Mapping urban noise pollution via smartphone

Kevin Boni, Giovanni Enrico Loni, Lorenzo Mancinelli, Alice Orlandini
k.boni@studenti.unipi.it, g.loni1@studenti.unipi.it, l.mancinelli1@studenti.unipi.it, a.orlandini10@studenti.unipi.it

ABSTRACT

Noise pollution is an increasing threat to health and quality of life in urban areas, yet its widespread and continuous monitoring is often limited by the high cost and sparse coverage of traditional sensing infrastructures. In this context, crowdsensing has emerged as a promising approach, leveraging the ubiquity of mobile devices to collect large-scale environmental data. In this context, we present NoiseCity, an Android application designed to enable users to record audio samples using their smartphone's microphone and contribute to the collaborative creation of urban noise intensity maps. The collected data is geolocated using the Fused Location Provider Client and stored in a MongoDB database hosted on a Flask server. Aggregated measurements are visualized on an interactive heatmap, built using Geohash encoding, allowing users to explore noise intensity by time of day. To encourage active participation and improve the spatial coverage of the recordings, the app also features a system of achievements. NoiseCity thus represents a collaborative, scalable, and technologically accessible platform for urban noise pollution monitoring.

1 Introduction

Urban noise pollution is a growing concern in modern cities, with proven impacts on both physical and mental health. While many solutions exist for environmental monitoring, most are limited by static infrastructure, high deployment costs, or lack of granularity. Current systems often rely on fixed sensor networks or municipal datasets, which provide limited temporal and spatial resolution and are not always accessible or up to date.

This project proposes a **crowdsensing-based solution** aimed at dynamically mapping the noisiest areas of a city by leveraging the microphones and GPS sensors of users' smartphones. Through a mobile application, users can record and submit audio samples that are automatically processed and geo-referenced. These samples are then aggregated and visualized on a **real-time heatmap**, allowing anyone to explore noise intensity across different areas and time windows.

Compared to traditional monitoring systems, our approach is:

- **Low-cost and scalable**, requiring no dedicated hardware infrastructure.
- **Adaptive**, since the granularity of data grows with user participation.
- **Interactive**, enabling users to explore the city's acoustic profile by selecting custom time ranges and navigating the map in real time.

By integrating FFT analysis, historical trends, and personalized statistics, the application not only informs users about urban noise conditions but also encourages active contribution and awareness about environmental health.

1.1 Related works

Similar approaches have been proposed by the applications Ear-Phone [1] and NoiseCapture [2]. Both applications investigate the potential of smartphones in developing crowdsensing-based noise mapping solutions. In the case of Ear-Phone, the application also provides visualizations of the recorded sound's waveform and FFT (Fast Fourier Transform).

The solutions presented in these works are technically more advanced and comprehensive, partly due to the integration of machine learning models (which will be discussed further in the Future Works section). However, these approaches adopt a more conventional strategy in engaging users in the data collection process. In contrast, our approach seeks to actively raise user awareness about the risks associated with noise pollution, also by encouraging the achievement of personal goals.

2 Architecture

The application's functioning can be divided into two main phases: an initial phase focused on user authentication and registration, and a subsequent phase where authenticated users can record sound levels, view interactive heatmaps, and track personal achievements. As a client-server application, it relies on continuous communication between the Android client and a remote backend to store, retrieve, and process data efficiently.

2.1 Server and database

The backend server interacts with a MongoDB database that stores all relevant user and measurement data. The following collections have been defined:

- **users**: Stores authentication credentials and user-specific metadata, including a running counter of submitted measurements and a set of achievement flags.
- **raw_measurements**: Contains individual records of user-submitted measurements. Each document includes data such as volume, timestamp, geographical position.

- **aggregated_measurements**: Used to group measurements by geospatial proximity, leveraging level-7 geohash precision to achieve a trade-off between heatmap resolution and query performance. This collection includes a time bucketing field to support time-based filtering when generating heatmaps over custom date ranges.
- **user_cities** and **user_countries**: These collections track the unique cities and countries visited by each user. Each document includes a reference to the user and the corresponding geographical identifier. This structure enables efficient tracking of user mobility and supports analytics on measurement coverage across different regions.

This schema has been designed to minimize computational overhead for common queries (e.g., achievements and heatmap generation) while preserving flexibility for future analytics and data aggregation tasks.

2.2 Login and registration

Before accessing the core functionalities of the application (such as heatmap selection, contributing with sound samples) the user must first authenticate through a login system based on email and password.

Login Flow

When the app is launched, it checks for the presence of a valid authentication cookie. If found, the user is automatically directed to the main interface, which includes access to the noise heatmap, sample collection, and statistics and achievements modules. Otherwise, the user is presented with the login screen.

The initial activity loaded when the application starts is the `LoginActivity`. This component utilizes the `AuthRepository` service to perform the login operation by communicating with the backend API. Upon successful authentication, the `SessionManager` is used to store both the user's email and the authentication cookie locally, ensuring the session persists across app launches.

Authentication is performed through a POST request to the `/login` endpoint. The request includes the user's email and password. Upon successful validation by the Flask-based backend server, an authentication cookie is returned. This cookie must be included in subsequent HTTP requests to authorize access to protected endpoints.

At the time of writing, there is no password recovery feature, though this is identified as a potential improvement for future versions.

Registration Flow

New users can register by clicking the "Register" button on the login screen. The registration process requires entering an email address, a password, and confirming the password. A POST request is sent to the `/register` endpoint, and upon success, the user is redirected to the login screen.

User registration is handled by the `RegistrationActivity`, which also relies on the `AuthRepository` to send user credentials to the backend.

Passwords are hashed onto the server before being stored in a MongoDB database, ensuring that no sensitive data is saved in plaintext. If the email is already registered or the passwords do not match, appropriate error messages are returned. Once registered, users gain full access to all app functionalities.

Currently, all users are treated equally within the system, there is no distinction between user roles, as the app does not require administrative privileges for map visualization, sample contribution, or statistics tracking.

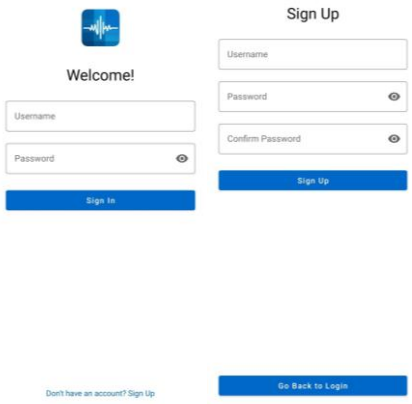

Figure 1: Login page (left) and registration page (right)

2.3 Sound recording

One of the most important architectural choices concerns the calculation and sending of acoustic measurements, which we wanted to make as efficient, lightweight, and privacy-friendly as possible.

For the sound level calculation, while we are aware of the advantage of A-weighting in modeling the sensitivity of the human ear, we preferred not to include a native A-weighting filter in Kotlin. Implementation would require extensive DSP libraries, tuning of digital coefficients, and significant computational load on mobile CPUs, resulting in increased battery consumption and development complexity. Instead, we obtain the sound level in full-

scale decibels (dB FS) by applying a simple RMS on one-second windows.
To this value we add a user-chosen calibration offset (in dB), stored in the local database, to account for hardware differences between different microphones. The user can thus calibrate their app according to a reference instrument or personal needs, resulting in a dB SPL result.

Figure 2: Compensation offset entry screen

To contain bandwidth consumption and protect privacy, we do not send the full waveform, only the aggregated data.
This strategy avoids sending tens of megabytes of samples per minute, reduces the latency of HTTP requests, preserves the device battery, and does not expose sensitive audio content.

UI Components

The `SoundWaveformScreen` provides users with a simple interface. At the bottom of the screen, two buttons—**Start Record** and **Stop Record**—allow the user to begin and end a sound capture session. When recording is active, a waveform visualization is displayed in real-time.

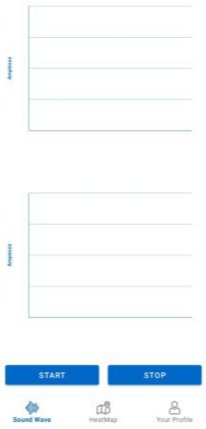

Figure 3: SoundWaveformScreen page

Signal Visualization

The WaveformView component shows two synchronized plots for each audio capture session. First, the **time-domain waveform** traces the signal's amplitude over time: the horizontal axis spans every millisecond of the recording (sampled at 44.1 kHz), and the vertical axis indicates the instantaneous amplitude normalised between –1 and +1. In this view, pronounced spikes correspond to transient events such as consonant bursts or percussive onsets, while sections hovering near zero reveal quieter moments or near-silence. Even a "pure" scream looks jagged here, since microphone characteristics, vocal tract resonances, and tiny environmental reflections all combine to create a complex waveform.

Beside it, the **frequency-domain spectrum**—computed in real time via a Fast Fourier Transform—maps the signal's energy distribution across pitch. Its horizontal axis stretches from 0 Hz up to 10 kHz, divided into evenly spaced bins, and vertical grid lines often highlight octave landmarks (1 kHz, 2 kHz, 4 kHz, etc.). The vertical axis plots each bin's magnitude (the square root of the sum of squares of its real and imaginary parts), so peaks reveal the dominant tonal components and the overall downward slope illustrates natural spectral roll-off. Together, these two complementary views give users an immediate, intuitive sense of **when** the sound peaks and **which** frequencies dominate—making on-device quality checks, calibration, and urban sound-scape analysis both straightforward.

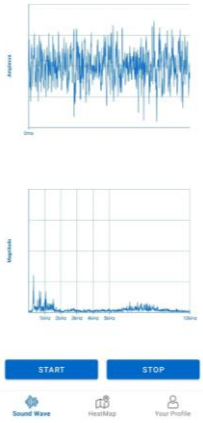

Figure 4: Sound wave visualization during recording

An additional calibration button is provided to allow the user to set a **decibel offset**, which can be used to align the recorded data with a known calibration baseline.

Recording Logic

When the user presses the start or stop recording buttons, the `SoundController` class manages the lifecycle of the recording session. Its responsibilities include:

- Ensuring that the required **permissions** for microphone and location access are granted.
- Controlling the recording process by interacting with the `AudioRecorder` , which handles the capture of raw audio data.

During the recording, real-time audio visualization is delegated to the `WaveformView`, which processes the incoming signal and computes the **time-domain amplitude plot** and the **frequency-domain plot**, using a Fast Fourier Transform (FFT) to extract magnitude and frequency components.

Each recorded session is represented by an instance of the `AudioSample` model, which includes data such as the mean value.

Once the recording is completed, the `DataSender` service is used to transmit the resulting `AudioSample` to the backend via the `/measurements` API. This component is responsible for constructing the HTTP requests and handling any network-related operations.

2.4 Heat map visualization

To provide users with a clear and interactive view of noise levels across the city, the application includes a heatmap feature rendered within a Google Map. This functionality is encapsulated in the `MapScreen` fragment.

User Interaction and Map Setup

When the `MapScreen` is loaded, it displays a Google Map interface configured using an API key stored in the `local.properties` file. This key is obtained through the Google Cloud Platform Console and enables access to the Maps SDK.

Upon launch, the screen presents a dialog prompting the user to input a **start** and **end date and time**. These values define the time range for which noise data should be visualized.

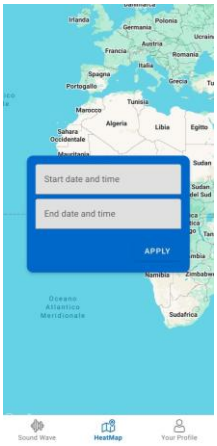

Figure 5: Start and end date and time selection

Once the user confirms their selection, the `HeatmapRepository` service is used to fetch the relevant sound samples from the backend using the GET API `/measurements`. The map is initially centered around the user's current location, and a query is performed within a **1 km radius**. As the user interacts with the map (such as panning or zooming) the application dynamically updates the query, adjusting the center coordinates and radius based on the **current viewport zoom level**. This ensures that the heatmap remains relevant to the area being displayed and allows for real-time exploration of noise levels across the city.

Heatmap Rendering

After retrieving the data, the application generates the heatmap using the `HeatmapTileProvider.Builder()` utility from the Google Maps Android library. This component overlays a heatmap layer on the map based on the geographic distribution and intensity of the recorded noise levels.

A **button** is available in the bottom-right corner of the screen, enabling the user to open the date range dialog again and refresh the visualization based on new temporal parameters. Instead, in the lower left corner is a button that allows you to reposition to its current position.

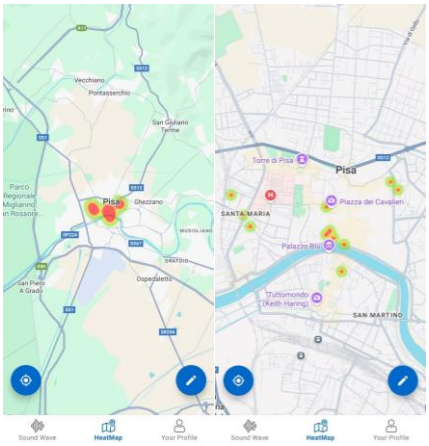

Figure 6: Heatmap visualization with different levels of zoom

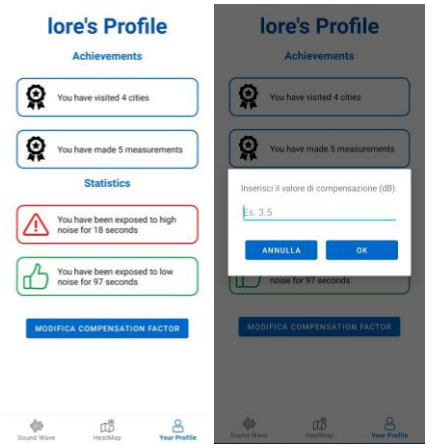

Figure 7: User profile summary page

2.5 Achievements and statistics visualization

When the user selects the "Your Profile" item in the bottom navigation bar, the navigation system defined in `nav_graph.xml` loads the `UserProfileScreen` fragment. During `onViewCreated`, the `fetchUserSummary()` function is invoked, which retrieves the current user's data from an HTTP endpoint via the `/profile` route, including the username as a parameter. The request is executed in the background (on the `Dispatcher.IO` thread) using `OkHttp`. The JSON response is converted into a `UserAchievements` object containing the username, a list of achievements with titles and descriptions, and noise exposure values. On the main thread (`Dispatcher.Main`), the UI is dynamically updated by displaying the username, the “City Explorer,” “World Traveler,” and “Measurement Master” badges (if they are actually present in the user table), and the noise exposure statistics, which are populated in their respective `CardView` elements, as we can see in Figure 7.

In addition, the profile page includes a **"Modifica Compensation Factor"** button, which allows the user to update the compensation factor used during noise level measurements. The new value is saved locally in the SQLite database and applied to future measurements.

Access to the `/profile` route occurs via an HTTP GET request, protected by authentication mechanisms that extract and validate the user based on a token. Once the user is authenticated, the backend calls three functions defined in the `repository.py` module to compose the response:

1. `get_by_id(user_id)` / `get_by_username(username)` – returns the user's information stored in the `users` table of the database.
2. `get_high_exposure(username)` – provides the total time of exposure to high noise levels.
3. `get_low_exposure(username)` – provides the total time of exposure to low noise levels.

Both exposure times are calculated by summing the duration of audio samples recorded by the user, obtained from the `raw_measurements` table by filtering based on the user.

3 Experimental results

To assess the accuracy and efficiency of our application, we conducted several tests, focusing on the correct rendering of the recorded waveform, the proper display of audio samples on the map, and the application's energy consumption. The tests were carried out using the following devices:

- Redmi Note 9s running Android 11
- Motorola E15 running Android 14

3.1 Sound wave

The waveform representation is reasonably accurate. This was verified by comparing the app-generated waveform with that of a known reference frequency. Additionally, a video demonstration was recorded to visually confirm the fidelity of the waveform measurement.

3.2 Heatmap

The visualization of data points on the map demonstrates satisfactory accuracy. This is evident from the distinct color

variations that correspond to different noise intensity levels, allowing intuitive interpretation of noise distribution. Additionally, the positional accuracy has been validated by comparing the mapped points with locations measured using external tools such as Google Maps. Potential improvements include implementing additional geohashing levels to enhance spatial resolution and detail.

3.3 Energy consumption

The app's energy consumption was measured using the Android BatteryOne app, with the device in battery saver mode at around 15–20% battery.

- A **10-minute session** with frequent switching between features (noise detection, waveform/FFT visualization, and map interaction) consumed about **104 mAh** (~624 mAh/hour), reflecting the high cost of continuous audio processing and frequent data uploads.
- A **2-minute session** with only noise detection active used approximately **6 mAh** (~180 mAh/hour), consistent with continuous microphone use and network activity.
- A **2-minute session** focusing solely on map visualization (fetching geohash-based heatmap data) consumed about **0.4 mAh** (~12 mAh/hour), showing a much lower energy impact.

Battery saver mode during testing may have affected power management and measurement accuracy. Overall, these results suggest optimization opportunities, especially in data upload frequency and UI rendering.

4 Conclusion

In this work, we presented NoiseCity, an Android-based application designed to support collaborative monitoring of urban noise pollution through crowdsensing. The architecture combines audio signal processing, geolocation, real-time heatmap visualization, and gamification elements to promote both user engagement and environmental awareness.

Field testing demonstrated the reliability of waveform visualization, the effectiveness of geolocating audio samples, and an acceptable level of energy consumption across different usage scenarios. These results confirm the feasibility of our approach in real-world settings.

However, several challenges emerged during development and testing. In particular, variations in smartphone hardware necessitated the implementation of a manual calibration feature to standardize measurement results. Additionally, the fixed geohash resolution used in the current implementation limited the spatial granularity of the heatmap in densely sampled areas. The lack of an A-weighting filter also affected the perceptual accuracy of the measured sound levels, since RMS-based calculations do not account for human auditory sensitivity.

Despite these limitations, NoiseCity represents a promising platform for participatory environmental monitoring, combining technological accessibility with user-centered design.

4.1 Future works

Several avenues for improvement and extension have been identified:

- **Noise classification:** Integration of machine learning models to distinguish between different types of noise (e.g., traffic, construction, human activity).
- **Enhanced geospatial precision:** Implementation of more levels of geohash resolution to improve the spatial detail of heatmaps in high-density areas.
- **Extended audio statistics:** Transmission of more detailed acoustic metrics – such as maximum and minimum peaks – to enhance server-side analytics and thus improve the user experience.
- **Gamification expansion:** Development of new achievements and personalized statistics to increase user engagement.

REFERENCES

[1] Rajib Rana, Chun Tung Chou, Nirupama Bulusu, Salil Kanhere, Wen Hu. "Ear-Phone: A Context-Aware Noise Mapping using Smart Phones" https://doi.org/10.48550/arXiv.1310.4270.

[2] Can, Arnaud, Audubert, Philippe, Aumond, Pierre, Geisler, Elise, Guiu, Claire, Lorino, Tristan and Rossa, Emilie. "Framework for urban sound assessment at the city scale based on citizen action, with the smartphone application NoiseCapture as a lever for participation" *Noise Mapping*, vol. 10, no. 1, 2023, pp. 20220166. https://doi.org/10.1515/noise-2022-0166