Mountain Lion Detection System Software Design Specification

Created by Vince Alihan, Carson Mucho, and Kaelin Facun

I. System Description:

This software system was requested by the San Diego Parks and Recreation. The purpose of this software is to detect mountain lions in various state parks throughout San Diego County. The San Diego Parks and Recreation wants this system to be able to warn patrons who enter the parks, that if a mountain lion is nearby, a park ranger will tell them to move away from the area as it is unsafe. The idea is that there will be various sensors throughout the park. The system will use an animal detection system created by the Animals-R-Here company to ping different locations of various animals and update the park's database. The system is designed to only be used by park rangers to see where animals are for themselves, so only they can access the data. This document will break down the various requirements such as user, system (functional and non-functional), and other features that this software system is specified to include.

II. Software Architecture Overview:

Below is our designated software architecture diagram, as well as our UML diagram for the Animals-R-Here system. Given the California State Parks system's interest in adopting our program, our detection system heavily depends on servers and databases accessible through a wifi connection and a computer.

Animal-R-Here Architecture

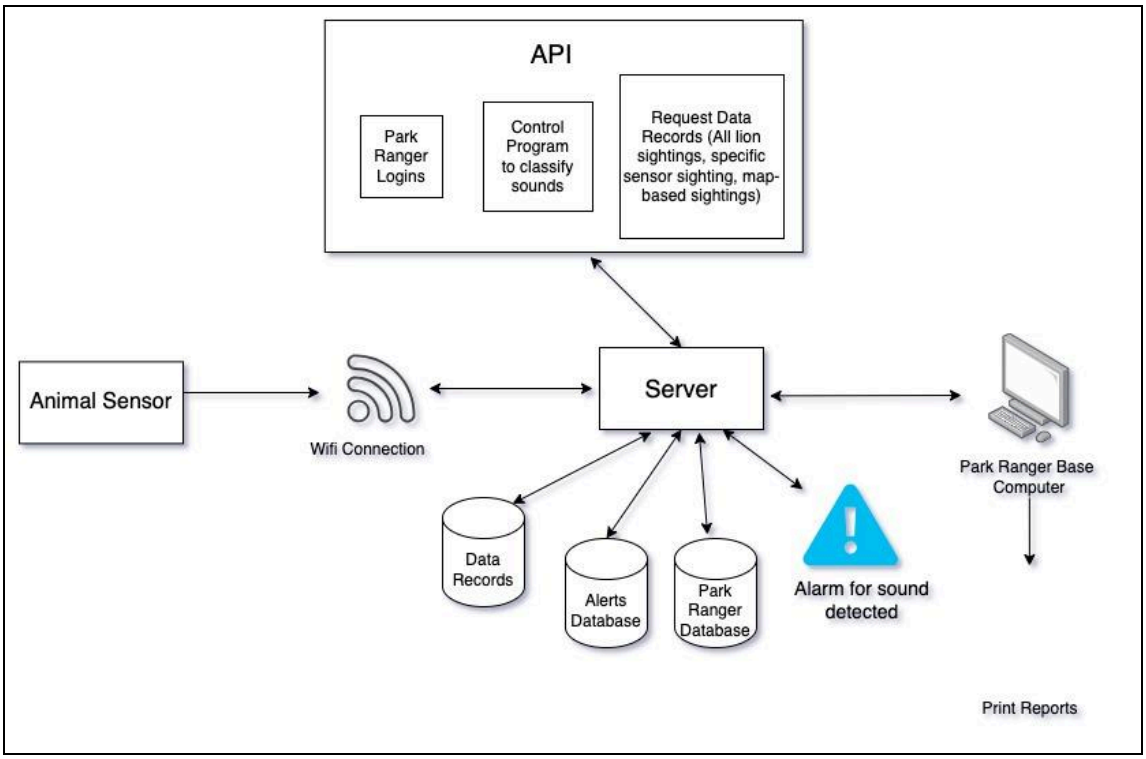

Figure 2.1: Animal-R-Here Software Architecture

Our software system begins with the **Animal Sensors** to detect mountain lions in close proximity, which are connected to **Wifi** in order to relay information to the **Server**. The **Server** is connected to 3 Databases, such as the **Data Records** which contains all mountain lion detections by date and all the locations they were detected at and were already classified by the ranger, the **Alerts Database**, which contains all alerts detected which are for the Park Ranger to classify, and the **Park Ranger Database**, which contains all Park Ranger information such as user preferences and information specific to the ranger logged in. These databases send and receive information to and from the Server. The Server is also connected to an **Alarm**, which sounds whenever an alert message is received from the animal detection system. Attached to the Server is the **Application Programming Interface (API)**, which provides functionality for the

server information, to be used by the **Park Ranger Base Computer** connected to the server. In this interface, functionality is provided to allow park rangers to log into the computer, a control program which allows the park ranger to classify sounds detected as definite, suspected, or false, and an option to request data records stored in the database. All of these functions can be accessed by the Park Ranger Base Computer connected to the Server, which can print reports with the information provided by the server connected to the databases, such as all mountain lion detections by date, all detections at a specific location, a graphical report showing detections on a map, and a report showing detection classifications by ranger.

Animal-R-Here UML Diagram

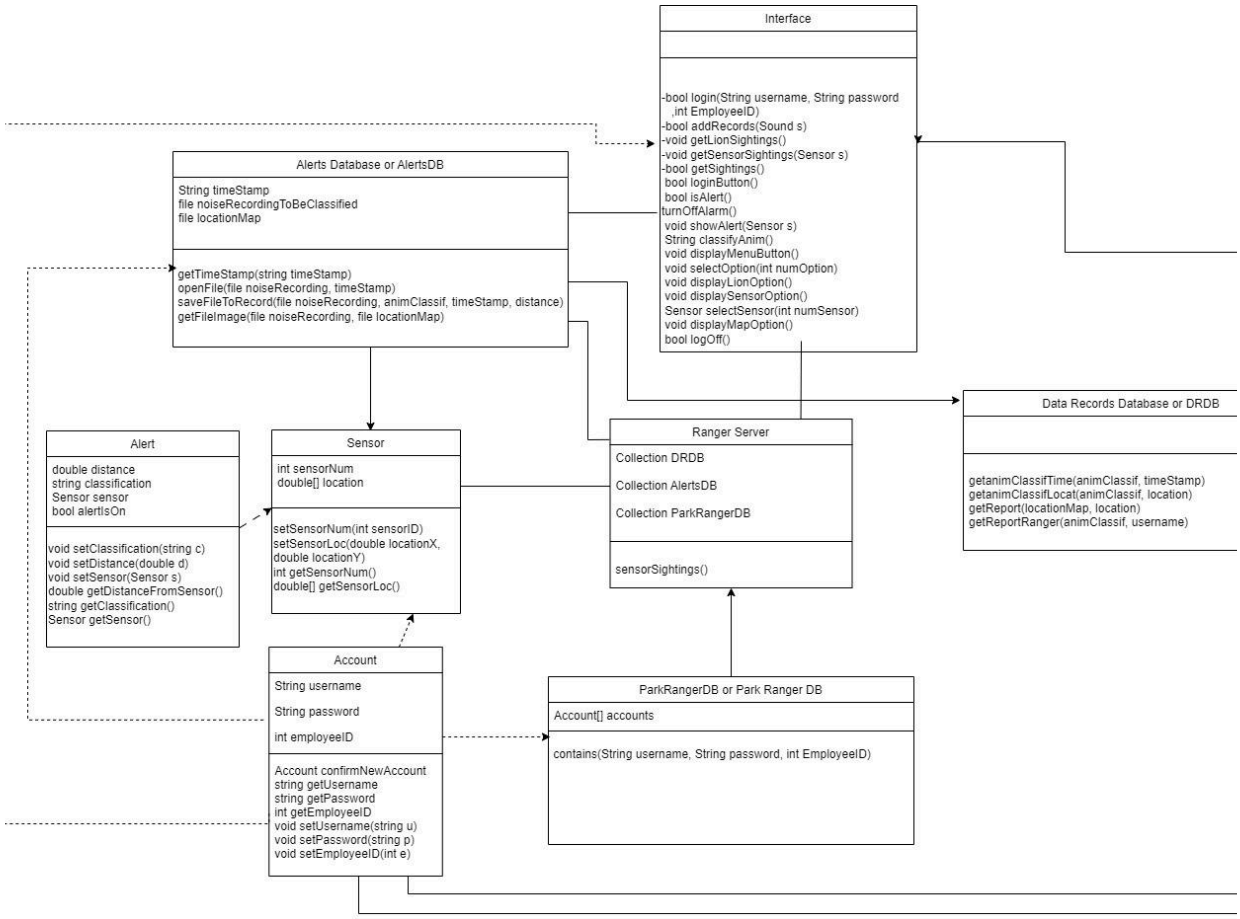

Figure 2.2: Animal-R-Here UML Diagram

*****This UML diagram has been updated from the original. As we began the testing phase we realized that things needed to be reorganized and things needed to be added. The following classes were added/updated:

-**Ranger Computer:** Ranger computer did not make sense for holding all of the account information, so we scrapped that and changed it to Account. The cleaned up the Ranger Computer and ParkRangerDB classes

-Alert: we added a whole new alert class because it made sense with how many different member variable it include and it tidied up classes like Data Records, Sensor,

Sensors: The animal sensor is connected to the Alerts Database from the Ranger Server as well as the Ranger Computer. The sensor program contains *bool animHere and double distance*.

- *bool isAnimHere(bool animHere)*: This is the alert that sets off the indication that an animal is in the vicinity of the sensor. Once indicated, the alarm sets off the sound recording and takes track of the time alerted. In the Ranger computer, the command *turnOffAlarm()* turns this boolean to false to turn off the alarm until there is another sound in the vicinity.
- *getDistance(distance d)*: This function collects the distance of the sound and tracks the vicinity from the sensor. This distance is collected and recorded in the databases to be used from the Alert and Data Record databases.

Account(was Ranger Computer): This class holds all of the information that a park ranger account would have. That includes a username, password, and EmployeeID. This class deals with getting and setting all of this information and as well validating the account. List below are its methods:

-*Account confirmNewAccount()*: This verifies that the username, password, and EmployeeID are in the database and returns the Account object that the method was called on

-*string getUsername()*: This method returns the Username that the park ranger put in.

-*string getPassword()*: This method returns the password that the park ranger put in.

-*int getEmployeeID()*: This method returns the EmployeeID that the park ranger in.

-*void setUsername(string u)*: Updates the Username to something else than what it currently is.

-*void setPassword(string p)*: This method updates the password to something else than what it currently is in case the user has forgotten it.

-*void setEmployeeID(int e)*: An employeeID cannot be changed once it has been set for the first time. Although this method can be called, it will only update the EmployeeID the first time it is called.

Ranger Server: The Ranger Server makes up the entirety of the system by keeping the collections of the databases of the Data Records database, Alerts database, and the Park Ranger database. The server contains *sensorSightings()*.

- *sensorSightings()*: Retrieves a report from all three databases to print out a specified report requested by the Park Ranger. Prints out the location, animal classification, Park Ranger who classified it, and the graphical location of the map.

Interface: The interface of the software controls what the user (park rangers) are able to do with the software system and is what the user sees when accessing the computer. Currently, there are several methods that the user can use and access from the interface. The following methods are included in the Interface: login(String username, String password, int EmployeeID), addRecords(Sound s), getLionSightings(), getSensorSightings(Sensor s), getSightings(), loginButton(), isAlert(), showAlert(Sensor s), classifyAnim(), displayMenuButton(), selectOption(int numOption), displayLionOption(), displaySensorOption(), selectSensor(int numSensor), displayMapOption(), logOff()

- *login(String username, String password, int EmployeeID)*: this method will verify that the user who is trying to login is a park ranger. By using the parameters of username, password, and EmployeeID, it searches for that criteria in the ParkRangerDB to verify the user is allowed to access the Sighting information.

Will have a boolean return type; returns true if the user has access and will then process, otherwise will return false and kick the user out.

- *addRecords(Sound s)*: the addRecords method will allow the user to add the new dataRecord to the AlertsDB. The method will take the alert parameter to add for when new reports are requested/ The alert includes information like animal Classification, timeStamp, and distance from sensor. Will have a boolean return type, and will return true every time since the user can always add data
- get*LionSightings()*: this method will retrieve all of the confirmed Lion Sightings based on the park rangers confidence of the sound. It will iterate through all of the alerts in the database and retrieve all of the ones that are positive for lion sightings. The method then returns a list of Alert type sightings to the user
- *getSensorSightings(Sensor s)*: This method gets all of the alerts from a specific sensor. It takes in the Sensor parameter and then returns a List of all the sightings from that sensor to the user.
- *getMapOfSightings()*: This method will retrieve all Sightings within a 2 mile radius of the park. It iterates through the alert database and uses the sensor number and sighting distance to populate a list that has all of those sightings. This list is then returned to the user.
- *loginButton()*: This method enacts once the user has put their information in the designated boxes. Inside it uses the login method from the Interface class using the user imputed data. Returns a bool that is based on the return of login() from Interface.

- *isAlert()*: This method is automatically checked once the user logs in. This method checks all of the sensors to see if there was an alert. If there was, it updates a variable with which the sensor was alerted and returns true. If no detection, then returns false.
- *showAlert(Sensor s)*: Ran once isAlert is checked and confirms there was a detection. Takes in the parameter of a Sensor which is pulled from when isAlert() is run. Once the Sensor has been confirmed it will display on screen the sound and location and the user then needs to classify it
- *classifyAnim()*: Once the user has listened to the sound of the animal they are given three options; definitely a mountain lion, could be a mountain lion, and not a mountain lion. Once a choice has been made returns a String of the classification to be added to the records.
- *displayMenuButton()*: Once the user has been verified through the loginButton() method and an alert has been taken care of the displayMenuButton is automatically run. The Menu consists of 4 possible choices which are displaying all of the mountain lion sightings, displaying every detection from a specific sensor, and displaying a visual representation of all the sightings within two miles of the park.
- *selectOption(int numOption)*: Each option is given a number and based on what the user clicks on, updates the numOption variable which is the parameter of this method. Using that number makes sure that the right option is run and will then run that method.

- *displayLionOption()*: If the user selects option one, they will be shown a list of all confirmed mountain lion detections. This is done by the database being iterated and sorting by animal classification.
- *selectSensor(int numSensor)* This helper method is inside displaySensorOption() which helps the user determine which sensor they want to show from. It will gather every detection from the sensor number put into the parameter and return a list of alerts.
- *displaySensorOption()*: When this second option is displayed, the user specifies which sensor they would like to see results from and then using the helper method selectSensor() displays those results.
- *displayMapOption()*: The third and final data option will display a map of sightings that are within two miles of the park. The alert database is iterated through and every detection within two miles is added to a list and is then displayed on screen.
- *logOff()*: The last option the user can pick is to log off the system. This resets the program back to the start, before the user logged in. Returns true if done successfully, which should be every time when it is run.

Databases: All of the databases of our software system are located in our servers. Currently, we have four databases dedicated to the San Diego County Parks and Recreation Department, but in the future, we plan on building multiple servers for multiple locations across California, if California State Parks would like to expand out of San Diego.

- **Alerts:** The Alerts Database contains all of the new detected recordings that need to be classified by the Rangers under the detected, suspected, and false labels. The Database

contains *String timeStamp*, *file noiseRecordingToBeClassified*, as well as *file locationMap.*

- *String timeStamp:* Holds the time when the alert was made. Can be reached through the Ranger Computer through *printReports(parameter[reportType])*.
    - *getTimeStamp(string timeStamp)*: Retrieves the timestamp of when the recording was recorded.
    - *openFile(file noiseRecordingToBeClassified, timeStamp)*: *openFile()* opens the file with the *file noiseRecordingToBeClassified* as well as *timeStamp*, to associate the noise file and the time recorded.
    - *saveFileToRecord(file noiseRecording, timeStamp)*: Saves the file to Data Records Database to be kept for future reference for retrieval and information.
    - *getFileImage(file noiseRecording, file locationMap)*: Retrieves the alert's graphical map image location as well with the noise recording.
- *file noiseRecordingToBeClassified*: Holds the noise recording in a file within the database that needs to be classified by the Park Ranger.

- **Park Ranger:** The Park Ranger Database consists of all of current Park Rangers within the San Diego Parks and Recreation Department and holds their account information for logins. The following method is contained in this class:
    - *contains()*: *contains()* as mentioned above, checks if the account is valid and holds the parameters of *contains(String username, String password, int EmployeeID)*.

- **Data Records:** The Data Records Database contains the classified file records of all of the alerts that have been previously analyzed by a park ranger. The following methods are contained within the data Records database:
    - *String getanimClassif*: Holds the value of the classification that has been analyzed by the Park Ranger.
        - *getanimClassifTime(animClassif, timeStamp)*: Retrieves the animal classification and the time of the recording of the alert.
        - *getanimClassifLocat(animClassif, location)*: Retrieves the animal classification and the distance of the alert from the sensor.
        - *getReport(locationMap, location)*: Retrieves the file of the graphical image from the Alerts Database and the distance from the sensor.
        - *getReportRanger(animClassif, username)*: Retrieves the animal classification and the Park Ranger who made the classification report.

Alert: This class deals with the Alerts that come from the sensors. The member variables for this class would be distance, classification, sensor, and alertIsOn. Distance is how far the alert is from the sensor, classification is how confident they are about the sound being a mountain lion, sensor is what sensor it came from, and alertIsOn is if the alert just happened, so the user can chck it. List below are the methods and their descriptions:

- *void setClassification(string c)*: This method will set the classification of the Alert based on how the park ranger interprets it. It can either be definitely a mountain lion, could be a mountain lion, or is not a mountain lion.

- *void setDistance(double d)*: This method will set the distance of teh alert from the sensor. This allows us to accurately see where the detection was(up to three meters from the sensor).
- *void setSensor(Sensor s)*: This method sets what sensor the alert came from. This helps later on with interacting with the interface class when we want to grab every alert from a specific sensor.
- *void setIsAlertOn(bool a)*: This method will set whether the alert from the sensor needs to go off to alert the park ranger. The sensor will send the signal of the alert and this member variable will be updated based off if there is alert or not
- *double getDistanceFromSensor()*: This method will return the distance that alert was from the sensor.
- *string getClassification()*: This method will return the classification of the alert, based off what teh park ranger classified it as when viewing the report.
- *Sensor getSensor()*: This will retrieve the sensor that a specific alert came from. This helps for when we want to pull all reports of a specific sensor to view the data.

III. Testing Plan for Animal-R-Here Project

As soon as we were about to begin testing we realized that our software system could be optimized by adding on to our UML diagram. We updated the Park Ranger Computer class to be an Account class. This made more sense when we determined what we wanted to test. We also added an Alert class. This pairs with sensor, so it was beneficial to give it its own class, since the biggest aspect of the project is analyzing every alert the sensor

The two features that we would be testing would be Alert and Employee Log-In. Through these tests, we conducted three scopes of granularities or unit, functional, and system.

Alert:

- **Unit:** For the Alert class, the unit testing would be testing out the predictability of the program when classifying sound. The function getDistanceFromSensor() sets the distance based on how strong the sound is. The computer takes in the noise detection and translates it to the distance based on the soundwaves and stores it into the database. For now, there is a scale for the intensity of sound that corresponds to the levels of classification of definite, suspected, or false. Below is the pseudocode for the unit case.

```
Sensor sensor
bool alertIsOn

if alertIsOn is true  //definite
if sensor > 50
        setDistance(5) //suspected
if sensor < 50
        setDistance(10) //false
if sensor >= 10
        setDistance(15)
```

- **Functional:** For the Alert class we are going to be testing that when we set the Sensor if it is being set to the right one. If this feature were to fail we would have messed up data of alerts pointing to the wrong sensor which messes with the database. Below is the pseudo code of how we would test this:

```
Sensor s
s.setSensorNum(1738)
s.setSensorLoc(41.7749, 87.6152)
Alert a
a.setClassification("definitely")
a.setDistance(33.8) //in meters
a.setSensor(s)
if (a.getSensor == s) {
return PASS
else {
```

```
return FAIL
}
```

- **System:** For the system to function correctly, it should be able to recognize if the Sensor is able to detect noise and make an Alert to store it into the database and alert the main ranger computer. When noise is detected, the sensor should be able to also recognize the distance from the sensor where noise was detected, which will assist the user on the main computer to be able to classify the noise as definite, suspected, or a false alarm. The alerts should also be sent by the correct and nearest sensor to the detected noise. For example, if two sensors pick up the noise, the nearest sensor should pick it up and alert the main computer, to prevent confusion and the incorrect judgment that there were two sightings. Once an alert for noise detection is sent to the ranger computer, the user will analyze the alert and classify it, and the classification and audio file is sent to the database for record keeping.

Employee Log-In:

- **Unit:** For the Employee Log-In, the team decided to test the verification if the employee trying to log-in is a valid employee. The unit test is designed to check the Employee database if the employee matches a valid employee identification credential, known as *username, password,* and *employeeID,* through the database. The unit test also allows purposely wrong credentials to also test the program can identify an invalid employee identification. Below is the pseudocode.

```
Account a
a.setUsername(“SmokeyTheBear”)
a.setPassword(“OnlyUPreventForestFires”)
a.setEmployeeID(123456)
```

```
if (a.confirmNewAccount() == a)
        return PASS
else
        return FAIL
```

- **Functional:** The functional test will test that when the user attempts to log in with their credential that the account is verified and can login. We will test this by creating an account and then using a method from the Interface class to see if they are allowed to login. This is done through the login() method. This test checks if the classes match together well and work as they are intended to. Below is the pseudo code to test this:

```
Account a

a.setUsername("MountainLion32")
a.setPassword("CatsRCoool098")

if (login(a.getUsername(), a.getPassword)
        return PASS
else
        return FAIL
```

- **System:** The user should be able to walk up to the Ranger Computer and be able to input their username, password, and employeeID. If the user's credentials are found in the Park Ranger Database, they will be able to access the computer's functions, such as viewing alerts sent to the computer (viewAlerts()), turn off the current alarm (turnOffAlarm()), classify alerts sent to the computer (classifyAlert()), and print reports of the defined type (printReports(reportType)).

IV. Data Management Strategy:

Explanation of SWA (why we didn't update it):

We chose not to update our SWA because we felt that the number of databases we already had was sufficient. The databases we had were Data Records, Alerts Database, and Park Ranger Database. All have their own special use which will be explained in more detail later on. The software design is simple enough to where we only need to store the data of the alerts, the alerts themselves, and the park ranger information database.

Specify data management strategy (SQL)

For our data management strategy we decided that SQL would be better suited for our software system. Listed below are the relational database diagrams and how they are connected.

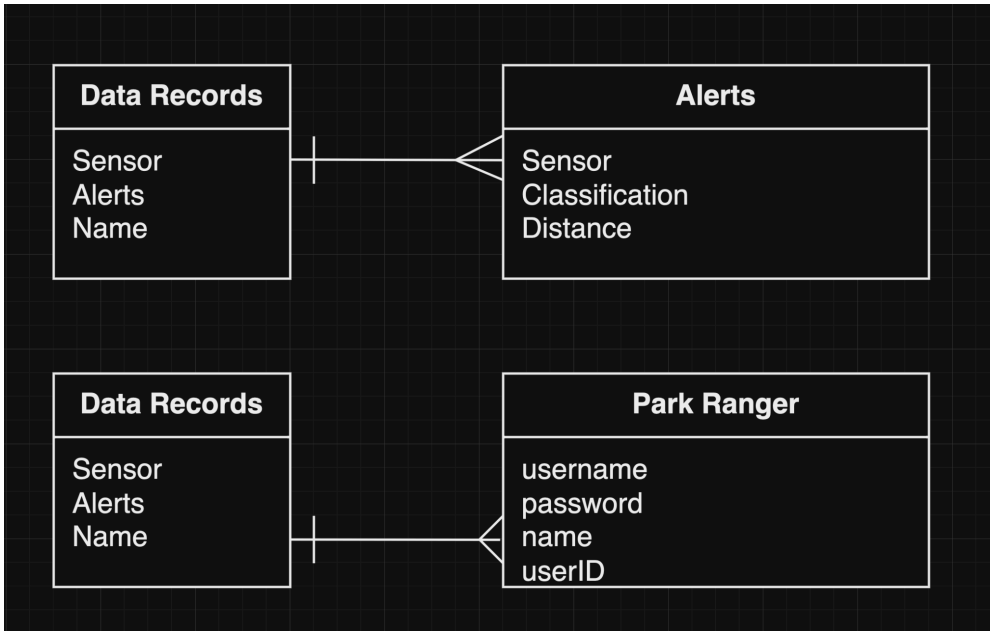

Figure 4.1: Data Records Database

Data Records is a database that has multiple relations of one to many. We only have one data records spot, but many alerts and park rangers using the software system. In our first relational diagram we have Sensor being the primary key to access the data records database and Sensor in the Alert class is our Foreign key. Our other relation is between Data Records and Park Ranger. The primary key in Data Records is Name and the foreign key in Park Ranger is Name. The data will also be encrypted because the information it provides should only be accessible by the park rangers who are authorized.

Discussing design decisions:

We chose SQL because we knew our databases had a relation to one another. For example the data in Alerts Database has to go into our Data Records database. Another feature of SQL is that we can use the language of SQL to manipulate the data, which is necessary when classifying each alert that comes in. Another advantage of going with SQL is that our data is not going to have any large scale changes to it. The whole point of our software system is to analyze data that we receive from the Sensors/Alerts, and SQL is great at data analysis, so it was only another benefit in selecting SQL.

Tradeoffs with our choice:

We decided on SQL because we found its ability to efficiently manage and analyze large data sets makes it incredibly necessary in how we store our data. With the large amount of potential alerts and other information needing to be stored in our 3 databases, SQL is indispensable in allowing us to view and manage our data. When we went with SQL, we decided with some tradeoffs to consider over something such as NoSQL. For example, we decided to prioritize

consistency over scalability, which is something that NoSQL is more capable of providing than SQL. However, we decided on consistency, meaning that our data would be more accurate, valid, and have a strong integrity from data attacks as opposed to NoSQL. When our data is more consistent, we ensure the reliability and correctness of applications that rely on the database. It also helps prevent data corruption, anomalies, and logical errors that can occur when inconsistent or invalid data is stored or manipulated. We also opted in for using SQL for its ability to utilize complex query capabilities, which involve multiple criteria, operations, and different relationships between data elements in our database. With these complex query capabilities, our system is better able at analyzing large volumes of data and assisting in generating meaningful conclusions from each analysis. We decided to go with the complex capabilities of SQL, over the simpler and more predictable capabilities that NoSQL offered. These were some tradeoffs that we considered before ultimately deciding to go with SQL for our databases.

How many databases we chose and why and how we split up the data:

We chose 3 databases. Data Records, Alerts Database, and Park Ranger Database. To reiterate, each database holds a different type of information in each. The Data Records contains all data of mountain lion detections, the locations of those detections, and their classifications. The Alerts Database contains all incoming alerts for the park ranger to classify. The Park Ranger Database contains all Park Ranger information, such as their user preferences and specific information for each park ranger (i.e: log-in info). We split up our databases because it facilitates better security and protection for our data in each database from being leaked and accessed. The possibility of 1 database with all the information being compromised is much higher than the possibility of having multiple, because if all the information is shared in one database, one simple compromise

of security can provide someone all of the information, rather than a portion of it. In our Software Design, we would be less concerned of the loss or leak of incoming alert data in the Alerts Database or Data Records, over the loss or leak of personal park ranger information, which would definitely be a serious security concern, which is why we decided to keep the types of data separate, in order to keep crucial data away from data we would be more willing to give up. Although, at the same time, we take our data security very seriously and will implement measures to keep all 3 of our databases secure.

Alternatives to the NoSQL Method for Data Management:

The team explored various approaches to managing our data before deciding on SQL management as the optimal solution for our project. Below, we outline the alternative strategies we considered.

- **NonSQL Data Management:**

NonSQL Data management would use a spreadsheet-like data organization to organize all of the databases in our project. Throughout our project, the Animals-R-Here project heavily relies on sound sensors that would accurately depict the motion of mountain lions in the area. With this, the team needed to establish accuracy and credibility of our project of recording every suspected sound as an animal. For extra accuracy, a ranger would also listen to the recording and label it as a suspected, detected, or false source of sound. Since there are a lot of sounds being recorded 24/7, NonSQL would be a viable option to organize the data as there are a lot of variables that need to be recorded along with the sound such as location, date, ranger who classified it, etc. Below are some screenshots of what the NonSQL data would look like:

Employee

Employee #	Last Name	First Name
000123456789	Alihan	Vince
000234567891	Anderson	Flower
000345678912	Facun	Kaelin
000456789123	Finnigan	Seamus
000567891234	Markus	Michelle
000678912345	Mucho	Carson

Figure 4.2: Employee NonSQL Database

Alerts

Date Detected	Distance (ft)	Alert Classif.	Ranger Classif.	Ranger #
04/14/2022	0.5	Suspected	FALSE	000567891234
07/07/2023	0.7	Detected	Detected	000567891234
12/29/2023	1.7	FALSE	Suspected	000234567891
02/16/2024	2.5	Suspected	FALSE	000567891234
04/08/2024	1.4	Dectected	Detected	000123456789
05/15/2024	3.5	Suspected	Suspected	000123456789

Figure 4.3: Alerts Database

Data Records

Date Detected	Sound File	Map Location	Ranger #
04/14/2022	soundFile.mp3	32°47'N 129°52	000567891234
07/07/2023	soundFile1.mp3	52°55'N 1°28'W	000567891234
12/29/2023	soundFile2.mp3	45°34'S 72°04'V	000234567891
02/16/2024	soundFile3.mp3	36°20'N 43°08'E	000567891234
04/08/2024	soundFile4.mp3	16°26'N 102°50	000123456789
05/15/2024	soundFile5.mp3	56°09'N 10°13'E	000123456789

Figure 4.4: Data Records Database

NonSQL Data Management would seem like a great choice in the prototype phase as it has scalability and cost-efficient for the first prototype, but as we progress into the future of our project, SQL-based data management would be the better option as SQL provides better

precision in its structure and establishes relationships between data, as well as better for more complex but organized systems. As our system is relatively organized with strict categories and databases involved, such as data records, alerts, and employees, our team ultimately decided that SQL is the optimal option.

V. Development Plan and Timeline:

The current team members of the Animal-R-Here consist of three team members: Carson Mucho, Vince Alihan, and Kaelin Facun.

Carson Mucho is our Lead Back-End Developer and is responsible for the back-end development and system of our project. As a Lead Back-End Developer, he is responsible for the system's operating systems and interactions with the databases and servers, and must ensure that the system works properly back-wise.

Vince Alihan is our Lead Database Architect and is responsible for the databases required for the Animal-R-Here Project. As a Database Developer, he is responsible for the prompt and efficient organization of information that will be provided by local park rangers, as well as upkeep the system based on the user's needs as the project progresses outside of California.

Kaelin Facun is our Lead Front-End Developer and is responsible for the front-end development of the interface of our project and is responsible for ensuring the project fulfills our park ranger's requirements in a system. As a front-end developer, she is responsible for the user interface and their interactions with the system and must ensure that it works properly front-wise.

Animal-R-Here Project will be expected to be released before the Spring 2025 or March 2025. As hiking trails and park visitor occupancy peaks during spring season, we want to make

sure that the Animal-R-Here program is installed in local parks to ensure public safety of both visitors and mountain lions alike.

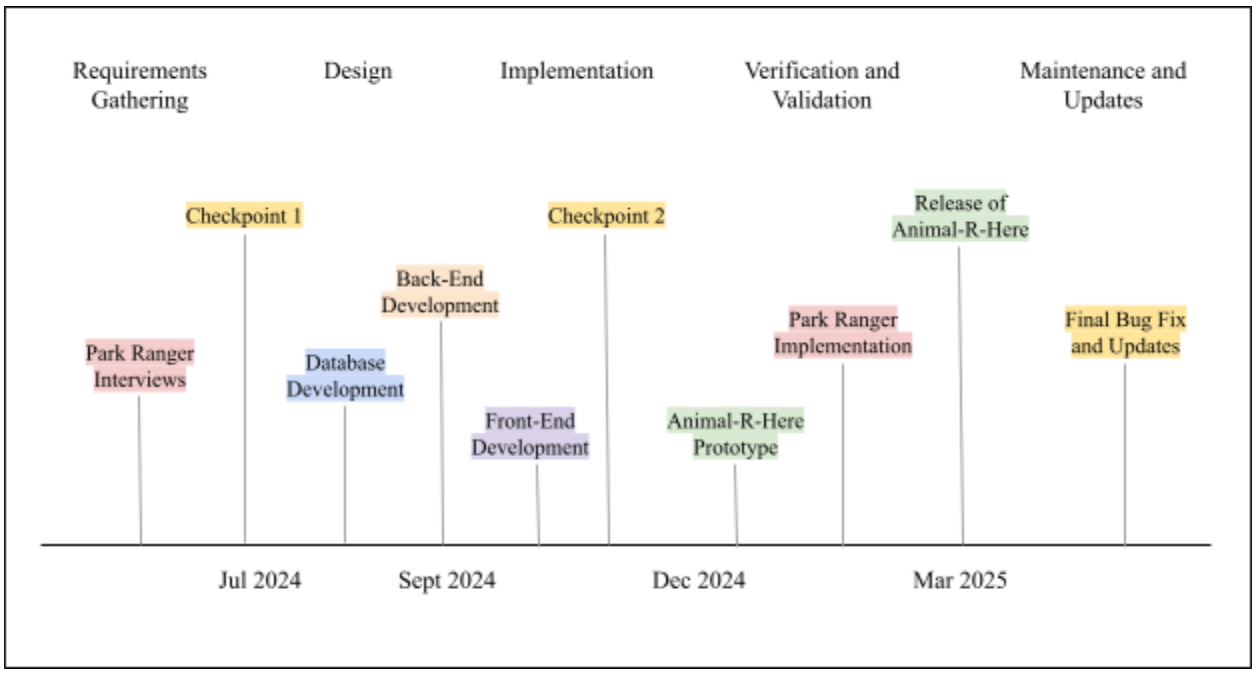

Figure 1.3: Expected Timeline for the Animal-R-Here Project.

Once the project is approved in March, 2024, the Animal-R-Here team can be expected to conduct Park Ranger Interviews as the team can gather requirements of what the park rangers would like to see through this project. Through these interviews, the project team can clarify expectations as well as set realistic expectations for the desired end goal.

Checkpoint 1 and Checkpoint 2 are used for meetings within the team and the client to ensure that the expectations are being met with the project, i.e as interface appearance and functions and database functionality.

From July 2024 and beginning of Dec 2024, the project team is expected to start development on the Animal-R-Here project. Vince Alihan, will start developing the databases and collections and work alongside Carson Mucho to ensure storage and project functionality from database to back-end development. Kaelin Facun will also start on development with the

front interface and work alongside Mucho as well to ensure functionality with back-end and front-end development.

The team is expected to release a prototype beta at the beginning of Jan. 2024. Once the Animal-R-Here Prototype is released, the San Diego Parks and Recreation Department will test the prototype at a small, secure location in San Diego to ensure satisfaction with the project and that it fulfills their needs as Park Rangers. The timeline also provides space from testing if the project is still not up to par or for bug fixes in the original code.

Once the project is satisfactorily completed, the final release of Animal-R-Here will happen in Mar. 2024, and further maintenance and updates will be conducted after the release to ensure any further bug fixes as well as updates with the program.