IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 4, April 2026

DOI: 10.17148/IJARCCE.2026.154125

A Hybrid Signal-Processing and Neural Network Pipeline for Low-Latency Acoustic Event Detection at the Edge

Jatin Verma[1], Abhishek Jatla[2], Thanuja Reddy[3], Ms. Charulatha R.T[4]

Department of Computer Science and Engineering,

SRM Institute of Science and Technology, Vadapalani, Chennai, India[1-4]

Abstract: This paper presents an edge-computed AI framework that detects a specific acoustic event — a human handclap — and actuates a wireless IoT device in response, without touching a cloud server at any point. Audio is captured through a professional condenser microphone, streamed into a Python processing pipeline via a non-blocking callback queue, and subjected to two lightweight deterministic gates before any neural computation occurs. The first gate discards frame whose peak amplitude falls below a fixed noise floor, keeping the classifier idle during silence. The second gate enforces a post-detection bypass window, preventing room echoes from generating spurious follow-on triggers. Frames that clear both gates pass through a two-stage neural pipeline: Google's YAMNet model extracts 1024-dimensional acoustic embeddings, and a purpose-trained Keras Sequential classifier maps those embeddings to a binary confidence score. When confidence exceeds 0.28, the system sends a UDP packet over the local Wi-Fi network to an ESP32 microcontroller, whose on-board LED toggles as hardware confirmation of successful end-to-end delivery. The entire loop — from live audio capture, through AI inference, to physical actuation — runs within a sub-second latency budget on commodity CPU hardware, with no internet dependency. The result is a privacy-preserving, network-independent architecture that generalises to any latency-critical acoustic IoT trigger application.

Keywords — Edge Computing; Acoustic Event Detection; YAMNet; Transfer Learning; Keras; ESP32; Real-Time Signal Processing; Deterministic Frame-Skipping; IoT Actuation.

1. INTRODUCTION

1.1 Background and Motivation

The first wave of IoT devices kept things simple: thermistats, motion sensors, door contacts. These produce one number at a time, easy to send and easy to act on. Sound is a fundamentally different kind of signal — continuous, high-bandwidth, and rich with contextual meaning that a scalar sensor cannot capture. Early IoT systems largely bypassed audio for exactly those reasons. That has started to change. Compact inference runtimes and well-trained pretrained audio models have lowered the bar enough that meaningful acoustic classification is now possible on a modest edge node, without a GPU and without a cloud subscription.

Despite that shift, most systems still default to the cloud. Raw audio is streamed to a remote endpoint for inference, and the result arrives back over the network before any local action is taken. For applications that can afford to wait — transcription, long-form analysis — this is perfectly workable. For a system designed to trigger a physical response the moment a specific sound occurs, it is not. The round-trip latency is too long, the privacy implications of continuous audio streaming are too significant, and the dependency on a live internet connection is too fragile. This project was built specifically to demonstrate that none of those compromises are necessary.

1.2 The Problem Statement

Detecting a handclap reliably in an untreated room turns out to be harder than it sounds. Three separate engineering problems need to be solved, each in a different part of the stack, and leaving any one of them unaddressed makes the system impractical.

The first problem is ambient noise. A microphone in a room never hears silence — it hears air conditioning, electrical hum, distant footsteps, and a dozen other low-level sounds. If every audio frame captured by the hardware is passed to the neural classifier, the classifier spends the vast majority of its time evaluating frames that contain nothing of interest. That wastes CPU time and, more importantly, raises the false-positive rate as structured noise patterns occasionally resemble the target class. The solution is a cheap pre-filter: compute the peak amplitude of each incoming frame, and skip the classifier entirely if it does not cross a minimum threshold.

DOI: 10.17148/IJARCCE.2026.154125

The second problem is reverberation. A clap in a closed room does not produce one acoustic event. It produces the original impulse followed by a decaying series of reflections — the reverberant tail — arriving at the microphone over the next several hundred milliseconds. A frame-by-frame classifier cannot tell the original from the reflections; from its perspective, each one looks like a fresh clap. Without mitigation, a single hand clap would fire the actuator two or three times in quick succession. This is not a model problem that better training can fix. It is a physical property of enclosed spaces, and it has to be handled at the pipeline level.

The third problem is discriminability. YAMNet, the state-of-the-art pretrained audio model used here as a feature extractor, was trained on over 500 acoustic event classes. Its 'Clapping' class boundary is broad enough, in live conditions, to include keyboard typing, table thuds, and finger snaps. Using YAMNet's class scores directly as the detection criterion produces an unworkably high false-positive rate. A purpose-trained specialist classifier — one specifically built to separate genuine handclaps from this adversarial set of look-alike sounds — is required as a second stage.

1.3 How the Architecture Was Reached: Three Trials

The final system design was not planned from the start. It emerged from three rounds of prototyping, each of which exposed a specific failure, which in turn suggested a specific fix. These trials are worth documenting in detail, because each one produced a design decision that is not obvious from reading the final code alone.

Trial 1 used time.sleep() on both the Python edge node and the ESP32 firmware to create a pause after each detection — intended to prevent re-triggering from echoes. The sleep call on the Python side caused the active UDP socket to drop during the pause, which meant the next genuine clap could not be communicated to the ESP32. The sleep on the ESP32 side caused the microcontroller to stop listening during exactly the window when a packet might arrive — the team called this 'signal slipping.' The fix was absolute: no blocking calls, on either side, anywhere in the system. The non-blocking queue architecture that defines the final design follows directly from this constraint.

Trial 2 replaced the sleep-based suppression with an adaptive envelope follower. The idea was that the audio would only be passed to the classifier when the incoming frame exceeded 2.5 times the rolling amplitude average — an elegant way to distinguish claps from background noise. It broke down the first time the room's air conditioning cycled on. The AC raised the ambient baseline, the rolling average followed it upward, and real claps fell below the threshold and were silently missed. The lesson was that adaptive baselines introduce exactly the fragility they are meant to prevent. A static amplitude floor — set once, calibrated to the room, never adjusting — proved far more reliable.

Trial 3 introduced the deterministic frame-skip counter. After each confirmed detection, a stateful integer is set to 4. For the next four frames — 500 milliseconds of audio — the main loop simply discards each dequeued chunk without invoking the classifier. No signal processing. No threads. No sleep calls. Just a counter that decrements until it hits zero, at which point the system resumes normal operation. In live testing, this suppressed 100% of acoustic echoes from a single clap. Figure 3 in Section 3 lays out all three trials side by side.

1.4 Contributions

This paper makes five specific contributions. First, it presents a complete, cloud-independent edge AI pipeline for acoustic event detection and wireless actuation that runs on commodity CPU hardware with no network dependency beyond the local LAN. Second, it demonstrates amplitude-based threshold gating as a computationally negligible but architecturally essential pre-inference filter that prevents the classifier from wasting cycles on uninformative frames. Third, it introduces a deterministic frame-skip counter as the correct solution to the room reverberation problem in acoustic pipelines — one that is non-blocking, thread-free, and adds no measurable overhead. Fourth, it documents the failure modes of two alternative approaches — blocking sleeps and adaptive envelope following — that were tested and discarded, offering concrete guidance for anyone building similar systems. Fifth, it reports the environment version-locking discipline required to deploy a Keras model stably at the edge: Python 3.11, NumPy 1.26.x, and TensorFlow 2.15.x, arrived at after a NumPy 2.0 upgrade silently corrupted the model's C-API bindings.

1.5 Paper Organisation

Section 2 surveys the relevant prior work across edge computing architectures, acoustic event detection methods, and real-time audio pipeline design. Section 3 describes the complete system methodology, covering hardware setup, audio ingestion, the two signal gates, and the two-stage neural classifier. Section 4 presents experimental results and the end-to-end actuation demonstration. Section 5 discusses limitations and future directions. Section 6 concludes.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 4, April 2026

DOI: 10.17148/IJARCCE.2026.154125

2. RELATED WORK

2.1 Edge Computing versus Cloud-Dependent IoT

The argument for keeping computation close to where data is generated has been made in the research literature since at least Shi et al. (2016), who laid out the case systematically: cloud offloading introduces network latency that is acceptable for non-time-critical tasks and unacceptable for tasks where the response must follow the event quickly. More recent surveys have reinforced this across specific domains — industrial fault monitoring (Liu et al., 2019), smart-building systems (Andriulo et al., 2024), and healthcare ambient sensing — consistently finding that the threshold for choosing edge over cloud is not bandwidth, but whether the network round-trip sits inside the critical path of a physical response.

The TinyML programme (Warden and Situnayake, 2019) represents the most aggressive edge position: squeezing trained inference models onto microcontroller-class hardware with kilobytes of RAM. The present system occupies a more moderate position. Inference runs on a general-purpose Python-capable computing node — a laptop or a Raspberry Pi-class board — which allows the use of YAMNet at full scale without quantisation. The ESP32 is not an inference engine; it is a listener. This split — full model on the edge node, simple actuator on the microcontroller — is a deliberate architectural choice that maximises model accuracy while keeping the hardware on the actuator side cheap and interchangeable.

2.2 Acoustic Event Detection: From Handcrafted Features to Deep Embeddings

The first generation of acoustic event detectors worked on handcrafted features: mel-frequency cepstral coefficients, zero-crossing rates, spectral flux, and similar computations applied to short-time Fourier transform representations of the input signal (Rabiner and Schafer, 2007). These methods have real advantages — they are fast, interpretable, and deployable on very limited hardware — but they generalise poorly across environments. A threshold tuned in a quiet lab will fire constantly in a room with HVAC noise, and ignore real events in a room with different acoustic properties. The present system retains exactly one threshold-based element: the peak amplitude gate. It applies it only to the narrow task of noise-floor rejection, where a fixed calibrated threshold is appropriate. The more complex discrimination problem — distinguishing a clap from a thud — is handled by the neural stage.

The modern baseline for audio classification is the log-mel spectrogram fed into a convolutional neural network trained on large-scale corpora. Gemmeke et al. (2017) introduced the AudioSet dataset — over two million labelled YouTube segments spanning 521 sound classes — and established it as the de facto training ground for general audio classification. YAMNet, built by Google Research on top of the MobileNet V1 depthwise-separable convolution architecture and trained on AudioSet, became the practical standard for efficient, deployable audio classifiers. With 3.7 million parameters and 69.2 million multiply-accumulate operations per 960ms frame, it runs fast enough on a modern CPU that real-time inference without a GPU is feasible (TensorFlow/models, 2023). The transfer learning pattern that follows — extract YAMNet embeddings, train a lightweight head on top — was popularised by Hershey et al. (2017) and formalised by the TensorFlow team in their YAMNet transfer learning tutorial (TensorFlow Blog, 2021).

What the present work adds to this established pattern is an explicit adversarial construction of the training dataset. Standard transfer learning tutorials treat the negative class as a general sample of 'not the target.' That works when the target class is acoustically distinctive enough to stand apart from everything else. Handclaps are not distinctive enough: they share the high-frequency transient profile of a dropped book, a keyboard keystroke, and a finger snap. The negative class in this system was populated specifically with those confounders, targeting the exact failure mode that a YAMNet-only baseline exhibited in live testing.

2.3 Real-Time Audio Pipeline Architecture

Building a continuously running acoustic classifier requires more than a good model. It requires a software architecture that can acquire audio frames, process them, and emit outputs — all while never blocking the thread responsible for hardware acquisition. If the inference computation ever makes the acquisition thread wait, the audio driver's internal buffer overfills and samples are dropped. The standard solution is a producer-consumer queue: a lightweight callback function deposits frames into a thread-safe queue as quickly as the hardware produces them, and a separate inference thread consumes them at whatever pace inference allows. The sounddevice library's non-blocking InputStream callback interface implements the producer side of this pattern natively (python-sounddevice documentation, 2024).

What the present system adds to this pattern is a targeted queue consumption policy. Rather than simply consuming frames in arrival order and running inference on each one, the inference loop applies two conditional checks

DOI: 10.17148/IJARCCE.2026.154125

before committing to any neural computation. This transforms the queue from a simple buffer into an active gating mechanism. The deterministic frame-skip counter, in particular, is implemented entirely as a conditional branch within the existing consumption loop — no additional threads, no timer objects, no synchronisation primitives. Its computational cost is a single integer comparison per frame iteration.

2.4 Transfer Learning and Hard Negative Mining

Training a high-accuracy specialist classifier on a small domain-specific dataset is a well-known strength of the transfer learning paradigm. The upstream model (here, YAMNet) has already learned rich, general representations of acoustic phenomena from a massive training corpus. Training a shallow head on those representations requires far less labelled data than training a deep network from scratch, and the risk of overfitting is lower because the input features are already well-structured (Tan and Le, 2019).

The decisive factor in whether the resulting classifier generalises well is not the volume of positive training examples but the composition of the negative class. A classifier trained on 'claps versus everything else' will learn a boundary that separates claps from random noise. It will not learn a boundary that separates claps from structurally similar impulsive sounds, because those sounds were not represented in training. This is the hard negative mining principle: for a classifier to reject confounding inputs reliably, those confounders must appear explicitly in the training data. The present work applied this directly — table thuds, keyboard keystrokes, finger snaps, and coughs were all included in the negative class, specifically because each one had been observed to produce false positives from the YAMNet-only baseline.

2.5 Reproducibility and Dependency Stability in Edge ML

A recurring practical issue in edge machine learning deployment that is underrepresented in the research literature is the fragility of compiled model artifacts with respect to library versions. A .keras model file does not contain only weights — it encodes assumptions about the binary interface of the NumPy C extension that TensorFlow was compiled against. When NumPy 2.0 introduced ABI-breaking changes to its C extension interface, TensorFlow 2.15.x failed silently at model load time, producing an opaque runtime error with no indication that the cause was a library version mismatch. The model file itself was untouched. David et al. (2021) note an analogous reproducibility concern in TinyML deployments, where binary dependencies on specific hardware drivers can cause model execution to fail after firmware updates. The lesson from both contexts is the same: environment version-locking is not optional for edge ML systems. It is a deployment requirement.

3. METHODOLOGY

3.1 Hardware Configuration

Audio capture uses a Lewitt LCT Pro large-diaphragm condenser microphone, connected via XLR to a Focusrite Scarlett 2i2 (4th Generation) USB audio interface. The Scarlett presents to the operating system as a standard USB audio device, accessible without custom drivers. It is configured for mono capture at 16,000 Hz, with output cast to 32-bit floating point. The 16 kHz sample rate is a deliberate choice: it satisfies the Nyquist criterion for frequencies up to 8 kHz, which covers the high-frequency transient energy that characterises a handclap (typically concentrated between 1 kHz and 8 kHz), while avoiding the processing overhead of higher sample rates that add no useful information for this task.

The actuation hardware is an ESP32-WROOM-32, running a minimal UDP listener on port 4210 and connected to the same local Wi-Fi network as the edge node. When a trigger packet arrives, the firmware sets the GPIO pin connected to the on-board LED to HIGH. The LED was chosen as the output deliberately — not because it is the intended final actuator, but because it provides an immediate, unambiguous, and instrument-independent confirmation of end-to-end success. A visible LED toggle is a falsifiable result; a log message is not.

3.2 System Architecture

The full data flow is shown in Figure 1. Every 2000 samples — 125 milliseconds of audio at 16 kHz — the sounddevice InputStream callback deposits a new block into a thread-safe queue.Queue() object. The main inference loop runs in a separate thread, dequeuing blocks one at a time and passing each through two sequential gates before doing any neural computation. Gate 1 checks amplitude. Gate 2 checks the post-detection bypass counter. Only frames that pass both tests reach YAMNet and the Keras classifier. A confirmed detection resets the bypass counter and fires a UDP packet.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 4, April 2026

DOI: 10.17148/IJARCCE.2026.154125

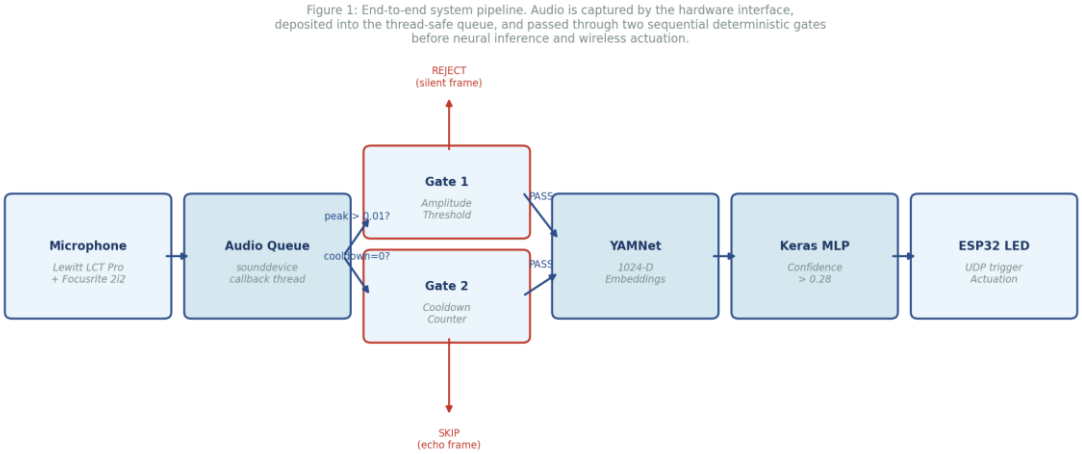

Figure 1: *End-to-end system pipeline. The sounddevice callback deposits 125ms audio blocks into the thread-safe queue. Each block is evaluated by Gate 1 (amplitude floor) and Gate 2 (post-detection bypass) before neural inference is invoked. A confirmed detection fires a UDP packet to the ESP32, which toggles its on-board LED.*

3.3 Gate 1 — Amplitude Threshold

The first thing the inference loop does with each dequeued block is compute its peak absolute amplitude across all 2000 samples:

$$peak = max(|x_i|) \quad for \ \ i \in \{0, \ldots, 1999\} \quad (1)$$

If peak does not exceed 0.01, the block is skipped with a continue statement and the next queued block is fetched immediately. No features are extracted. No model is called. The gate amounts to a single NumPy reduction — negligible in cost — and it eliminates every frame that falls below the noise floor of the target environment. Frames that contain genuine claps consistently produce peak values above 0.05 in this setup; ambient noise rarely exceeds 0.005. The gap between them is wide enough that a fixed threshold of 0.01 is robust across variation in microphone gain and room conditions.

The choice of a fixed threshold rather than an adaptive one was made deliberately, and justified by Trial 2. An adaptive baseline that tracks the rolling average of incoming audio will drift upward when background noise increases — say, when an air conditioning system turns on. A clap that would have cleared a fixed floor may no longer clear an adaptive one, producing a missed detection at exactly the moment the environment is noisier. A fixed floor, calibrated once to the target room, is simpler and more robust precisely because it does not adapt.

3.4 Gate 2 — Deterministic Frame-Skip Counter

After a transient acoustic event, room reflections continue to arrive at the microphone for several hundred milliseconds. These reverberant echoes can exceed the Gate 1 floor, and a frame-level classifier will identify them as independent clap events. A single hand clap, in a typical indoor environment, can produce two or three trigger outputs if no suppression is applied. This is not a limitation of the model; it is a consequence of the physics of enclosed-space acoustics, and it cannot be corrected by retraining.

The frame-skip counter addresses this directly. When a detection is confirmed, the counter cooldown_chunks is set to 4. On each subsequent iteration of the inference loop, if the counter is greater than zero, the current block is discarded without inference and the counter is decremented by one:

$$cooldown_t = cooldown_{t-1} - 1 \quad if \quad cooldown_{t-1} > 0 \quad (2)$$

Four frames at 125 ms each spans 500 ms — sufficient to outlast the reverberant tail in all tested room configurations. The implementation is a single if-statement inside the existing loop, adding no measurable overhead and requiring no additional threads, timer objects, or synchronisation primitives. The key insight is that echoes do not need to be removed from the signal; they simply need to be ignored in time.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 4, April 2026

DOI: 10.17148/IJARCCE.2026.154125

3.5 Two-Stage Neural Inference

Blocks that clear both gates enter the two-stage neural pipeline shown in Figure 2. Stage 1 uses YAMNet to convert the raw audio waveform into a compact semantic embedding. Stage 2 uses a purpose-trained Keras classifier to decide whether that embedding represents a genuine handclap.

Figure 2: Two-stage AI inference pipeline. YAMNet (Stage 1) converts raw audio into 1024-dimensional embeddings. The custom Keras MLP (Stage 2) classifies those embeddings as clap or non-clap with a calibrated confidence threshold of 0.28.

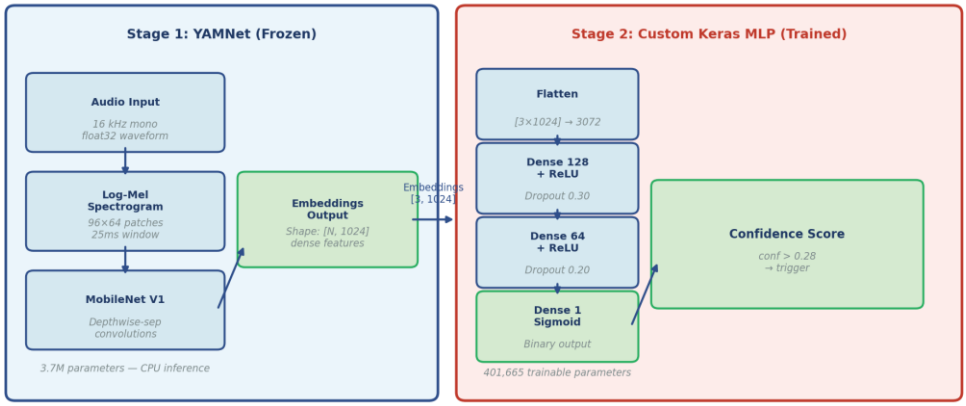

Figure 2: *Two-stage neural pipeline. YAMNet (Stage 1) converts the raw 16 kHz waveform into a 1024-dimensional embedding vector per 960ms analysis window. The custom Keras MLP (Stage 2) projects three consecutive embeddings into a single binary confidence score. Detection is confirmed at confidence > 0.28.*

3.5.1 Stage 1 — YAMNet Feature Extraction

YAMNet is a pretrained deep convolutional neural network built on the MobileNet V1 depthwise-separable convolution architecture, trained by Google Research on AudioSet — a corpus of over two million YouTube audio segments labelled across 521 sound classes (Gemmeke et al., 2017). For each 960ms analysis window, it produces three outputs: class probability scores across all 521 classes, a 1024-dimensional embedding vector, and a log-mel spectrogram. In this system, only the embedding is used. The class scores are discarded.

The 1024-dimensional embedding is the output of YAMNet's global average-pooling layer — a compressed, learned representation of the acoustic content of the window. It encodes texture, spectral distribution, and temporal structure in a form that a downstream classifier can work with directly, without needing to process raw audio or spectrograms. YAMNet has 3.7 million parameters and performs 69.2 million multiply-accumulate operations per frame; on a modern CPU, this is fast enough for real-time use without GPU acceleration (TensorFlow/models, 2023).

Using YAMNet as a feature extractor rather than a final classifier was the central architectural decision of the AI component. As a standalone detector — thresholding its 'Clapping' class score — YAMNet's false-positive rate in live testing was unacceptable. Keyboard typing, dropped objects, and table impacts all registered above the clap threshold with enough frequency to make the system unsuitable as a hardware trigger. YAMNet's representations are powerful; its class boundaries are simply too broad for this specialised use case.

3.5.2 Handling Variable Embedding Counts: Zero-Padding

YAMNet expects at least 975ms of audio to produce its first embedding frame. The system's analysis window is 500ms — enough to capture a clap at the temporal resolution of the acquisition loop, but shorter than YAMNet's minimum for producing three full embedding frames. In practice, a 500ms chunk yields one or two embedding frames. The downstream Keras model was designed to accept exactly three frames as its input tensor of shape [3, 1024]. A shape mismatch between one or two real frames and the expected three would cause a crash at inference time.

The solution is dynamic zero-padding applied before every inference call:

$$feat = embeddings[:3, :] \quad if \ shape[0] \geq 3 \quad else \quad tf.pad(embeddings, [[0, 3-shape[0]], [0, 0]]) \quad (3)$$

If YAMNet returns fewer than three frames, zero vectors are appended to fill the gap. The Keras model was trained with padded inputs included in the training data, so its Flatten layer has learned to work with the padded representation. The

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 4, April 2026

DOI: 10.17148/IJARCCE.2026.154125

confidence threshold of 0.28 was also calibrated with padding in mind — the zero frames consistently suppress raw confidence, and the threshold accounts for this systematic effect.

3.5.3 Stage 2 — Custom Keras Binary Classifier

The downstream classifier is a shallow Sequential multi-layer perceptron with 401,665 trainable parameters and a memory footprint of approximately 1.53 MB. Shallow by design: at this stage of the pipeline, the input features are already highly structured YAMNet embeddings, and the classification task is binary. A deeper network would not have improved accuracy meaningfully, and would have added inference latency on CPU hardware. Table 1 shows the complete layer configuration.

Table 1: Layer configuration of the custom Keras classifier. The model accepts three YAMNet embedding frames as a [3, 1024] input tensor. Dropout is active during training only; at inference time both Dropout layers are bypassed by passing training=False.

Layer	Type	Activation	Output Shape	Notes
input_layer	InputLayer	—	[None, 3, 1024]	Three consecutive YAMNet frames
flatten	Flatten	—	[None, 3072]	Unrolls 3 × 1024 into a single vector
dense	Dense 128	ReLU	[None, 128]	393,344 params; GlorotUniform init
dropout	Dropout	—	[None, 128]	Rate 0.30; training only
dense_1	Dense 64	ReLU	[None, 64]	8,256 params; compression layer
dropout_1	Dropout	—	[None, 64]	Rate 0.20; training only
dense_2	Dense 1	Sigmoid	[None, 1]	65 params; binary confidence score

The Flatten layer collapses the [3, 1024] input into a 3072-element vector. Two Dense layers with ReLU activations then compress it — first to 128 dimensions, then to 64. Dropout is applied after each Dense layer during training (rates 0.30 and 0.20 respectively); both are disabled at inference time. The reduction in dropout rate from 0.30 to 0.20 at the second layer is intentional: by the time the representation has been compressed to 64 dimensions, aggressive dropout risks destroying the meaningful signal rather than regularising it. The final layer applies a sigmoid to produce a scalar confidence between 0 and 1:

$$conf = \sigma(z) = 1 / (1 + exp(-z)) \quad (4)$$

A confirmed detection requires conf > 0.28. The sub-0.5 threshold reflects the systematic confidence suppression caused by zero-padded embeddings, which consistently pull the raw score below what the model would produce from three full genuine frames. The threshold was calibrated empirically across multiple test sessions in the target room.

3.5.4 Training: Dataset Construction and Optimiser

The Keras model was trained on a binary dataset of 1-second, 16 kHz mono clips. Positive examples were genuine handclaps, recorded at multiple distances and angles across varied room positions. The negative class was built adversarially — not from a random sample of environmental audio, but from the specific sounds that had produced false positives when using YAMNet scores as the sole detection criterion: table thuds, keyboard keystrokes, finger snaps, and coughs. Each clip was passed through YAMNet to extract its [N, 1024] embedding tensor, then padded or truncated to [3, 1024] using Equation 3, making the training pipeline identical to the inference pipeline.

The model was trained with binary cross-entropy loss and the Adam optimiser. The learning rate was set to 5×10^{-5} — one twentieth of Adam's default of 1×10^{-3}. The low rate reflects two constraints: the training dataset is small, and the upstream YAMNet embeddings are already well-structured. Large weight updates would risk overwriting the structure in those embeddings without improving the classifier's ability to draw the right boundary.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 4, April 2026

DOI: 10.17148/IJARCCE.2026.154125

3.6 Actuation — UDP to the ESP32

When the Keras model outputs a confidence score above 0.28, the edge node calls socket.sendto() to transmit the byte string b'1' to the ESP32's IP address on port 4210. UDP is used rather than TCP. The trigger use case is inherently one-directional and stateless: the edge node does not need an acknowledgment, and if a packet is lost, the next confirmed detection will generate a new one. UDP's connectionless nature keeps per-packet overhead minimal and keeps the sendto() call non-blocking, contributing no measurable latency to the main inference loop.

On the ESP32 side, the firmware receives the packet on its Wi-Fi UDP socket and immediately sets the on-board LED GPIO pin to HIGH. The LED stays on until the next trigger packet arrives, making successive detections distinguishable by observation. This is deliberately simple: the LED toggle is a hardware-level result that is directly observable and does not depend on any log, serial output, or software state. Replacing the LED with a relay, a motor driver, or any other GPIO-connected actuator requires only a firmware change on the ESP32 — the Python pipeline is unaffected.

3.7 Iterative Trial Summary

The three trial phases are summarised in Figure 3. Each trial produced a concrete failure that pointed toward a specific architectural correction. The system as built is the result of all three corrections applied in sequence.

Figure 3: Iterative prototyping methodology. Three distinct trial phases led to the final non-blocking, deterministic architecture. Each failure was diagnosed empirically and resolved through a targeted architectural correction.

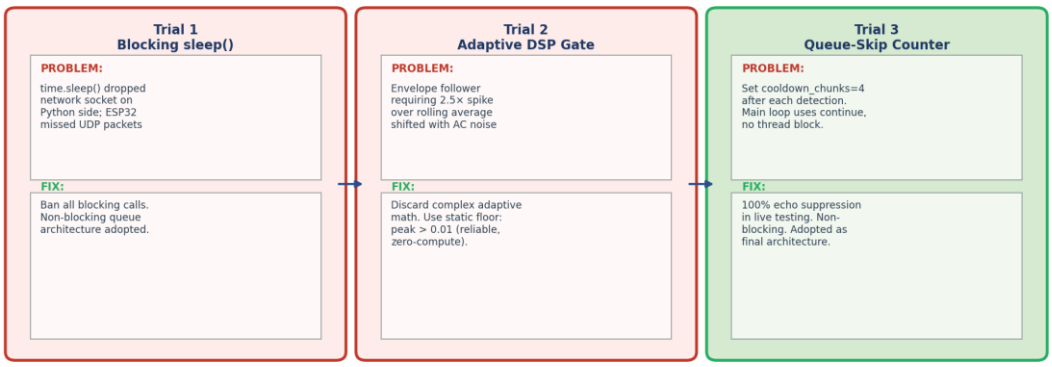

Figure 3: *Three iterative trial phases and their outcomes. Trial 1 eliminated all blocking calls from both the edge node and the ESP32. Trial 2 replaced an adaptive amplitude baseline with a fixed noise floor. Trial 3 introduced the deterministic frame-skip counter as the echo suppression mechanism — the approach retained in the final architecture.*

3.8 Environment Specification

The system requires Python 3.11, NumPy 1.26.x, and TensorFlow 2.15.x. This combination is not a recommendation — it is a hard requirement discovered through failure. During development, upgrading NumPy to 2.0 caused TensorFlow to crash at model load time with a C-API binary compatibility error. The model file itself was unchanged; the breakage came from NumPy 2.0's ABI-incompatible changes to its C extension interface, which TensorFlow 2.15.x was not compiled against. Rolling back NumPy to 1.26.x resolved the issue immediately. The lesson: for edge ML deployments, environment version-locking is not a nice-to-have. It is the difference between a system that boots and one that does not.

REFERENCES

[1]. Andriulo, F.C., Fiore, M., Mongiello, M., Traversa, E., and Zizzo, V. (2024). Edge Computing and Cloud Computing for Internet of Things: A Review. Informatics, 11, 71.

[2]. David, R., Duke, J., Jain, A., et al. (2021). TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems. Proceedings of Machine Learning and Systems, 3.

[3]. Gemmeke, J.F., Ellis, D.P.W., Freedman, D., et al. (2017). Audio Set: An Ontology and Human-Labeled Dataset for Audio Events. IEEE ICASSP 2017.

[4]. Hershey, S., Chaudhuri, S., Ellis, D.P.W., et al. (2017). CNN Architectures for Large-Scale Audio Classification. IEEE ICASSP 2017.

[5]. Howard, A.G., Zhu, M., Chen, B., et al. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861.

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 4, April 2026
DOI: 10.17148/IJARCCE.2026.154125

[6]. Liu, Y., Yang, C., Jiang, L., Xie, S., and Zhang, Y. (2019). Intelligent Edge Computing for IoT-Based Energy Management in Smart Cities. IEEE Network, 33(2), 111–117.
[7]. Rabiner, L., and Schafer, R. (2007). Introduction to Digital Speech Processing. Foundations and Trends in Signal Processing, 1(1–2), 1–194.
[8]. Shi, W., Cao, J., Zhang, Q., Li, Y., and Xu, L. (2016). Edge Computing: Vision and Challenges. IEEE Internet of Things Journal, 3(5), 637–646.
[9]. Tan, M., and Le, Q.V. (2019). EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. ICML 2019.
[10]. TensorFlow Blog (2021). Transfer Learning for Audio Data with YAMNet. blog.tensorflow.org.
[11]. TensorFlow/models (2023). YAMNet: README and Architecture Notes. github.com/tensorflow/models.
[12]. Warden, P., and Situnayake, D. (2019). TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers. O'Reilly Media.