IoT Edge-Based Pest Detection System

Using audio sensors and machine learning at the network edge to detect and identify pests in real time

January 2026

SUBMITTED TO

University of Craiova
Faculty of Automation, Computers and Electronics
Ș.l. dr. ing. Ionuț-Dorinel Murarețu

SUBMITTED BY

Dan-Cosmin Dăgădiță, Cristiana-Andreea Foamete, Ștefan-Marian Preda,
Theodor-Bogdan Vânătoru

Contents

1. Demo 1
2. Introduction 1
3. System Architecture and Methodology 1
4. Software Design and Implementation 2
5. Dataset and Model Training 3
6. Experiments and Results 3
7. Limitations and Future Work 4
8. Conclusion 4

1. Demo

YouTube link for the demo: https://youtu.be/gAWUZB2IzyQ

2. Introduction

One of the biggest threats to food storage facilities is the presence of pests, which cause substantial economic losses through consumption and contamination. Traditional monitoring relies on manual inspections or passive traps, which are reactive rather than proactive. This project implements an Edge-based IoT system designed to detect pest activity in real-time using acoustic signatures.

Definition 1

Edge Computing: A distributed computing paradigm that brings computation and data storage closer to the sources of data, such as IoT devices. This reduces latency and bandwidth use by processing data locally rather than relying solely on a centralized cloud.

Monitoring food storage environments presents several technical hurdles:

- **Environmental noise:** Facilities are often noisy due to ventilation and machinery, making it difficult to isolate the subtle sounds of scratching or squeaking.
- **Bandwidth constraints:** Streaming high-quality audio from dozens of sensors to a cloud server for analysis is prohibitively expensive in terms of network load.
- **Real-time response:** Detection must be instantaneous to trigger local deterrents or alert staff before significant damage occurs.

To address these challenges, this system utilizes **Tiny Machine Learning (TinyML)**. By deploying an optimized Neural Network directly onto an ESP32 microcontroller, the device can "listen" and classify sounds locally. Only the metadata (probability of pest presence) is sent over the network via the MQTT protocol, ensuring privacy and low data consumption.

Definition 2

TinyML: A field of study in Machine Learning and Embedded Systems that explores the types of models that can be run on low-power, resource-constrained hardware like microcontrollers.

3. System Architecture and Methodology

The system architecture follows an end-to-end pipeline: from digital signal acquisition to visualization.

- **Acoustic sampling:** The system uses a KY-038 microphone sensor to capture ambient sound. Instead of processing full waveforms (which would exceed the ESP32's RAM), the system calculates **acoustic intensity** over a 10ms sampling window.

- **Feature engineering:** A rolling window of 300 intensity samples (representing 3 seconds of audio) is maintained in the device's memory. This window acts as the input vector for the model.
- **Inference layer:** A Deep Neural Network (DNN) implemented via **TensorFlow Lite for microcontrollers** processes the window.
  - **Input layer:** 300 nodes (normalized intensity values).
  - **Hidden layers:** Dense layers with ReLU activation to extract patterns of pest activity.
  - **Output layer:** A Sigmoid neuron providing a value between 0.0 and 1.0, representing the probability of pest activity.

Definition 3

Inference: The process of a trained machine learning model making predictions on new, unseen data. In this project, inference happens entirely on the ESP32 chip.

- **Telemetry and configuration:** The device communicates using the **Message Queuing Telemetry Transport (MQTT)** protocol. It publishes detections to a telemetry topic and subscribes to a configuration topic, allowing users to update the detection threshold remotely without reflashing the hardware.

4. Software Design and Implementation

The software ecosystem is divided into three distinct components: the Firmware (C++), the Backend (Go), and the Frontend (Vue.js/Quasar).

Firmware (Edge Node) The ESP32 firmware is built using the PlatformIO framework. It utilizes a multi-threaded approach via FreeRTOS. The `DetectorTask` is pinned to Core 1 to handle high-frequency sensor sampling and model inference, while the main loop handles Wi-Fi and MQTT connectivity on Core 0. This ensures that network jitter does not drop audio samples. The firmware includes modules for hardware abstraction, WiFi management with captive portal configuration, MQTT communication, and persistent storage of settings.

Backend (Edge Gateway) The backend is implemented in **Go**, chosen for its simplicity, high performance, and native support for concurrency.

- **MQTT Broker:** The system integrates the `mochi-mqtt` library to act as an embedded broker with WebSocket support for the dashboard.
- **Database:** A SQLite database using the GORM library stores device metadata and detection events. It uses **Write-Ahead Logging (WAL)** mode to allow simultaneous reads and writes during high-frequency detection events.
- **API:** An Echo-based REST API provides endpoints for the dashboard to retrieve historical data and update device configurations.

Frontend (User Dashboard) The dashboard is a Single Page Application (SPA) built with Quasar. It provides real-time monitoring through WebSockets connected to the MQTT broker, while also consuming the REST API for historical data. The interface features device overview pages, detailed telemetry visualization, interactive threshold controls, and a timeline of detection events.

DEFINITION 4

Latency: The time delay between a physical event (a mouse scratching) and the system's response (an alert appearing on the dashboard). Lower latency is critical for real-time monitoring.

Infrastructure and Reproducibility To ensure the system can be deployed in any environment, the entire stack is containerized. A `compose.yml` file manages the Google Colab local runtime for training (with Nvidia GPU support), the Go backend, and the Quasar frontend. The training environment includes the Nvidia Container Toolkit, allowing for rapid model iteration using the provided `notebook.ipynb`.

5. Dataset and Model Training

The model was trained on a custom-curated dataset stored in `.csv` format, categorized into three distinct classes to ensure robustness.

- **Background (background.csv):** Ambient noise from warehouse environments.
- **Distractors (distractor.csv):** Human footsteps, door sounds, and machinery noise.
- **Target (target.csv):** Recorded audio of mice and rats (scratching and high-pitched squeaks).

DEFINITION 5

Window Size: The specific amount of data points the model looks at to make a single prediction. In this project, a window size of 300 represents 3 seconds of monitoring data.

To improve the model's ability to "hear" pests in noisy environments, **Data augmentation** was used. During training, target pest sounds were mathematically mixed with distractor sounds (30% intensity) to simulate realistic warehouse conditions. The final dataset was split into 80% for training and 20% for validation.

The neural network was trained for 50 epochs using the Adam optimizer. Class weighting was applied to penalize false positives more heavily than false negatives, reducing the likelihood of "alarm fatigue" in operational settings. The trained Keras model was converted to TensorFlow Lite format and optimized for microcontroller deployment.

6. Experiments and Results

Several experiments were conducted to measure the efficiency of the edge node and the accuracy of the detection model.

Edge Performance Benchmarks

The execution time of various system components was measured to identify bottlenecks:

- **Optimized serial logging:** 0.5ms
- **MQTT publication:** 5.0ms
- **Model inference:** 12.0ms
- **LCD display update:** 62.0ms (Identified as a major bottleneck, mitigated using a non-blocking timer).

DEFINITION 6

False Positive: An error where the model incorrectly detects a pest when none is present. In food storage, too many false positives lead to "alarm fatigue" for workers.

Detection Accuracy The final TFLite model achieved an accuracy of **98%** on the validation set.

- The model was highly successful at detecting continuous vocalization patterns and scratching sounds.
- The use of a remote **threshold** parameter (default 0.90) allowed for real-time sensitivity tuning via the dashboard.
- The false positive rate was maintained below 2%, ensuring practical usability.

7. Limitations and Future Work

While the system performs well, several limitations exist:

- **Environmental dependence:** The model's performance may degrade in environments with acoustic profiles significantly different from the training data.
- **Sensor limitations:** The KY-038 microphone has frequency response constraints that may miss certain pest sounds.
- **Unimodal sensing:** The system currently relies only on acoustic data, without cross-validation from other sensors like motion detectors.
- **Power considerations:** The design assumes continuous power rather than battery operation.

Future improvements could include:

- **Adaptive thresholding:** Automatic sensitivity adjustment based on time of day or ambient noise levels.
- **Edge coordination:** Enabling devices to communicate for sound localization.
- **Predictive analytics:** Using historical detection patterns to predict infestation risks.

8. Conclusion

This project successfully demonstrates an end-to-end Edge IoT system for pest detection. By utilizing an ESP32 and TinyML, we achieved a high detection accuracy (98%) while maintaining a very low data footprint.

The experiments highlighted that while the neural network is efficient, peripheral tasks like updating an LCD screen can significantly impact real-time performance if not managed with asynchronous programming.

Furthermore, the integration of a Go-based backend and a Quasar frontend proves that "Edge" devices do not have to exist in isolation. The ability to update detection sensitivity (thresholds) via MQTT provides a level of flexibility usually reserved for cloud-heavy applications, but with the speed and privacy of local processing. The system provides a practical, scalable foundation for industrial monitoring that balances performance with resource constraints.