TEAM BIOSPHERE

Final Report

April 26, 2021

Project Sponsor: Chris Doughty
Team Mentor: Andrew Abraham
Team Members: Kainoa Boyce, McKenna Chun, Gregory Geary, and Wesley Smythe

Table of Contents

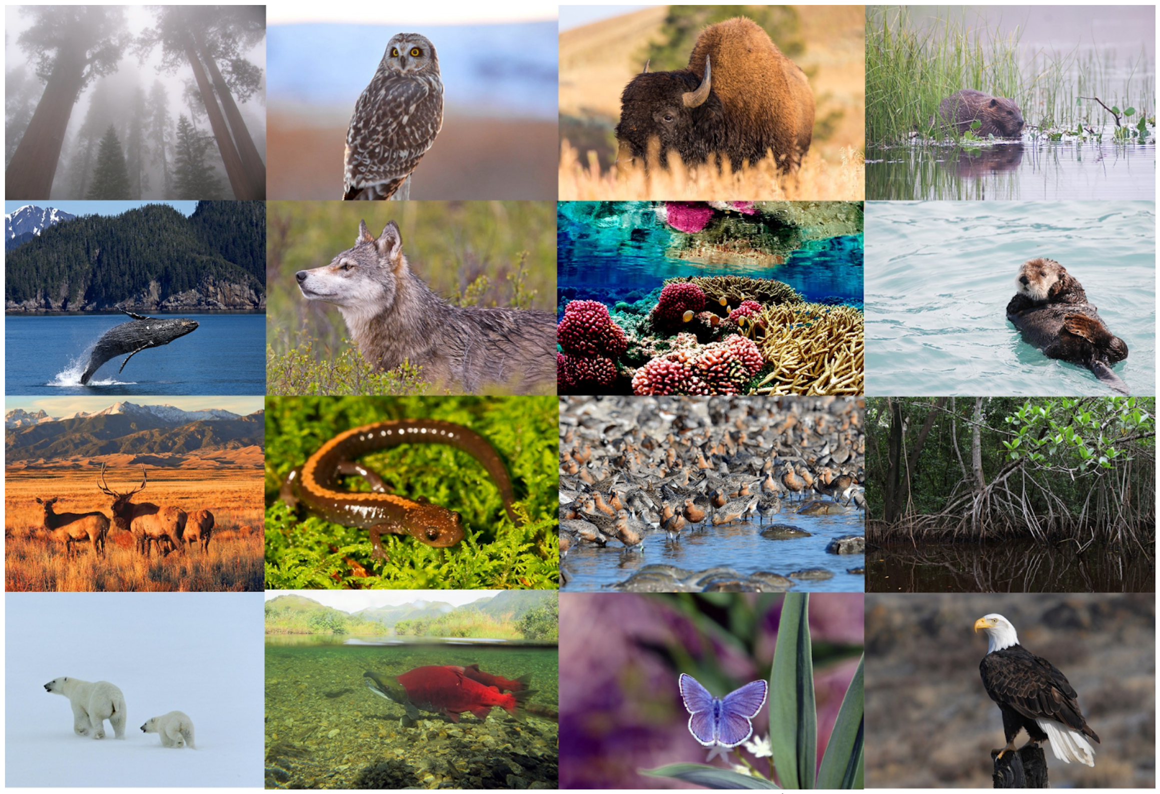


Figure 0: Earth's Biodiversity[1]

[1] https://medium.com/wild-without-end/international-day-for-biological-diversity-our-solutions-are-in-nature-b480e11fa193

Introduction

Tropical forests are biodiverse hotspots filled with many species of flora, fauna, and fungi. Thousands of these species in recent years have become extinct, in addition to this, globally there are one million species at risk of extinction[2] due to an influx of human activity in the wild. While the extinction of species is a natural process, the fact that species extinctions are up by one thousand percent since humans have gotten involved[3], is not. The root of this current mass extinction lies in a variety issues that have been unresolved for years, these include:

- Increasing Climate Change
- Illegal Deforestation / Logging
- Illegal Poaching

While these are not the only problems, they are some of the most well known and afflict much of the damage to biodiversity in tropical forest regions. Many scientists around the world have dedicated themselves to specifically researching and collecting information on this topic. From this, many solutions and models have been created in attempt to remedy the predicament that vulnerable biodiverse areas like tropical forests are in. One such model however, has stuck out the most in recent years due to its unique and beneficial data it provides, this model is **the Madingley Model.**

The Madingley Model is a revolutionary biodiversity and ecosystem model that has the capability to generate data over wide areas of land, limited only by the maximum surface area of Earth. The model is also able to include data for both oceanic and terrestrial areas and factor in a multitude of different scenarios, most of which are human driven, such as the aforementioned climate change or deforestation. For each of these scenarios, the model produces data on the interactions between different species in the environment. Figure 1 shows how the Madingley Model views and categorizes species into specific areas, and is able to then use this information for the scenarios it supports.

[2] https://www.nationalgeographic.com/environment/2019/05/ipbes-un-biodiversity-report-warns-one-million-species-at-risk/
[3] https://www.nationalgeographic.com/news/2014/5/140529-conservation-science-animals-species-endangered-extinction/

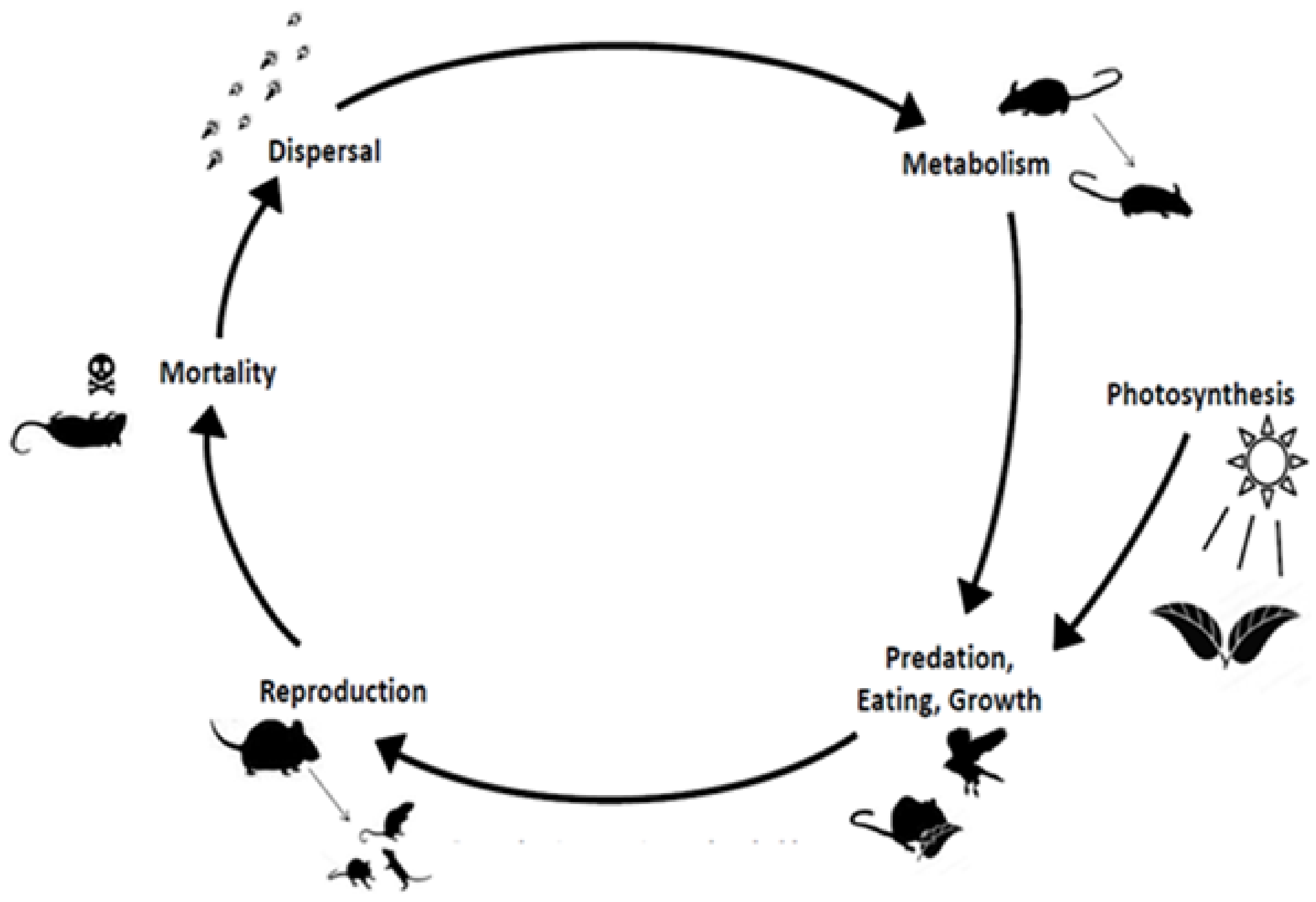


Figure 1: The Madingley Model Interactions[4]

Another feature of the model is the ability to predict data in the **future**, this allows scientists to see potential outcomes from the current standards in place. That is why the model itself was created with the intent of providing policy makers accurate biodiversity data, of which they could then use to influence the types of policies they put into place. However, due to the underlying fact that nature is incredibly complex, the data sets generated by this model are extremely large and very difficult to manipulate and interpret for the average human being.

To solve this, **Biosphere** was developed as an **Ionic Progressive Web Application** that allows the user to view different summaries of Madingley Model data in a cost effective, easy, and efficient manner, requiring little prior knowledge on biodiversity and

[4] https://madingley.github.io

ecosystems. The overall functionality and user experience of the application typically follows these steps:

1. The application starts by prompting the user to input various data (user-type, location, scenario, and intensity of that scenario)
2. These are then used to determine the specificity of the scenario data, where the user selects the desired scenario and the intensity of it
3. The application will then fetch the corresponding Madingley Model data to be displayed via a heatmap, line or bar graph, and tables with averaged data

The visualizations from this process allows the application to turn the previously visually unappealing data into readable and understandable formats that the user is able to retain and extract information from.

This overall goal of having users learn about the Madingley Model data helps to constitute a better application, by allowing us to maintain a consistent standard of performance for the product. With this one major goal in mind, other stretch goals and minor features were planned and implemented for a better user experience on the application. This included:

- Translations of the text into other languages
- Data exportation

While not implemented perfectly, they help to both enhance the user experience on the application, and help to increase the user base of the application to a wider audience. Therefore, we are able to spread the data from the Madingley Model more, and thus helps to further improve the application to fit the main goal.

All of the aspects and features from the application all have a specific usage and play a key role in the overall functionality of the application. Changing or removing a feature requires deep understanding as to how each of these features connect to one another, this can be read and will be explored further in this document in the upcoming sections.

Process Overview

After our requirements acquisition process, we were able to generate a general tech stack which operates as follows:

- Backend
  - Data storage
  - Data processing
  - Communication vector
- Frontend
  - PWA framework
  - Geolocation library
  - Visualization library
  - Stretch goal libraries

These general components were then split amongst the team, where each team member would be responsible for 1-2 components, which allows for overlap, and strengthens our overall component understanding. As a result of our component-based delegations, we each took the role of a *Subject Matter Expert (SME)* on top of administration responsibilities like: team leader(s), client communication, recorder, and release manager. With regards to the technical aspects of this project, the release manager is the only role of importance. The primary responsibilities of the release manager include: ensuring all team members are using the GitHub repository correctly, ensuring all versions submitting to the repository were merged, testing, and operable.

Once we were given the green light to develop, version control became useful to ensure compatible components.

Tracking issues that arose either within a localized development environment, or from the repository were tracked and discussed informally using a Discord server. Initially we hoped to use Trello to keep track of tasks and their status. However, it became clear that due to the frequency of our meetings, and meeting notes, it was easier to simply ask for status updates at each meeting rather than use a formal task management tool like Trello. If matters needed to be communicated during non-meeting times, then it was done using the team Discord server and dedicated channels per domain i.e. front-end and back-end. In our Team Standards, and Communication documents, it was stated that all directed messages must be responded or acknowledged within 12-24 hrs. As such, communication lines were always open and issues were able to be resolved off the clock.

Requirements

The first, and arguably most important step of the development process is the acquisition of requirements. This step involved meeting with our client on a weekly basis to brainstorm ideas about what they / and their stakeholders want and why. Since our clients are not experienced in software development, the conversation was aimed at the "why" aspect, while we are a team were able to determine what will be delivered. For example, our client initially told us that they wanted a mobile application because it would reach the largest number of users. As a team we determined that since their goal was to reach the largest target audience a progressive web application would be the best option, since it is compatible with all standardized browsers. Our client agreed with our decision. This back-and-forth progress went on for 2-3 months.
The overarching goal of this project was to provide the *target audience with meaningful information/visualizations generated from the Madingley model, in a fast, easy to use, and lightweight fashion.*
Using that statement, as well as conversations with our client during the requirement acquisition phase, we came up with six key components

- Data storage
- Data processing
- Visualization
- Graphics User Interface (GUI)
- Location services
- Data exportation

Given that our system was required to be lightweight for the end user, it was apparent that heavy lifting must be done off device, meaning we needed a cloud based, or dedicated hardware solution. Given that dedicated hardware comes with a considerable overhead, we decided to use an AWS backend which would store and process the data. Next, the user would need the ability to generate unique requests through an interface, this is where the GUI comes into play. The ionic framework was used to generate a universally compatible GUI for most devices, and all standard browsers. In order to generate unique requests, the user must first be able to select a geographic location of interest. Our client required us to provide 2-3 tools to help the users through this process. These tools include a manual entry form, an interactive drag and drop map modal, and location selection based on the user's GPS location. In order to satisfy the "meaningful information" requirement we generated at least two graphs for the user to interact with. The goal was to provide them with visualizations that would accurately and easily communicate the information that the user requested. In order to prevent unnecessary reruns of the scenarios, one of our stretch goals was to allow for data exportation, which was done by exporting the visualized components to PDF, which then gives the user the option to save, or message the visualizations to any platform of their choice.

In order to further satisfy the requirements that we were given we devised several performance, or nonfunctional based requirements which include: application access, general responsiveness, user/platform security, and language accessibility.

Application access refers to the applications ability to be accessed from around the world. In order to satisfy this criterion, we chose to host the backend of our application on Amazon Web Services (AWS) which has servers worldwide and guarantees 99.9999% service availability. As a result of this global accessibility, the application would inherently be responsive from an availability standpoint. However, this is only half of the puzzle. The other half revolves around efficient and non-repetitive code so that the user is able to process requests, and step through the application with as few limitations as possible. Next, in order to gain the user’s trust was important for us to develop a secure application. Thankfully the libraries that we were using require a security by default mindset, which allows us to develop the application with security as a forefront, rather than as an afterthought. The final performance requirement teeters the line between performance, and functional is language translation. This was important to our client to further expand the range of users from only English-speaking users to now: English, Spanish, French, German, and Portuguese. Numerically this would expand our reach from an estimated 360 million possible users, to over 1.3 billion possible users.

Architectural Overview

In this section, we will discuss the application components and how they all fit together. Our project will have four main components: AWS S3 bucket, AWS lambda functions, AWS API Gateway, and the Ionic interface. The S3 bucket will store all of the Madingley Model data. In order to retrieve the data, we will use the lambda functions. They will fetch the data and process it. After it has been processed, the API Gateway will transport the data from the lambda function to the Ionic interface. When the data reaches the Ionic interface, the user will get the output that resulted from their initial input. Finally, we will discuss some of the reasons why we selected each of these components.

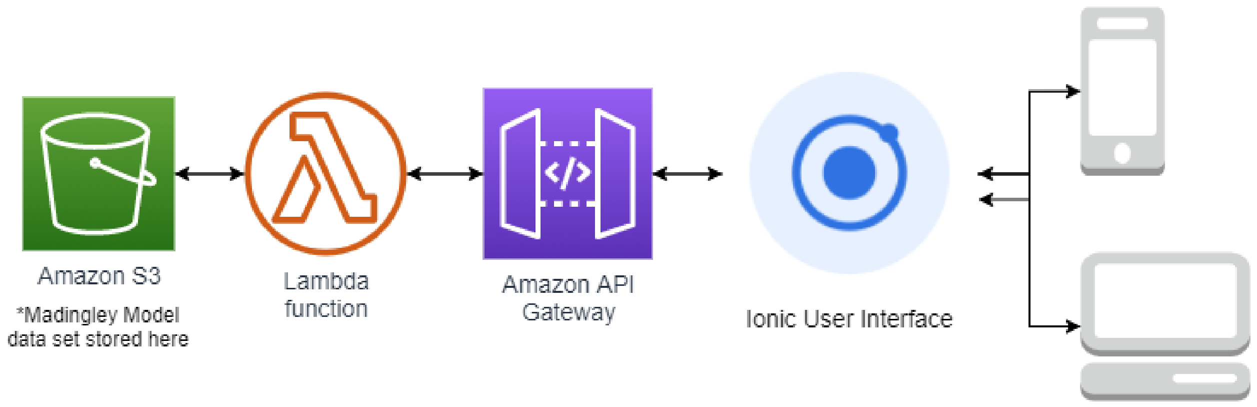


Figure 2: Architectural Overview Diagram

Backend Components

AWS S3 Bucket

An Amazon S3 Bucket is a data storage service offered by AWS that basically allows for users to store data of almost any type in a file-based system within the AWS Cloud, which makes for easy and efficient management. For our application, we will be using an S3 bucket to store the Madingley Model dataset that will essentially be used for displaying different biological stats in a user's given area.

AWS Lambda Function

The entire backend of our application will consist of a single AWS Lambda function. This is a serverless application that when prompted, will receive the data passed in by the user from the User Interface, and will use that received data to then parse and retrieve the corresponding data from our previously mentioned Madingley Model data set stored within an S3 bucket, process it, and send it back to the User Interface to be further processed and displayed.

AWS API Gateway

The Amazon API Gateway is essentially what connects the front end to the backend. The previously mentioned Lambda Function(s) will be mounted to an API gateway endpoint. This API uses REST(ful) HTTP methods to pass the necessary variables from our UI to the designated endpoint, in the form of a JSON to be received by Lambda. Once the Lambda function processes this input and pulls necessary response data from the Madingley Model data set, a REST(ful) response is then formatted and all visualization data is sent back to the UI.

Frontend Components

Ionic (Ionic v6 Angular v9) User Interface

The user interface being used for this application is Ionic. The Ionic framework uses universal web programming concepts to allow for one code base to be written which is then packaged into a progressive web application (PWA). This PWA format is able to be downloaded and run on mobile devices (both iOS and Android) and can also be used to generate a browser file pack that can be hosted and run as a responsive web application from any device with a modern browser (Chrome, Safari, Edge, etc.). Within our frontend framework, there were a few different subcomponents used to help fulfill the major requirements of our application. These are listed below:

- Ngx Translate - this node module was used to allow users to select from various languages we've built out compatibility for within our application. The module also used to detect the user's device language settings to automatically set the correct language upon loading of the application.
- Chart.js - this node module was used to build out almost all our data visualization tools within the application.
- Google Maps API - this API was used to assist in the development of the interactive map that allows users to easily choose a location by moving an adjustable circle over an interactive map of the entire earth.

Influences

Each of the components in our application’s architecture has been chosen through intensive research and comparison, and has only been chosen because we feel it is truly the best candidate for its job. The Ionic UI was chosen because we believe it allows for a smooth/efficient Cross-platform development experience, with the ability to not only share a single code base, but to share even individual written UI components that are then compiled differently to match whichever iOS, Android, or Web application interface is being built. The AWS API gateway, Lambda, and S3 backend services were all chosen because Amazon has the highest regarded web services due to them being extremely easy to integrate with each other. They are also very reliable, and easy to maintain especially for someone who doesn’t have a computer science background (via the AWS Console and account permissions we’ve built out). Lastly, AWS Lambda is one of the only/leading serverless application tools which gets rid of the need for servers and all the overhead associated with them, making it even easier for our clients to maintain.

Component Interactions

To provide a better understanding of exactly how each of these components interacts with each other, the application component interactions are described in further detail below:

AWS S3 Bucket <---> AWS Lambda

After the Lambda function receives and interprets the data sent from the UI, it will then pull the corresponding data from the Madingley Model data set stored in the application’s S3 bucket. This raw data will be in the form of a CSV, which will then be processed by the Lambda function to be packaged into a response and sent back to the UI via the API gateway.

AWS Lambda <---> API Gateway

After a request is sent via the Ionic UI, the API gateway will trigger an instance of our serverless Lambda function. The API gateway will pass the request data into the Lambda function to determine what Madingley data to be pulled from the S3 bucket. Once the lambda has pulled the desired scenario data from the S3 bucket, it will then return a REST(ful) response to the UI with that data in the body.

API Gateway <---> Ionic

Once the user has set all their desired variables/options, the Ionic application will use an imported angular HTTP client library to make a REST API call to the API gateway endpoint at which we've mounted our Lambda function to. This REST API call will be in the form of a POST method which is named so because it "posts" the JSON data that contains the desired user/location/scenario options to the body of the request. After the Lambda function handles the rest of the data pulling/processing, a REST HTTP response will be sent back to the front-end, and in the body of that response will be the data required to visualize the results of the imputed scenario.

Implementation

The actual implementation of our application went almost exactly as planned. Each of these components was implemented in our final application. The only issue we ran into was a strict time limit on the API gateway. Since the Madingley Model data sets consist of around 475 different csv file's each containing many megabytes of data, we found our API calls were timing out resulting in a CORS error instead of the correct data being returned. In order to work around this time limit, we implemented an algorithm we called the 'double onion' that parses what would be a single request body into many based off of the selected radius size and user type.

Instead of sending one request that contains a radius and a center to describe the entire circle the user would like to obtain Madingley data for, we split the single request into many containing a min and max distance from the center. The backend will then use the min and max distance to parse and obtain data strictly for that 'layer of the onion'. This workaround seemed to do the trick for the *general* user type (only parses 4 files). However, some requests were still timing out for the *policy maker* (needs to parse 10 files) and *scientist* (needs to parse 19 file) user types. To solve this problem, we took our algorithm one step further and had it parse the request bodies not only by radius, but by a file increment as well. This was to inform the backend to only search through a designated set of files at a time, for whatever desired radius layer (hence the double onion). Figure 3 provides a visual display of how it works.

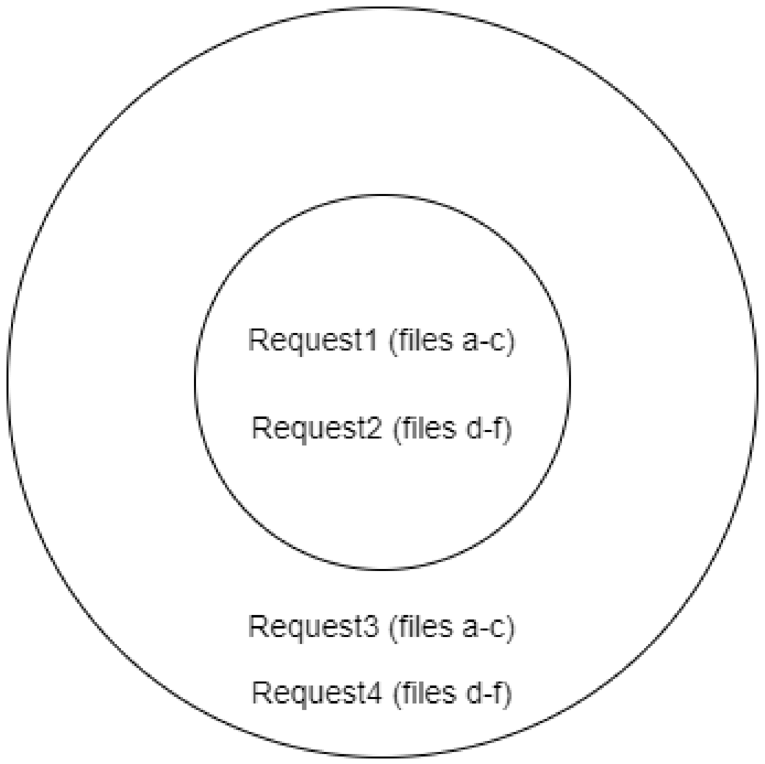


Figure 3: Layered Onion Approach Diagram

We were then able to control these two parsing variables (radiusIncrement and fileIncrement) to find a max number of files for the maximum sized radius layer that doesn't cause the API to timeout.

The example request body and resulting request array below helps to provide a better understanding of exactly how the algorithm works for a given example request, radiusIncrement, and fileIncrement:
Note: The resulting array would have 12 requests in it, but has been concatenated for viewing purposes

```
radiusIncrement = 400000;
fileIncrement = 3;
example_ request_body (before being parsed with algorithm) =  {
      file_end: 0
      file_start: 0
      lat: 35.198284
      lng: -111.651299
      max_distance: 781472.6625472013
      min_distance: 0
      scenario: "EXTINCTIONS"
```

```
     scenario_option: "Holocene"
     user_type: "policy_maker"
}
// After being parsed by algorithm:
requestArray = [
{
     file_end: 5
     file_start: 0
     lat: 35.198284
     lng: -111.651299
     max_distance: 200000
     min_distance: 0
     scenario: "EXTINCTIONS"
     scenario_option: "Holocene"
     user_type: "policy_maker"
},
{
     file_end: 9
     file_start: 5
     lat: 35.198284
     lng: -111.651299
     max_distance: 200000
     min_distance: 0
     scenario: "EXTINCTIONS"
     scenario_option: "Holocene"
     user_type: "policy_maker"

},
{
     file_end: 9
     file_start: 5
     lat: 35.198284
     lng: -111.651299
     max_distance: 200000
     min_distance: 0
     scenario: "EXTINCTIONS"
     scenario_option: "Holocene"
     user_type: "policy_maker"
},
{
     file_end: 10
     file_start: 9
     lat: 35.198284
     lng: -111.651299
     max_distance: 200000
     min_distance: 0
     scenario: "EXTINCTIONS"
     scenario_option: "Holocene"
     user_type: "policy_maker"
},
{
     file_end: 5
     file_start: 0
     lat: 35.198284
     lng: -111.651299
     max_distance: 400000
     min_distance: 200000
     scenario: "EXTINCTIONS"
     scenario_option: "Holocene"
     user_type: "policy_maker"
}.............
{
     file_end: 10
     file_start: 9
     lat: 35.198284
     lng: -111.651299
     max_distance: 781472.6625472013
     min_distance: 600000
     scenario: "EXTINCTIONS"
     scenario_option: "Holocene"
     user_type: "policy_maker"
}
```

The last sub issue that came up in implementing this algorithm properly was that as we continued to decrease our radiusIncrement and fileIncrement (in order to find that max for each that was guaranteed to be fully processed without timing out), The resulting request array length grew substantially, and since it would take upwards of 10 minutes for some request arrays to be fully processed and retrieved asynchronously, we decided, since one of the advantages to developing a backend using AWS Lambda is infinite scalability (meaning that the function can be spun up infinite times simultaneously), to implement a request structure that sends all requests at once without waiting for the previous request response to be received. This decreased our loading time substantially so that now all requests only take 20-30 seconds to be retrieved.

Aside from our ‘double-onion’ algorithm, all other components were developed as planned, and the overall development process went fairly smoothly with little to no issues.

Testing

In order to check that we correctly implement all the architecture listed in the previous section, we used a three-part testing strategy. The first part was usability testing. This checked to make sure that each individual part of our application worked perfectly. The second part was integration testing. For this version of testing, we checked to see if a small portion of the individual components could function together properly. Finally, we did some usability testing. This involved having different groups of users walk-through the application to see if there were areas of confusion or areas that worked really well.

1. Unit Testing

As one of the key components of our testing strategy, unit testing was used to break large pieces of software into individual components or functions. This ensured that each unit performed as expected. Unit testing was used both automatically and manually. In order for the unit tests to be validated, the team generated tests and recorded their expected outputs. It was important that these tests covered a wide array of possible inputs. Next, they were tested, and their results were analyzed. We didn't have any problems, but if there are failures in unit testing, it is usually caused by a logical disconnect between the code written and the intended output. The secondary purpose of unit testing was to find obvious issues like, unrestricted access to various parts of a system, or the mishandling of control or special characters. An example of this would be an SQL injection as a result of improper handling of the input. This would have allowed it to run as code, rather than constricting it to a string datatype.

For our project specifically, the backend was broken down into three major components: data storage and retrieval, data processing, and the back-end interface. Each unit was tested extensively with each unit containing its own criteria. The backend unit tests can be seen in Table 1. The frontend, similarly to the backend, was broken down into specific modules that each performed a specialized task. While there were many smaller modules included in our front-end, some of the major modules that were prone to invalid responses included: location selection and data retrieval. Like the backend, each unit was tested for a variety of scenarios, both successful and not. The frontend unit tests can be seen in Table 2.

Unit Test	Description	Boundary Values	Example Input	Expected Response

API: Single Valid	Generate a single valid response that is then passed through the API Gateway.	distance values cannot exceed 800km.	min_distance=0
			max_distance=200000
				{
				"Next": None
				}

API: Onion Valid	Generate an onion handled response that is then passed through the API Gateway.	distance values must be greater than 800km	min_distance= 2345
			max_distane = 340953
				{
				"Next": event
				}

Data Retrieval: Non-Existent File	Create a data request for a file that clearly does not exist.	A string that looks like a file i.e. some_dir/some_file.csv	get_object(file="fake.csv")	FileNotFoundError
Data Retrieval: Out-of-Bounds Request	Create a data request for a file that is located in a different file system	A file that exists. The file path must start from the root.	get_object(Bucket=wrong_bucket, file="file.csv")	FileNotFoundError ForbiddenAccessError
Data Validation: Correct Data Types	Create a request with a valid size and type to be tested against validation library	The request must conform to the restrictions of the requested input	{ "user_type": public … scenario: "CLIMATE" }	"[DEBUG]: Validation: Passed"
Data Validation: Incorrect Data Types	Create a request with a valid size and type to be tested against validation library	The request must be incorrect within the constructs of the validation functions	{"fish": "taco42"}	"[DEBUG]: Validation: Failed"
Data Processing: No Returned Data	A data request using semi-legitimate parameters. For a dataset that does not exist.	The request must be approved by validation functions	{ "user_type": public … scenario: "CLIMATE" }	{"statusCode": 400, "body": None}

Data Processing: Improper File Format	A data request using legitimate parameters for a file that is mislabeled, or of an invalid type	The request must be approved by validation functions	{ “user_type”: public … scenario: “CLIMATE” }	{“statusCode”: 400, “body”: None}
Data Processing: Invalid Values	A data request using valid parameters, for a non-numeric value	The request must be approved by validation functions	{ “user_type”: public … scenario: “CLIMATE” }	{“statusCode”: 400, “body”: None}

Table 1: Back-end Unit Testing Criteria

Unit Test	Description	Boundary Values	Example Input	Expected Response
Location Selection: Map Input	Receive input from Google Maps API selection, and send a data request to back-end	Latitude: [-90,90] Longitude: [-180,180] Radius: [0, 12000]	Latitude: 52 Longitude: 105 Radius: 1240	Request made to Back-end
Location Selection: Valid Manual Input	Receive manual location input, and send a data request to back-end	Latitude: [-90,90] Longitude: [-180,180] Radius: [0, 12000]	Latitude: 40 Longitude: 105 Radius: 2421	Request made to Back-end
Location Selection: Invalid Manual Input	Receive an invalid manual location input and produce an error response	Latitude: [-90,90] Longitude: [-180,180] Radius: [0, 12000]	Latitude: -105 Longitude: 185 Radius: -2752	Error: Invalid Input
Data Retrieval: Successful Data Retrieval	Request and retrieve the correct files selected from the location and scenario options	Value 1: 0 <= Val1 < Val2 Value 2: Val1 < Val2 <= 1625	generateRequest (0, 249)	Madingley Onion Data, statusCode: 200
Data Retrieval: Failed Data Retrieval	Failed request of correct data and display an error	Value 1: 0 <= Val1 < Val2 Value 2: Val1 < Val2 <= 1625	generateRequest (0, -515)	No Data, statusCode: 400

Table 2: Front-end Unit Testing Criteria

Finally, for the purposes of security, the back-end components were compared against the Common Vulnerabilities and Exposures (CVE) database to ensure that most known vulnerabilities were patched. According to preliminary research there were 115 vulnerabilities related to AWS and 494 vulnerabilities linked to Python for the entry dated on March 4, 2021. It should be noted that not all vulnerabilities were related to this application, and some were illegal to test without a Certified Ethical Hacker (CEH) degree or being contracted as a bug or vulnerability finder.

2. Integration Testing

In addition to unit testing, another form of testing that we used was integration testing. Integration testing looked for a seamless interaction between different components within our program. In particular, we needed the data storage, data visualization, API Gateway, and the Lambda functions to all communicate effectively with one another (seen Figure 4). After this part of testing, we didn't find any alarming or unexpected results.

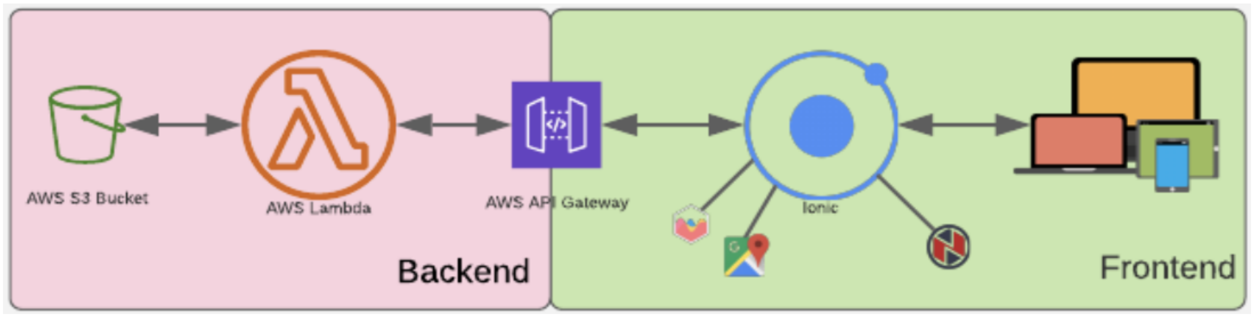


Figure 4: Interactions Between Different Components of our Project

Back-end: Data Storage and Retrieval

A key component of the backend involved requesting, and retrieving data files from the pre-specified file system. This was done through the lambda function that was given special, and limited privileges in order to promote the principle of isolation, and least privilege. In order to test the scope and limitation of this aspect, an integration test was completed.

Integration Test	Description	Expected Response
Perform a valid action specified by the Lambda's IAM policy	A request will be generated that is within the scope of the predefined policy	This test will be given a passing mark if it performs the action and does not return AccessDenied.

Table 3: Back-end Data Storage and Retrieval Integration Test Table

Back-end: Data Validation and Interpretation

Once the data was retrieved from the file storage system, it was validated and translated to a valid data type that was then handled by the remaining aspects of the backend.

Integration Test	Description	Expected Response
Take a valid data file and check its internal values.	Take a valid data file as an input, then check the internal values for column names, length, and cell type.	If this action is performed correctly then no errors should be returned, since cell values and column indices are hard-coded into the backend.
Take a valid data file and transform it into a non-networked data type.	Take a valid data file then perform the static and dynamic operations translating it from bytes to JSON.	If the actions are performed correctly then a populated dictionary object will be returned.

Table 4: Back-end Data Validation and Interpretation Test Table

Back-end: API Gateway and Lambda

In order to communicate between the frontend and backend, a gateway was used. The Gateway acted as a trigger for the back-end components. Once the lambda function was triggered it performed a pre-specified task. Then, it returned numerous headers outlined by AWS. These headers were then parsed by the frontend, so that they could be visualized.

Integration Test	Description	Expected Response
Generate an API Request and wait for the output.	Use the Live API Gateway or API Sim to pass a request to the lambda function to then be called, and data returned.	If the request is valid, and does exist, then it should return a response with either a statusCode of 200 or 400.

Table 5: Back-end: API Gateway and Lambda Test Table

Front-end: User Selections and Data Requests

The location selection module needed to pass three valid numbers for the latitude, longitude, and radius of the circle. This data then needed to be retrieved and stored. Once the user selected their user type and scenario options, the API Gateway created a request for the appropriate data on the backend.

Integration Test	Description	Expected Response
Collection of scenario and location data, then request data.	Different scenario data is selected by the user and stored, then used to request specific sets of data from the backend via the API Gateway.	A successful response will return no errors. Wrongly formatted requests will produce errors. Invalid input will cause errors elsewhere.

Table 6: Location/Scenario Selection and Data Requests Test Table

Front-end: Data Retrieval and Data Visualization

The API Gateway was used to retrieve data from the backend that was previously requested using the parameters selected by the user. This data, if successfully retrieved, was then used to dynamically generate graphics. These graphics included buttons, charts, and a heat map produced by the Google Maps API, along with an appropriate legend color scale.

Integration Test	Description	Expected Response
Retrieval of JSON data used to create visuals and other UI components.	The API Gateway gets the previously requested data that the backend has parsed and allocated for use. This data is then retrieved and used by the front-end to visualize the datasets using heatmaps, graphs, and other graphics.	If successful a series of requests will return a finite amount of data and a statusCode of 200. The graphics and other UI components will then be generated based on this input. Otherwise, it will return no data and a possible statusCode of 400.

Table 7: Data Retrieval and Data Visualization Test Table

3. Usability Testing

Besides unit testing and integration testing, a third type that we used was usability testing. The purpose of usability testing was to test the interactions between the application and the target audience. This type of testing focused on the overall quality, and intuitiveness of the application. It simulated what a typical user will do in the app. We specifically analyzed the speed of the app, the time it took to get over the learning curve in the app, and whether the app could be changed to make it more user friendly.

Usability testing was extremely important for the end-user facing aspects of the application. In particular, the users were able to use the app without any outside assistance from a team member. If they weren't able to navigate the app by themselves, we would have needed to change something. Also, this was one of the first times that someone outside of our team used the app, so it was a good way of getting constructive feedback from others. For example, we realized that the submit location button wasn't obvious to the user and that we needed to provide a glossary explaining what each of the scenarios meant.

Since our application was a Progressive Web Application (PWA), we got feedback for the web version, the iOS version, and the Android version of the app. The only difference we noticed was the fact that Android and iOS didn't have a way to export the results to PDF. Neither platform supports this feature, so we couldn't implement it for those devices. Overall, we spent most of our time testing the web version.

In order to get the best feedback, our team used zoom to visualize the user's screen and get their input on certain areas of our application. Most of the testing required us to take notes on the user's interactions with our app, but we also got some verbal feedback from the user about their experience. To accomplish this, we gave them a very vague task to complete within the app. Then, we watched as they progressed through the app. We documented specific information on how long it took for the user to get to the destination and if there were any spots where they were confused on what to do next. When possible, we got several people from the same user type to test out the application at the same time. By doing this, cultivated useful discussion and questions while they were going through the application. While on zoom we were able to tailor the testing to the user's specific background.

Since we had three very different user types, it was also important to see if we addressed each of their needs. To begin, we are assumed that the general users hadn't heard of the Madingley Model prior to our testing meeting. As a result, we made sure they didn't get lost while navigating the app. Both the scientist and policy maker groups had a detailed scientific background. It was important for us to provide them a version of the app with this in mind. It was important that we tested each of the user versions because they all produced different variations of the data. For example, the general user had the option to select 4 different output scenarios, but the scientists had the option to select from 19 different variables. Therefore, if we only tested the app from the general user's perspective, we would have missed the other ⅔ of our application. It was important to test every part of the app that any user could interact with. In the case of our application, the user interacted with the: geolocation module, visualization module, and the user interface.

Geolocation Module

In this module, there were three main pages we will need to test. First, we needed to test the select location map page. On this page, we made sure that the user could intuitively navigate the map to select their desired location. Also, we needed to test whether the user knew that they could change the radius of the circle. Second, we checked to see if the user had any problems with the manually input coordinates page. On this page, we investigated if the user had any problems entering the latitude, longitude, and radius. The third geolocation check we needed to test for was when the user selects to use their current location. The program needed to correctly retrieve the device's location and then properly display it on the map page. In the process, it alerted the user if the location services were turned off in their browser or computer.

Visualization Module

For this module, there was only one page to test. However, there was a lot of important information that could have been wrong. First, we needed to make sure that the user could correctly export a pdf of the results. There was an export PDF button within the page. We found out that this wasn't possible on Android and iOS. We also needed to verify the user could locate the generate PDF button and that they knew to click the button. Second, we checked to see if the data was displayed in a way that makes sense for the user. If there was too much information on one page, we might have confused the user. If there was too little information on the page, the user might not use our app in the first place. Lastly, the team also needed to test if the table and map were readable. In other words, we wanted to find out if the user could interpret the data in these areas.

User Interface

One of the most important UI tests for our team was checking if the user could correctly navigate from one page to the next. For example, we noticed that all of the users were able to submit their location (on the map page) and move onto the scenario option page without any problems. Even though we didn't have all the languages implemented yet, it was also beneficial to test out the translation feature. We were able to test the French and Spanish Translations. It showed us English text that we forgot to translate or text that was incorrectly translated.

Back-End: Lambda Functions, API Gateway, and S3 Bucket

Given that the end-user did not natively interact with the backend, this module was excluded from usability testing. We specifically designed our app to hide these aspects from the user. If they had access to these parts, they could change the data or even destroy the inner workings of our application.

Examples of Tests for All Users		
Tasks	Acceptance Testing	Testing Results (up to March 19)
The user should be able to create a PDF of the results	Success: ● If the user can correctly navigate the app and get a PDF using the button on the results page Needs Work: ● If the user can't get to the results without assistance ● If the user uses the browser to create a PDF instead of the provided button	**Success** 5/5 groups were able to create a PDF. However, ⅗ groups had at least one person who took longer to find the button than we would like.
The user should be able to start a new simulation and get the results for ____ (insert a mixture of countries all over the world)	Success: ● If the user can get results for the specified country. ● If the user can start a new simulation and it displays new information from before Needs Work: ● If the user can't start a new simulation without assistance ● If the user gets results for the wrong country	**Success** 5/5 groups were able to move the circle to the specified country and get results for that specific region.
The user should be able to use these latitude and longitude coordinates to find the corresponding results	*Send latitude and longitude coordinates to the user in the zoom chat* Success: ● If the user can get results for their specified location ● If the results page shows a circle with the specified radius Needs Work: ● If the user doesn't know where to input the values without assistance ● If the user gets an error or the results don't populate correctly	**Success** All the tested groups* were able to get the results matching the latitude and longitude coordinates that were entered. *only tested on ⅗ groups.
The user should be able to walk through the app in the French language	Success: ● If the user understands what all the text means in the French language. Needs Work: ● If the user has trouble navigating the app because they don't understand the text without assistance ● If the user uses Google Translate outside of our app	**Success** Camille from scientist group 1 was able to navigate the app.

The user should be able to explain what the Madingley Model is and explain what each of the scenario options mean	Success: ● If the user got a good understanding of the app from the About page ● If the user correctly summarizes the main points of the Madingley Model and the scenario variables Needs Work: ● If the user doesn't understand what the Madingley Model is without Googling it. ● If the user is just clicking buttons and doesn't understand what the scenarios mean	**Needs work** ⅖ groups (which had no previous experience with the Madingley Model) had a hard time understanding that the results were based on the Madingley Model. The getting started page should be edited to include some Madingley information

Table 8: User Study for All User Types

Examples of Tests Specifically for General Users		
Task	Acceptance Testing	Testing Results (up to March 19)
A general user should be able to easily select a new output variable (out of the 4 possible options)	Success: ● If the general user can select a new variable and the map refreshes to display this new information Needs Work: ● If the general user doesn't realize that they can select a new output variable without assistance	**Success** 5/5 groups were able to change the variable from the current default of "allelic diversity" to any of the other output variables.
A general user should be able to describe what each of the output variables mean	Success: ● If the general user can define what each of the output variables mean Needs Work: ● If the general user has no idea what they are clicking.	**Needs Work** 3/3 of the general user groups had a hard time understanding what the variables meant. Suggestion: add descriptions next to the variable name

Table 9: User Study for General Users

Examples of Tests Specifically for Scientists		
Task	Acceptance Testing	Testing Results (up to March 19)
A scientist should be able to easily interpret the results shown on the results page	Success: ● If the scientist can explain what the results are showing Needs Work: ● If the scientist doesn't know what is being displayed on the page. ● If the scientist is overwhelmed with all the information being displayed	**Needs work** Scientists weren't able to get the results to display. The users were stuck on the "Madingley Data loading…" notification.
A scientist should be able to select between the 20 raw data output variables	Success: ● If the scientist can switch from 1 output variable to another one and the results change as a result Needs Work: ● If the scientist doesn't know that they can select one of the other 19 output variables ● If the scientist gets the same map even after they individually select several of the other variables	**Needs work** Scientists weren't able to get the results to display. The users were stuck on the "Madingley Data loading…" notification.

Table 10: User Study for Scientists

Examples of Tests Specifically for Policymaker		
Task	Acceptance Testing	Testing Results (up to March 22)
The policymaker should know that the variables on the results page are calculated EBV values. Test by asking the policymaker what each of the variables represent	Success: ● If the policymaker can identify that the variables are calculated based on their selection in the app Needs Work: ● If the policymaker thinks that the variables are hardcoded values	**Needs work** The policymaker wasn't able to get the results to display. The user was stuck on the "Madingley Data loading…" notification.

The policymaker should be able to get results in a reasonable amount of time or get a warning that it might take a long period of time to retrieve the requested data	Success: ● If the policymaker gets results in less than 2 minutes. ● If the policymaker gets an alert that it might take a long time to process their results Needs Work: ● If the policymaker thinks the app is broken because the app is on the "Loading Madingley Data" page for too long	**Needs work** The policymaker wasn't able to get the results to display. The user was stuck on the "Madingley Data loading…" notification.

Table 11: User Study for Policymaker

Resulting Project Changes

As a result of testing, we came up with nine different changes that we implemented before the final version was completed. You can see all of these changes in Figure 5. Two of the changes that we want to highlight are parallel parsing and adding a layered approach to the backend. We decided parallel parsing was important because if the user selected a large radius, they would have to wait a really long time in order to get their results. By adding the parallel parsing, we were able to make all requests between 20 and 30 seconds (regardless of how big the selected radius was). To make this possible, we also implemented a layered approach. During testing, we noticed that the API Gateway would timeout and wouldn't return any values to the user. By retrieving small sections of the data at a time, it allowed us to get around this issue and speed up the data retrieval process.

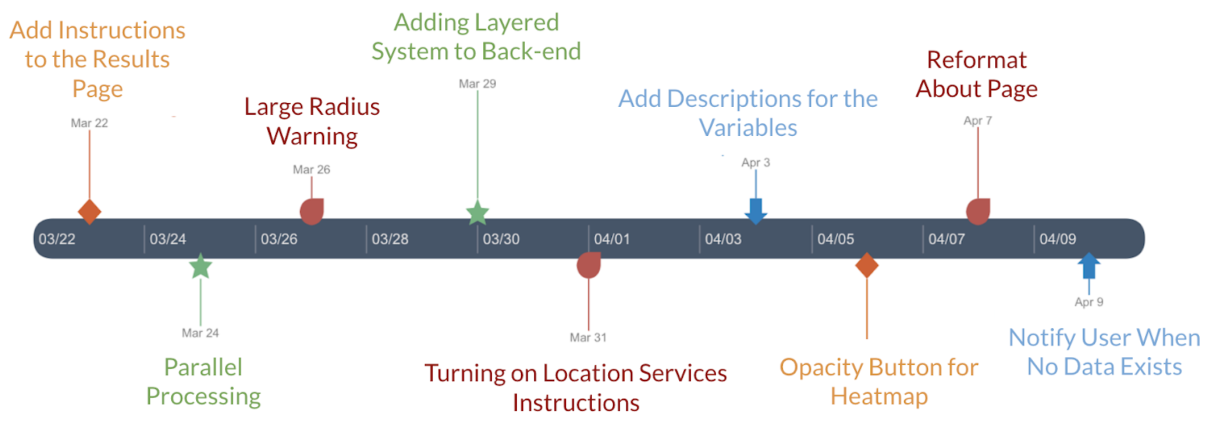


Figure 5: Post Alpha Prototype Changes

Project Timeline

This section outlines the key tasks and timeline on which we completed our project.

As you can see in Figure 6, we spent the first five months planning and researching solutions for our product. In September 2020, the team completed a Team Standards document. We felt it was important for us to set a good foundation for the team as we progressed through the project. After that, one of the main things we accomplished in the first half of our project was refining our Minimum Viable Product (MVP). In addition to that task, we also researched possible software, solidified our product requirements with the client, and created a technical demonstration. This technical demonstration showed what software we were hoping to implement in the Spring 2021 semester.

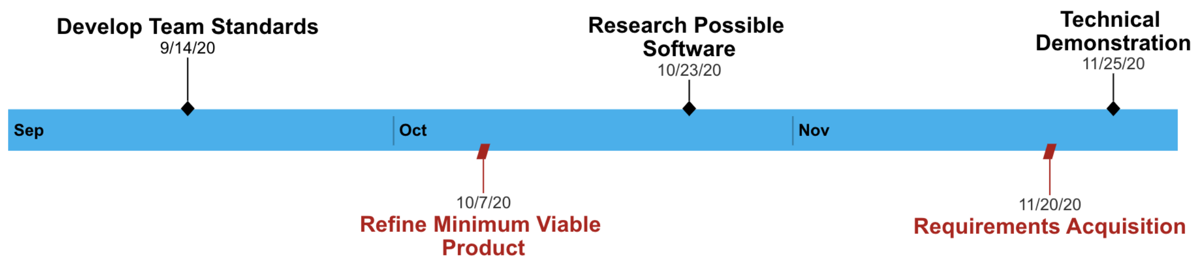


Figure 6: Fall 2020 Semester Milestones

Going into part two of the project, we were able to move onto the implementation stage (as seen in Figure 7). During the Spring 2021 semester, we wrote all the code needed to satisfy the requirements. Throughout the implementation process, the team simultaneously worked on the frontend and the backend components. As you can see in Figure 8, the frontend of the application was coded by Wes, Greg, and McKenna. The backend was completed by Kainoa, Greg, and McKenna. In terms of the frontend, some of the things we were able to accomplish were having a working map selection component, successfully implementing the ability to export the results, and creating a results page that contained visuals that were easy for the user to understand. In addition to that, we were also able to get the AWS Lambda, AWS S3 Bucket, and API Gateway working on the backend. In addition to those important milestones, we also completed a software testing plan. This document outlined our testing strategy. On April 22nd, we had a final demonstration with our clients. During this presentation, we showed off our application and demonstrated how each of the requirements were implemented. As a result, the clients conveyed that they were very pleased with the product and that we did a great job of implementing all of the requirements. They are hoping to build off what we have created to make it an even better and more informative application in the future.

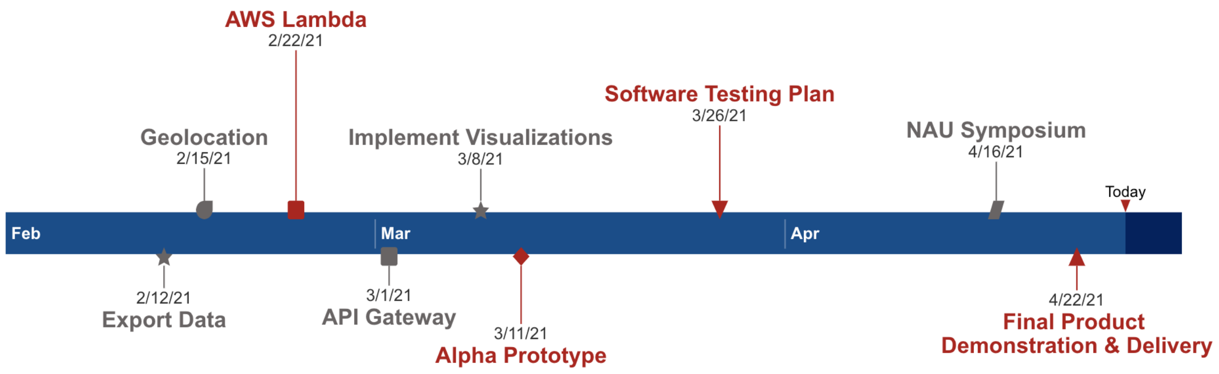


Figure 7: Spring 2021 Semester Milestones

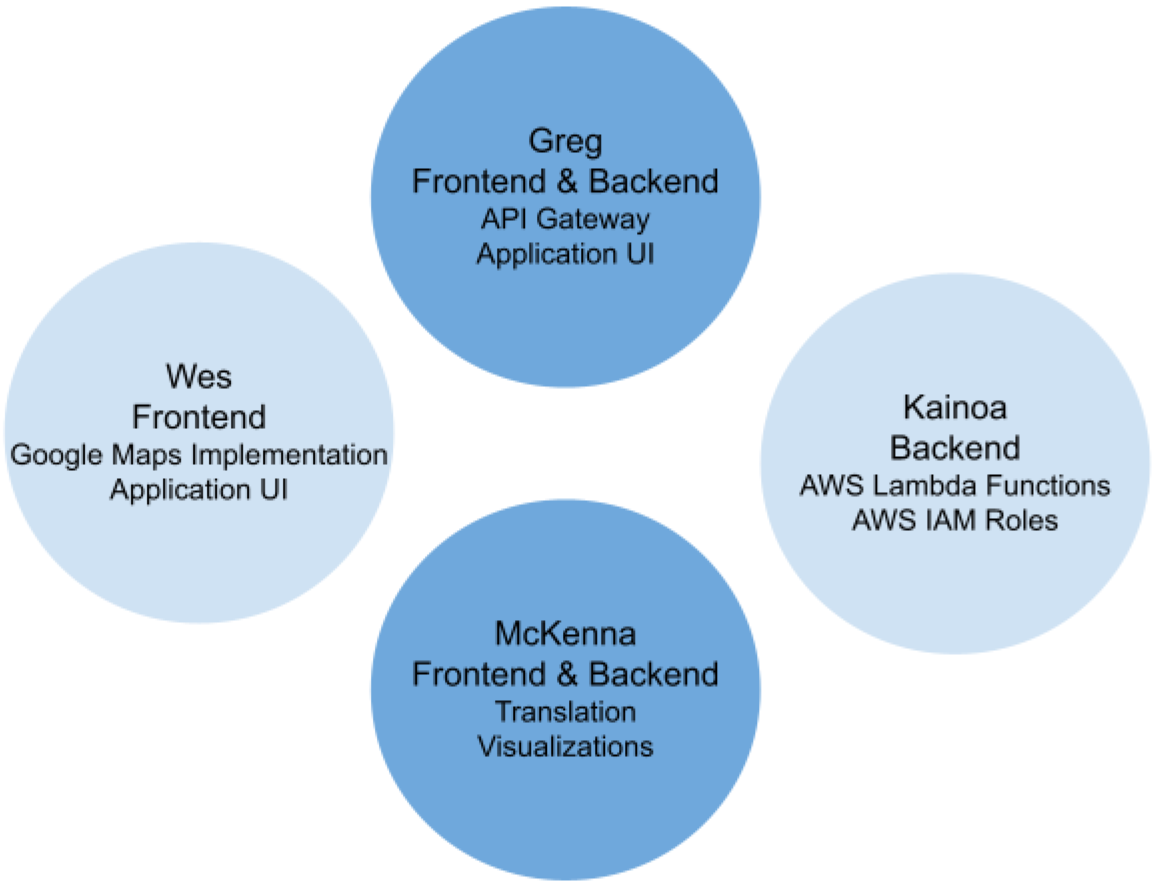


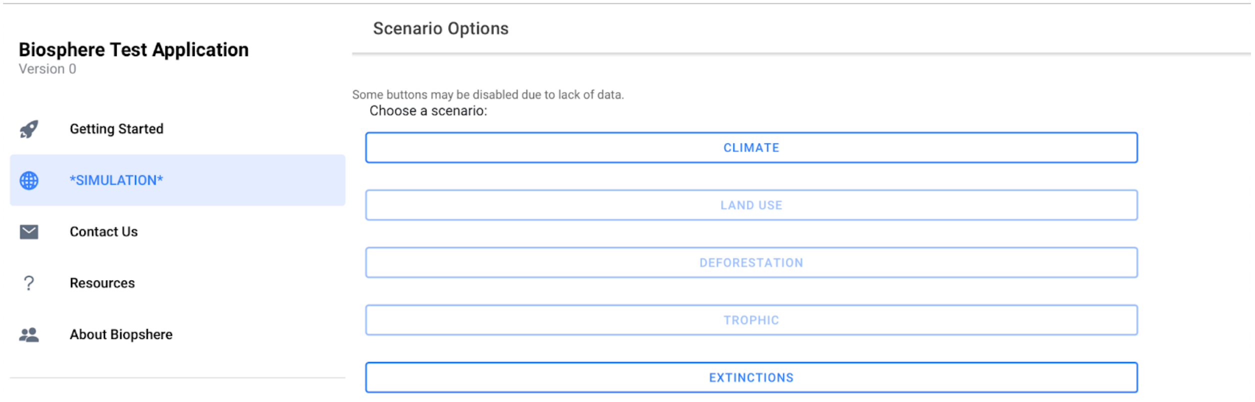
Figure 8: Each Team Members' Task Focus

Future Work

Even though we addressed all of the requirements in our final product, there are two main areas for future improvement: implementing new scenarios and integrating support for a wider range of languages. First, as new information becomes available to our clients, they plan on adding new scenarios to the application. They are also hoping to find people who can help translate the application for languages that aren't currently supported.

Implementing New Scenarios

As you can see in Figure 9, not all of the scenario buttons included in our application have the data to support their use. This is why some of them are unclickable. Therefore, as the science becomes available, our clients hope to add these new scenarios. Some of the scenarios, such as logging, might take a couple of years to complete. Other scenarios, such as land use might just require some time to translate the data from the complicated Madingley Model data into something that our application can interpret.



Some buttons may be disabled due to lack of data.

Choose a scenario:

Figure 9: Current Scenario Options Page

Integrating Support for New Languages

Currently our application supports four different languages. These languages include English, Spanish, French, and German. We originally prioritized these languages since the Madingley data is focused on areas of the tropics and these are some of the common languages in those areas. For example, our client is hoping to get policy maker's attention in Gabon and Peru. In addition to those, the client is hoping Brazil will also find interest in the information that the app is producing. Therefore, our client's next language priority is Portuguese. In order to get the best possible translations, our team and the client both agreed to get native speakers to translate for us rather than getting a mediocre translation from Google Translate.

Conclusion

Tropical forests are biodiversity hotspots filled with countless species of flora, fauna, and fungi. Thousands of these species in recent years have become extinct, and around one million are at risk of extinction due to an influx of human activity in the wild. The root of this current mass extinction lies in issues that have been unresolved for years. This includes, are but not limited to: climate change, illegal deforestation, logging, and poaching. The rapid decline of biodiversity has extremely negative effects on those who inhabit the Earth. Biodiversity supports ecosystem service include: air and water filtration, renewable energy, and climate regulation.

In order to prevent or slow the loss of biodiversity, our client Chris Doughty is using the Madingley model; a 'next-generation' ecosystem and biodiversity model. The purpose of this model is to inform policymakers about the impacts of their choices on biodiversity, ecosystem services, and on trajectories of biodiversity change under different scenarios of human development.

The problem we have been tasked to solve is that the Madingley model requires scientific expertise to operate, and uses a large amount of computational power, which is not readily available.

In order to solve this problem, we developed a progressive web application that allowed the user to generate unique queries regarding pre-run Madingley model scenarios. This data is then processed using services through AWS. Finally, the data is visualized for the user to see, interrupt, and save. Some of the key features include:

- Worldwide availability
- Universal device and browser compatibility
- Multi language support
- Customized visualizations
- Interactive and familiar location selection tools
- Extremely responsive application design

Not only did this satisfy the requirements set before us, and then some, we are also able to save our clients over 1,000 hours of development and anywhere from $26,000 to $52,000 in development cost.

Our client and their supporters have been extremely happy with the progress and final status of this project. A segment of the target audience includes policy makers associated with national and international level organizations or governments. This means that this application could be used to influence environmental policy around the world which will have a global scale impact.

This process has taught us about the importance of patient, collaboration, and most importantly communication. Our team has experienced and overcome numerous frustrations and complications and as such we have not only bonded but take this with us as real-world experience. We understand the impact that open and honest communication can have on a project. We know that everyone is human, and we all make mistakes, and that what’s important is to pick each other up and keep going.

‘A‘ohe hana nui ke alu ‘ia.
No task is too big when done together by all.

Glossary

Amazon Web Services (AWS): is a subsidiary of Amazon providing on-demand cloud computing platforms and APIs to individuals, companies, and governments, on a metered pay-as-you-go basis.

Progressive Web Application (PWA): is a type of application software delivered through the web, built using common web technologies including HTML, CSS and JavaScript. It is intended to work on any platform that uses a standards-compliant browser, including both desktop and mobile devices.

Appendix

Appendix A: Development Environment and Toolchain

In this section, we will explain some of the basics of our application. First, you can find information on how to configure your machine to support our application. Second, we will cover the steps you need to take to get the downloaded code from GitHub to run properly on your device. Overall, this appendix will serve as a how-to manual for our application. After reading this part of the appendix, you should be able to go from knowing nothing about our application to being able to upload a new version to GitHub.

- **Hardware**:
  As a team, we used a wide variety of platforms to run and build our application. In Table 12, you can see that we all used an Intel processor. However, we had different amounts of memory storage on our machines.

Kainoa	Ubuntu / Intel processor / 64GB RAM *This was completely overkill, the backend could be locally tested/developed and run on a Raspberry Pi if needed.*
Greg	Windows 10 / Intel processor/ 16GM RAM *The program was able to easily be run, stopped, and restarted throughout many tests without any problems.*
McKenna	Windows 10 / Intel processor / 16 GB RAM *The program was able to easily be run, stopped, and restarted throughout many tests without any problems.*
Wes	macOS / Intel processor / 8 GB RAM *I didn't have any problems running the Google Maps components and the rest of the application on my computer.*

Table 12: Team Member Machine Technical Specifications

- **Toolchain**:
  This subsection outlines all the tools we used to get our application working. Some of the tools listed in Table 13 are essential for our application to run properly. These tools include ngx-translate, the Geolocation module, and the Google Maps API/Visualization Library. The team also used a wide variety of IDEs. Any of the IDEs we used can be substituted with your personal preference.

Kainoa	**Atom w/ custom linter** (used for code standardization) No package manager, all software developed used native python3.8 modules
Greg	**Microsoft Visual Studio Code** ● All-around IDE for managing and editing source code in Angular (TypeScript), the terminal also allowed for the installation of libraries and command line for Git **Android Studio** ● Android IDE that allowed for deployment of the application to a native Android environment using an emulated device **XCode** ● iOS IDE that allowed for local building and testing of applications on iOS devices.
McKenna	**Microsoft Visual Studio Code** ● All-around IDE for managing and editing source code in Angular (TypeScript), the terminal also allowed for the installation of libraries and command line for Git **Android Studio** ● Android IDE that allowed for deployment of the application to a native Android environment using an emulated device **Ionic Capacitor** ● Compiled source code for web clients and translated the TypeScript code into native Android languages for testing **ngx-translate** ● Translation library that allowed for the automatic detection of languages on a user's device, and allowed for easy manual translations to be implemented **Chart.JS** ● Visualization library used for creating data graphics **Google Maps API + Visualization Library** ● Used to plot heatmap points on map with weights
Wes	**JetBrains WebStorm** ● I used this IDE to edit and add to the codebase. **Geolocation Module** ● You need to import this module in order to get the user's current location **Router and Navigation Extras Modules** ● You need to import these two modules in order for the user input to be transferred from one page of the application to another

Table 13: Essential or Nice-to-Have Tools Used for Our Application

- **Setup**:
  Tips on how to set up your device can be found in this subsection. We will outline the things you need to download in order to get the application to run properly. These include both backend and frontend components.

Backend	$ sudo apt-get update # update system to most recent packages
	$ sudo apt-get install atom # installs atom
	$ sudo apt-get install pylint # installs pylint

Frontend	1. Install NPM (Node Package Manager) 2. Use the terminal to 'cd' into application source code folder 3. Run `npm i` in that folder (this installs all necessary node modules) 4. Run `ionic serve` OR `ionic serve --lab` (this builds and host the application on your local machine) Note: instructions on local development setup are also included in further detail in the README.md file of the application source code.

Table 14: Steps for Setting up Your Environment

- **Production Cycle**:
  Now that you have set up your machine, you can now run and edit the application. In Table 15, there are instructions on how to edit both the backend and frontend components of our application.

Backend	**Local Development** 1. Change Input.json for application inputs 2. Run APIGatewaySimulation.py for testing 3. Main file is lambda_function.lambda_handler 4. All other files are libraries and helper functions which are described in their file docstrings.
Frontend	1. Edit desired file 2. Save changes 3. In terminal run *ionic serve* (optional) 4. Push code to GitHub

Table 15: Production Cycle Steps