NoiseWatcher:
Cloud Based Real Time Noise Monitoring and Reporting System

Group 3:
Sun Sitong (1007132)
Shrinidhi Arul Prakasam (1007007)
Lim Jia Hui (1006924)
Ho Xiaoyang (1007006)

Table of Contents

1. Background and Problem

Noise disturbances have become an increasingly common issue in high-density urban environments, especially in Housing and Development Board (HDB) estates where residents live in close proximity. More than 30,000 neighbour-related noise complaints are submitted to public agencies every year, and over 130 cases escalate to the Community Disputes Resolution Tribunals. Despite the prevalence of these disputes, the current reporting and investigation process remains highly inefficient. Residents are required to manually record audio or video evidence following strict guidelines, while officers often need to conduct in-person investigations with no reliable means of verifying the authenticity of submitted recordings. The lack of objective, tamper-proof noise monitoring not only prolongs resolution times but also contributes to heightened frustration and tension among residents.

2. Current Solutions and Limitations

Current noise-monitoring solutions include consumer tools such as the Apple Watch Noise application, a wearable device that provides real-time measurements of ambient sound levels. Mobile applications like Sound Meter Pro offer short-term visualisations by displaying the most recent 30 seconds of sound data. In addition, professional real-time noise monitoring systems provide live noise level readings from any location with an internet connection. However, these existing solutions remain limited in practical deployment. Professional devices are often bulky and expensive, while consumer apps and wearables typically cannot capture higher sound intensities above 100 dB and lack the capability to support continuous, reliable monitoring for residential dispute contexts. Together, these limitations make current solutions unsuitable for scalable, tamper-proof, and long-term noise logging.

Current Solutions	Cost (S$)	Size	Max. Sound Level (dB)
Apple Watch Noise App	599 (Apple (SG), 2025)	Compact	100
Sound Meter Pro App	4.50 (Google, 2025)	Compact	<= 100, depending on phone's microphone
Class 1 Sound Level Meter (used in professional real-time noise monitoring systems)	~ 2000 - 7000 (RS Components, 2025)	Bulky	140

Our Solution	Cost (S$)	Size	Max. Sound Level (dB)
INMP441 Sensor	2	Compact	120

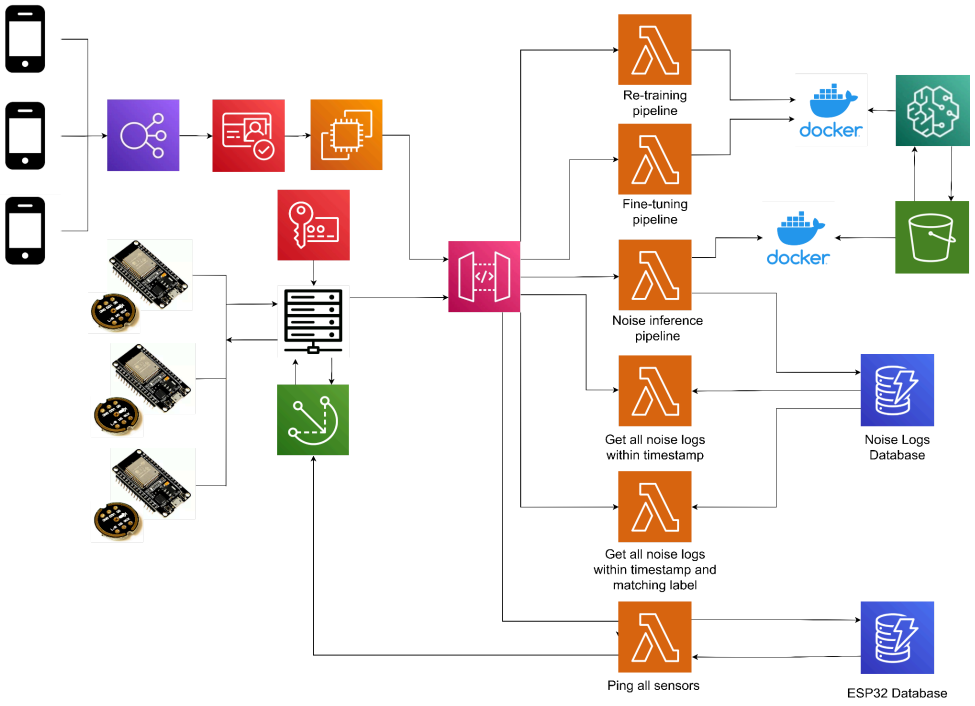
3. Problem Statement

How might we build a scalable system that conveniently automates the verification and submission of evidence of noise complaints, in order to combat inefficient manual processes faced by residents of HDB flats?

4. Our Solution

To address these challenges and foster a safer, more harmonious living environment, we present NoiseWatcher - a smart, scalable, and privacy-preserving noise monitoring and logging system designed specifically for HDB estates. Leveraging IoT sensors, cloud services, and machine learning, NoiseWatcher captures real-time acoustic data, classifies noise events, and securely stores verified logs for residents and administrators. Our solution automates evidence collection, enhances the accuracy of noise classification, and simplifies the dispute resolution process. With features such as sensor health monitoring, noise log verification, and retrainable ML models, NoiseWatcher offers a robust, end-to-end system that overcomes the limitations of existing tools while maintaining affordability, scalability, and user trust.

5. Architecture Overview

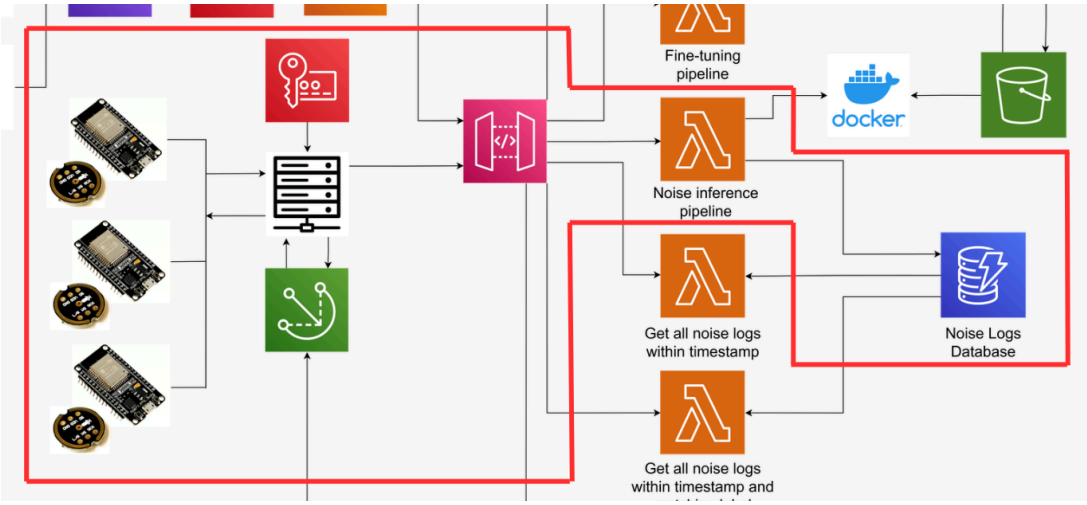


This is how our system would look like if it were to be implemented in real life. Below, we will describe the main features, referring to services that would be utilised if we were to design our system to service 1 million HDB flats.

6. Requirements of Main Features

Our solution is built around four key features that work together to provide accurate, reliable, and scalable noise monitoring for HDB estates.

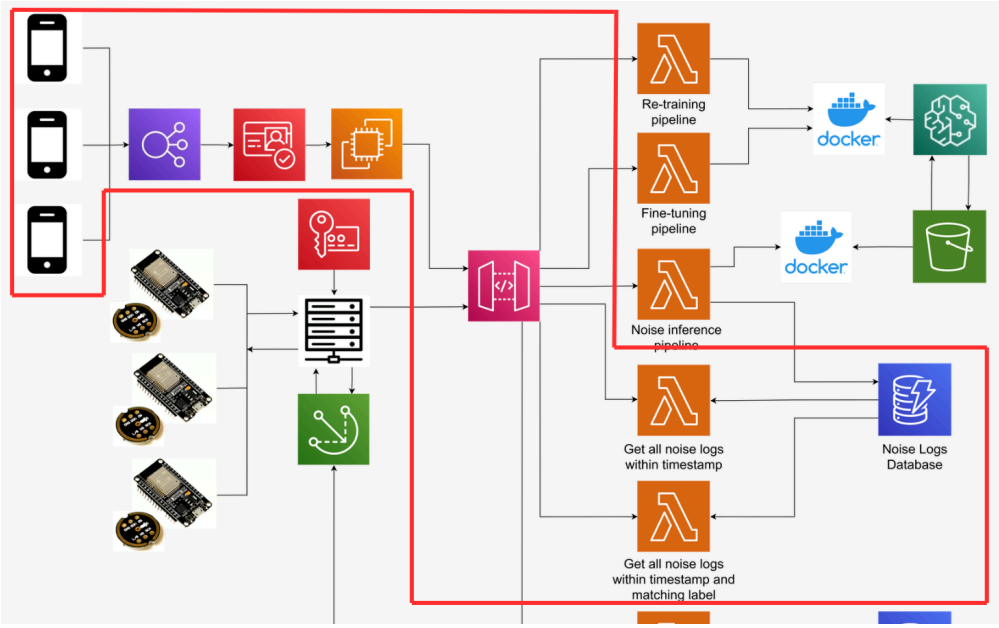
Feature 1: Smart Noise Monitoring



First, smart noise monitoring is achieved through INMP441 sensors installed outside every housing unit, enabling the system to capture acoustic events that cannot be observed through CCTV while preserving resident privacy.

The sensors automatically collect data from 10pm to 8am every day. A local server will be installed for every block, and each server handles the receiving of noise data from all the sensors within that block. The local server is registered as an AWS Greengrass IoT core device, and authenticated with API keys from AWS API Gateway. Since we are using MQTT, devices continue to collect data even if the internet disconnects, and sync when it reconnects. All requests are filtered in each local server, which is crucial since we want to limit the number of HTTP post requests to the noise inference Lambda function. For each filtered noise log that is above a certain threshold, it will be sent to the API gateway hosting all the Lambda functions. The inference of each noise log is done directly in the Lambda function, where the noise logs are assigned the noise classification and stored in the DynamoDB database. This allows for fully automated logging and classification of the noise captured by the sensors, hence reducing the burden on residents to have to manually collect noise data, which can be unverified and inaccurate.

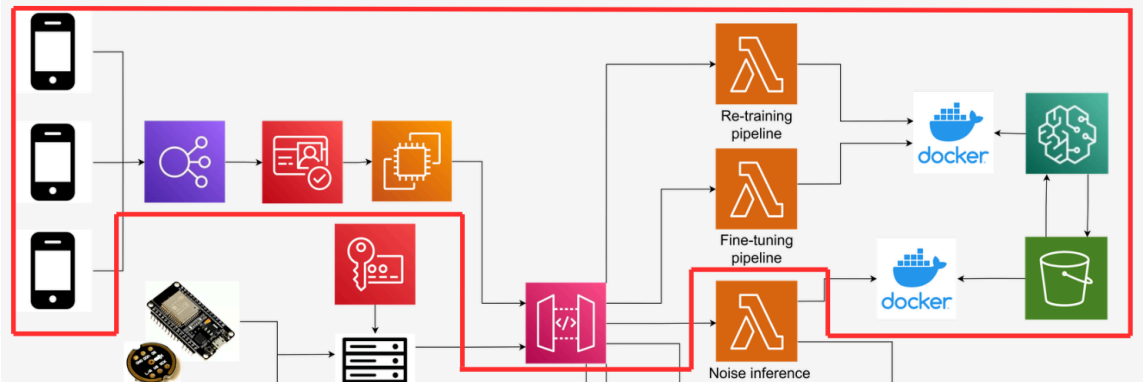
Feature 2: Noise Log Verification



Second, a centralized web and mobile application provides trusted noise log verification, allowing residents and administrators to access authenticated noise records by timestamp, location, and noise classification, when filing or reviewing complaints.

This feature allows the residents to directly access noise logs that were previously inserted in the first feature. The residents can access the web application and click on the options to select a start and end timestamp, as well as the noise label that they are looking for, effectively narrowing down their search. The results returned show the corresponding unit number, timestamps of noise as well as the noise labels, which they can use as verifiable noise log information to conveniently file complaints right in the application itself. This allows for convenient retrieval of up-to-date data, where residents can rest assured that they can access anytime, and anywhere.

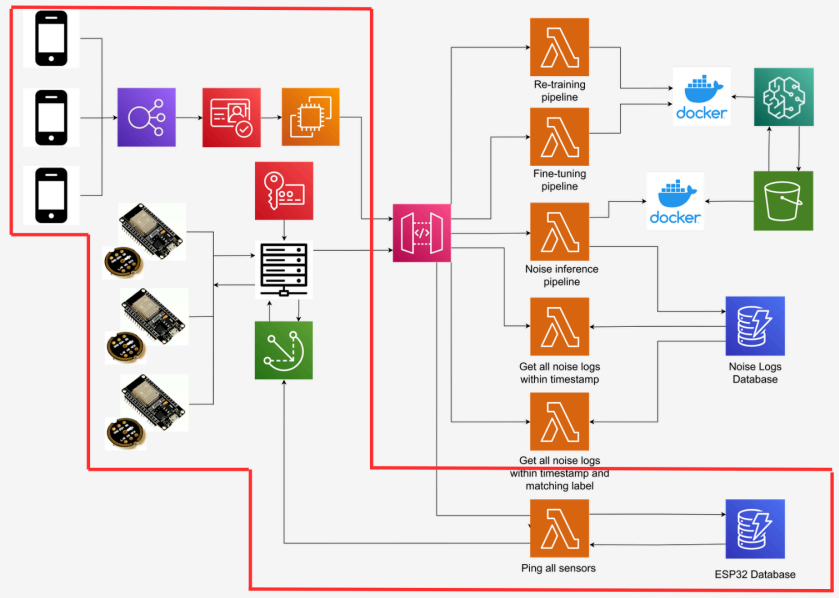
Feature 3: Retraining and Finetuning



Third, we offer retraining and finetuning capabilities for administrators, ensuring the machine learning model can be continually improved to maintain high accuracy in noise classification as new data patterns emerge.

This feature primarily allows the administrators to access our deployed retraining and finetuning pipelines through the web application. Unlike the residents, they will need to do a separate login, which is made possible by AWS services like AWS Cognito. After authentication, they will be able to choose between fine tuning and retraining the model that is stored in the S3 bucket. We utilise technologies such as Docker containers to contain the dependencies (such as Python machine learning libraries) for easy deploying of the training. This allows the manual effort of retraining the machine learning model with new labels to be abstracted away from the administrators.

Feature 4: Maintenance and Monitoring



Lastly, a maintenance and monitoring dashboard supports system reliability by enabling administrators to run ping checks and quickly identify faulty or unresponsive sensors.

This is made possible due to the “Time of Death” feature in the MQTT communication protocol. A database of all the ESP32 sensors are kept in a DynamoDB database, and periodic status updates are conducted. The local server will send any “Time of Death” messages to the Lambda function, where the status of the specific ESP32 will be updated in the database as “offline”. After the ESP32 reconnects to the local server, the status is updated to “online”. When administrators want to conduct checks on all the sensors, they will call the Lambda function in order to get all the statuses in the ESP32 database. This makes monitoring of all ESP32 devices

easy since the administrators of the system can know at a glance which ESP32 sensors might be faulty, and in need of a replacement.

Together, these features form a robust, end-to-end solution for effective noise management in residential environments.

7. IOT section exploration and setup process

As mentioned in previous sections, the IOT network consists of having one microcontroller with a sound sensor in front of each HDB unit. For each HDB block, there will be a local server doing first layer filtering, which will then be sent to AWS cloud. In this section, we will discuss how we implemented this process in our POC in detail.

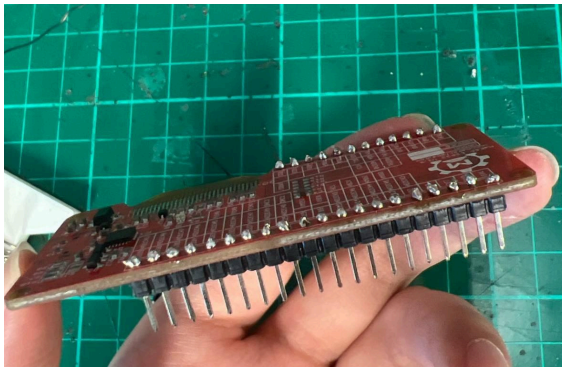
Sensors

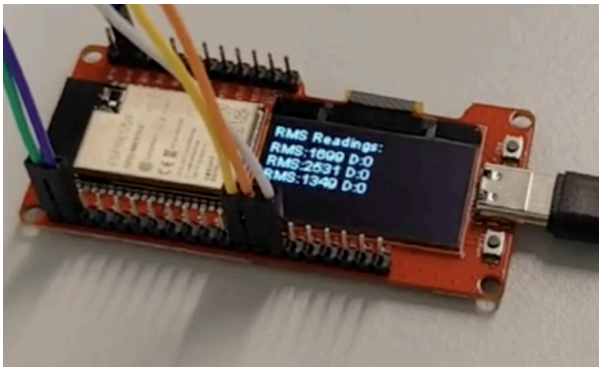
At the start of the project, we chose to use the KY-037 sensor, which is a simple electret condenser microphone module utilizing an analog voltage output. However, when we tried to use it in our real setup, we realized that it was hard to calibrate consistently, primarily due to its low data resolution and low level of noise capturing range. Even after calibration, it only captured noise reliably within a small, immediate range, so we needed to replace it with a new sensor. After research, we found the INMP441, which is a high-fidelity digital MEMS microphone utilizing the I2S protocol. This proved to be a good fit for our project because its noise immunity and high-resolution 24-bit digital sampling enable it to capture acoustic events from a greater distance. Using this digital sensor by following a tutorial video (Owen O'Brien, n.d.), we established reliable noise capture that extends effectively across the crucial monitoring zone between the doors of two HDB units.

The INMP441 sensor, transmitting high-fidelity digital data via the I2S protocol, operates at a high sample rate (typically 44.1 kHz), resulting in a continuous stream of digital values that fluctuate rapidly around zero because the alternating current component of the sound wave has a zero average over time. To analyze this raw data stream for meaningful events, the ESP32 must implement a buffering mechanism, collecting samples within a defined short period, such as the 20-millisecond interval used in our noise logging system, to stabilize the measurement. Within this buffer, the system calculates the Root Mean Square (RMS) value, which effectively solves the problem of the signal averaging to zero by first squaring every sample (making all values positive) before averaging and taking the square root.

This calculated RMS value precisely represents the electrical power or average energy contained within that 20-millisecond sample window, providing a proportional and accurate measurement of the sound wave's loudness or intensity, which is the required metric for reliable acoustic triggering.

In order to clearly separate actual loud noise from the bearable background sound, we decided to use a precise threshold to classify the data. We set this threshold to 70,000 (on the 24-bit digital scale), which serves to effectively separate the sound level into two categories: loud noise or normal background activity. This classification is then recorded in a simple binary form (1 for noise, 0 for background). This specific threshold was chosen after calibration tests confirmed it reliably captures acoustic events within a two-meter range, which is the approximate distance between the doors of two HDB units that we needed to monitor.





The INMP441 has an advantage of high-speed digital communication with the microcontroller because it uses the I2S Protocol. However, when we first tried to integrate the sensor into our system, this expected advantage slowed down our setup process. At first, we did not solder the pins onto the INMP441 sensor; instead, we just loosely let the wires or jumpers touch the metal contacts. This bad connection immediately caused the microcontroller to fail, often returning meaningless error readings like -1. After searching online, we realized that the high-frequency digital transmission of the I2S protocol relies completely on a stable, low-noise electrical connection. Therefore, we went back and correctly soldered the pins onto the sensor board, which immediately resolved the communication errors and resulted in normal data readings. This taught us an important lesson: in any project that involves digital hardware communication, especially at high frequencies, it is always the best practice to solder connections right from the start to ensure signal integrity.

Local Server

After the sensor collects data, it will be communicated with a local server in MQTT protocol (DIYIoT, n.d.). We choose to have a local server instead of directly uploading to the cloud, because we want to ensure that even if the AWS cloud services are down, the data collected will not be lost during the down period. The publisher of the communication will be all the microcontrollers in front of each unit, while the broker and subscriber are the local server.

In this project, we chose to use the RPi as our Proof of Concept local server. The communication system is set up so that every microcontroller publishes data across two separate topics. The first

topic is responsible for the identifier of that specific node, such as "Block 1 Unit 1"; this identifying message is sent sharply at the start of the MQTT connection. The second, high-frequency topic carries the collected data itself, which is packaged as a JSON object. This data includes the precise timestamp in Unix milliseconds and the calculated output value from the sensor. Crucially, the payload also contains a binary indicator that records whether the noise at that exact timestamp is classified as a loud event, based on the specific threshold we discussed earlier in the sensor part.

```
{timestamp:1765337058696, value: 870986, is_noise:1}
{timestamp:1765337058716, value: 870986, is_noise:1}
```

The data points are sent from the microcontroller at a quick 20-millisecond interval. When these points arrive at the RPi, an initial filter logic runs on the RPi to decide what structured data will be sent onward to the cloud. This logic, implemented in a Python script, assumes 30 data points constitute a single frame of an acoustic event. During the continuous data transmission, if the script sees a data point labeled as noise (the binary indicator is '1'), it immediately loads this data point into a local buffer. The logic then dictates that the next 29 subsequent data points will automatically be loaded into this buffer, regardless of their own noise label. If any other timestamp within this 0.6 second collection period is also labeled as noise, the script is designed not to restart a new buffer, assuming that all activity belongs to the same acoustic event frame. Once the buffer is full with the required 30 data points, this whole frame is sent to the cloud and processed by our machine learning system.

The actual sending is handled by another dedicated thread, which runs the Lambda function API call asynchronously. This crucial use of threading ensures that the long, variable latency of the cloud transmission process does not delay the local system's ability to empty the buffer and immediately resume waiting for the next independent acoustic event, keeping the local processing pipeline clear and responsive.

8. Preparation for noise classification model

Next, we had to think about how we wanted to further process the data points that were sent from the Raspberry Pi. We wanted to fully implement the machine learning training and inference for noise classification in our POC. By searching online for numerous pretrained deep learning models, we found that there does not exist a good pretrained model that could suit our noise data that consisted of pure RMS values and timestamps. Online pretrained models mainly focus on classifying noise data in the form of .mp4 or .wav files, which are all uncompressed data. We did not want our system to capture noise data in these data formats, as it takes up space and compromises the privacy of residents. Our ESP32s and relatively cheap INMP441 sensors also cannot possibly store and process data in these formats. We needed to find a computationally lightweight model that was able to process compressed sound data, hence we decided to train our

own machine learning model from scratch. This allows us to have fast training time, as well as fast inference time.

Data collection

After finalising our IoT sensor data collection logic, we move on to collecting the data that is needed for our noise classification. We needed to establish a consistent way of collecting, as well as preprocessing our data so that we can utilize it in our noise classification model, so residents can verify the noise logs on the web application.

We collected 250 seconds of data in total, specifically for ‘shout”, “drill”, and “background” noise. We used a python script to help us collect the data from the Raspberry Pi and format it in json files. The noises are emitted from around 2 metres from the sound sensors during collection, in order to simulate average half-distance between two HDB flat housing units. This is crucial as we want our data to be as accurate as the actual data that would be collected in the HDB flats. We added a script that logged all the raw RMS value data from the INMP441 sensor in our Raspberry Pi, in a neat json file. It captures the timestamp and corresponding RMS value, in 20-millisecond intervals.

Our data collection method is consistent, but the noise that is captured is deliberately done under imperfect conditions (with background noise, overlapping slightly with other types of noise), since we want our model to be robust and to recognise the main features of what makes a sound be classified as a certain label. The model should not be overfit on “perfect” sounds, and there should be variance in the type of “shout”, “drill”, and “background” noises captured. For example, we made sure that the “shout”, “drill”, and “background” audio played was not from the same audios every time.

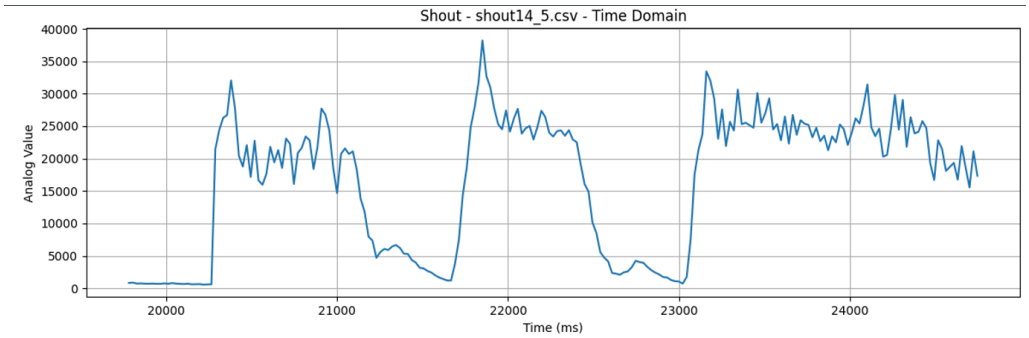
Data preprocessing

In order for the noise classification model to accurately predict the sound, we have to extensively preprocess the data. Since there was background noise in the data, as well as in real life, there is a need to isolate distinct frequency signals so that the model can be trained to identify these signals, instead of trying to train off pure RMS values, which would compromise training efficiency and accuracy. The following details the steps that we took to transform our raw RMS values to frequency values, using the Fourier Transform that is popularly used in signal processing.

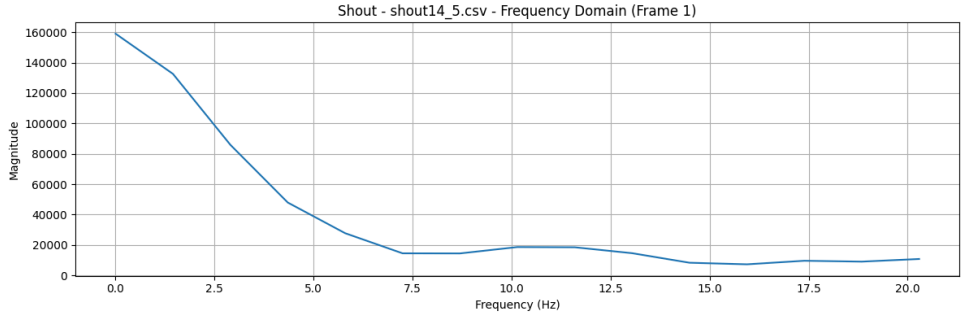
Data preprocessing is crucial as we want our data to be clean so that there would be no issues (example, null values) in our training of the noise classification model. We have to take note that there were initial overlaps in the json files as some values were being overwritten due to high sampling rates. We substitute each erroneous value (that was over the maximum possible value

processed by the INMP441 sensor) with the previous value of the datapoint. In order to not get a string of values that were all erroneous, and thus after preprocessing would get a constant set of the same value, it is also possible to introduce some random variance (so that the frequency value after the fourier transform will not be 0). This makes the system more resilient to erroneous data. We might also consider classifying this type of noise (where the RMS values are all over the maximum possible value) as an "error", which would trigger the operators to replace this sensor. In our proof-of-concept, we omitted this step as our sensor was working well and we did not have any erroneous data.

After collecting the raw RMS values for each 20-millisecond interval, we will then group up the values into frames of around 30 datapoints each (which will be around 0.6 seconds for each frame). In the actual system, the data will be grouped sequentially, by strict continuous intervals. However, for our data preprocessing, we wanted to overlap the frames so that we can multiply the amount of data that we have to be trained and classified. Hence, we overlapped frames by a factor of 80%, which we found to have the most significant improvement in our test accuracy. This is an example of what a frame would look like for a noise with a "shout" noise label:

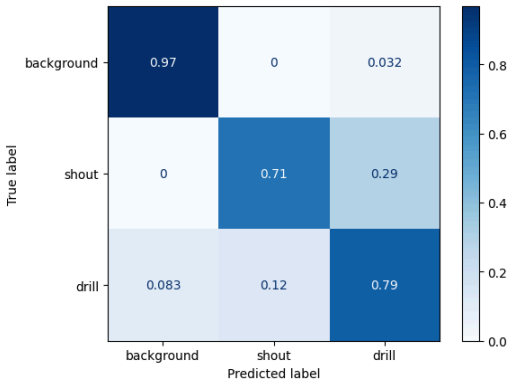


After overlapping the frames, we used a python library to perform Fourier transforms for all of our data frames. This is what the above frame for "shout" would look like after the transform:



We split all data frames into a split of 85% for the train set, 7% for the validation set, and 8% for the test set. In total, there are 1324 frames for training, 156 frames for validation, and 79 frames for testing. The results are then fed into our noise classification model for training and

classification. We used the scikit-learn machine learning library for its configurable models. We used a Random Forest model and a Voting Classifier layer to obtain the final predictions, and to optimise the model weights to shift it so that the final predictions are better.



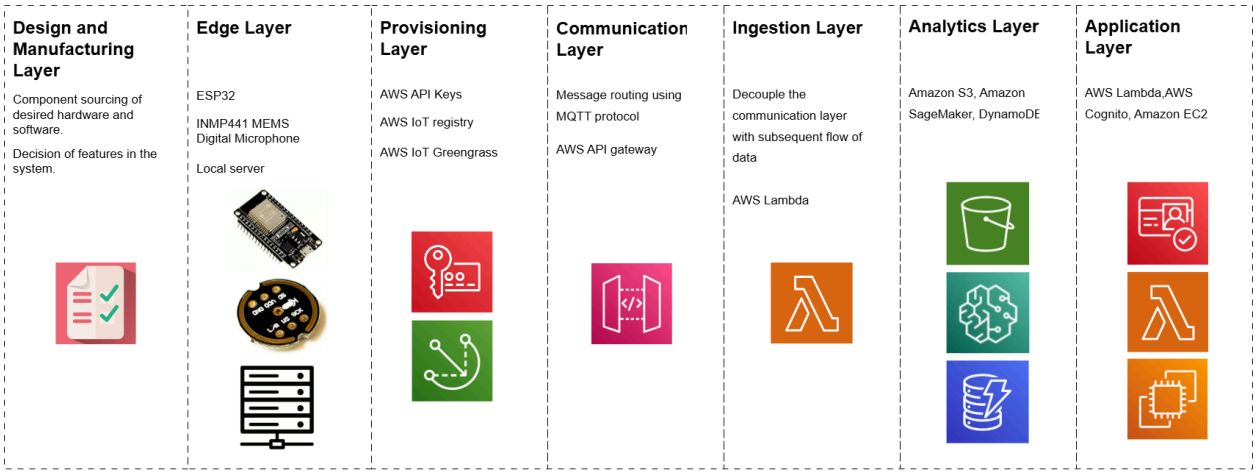

```
Test Classification Report:
              precision    recall  f1-score   support

  background       0.94      0.97      0.95        31
       shout       0.85      0.71      0.77        24
       drill       0.70      0.79      0.75        24
...
    accuracy                           0.84        79
   macro avg       0.83      0.82      0.82        79
weighted avg       0.84      0.84      0.83        79
```

The results for the confusion matrix (denoting how accurate our model is) shows that our model is able to achieve above a 0.7 F1 score for all our classes, achieving good predictions.

Though the training is done locally, this can also be deployed on AWS SageMaker. We have deployed our fine tuning pipeline on AWS, which is able to fine tune our model and update the model in the S3 bucket with the latest updated model, depending on whether it is able to achieve an overall test accuracy of above 0.85. If it is not, the model weights are discarded. This is to protect our model from being updated with samples that are not clean or do not perform well on unseen data.

9. Cloud section exploration and setup process

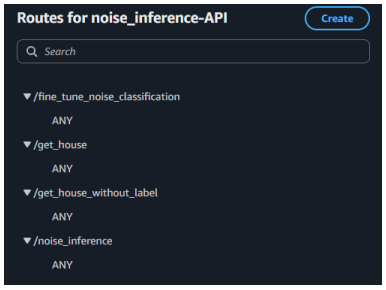


After our IoT setup, we have to consider how we will link the data that is collected and preprocessed to our AWS cloud services. These are the IoT and cloud services that we will

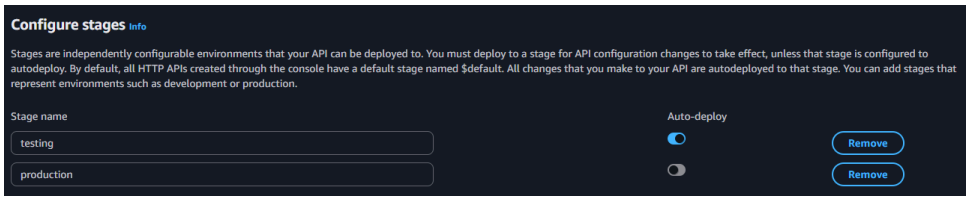
integrate into our NoiseWatcher system that are classified into the corresponding well-architectured layers. We will focus on explaining how we deployed the communication layer, ingestion layer, and analytics layer, as these are the specific services that we deployed for our POC.

Communication Layer

As mentioned in the section about the local server, we use MQTT communication protocol to transmit data amongst the hardware components. We will then need to use AWS API Gateway that is able to connect the data from the local server to the cloud, in order to capture the data from the sound sensors of approximately 1 million HDB units in Singapore. It is important for us to establish this layer so that we can make it easier to track where each pipeline, feature or function requires which sort of data. For our POC, we implemented an AWS API Gateway that facilitated communication between 4 major AWS Lambda functions and our hardware and other AWS services.



These are the API routes that we deployed for our POC. All API routes for our POC are grouped in one singular API. Ideally, for the real system, we should group the API routes used by the ESP sensors (noise_inference) under one API, and the API routes used by the web application itself (fine_tune_noise_classification, get_house, and get_house_without_label) under another API.

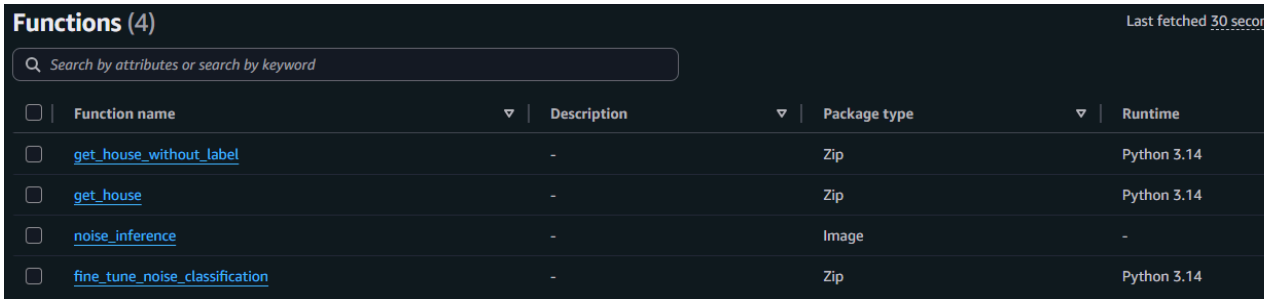


This is because we should be able to configure environments for each stage of the deployment of the system, such as for development and for production. This is to ensure that all our APIs for the ESP32s and the APIs for the web application are able to be tested safely, with its own separate authorizations (with JWT tokens, and permissions for accessing other AWS services), before deploying on a large scale.

Ingestion Layer

This layer is essential to decouple the communication layer with other layers, such as the analytics layer. It is crucial for us to clean the data coming from the hardware sources, so that it can be stored and analysed in a systematic, ordered fashion in the analytics layer. We primarily used AWS lambda functions for the ingestion layer. Some benefits of using lambda functions in our system are:

1. They are serverless functions. They are self-contained, as each lambda function only serves the purpose of ingesting one type of data, for one singular purpose.
2. They are stateless and ephemeral, meaning that they do not maintain any data between executions, apart from the code and dependencies required to execute the function. This serves our application and makes the data that is processed from each ESP32 independent from others.
3. It is possible to create AWS lambda functions either by authoring from scratch, or from a container image. This makes it flexible since we have functions that require certain python packages, as well as functions that are very lightweight.
4. The pricing of these functions depend on how long they are used for, and not how long they have been deployed for. Compared to other storage based systems, this works for our system, since our ESP32 devices will only be automated to collect data from 10pm to 8am daily.



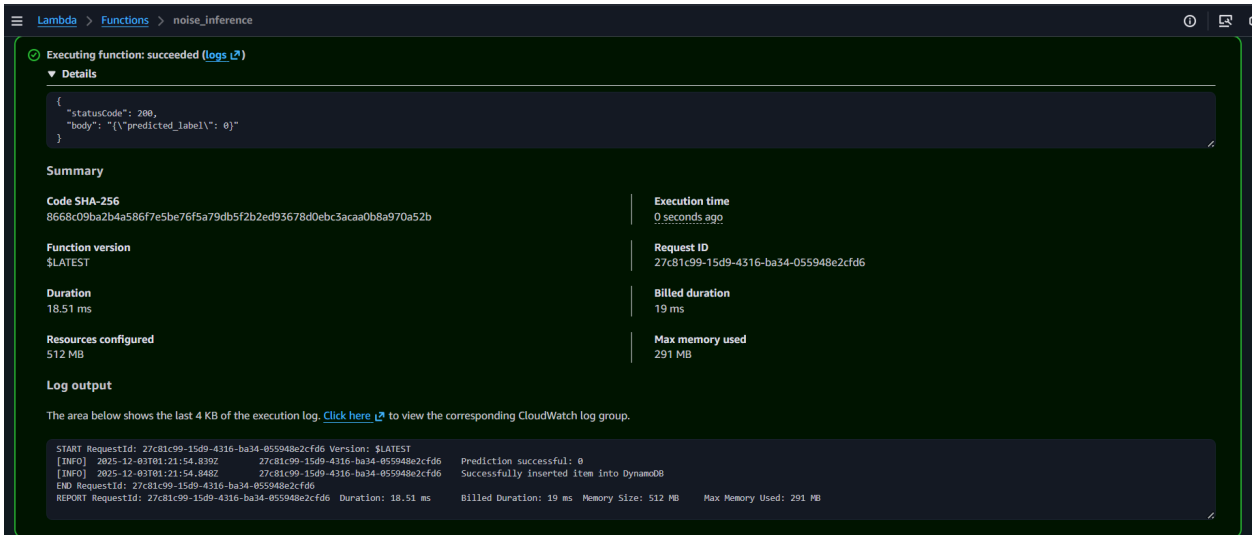
These are the lambda functions that we have deployed for our POC, that are connected to the same API in the AWS API Gateway. 3 of the lambda functions that are deployed as zip packages are the ones that we have created from scratch. These are the functions that are lightweight and are only required to get data/input data/fetch data from other services, like AWS IoT Core, and AWS Sagemaker. They are triggered by ESP32 sensors calling its respective endpoints, and by the administrators when they want to fine tune the noise classification model.

However, we have one lambda function that is deployed as an image, which is the noise_inference function. This is because we wanted the smart noise monitoring feature to be

ultra-fast, and hence wanted the ingestion of the data from the ESP32s and the subsequent classification to be extremely fast. We had two options for the noise_inference function:

1. Copy what fine_tune_noise_classification function does, and create the function from scratch, connecting the lambda function to AWS SageMaker. Then, after noise classification, we need to fetch the result of classification from SageMaker again. This results in unnecessary I/O operations.
2. Create the function from scratch, and place all the dependencies in a dedicated Lambda Layer. This was not feasible as the python package that we needed for the noise inference, scikit-learn, could not fit in the size limit. This was also unfeasible because Lambda Layers do not specify what type of packages are contained within it in a declarative fashion. It is tedious to track which packages are contained within the Layer.
3. Deploy an image with the corresponding files and python package dependencies for the lightweight noise classification machine learning model in an ECR repository. This requires us to specify the dependencies in a Dockerfile. This method is superior since it uses declarative architecture principles.

We used method 3 to deploy the noise_inference lambda function. It is easy to track different versions of the container image, in the case of updates. Moreover, our lambda function is set to track the latest version of the container image, using the ":latest" tag. When additional dependencies are added to the lambda function in the future, it can be declared in the Dockerfile, and pushed to the ECR repository, where it does not limit the size of the packages. This is the most scalable method.



Furthermore, the average time taken to execute the noise_inference lambda function is about 19ms, even though our function was written in Python. This is extremely fast as compared to other large pretrained deep learning models. By using declarative architecture principles, and by

removing the I/O by decoupling this lambda function from other services, we are able to achieve an extremely fast execution time, which greatly reduces the time taken for the data to travel from our hardware, to the ingestion layer, to the analytics layer, and to the subsequent application layer where our users will fetch the data required for their noise complaint report.

Analytics Layer

We used 3 main services to build the analytics layer. We implemented them in the processes detailed below:

1. **Amazon S3**

Firstly, we use S3 buckets to store our noise classification model weights. These weights are produced after training or fine tuning the machine learning model. For our POC, this model is stored as a .joblib file, and takes up around 3MB of storage space. If our model were to be trained with more than the current 250 seconds of data, the model weights would also increase in size. We need a scalable solution to store these model weights.

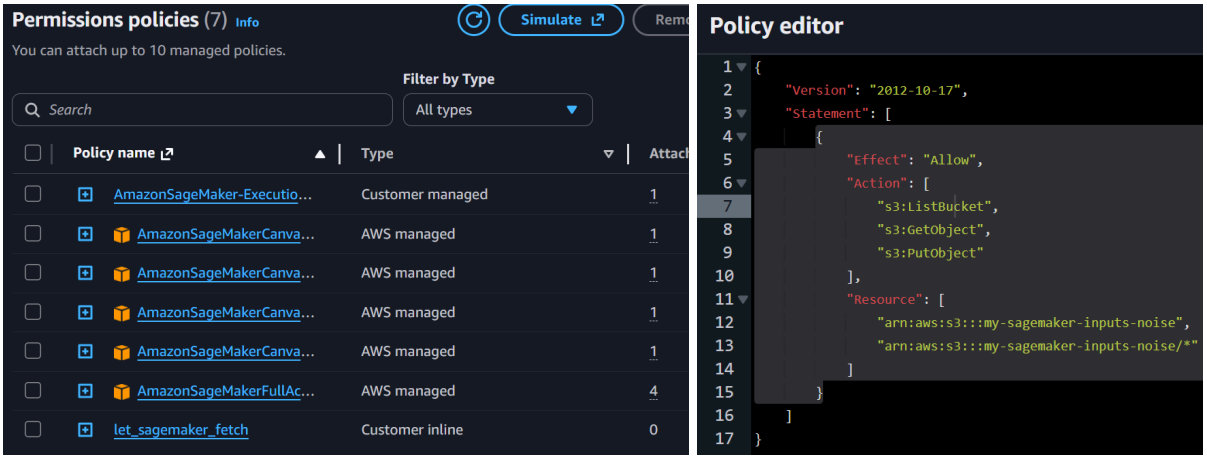
Similarly, we use S3 buckets to store the noise data required in csv format for our POC. This data is required for model retraining purposes, where we would need our entire dataset as well as the new data that administrators can add, in order to train the model to predict noise labels that were not previously in the dataset. S3 buckets can store data of any size, and can thus store huge amounts of audio data if required. Alternatively, for our data to take up less space in the S3 bucket, we can also store it in more compressed forms, such as parquet files instead of csv files.

Our S3 bucket is designed to enable Amazon SageMaker to fetch data from it, which will be explained in the next section.

2. **Amazon SageMaker**

We use this for our retraining and finetuning features. We used Amazon Sagemaker, specifically Amazon Sagemaker AI for our POC, in order to deploy our training pipeline. By creating a notebook instance, configurations for the specific Roles, estimators, and inputs can be specified. Using this service for our training pipelines is optimal, as compared to using other services, as it includes the option for us to choose instance types, which includes support for numerous GPUs and CPUs required. Our system only requires minimal instance types such as ml.t3.medium, as our RandomForest machine learning model is extremely lightweight and fast to train. Our system does not require the training to be completed extremely quickly (as compared to the noise classification inference which is done in the lambda function), and can be done in the background, which is why we do not need to spend money on instances with extremely strong compute power.

We created a notebook instance, which is actually containerised. Training code can either be stored in the SageMaker notebook instance or the S3 bucket. We stored it in the SageMaker notebook itself for easier editing of the code for our POC, but it should be stored in the S3 bucket as it is cheaper and should not require administrators of the system to edit the code at all.



Roles can be specified in the notebook instance as well. As shown above, we created a role for our Sagemaker notebook, using the AWS IAM service. This role allowed our SageMaker notebook instance to fetch from and place data in our S3 bucket which stores all the model weights. We also want to allow our lambda function (called by the fine_tune_noise_classification endpoint) to be able to trigger the sagemaker notebook instance. We created a separate execution role for lambda function with the necessary permissions to be able to interact with our SageMaker notebook instances.

After our lambda function triggers the sagemaker notebook instance, the training job will be run in the background, and a response status of 200 will be returned for the successful initiation of the training job. The SageMaker notebook instance will be run as a container image, with all the dependencies specified in a requirements.txt file. After the successful training of the model, if the model reaches an overall accuracy of over 85%, the weights will automatically replace the old model weights that are stored in the S3 bucket.

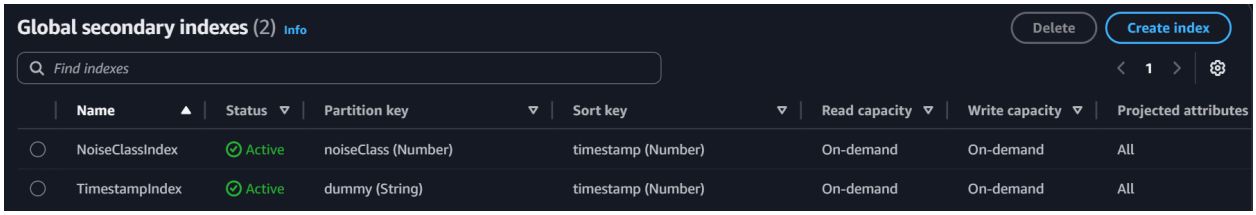
This allows the tedious work of training and fine tuning of the model to be completely abstracted away from the administrators, reducing configuration drift and manpower workload. We only require them to provide the csv files for the raw RMS values, noise label, and the timestamps for any ESP32 sensor.

3. DynamoDB

DynamoDB was necessary for our system, to store noise logs and the statuses of our ESP32 devices. DynamoDB offers unlimited storage and is extremely low-latency, which fits our use case of having possibly millions of noise logs stored in the database. In our POC, we focused on deploying the table used to store noise logs.

We structured our DynamoDB table using a Composite Primary Key (Partition Key + Sort Key). This structure provides the uniqueness required for our data entities while optimizing physical storage. The partition key drives DynamoDB's internal hashing mechanism, effectively distributing our workload across partitions to prevent storing all our data in one singular partition. Following best practices, the table infrastructure was defined in a .yaml file and deployed using AWS CloudFormation. This allowed our AWS Lambda functions to immediately interact with the table for data ingestion.

A critical design challenge was retrieving logs filtered by both label (noiseClass) and time (timestamp). In DynamoDB, a Scan operation reads the entire table, which is computationally expensive and slow. To avoid this, we engineered Global Secondary Indexes (GSI). These indexes allowed us to utilize the Query operation instead of a Scan. By performing an exact match on the index's partition key (noiseClass) and a range comparison on the sort key (timestamp), we reduced our database reads from O(N) (total items) to O(k) (matching items), significantly lowering costs and latency.



These are the secondary indexes we have implemented for our application. We specified two indexes:

1. NoiseClassIndex, which is used to fetch the noise logs/frames according to the noiseClass and the start and end timestamps specified by the user. Our get_house lambda endpoint fetches noise logs using NoiseClassIndex.
2. TimestampIndex, which is used to fetch the noise logs/frames according to the start and end timestamps specified by the user. This is used when users want to fetch the noise log without knowing what the noise can be classified as. We used a dummy string for the partition key, as this enables us to use the Query operation (which is faster than a Scan operation) to let DynamoDB handle the search according to the timestamp attribute, which is the sort key. This makes the operation extremely fast even though users could

query up to thousands of noise logs at once. We have tested that the maximum lambda execution time to get the noise logs (filtered by start timestamps and end timestamps that encompassed all our stored noise logs) is around 300ms, with a Python runtime. To further reduce this time, we can restrict our users to query start and end timestamps within an hour of each other, but otherwise, this operation is still quite fast.

10. User interface exploration and setup process

The web application allows users to make noise disturbance reports within a few minutes, and allows administrators to fine-tune or re-train the noise classification model, as well as verify the functionality of the sensors for maintenance.

The web application is a Next.js (App Router) application built with TypeScript and React. It uses Tailwind CSS for styling, client-side components for interactivity, and Next.js API routes to integrate with the AWS Lambda functions. The app implements form validation, state management, and session storage for multi-step workflows, such as when the user fills in the reporting form.

We have deployed our application on an EC2 instance. The application is viewable here. For our POC, we have done a basic deployment where we simulate how users can navigate the pages.

For your reference, when you navigate to the administrator login page, the admin email is: admin@gmail.com and the password is: admin123. Try querying for noise logs between 1 December to 3 December, as this was when data was recorded for our presentation.

To extend this to what would be applied in real life, when managing passwords for users, the passwords would be hashed and stored in a secure cloud database, and JWT tokens would be employed to help validate user sessions. Additional validation checks such as CAPTCHA can be implemented to prevent bots from entering our web application, to reduce the amount of requests made to the lambda function endpoints or to prevent fraudulent logins.

Noise Disturbance Report

To report noise disturbances, the user will fill in a noise disturbance form (Appendix, Figure 1). They will key in their address and unit number, and select the start and end time of the noise disturbance down to the specific minute, as well as the description of the noise disturbance. The options that we provide for the description of the noise are “Shouting” and “Drilling”, to align with the noise classes that we trained our noise classification model on. If the two categories do not match what the user has heard, the user can select “Other”, and key in their own description of the noise disturbance. After filling in the form, the user will click the “Find Matches” button

to find the matching noise records recorded by the sensors that correspond with the time period and description of noise indicated by the user.

If the user selects one of the noise description categories that we have defined (Appendix, Figure 2), matching noise records will be retrieved using the get_house Lambda function, which returns all records that correspond to the specified noise class and fall within the start and end timestamps provided by the user (Appendix, Figure 3). If the user chooses to key in their own description of the noise disturbance (Appendix, Figure 4), matching noise records will be retrieved using the get_house_without_label Lambda function, which returns all records that fall within the start and end timestamps provided by the user, regardless of the noise class (Appendix, Figure 5). The user will be able to scroll through each categorised group of noise records and view the timestamped frames to verify if they match with the noise disturbance that the user experienced. A confidence score is also calculated for each noise class at each housing unit, by taking a ratio of the number of frames recorded for that particular noise class at that particular housing unit to the total number of frames recorded across all noise classes at all relevant housing units. This allows users to make a more informed assessment of the records. After the user has identified the record that best matches their complaint, they can click the "Select Records" button to make their selection.

The user will verify their identity by keying in their NRIC/FIN, full name, and contact number (Appendix, Figure 6). This ensures that users can be held accountable for their reports and are able to be contacted by the authorities for additional information or updates. The user can then click the "Submit Complaint" button to officially submit their report.

A confirmation page will be displayed to confirm that the complaint has been successfully submitted (Appendix, Figure 7). The user will be able to view the details of their report, including the type of noise detected, the location of the suspected offending housing unit, the time range of the noise disturbance, and the confidence score. They will also get a reference ID for their report so that they can keep track of the investigation process.

Administrative Page

When a user first accesses the admin page (Appendix, Figure 8), they are greeted with a login screen that restricts access to authorised administrators only. This authentication layer ensures that only permitted users can interact with model training pipelines and sensor management functions. The user enters an admin email and password and clicks the login button. If the credentials are incorrect, an error message is shown immediately. Once authenticated, the user is granted access to the main admin interface, which connects them to the backend services responsible for model management and system monitoring.

After logging in, the user arrives at the admin dashboard, where three main sections are available: Fine-tuning (Appendix, Figure 9), Re-Training (Appendix, Figure 10), and Maintenance (Appendix, Figure 11). These sections are presented as tab-like buttons at the top of the page. Switching between them does not reload the page, allowing the user to seamlessly interact with different backend pipelines, such as training workflows or sensor health checks, from a single interface.

In the Fine-tuning section, the user uploads newly labelled data to incrementally improve the deployed noise classification model. The process begins by selecting the sensor associated with the data, which allows the system to correctly link the uploaded data to the corresponding ESP32 device. The user then selects an existing label, such as “shouting” or “drilling,” or defines a custom label if required. After selecting a CSV file containing labelled noise data, the user initiates fine-tuning by clicking the “Start Fine-tuning” button. This action sends the CSV data through an API endpoint to the fine-tuning pipeline, which runs inside a containerised environment. The pipeline processes the data, updates the model parameters, and prepares the refined model for redeployment into the noise inference pipeline. During this process, the interface displays a processing state and provides feedback once the request is successfully completed or if an error occurs.

The Re-Training section follows a similar interaction pattern but serves a different functional purpose. Instead of incrementally updating an existing model, re-training is used to rebuild the model using a larger or updated dataset. The user again selects a sensor, chooses or creates a label, and uploads a CSV file. When the “Start Re-training” button is clicked, the data is sent to the re-training pipeline, which runs independently of the live inference system. This separation ensures that real-time noise classification remains uninterrupted while a new model is trained. Upon completion, the updated model can be redeployed to replace the existing one, improving overall system accuracy.

The Maintenance section supports system reliability by allowing administrators to monitor sensor connectivity and operational status. Each ESP32 sensor is displayed with a real-time status indicator, such as idle, pinging, online, or offline. When a user pings a sensor, a backend function is triggered to check the device’s availability and network responsiveness. The results of these checks are reflected in the interface and recorded in the DynamoDB database, enabling persistent tracking of sensor health. The “Ping All” function allows administrators to quickly assess the status of the entire sensor network, while the reset option clears all statuses to provide a clean monitoring state.

11. Further enhancements

Innovation

Our system is innovative because we process and classify loud sounds using only the sensor's electrical power measurement, meaning we never have to record audio files like MP4s like other solutions, which is a big win for people's privacy. This method allows our detection logic to run very quickly directly in a simple serverless function (like Lambda), avoiding the need for expensive, high-power GPU inference machines like other solutions. This simple, low-cost, serverless approach is easy for any administrator to deploy at a large scale, resulting in a much lower total cost than typical surveillance solutions.

Scalability

Our solution is designed to be highly scalable. Firstly, although we have currently only trained our machine learning system to recognize common events like shouting and drilling, we have established a dedicated retraining and fine-tuning pipeline that makes it easy to add many other types of acoustic labels to the system over time.

Secondly, our solution is highly deployable because the entire infrastructure is defined using AWS CloudFormation, enabling all resources such as DynamoDB tables, IAM permissions, API Gateway configurations, and Lambda functions to be specified in declarative .yaml files. This allows the full system architecture to be automatically recreated in any AWS region with a single deployment command, eliminating the need for manual setup.

Lastly, the overall cost of our system is financially manageable. Based on our detailed cost analysis for the first year of setup for one HDB block, the total spending is estimated at USD 2,768. Crucially, this cost will significantly decrease in consecutive years because the large, one-time setup fee (which includes the CAPEX for hardware and initial labor) will be removed, making the long-term operational costs very low.

	Formula	Value
Calculation duration	N/A	1 year
No. of Units	N/A	108
One time hardware setup fee (CAPEX)	(cost of MCU + cost acoustic sensor + other hardware) * No. of units	($3.5+$2+$8) * 108 = $ 1458
One year operational fee (OPEX)	((power per unit in watts * No. of units)/1000) x hours in a year * fee rate	((0.5W * 108)/1000

		W/kW) * 8760h * 0.3/kWh = $141
Lambda function	calculated by AWS fee calculator	$69.02
Local server cost	server hardware + network access point + setup labour	$350 +$150 + $600 = $ 1100

Feasibility

To further extend our feasibility, although we use the ESP32 as the microcontroller, the INMP441 as the sensor, and MQTT over Wi-Fi as the transmission protocol, and the RPi as the local server for our Proof of Concept, we still have alternative selections for them that are lower cost, offer higher reliability, and promise significantly less power consumption.

We can replace the sensor and microcontroller combination with the INMP552 and the ESP32-C3. The ESP32-C3 is a newer chip that is much more energy efficient for running simple sensor tasks. For example, while the original ESP32 draws around 5 milliamps in deep sleep mode, the ESP32-C3 draws only about 43 microamps. This dramatic reduction in power consumption means the sensor nodes could stay on battery power much longer. Furthermore, we can switch the transmission protocol from high-power Wi-Fi to the LoRaWAN protocol, since its design is based on low-power, long-range communication. This change would allow the sensor nodes to run for years on a single small battery, completely solving the problem of providing power lines to every unit.

12. Evaluation and further improvement against plan

Overall, our actual presented Proof of Concept aligned quite well with our plan. However, due to time and resource limitations, we did not fully implement Feature 4: maintenance and monitoring. Additionally, it would be better to create a dedicated backend for our user application to interact with AWS storage, instead of just calling an API endpoint in the frontend alone. To further improve our machine learning model, it is critical to collect more data from various sources and backgrounds. For example, noise sources from different directions and distances, several noises occurring at the same time, as our current model still needs improvement on accuracy.

A question was raised during our presentation session: If the noise classification model makes a mistake, how can an investigator verify the true type of noise, especially since recording only the RMS loudness value may not be enough evidence, requiring a delicate balance between accuracy and privacy?

Since our core system only records the loudness numbers and the classification, and never the actual audio file, verifying past misclassified events is impossible. To balance this and still improve accuracy, we can propose an on-demand Verification Mode. If a specific sensor node starts sending many wrong alerts, an administrator can temporarily switch it into this mode. In Verification Mode, the sensor is allowed to temporarily record a very short audio clip (e.g., 5 seconds) only when an alert happens. The investigator can then download this new, temporary clip, check the sound to see what the model missed, use that knowledge to fix the model, and then immediately delete the audio file and switch the sensor back to the normal (no-recording) privacy mode. This allows us to fix mistakes and improve the model without ever keeping private audio permanently.

13. Summary

Developing NoiseWatcher was a significant learning experience. Our initial technical decisions, such as using the simple KY-037 sensor, and not soldering components, quickly proved inadequate, teaching us to follow scope and standard of hardware projects. We learned the value of Edge Computing, using the RPi for local filtering was essential to manage the high data volume and prevent costly, unnecessary cloud calls. We learnt how to engineer scalable cloud solutions that were able to preprocess data from various IoT devices, so that we could ingest it into cloud services that were neatly decoupled into its separate functions. We were able to optimize our system so that it could perform at a high level, and could theoretically meet the needs of about a million HDB flats in Singapore. This allowed us to cleanly fulfil the requirements of our system at a high level, without compromising the latency of the system. Finally, we were able to find an innovative balance between respecting the rights of the residents by maintaining their privacy, as well as providing accurate data and noise classifications for our system. Overall, the project reinforced the necessity of agile design thinking, where real-world deployment challenges drive both technical solutions and ethical policies.

References

DIYIoT. (n.d.). *Microcontroller to Raspberry Pi WiFi MQTT communication*. DIYIoT. Retrieved from https://diyi0t.com/microcontroller-to-raspberry-pi-wifi-mqtt-communication/

Owen O'Brien. (n.d.). *INMP441 Digital Microphone Wiring Guide for ESP32* [Video]. YouTube. https://www.youtube.com/watch?v=UkJIMCtsypo

Apple (SG). (2025). *Apple Watch Series 11*. Apple (SG). Retrieved from https://www.apple.com/sg/shop/buy-watch/apple-watch

Google. (2025). *Sound Meter Pro*. Google. Retrieved from https://play.google.com/store/apps/details?id=kr.aboy.sound&hl=en_SG

RS Components. (2025). *Sound Level Meters*. RS Components. Retrieved from https://sg.rs-online.com/web/c/test-measurement/environmental-test-measurement/sound-level-meters/?sortBy=price&sortType=ASC&selectedNavigation=attributes.Accuracy_Class

Appendix

Our code can be found on this GitHub repository. The code folder structure is as follows, and README.md files can be found in each folder:

Folder name	Purpose
app	Code for the UI of the Noise Complaint System (Typescript). All backend functions are linked to AWS services. To run this, do: 1. `npm install` 2. `npm run dev`
aws_sagemaker	Code for the AWS Sagemaker model finetuning instance (Python).
lambda	Code for all four deployed lambda functions (Python).
noise_prediction	Code and raw data for local training of the noise classification model (Python). You can run all cells in `train.ipynb` to inspect how training and evaluation is performed.
rpi	Code for script to be transferred to Raspberry Pi for MQTT (Python).
sensors	Code for ESP32s to obtain data from sound sensors (C++).

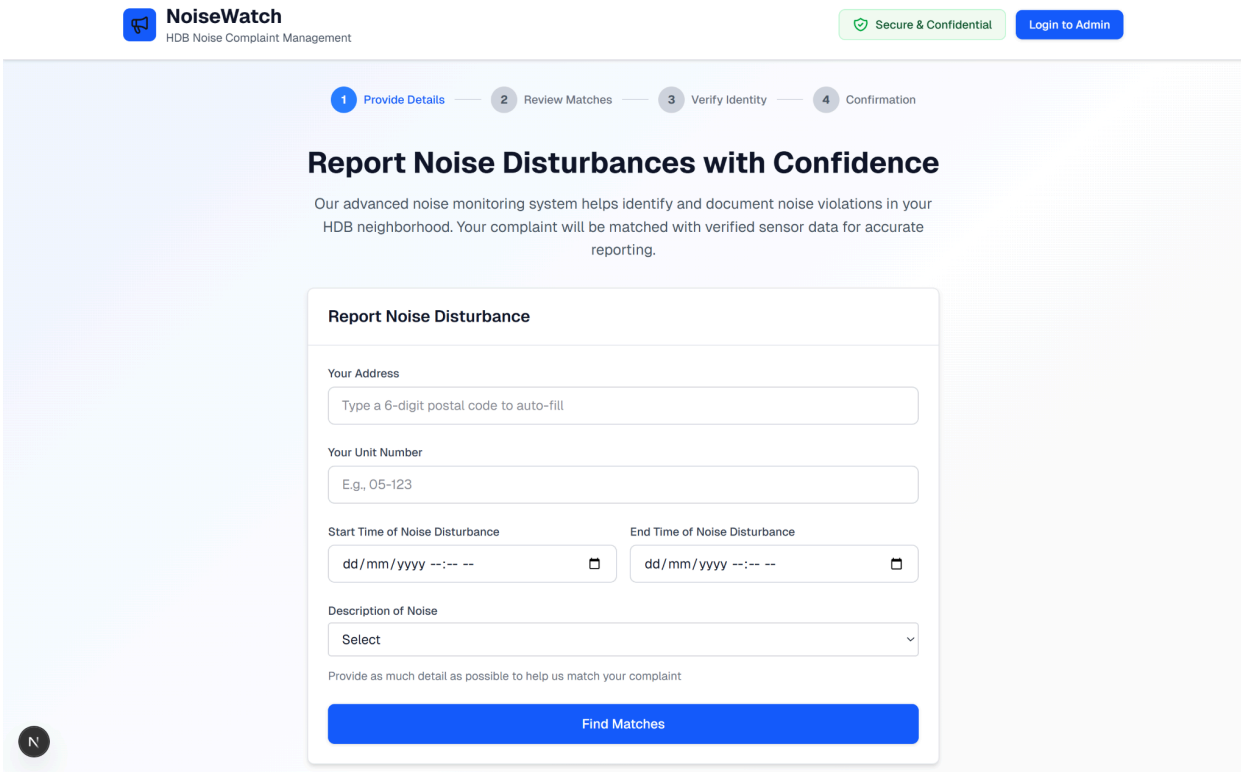


Figure 1. Noise Disturbance Form

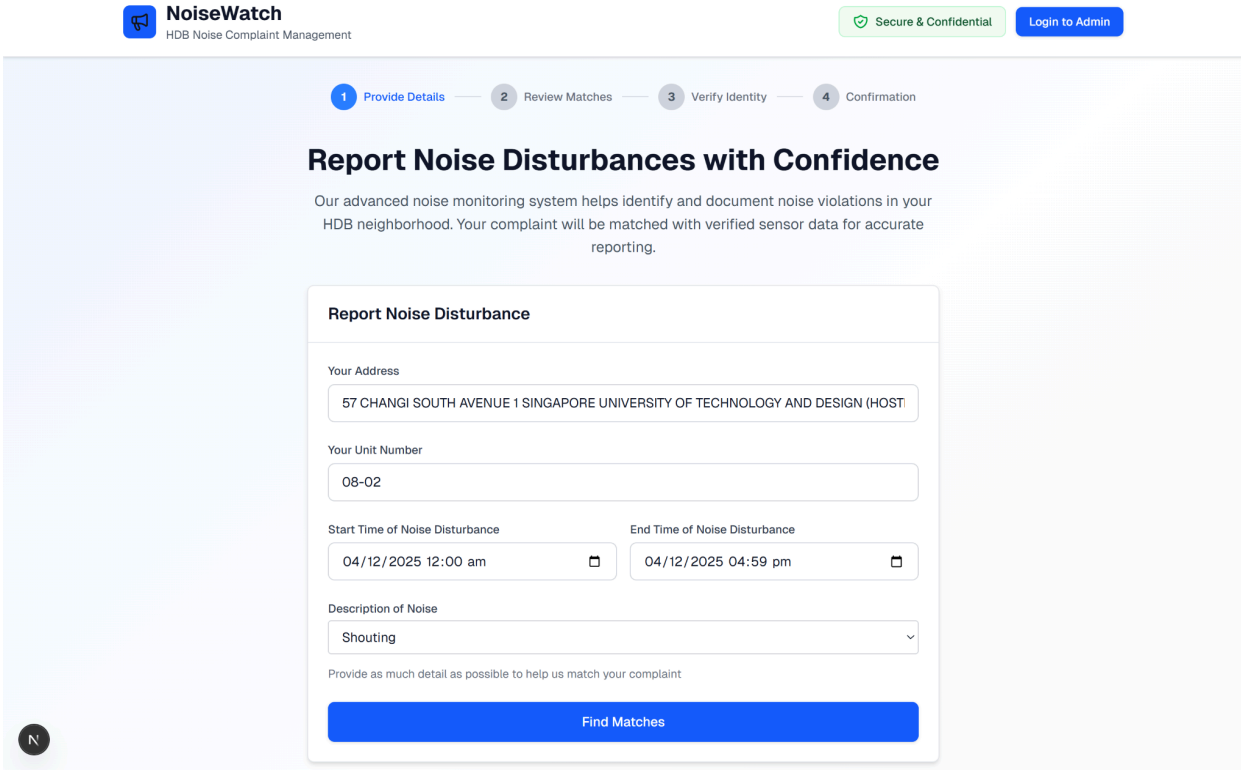


Figure 2. Noise Disturbance Form (Shouting)

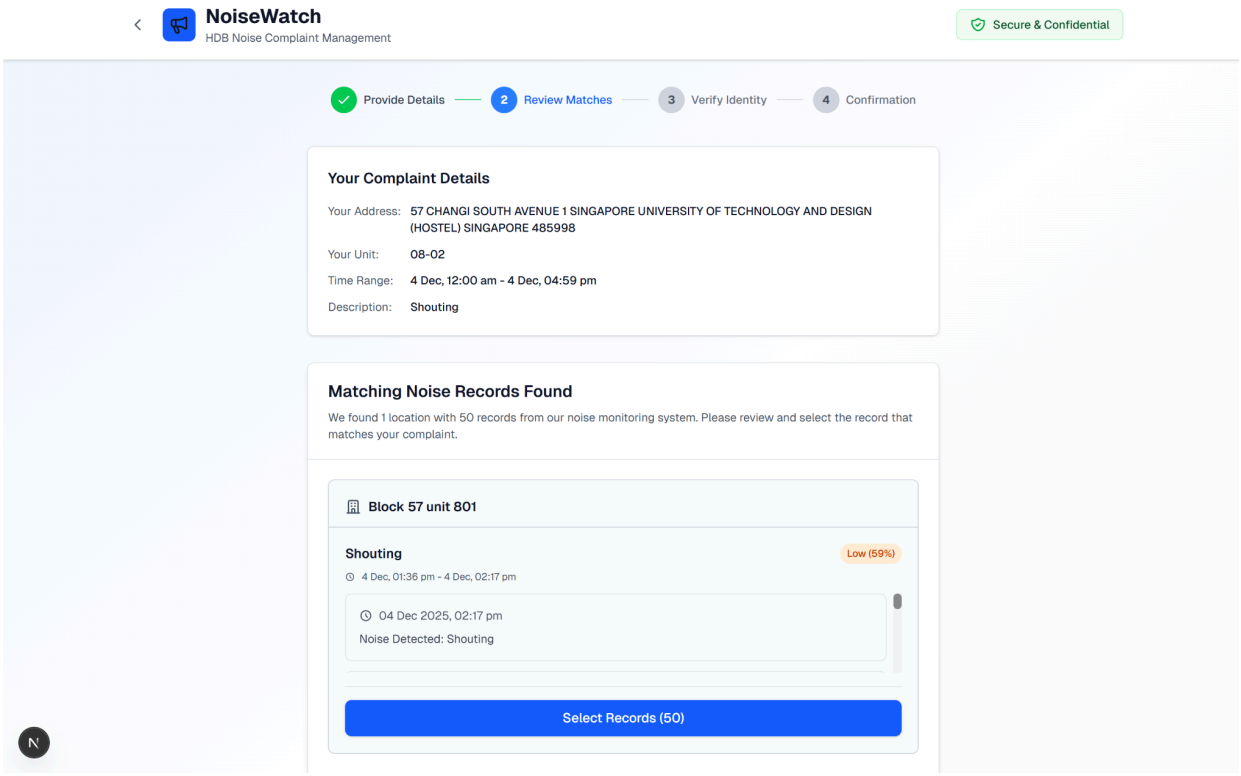


Figure 3. Matching Noise Records (Shouting)

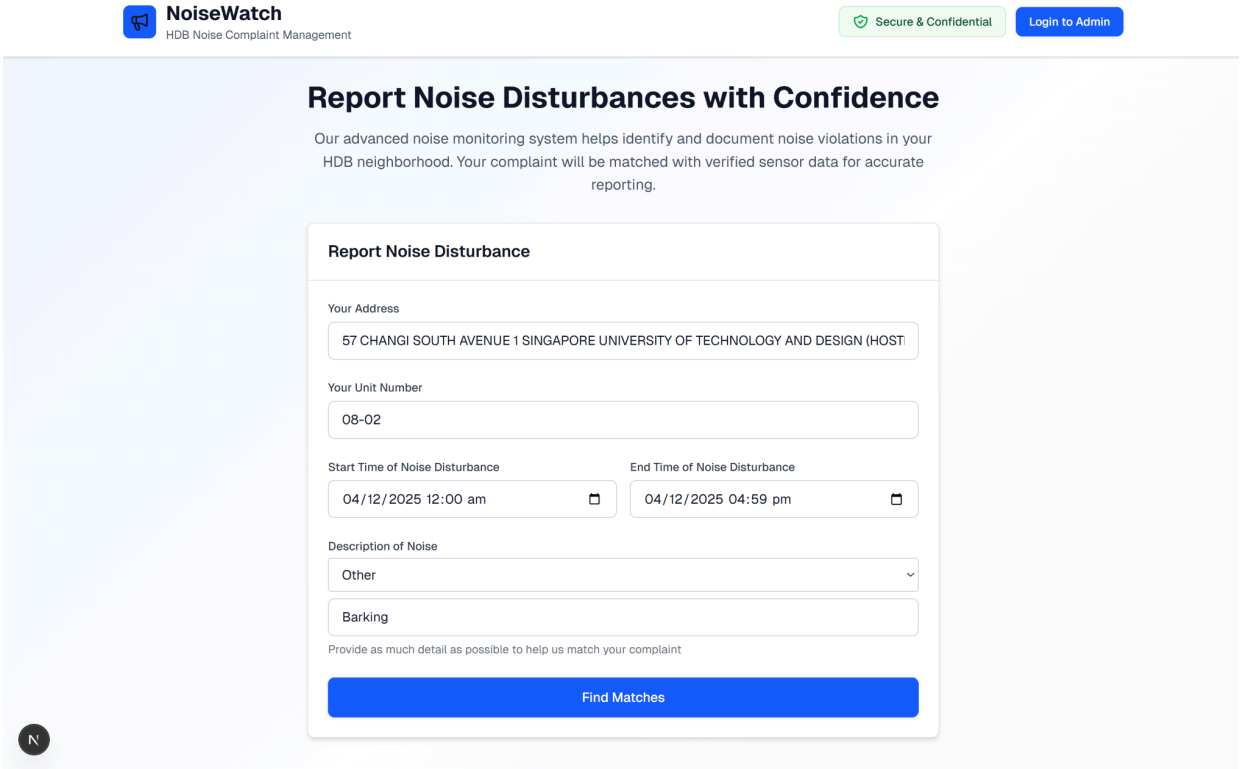


Figure 4. Noise Disturbance Form (Other)

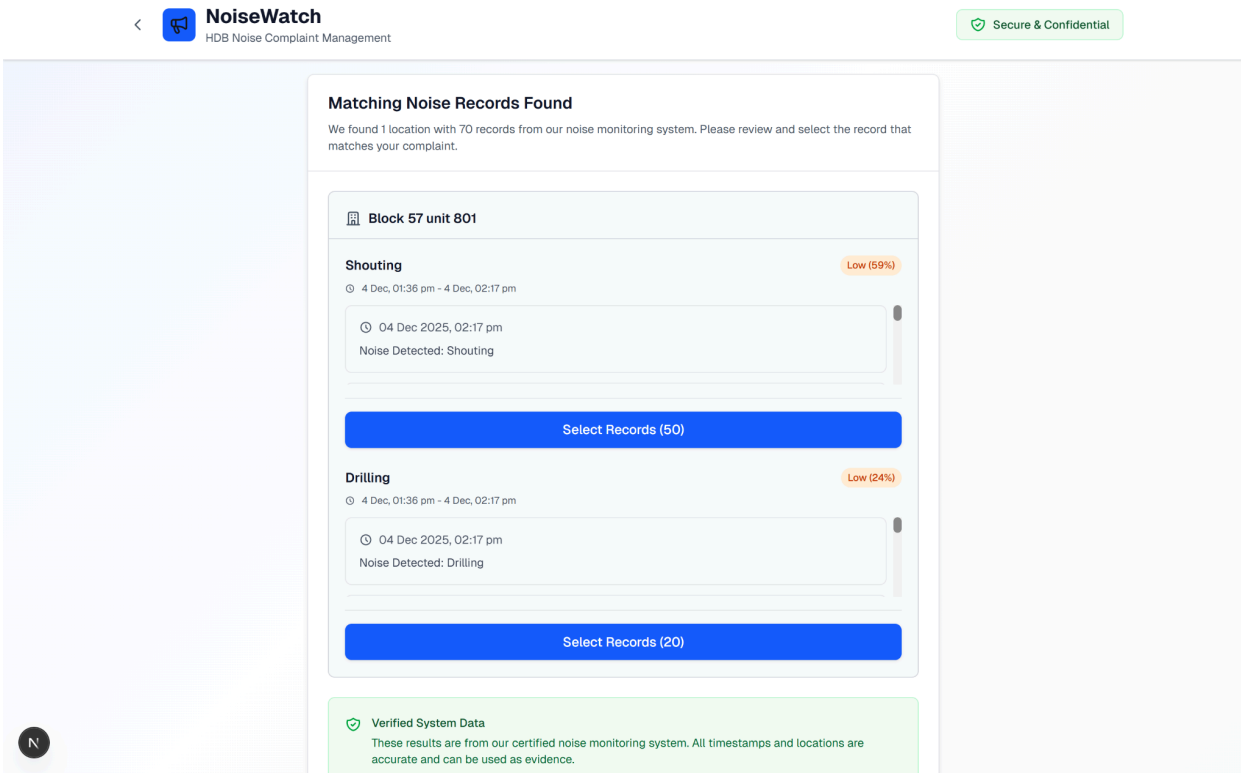


Figure 5. Matching Noise Records (Other)

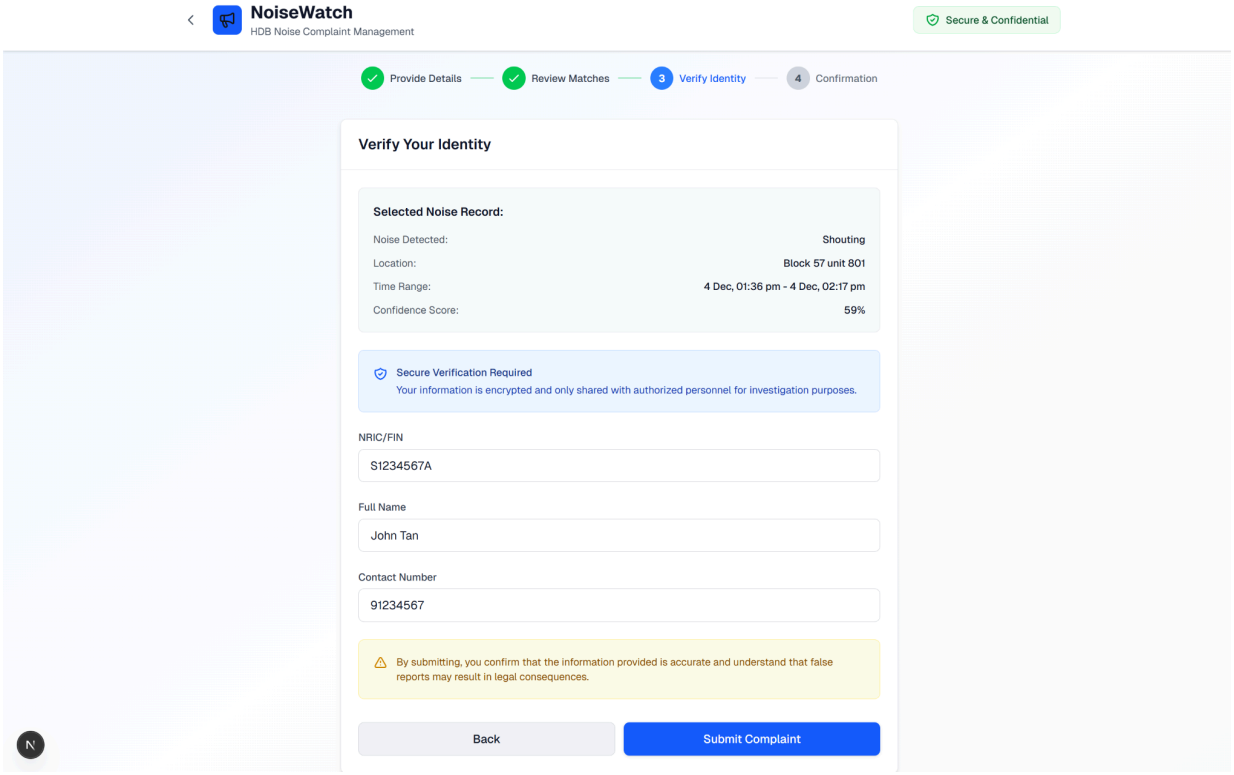


Figure 6. Identity Verification

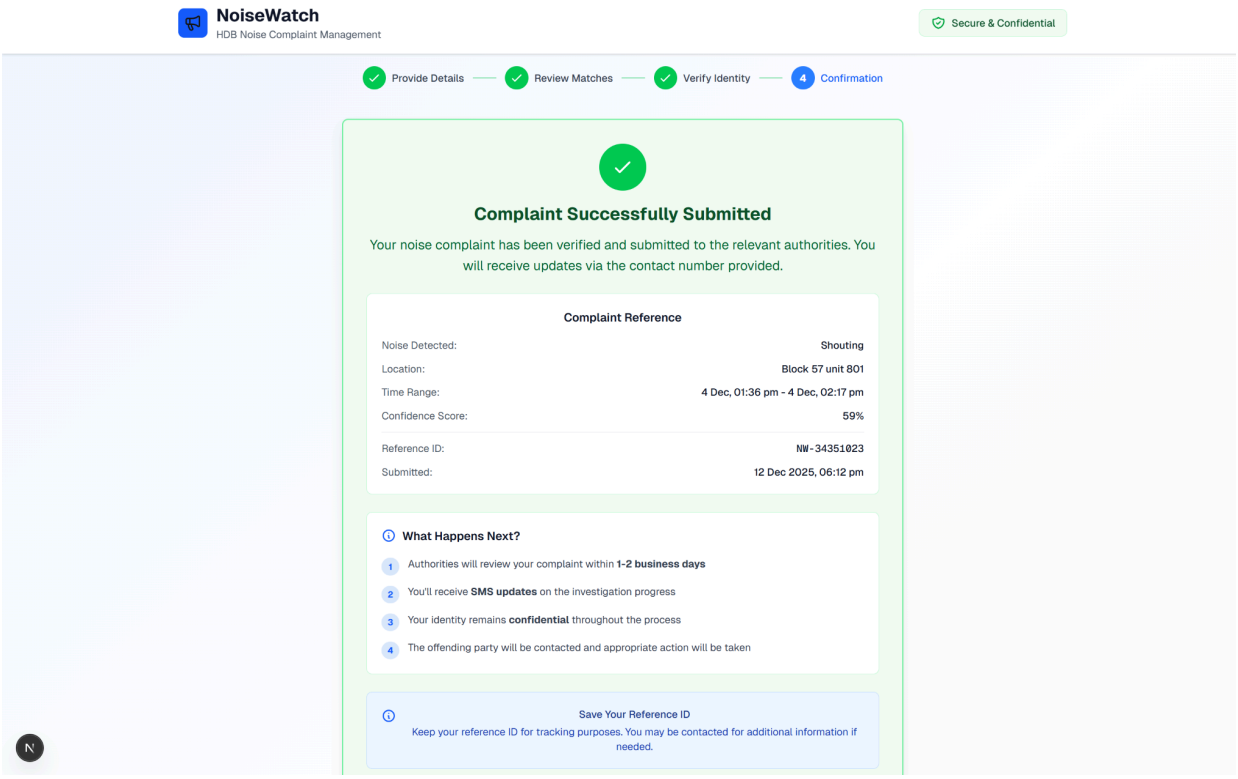


Figure 7. Confirmation

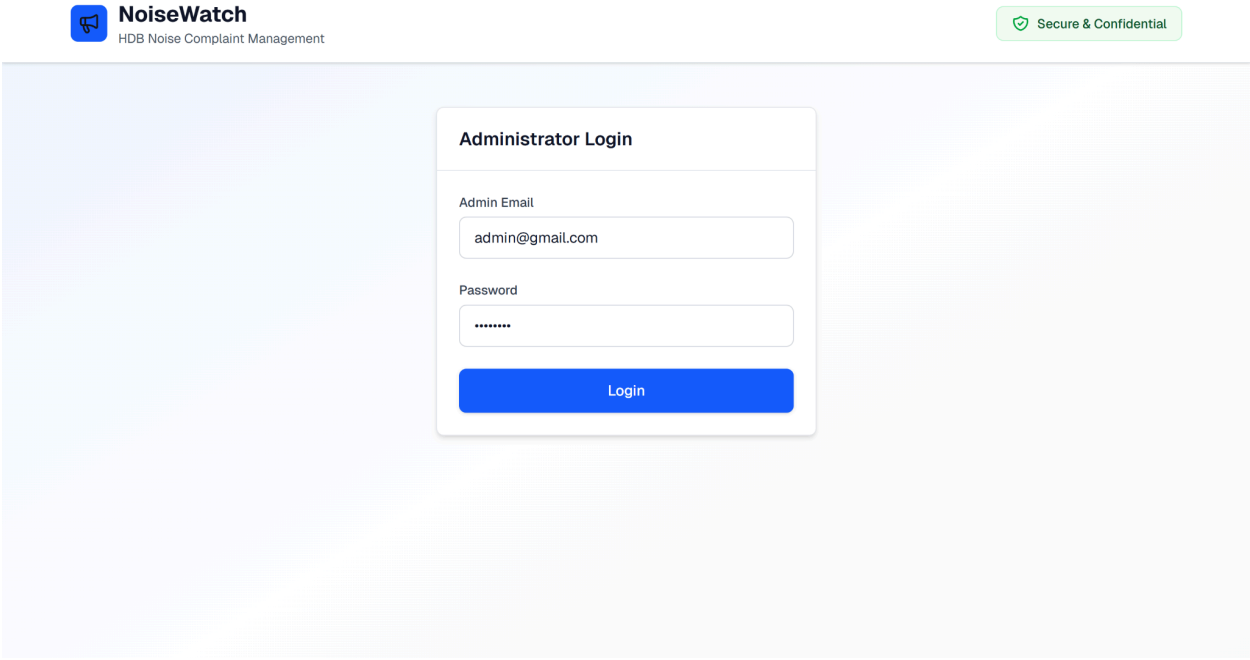


Figure 8. Administrator Login

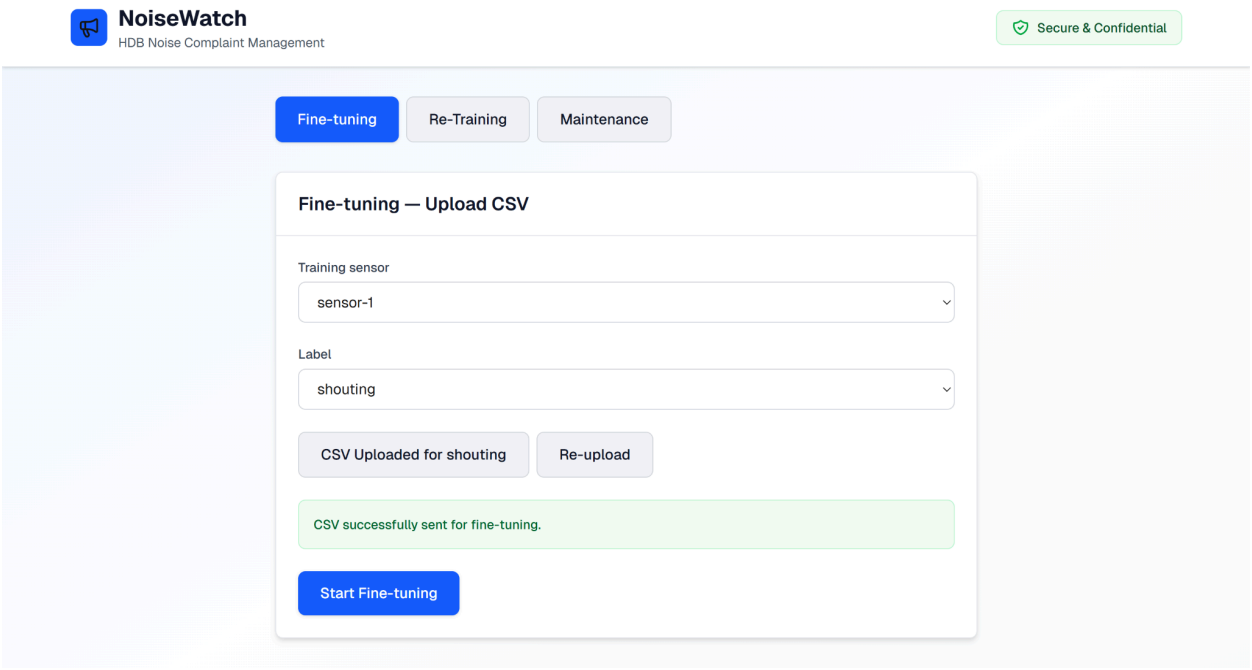


Figure 9. Fine-Tuning

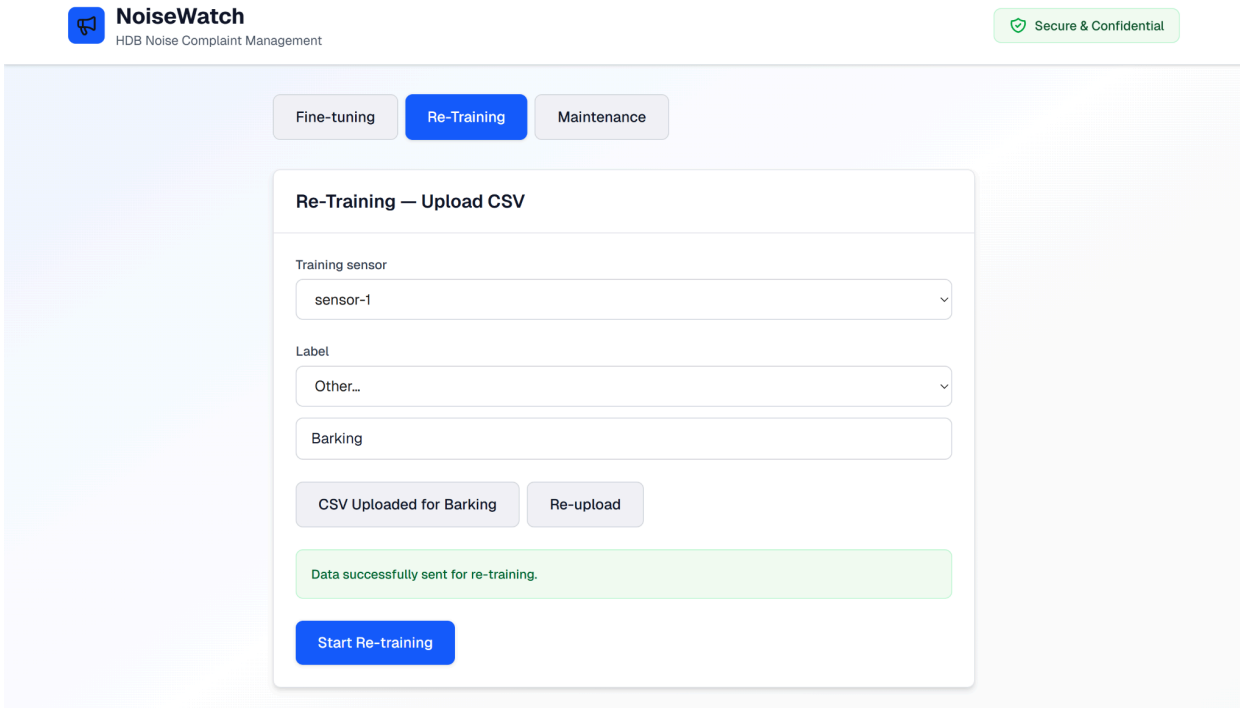


Figure 10. Re-Training

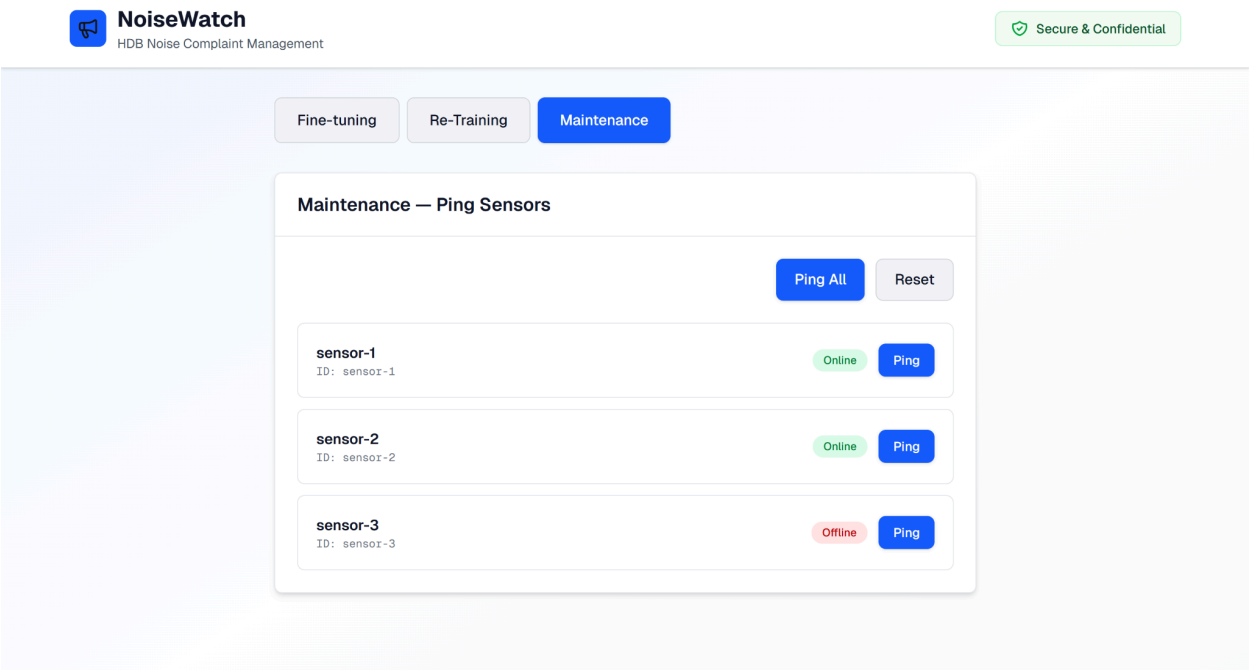


Figure 11. Maintenance

Personal Reflections

Shrinidhi Arul Prakasam - 1007007

Through this project, I was responsible for designing and implementing the Administrative Page, which served as the primary control interface for managing model training pipelines and monitoring sensor health within the system. Implementing the authentication mechanism reinforced the importance of access control in cloud-based IoT systems, as only authorised administrators are allowed to trigger sensitive backend operations such as fine-tuning, re-training, and sensor availability checks. This directly reflected course discussions on secure IoT architectures and the role of application-layer controls in protecting cloud resources.

Developing the admin dashboard allowed me to apply core architectural concepts taught throughout the course. The separation of the interface into Fine-tuning, Re-Training, and Maintenance sections mirrors the separation of concerns in the backend architecture, where inference, training, and device monitoring operate as independent pipelines. Implementing tab-based navigation without page reloads helped me understand how frontend applications act as orchestration layers that asynchronously interact with serverless functions and containerised services, a concept introduced during the cloud services and serverless computing segments of the course.

In the Fine-tuning section, I implemented functionality that enables administrators to associate uploaded CSV data with a specific ESP32 sensor and noise label before sending it to a containerised fine-tuning pipeline. This helped me better understand data provenance and traceability in IoT–ML systems, as sensor identity and label information must be preserved throughout the training lifecycle. It also reinforced lessons from the container foundations and Kubernetes topics, where workload isolation and reproducibility are critical for reliable model updates.

The Re-Training section further strengthened my understanding of cloud scalability and service availability. By designing re-training as a pipeline that runs independently from the live inference system, I applied the course principle of isolating compute-intensive tasks to avoid disrupting real-time services. This made the trade-offs between system accuracy, computational cost, and uptime much clearer than in theoretical examples, and demonstrated how cloud infrastructure supports parallel workloads at scale.

The Maintenance section directly applied IoT monitoring and debugging concepts taught in the course. Implementing sensor ping functionality required an understanding of device connectivity, backend health checks, and persistent state storage using DynamoDB. Recording sensor status in a cloud database highlighted the importance of observability and historical tracking in distributed IoT systems, particularly when managing multiple edge devices. This experience aligned closely

with course outcomes related to systematic testing, debugging, and identifying bottlenecks in IoT architectures.

Overall, this course significantly improved my ability to design and implement end-to-end cloud-based IoT systems. Beyond individual technical skills, I learned how to integrate embedded devices, IoT protocols, cloud services, containerised workloads, and user interfaces into a cohesive and scalable architecture. The Administrative Page, in particular, served as a practical demonstration of how theoretical concepts from the course such as edge-to-cloud integration, service isolation, scalability, and system monitoring translate into real-world implementations.

Lim Jia Hui - 1006924

Through this project, I was able to see how the concepts introduced in class were relevant in a real-world context. My key areas of learning included the use of the MQTT (Message Queuing Telemetry Transport) protocol, the limitations of different hardware sensors, the integration of multiple AWS cloud services into a single system, and the analysis of trade-offs when designing a user-friendly interface. Rather than learning these concepts in isolation, this project demonstrated how they interact within a complete end-to-end system.

In our project, the MQTT protocol was used to facilitate communication between the noise sensors acting as publishers, and the Raspberry Pi acting as both a broker and a subscriber. Through this implementation, I learnt how a single Raspberry Pi device can simultaneously serve as both a broker and a subscriber, and how this setup simplifies the overall system architecture and allows for edge computing, where data can be processed locally before being forwarded to the cloud. Processing the data locally on the Raspberry Pi reduced latency, minimised unnecessary data transmission, and improved system resilience in cases of network instability. This allowed me to understand how edge computing can improve efficiency and responsiveness while also reducing cloud costs, which reinforced how intelligent distribution of computation across edge and cloud layers is a critical design consideration in real-world IoT systems.

This project also highlighted the importance of sensor selection and the limitations that come with different hardware choices. Beyond cost considerations, I learnt that it was important to take into account other factors, such as data resolution and noise capture range for sound sensors, as they could significantly affect the system's performance depending on the use case. I learnt that hardware limitations can impose strict constraints on what a system is capable of, regardless of how well the software is designed. Different sensors are optimised for different use cases, and selection involves balancing a multitude of factors such as accuracy, range, cost, and system complexity. This experience helped me understand that effective system design starts with choosing appropriate hardware, rather than attempting to compensate for hardware shortcomings through software alone.

Another major learning outcome from this project was understanding how multiple AWS services work together to form a complete cloud-based pipeline. AWS IoT Greengrass enabled edge computing on the Raspberry Pi, allowing preliminary data processing before transmission to the cloud. API Gateway served as a secure entry point for data, while AWS Lambda enabled serverless, event-driven processing. Implementing these services helped me understand the advantages of serverless architectures, particularly in terms of scalability and reduced operational overhead. For storage, DynamoDB was used for noise logs where low latency was important, while Amazon S3 was used for larger and less frequently accessed objects such as model weights and training datasets. This separation taught me the importance of using different storage services to align with different data access patterns. Finally, Amazon SageMaker allowed

training and deploying of machine learning models, completing the pipeline from data collection to intelligent analysis. Seeing these services work together transformed my understanding of cloud architectures from individual services into a cohesive system.

This project also required careful analysis of trade-offs when it came to UI/UX design considerations. For example, designing the noise disturbance reporting form to proceed in stages rather than presenting the whole process at once improved usability and reduced cognitive load, but prevented users from previewing the process. Additionally, displaying only noise classification results made the interface more interpretable to users, while exposing raw endpoint data could overwhelm users but benefit advanced analysis of the data. Each design decision involved their own set of trade-offs, and taught me to think beyond implementation and consider how real people would interact with and benefit from the system. I learnt that successful systems are not defined solely by their technical sophistication, but also by how effectively they meet user needs.

Overall, this project helped me in bridging theory and practice, by demonstrating how different aspects such as IoT protocols, hardware choices, cloud services, and user interfaces come together as a cohesive system.

Ho Xiaoyang - 1007006

My role in this project was to construct the logical flow of the system, as well as to interact with all the cloud services to ensure that all the layers of the system were able to be integrated with each other. I was responsible for the construction of the NoiseWatcher system design, experimentation with the initial sound sensor calibration (for KY-037 sound sensor), training and deployment of the machine learning model, the setting up of the code base for the Next.js user application, and the experimentation as well as the deployment of all cloud services (AWS Lambda, Sagemaker, DynamoDB).

Initially, the group was discussing ways to develop a well-rounded IoT solution for the class. After several iterations, I chanced upon this Straits Times article and realised that this noise monitoring system was actually not deployed yet, probably due to planning and budget constraints, I started to wonder about how we could utilise the knowledge learnt during class in order to develop an economical and effective solution, that was easy to maintain, sustainable, and most importantly, built for all residents of HDB flats to use. Thus, I began to shape the design of the system. We initially wanted the project to be a simple noise log system, but I wanted the group to challenge themselves by incorporating more advanced techniques to process and make meaning out of the values collected from the IoT sensors.

I was looking at various deep learning solutions online for noise classification and realised that all the pre-trained deep learning models were going to be too computationally heavy, as well as impractical for the limited analog values of the sound wave that our cheap sound sensors could produce. Therefore, by leveraging my previous knowledge on machine learning using signal processing, I devised a plan to incorporate this into our system design, so that there was an added layer of verification for the sound that was going to be in our verification system.

After experimentation with the initial sound sensor that we were going to use, I realised that the sensitivity of the sensor was too limited and that we were going to have to purchase new sensors. However, the initial information from the weaker sound sensor served as a starting point for data collection, to validate that signal processing techniques (using the Fourier transform) was sufficient to classify compressed sound data. From this, I was able to code the whole machine learning pipeline on my local computer and achieved a sufficiently good accuracy.

I also utilized the MQTT protocol in our system design as I found it to be especially useful, due to its ability to retain information from the core devices even after disconnection, as well as the “Time of Death” feature which serves as health checks for the many ESP32 sensors. This was how I decided to have the maintenance and monitoring feature since it could theoretically be easily integrated into our system design (even though we did not manage to implement the functionality). We utilised our knowledge of accessing the Raspberry Pi remotely, as I wanted to demonstrate the autonomous nature of our system, where the process of data collection and preprocessing can be started and continued independently from anywhere.

After setting up the main logical flow for the preprocessing and filtering of the data by frames (which were crucially overlapped during training to maintain robustness of the model), I moved on to setting up the basic web application UI, defining the routes for the endpoints provided by the cloud backend system.

After this was done, I focused on implementing the functionality of the cloud backend system. Initially, I did a lot of manual navigation on the AWS console UI, which proved to be very inefficient as there were 4 lambda functions to deploy, as well as a scalable database and the AWS SageMaker notebook. Thus, I decided to employ good coding practices to write the configurations in a .yaml, and Dockerfiles instead. This proved to be easier to deploy since I could change configurations directly in the file, without having to repeat the tiring process of clicking through the UI again.

I started to experiment with optimising system latency. Our system had the added benefit of utilising a very lightweight machine learning model (the model.joblib file is only about 3MB, compared to deep learning models that can be a few GB in size), which was why I pushed the entire code and dependencies for the noise_inference endpoint into an ECR instance, which was subsequently deployed as a container image on AWS Lambda. This resulted in our execution time being around ~19ms. Another optimization was the DynamoDB optimization of indexes. The dummy variable mentioned in the report actually enabled the system to utilise Query operations, which are much faster than Scan operations. Querying the whole database of around 600 items at testing resulted in a ~300ms execution time, as compared to the original Scan operation which would scan the entire table. One thing I regret not doing was experimenting writing the lambda functions in a low-latency language like C++. However, I eventually chose to use Python due to time constraints, as well as the APIs of machine learning libraries (like scikit-learn) being available only in Python.

To integrate all the system components together was challenging since it required me to understand exactly what components had to interact with each other, and the HTTP response formats returned from the lambda endpoints. It required an understanding of what key features our POC system needed, which was the fetching and filtering of noise logs (by timestamp and by noise classification), the fine tuning of the model, and most crucially the inference endpoint. I was able to deploy the Lambda functions and AWS Sagemaker instances and communicate the expected responses and the format of inputs to my group members, and we were able to successfully make the POC run as anticipated.

Finally, I also deployed the web application on an EC2 instance so that we can use it for future demonstration purposes.

Overall, I really enjoyed the process of learning about different IoT sensors, its constraints, and also about the convenience that cloud technologies bring in terms of being able to scale applications. I gained a deeper appreciation of microcontrollers and how they can be used to

power large systems, even though I had very little experience working with this before. I am more comfortable using AWS cloud now, though I would also like to explore more about open-source alternatives that can be cheaper. By designing the whole system from idea to fruition, I had to think about every single aspect of the constants of the system, such as the sampling rate and frame size of the collected data from the ESP, to the type of model used (Random Forest), to the requirements and constraints of certain cloud services (such as the Lambda layers having a certain limit, and various configuration set ups for the execution roles). I had to think very logically to figure out whether our system made sense and was feasible. I also gained a lot of soft skills like communication, as well as to write proper documentation of the entire process, through writing code documentation as well as writing the report. I learnt about treating presentations as a pitch, where we were able to pitch our ideas to the professor and TA.

By attending class, I also had the chance to validate this project idea with the professor and the TA, and attending the lesson on edge inference/cloud inference/federated learning really motivated me, as I had the sense that this project was relevant to current applications of IoT and Cloud.

As a person who has a deep appreciation for system design as well as machine learning applications, I wanted to utilize this project to help me learn how to integrate simple but smart solutions in a scalable way. By using a very small sound sensor connected to an ESP32, it is quite wonderful that a whole system of interconnectivity between IoT and Cloud can be built in such a way that people can use it in real-time, and will greatly benefit them in the long run. I will use this information in the future in my work, as well as in my personal projects.

Sun Sitong - 1007132

In this project, I mainly focused on implementing the sensor parts. This was a huge learning curve for me, and there are some specific highlights I want to mention that are not in the main report.

The first big hurdle I faced was collecting data to train our noise classification model. I used the ESP32 and the sound sensor to listen to noises like shouting or drilling. At first, I had this idea to save the data directly into a text file inside the ESP32's own memory storage. I thought this would be the easiest way to keep the data safe. However, I soon found out that getting that file back onto my computer was incredibly difficult. I tried using some special plugins to extract the files, but these methods were very buggy and gave me a lot of errors that I didn't understand.

So, I decided to try an easier way: just printing the data to the Serial Monitor on my laptop screen. I thought I could just copy and paste the text from there into a file. But after trying this a few times, I noticed something was wrong. The amount of data appearing on my screen was much less than what the sensor was actually collecting. I realized that the printing speed of the Serial Monitor was just too slow. The I2S sensor collects audio data very fast and my computer screen just couldn't keep up with printing it all out. I was losing valuable data without realizing it. To fix this, I had to change my whole approach. I decided to send the data over the Wi-Fi network to the Raspberry Pi. I send those data to Rpi in json format and pass to downstream process

I also want to talk about my experiments with different ESP32 boards. I originally wanted to show a cool demo with two sensors working at the same time for our PoC. But since we only had one standard ESP32 board available, I tried to use an ESP32-CAM board as a substitute for the second sensor. I spent a lot of time researching and testing it, hoping it would work just like the normal one. Unfortunately, I found out that the ESP32-CAM could not connect using the MQTT protocol like the normal ESP32 could. This is mainly because the camera board uses specific pins and software libraries for the camera that conflict with the standard Wi-Fi and MQTT libraries we were using. It was really frustrating to dig through code only to find out the hardware just wouldn't cooperate. Because of this, I couldn't make it wireless. I had to compromise and connect it using a basic wired connection (UART) just to show that the sensor itself was working, even if it wasn't wireless.

Another big lesson I learned here was about physical connections. When I first hooked up the high-quality microphone, I didn't solder the pins because I wanted to be quick. I just used jumper wires and let them touch the sensor's pads. This was a mistake. The data came back as errors or just "-1" constantly. I learned the hard way that high-speed digital signals need a solid connection. Once I actually soldered the pins properly, everything started working perfectly.

A tricky part of switching from the cheap KY-037 sensor to the high-quality INMP441 was dealing with the data output. The old KY-037 sensor gave us two outputs: an analog value (loudness) and a digital value (1 or 0) that triggered automatically when a sound was loud. Our entire system pipeline was already built to expect this "digital trigger." However, the INMP441 is purely digital and only outputs raw sound data; it doesn't have a built-in trigger pin. To fix this without breaking our code, I had to manually create a "digital" value in the software. I calibrated the sensor by testing it at a distance of about 2 meters, which mimics the real-world distance between two apartment doors in an HDB block. Based on these tests, I found a specific loudness value that consistently represented a "loud noise" at that distance. I then wrote a simple logic check in the code: if the raw sound value exceeded this threshold, the system would generate a digital "1", mimicking the behavior of the old sensor and keeping our pipeline intact.

Struggles with the Raspberry Pi Using the Raspberry Pi as a local server was honestly one of the most frustrating parts of the project. The connection to the RPi was extremely unstable. Many times, I tried to connect to it remotely using SSH from my laptop, but it would fail or disconnect suddenly.

This instability often corrupted the files inside the Pi. When the connection dropped or if the power wasn't turned off perfectly, the SD card would get messed up. I lost count of how many times I had to completely wipe the SD card, re-download the Operating System, and set everything up from scratch. I suspect this was likely caused by the SD card being old or having writing errors when the power was cut off suddenly. It took a lot of time to fix these boring system issues instead of actually writing the code for our project. And this issue even affected our final presentation, since it broke again just before our presentation started. Due to the instability of Rpi, maybe using other hardware to demo is a better choice.

Finally, before this project, I was completely blank about how IoT devices connect to the cloud. I didn't know how a tiny sensor could talk to a big server like AWS. Through the course materials and the lab exercises, I gradually learned how these things link together. It was like connecting the dots. I learned how to use AWS Lambda to run small pieces of code without setting up a huge server, and how to use DynamoDB to save our noise logs neatly in a table.

Overall, I think the project is highly relevant to what we have learnt in class and lab, for example, we experiment with different transmission protocols, also we learn about the layers of cloud application in class, which help us on the system architecture design. As we have tried out some AWS services in the lab, we are able to quickly set up the cloud services in our project, and dockerize our features.
