Tech Feasibility

12/6/2024

Forest Frames

Sponsored by Dr. Camille Gaillard, Dr. Chris Doughty, Dr. Duan Biggs, and Dr. Jenna Keany

Mentor: Scott Larocca

Members: Dalton Tippings, Daniel Austin, and Nicholas Greco

Version 1.1

Table of Contents

Table of Contents....2
1. Introduction....3
2. Technological Challenges....3
3. Technology Analysis....4
3.1 Ensuring App Performance and Low Memory Requirements....4
3.2 Ensure Compatibility with Older Android Versions....5
3.3 Collecting Audio and Visual Data from the App....6
3.4 App Functions Seamlessly while Offline....7
3.5 Verify Accuracy of User Submitted Data....8
3.6 Providing Secure and Accessible Storage of User Data....10
4. Technology Integration....11
5. Conclusion....12

1. Introduction

As humans encroach on natural habitats and resources, the ability to gather data on wildlife and support conservation efforts is becoming increasingly critical. Collecting and analyzing this data, however, is an often challenging and time consuming endeavor that can require intensive travel and research. Because of this, large areas are left unexplored and unknown to the world which limits the impact of conservation efforts. This is part of the reason citizen science initiatives are becoming so popular, as it uses the general population to collect information about their local area.

While there are currently many efforts to engage the public in conservation projects, these projects often do not have the reach necessary to collect comprehensive data. This is especially problematic in areas where people do not have the extra time or money to be spending on conservation projects. The reality is that unless there is an incentive for the user, these projects will not get large-scale engagement. Another issue many of these initiatives face is a lack of verification on user-data. Many apps allow for users to submit audio or visual recordings to their collection, however it often requires manual review of the user-data and can come at a much later date than the original recording time. Waiting for a user's data to be verified can often lead to a disjointed experience, especially if their data is deemed inaccurate which may make the user feel like they have wasted their time. Similarly, our app needs to verify where the data came from, using Nasa's GEDI to verify where exactly the data was collected from. If the app is to provide incentives, it needs to be able to verify that all parts of the data are legitimate.

Our solution is a mobile app designed to unify wildlife tracking databases by allowing users to collect information from their local wilderness and potentially be provided incentives for their time. Through the app, users can submit audio and visual data of fauna they come across, which will then be sent to a server to be verified against existing wildlife databases. This app aims to use the power of crowdsourced citizen science and a more streamlined method of data verification to hopefully engage more people in conservation efforts.

While our project is still in early stages of development, we have carefully considered the technical challenges and limitations our project may encounter. Below we will outline these problems our project will face and analyze them more thoroughly with potential solutions.

2. Technological Challenges

These are the technological challenges we will be facing during this project's development:

- Ensuring App Performance and Low Memory Requirements
    - This app will often be used by low-end phones, so this app must be able to perform well on these kinds of devices
- Ensure Compatibility with Older Android Versions
    - This app will need to be used by those with older Android phones, so the app must be compatible with older versions of devices.
- Collecting Audio and Visual Data from the App
    - The app needs to be able to collect the required data from the user which will then need to be uploaded to our server.
- App Functions Seamlessly while Offline
    - Users may often be collecting data in areas without a network connection, so there must be a way for the app to still have some functionality while offline.
- Verify Accuracy of User Submitted Data
    - The service must be able to ensure that data supplied by the user is accurate in order for it to be used for official and reliable sources.
- Providing Secure and Accessible Storage of User Data
    - Our app must be able to gather data and transport it efficiently and make the data storage and transport secure to avoid tampering.

3. Technology Analysis

3.1 Ensuring App Performance and Low Memory Requirements

3.1.1 Introduction

This app is meant to be able to be used on many kinds of devices, such as low-end devices that are unable to handle significant strain. This is important as many of the devices we are hoping to target are unlikely to be able to handle lots of stress so the app, thus the app's performance and memory requirements are a critical aspect of the desired app functionality.

3.1.2 Desired Characteristics

The technologies chosen should be both **lightweight** and **efficient** to use. They should take up as little memory as possible to download and then not need to use significant amounts of processing power to run the functionalities needed for the app's purpose. The app should be able to run without lagging or lots of loading times, and it should ensure that the actual size of the app on the phone does not exceed what users are willing to use.

3.1.3 Alternatives

There are various methods to ensure the app runs efficiently. One is using Android NDK to use C/C++ code for certain areas of optimization. There is a service called Google Play Services that allows for various optimizations for both the app itself and data transfer. Along with that, ensuring most of the functionality that would cause strain is handled on the server itself.
Various languages are used in Android development such as Kotlin and Java, along with other languages for specific tasks.

3.1.4 Analysis

Here we will focus on the pros and cons of Java and Kotlin.

Java: Java is the native language of android and the primary language used by most apps and developers of the system. Thus it has a wide array of tools and many libraries built to support it. Along with that the Java virtual machine(JVM) ensures that things are **efficient** and **lightweight** as possible.

Kotlin: Kotlin is an alternative to Java that still has many of the tools that Java has. Kotlin is also able to compile using the JVM so performance will not differ a significant amount between Kotlin and Java. Along with this, Kotlin has many additional features that ensure ease of use and stability. Kotlin is also the language that the group is most familiar with in general so it will take less time to begin developing in it.

3.1.5 Chosen Approach

For these reasons, we decided to use Kotlin as we have more experience with it and it offers benefits compared to Java without any significant sacrifices to performance. Other technologies such as Android NDK, Google Play Services, etc. will be implemented as needed for app development as there are no notable alternative tools for Android Development.

3.1.6 Proving feasibility

When implementing these technologies, we will have to develop ways to test for size and speed on our app. We will implement profiling and various test versions of our app to find out and document these technologies and find the smallest and most efficient way we can make our app.

3.2 Ensure Compatibility with Older Android Versions

3.2.1 Introduction

An important aspect of this app is to allow people from diverse backgrounds and areas to use this app. One way to achieve this is to ensure that the app is highly backward compatible with older versions of the Android Operating system. This is important because many of those we are targeting for our app may not have access to the latest versions of Android and all its functionality. Thus we face the challenge of ensuring compatibility with as many old Android phone operating systems as is possible.

3.2.2 Desired Characteristics

Particularly we want to ensure that we can **reach as many people as possible while maintaining as much functionality and performance as possible**. The reason that this is important is that the kind of data that we seek to gather is from communities that would normally not have significant access to high-end technology, so to ensure that those we wish to work with are able to use the app we provide.

3.2.3 Alternatives

While the solution could be to provide versions for all devices and all versions, that is outside the scope of possibilities. Of the devices that we could choose to develop for: IOS, Huawei, etc…, we decided to develop for Android as that is the most commonly used phone OS in our target markets. This meant we needed to hone in on the version we would develop for particularly Android phones. To decide on the version we needed to look at both what functionality a particular version has and how many users the version provides for. So we analyzed every version to see what each one can provide.

3.2.4 Analysis

Of all the versions available we decided to focus on these three versions: Android 14, Android 12, and Android 5, with each providing various benefits. One aspect to keep in mind is that Android apps have a feature where they have a target version and a minimum version where an app can work with the minimum, but is meant for the target. Our target version would be version 14 which is required to release new apps. So the determination needs to be made for the minimum version, whether it will be 14, 12, or 5. Version 14 will be the most simple and easy to develop for, along with having all the latest features, but the problem is that very few old devices support this version. Android 12 has many of the features of Android 14 and is far more compatible with older versions, along with being the oldest version still supported by Google. Android version 5 is the oldest version that is realistically possible to develop for as it has many basic features but lacks many new features that could be useful for our app, conversely, Android 5 is compatible with nearly every single Android device currently in use and can work for our entire target audience.

3.2.5 Chosen Approach

The decision we came to is to focus on developing compatibility with Android 12, as it won't be a significant strain on development but will still be compatible with many devices. This provides many

benefits, one being it provides significant compatibility with older versions while still being relatively up-to-date. We will begin the development of the app and further research all the features the version contains, what it lacks, and how it can be used.

3.2.6 Proving Feasibility

When developing the app with backward compatibility in mind, we will have to test each feature to make sure that it works in the most current version and the oldest version being developed for. The process to test this will be to follow basic testing for all features for each version being worked on.

3.3 Collecting Audio and Visual Data from the App

3.3.1 Introduction

An important aspect of this project is allowing for the user to collect ecological data and send them to our end for verification. Thus, we must consider both how this data will be collected on the user's end and how it will be transferred to our servers. The user will be collecting/sending different types of data ranging from simple (numerical values, description strings) to more complex formats (images, audio, video). As for the actual transfer of this data, our main concern is ensuring its reliability.

3.3.2 Desired Characteristics

As mentioned before, we will be dealing with numerous different types of user-collected data, so we will need to account for each one when designing this part of the app. It would be helpful to use a modular approach as the types of data being collected will be different for each area a user is gathering data in. As for the networking, we will just need a Kotlin library that can handle basic data transfer between a mobile device and a server. For more complex data types being collected, similar to the networking where we only need a higher-level view, we could use another Kotlin library to handle that collection and storage.

3.3.3 Alternatives

Android provides its own multimedia API which allows for a higher-level use of lower-level multimedia operations, making it more convenient to collect audio and video data. The alternative would be working directly with that lower-level functionality which that API uses. As for networking capabilities, there are multiple libraries that work directly with Kotlin, such as Ktor, Retrofit, and Fuel, which all achieve the basic networking functions we require. The collection of simple text-based input data is directly tied to the language used for programming the Android app itself, thus has no alternatives.

3.3.4 Analysis

Android's multimedia API is extensive in both functionality and documentation which would be very helpful for the development of this app's section. More importantly it provides access to recording/saving images (Image Reader), audio and video (Media Recorder). The only advantage dealing directly with the lower-level functionality would give is that we would have more control and optimization. Ktor's main advantage over the other two libraries is its compatibility with KMM (Kotlin Multiplatform Mobile), which would allow for our app to work on more than just Android. Ktor seems to be the upcoming standard for Kotlin networking, while Retrofit has been the standard. Retrofit is the most used of the options and is also type-safe, meaning it validates data type assignments at compile-time. Fuel takes a more simplified approach, which would also come with less control over the networking

capabilities, but could save development time depending on the depth needed in transferring the data across the network.

3.3.5 Chosen Approach

We will be using Android's multimedia API over its basic lower-level functionality as it will save on development time through its ease-of-use. The types of data it deals with spans across all of the three needed for the user to collect their required data. Ktor will be our preferred network library of choice due to its multi platform capabilities and balance between control and ease-of-use. We will have Retrofit as our backup if we are not able to perform the required functionality with Ktor.

3.3.6 Proving Feasibility

To ensure the reliability of the data collection, we will have to implement the following tests while building/refining this feature: test if the user experience is being hindered due to the performance of the data collection and validate that the data is being correctly stored within the device. For the networking side, we would first test a basic data transfer from an Android device to a server with a data size similar to the average amount we would be sending in our finished product. We could then expand to testing sending test data through different network points outside of our local area to further test the effectiveness of our networking code.

3.4 App Functions Seamlessly while Offline

3.4.1 Introduction

Since the purpose of this app is to acquire data in rarely-documented areas, and thus the target users being those local to those types of areas, we must consider the environment in which they will be using the app. Even though they will need to be connected to the internet in order to transfer the data from the app, there is no guarantee that they will have a stable, or any, internet connection while they are collecting the data. So, we must ensure the app can function reliably while offline.

3.4.2 Desired Characteristics

The main way to guarantee offline app functionality is to store the data, which would usually come from over the network, onto the user's device. However, along with storing and retrieving these files locally, we must also consider the storage/processing limitations of the device, in accordance with that previous technological challenge. The technology must allow us to decide what exactly gets stored on the device, along with when it should be removed to maximize space and efficiency.

3.4.3 Alternatives

The Kotlin language provides operations which allow for basic I/O (input/output), which would be used to store required information as files on the device. It also provides functionality for basic caching, which stores data which gets requested repeatedly. There are also many libraries that work on top of Kotlin which improves those capabilities. For I/O, there are Okio and Kotlinx, while for caching, there are Kache and Ktor's caching.

3.4.4 Analysis

The alternatives to Kotlin's basic I/O and caching functionality all provide the main benefit of being multiplatform, which could help for expanding to different devices in the future. Okio provides

more ease-of-use on the developer-side by abstracting Kotlin's lower-level operations for I/O and is well known in the Kotlin I/O sphere, while Kotlinx is much newer and builds off of Okio. Kache and Ktor's caching abilities provide relatively the same benefits (persistent caching, in-memory caching, multiplatform, coroutine-friendly). Both Kache and Okio have similar issues of not being likely to function well with our Ktor networking capabilities.

3.4.5 Chosen Approach

We will be using the Kotlinx library for our app's I/O functionality and Ktor's caching functionality. Both were chosen mainly because of compatibility, as mentioned prior with the networking. Since all of the options had very similar benefits to one another, its compatibility was the only major factor. If these libraries do not end up providing the functionality we need, we could always fallback to Kotlin's basic functionality of both I/O and caching.

3.4.6 Proving Feasibility

To ensure that the app can function offline effectively, we will have to implement various tests for both the I/O and caching libraries. Tests for the I/O would include: testing that the correct data is stored on the device and the data stored is able to retrieve effectively and more efficiently than regular Kotlin use. Tests for caching would include: analyzing network usage from the app with caching versus having caching disabled. For testing overall, we would have to implement assertions for the main test, which goes over a full run of collecting data, which ensures 0% network usage during that test.

3.5 Verify Accuracy of User Submitted Data

3.5.1 Introduction

As previously mentioned, verification of the data users submit will be necessary in order to compensate them both efficiently and fairly, meaning that once the data is sent from the user's device to our server, we need to ensure the data can be validated correctly and in a reasonable amount of time. For each verification tool we use on some given user data the result will be added to a cumulative total value (validity score), which will be used to determine whether that user data is valid, or more likely not to be fabricated. That conclusion would then be used in both relaying that message to the user and whether the data would be forwarded to the GLOBE database or iNaturalist.

3.5.2 Desired Characteristics

Since different types of data will be recorded, we will need multiple types of verification tools and a way to help decide when to use which one. In order to prevent "reinventing the wheel", we will be using open-source verification tools along with publicly available data to aid in the verification process. As for what will be deciding the types of verifications, the language we use will have to be capable of recognizing data types, brief metadata analysis, and simple decision-making.

3.5.3 Alternatives

On the validation side, we have MegaDetector for video/image recognition, which most ecological recognition software (EcoAssist, CamTrap Detector, etc.) builds upon, and OpenSoundscape for audio recognition. These would then be paired with public datasets, such as ESC-50 (Environmental Sound Classification Dataset), to complete the verification process. For processing and determining what to do with the data, we have the programming languages Python and C++.

3.5.4 Analysis

MegaDetector and OpenSoundscape are our only options for recognition on both the visual and audio side. Other recognition software/libraries are not open-source and high-level enough to use without training a machine learning model for ourselves. However, the program used for data management on the server is a more difficult decision. Python is known for its excellent data management features such as numpy and pandas, however it is generally slower than C++ for real-time and more intense calculations. However, aside from speed Python beats C++ in almost every other margin that matters. Python has a very high ease of use as its syntax is very minimal and forgiving. Python also has the upper hand on external libraries, outside of the previously mentioned numPy, Python has a wide variety of data management tools to assist our servers data processing. Importantly, Python also has strong support for database integration, which will be crucial for storing our users information.

3.5.5 Chosen Approach

We have decided to use Python for our server-side program, which will handle both the decision making (for which verification tool to use) and how those tools will be executed. The team is very familiar with the language, it is compatible with the aforementioned verification tools, and has extensive libraries for data management. Since MegaDetector and OpenSoundscape are only options for eco-recognition, we will be implementing them on the server-side.

3.5.6 Proving Feasibility

When developing our server processing code, we will need to develop unit tests to be sure our verification tools are verifying the correct information as well as a way to measure the speed data is being received at. Using test data with the verification will be very important as we will be setting thresholds for which verified data will be accepted and forwarded to the GLOBE database. We will also have to implement error handling and logic for it to ensure the server is still functional even if a verification is not fully carried out.

3.6 Providing Secure and Accessible Storage of User Data

3.6.1 Introduction

In order to store our users' login information, as well as their collected biodiversity data, our app will need a secure and fast way of storing said data.

3.6.2 Desired Characteristics

An important constraint of our app is that it must work offline, which means that until the device is connected to the internet once more, our app must be able to store collected data locally on the device. Another note about our app is its ability to connect to our processing server, the storage must be able to quickly and securely send and receive user data to perform important tasks like data verification and storage. When storing user data locally on their device, our app will be capable of storing it on the device in .csv files. This file format is generally accepted across most database services, and will be the simplest way to send their data to a database once the user is reconnected to a network.

3.6.3 Alternatives

The most common database service is Amazon Web Services (AWS), which has a fantastic API that is well documented and supported. Our team has had experience with it in the past which would make

it a great choice. Another common database service that we were familiar with was MongoDB. MongoDB provides similar services to AWS servers and has a well documented API.

3.6.4 Analysis

While both services provide easy to use databases, Amazon Web Services have specific advantages over MongoDB. Aside from the team's familiarity with it, AWS has a more flexible database server that supports MySQL, PostgreSQL, and NoSQL, while MongoDB is primarily a NoSQL service. Another thing to consider is the security of our database, since our users will be sending potentially sensitive information across our servers like login information. AWS is known for its strict compliance with security standards and has great options for logging and auditing. Finally, the last thing to consider in our database is integration with other services. While MongoDB has integration with other cloud services, it can’t really compete with Amazon’s extensive integration services.

3.6.5 Chosen Approach

To handle storage of our users information and data, we are going to use the database service Amazon Web Services. We chose this service because they have a well documented API, and are compatible with Android Studio. They also work with multiple languages like javascript, PHP, and importantly Kotlin when handling query requests to and from the database.

3.6.5 Proving Feasibility

We can run test queries on the database before transmitting any actual user data to be sure that the database meets all of our application requirements.

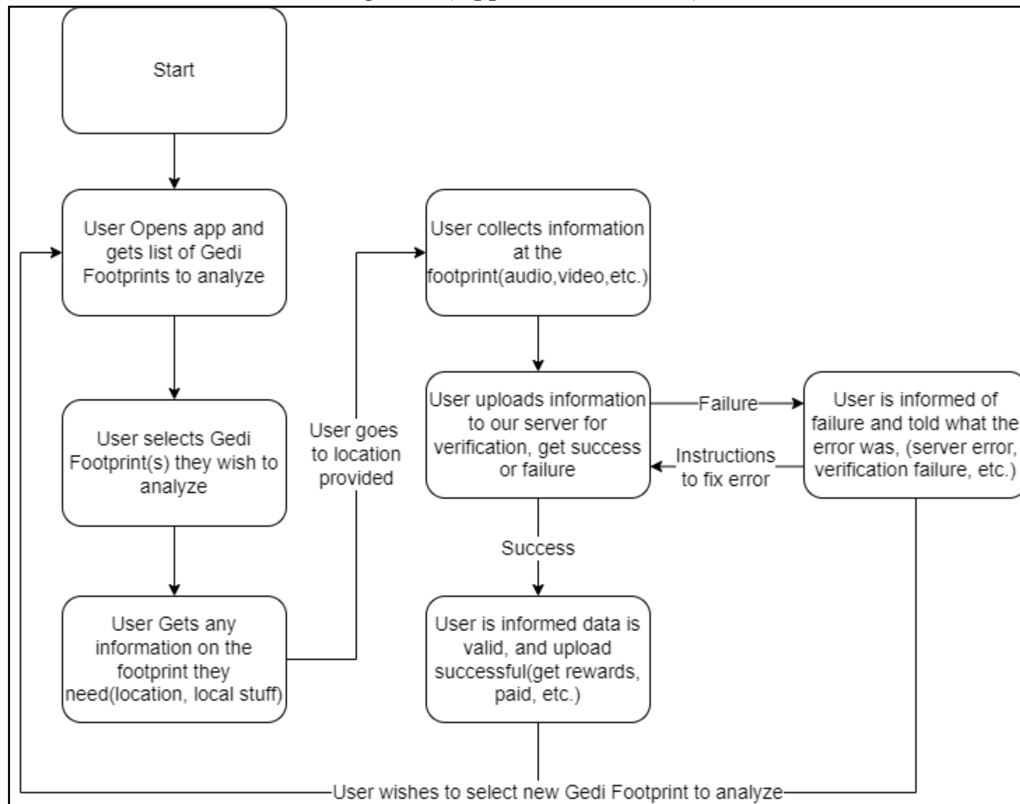
4. Technology Integration

It is important that all of the solutions to these challenges are seamlessly integrated with each other. We intend to do that by dividing our solution into three distinct parts. Each part will handle certain aspects of the project and ensure ease of use, these three parts consist of the app itself that will be downloaded, a server to handle verification, and a database to store any data we need.

Each section will handle a particular part of the challenges talked about above and integrate the different technologies. Starting off with the app itself, this will be as lean and efficient as possible using our decisions from the Ensuring App Performance and Low Memory Requirements, Ensure Compatibility with Older Android Versions challenges, App Functions Seamlessly while Offline, and Collecting Audio and Visual Data from the App. Creating a highly compatible and small app that will then be able to supply any gathered data to the server for more costly processes.

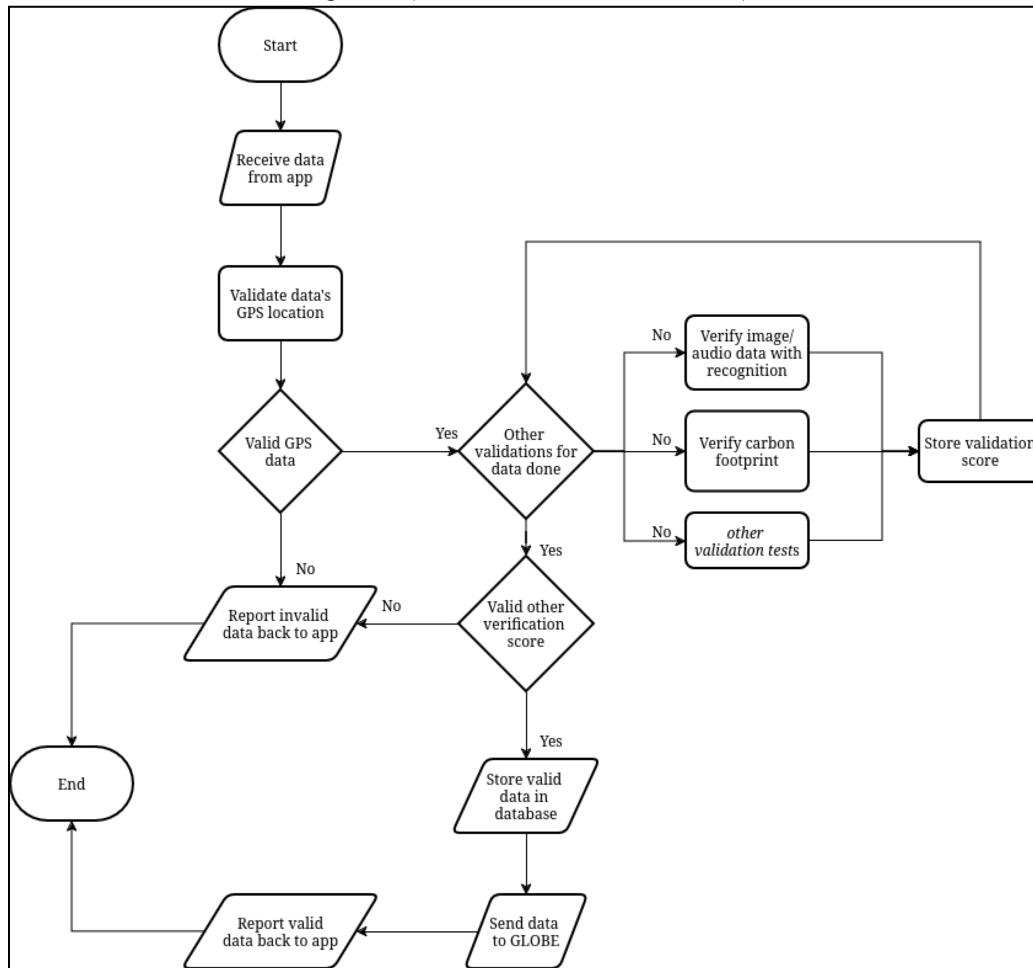
The user’s experience with the app should be one of ease, simply by choosing where they will be gathering data, collecting them using prompts from the app, and they receive whether their data was valid once their collected data is verified by the server. This process is displayed in Figure 1 (shown below):

Figure 1 (App User Flowchart)



The server will handle most of the verification, particularly the challenge Verify Accuracy of User Submitted Data. Since this requires significant processing time and access to many resources, our server using Python will conduct all the functions needed as described in Figure 2 (shown below):

Figure 2 (Data Verification Flowchart)



The database will be a collection of all the data we need to hold onto. It will implement AWS, or MongoDB, and be accessible from the server. It will store any data we need and be what implements our solution for the Providing Secure and Accessible Storage of User Data challenge.

5. Conclusion

The ability for anyone to participate in conservation by gathering data on the environment is important for the future. Few people in very rural and poor areas are able to actively participate in an equitable and sustainable way, and our app aims to solve this problem. Through solving the challenges as described above, we will provide the tech many people need in order to contribute to the broader scientific community. By creating a highly compatible and non-resource-intensive app we will ensure as many people as possible can participate and use our solution, and through the verification of the data, we can be certain that the data provided to us by our users is helpful and reliable in the future. Through the creation of this app and solving the challenges that face this development we can help contribute to the cause of conservation and scientific advancement.