Smart Noise Monitoring System

Nicola Francesco Mancini
DEI – Electrical, Electronic, and Information Engineering
University of Bologna
40136 Bologna, Italy
nicola.mancini11@studio.unibo.it

Abstract— This paper presents the design and implementation of a Smart Noise Monitoring System using Internet of Things (IoT) technologies. This project aims to detect and predict noise values in a domestic environment using a single sensor, such that after assessing that the setup is reliable, it can be implemented to detect noise pollution in urban areas by leveraging multiple sensors. This system, built using the ESP32 microcontroller and a microphone sensor, employs the MQTT protocol for data acquisition and configuration updates. The sensor data is processed by a Python-based Data Proxy, stored in Influxdb, and analyzed for noise prediction using a Prophet model. Furthermore, a Grafana dashboard visualizes real-time noise levels, alarm events, Wi-Fi RSSI data, and the predicted noise values. The performance of the system is evaluated in terms of data acquisition latency and forecast error (MSE). The system also includes a noise classification to some possible noise sources feature and a Telegram bot for alert notifications and system configuration.

Keywords— ESP32, noise monitoring, data proxy, time-series database, noise values forecasting

I. Introduction

Noise pollution is a growing concern in urban environments. The constant hum of traffic, construction, and other human activities can have negative impacts on the health and well-being of city dwellers. Prolonged exposure to high noise levels can lead to hearing loss, disturbed sleep, increased stress, and other noise-related health issues. Consequently, the monitoring and control of noise pollution have become significant for improving urban life quality.

Knowing these issues, the development of smart noise monitoring systems has gained importance. Such systems can not only detect high noise levels but also predict and classify the source of the noise, providing valuable insights for city planners and health organizations.

The objective of this paper is to present the design, implementation, and evaluation of an IoT-based smart noise monitoring system. This system employs a microphone sensor to detect high noise pollution. Using AI techniques, it also predicts future noise values and can classify the source of the noise.

The noise values are monitored via the microphone sensor "Modulo Rilevatore di Suono – Uscita Analogica e Digitale" by FuturaGroup srl. Data is then sent to a data proxy script made in Python programming language. The data proxy forwards the data to the database InfluxDB for visualization and analysis. The user can modify four parameters of the monitoring system acquisition in real-time, submitting the desired values through the MQTT Broker (HiveMQ) or the Telegram Bot (Noise_Alarm bot).

Such parameters are ***sampling rate*** (the interval between two consecutive sensor's readings), ***noise threshold*** (only values higher than this threshold are sent to the proxy), ***alarm level*** (the threshold to generate an alarm), and ***alarm counter*** (the minimum number of consecutive readings of noise values, the alarm level, after which the alarm is triggered). The user is then notified when the alarm is raised, through the Noise_Alarm Telegram Bot. To have an idea of the possible trend of future noise values the data is provided via a machine learning prediction based on the collected data. In this way, the user can get an idea of the noise values trend and how the noise levels change throughout a seasonality period (which can be tuned in the Prophet model).

II. Project Architecture

This part is just an overview of the whole project. An in-depth analysis will be made in the "Project Implementation" section.

A. Hardware

For the Smart Noise Monitoring System, the ESP32 board was chosen. To collect the noise values, the choice had to be made between two sensors: the "Modulo Rilevatore di Suono", and the "Modulo Rilevatore di Suono – Uscita Analogica e Digitale", both from FuturaGroup Srl. The latter one was in the end used, with the analog output.

The specifications of both sensors are written in *Table 1* and *Table 2*. It is also presented the setup used, in *Figure I1*.

	Modulo Rilevatore di Suono
Supply Voltage	3,3 V – 5 V
Digital Output	5 V (No sound registered) 0 V (Sound registered)

Table I: first sensor specifications.

	Modulo Rilevatore di suono – Uscita Analogica e Digitale
Supply Voltage	5 V
Digital Output	(When the sound intensity reaches a certain threshold, the output switches from high level to low level)
Analog output	(Voltage output according to the intensity of the sound picked up by the microphone)

Table II: second sensor (the chosen one) specifications.

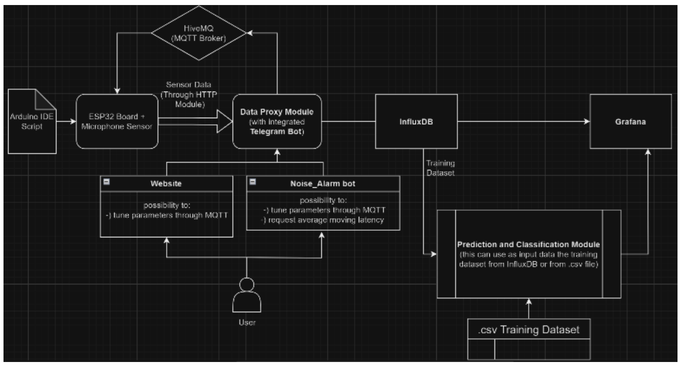


Figure I: Full Data Pipeline.

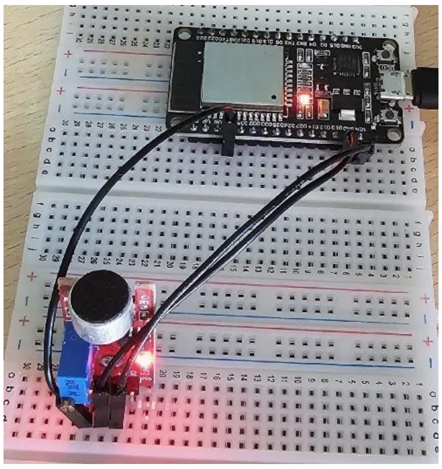


Figure II: Setup of Board plus Sensor.

A premise must be made here before discussing the actual acquisition process.

To convert the analog value read by the acoustic sensor into decibels (dB), it is necessary to have a calibration point. However, since the sensor specifications provided do not give a direct relationship between the analog value and the sound intensity in dB, only a very rough approximation could be made; it has been decided not to implement that conversion function in the Arduino IDE script, as a slight approximation could result in a significative change when applying the logarithmic function.

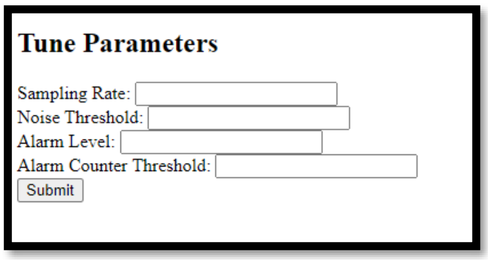
B. Data Acquisition

A key component in the data acquisition pipeline is the Data Proxy. This is a Python application that is run outside the microcontroller (on a laptop), unlike the Arduino code which is uploaded on the microcontroller. This application serves as the intermediary between the sensor and the database. It receives sensor data from the ESP32 microcontroller via the HTTP protocol and forwards it to the InfluxDB instance. The Data Proxy also manages configuration changes, receiving updates via the MQTT protocol (coming from the MQTT Broker, from the dedicated webpage, or the Noise_Alarm Telegram bot) and transmitting them to the microcontroller.

The InfluxDB instance is used as a bridge to display the values on Grafana.

	Initial value
Sampling Rate	60 seconds
Noise Threshold	10
Alarm Level	146
Alarm Counter	2

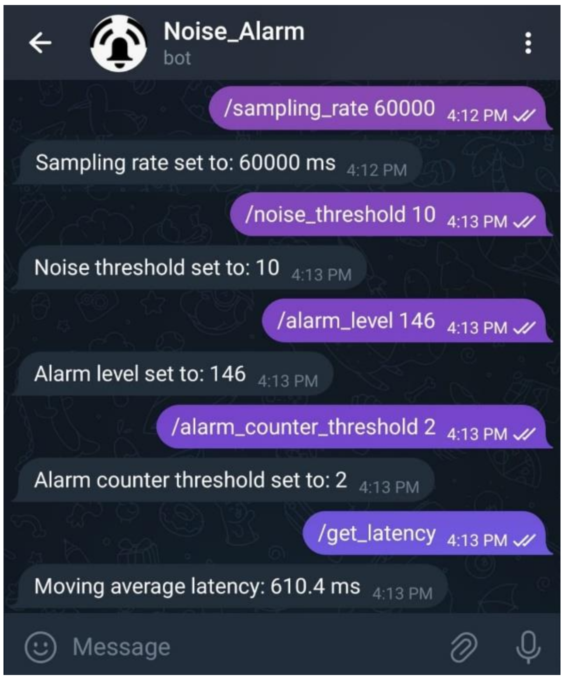
Table III: Initial values of the tunable parameters during the data acquisition part.



Tune Parameters

Sampling Rate:
Noise Threshold:
Alarm Level:
Alarm Counter Threshold:
Submit

Figure III: Webpage that allows tuning the parameters.



Noise_Alarm
bot

/sampling_rate 60000 4:12 PM
Sampling rate set to: 60000 ms 4:12 PM
/noise_threshold 10 4:13 PM
Noise threshold set to: 10 4:13 PM
/alarm_level 146 4:13 PM
Alarm level set to: 146 4:13 PM
/alarm_counter_threshold 2 4:13 PM
Alarm counter threshold set to: 2 4:13 PM
/get_latency 4:13 PM
Moving average latency: 610.4 ms 4:13 PM

Message

Figure IV: Noise_Alarm Telegram Bot that allows tuning the parameters.

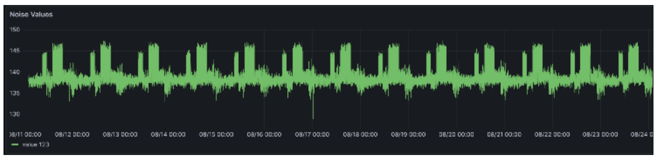


Figure V: Collected noise values visualization from the Grafana dashboard.

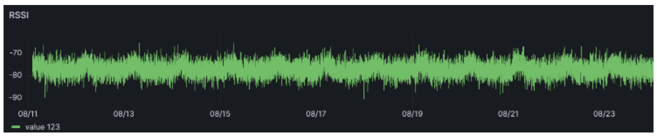


Figure VI: RSSI visualization from Grafana dashboard.

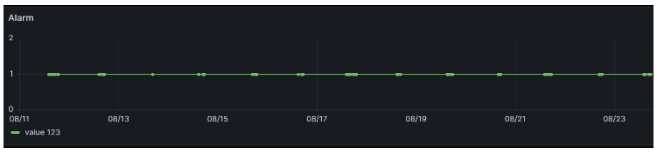
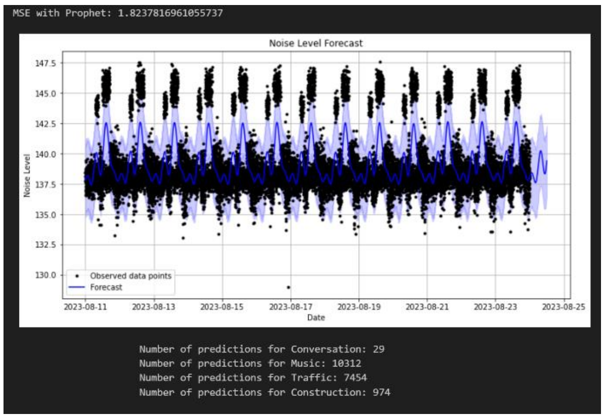


Figure VII: Alarm triggered visualization from Grafana dashboard.

C. *Prediction*

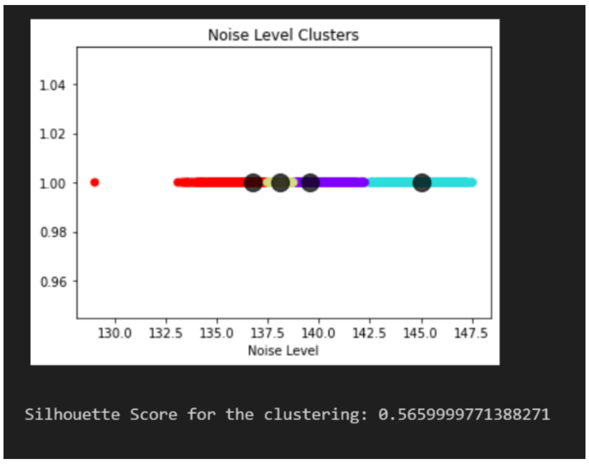
To forecast the future noise values, several models have been used among the ARIMA, SARIMA (Seasonal ARIMA), SARIMAX (Seasonal ARIMA with eXogenous values), and Prophet. After testing all of them, the Prophet had been chosen, because of the best overall performance obtained in terms of repetition in the accuracy of predictions. Initially, the training was based on the data acquired in one week, downloaded from InfluxDB in a .csv file. Then, the dataset covering a longer period of two weeks had been used. The forecasted values have then been displayed.



```
MSE with Prophet: 1.8237816961055737
```

```
Number of predictions for Conversation: 29
Number of predictions for Music: 10312
Number of predictions for Traffic: 7454
Number of predictions for Construction: 974
```

Figure VIII: Forecasted noise values visualization



```
Silhouette Score for the clustering: 0.5659999771388271
```

Figure IX: Actual noise values with the forecasted noise values trend visualization from the Grafana dashboard.

III. Project Implementation

The convergence of IoT capabilities with real-time environmental monitoring is represented in this ESP32-based sound monitoring system. The system records, examines, and remotely communicates ambient noise levels by utilizing the adaptability of the ESP32 microcontroller.

The main parts of the system and its operational logic are explained, more in depth, in this section.

A. *ESP32 Sketch*

The project starts by defining the ESP32 board script, written using the Arduino IDE environment.

A suite of libraries is used for the system's functionality. Particularly, “WiFi.h” and “WiFiClientSecure.h” ensure the integrity and security of WiFi connections. Simultaneously, two other libraries oversee data transmission dynamics: “PubSubClient.h” manages the MQTT protocol, while “HTTPClient.h” creates and dispatches HTTP requests. The temporal context is also crucial for precise data logging, and it is managed by “WiFiUdp.h”, “NTPClient.h”, and the standard time library, ensuring accurate timestamping.

During the initial configurations, the system specifies WiFi credentials to access the network. Details about the MQTT server, fortified with a secure connection certificate, shape the architecture for data transfer to a server; the sound sensor establishes communication through a specified ESP32 pin.

Predefined operational metrics, such as the sampling rate, noise threshold, alarm level, and alarm counter, guide the system's monitoring behavior. A salient feature of the system is its adaptability; it has the ability to respond to external MQTT-based commands, demonstrating its aptitude to recalibrate operational parameters in real time, aligning with dynamic monitoring requirements. These parameters can be modified through the MQTT broker (HiveMQ), the dedicated webpage, or even the Telegram Bot (Noise_Alarm bot).

The initialization phase commences with the activation of serial communication channels, facilitating diagnostics and system status relays. In this phase, two points are important to be tracked: the network subscription and the MQTT broker connection. If either connection stops, the system proactively attempts reconnection, enhancing the overall reliability. Once the network connectivity is confirmed, further steps solidify its communication framework by initiating a secure link with the MQTT server. The system also subscribes to MQTT topics, indicating its readiness to process and respond to external commands.

During its operational cycle, the system remains vigilant, constantly ensuring sustained connections to both the WiFi and MQTT platforms. At the core of its operations is the acquisition of readings from the sound sensor. When the reading exceeds the defined threshold for a defined number of consecutive times, the system activates an alarm protocol. Concurrently, it transmits the collected noise data, together with WiFi signal metrics and precise timestamps, to both the MQTT broker and an HTTP server (via a POST request). This dual-transmission strategy guarantees optimal data accessibility and facilitative analysis.

As reported before in section II.A, a conversion from the read analog value to a value in dB was not implemented, as the specifications of the sensor didn't specify a calibration point. The effective formula to make such a conversion, however, is the following one:

$$dB = 20 * \log_{10} \frac{V_{sensor}}{V_{ref}}$$

Where:

- V$_{sensor}$ is the voltage value read from the sensor;
- V$_{ref}$ is the reference voltage (which is in general the maximum voltage that the sensor can have as output. In this case the sensor is powered with 5V, so it is 5V).

But the analogRead() function in Arduino returns a value in the interval 0-4095 (because of the 12-bit ADC in the ESP32); so it is converted into a voltage value in this way:

$$V_{sensor} = \left(\frac{analogValue}{4095}\right) * 5$$

B. *Data Proxy and Telegram Bot*

The code delineated in this section showcases various components: threading for multitasking, MQTT for efficient messaging, and a Flask-based web application as the user interface.

The integration starts with some libraries: "threading" for concurrent task execution, "datetime" and "pytz" for managing time-centric operations, and "deque" from the "collections" library that streamlines the storage of latency metrics. The system also incorporates the "get_ntp_offset" function to compute the time difference between the system time and a given NTP server; this offset is essential for accurately timestamping data. The MQTT communication is made possible by the "paho.mqtt.client" library, which is suited for lightweight messaging, especially in constrained environments. Secure data transit is then used, leveraging SSL/TLS encryption via the "ssl" library. The system's interface with InfluxDB is validated, ensuring an efficient data storage mechanism. An MQTT setup, using an online broker (HiveMQ), is established, and to fortify its trust, a CA certificate is also used. The InfluxDB setup is implemented for time-series data storage, and the integration with the Telegram bot ensures instantaneous alerts and the possibility of real-time parameter tuning.

Two important MQTT callback functions are the center of the system's real-time dynamics: \`on_connect\` which manages topic subscriptions, and \`on_message\` which decodes MQTT payloads, computes latencies, assesses alarm conditions, and ensures data's persistence in InfluxDB. The Flask application defines routes to handle a variety of requests, from parameter tuning to average latency retrieval. The system allows three possible methods for parameters tuning: the MQTT Broker (HiveMQ), the dedicated Telegram Bot (Noise_Alarm bot), and the dedicated web interface hosted locally (" http://localhost:5000 "). It is also present the "RepeatedTimer" class, which offers periodic insights into the remaining time for the next average latency report (as output in the data proxy code interface.

This system uses a moving average approach for the average latency, creating a window for a selected set of latency values, and upon each iteration, a new value is stored while the oldest is discarded.

The average latency metric and the tunable parameters can be accessed through the Telegram Bot (Noise_Alarm bot).

C. *Prediction Module and Noise Classification*

This module delineates an approach to time-series forecasting and classification of noise data. This system combines the Prophet statistical model with machine learning algorithms to provide valuable insights.

The system integrates a set of specialized libraries. The "pytz" library is important for timezone-aware operations. The "influxdb_client" is implemented for interactions with the InfluxDB time-series database. The Scikit-learn library, which includes machine learning tools, is employed for clustering ("KMeans") and model evaluation ("silhouette_score", "mean_squared_error").

A timezone object for Rome (the local timezone) is instantiated to ensure that all datetime operations are locally relevant.

Configuration parameters are established for InfluxDB; the client is set up for data retrieval and storage operations.

A continuous loop fetches the noise data from InfluxDB from the last specified temporal arc, facilitating dynamic updates (there is also the possibility to load the dataset from a .csv file). The retrieved data goes through a preprocessing phase – to be sure to have a uniform time index and missing values are addressed via forward-filling.

Subsequently, the dataset is divided into training (80% of data) and testing (20% of data) subsets. Outliers, which can skew predictions, are identified using Z-scores and removed, ensuring the model's robustness.

The Prophet model, the central part of the forecasting process, is particularly good at managing datasets with seasonal patterns (and this is also one of the reasons why it was chosen). After initializing it with daily seasonality, the model is trained on the pre-processed data. By exploiting Prophet's capabilities, future timestamps are created, to be able to make subsequent predictions.

For model evaluation, the Mean Squared Error (MSE) between predicted values and the testing subset is computed, providing a quantitative metric of the model's predictive accuracy.

The actual and forecasted noise levels are plotted on the same graph, providing an intuitive understanding of the model's predictive power (in fact, the model also predicted the actual values based on the testing on the unseen data to see how the actual future forecasts would be good).

The KMeans algorithm then clusters noise data into four categories, each representing a plausible noise source: Conversation, Music, Traffic, and Construction. The centroids of these clusters, indicative of average noise levels, are determined, and the forecasted data is subsequently classified into one of these categories; after this process the occurrences of each cluster are counted and printed out, to make better further possible analysis.

The silhouette score, a metric that quantifies the quality of clustering, is so computed. It is a metric that goes from -1 to +1. A higher silhouette score indicates that the clusters are well-separated.

Basically:

- Values near +1 indicate that the sample is far away from the neighboring clusters.
- A value of 0 indicates that the sample is on or very close to the decision boundary between two neighboring clusters.
- Negative values indicate that those samples might have been assigned to the wrong cluster.

The forecasted noise levels, along with their predicted sources, are stored back into InfluxDB for analysis or visualization; InfluxDB is also linked to Grafana for better visualization purposes.

After every forecasting and storage operation, the system pauses for a defined interval, which can be tuned, before the next data acquisition and prediction cycle recommences, always ensuring that the predictions remain as coherent as possible with the evolving environment.

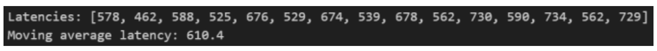
IV. RESULTS

The system performance has been evaluated utilizing two metrics: the Mean Square Error (MSE) of the Prophet model's predictions and the mean network latency to send the acquired data to the proxy. For the MSE the following values are obtained: using the dataset over 1 week, the resulting MSE is circa 1.627; extending the dataset over two weeks, the MSE is 1.823.

For the noise classification, it has been computed also the Silhouette Score metric, explained in the previous section, and the obtained values are 0.568 for the dataset over one week and 0.566 for the dataset over two weeks.

Overall, the MSE (and also the Silhouette Score) achieved with both datasets can be regarded as a sufficiently good result, which can be further improved by enlarging the available training data, changing the location of the measurements, and also by using a more accurate or professional microphone sensor.

For the network latency, the moving average window approach has been used, with a window size of 15 samples to have a stronger idea of the behavior of the setup. The result obtained here is an average moving latency of < 800 ms at most (several runs of the command /get_latency on the Telegram Bot were done to have an idea of the performance of the model over the maximum period. This was done to check the reliability of the system over longer periods, so two weeks were considered). This is a great result and also compatible with the application if it is compared to the sampling time (the starting one was 1 minute and was not modified significantly).



```
Latencies: [578, 462, 588, 525, 676, 529, 674, 539, 678, 562, 730, 590, 734, 562, 729]
Moving average latency: 610.4
```

Figure X: Visualization of a filled Average Window with the corresponding Moving Average Latency value.

V. CONCLUSIONS AND FUTURE WORKS

This project has successfully demonstrated the viability of a Smart Noise Monitoring System using IoT technologies. The results confirm that the system can effectively detect, predict, and classify noise in domestic environments, showing potential for broader application in urban noise pollution monitoring. Future works will focus on enhancing the system's predictive accuracy by incorporating larger datasets and advanced sensor technologies. Additionally, exploring the integration of machine learning algorithms for more refined noise source classification will be a priority. Expanding the system to operate in diverse urban settings will also be crucial, aiming to provide a more comprehensive solution for noise pollution management in cities. The potential for real-time urban noise mapping and its implications for urban planning and public health will be a key area of exploration.