UNIVERSITY OF WESTERN AUSTRALIA

Project Report

Smart Livestock Tracking System

Group: 22

Student Name	**Student ID**
1. Cedrus Dang	24190901
2. Ella Zhang	23844446
3. Faraz Khan	24427672
4. Yifan Lu	24094198
5. Yapei Chen	23764722

Supervised by
Dr. Atif Mansoor
School of Physics, Mathematics and Computing
CITS5506 - Internet of Things
20 October 2024, Semester 2

Abstract: The Smart Livestock Tracking System aims to address significant challenges in modern livestock farming, such as limited visibility, theft, and operational inefficiencies, by integrating IoT technologies. Traditional methods like manual inspections and record-keeping fail to provide real-time monitoring, resulting in delayed responses to livestock wandering, theft, health issues and significant financial and operational challenges for farmers. Our system utilises TTGO LoRa32 devices equipped with GPS tracking and LoRa communication and an MQTT protocol connection to the cloud, which serves to achieve accurate real-time livestock monitoring across large grazing areas with high accuracy, ease of scale-up and low cost. The system includes several additional features besides live tracking, such as geofencing alerts, which notify farmers if animals stray outside designated boundaries and data analysis tools that optimise farm operations. During the valuation, power-efficient modes, including deep sleep, significantly extend battery life and can last up to 50 days, while GPS data accuracy is 99% in outdoor areas. Although there are limitations, such as disconnecting GPS signals in indoor areas, the system still provides a cost-effective and reliable livestock management solution that suits vast pasture-lands. Future enhancements will focus on optimising power consumption, improving scalability, incorporating additional sensors for health monitoring, enhancing the user interface for better accessibility, and using AI models to analyse animal behaviours further and fill up the sleep period in GPS data collection. This project demonstrates the transformative potential of IoT technologies in livestock farming, enhancing efficiency, sustainability, and economic benefit.

Keywords: Livestock tracking, Geo-fencing, GPS, LoRa, MQTT, Internet of Things

I. INTRODUCTION

Managing large herds across expansive grazing areas has become increasingly complex and challenging in the evolving livestock farming landscape. Traditional methods, such as manual logs and physical inspections, have become insufficient for handling the issues of modern livestock management. A timely, accurate solution is required for effective decision-making, particularly in large-scale operations.

There are two main challenges modern livestock farms have to face and require innovative solutions. Firstly, the high risk of theft and loss is a significant concern, as farmers often lack the technology to detect and respond to straying and stolen animals quickly, creating financial losses for the farmers [1]. Secondly, manual tracking methods require substantial staffing. They are prone to human error, combined with the lack of information and slow response from the lack of live data. This creates operational inefficiencies from high costs and the inability to predict and handle strange animal behaviours quickly. The delays in detecting abnormal animal behaviours can negatively impact animal welfare, leading to health problems that ultimately affect the quality and yield of livestock products [5].

In response to these challenges, this project aims to develop the Smart Livestock Tracking System, which leverages IoT technologies to provide real-time GPS tracking, geofencing, and data analysis. By integrating Long Range (LoRa) and MQTT communication protocols, the project expects to create a low-power, long-distance monitoring system, making it an ideal solution for vast rural areas. This project will aim to enhance livestock monitoring and security and significantly improve operational efficiency through timely interventions and data-driven insights into animal behaviour. The following sections will explore this system's design, implementation, and evaluation, demonstrating how IoT solutions can transform modern livestock farming by addressing its most pressing challenges.

II. TECHNICAL CONTENT

1. Existing Solutions

There are multiple solutions from various previous works for real-time tracking of animals with geo-fencing. GPS was chosen for accurate location tracking in the same project despite its high power use and potential signal loss in areas like forests [2][5]. Some other systems also integrate Long Range (LoRa) technology for its low power usage and ability to cover large distances, making it better for farming than other technologies like GPRS, which uses more power and is less scalable in rural areas [3][6][7][9]. Custom alerts and insights into animal behaviour can also help farmers monitor their livestock more effectively, reduce theft, and enhance farm efficiency [2][7][6]. Other tracking methods, such as video, were deemed less appropriate for outdoor settings due to occlusion and image distortion[4]. Beacon-based trilateration was also used. However, their complexity and lower accuracy made them less suitable [8].

2. Designing Approach

After determining all potential requirements and their solution from the literature, six crucial requirements and their solution is summarized in order to create a smart tracking livestock system that is both practical and effective:

- **High Accuracy Tracking data:** For a live tracking system, accuracy is crucial, especially with the reliability of the geo-fencing function in the system. The system uses GPS (GNSS) technology due to its theoretical short fixing period (a few seconds in perfect conditions) and high accuracy (in open field), which will provide high accuracy for the live functions.
- **Power Efficiency:** Battery life optimisation is crucial for wearable devices. Specifically, the battery charging process is resource-exhausted for large-scale operations as the herd size can be up to thousands of animals in most cases. To solve the problem, incorporating GPS power-saving mode, deep sleep mode for night time (inactive and indoor period), and a tested efficient delay period in GPS data collection and transmission were used. These methods also were optimised to balance power efficiency with tracking accuracy.
- **Efficient Communication:** For devices to use in base station communication, the LoRaWAN protocol was chosen due to its theoretical long range, up to 15 kilometres in open fields, and low power consumption. For the base station to servers and servers to the user interfaces, MQTT was chosen for live position data due to its publish/subscribe architecture and efficient bandwidth usage.
- **Scalability:** The system is designed to handle an increasing number of livestock without compromising performance. AWS Cloud cloud infrastructure is utilised for scalable clouding servers and databases, making developing projects of various sizes possible.
- **High Usability:** The user interface was designed to be simple but still deliver needed insights for farmers. It can also be used on multiple devices, such as smart phones, desktops, and tablets. This ensures that farmers can interact with the system efficiently and on-the-go, ensuring up-to-date herd monitoring. Geo-fencing also provides utility value to the users as now they can depend on this feature and receive alarms in multiple ways, such as pop-ups, SMS and even phone calls, depending on their liking.
- **Cost-Effectiveness:** The system balances technical capability with affordability. The system aims to significantly reduce initial setup and operational costs using the license-free communication protocols LoRa and MQTT. The system uses affordable Lily TTGO ESP32 LoRa modules for devices and the base station. For servers, it used scalable AWS cloud infrastructure. This makes the system a practicable solution for the livestock industry with tight profit margins.

3. System Architecture

The overall system architecture of the Smart Livestock Tracking System is shown in the block diagram below:

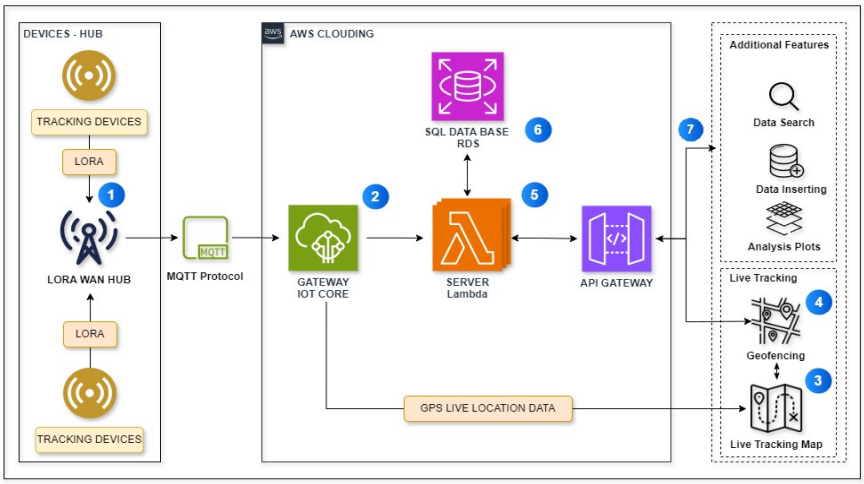

Figure 1: Block Diagram

Initially, the wearable devices on the animal will collect GPS data after a period of delay and then transmit the data to the LoRa WAN Hub (Base station). This live GPS data will then be sent to IoT Core Gateway using MQTT protocol. The Live Tracking Map function in the web app will then use this live data as a subscriber and show the GPS marks on the Google API map. The Geo-fencing feature is also connected to this live tracking map and will alert the user with a pop-up (which can be connected with the AWS SMS, email and phone calls sender if required) using the Geo-fencing setup that the user has set up and saved in the database. The live data has also been collected from the broker by the AWS lambda function and sent to the RDS database for storage. This data will be queried for additional features: data search and analysis plots (Heat map and timeline plot) by API protocol. Animals' information from data inserting feature and user information also uses API protocol to save in the RDS database.

4. Hardware

The system's hardware is wearable tracking devices and the LoRa WAN hub.
The hub uses a Lily TTGO ESP32 LoRa module as its main component. The prototype will be connected to the main power source of 5V-1A and a WiFi network with 2.5 GHz without additional components; however, when moved to the commercial stage, a cover box should be used to protect it from environmental factors. The figure of the prototype hubs has been connected with the main power source and has additional batteries for debugging as below:

Figure 2: LoRa WAN Hub

The device uses a similar Lily TTGO ESP32 LoRa module and is powered with a 20,000mAh-5V-1A ROMOSS battery with a screen showing the power percentage. Those components will then be encased by a box that has a wearable strap to attach it to livestock. The LoRa frequency used to communicate this device with the hub is 915MHz (ISM band of Australia). However, a different version of the LoRa module needs to be selected in different areas, such as LoRa 868MHz in Europe or LoRa 433MHz in some Asian countries. The design for the device and wearing position are showed as follow:

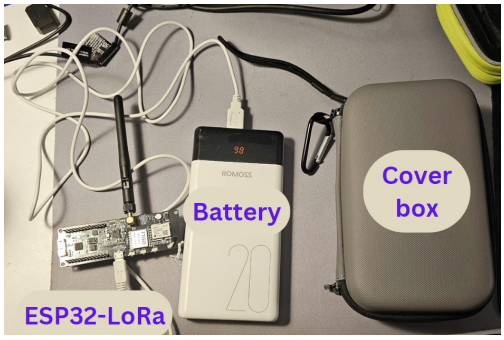

Figure 3: Wearable Tracking Devices components

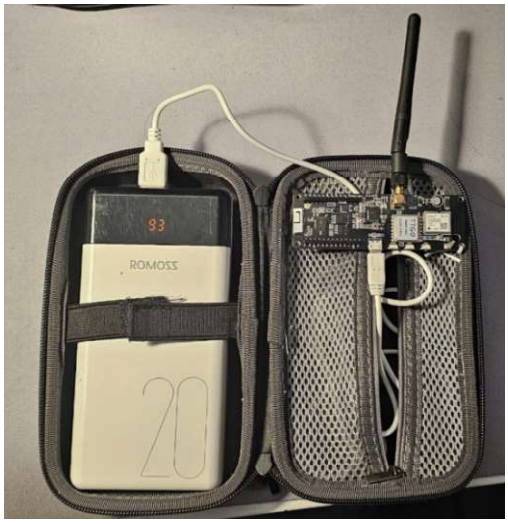

Figure 4: Assembled Wearable Tracking Devices


Figure 5: Wearable Tracking Devices with strap

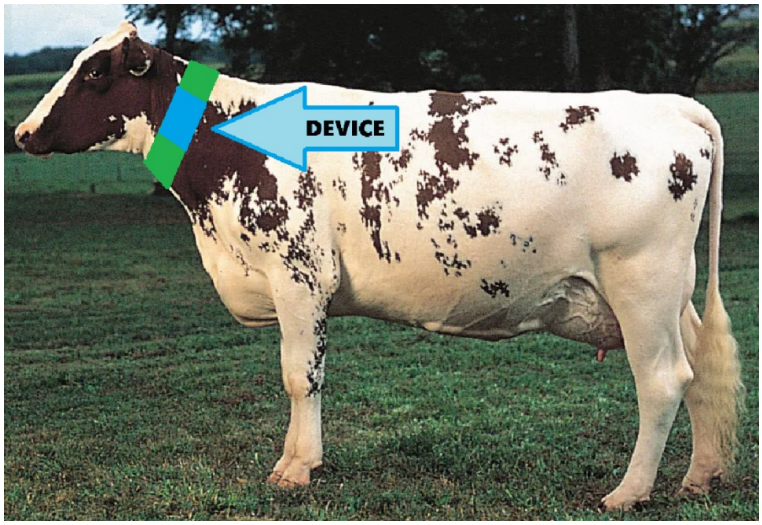

Figure 6: Wearing position of the Device

5. Software

5.1. Firmware for Wearable Devices

The firmware of wearable devices includes GPS data acquisition, power management, and LoRa data transmission. It uses the TinyGPS++, LoRa, and TimeLib libraries for GPS, LoRa and time management, respectively. The firmware reads the GPS data (coordinates, speed, time, etc.), applies a time zone adjustment (to UTC +8 for Perth), and formats the data as a list for transmission. A security key for Pre-Shared Key (PSK) Authentication is included as the first component of the list. Once the GPS is fixed and the satellite data is received, the firmware will use LoRa to send data to the hub. After the first data is successfully sent, the GPS will change to low-power mode and go into a delay period, which is adjustable and, for testing purposes, is now 5s, to extend battery life.

5.2. Firmware for Base Station

The firmware of the LoRaWAN hub manages LoRa data reception and authentication. It also handles AWS IoT Core communication using MQTT protocol. It uses LoRa, WiFiClientSecure, and PubSubClient libraries for LoRa communication, secure Wi-Fi connection, and MQTT data transmission to AWS. The firmware will catch the incoming LoRa data from wearable devices. It then will check the security key for PSK authentication, and if it is verified, the firmware will process the remaining GPS data to a JSON object. This JSON object will then be published to AWS IoT Core Gateway via MQTT using AWS's security protocol. After processing, the firmware returns an acknowledgement to the device over LoRa for debugging and continues to listen for new packets.

5.3. AWS Cloud Infrastructure and User Interface

The cloud infrastructure for the system was built using AWS IoT Core Gateway, Lambda, RDS, and API Gateway. Python with Flask library was used for the webserver.
The front end of the website was developed with Bootstrap, a CSS and JS framework for rapid prototyping. For styling, we applied the 'Quartz' theme from Bootswatch, a Bootstrap-based theme. The website was made responsive by using Bootstrap's flex layout. Additionally, components such as modals were utilized for interactivity. Some personalized styling was also done, which included the use of `Poppins` as the default font, changing the default background colours, and others. Google Map API and relevant libraries were integrated into the user interface's map component. Google Map API renders the Google map on the client side while data are parsed in the client device. The map is updated with markers and relevant symbols accordingly. For real-time tracking, the client connects to the AWS IoT Device SDK and listens to the defined topic.

When a message about that topic is published, which should contain the geo-location and other information about livestock, the client-side script parses the JSON and updates the map with markers accordingly. The data is also saved in the cookies to avoid losing previous data points when page switching happens. The cookie is deleted when the user is logged out.
When connecting to the MQTT client from AWS IoT Device SDK, we encountered Node.js dependencies in the provided code. To address this, we compiled the AWS SDK-related scripts using Browserify, allowing them to run in the browser as standard JavaScript. The speed plotting was done using the `Plotly` library. Plotly provides interactive graph components, and it was leveraged to show the speed plot on the analytics page. The design for user interface as follow:

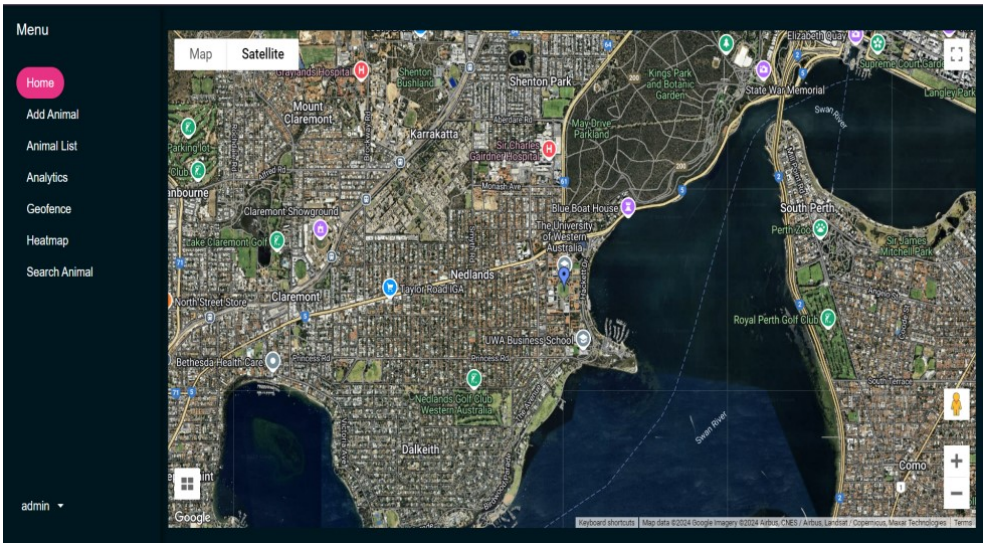

Figure 7: Tracking Live Map

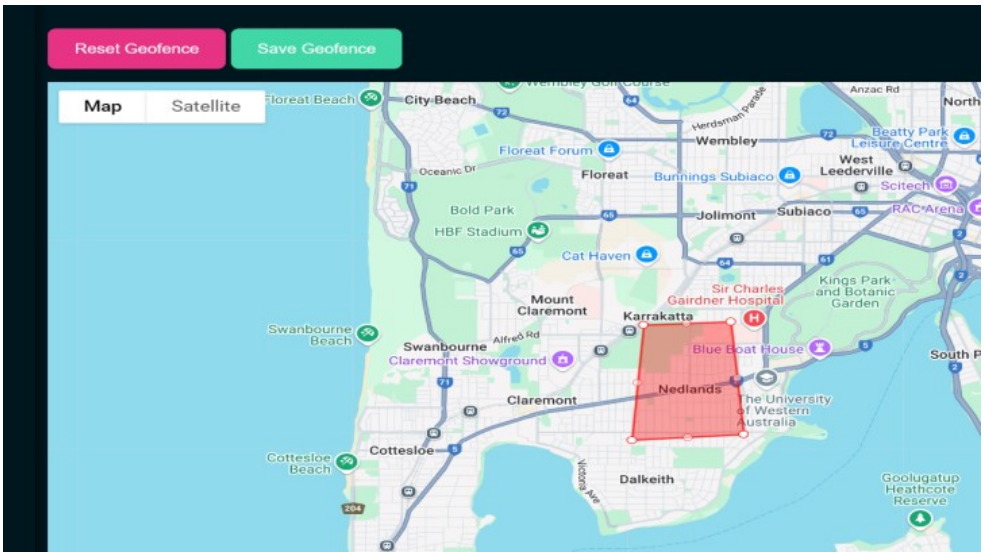

Figure 8: Geo-fencing set up interface

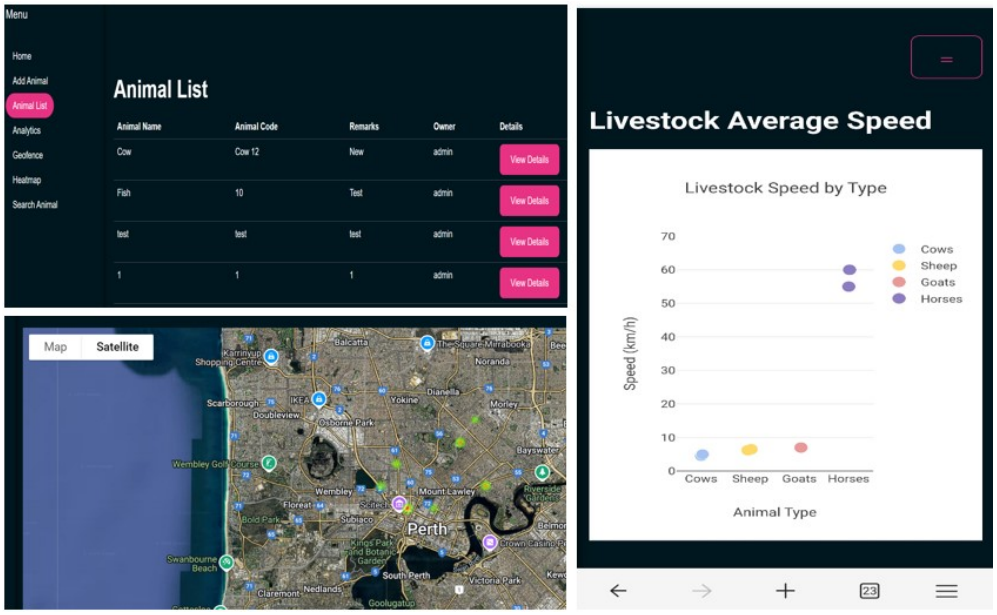

Figure 9: Additional features

6. Evaluation

The evaluation procedures focus on six categories: GPS fix accuracy and proficiency, LoRa connection capacity, MQTT connection reliability, user interface workflow, geo-fencing functionality, and battery capacity testing to assess the system's performance.
The GPS evaluation was set up using five scenarios: open area, under canopy - low density, under canopy- high density, under roof, and inside the building. Test durations were 20 minutes for each. Figure as follows:

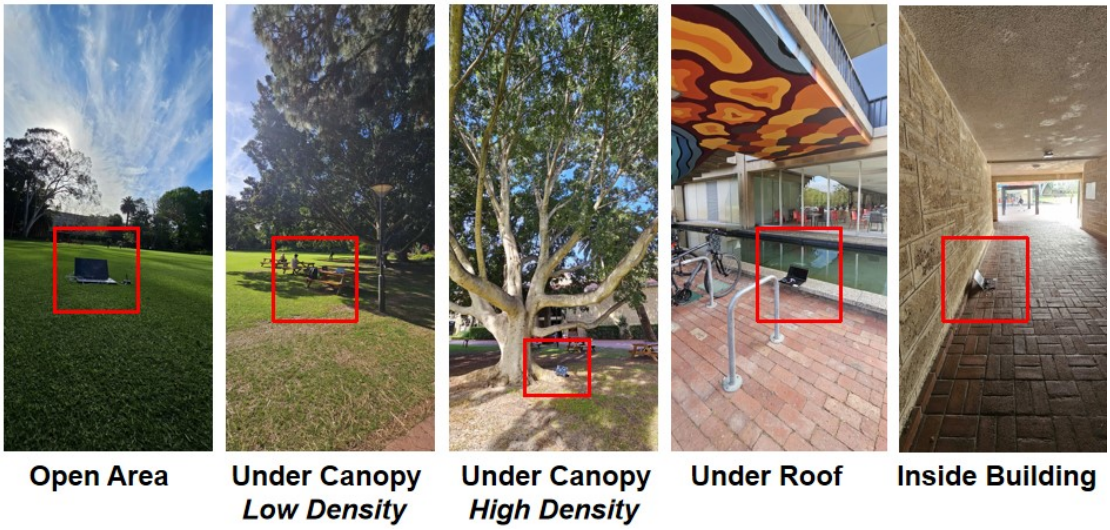

Figure 10: GPS testing scenarios

The GPS testes used test tools such as Andruino IDE serial console, GMap GPS service, AWS test tool for IoT Core, and the project's live map. The testes's result as follow:

Table 1: GPS testing result

Scenario	Accuracy	Mode	GPS fixing time (s)
Open area	99%	Normal Mode	20
	99%	Save Power Mode	22 Variance: 1 ~ 4
Under canopy (Low density)	99%	Normal Mode	20
	99%	Save Power Mode	22 Variance: 1 ~ 4
Under canopy (Highdensity)	99%	Normal Mode	20
	99%	Save Power Mode	22 Variance: 1 ~ 4
Under Roof	99%	Normal Mode	20
	99%	Save Power Mode	22 Variance: 1 ~ 4
Inside building	0.0%	Normal Mode	+ inf
	0.0%	Save Power Mode	+ inf

The test results show a consistent GPS fixing performance across all open and semi-obstructed environments, with a near-perfect accuracy of 99%, and the GPS fixing times remain stable across both Normal Mode and Save Power Mode, with a slight increase in fixing time in Save Power Mode, around 1 to 4 seconds variance. This indicates that power-saving modes do not impact GPS performance.

However, GPS fails to acquire a fix for both modes inside buildings. This indicates that the GPS signal was completely obstructed. Overall, the system demonstrates excellent performance in open and semi-obstructed areas but can not used if the tracking device is

indoors where GPS signals cannot penetrate. This mean area like closed barns will not have GPS signals at all.

In the LoRa connection capacity evaluation, the signal range was reduced when the hub was in an apartment near a window facing the device when the connection lasted for less than 100 meters. During this indoor testing, data transmission became unreliable, with intermittent breaks and invalid data occurring before the device disconnected and the first transmission after reconnect. This indicates that LoRa can significantly impact if there are high density obstacles, which in real scenarios could be mountains, cars or buildings. Due to this, the hub should be installed in high place and the high density that can obstacle the LoRa signals in pasturelands should also be cleared. In contrast, the LoRa connection remained stable for at least 1 km without disruptions in an open field.

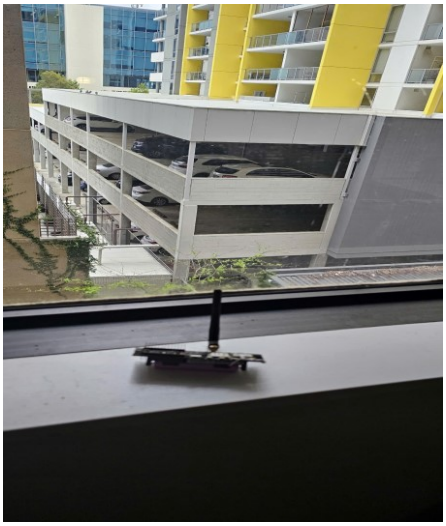

Figure 11: LoRa indoor testing scenarios

MQTT Performance: The waiting time between each transmission varies when using AWS IoT Core to send data to the web interface, affecting real-time tracking responsiveness. This phenomenon is unpredictable and cannot be calculated previously due to the complexity and security methods of AWS services.

Geo-Fencing: The tests show that if the mark appears in the live map, the geo-fencing alarm is activated right after. However, the accuracy of geofencing tests was also heavily influenced by the sleep period of the GPS sensor and the latency of both LoRa and AWS services. A shorter sleep period will improve accuracy but will also increase power consumption.
Battery Tests: With three tests were conducted to measure how long for each percent of the power capacity to decrease in the 20,000mAh battery, the average is around 1 hour.

$$\text{Power Consumption (mAh/hour)} \approx \frac{\text{Battery Capacity (mAh)} \times \text{Percentage of Capacity (\%)}}{\text{Time (hours)}}$$

Equation 1: Power consumption estimation

Using equation 1 to calculate the power consumption rate, the average power consumption rate is about 200mAh/hour.

$$\text{Battery Life (days)} \approx \frac{\text{Battery Capacity (mAh)}}{\text{Power Consumption (mA/h)}} \times \frac{1}{\text{Active Hours Per Day (h) x}} \times \frac{\text{Sleep Period (s)}}{\text{Active Period (s)}}$$

Equation 2: Battery life estimation

Then, using equation 2 to calculate the days of battery life, with a 30-second sleep period, the battery could last for approximately 25 days. Meanwhile, with a deep sleep period from 6 PM to 6 AM (12 hours per day), the battery could last up to 50 days.
An additional evaluation is the prototype cost, this will measure the affordable and scalable of the project, the cost summary as follow:

Table 2: Prototype cost summary

No.	Items	Description	Amount	Cost/unit (AUD)	Web Address
1	TTGO LoRa32	LoRa32 GPS device with Wi-Fi and Bluetooth - 915MHz	2	43.00	Temu's store
2	Romoss Battery	20,000 mAh Power storage	1	43.00	Amazone.au's store
3	Cables	Connection components	2	0.00	In TTGO LoRA 32 package
4	AWS Cost	Cloud services, varies based on usage	*vary*	*vary*	https://aws.amazon.com/pricing/
		SUM		**129.00**	

The total cost of one tracking device is 43 AUD, while the LoRa hub costing 43 AUD. Compared to the potential savings from reducing livestock loss and increasing operational efficiency, the system offers a cost-effective solution for farmers, particularly in large-scale operations.

III. CONCLUSION:

This project aims to develop a cost-effective and reliable livestock tracking system to solve the challenges and improve efficiency in tracking livestock in the husbandry industry using IoT devices and technologies. The primary objectives are to provide high-accuracy tracking data, good power efficiency, efficient communication, flexible scalability, high usability and reasonable cost-effectiveness tracking solutions. Archiving those objectives will improve farmers' on-farm management efficiency and economic benefits.

We have successfully developed a livestock tracking system using TTGO LoRa32 devices to achieve accurate long-range real-time GPS tracking. Although GPS can not work indoors, it works effectively in outdoor environments with 99% accuracy. The project has also successfully extended battery life through finetuned power-saving modes, sleep periods, deep sleep periods, and efficient communication protocols while still having a balanced tracking accuracy trade-off. This extension leads to a practical working duration of up to 50 days while using a 30-second sleep period, 6 am to 6 pm deep sleep, save power mode in GPS, and 20,000 mAh battery capacity. Using LoRa and MQTT communication for reliable and resource-saving data transmission, the system can now cover a vast area. The distance between the hub and the device can be up to 15km, with a minimum of 1km. Geofencing and additional features such as database query, management, and analysis plots also successfully provide a solution for effective and deep management. Reasonable hardware selection, which uses the TTGO ESP32 LORA module, has provided a low-cost system, with a prototype cost of 43 AUD for the hub main model and 43 AUD for each device main model (Without covers and additional equipment), compared to the value of the animal and the duration life of the hardware, this is a financially practical.

The project also faced several limitations. First is indoor GPS signal loss in indoor area. During the test, GPS connectivity indoors is unable to form. Next is the trade-off between power saving and geofencing accuracy. The power-saving modes, deep sleep and extended sleep periods significantly increase battery life but come at the cost of tracking and geofencing accuracy when it creates a gap between each location update. Shortening the GPS sleep period improves accuracy but at the expense of battery longevity. This limitation underscores the challenge of balancing energy efficiency with precise geolocation tracking. Another one is the limited LoRa range if there are high-density obstacles. Although LoRa communication is effective in open fields (up to 1 km), its range is severely limited when it has

to go through walls and other objects with large covers. During tests, the LoRa signal could only transmit reliably within a 100-meter radius when the hub was placed indoors, near the window. This creates challenges in environments with significant obstructions. However, as in most cases, the pasturelands will be in open fields, this limitation is insignificant to the system. There was noticeable latency in data transmission when using AWS IoT Core and MQTT protocols. This delay decreases the responsiveness of real-time data updates, which reduces the system's effectiveness for immediate alerts and livestock tracking. Another problem is the cost of cloud solutions. Cloud services, such as the AWS RDS database, introduce significant costs that may not be sustainable for all users and will need to be managed. There is a need to balance between cloud and local computing solutions to manage ongoing operational costs effectively.

Future work on this project could focus on several key areas to improve system performance and usability. Firstly, SMS, email and phone call senders from AWS can notify farmers when an animal steps out of its geo-fencing enclosure by integrating with Amazon's Simple Notification Service (SNS). In addition, the balance between optimising power consumption and tracking data accuracy needs to be further analysed and addressed. One solution is using prediction models to fill the GPS data during the sleep period using course and speed values. This allows reduced power consumption without compromising tracking and geofencing accuracy. Another problem is expensive clouding service, this may hinder the product's competitiveness. Balancing local servers and other solutions to save running costs can be a good solution. The system can also integrate sensors to monitor animal health and incorporate weather and environmental data along with AI models for more comprehensive analysis and prediction of animal behaviour. These insights enable farmers to optimise livestock management strategies and improve farm economics, creating selling points for the product.

Overall, this project demonstrates the potential of IoT technologies in enhancing livestock management, offering significant improvements in efficiency, scalability, and cost-effectiveness for farmers. As more features are added, the system can monitor the comprehensive status of animals and farm. A smarter, more automated farm will be realised.

ACKNOWLEDGMENTS

This project made use of OpenAI's GPT for code suggestion and research assistance, as well as Grammarly for proofreading and grammar correction. These tools were instrumental in improving the efficiency and quality of the work.

REFERENCES

[1] Australia’s agricultural crime could be prevented with new technology, [Online]. Available: https://www.agricrew.com.au/news/australia-s-agricultural-crime-could-be-prevented-and-reduced-with-the-help-of-new-technology/59849/. Accessed: Oct. 2024.

[2] J. Cabezas, R. Yubero, B. Visitación, J. Navarro, M. J. Algar, E. D. Cano, et al., "Analysis of accelerometer and GPS data for cattle behaviour identification and anomalous events detection," *Entropy*, vol. 24, no. 3, p. 336, 2022. doi: 10.3390/e24030336.

[3] R. Casas, A. Hermosa, Á. Marco, T. Blanco, and F. J. Zarazaga-Soria, "Real-time extensive livestock monitoring using LPWAN smart wearable and infrastructure," *Appl. Sci.*, vol. 11, no. 3, p. 1240, 2021. doi: 10.3390/app11031240.

[4] S. Han, A. Fuentes, S. Yoon, Y. Jeong, H. Kim, and D. S. Park, "Deep learning-based multi-cattle tracking in crowded livestock farming using video," *Comput. Electron. Agric.*, vol. 212, p. 108044, 2023. doi: 10.1016/j.compag.2023.108044.

[5] R. Handcock, D. L. Swain, G. Bishop-Hurley, K. P. Patison, T. Wark, P. Valencia, et al., "Monitoring animal behaviour and environmental interactions using wireless sensor networks, GPS collars and satellite remote sensing," *Sensors*, vol. 9, no. 5, pp. 3586–3603, 2009. doi: 10.3390/s90503586.

[6] Q. M. Ilyas and M. Ahmad, "Smart farming: an enhanced pursuit of sustainable remote livestock tracking and geofencing using IoT and GPRS," *Wireless Commun. Mobile Comput.*, vol. 2020, p. 6660733, 2020. doi: 10.1155/2020/6660733.
[7] B. C. Mallikarjun, K. V. Suresh, R. N. Sumana, S. Pooja, H. S. Prajwal, and P. Pooja, "Animal tracking system using LoRa technology," *Int. J. Adv. Res. Basic Eng. Sci. Technol.* (IJARBEST), vol. 5, no. 7, pp. 359–363, 2019. [Online]. Available: https://ijarbest.com/journal/v5i7/1950. Accessed: Oct. 2024.
[8] N. A. Molapo, R. Malekian, and L. S. Nair, "Real-time livestock tracking system with integration of sensors and beacon navigation," *Wireless Pers. Commun.*, vol. 104, no. 2, pp. 853–879, 2018. doi: 10.1007/s11277-018-6055-0.
[9] L. Schulthess, F. Longchamp, C. Vogt, and M. Magno, "A LoRa-based and maintenance-free cattle monitoring system for alpine pastures and remote locations," in *Proc. 11th Int. Workshop Energy Harvesting & Energy-Neutral Sens. Syst.*, 2023. doi: 10.1145/3628353.3628549.

APPENDIX A: DEVICE SOURCE CODE

The device, base station, web server, geo-fencing feature, and source code are in the appendices below. Due to the large size of HTML files, they will not show here; instead, they will be published on the GitHub repository, link as follows:

https://github.com/Cedrus1994/IoT_UWA_Project1_G22.git.

1. Device Source Code

This code is uploaded to the device to collect GPS data and send it using the LoRa protocol. It also has a deep sleep period from 6 am to 6 pm, a 30s sleep period, a security key and time zone adjustment.

How to: Change variables in the code to match the case demand. Also, change the type of animal whose ID is required and the shared security key with the hub. Arduino IDE was used to develop the code for the hardware and reset the device using three buttons on the board.

```
#include <TinyGPS++.h>
#include <SoftwareSerial.h>
#include <LoRa.h>
#include <SPI.h>
#include <TimeLib.h>
#include <esp_sleep.h>

// GPS Configuration
static const int RXPin = 34, TXPin = 12;
static const uint32_t GPSBaud = 9600;

// LoRa Configuration
#define SCK     5    // Serial Clock pin for LoRa (SPI)
#define MISO    19   // Master In Slave Out pin for LoRa (SPI)
#define MOSI    27   // Master Out Slave In pin for LoRa (SPI)
#define SS      18   // Slave Select pin for LoRa (SPI)
#define RST     14   // Reset pin for LoRa
#define DI0     26   // Digital I/O 0 pin for LoRa
#define BAND    915E6 // Set LoRa frequency to 915 MHz (Australia's frequency band)

// Other Variables
int security_key = 13;  // Default security key for the device
int id = 1;  // Unique identifier for the device
String type = "Cow";  // Type of device or data
String data_to_send;  // Data that will be sent to the hub
int adjust_hour = 8;  // Adjust to Perth time (UTC+8)
```

```
int sleep_duration = 5 * 1000;  // Sleep duration in milliseconds
int max_retries = 10;  // Set a reasonable retry count for sending data
bool gps_time_received = false;  // Flag to indicate if GPS time is received

int deep_sleep_start_hour = 6;  // Start deep sleep at 6 AM
int deep_sleep_end_hour = 18;   // End deep sleep at 6 PM

// Create objects
TinyGPSPlus gps;
SoftwareSerial ss(RXPin, TXPin);

void setup() {
  Serial.begin(115200);
  ss.begin(GPSBaud);

  // LoRa Setup
  SPI.begin(SCK, MISO, MOSI, SS);
  LoRa.setPins(SS, RST, DI0);
  if (!LoRa.begin(BAND)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  // GPS Power Mode Configuration - Full Power for initial fix
  ss.println("$PMTK161,0*28");  // Disable standby mode (GPS stays on continuously)
  ss.println("$PMTK220,30000*1B");  // Set GPS update rate to 0.2Hz (update every 30
seconds to save energy)
  Serial.println("Setup complete. Waiting for GPS fix...");
}

String prepareDataToSend(TinyGPSPlus &gps) {
  // Adjust the hour to Perth time (UTC + 8)
  int adjusted_hour = (gps.time.hour() + adjust_hour) % 24;
  // Prepare the data to send
  String utcDateTime = String(gps.date.year()) + "-" + String(gps.date.month()) + "-" +
String(gps.date.day()) +
                       " " + String(adjusted_hour) + ":" + String(gps.time.minute()) + ":" +
String(gps.time.second());
  String data = String(security_key) + "," +
                String(id) + "," +
                utcDateTime + "," +
                String(gps.speed.kmph()) + "," +
                String(gps.location.lng(), 6) + "," +
                String(gps.location.lat(), 6) + "," +
                String(gps.course.deg()) + "," +
                String(type);
  return data;
}

bool sendDataToHub(String data) {
  int retries = 0;
  while (retries < max_retries) {
    // Check LoRa status and reconnect if needed
    if (!LoRa.begin(BAND)) {
      Serial.println("Reconnecting LoRa...");
      LoRa.end();
      if (!LoRa.begin(BAND)) {
        Serial.println("LoRa reconnection failed!");
      } else {
        Serial.println("LoRa reconnected successfully.");
      }
```

```
    }

    if (LoRa.beginPacket()) {
      LoRa.print(data);
      LoRa.endPacket();

      // Print the data to the console
      Serial.println("Data sent: ");
      Serial.println(data);

      return true;  // Assume success if packet was sent
    } else {
      Serial.println("Failed to start LoRa packet. Retrying...");
      retries++;
      delay(1000);  // Wait before retrying
    }
  }
  return false;  // Failed to send after max retries
}

void loop() {
  // Get the current hour
  int current_hour = hour();

  // Check if it is within deep sleep hours
  if (current_hour >= deep_sleep_start_hour && current_hour < deep_sleep_end_hour) {
    Serial.println("Within deep sleep hours. Going to sleep...");
    esp_sleep_enable_timer_wakeup(12 * 60 * 60 * 1000000LL); // Sleep for 12 hours (or until
external wake-up)
    esp_deep_sleep_start();
  }

  // Read GPS data
  while (ss.available() > 0) {
    if (gps.encode(ss.read())) {
      // Check if GPS location is valid (has a fix)
      if (gps.location.isValid() && gps.location.isUpdated() && gps.time.isValid()) {
        Serial.print("GPS fix acquired!");
        gps_time_received = true;

        // Set time directly from GPS (AWST is +0800)
        int adjusted_hour = (gps.time.hour() + adjust_hour) % 24;
        setTime(adjusted_hour, gps.time.minute(), gps.time.second(),
                gps.date.day(), gps.date.month(), gps.date.year());
        data_to_send = prepareDataToSend(gps);

        // Send data to the hub
        if (sendDataToHub(data_to_send)) {
          Serial.println("Data sent successfully!");
          ss.println("$PMTK225,2*2E");  // Enable power-saving mode
        } else {
          Serial.print("Failed to send data");
        }

        delay(sleep_duration);
      } else {
        Serial.print("GPS not fixed yet");
      }
    }
  }
```

```
}
```

2. Hub Source Code

Hub.ino & secret.h: This code enables the TTGO T-Beam device to receive sensor data over LoRa, validate it, and then publish the data as a JSON payload to AWS IoT Core using MQTT protocol.
How to: Change variables in the code to match the case demand and change the share security key with the device. Also, insert the AWS keys and wifi password into the secret.h file. Arduino IDE was used to develop the code for the hardware and reset the device using three buttons on the board.

Hub.ino:

```
//T-BEAM TTGO HUB For LORAWAN. Connect to AWS
#include "secrets.h"
#include <WiFiClientSecure.h>
#include <PubSubClient.h>
#include <ArduinoJson.h>
#include "WiFi.h"
#include <LoRa.h>
#include <SPI.h>

#define AWS_IOT_PUBLISH_TOPIC "hub/pub"
#define AWS_IOT_SUBSCRIBE_TOPIC "hub/sub"

// LoRa Configuration
#define SCK     5
#define MISO    19
#define MOSI    27
#define SS      18
#define RST     14
#define DI0     26
#define BAND    915E6

// Security Key
int expected_security_key = 13;

WiFiClientSecure net = WiFiClientSecure();
PubSubClient client(net);

void connectAWS()
{
  WiFi.mode(WIFI_STA);
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);

  Serial.println("Connecting to Wi-Fi");

  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);
    Serial.print(".");
  }

  net.setCACert(AWS_CERT_CA);
  net.setCertificate(AWS_CERT_CRT);
  net.setPrivateKey(AWS_CERT_PRIVATE);

  client.setServer(AWS_IOT_ENDPOINT, 8883);
```

```
  client.setCallback(messageHandler);

  Serial.println("Connecting to AWS IOT");

  while (!client.connect(THINGNAME))
  {
    Serial.print(".");
    delay(1);
  }

  if (!client.connected())
  {
    Serial.println("AWS IoT Timeout!");
    return;
  }

  client.subscribe(AWS_IOT_SUBSCRIBE_TOPIC);

  Serial.println("AWS IoT Connected!");

  // Publish "Hub connected" message
  StaticJsonDocument<200> jsonDoc;
  jsonDoc["message"] = "Hub connected";
  char jsonBuffer[512];
  serializeJson(jsonDoc, jsonBuffer);
  publishMessage(jsonBuffer);
}

// Function to publish JSON message
void publishMessage(const char* jsonPayload) {
  if (client.publish(AWS_IOT_PUBLISH_TOPIC, jsonPayload)) {
    Serial.print("Published JSON Payload: ");
  } else {
    Serial.print("Failed to publish JSON Payload: ");
  }
  Serial.println(jsonPayload);
}

void messageHandler(char* topic, byte* payload, unsigned int length)
{
  Serial.print("Incoming message: ");
  Serial.println(topic);

  StaticJsonDocument<200> doc;
  deserializeJson(doc, payload);
  const char* message = doc["message"];
  Serial.println(message);
}

void setup() {
  Serial.begin(115200);
  while (!Serial);

  // LoRa Setup
  Serial.println("LoRa Receiver");
  SPI.begin(SCK, MISO, MOSI, SS);
  LoRa.setPins(SS, RST, DI0);
  if (!LoRa.begin(BAND)) {
    Serial.println("Starting LoRa failed!");
    while (1);
```

```
  }

  // AWS IoT Setup
  connectAWS();
}

void loop() {
  // Check for LoRa packets
  int packetSize = LoRa.parsePacket();
  if (packetSize) {
    String receivedData = "";
    while (LoRa.available()) {
      receivedData += (char)LoRa.read();
    }
    processReceivedData(receivedData);
  }

  client.loop();
  delay(1);
}

void processReceivedData(String data) {
  // Split the data into its components
  int commaIndex = data.indexOf(',');
  int received_security_key = data.substring(0, commaIndex).toInt();

  // Check security key
  if (received_security_key == expected_security_key) {
    Serial.println("Security key matched, processing data...");
    String remainingData = data.substring(commaIndex + 1);
    int idCommaIndex = remainingData.indexOf(',');
    String id = remainingData.substring(0, idCommaIndex);
    String message = remainingData.substring(idCommaIndex + 1);

    // Extract individual fields
    String dataList[7];
    int lastCommaIndex = idCommaIndex;
    int currentCommaIndex;

    for (int i = 0; i < 6; i++) {
      currentCommaIndex = message.indexOf(',', lastCommaIndex + 1);
      dataList[i] = message.substring(lastCommaIndex + 1, currentCommaIndex);
      lastCommaIndex = currentCommaIndex;
    }

    dataList[6] = message.substring(lastCommaIndex + 1);

    // Get RSSI
    int rssi = LoRa.packetRssi();

    // Create JSON object
    StaticJsonDocument<200> jsonDoc;
    jsonDoc["ID"] = id;                    // Device ID
    jsonDoc["UTC DateTime"] = dataList[0];   // UTC Date and Time
    jsonDoc["speed"] = dataList[1].toFloat(); // Speed in km/h
    jsonDoc["Lon"] = dataList[2].toFloat();   // Longitude
    jsonDoc["Lat"] = dataList[3].toFloat();   // Latitude
    jsonDoc["Course"] = dataList[4].toFloat();// Course in degrees
    jsonDoc["Type"] = dataList[5];            // Type of data (e.g., "Cow")
    jsonDoc["RSSI"] = rssi;                // RSSI from LoRa packet
```

```
    // Serialize JSON to a string
    char jsonBuffer[512];
    serializeJson(jsonDoc, jsonBuffer);

    // Print and publish the JSON payload
    Serial.print("Received Data (JSON): ");
    Serial.println(jsonBuffer);
    publishMessage(jsonBuffer);

    // Send an acknowledgment back to the sender
    sendAcknowledgment();

    // Confirmation print
    Serial.println("Data successfully processed.");
  } else {
    Serial.println("Invalid security key. Ignoring data.");
  }
}

void sendAcknowledgment() {
  LoRa.beginPacket();
  LoRa.print("ACK");
  LoRa.endPacket();
  Serial.println("Acknowledgment sent to the sender.");
}
```

secret.h:

```
#define SECRET
#define THINGNAME "hub"

const char WIFI_SSID[] = "Insert WiFi ID here";
const char WIFI_PASSWORD[] = "Insert WiFi password here";
const char AWS_IOT_ENDPOINT[] = "AWS IoT Core Endpoint here";

// Amazon Root CA 1
static const char AWS_CERT_CA[] PROGMEM = R"EOF(
-----BEGIN CERTIFICATE-----
.....Insert your key here .....
-----END CERTIFICATE-----
)EOF";

// Device Certificate
static const char AWS_CERT_CRT[] PROGMEM = R"KEY(
-----BEGIN CERTIFICATE-----
.....Insert your key here .....
-----END CERTIFICATE-----
)KEY";

// Device Private Key
static const char AWS_CERT_PRIVATE[] PROGMEM = R"KEY(
-----BEGIN RSA PRIVATE KEY-----
.....Insert your key here .....
-----END RSA PRIVATE KEY-----
)KEY";
```

3. Web Server

The flask application folder should be structured like this:

```
Project/
├── application/ # Main application packaage
│   ├── application.py # Package initialization
│   ├── models.py # Database models (using SQLAlchemy or similar)
│   ├── routes.py # URL routes and view functions
│   ├── instance/ # Database folder
│   │   ├── database.db # Default database for the local server
│   ├── templates/ # Jinja2 HTML templates
│   │   ├── base.html # Base template for inheritance
│   │   ├── index.html # Template for the home page
│   │   └── ... # Other templates
│   ├── static/ # Static files (CSS, JavaScript)
│   │   ├── css/
│   │   │   ├── style.css # Default CSS
│   │   ├── js/
│   │   │   ├── mqtt.js # MQTT client JS
│   │   │   ├── mqtt_compiled.js  # MQTT client compiled JS
```

How to: If deploying locally, execute the application.py file and run the command python application.py (or py application.py in PowerShell). If using AWS Elastic Beanstalk or other cloud services, add all necessary files the platform requires into the folder, compress the project folder into a zip file, and upload it to the platform. Otherwise, follow alternative instructions if there are any.

- **model.py:** The overall functionality of this code is to manage users along with their associated animals and geofences in a relational database using SQLAlchemy.

```
from flask_sqlalchemy import SQLAlchemy
from werkzeug.security import generate_password_hash, check_password_hash

db = SQLAlchemy()

class User(db.Model):
    """
    Represents a user in the system.

    Attributes:
        id (int): The primary key of the user.
        username (str): The unique username of the user.
        password_hash (str): The hashed password of the user.
        animals (relationship): A relationship to the Animal model indicating the user's animals.
        geofence (relationship): A one-to-one relationship with the Geofence model for the user.

    Methods:
        set_password(password): Hashes and sets the user's password.
        check_password(password): Checks if the provided password matches the stored hashed password.
    """

    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    password_hash = db.Column(db.String(128))

    animals = db.relationship('Animal', backref='owner', lazy=True)
    geofence = db.relationship('Geofence', backref='owner', lazy=True, uselist=False)  # One-to-One relationship

    def set_password(self, password):
        """
```

```
        Hashes the password and sets it for the user.

        Args:
            password (str): The plaintext password to be hashed.
        """
        self.password_hash = generate_password_hash(password)

    def check_password(self, password):
        """
        Checks if the provided password matches the user's hashed password.

        Args:
            password (str): The plaintext password to check against the hash.

        Returns:
            bool: True if the password matches, False otherwise.
        """
        return check_password_hash(self.password_hash, password)

class Animal(db.Model):
    """
    Represents an animal owned by a user.

    Attributes:
        id (int): The primary key of the animal.
        name (str): The name of the animal.
        code (str): A unique code for the animal.
        remarks (str): Additional remarks about the animal.
        owner_id (int): The foreign key linking to the User model, indicating the owner of the
animal.
    """

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), nullable=False)
    code = db.Column(db.String(50), nullable=False)
    remarks = db.Column(db.String(200), nullable=True)
    owner_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)

class Geofence(db.Model):
    """
    Represents a geofence associated with a user.

    Attributes:
        id (int): The primary key of the geofence.
        coordinates (JSON): The geographic coordinates defining the geofence area.
        owner_id (int): The foreign key linking to the User model, indicating the owner of the
geofence.

    Relationships:
        user (relationship): A relationship back to the User model.
    """

    id = db.Column(db.Integer, primary_key=True)
    coordinates = db.Column(db.JSON, nullable=False)
    owner_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)  # ForeignKey
to the User model

    user = db.relationship('User', backref=db.backref('geofences', lazy=True))
```

- **routes.py:** The file contains logic regarding how the website will respond to different URL requests.

```
from flask import Blueprint, render_template, request, redirect, url_for, session, flash, jsonify,
make_response
from models import db, User, Animal, Geofence
import json

# Create a blueprint for main routes
main_routes = Blueprint('main_routes', __name__)

@main_routes.route('/index')
def index():
    """
    Render the homepage.

    Returns:
        Rendered template for the index page.
    """
    return render_template('index.html', active_page='home')

@main_routes.route('/register', methods=['GET', 'POST'])
def register():
    """
    Handle user registration.

    GET: Render the registration form.
    POST: Process registration data, create a new user, and redirect to login.

    Returns:
        Rendered template for the registration page or redirects to the login page.
    """
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        # Check if username is already taken
        user = User.query.filter_by(username=username).first()
        if user:
            error_message = 'Username already taken. Please choose another one.'
            session['error_message'] = error_message
            return redirect(url_for('main_routes.register'))

        # Create a new user
        new_user = User(username=username)
        new_user.set_password(password)
        db.session.add(new_user)
        db.session.commit()

        session['success_message'] = 'Registration successful. You can now login.'
        return redirect(url_for('main_routes.login'))

    # Pop error and success messages from session
    error_message = session.pop('error_message', None)
    success_message = session.pop('success_message', None)
    return render_template('register.html', error_message=error_message,
success_message=success_message)

@main_routes.route('/', methods=['GET', 'POST'])
def login():
```

```
    """
    Handle user login.

    GET: Render the login form.
    POST: Process login data, authenticate user, and redirect to the homepage.

    Returns:
        Rendered template for the login page or redirects to the index page.
    """
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        user = User.query.filter_by(username=username).first()

        # Check credentials
        if user and user.check_password(password):
            session['user_id'] = user.id
            session['username'] = user.username
            return redirect(url_for('main_routes.index'))
        else:
            error_message = 'Wrong username or password.'
            session['error_message'] = error_message
            return redirect(url_for('main_routes.login'))

    error_message = session.pop('error_message', None)
    return render_template('login.html', error_message=error_message)

@main_routes.route('/logout')
def logout():
    """
    Handle user logout.

    Clears the user session and redirects to the login page.

    Returns:
        Redirect response to the login page.
    """
    # Clear session
    session.pop('user_id', None)
    session.pop('username', None)

    # Prepare the response for redirection
    response = make_response(redirect(url_for('main_routes.login')))

    # Delete cookies by setting them with an expiration time in the past
    response.set_cookie('livestock', '', expires=0)

    return response

@main_routes.route('/save-geofence', methods=['POST'])
def save_geofence():
    """
    Save or update the user's geofence.

    Receives coordinates via a POST request, either creates a new geofence
    or updates an existing one.

    Returns:
        JSON response indicating success or failure.
```

```
    """
    data = request.get_json()
    user_id = session['user_id']
    coordinates = data.get('coordinates', [])

    if coordinates:
        # Convert list of coordinates to string format for storage
        coordinates_str = json.dumps(coordinates)

        # Check if the user already has a geofence entry
        geofence = Geofence.query.filter_by(owner_id=user_id).first()

        if geofence:
            # Update existing geofence
            geofence.coordinates = coordinates_str
        else:
            # Create a new geofence
            geofence = Geofence(owner_id=user_id, coordinates=coordinates_str)
            db.session.add(geofence)

        db.session.commit()
        return jsonify({'status': 'success', 'message': 'Geofence saved!'}), 200

    return jsonify({'status': 'error', 'message': 'No coordinates provided!'}), 400

@main_routes.route('/get-geofence', methods=['GET'])
def get_geofence():
    """
    Retrieve the user's geofence.

    Returns:
        JSON response with the geofence coordinates or None if not found.
    """
    user_id = session['user_id']  # Retrieve the user_id from the session

    # Retrieve geofence from the database using user_id
    geofence = Geofence.query.filter_by(owner_id=user_id).first()

    if geofence:
        coordinates = geofence.coordinates
        return jsonify({'coordinates': coordinates}), 200

    return jsonify({'coordinates': None}), 200

@main_routes.route('/reset-geofence', methods=['POST'])
def reset_geofence():
    """
    Reset (delete) the user's geofence.

    Returns:
        JSON response indicating the success or failure of the operation.
    """
    try:
        user_id = session.get('user_id')  # Ensure you safely get user_id
        if not user_id:
            return jsonify({'error': 'User not logged in'}), 401

        # Find the existing geofence for the user
        geofence = Geofence.query.filter_by(owner_id=user_id).first()
```

```
        if geofence:
            # Remove the geofence from the database
            db.session.delete(geofence)
            db.session.commit()
            return jsonify({'message': 'Geofence reset successfully!'}), 200
        else:
            return jsonify({'message': 'No geofence found for this user.'}), 404

    except Exception as e:
        # Log the error for debugging
        print(f"Error occurred: {e}")
        return jsonify({'error': 'An error occurred while resetting geofence.'}), 500

@main_routes.route('/add_animal', methods=['GET', 'POST'])
def add_animal():
    """
    Add a new animal for the logged-in user.

    GET: Render the add animal form.
    POST: Process the animal data and save it to the database.

    Returns:
        Rendered template for the add animal page or redirects to the animal list.
    """
    if request.method == 'POST':
        name = request.form['name']
        code = request.form['code']
        remarks = request.form.get('remarks', '')

        if 'user_id' not in session:
            flash('You must be logged in to add an animal.', 'danger')
            return redirect(url_for('main_routes.login'))

        owner_id = session['user_id']

        new_animal = Animal(name=name, code=code, remarks=remarks, owner_id=owner_id)
        db.session.add(new_animal)
        db.session.commit()

        flash('Animal added successfully!', 'success')
        return redirect(url_for('main_routes.animal_list'))

    return render_template('add_animal.html', active_page='add_animal')

@main_routes.route('/search_animal', methods=['GET'])
def search_animal():
    """
    Search for animals based on user input.

    Returns:
        Rendered template showing the search results.
    """
    query = request.args.get('query', '')
    if query:
        animals = Animal.query.filter(
            (Animal.name.like(f'%{query}%')) |
            (Animal.code.like(f'%{query}%'))
        ).all()
    else:
        animals = []
```

```
    return render_template('search_animal.html', animals=animals,
active_page='search_animal')

@main_routes.route('/profile', methods=['GET', 'POST'])
def profile():
    """
    Display and manage user profile.

    GET: Render the user's profile with their associated animals.
    POST: Update the user's password if provided.

    Returns:
        Rendered template for the user's profile page.
    """
    if 'user_id' not in session:
        return redirect(url_for('main_routes.login'))

    user_id = session['user_id']
    user = User.query.get(user_id)

    animals = Animal.query.filter_by(owner_id=user_id).all()

    if request.method == 'POST':
        if 'password' in request.form:
            new_password = request.form['password']
            user.set_password(new_password)
            db.session.commit()
            flash('Password updated successfully.')

        return redirect(url_for('main_routes.profile'))

    return render_template('profile.html', user=user, animals=animals)

@main_routes.route('/heatmap')
def heatmap():
    """
    Render the heatmap page.

    Returns:
        Rendered template for the heatmap.
    """
    return render_template('heatmap.html', active_page='heatmap')

@main_routes.route('/geofence')
def geofence():
    """
    Render the geofence page.

    Returns:
        Rendered template for the geofence.
    """
    return render_template('geofence.html', active_page='geofence')

@main_routes.route('/analytics')
def analytics():
    """
    Render the analytics page.

    Returns:
```

```
    Rendered template for analytics.
    """
    return render_template('analytics.html', active_page='analytics')

@main_routes.route('/animal_list')
def animal_list():
    """
    List all animals in the system.

    Returns:
        Rendered template for the animal list page.
    """
    animals = Animal.query.all()
    return render_template('animal_list.html', animals=animals, active_page='animal_list')

@main_routes.route('/animal_detail/<int:animal_id>', methods=['GET'])
def animal_detail(animal_id):
    """
    Retrieve details of a specific animal.

    Returns:
        JSON response containing the animal's details.
    """
    # Query the animal by ID
    animal = Animal.query.get_or_404(animal_id)

    # Return animal details as JSON
    return jsonify({
        'name': animal.name,
        'code': animal.code,
        'remarks': animal.remarks,
        'owner': {
            'username': animal.owner.username
        }
    })
```

- ***application.py:*** This code sets up a Flask web application with a database. It configures the app to use an SQLite database, defines the database schema in models.py, and sets up the URL routes for the app in routes.py. This code should be run from the server to deploy the website.

```
from flask import Flask
from models import db  # Import the db instance and models
from routes import main_routes

def create_app():
    """
    Create and configure the Flask application.

    Returns:
        Flask application instance.
    """
    app = Flask(__name__)

    # Configure the database URI and other app settings
    # app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/your_database.db'  # Use
for live
    app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///database.db'  # Use a local
SQLite database
```

```
    app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False  # Disable modification
tracking
    app.config['SECRET_KEY'] = 'SMART_TRACKING'  # Secret key for session management
and CSRF protection

    # Initialize the db instance with the app
    db.init_app(app)

    # Import and register the main routes blueprint
    app.register_blueprint(main_routes)

    # Create the database tables if they don't already exist
    with app.app_context():
        db.create_all()

    return app

# Create the Flask application instance
application = create_app()  # Set 'application' to the WSGI callable

if __name__ == '__main__':
    # Run the application on all interfaces at port 8080
    application.run(host='0.0.0.0', port=8080)
```

3.1. mqtt.js: The file contains the code to connect to the AWS IoT device SDK or the MQTT client. Credentials need to be inserted in this code. This file also needs to be compiled by Browserify, and the compiled JS should be linked to the relevant HTML file.

```
const awsIot = require("aws-iot-device-sdk");

// AWS IoT credentials
const clientId = "mqttjs_" + Math.random().toString(36).substring(7);
const mqttClient = awsIot.device({
  region: "ap-southeast-2",
  host: "", // Insert host name
  clientId: clientId,
  protocol: "wss",
  accessKeyId: "", // Insert Access Key
  secretKey: "", // Insert Secret Key
  keepalive: 30,
});

let map;
const markers = {};
let geofencePolygon = null;
let topic_name = "hub/pub";  // MQTT topic to subscribe to for livestock data

/**
 * Function to set a cookie with a specified name, value, and expiration days.
 *
 * @param {string} name - The name of the cookie.
 * @param {any} value - The value to store in the cookie.
 * @param {number} days - The number of days until the cookie expires.
 */
function setCookie(name, value, days) {
  const d = new Date();
  d.setTime(d.getTime() + days * 24 * 60 * 60 * 1000);
  const expires = "expires=" + d.toUTCString();
  document.cookie = name + "=" + JSON.stringify(value) + ";" + expires + ";path=/";
}
```

```
/**
 * Function to get the value of a cookie by its name.
 *
 * @param {string} name - The name of the cookie to retrieve.
 * @returns {any|null} - The value of the cookie or null if not found.
 */
function getCookie(name) {
  const nameEQ = name + "=";
  const ca = document.cookie.split(";");
  for (let i = 0; i < ca.length; i++) {
    let c = ca[i];
    while (c.charAt(0) === " ") c = c.substring(1);
    if (c.indexOf(nameEQ) === 0) {
      return JSON.parse(c.substring(nameEQ.length, c.length));
    }
  }
  return null;
}

/**
 * Function to get the URL of an icon based on the type of animal.
 *
 * @param {string} type - The type of the animal (e.g., cow, sheep).
 * @returns {string} - The URL of the corresponding animal icon.
 */
function getAnimalIcon(type) {
  switch (type.toLowerCase()) {
    case "cow":
      return "https://maps.google.com/mapfiles/ms/icons/blue-dot.png";  // Icon for cows
    case "sheep":
      return "https://maps.google.com/mapfiles/ms/icons/yellow-dot.png";  // Icon for sheep
    case "horse":
      return "https://maps.google.com/mapfiles/ms/icons/red-dot.png";  // Icon for horses
    case "goat":
      return "https://maps.google.com/mapfiles/ms/icons/purple-dot.png";  // Icon for goats
    default:
      return "https://maps.google.com/mapfiles/ms/icons/green-dot.png";  // Default icon for
other animals
  }
}

/**
 * Function to merge new livestock data with existing data stored in cookies.
 *
 * @param {Array} existingData - The existing livestock data from cookies.
 * @param {Array} newData - The new livestock data received from the MQTT topic.
 * @returns {Array} - The merged array of livestock data.
 */
function mergeLivestockData(existingData, newData) {
  const dataMap = {};

  // Add existing data to the map by ID
  existingData.forEach((animal) => {
    dataMap[animal.ID] = animal;
  });

  // Add/Update new data into the map
  newData.forEach((animal) => {
    dataMap[animal.ID] = animal;
```

```
  });

  // Return the merged data as an array
  return Object.values(dataMap);
}

/**
 * Function to check if a given point (latitude and longitude) is inside the geofence.
 *
 * @param {number} lat - Latitude of the point.
 * @param {number} lng - Longitude of the point.
 * @returns {boolean} - True if the point is inside the geofence, false otherwise.
 */
function isPointInGeofence(lat, lng) {
  if (!geofencePolygon) {
    console.log("No geofence polygon defined.");
    return true; // Assume inside geofence if no geofence is set
  }
  const point = new google.maps.LatLng(lat, lng);
  return google.maps.geometry.poly.containsLocation(point, geofencePolygon);
}

/**
 * Function to update map markers based on livestock data and check geofence status.
 *
 * @param {Array} livestock - The array of livestock data.
 */
function updateMarkers(livestock) {
  let outsideGeofence = false;

  livestock.forEach((animal) => {
    const position = new google.maps.LatLng(animal.Lat, animal.Lon);

    // Check if the animal's position is inside the geofence
    if (!isPointInGeofence(animal.Lat, animal.Lon)) {
      outsideGeofence = true;
      alert(`Animal ${animal.ID} is outside the geofence!`);
    }

    if (!markers[animal.ID]) {
      const marker = new google.maps.Marker({
        position: position,
        map: map,
        title: animal.name,
        icon: {
          url: getAnimalIcon(animal.Type), // Assign icon based on the animal type
        },
      });

      const infoWindow = new google.maps.InfoWindow({
        content: `<h3>ID: ${animal.ID}</h3><p>Type: ${animal.Type}</p><p>Location:
(${animal.Lat.toFixed(4)}, ${animal.Lon.toFixed(4)})</p><p>Speed:
${animal.speed.toFixed(2)}</p>`,
      });

      // Add click listener to show info window
      marker.addListener("click", () => {
        infoWindow.open(map, marker);
      });
```

```
      markers[animal.ID] = { marker, infoWindow };  // Store the marker and info window
    } else {
      // Update existing marker position and info window content
      markers[animal.ID].marker.setPosition(position);
      markers[animal.ID].infoWindow.setContent(
        `<h3>${animal.ID}</h3><p>Type: ${animal.Type}</p><p>Location:
(${animal.Lat.toFixed(4)}, ${animal.Lon.toFixed(4)})</p><p>Speed:
${animal.speed.toFixed(2)}</p>`
      );
    }
  });

  // Merge new livestock data with existing data in the cookie
  let existingLivestock = getCookie("livestock") || [];
  const mergedLivestock = mergeLivestockData(existingLivestock, livestock);

  // Update the cookie with the merged livestock data
  setCookie("livestock", mergedLivestock, 7);
}

/**
 * Function to initialize the geofence polygon from the server.
 */
function loadGeofence() {
  fetch("/get-geofence", {
    method: "GET",
    headers: {
      "Content-Type": "application/json",
    },
  })
    .then((response) => {
      if (!response.ok) {
        throw new Error("Network response was not ok");
      }
      return response.json();
    })
    .then((data) => {
      if (data.coordinates) {
        let coordinates = JSON.parse(data.coordinates);

        coordinates = coordinates.map(coord => {
          const [lat, lng] = coord.split(",");
          return { lat: parseFloat(lat), lng: parseFloat(lng) };
        });

        // Create a geofence polygon on the map
        geofencePolygon = new google.maps.Polygon({
          paths: coordinates,
          editable: false,
          draggable: false,
          fillColor: "#FF0000",
          fillOpacity: 0.35,
          strokeColor: "#FF0000",
          strokeOpacity: 0.8,
          strokeWeight: 2,
        });

        geofencePolygon.setMap(map);  // Set the polygon on the map
      } else {
        console.log("No geofence found for this user.");
```

```
      }
    })
    .catch((error) => {
      console.error("Error loading geofence:", error);
    });
}

/**
 * Function to initialize Google Maps and load existing geofence and livestock data.
 */
function initMap() {
  // Initialize the map centered at specified coordinates
  map = new google.maps.Map(document.getElementById("map"), {
    center: { lat: -31.97, lng: 115.81 },
    zoom: 13,
    mapTypeId: google.maps.MapTypeId.HYBRID,
  });

  // Load the geofence polygon if it exists
  loadGeofence();

  // Load livestock positions from the cookie if available
  const savedLivestock = getCookie("livestock");
  if (savedLivestock) {
    updateMarkers(savedLivestock);
  }
}

// MQTT message handling
mqttClient.on("connect", () => {
  console.log("Connected to AWS IoT");
  mqttClient.subscribe(topic_name, { qos: 1 }, (err, granted) => {
    if (err) {
      console.error("Subscription error:", err);
    } else {
      console.log("Subscribed to topic:", granted);
    }
  });
});

// Handle incoming MQTT messages
mqttClient.on("message", (topic, payload) => {
  console.log("Message received on topic:", topic);
  let livestock;

  try {
    livestock = JSON.parse(payload.toString());
  } catch (e) {
    console.error("Failed to parse JSON:", e);
  }

  // Ensure livestock is wrapped as an array for compatibility with updateMarkers
  if (!Array.isArray(livestock)) {
    livestock = [livestock]; // Convert to an array if it's not
  }

  updateMarkers(livestock);  // Update markers based on received livestock data
});
 // Ensure initMap is globally accessible for the Google Maps API callback
window.onload = initMap;
```