to find the best one. While this sounds challenging enough, it should be noted that arriving at the answer must be done relatively quickly and constantly updated since transit systems are subject to real-time changes.

Given this complexity, one might imagine why a public transit provider, or a third-party maps application cannot possibly compute all these variables for every possible route, for millions of riders - each with different destinations, in real-time and always produce an accurate answer. It requires extreme levels of flexibility and detail to optimize transit routes for large-scale user bases which is typically achieved via node-based estimation on high-performance computers and usually relies on estimation which is only accurate to a certain degree (Church, & Niblett, T. J. 2020).

1.3. Scaling the Problem

Despite the difficulty of this problem, it can be feasible to solve it at a smaller scale. To scale down this problem to a manageable size, a caveat to consider is that most users of public transit tend to take the same few routes repeatedly.

With this assumption, one can effectively reduce the number of unknown variables in this challenging problem by limiting computation of the most efficient route to *only a select few common routes*, and eliminating variables that are already known and do not require estimation. Rather than create a universal oracle that can optimize all possible transit routes, a plausible solution could be an embedded system that relies on given assumptions and tolerances and only takes a few changing variables into account for a small static set of common routes.

Scaling the problem down in this manner reduces the complexity of the computation enough that it could be automated. The scaled problem can be adapted to run as an embedded system optimized for a single user and their most used transit routes.

1.4. System-Level Description

An effective embedded system can use a variety of techniques in combination to solve this problem at a reasonable scale. Rather than a universal transit solver, this embedded system will be programmed and implemented specifically for a single user, using as many known variables as possible to lighten the computational load enough for a modest computer to be able to compare and optimize the best routes via public transit. The development strategy would follow a typical path of subsystem/subroutine development and starting off with historical data. Transit arrival and departure schedules, and the associated routes can be planned to begin training and optimizing a model that caters to the unique user. Beginning the development cycle with by characterizing and organizing

canned data analysis doubles as a stand in for training and validation right before the system would be expected to operate in real-time with real data.

Once the embedded system is comfortable with performing mock scenarios and operating with the ability to solve and provide outputs in a timely fashion, a transition to real-time data is the next logical step. The usefulness of this embedded system depends on the ability to update information in real-time, so that means aggregating and data from public transit providers in the form of schedules, departure and arrival alerts, and GPS location data from buses and trains. Since the routes are going to be limited in number, the computational workload will not focus on finding and comparing all possible routes. Instead, it will simply brute force solve each pre-defined route and compare output data. It will use simple algorithms to predict future arrival and departure times, considering the different possible modes of transportation in each route.

If the system can perform this in "near-real-time", meaning fast *enough* that a user can comfortably live with latency between outputs within a reasonable time window and be updated regularly or with any changes, it will have served its purpose.

The user-interface to the system is necessary to consolidate the data and provide a convenient way to access the functions as needed. The embedded system must have the ability notify the user of changes, delays, or better routes available at regular refresh rates, and be easily accessible and ease the burden by being convenient.

A fitting name has been assigned to the project that describes the overarching function of this system: DIRECT – "Digital Interface for Real-time Estimation of Commute and Transit" and will be referred to by the acronym throughout this design review.

1.5. Requirements

To meet the system level description functionality, the following provides a list of derived requirements and capabilities for the system.

1.1. Category: Computational Outputs

1.1.1. The system shall compute an optimal transit route upon request.

1.1.2. The system shall continuously compute and provide an optimal transit route no later than every 60 seconds.

1.1.3. The optimal transit route shall be defined as the route with the soonest estimated time of arrival (ETA) from a requested departure time.

1.1.4. The system shall compute optimal routes for at least four predefined starting and ending points.

1.1.5. The system shall provide the legs or path of the optimal route consisting of subway lines and/or bus lines and/or walking.

1.1.6. The system shall compute a predicted departure time required to start the optimal route.

1.2. Category: Computational Inputs

1.2.1. The system shall query the MBTAv3 API for real-time input data using the JSON Requests format.

1.2.2. The system shall accept, and parse data returned from the MBTAv3 API using the JSON Requests format.

1.2.3. The system shall support MBTAv3 API data categorized as:

1.2.3.1. Alerts

1.2.3.2. Predictions

1.2.3.3. Routes

1.2.3.4. Schedules

1.2.3.5. Stop

1.2.3.6. Trip

1.2.3.7. Vehicle

1.2.3.8. Vehicle.GPS.Location

1.2.4. The system shall query the MBTAv3 API no later than every 60 seconds for updated data.

1.3. Category: Human Interface Inputs

1.3.1. The system shall provide a means for a user to select a predefined route.

1.3.2. The system shall provide a means for a user to set and define a requested departure time.

1.3.3. The system shall provide a means for a user to select audible notification enable/disable.

1.3.4. The system shall provide a means for a user to start or trigger the system to begin computation.

1.3.5. The system shall provide a means for a user to reset the system and stop computation.

1.4. Category: Human Interface Outputs

1.4.1. The system shall provide a visible display of estimated departure time.

1.4.2. The system shall provide a visible display of ETA.

1.4.3.The system shall provide a visible display of the optimal Route Legs/Path

1.4.4.The system shall provide a countdown display of time until estimated departure.

1.4.5.The system shall provide audible alerts consistent with user defined enable/disable.

1.4.6.The system shall refresh the display with updated outputs at least once every 60 seconds.

1.6. Limitations of the system

There are several limitations of the system that can be inferred from the requirements. It is helpful to explicitly list these limitations to clearly delineate the scope of the project and ensure that the requirements can be met at a minimum before adopting any additional functionality.

Requirement 1.1.4 dictates that the system compute optimal routes for at least four predefined starting and ending points. While this does seem like a small number of routes, recall that start and end destination can have many potential satisfiable roots. If it is assumed that each start and end destination has six possible routs as in the case study, one should consider that the system is required to query information from the transit provider API, aggregate predictions, schedules, delays, and GPS data, with its own pre-defined route and calculate the estimated arrival time by accumulating predictions over each segment of the trip, for each route. If this sounds complex, it only gets more convoluted the more routes you add and must define. For the scope of this system, it is reasonable to expect that a small embedded system would not have the ability to parse all this data and brute force calculate all possibilities fast enough to be real-time (or at least every 60 seconds).

For this reason, this project is catered specifically to the engineer designing it, and it is not expected to be a generalized tool available for use by others. It is application and user specific.

The system depends on internet access to fetch the latest transit data in real-time, and thus the system is limited to online use only. Finally, the system may not be able to monitor a route in-progress. Planning transit routes beforehand is difficult but consider a scenario where an unexpected delay or missed connection occurs for a traveler during a trip. It is not expected that the system will be able to monitor and reroute the destination during a trip and can notify the user remotely. As

such, the routes are planned from start point to endpoint and a segment of the route would be an entirely different model that needs to start from a midpoint that is not defined. It is expected and within scope that the system is strictly limited to pre-planning routes before starting a journey.

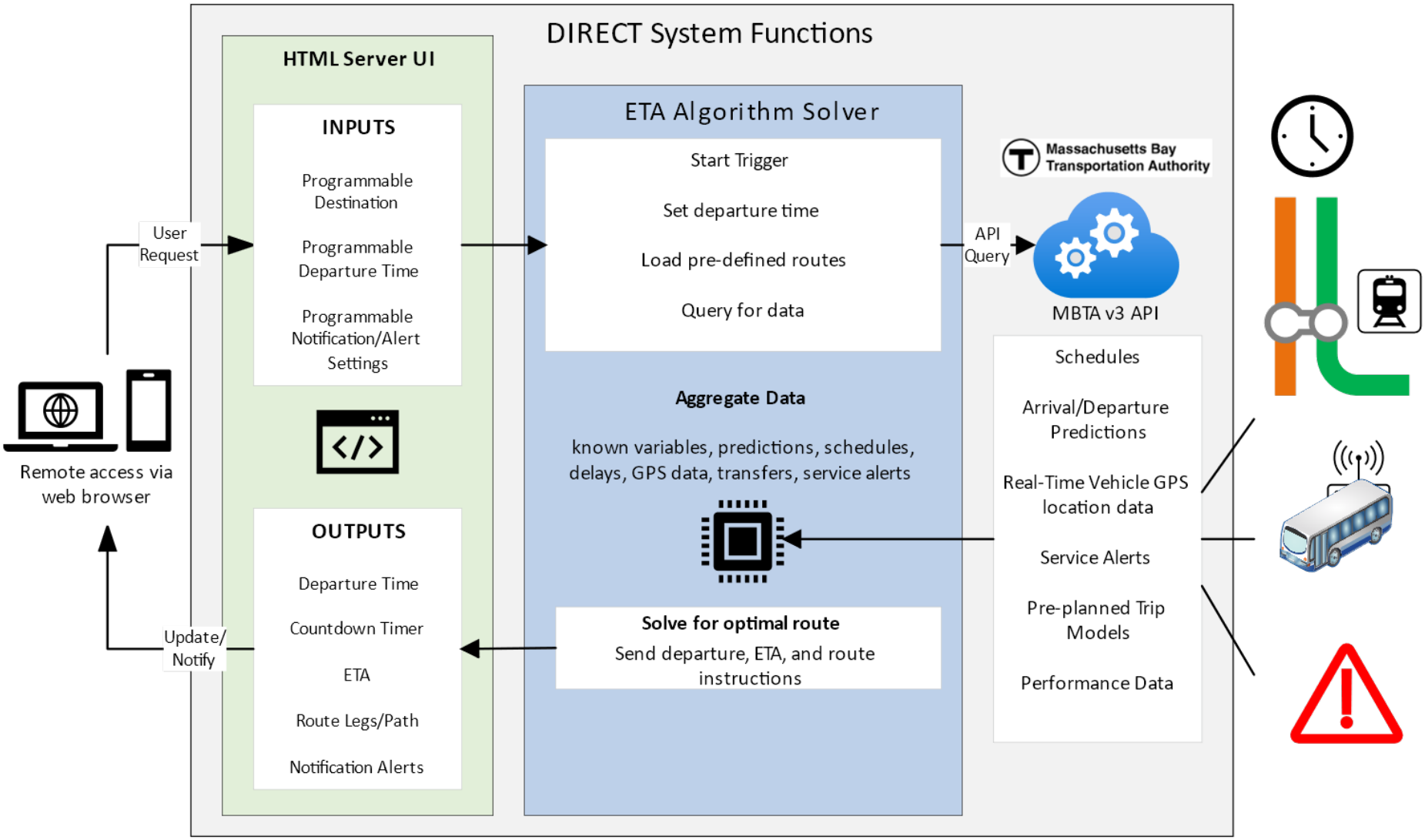
1.7. Benefits

DIRECT offers several benefits to its primary user, including more accurate ETAs, reduced hassle, and improved efficiency. By automatically calculating and optimizing transit routes in real-time, users would be free to make informed decisions about their commute and avoid being late all without the burden of constantly refreshing transit pages.

2. Functional Description

The embedded system architecture can be broken down into discrete functional blocks indicated in the system architecture. The design includes separated functions for input, output, a user interface, independent processes, and their relation to one another within the system as indicated by arrows. Items within the larger “Raspberry Pi” block are intended to be software functions, whereas the other items are designated as input or output devices and external interfaces.

2.1. System Architecture Block Diagram



3. Interface Descriptions

3.1. Internal Interfaces – Embedded Software

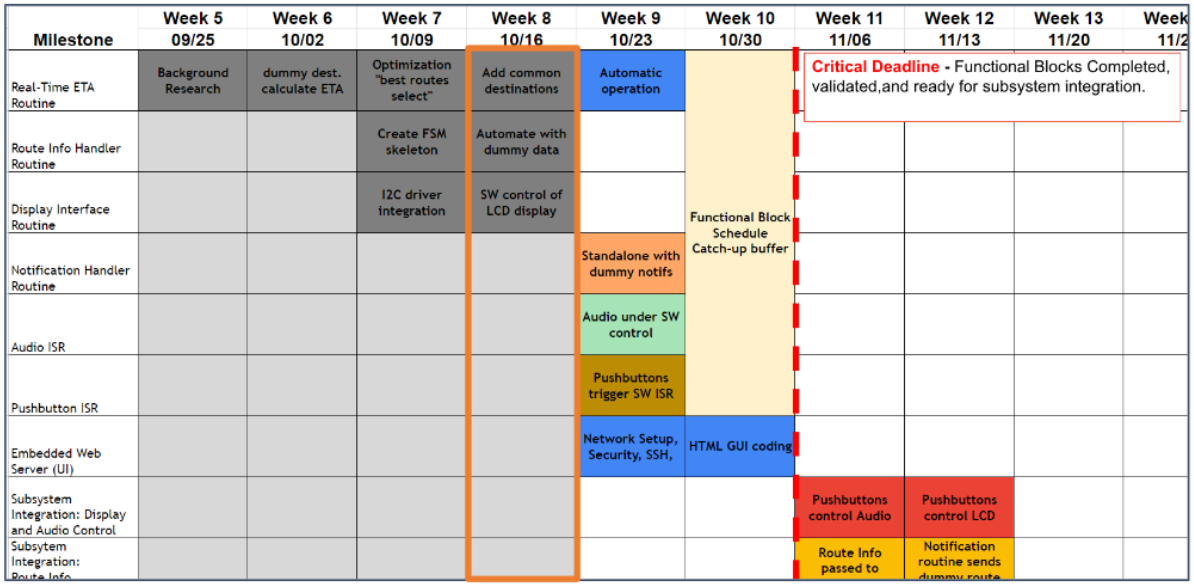
This project relies completely on embedded software. While the original implementation discussed in the CDR included hardware functions, as the project evolved, it became feasible and advantageous to rid the system of the physical hardware in favor of adding more capabilities.

The embedded software is a mix of HTML, javascript, python, and JSON files, however the “functions” of the embedded software instances themselves are well aligned with the diagram in section 2. A few software items are responsible solely for the user interface to collect user data, and display outputs to a user on an HTML webserver. The computational block is responsible for taking triggers, requests, and fetching data to aggregate and solve for an optimal transit route.

3.2. Removal of Hardware Interfaces from Initial Design

While the diagram in Section 2.1 shows the system at a high-level systems functionality, this project in its final implementation certainly stretches the definition of “Embedded System”. I take the definition in stride: an embedded system is a combination of computer hardware, computer software, and inputs and outputs to perform a dedicated purpose.

Originally, the system was going to run completely locally – with the computation performed by a raspberry pi, and the inputs and outputs small button and LCD peripherals that a user could interface with. It was clear that that kind of system had plenty limitations as time progressed.



The biggest motivator was about 4 weeks into the start of development when the project schedule had a task to integrate LCD displays with the Raspberry Pi. At this time, I had discovered that one 16x2 LCD screen did not have enough character space to print out the ETA, route, and departure time. At that point, the schedule was buffered with time to compensate for any delays in functional block testing. At this time, I decided to move to an HTML GUI based input and output system. With that setup it removed any constraints that would have held the user interface to the actual computer running the algorithm to optimize transit routes. The plan from then on was to use a tablet connect to my local Wi-Fi and allow the raspberry pi to host the embedded web server.

This began to spark some ideas and opened the possibility, time permitting, to allow for features that would make the system better to use and decentralized. One limitation of the hardware-bound architecture is that this system could handle routes departing from home, but since I could not physically access the system, the return journey could not have utilized the system.

Having a remove GUI seemed like a promising step in solving that problem. While I won't dive full into the implementation details in this section, it should be noted that this system is still parsing inputs, doing a computation for optimizing transit on hardware and software, and then outputting that result and responding in some way.

3.3. External Interfaces

While the embedded software for user interfaces and computation are query are internal to the architecture, the real-time estimation depends on access to API access to the following MBTAv3 API schedules:

- 78 Bus
- 74 Bus
- Red Line
- Orange Line
- Silver Line

Additionally, an external interface that is outside of direct system control is both power and internet availability. The embedded system will always require power and internet access to be functional.

4. Material and Resource Requirements

4.1. Bill of Materials - Consumables

While the initial project proposal included consumables for hardware purchases, the new design which relies only on embedded software, does not have any consumables. The computational model and webserver are agnostic as to *where* they are run or located, whether that be a laptop, a single board computer, or the cloud.

4.2. NRE and Development Tools

Non-consumable items required for non-recurring engineering and development include tools and items needed that are not directly part of the system. It is anticipated that all required development tools will be free and open source to keep development costs as low as possible. The resources are summarized as follows:

<u>Development Tools</u>

- Workstation computer
- (optional) IDE for software development (Visual Studio Code, free)
- Python 3 and Packages:
    - NumPy

- Requests
- Bs4 (BeautifulSoup)
- Azure-functions

Knowledge Bases

- User guide for LCD interface operation, or FOSS Linux/python high-level driver
- Optimization algorithms for Transit/route planning from published papers (free access provided by JHU libraries)
- Azure Documentation and Trainings
- JSON API documentation
- MBTAv3 API documentation

5. Development Plan & Schedule

5.1. Approach

The design for DIRECT is best thought of in “Functional Blocks” where incremental milestones are represented by the functionality of a single block. Functional blocks are then be integrated into functional subsystems. The subsystems are then combined to create the full system-level design. Integration and test are accounted for at the block-level, subsystem-level, and finally the full system-level as each milestone completion will require verification of functionality.

5.2. Updated Milestones & Schedule

A detailed proposed Gantt schedule was provided in the CDR, but as the project developed, certain functions and milestones were updated, removed, and changed with respect to task and schedule. The following is the updated Gantt chart that reflects the actual resulting schedule and milestones throughout the development from start to completion.

Milestone	Week 1 08/28	Week 2 09/04	Week 3 09/11	Week 4 09/18	Week 5 09/25	Week 6 10/02	Week 7 10/09	Week 8 10/16	Week 9 10/23	Week 10 10/30	Week 11 11/06	Week 12 11/13	Week 13 11/20	Week 14 11/27	Week 15 12/04	Week 16 12/11
Project Planning (PDR & CDR																
MBTAv3 API Fetch Departure Prediction					Manual Fetching Demo	Automatic 60 sec fetch				Functional Block Schedule Catch-up buffer						
Real-Time ETA Routine					Background Research	dummy dest. calculate ETA	Optimization "best routes select"	Add common destinations	Automatic operation	API wrap-up						
Route Info Handler Routine							Create FSM skeleton	Automate with dummy data								
Display Interface Routine							I2C driver integration	SW control of LCD display	Explore alternative display options	Implement new Display Interface						
Notification Handler Routine									Standalone with dummy notifs	Functional Block Schedule Catch-up buffer						
Audio ISR									Audio under SW control							
Pushbutton ISR									~~Pushbuttons defunt~~							
Embedded Web Server (UI)									Network Setup, Security, SSH,	HTML GUI coding						
Subsystem Integration: Display and Audio Control											Display control via tablet touchscreen	Audio Alarm ctrl by HTML GUI, global volume ctrl by tablet				
Subsytem Integration: GUI -> Python ETA Solver											HTML GUI passes route select to Python ETA solver	HTML GUI Accepts data from Python ETA solver	HTML GUI Accepts data from Python ETA solver			
Subsystem Integration: Python ETA Solver -> GUI											Trigger ETA solve when HTML GUI sends route	Python ETA Solver passes route/ETA data to HTML GUI	Python ETA Solver passes route/ETA data to HTML GUI			
Subsystem Integration: Physical User Interface														Setup Tablet Access to web server	Setup full remote Access to web server	
Full System Validation & Fine Tuning													Full System Integration	Testing & Validation	Fine Tuning and benchmarking	
Final Presentation															Presentation, Final Assembly Doc	Submission Due

6. Assembly & Detailed Design

This section provides the detailed steps required to do a ground-up-build of the DIRECT project along with relevant approach arguments, lessons, and results experienced along the way.

6.1 Development Environment setup

The source code for DIRECT is available on git at https://github.com/jhammo32/direct and can be clone directly. This project requires python 3 and the requirements for python3 packages and dependencies can be installed using "pip install requirements.txt".

While cloning the source code is an available reference, it is not required as code snippets and approach will be shared in the coming sections.

6.2 Fetching Transit Data

This project is based on collecting data from the MBTAv3 API which is a developer application programming interface that provides real time data of transit infrastructure in Boston, Massachusetts.

In order to use the MBTAv3API, you must first register for an account on their website https://api-v3.mbta.com/login and obtain an API key which allows you to make requests and train models on their API by allowing a large amount of requests that they would otherwise block.

Fetching transit data is a simple task with the API. The MBTA has a large amount of functionality that they cover in the API reference manual, but this project only requires a select few functions. The source code of this project is included in the appendix, but the following is a code snippet that utilizes json request to query the MBTAv3 API and get a vehicle departure time.

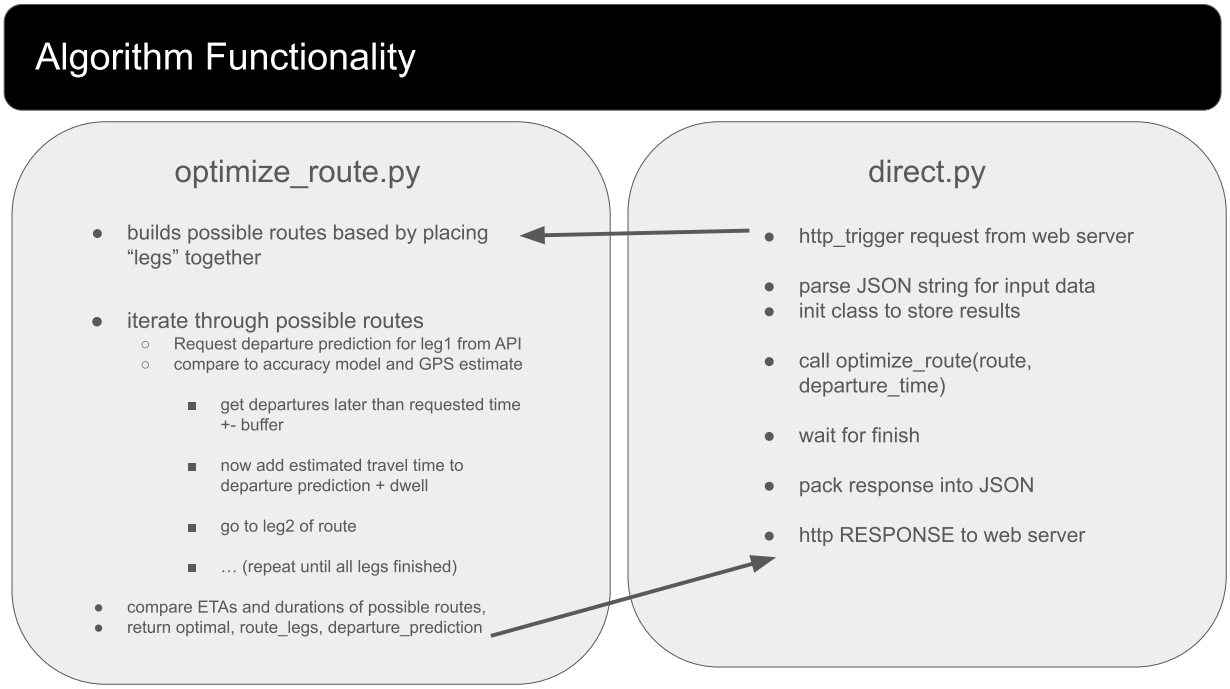
```
def get_vehicle_departure(STOP_ID, line, direction, overlap_time):
    # Make a request to the MBTA API to get the status of arrival times at the stop
    headers = {
        "x-api-key": API_KEY
    }
    response = requests.get(
        f"https://api-v3.mbta.com/predictions?filter[stop]={STOP_ID}",
        headers=headers
    )
    upcoming = []
    # Check the response status code
    if response.status_code == 200:
        # Success! Parse the JSON response
        data = json.loads(response.content)
        # for each element in the data array, get only those with relationships.route.data[0].id == line
```

```
        for element in data["data"]:
            # check if the element is for the correct direction
            if element["relationships"]["route"]["data"]["id"] == line:
                if element["attributes"]["direction_id"] != direction:
                    continue
                else:
                    # get the predicted arrival time of when vehicle is predicted to be at the stop
                    departure_time = (element["attributes"]["departure_time"])
                    if ((departure_time is None) or (departure_time == "null")):
                        # skip if null or no departure listed
                        continue
                    else:
                        # otherwise convert to datetime object
                        departure_time = datetime.fromisoformat(departure_time)
                        # append to upcoming_trains list
                        upcoming.append(departure_time)
        # if length of upcoming list is 0, return error
        if len(upcoming) == 0:
            return ("null", "null")
        # sort the upcoming list by soonest departure time
        upcoming.sort()
        # in the upcoming list, find the one with the shortest time difference greater overlap_time
        for departure in upcoming:
            # subtract one minute from departure for a buffer
            departure = departure - timedelta(minutes=1)
            delta = departure - datetime.now(departure.tzinfo)
            if delta > timedelta(minutes=overlap_time):
                #return the arrival time of the soonest departure within overlap_time
                return departure, delta
            else:
                continue
    else:
        return (f"Error: {response.status_code}")
```

This is the most common function call throughout the project which relies the ability to process some data, and do some calculations, and then parse through upcoming vehicle departure times to get an estimate on possible routes.

6.3 Route optimization and ETA

This is the main purpose of this project. Below is a high-level description the main two python functions and their responsibilities with respect to finding the optimal route.



The route optimizer works by creating the route from its segments from the ground up. Since most of the implemented routes start from my home, it turns out that many of the routes I take are identical to others for at least a segment or portion.

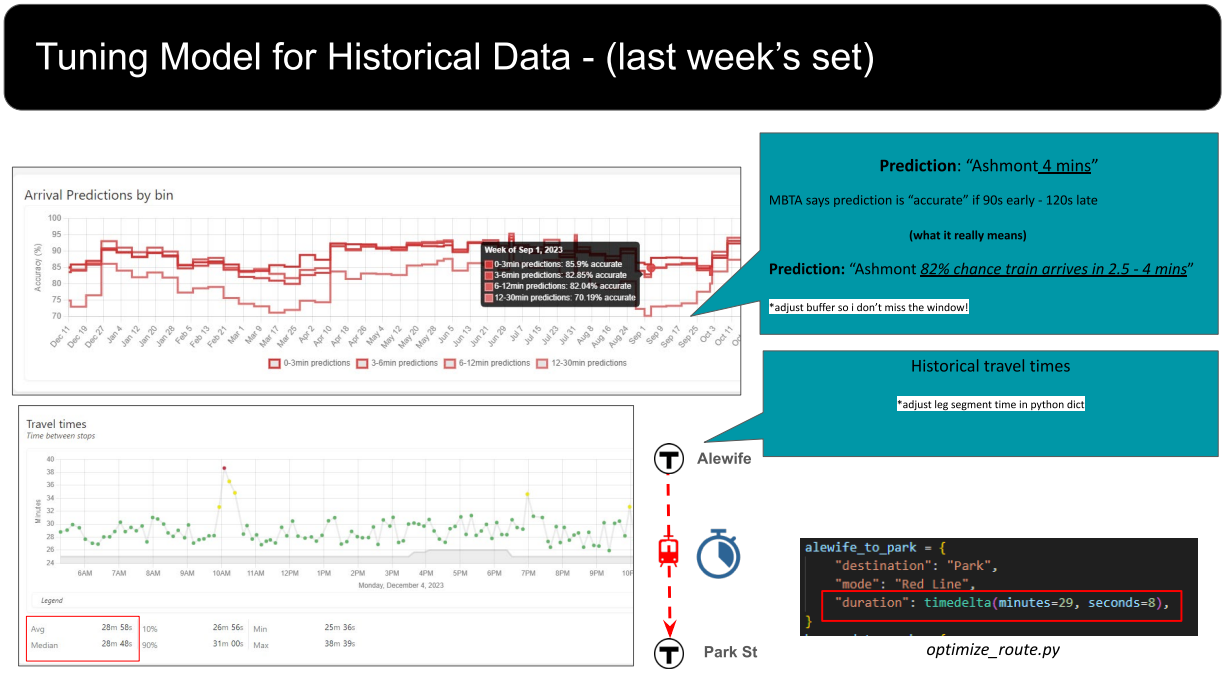
This fact is a large driver in the feasibility and performance of this project. One of the requirements is that the system run at regular refresh rates. As we covered in the case study, manually comparing every possible route gets very difficult very fast. It is to the model's advantage that the routes are segmented this way since it means a calculation doesn't have to be run again for another route that shares the same leg as one that was already computed.

An optional step that I did along the way through this project was apply weights to try to improve the model as time went on or I noticed a bug. Since at the time of writing this document the system has not had a lot of runtimes, there are likely many improvements to be made and still undiscovered bugs.

By far, the largest improvements I saw to the reliability of my model was the discover that MBTA predictions are not "accurate" in the common sense. In fact, the MBTA classifies any prediction as "accurate" if a vehicle arrives within some time windows as large as 6 minutes *later* than predicted and their model will still label that prediction as "accurate." This is obviously a silly thing to know and speaks volumes as to why relying on the prediction data alone has not proven reliable.

But discovering this fact during development lead me to introduce the idea of dwells and buffers. Those that take public transit know better than anyone else that buffer times between station transfers or bus/subway transfers take time. If you don't plan enough buffer time, 30 seconds late can mean a barely missed connection leaving you waiting for the next train or bus and 30 minutes later than planned. It is a frustrating scenario, but just like riders like that know, it is customary to plan for a buffer, especially if you're banking on transferring from a high frequency service mode (like the subway that runs every few minutes), to a bus that runs once per hour.

Below is an example of using MBTA data, discovering something new, and realizing that I can somehow compensate for it in the project to improve the reliability! This specific example shows why buffers are important when it comes to using MBTA departure values, and that the buffers must be weighted in the computation because the windows of prediction error are so large. Additionally it shows how I use the travel time data to get the duration deltas and load them into a segment/leg class of the model.



6.4 User Interface

The user interface was created as a simple HTML file and added very basic dropdown boxes to set div-ids to values. These can be stored directly in the HTML file when a user sets a document element so something else by interacting with the website. The HTML file also requires some added javascript to actually perform these functions when the user interacts with them. To accomplish this, the following javascript functions can be created to manage setting volume, handling a button push, and triggering both a “GO” and a “RESET”.

```
<!--Javascript Functions-->
  <audio id="alarm" src="alarm.mp3"></audio>

  <script>

      // Set notification volume
      function set_volume() {
          var select = document.getElementById("volume");
          var audio = document.getElementById("alarm");
          var selectedVolume = select.options[select.selectedIndex].value;
          if (selectedVolume === "off") {
              audio.volume = 0.0; // Sets volume to 0%
          } else if (selectedVolume === "on") {
              audio.volume = 1.0; // Sets volume to 100%
          }
      }

      //button click change
      function button_change() {
          var button = document.getElementById("start-btn");
          if (button.innerHTML === "START") {
              set_volume();
              refresh_loop();
              button.innerHTML = "RESET";
          } else if (button.innerHTML === "RESET") {
              location.reload();
              button.innerHTML = "START";
```

```
            }
        }
        var departureTime;
        //get_route JSON fetch
        function get_route() {
            // Get current time from dropdown menus
            var departureHr = document.getElementById("departure-hr").value;
            var departureMin = document.getElementById("departure-min").value;
            var departureAmPm = document.getElementById("departure-am-pm").value;
            var requested_route = document.getElementById("routes").value;
            var requested_time = departureHr + ":" + departureMin +":"+ departureAmPm
            //concatenate url https://fetcheta.azurewebsites.net/api/req
            var url = " https://fetcheta.azurewebsites.net/api/req?&route=" + requested_route +
"&departure=" + requested_time;
            // jQuery get
            $.get(url, function (data, status) {
                // Parse JSON data
                var response = JSON.parse(data);
                // Set route legs display
                document.getElementById("routeLegs").innerHTML = response.i_legs;
                // Set departure time display
                document.getElementById("departure").innerHTML = response.i_start;
                // Set ETA display
                document.getElementById("eta").innerHTML = response.i_end;
                // Set departure time for countdown timer
                departureTime = response.i_departure;
                // log status
                console.log("Data: " + data + "\nStatus: " + status);
            }
```

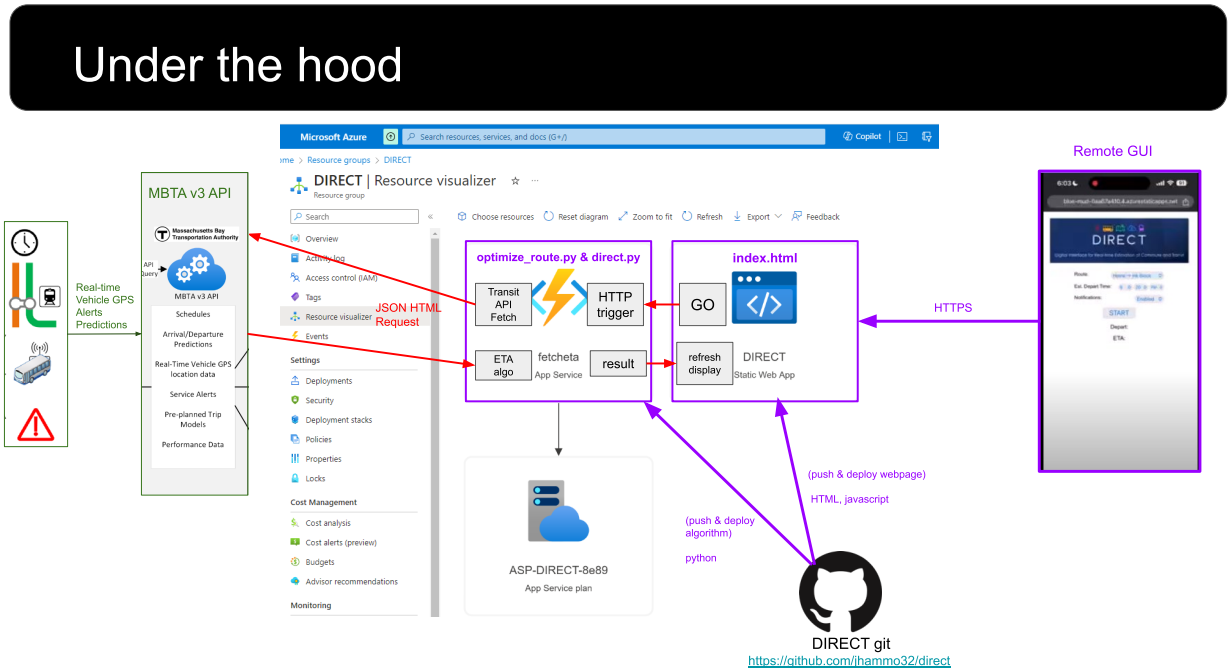
While this could certainly be improved, the functionality of the web server only exists to get the users requested route, the requested departure time, and pass those to the algorithm model to start working on the optimal route. Additionally, the webpage features notifications that will play an alarm when the departure time is reached. A counter is initialized large on the screen as soon as data is returned from the python model and starts to countdown to notify the user the time left until departure. ETA, departure time, and the route steps are shown (in the above example it can be read as 74 bus to 143 bus to 9 bus then walk 0.2 miles to destination).

6.5 Deployment to Cloud

If you clone the code directly from the repo, the DIRECT program should work out-of-the-box within VS-code. The IDE will allow you to host the webpage directly on localhost in a container within VS-code. Even though the HTML web server is not “online”, the instance is a host and if your local computer has internet access, the GUI is fully operational. While the original plan was to use a raspberry pi to run an apache web server to host the lightweight html server, I ultimately opted in using Microsoft Azure for several reasons.

First the Raspberry pi method would have constrained the web access to my local home network. I don't have the capacity to host my own website to the world wide web. I also did not want to leave any of my router ports open for remote access out of security concerns, so that method, while not an LCD screen would have still limited the system to being at home use only. I did entertain the idea of using a VPN also hosted on the Pi, but it seemed rather complex and cumbersome that I doubt I would be willing to connect to vpn and wait for it to authenticate to access the website remotely.

The answer came in Azure web services. There are some training videos that you see when you start an azure account, but the neat thing about the platform is that I was able to create "virtual" resources that mimic the functions, and without changing any of the code save a singe http trigger function, was able to drag and drop all the software as-is into the platform.



The above diagram shows how the functions interact with one another in the same way, only they are now located in the cloud and can be accessed anywhere. The web server was automatically assigned a unique website name and address, and the python functions are stored in small linux containers that run the python functions as-is anytime there is an http request. The only minor difference with this setup is that instead of hosting the webserver locally and "listening" or "serving" indefinitely, that part of the code is not needed in the cloud. Instead, Azure takes care of all cross-

resource requests natively and allows you to manage how resources can transfer data without setting up a server.

```python
import azure.functions as func
import logging
from direct import eta

app = func.FunctionApp()
@app.route(route="req", auth_level=func.AuthLevel.ANONYMOUS)

def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')
    # check if method is GET
    if req.method == "GET":
        # get the data from the query string
        route = req.params.get("route")
        departure = req.params.get("departure")

        # call eta function to get JSON update content to send as response to client
        response = eta(route, departure)

        # send response to client
        return func.HttpResponse(response, status_code=200)
    elif req.method == "POST":
        # get the data from the body
        route = req.get_json().get("route")
        departure = req.get_json().get("departure")

        # call eta function to get JSON update content to send as response to client
        response = eta(route, departure)

        # send response to client
        return func.HttpResponse(response, status_code=200)
```

To create the http_trigger in Azure, a function is generated for you automatically, but must be edited to call the python function of interest. In this case the main function is “eta()” and we parse the JSON data going into the function and coming out of the function and use this module as the data handler.

7. Performance and Validation

The DIRECT project has not been in use long enough to generate fullscale performance data. I plan to use the system after the completion of this project nonetheless and might find myself tweaking and updating in the future and tracking changes in git. In lieu of real-world performance data, provided are major validation steps for subsystems and integration tasks throughout the development cycle

- When “GO” is pressed on the GUI, the local eta.py program is launched with Python CGI (Common Gateway Interface).
  - using Event Listener (javascript) to monitor button press already
  - Python CGI provides standard for interfacing web servers with Python scripts

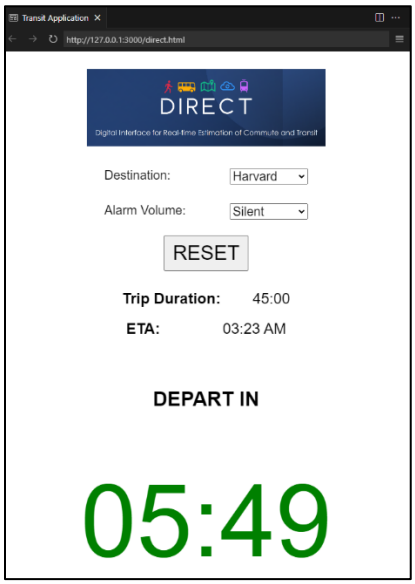
```
<!-- Start Button Event Listener -->
<div style="display: flex; justify-content: center; margin: 10px;">
<button id="start-btn" onclick="fetch_eta(); set_volume();" style="font-size: 2em; padding: 8px 12px;">GO</button>
</div>
```

```
function fetch_eta() {
    var button = document.getElementById("start-btn");
    if (button.innerHTML === "GO") {
        button.innerHTML = "RESET";
        // Get selected route from dropdown menu
        const selectedRoute = document.getElementById("routes").value;
        // Assuming departureTime is the time in milliseconds until departure
        //query eta algorithm her for transit time, ETA, and departure time
        <form action="/cgi-bin/eta.py" method="post">
        </form>
```

- Python outputs to terminal (for now):

```
The next Red Line Train from Alewife departs at 2023-11-13 18:57:00-05:00, in 0:23:06.258397.

The next 74 Bus departs from Concord opp Fawcett at 2023-11-13 18:49:50-05:00, in 0:15:55.847645.

The next 78 Bus departs from Concord opp Fawcett at 2023-11-13 18:39:50-05:00, in 0:06:04.473576.

The next Red Line Train from Harvard departs at 2023-11-13 18:48:34-05:00, in 0:14:39.085054.

The next SL5 Bus departs from Temple Pl. at 2023-11-13 18:59:41-05:00, in 0:25:45.713304.

The next Orange Line Train from Downtown Crossing departs at 2023-11-13 18:39:50-05:00, in 0:06:02.287738.

Fastest Route Duration is 0:42:28
ETA 07:16:23 PM

Route steps:
- Walk to Concord Ave/Fawcett bus stop
- 74/78 bus to Harvard
- Red Line to Park
- Walk to Temple Pl. Bus Stop
- SL5 bus to Herald St.
- Walk to Ink Block
```

Subsystem Validation - Creating an event listener to link the “GO” button and the python script launch



Functional Block Validation – Web GUI and counter functional with fake input data

Real-Time ETA - Best route optimization - COMPLETE
Sub tasks to complete milestone:

- Created python dictionaries (data structures) for all 20 possible "legs"
    - Destination, mode, minutes, seconds, dwell/wait time (using historical mean data for now, but will use API for realtime later)
- Constructed 6 routes using legs
- Calculated total duration of all routes, identified the fastest, printed duration, and the steps/legs of the best route
- Calculated dummy ETA using estimation using datetime objects in python

```
Fastest Route Duration is 0:45:16
ETA 08:11:46 PM

Route steps:
- walk to Concord Ave/Fawcett bus stop
- 74/78 bus to Harvard
- Red Line to Central
- Red Line to Kendall/MIT
- Red Line to Charles/MGH
- Red Line to Park St.
- walk to Temple Pl. Bus Stop
- SL5 bus to Herald St.
- walk to Ink Block
```

Route Info Handler - FSM Skeleton - COMPLETE
Sub tasks to complete milestone:

- Python code started (not fully functional but skeleton and transition conditions are set and structure if defined)

Testing the Best Route Optimization technique – Successful identification of shortest route

Web API Fetcher - Automatic 60 sec Fetch - COMPLETE
Sub tasks to complete milestone:

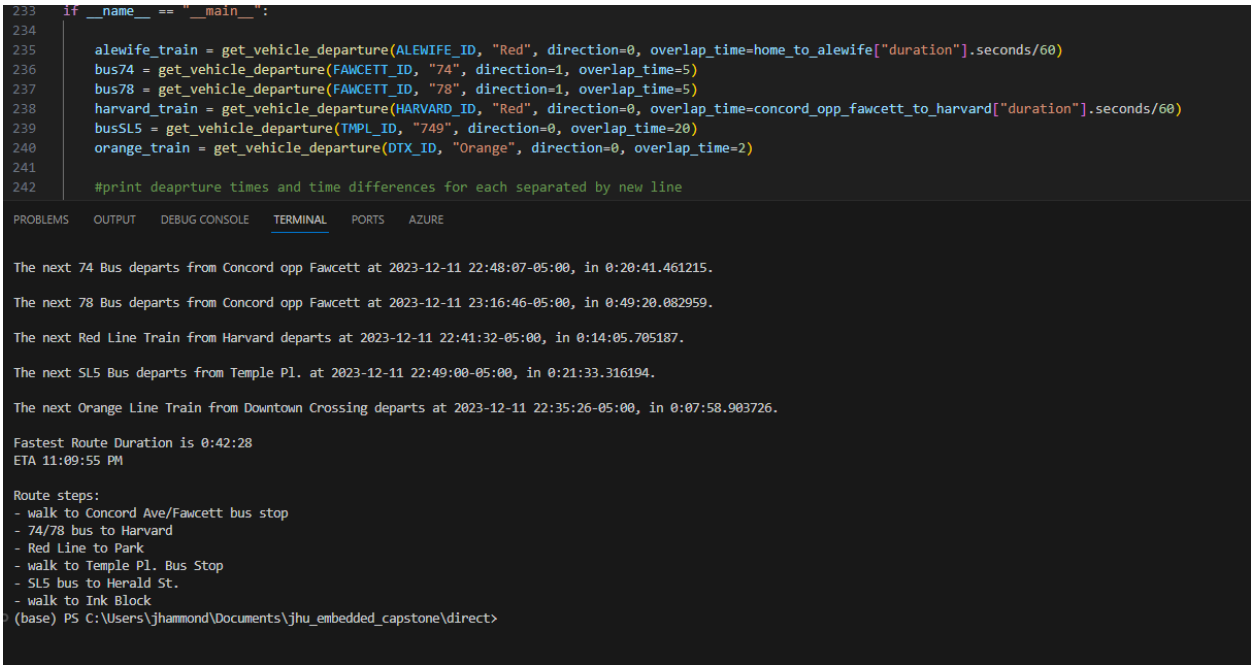
- Learned how to use the MBTA v3 JSON API from API documentation
- Created a developer account and requested API Access Token
- Learned how to use Python Libraries for JSON and Requests
- Created a standalone python script - fetch.py
- Verified by test that script successfully retrieves new data for both bus routes and subway route every 60 seconds while comparing to MBTA.com schedule dashboards for accuracy

real-time information.

- The V3 API uses the JSON API format, so you can get started quickly using any of the available libraries
- V3 API documentation is available using Swagger

```
# pseudo code sample for fetch.py
while (true):
        JSON_API_request(arrival_prediction) for
                STOP_ID = bus_stop #74/78 bus
                STOP ID = alewife_station #Red Line
                Parse_JSON_request(arrival_time)
                delta = arrival_time - current_time
                Print(delta)
                time.sleep(60)
```

MBTAv3 API Fetch Validation



Subsystem Validation MBTAv3 API Fetch used to get real departure data for all vehicles in routes, return is a now a real-time and accurate calculation of best route using the full model

8. References

Arduino. (n.d.). LiquidCrystal library. Retrieved from https://www.arduino.cc/reference/en/libraries/liquidcrystal/

Church, & Niblett, T. J. (2020). TRANSMax II: Designing a flexible model for transit route optimization. Computers, Environment and Urban Systems, 79, 101395–. https://doi.org/10.1016/j.compenvurbsys.2019.101395

Massachusetts Bay Transportation Authority. (2023). Schedules. Retrieved from https://www.mbta.com/schedules

Paschos (Ed.). (2014). Applications of combinatorial optimization (Revised and updated 2nd edition.). Iste Ltd.

Raspberry Pi Foundation. (n.d.). Software. Retrieved from https://www.raspberrypi.com/software/

TransitMatters Labs. (2023). TransitMatters Labs. Retrieved from https://transitmatters.org/transitmatters-labs

Zhao, & Zeng, X. (2008). Optimization of transit route network, vehicle headways and timetables for large-scale transit networks. European Journal of Operational Research, 186(2), 841–855. https://doi.org/10.1016/j.ejor.2007.02.005