TinyML at the Edge: A Whitepaper on Predictive Maintenance and Environmental Monitoring with ESP32 and C

B.Vignesh Kumar, <vigneshkumarb@bravetux.com> <ic19939@gmail.com>

1. Executive Summary: The Imperative of On-Device Intelligence

The proliferation of Internet of Things (IoT) devices in industrial and environmental sectors has generated immense volumes of data. However, the traditional approach of transmitting all raw sensor data to the cloud for analysis is fraught with challenges, including high latency, security vulnerabilities, and significant power consumption.[1] Tiny Machine Learning (TinyML) represents a paradigm shift, bringing the power of artificial intelligence to the most resource-constrained devices at the network edge.[3] This report examines the strategic value of this approach, focusing on two compelling use cases: predictive maintenance and environmental monitoring. It details the end-to-end workflow, from data acquisition and on-device optimization to deployment on the ESP32 family of microcontrollers, a platform uniquely suited for this task due to its low-power architecture and growing support for AI acceleration.[5] The analysis further explores the complexities of using the C programming language, revealing a pragmatic hybrid approach that leverages the best aspects of both C and C++ toolchains. The findings demonstrate that by combining the strengths of TinyML, the ESP32, and careful programming practices, it is possible to create a new class of intelligent, autonomous, and highly efficient edge devices.

2. The Convergence of TinyML and the ESP32 Ecosystem

2.1 Defining TinyML: Beyond the Cloud

TinyML is a specialized subset of machine learning dedicated to the deployment of models on microcontrollers and other ultra-low-power edge devices.[3] It contrasts with traditional cloud-based AI, which relies on vast computational resources and internet connectivity.[7] By processing data directly on the device, TinyML offers several critical advantages.[1] Firstly, it provides real-time processing and ultra-low latency, which is essential for time-critical applications like industrial monitoring or safety-critical systems.[2] Secondly, it significantly enhances data privacy and security by analyzing sensitive sensor data locally, eliminating the need to transmit it over a network.[2] Lastly, its ultra-low power consumption, often operating in the milliwatt range or below, makes it an ideal solution for long-term, battery-powered deployments.[1] While the broader field of Embedded AI can encompass powerful edge servers, TinyML specifically targets devices at the smallest end of the spectrum, which are often battery-powered.[3]

2.2 A Microcontroller for the Edge: The ESP32 Family

The ESP32 is a versatile system-on-a-chip (SoC) that has become a popular choice for TinyML applications due to its compelling feature set.[9] The original ESP32 features a dual-core Xtensa LX6 CPU operating at up to 240 MHz, along with 520 KiB of RAM and 448 KiB of ROM.[5] It integrates Wi-Fi and Bluetooth connectivity, numerous peripheral interfaces, and robust power management capabilities, including a deep sleep mode with a current draw of only 5 µA.[5]

Espressif's product line has evolved with the specific demands of AI and machine learning workloads in mind. The ESP32-S3, for instance, is an enhanced variant featuring a dual-core Xtensa LX7 CPU that includes dedicated vector instructions to accelerate machine learning applications and signal processing.[5] It offers 512 KiB of SRAM and the ability to connect to external PSRAM, directly addressing the memory limitations of its predecessor.[5] The ESP32-C3 is a more cost-effective option, built around a single-core 32-bit RISC-V CPU.[5] While less powerful, its open-source Instruction Set Architecture (ISA) eliminates licensing costs, making it suitable for price-sensitive projects.[6] The ESP32-P4 represents a significant leap forward, powered by a dual-core RISC-V CPU with custom AI and vector instructions.[5] It

is designed for high-performance applications such as rich Human-Machine Interfaces (HMI) and complex edge computing tasks. Unlike other variants, it does not include on-chip Wi-Fi or Bluetooth, instead functioning as a host for companion wireless chips.[5] The tiered product line, from the cost-sensitive ESP32-C3 to the high-performance ESP32-P4, allows developers to select a device that perfectly balances price and performance for their specific TinyML application. This evolution from a general-purpose SoC to one with specialized AI hardware demonstrates a direct response to the increasing computational demands of on-device intelligence.

The causal relationship between the evolution of the ESP32 family and the requirements of TinyML is evident. Early microcontrollers were capable of running only the simplest models, often requiring significant compromises. The addition of specialized instructions in the ESP32-S3 and the custom AI hardware in the ESP32-P4 fundamentally alters this. It signals a move from a software-only optimization paradigm to one of hardware-software co-design.[12] This allows engineers to deploy more sophisticated models without a proportional increase in power consumption, making previously infeasible applications viable.

ESP32 Family Comparison for TinyML
Feature
CPU Core(s)
Clock Speed
On-chip RAM
AI Hardware Acceleration
Wireless Connectivity
Target Application

2.3 Hardware Acceleration for Edge AI: The Rise of Specialized Cores

The ability of modern ESP32 variants to perform machine learning inference efficiently is largely due to their specialized hardware. The ESP32-S3, for example, integrates SIMD (Single

Instruction, Multiple Data) vector instructions.[10] In contrast to a standard CPU that processes one data element at a time, SIMD enables a single instruction to operate on multiple data elements simultaneously.[13] This significantly accelerates core AI operations, such as the dot product, which is a foundational component of neural network computations.

Espressif provides its own optimized libraries, ESP-DSP and ESP-NN, to allow developers to take advantage of these vector instructions.[10] This integration with software frameworks is crucial, as evidenced by Edge Impulse, which has incorporated ESP-DSP acceleration into its ESP32 deployments.[14] This integration has been shown to provide a speed-up of approximately 5-6x for computationally intensive tasks like MFCC (Mel-frequency cepstral coefficients) feature extraction for audio analysis.[14] The ESP32-P4 extends this concept further with custom AI/vector instructions and hardware accelerators for tasks such as image and video encoding.[5] The inclusion of these dedicated components marks a shift in embedded systems design, where processors are no longer just general-purpose but are specifically tailored to the unique demands of AI workloads.

2.4 The C/C++ Paradox: A Foundational Choice

The report's focus on the C programming language for this application requires a careful examination of its role in a domain largely dominated by C++ toolchains. C is a cornerstone of embedded development, prized for its efficiency, deterministic performance, and direct control over hardware.[15] It is the language of choice for many low-level device drivers and real-time systems, and its backward compatibility with existing C libraries is a significant advantage for many projects.[15]

However, the TinyML ecosystem, particularly frameworks like TensorFlow Lite for Microcontrollers (TFLM) and platforms like Edge Impulse, is built primarily on C++.[9] Modern C++ offers powerful abstractions that streamline development, such as Object-Oriented Programming (OOP) for modularity and a rich Standard Template Library (STL) for optimized data structures and algorithms.[15]

This presents a seeming paradox: to build an efficient, professional-grade TinyML application, an engineer must reconcile the low-level control of C with the high-level convenience of C++. The resolution lies in a hybrid development approach. The C++-based TinyML model, typically exported as a C++ library or a C header file containing a byte array of the model [9], can be treated as a component within a larger C-based application.[9] The core device logic—including sensor data acquisition, hardware-level optimizations, and power management routines—can be written in C, while the model inference is handled by the C++ library. This pragmatic approach leverages the strengths of both languages: C for its resource efficiency and direct

hardware control and C++ for its powerful, pre-optimized AI frameworks.

The use of C++ in this context, while convenient, introduces potential complexities. For example, C++ can result in slower compile times and a larger binary size if not carefully managed, especially with heavy use of templates.[15] In a resource-constrained environment like the ESP32, where every byte of memory matters, these trade-offs are not theoretical but critical to project success.[9] The decision to use C, as specified in the query, highlights a nuanced understanding of these constraints. The most effective approach is a careful blend of both languages, using the C++ toolchain for the highly optimized AI inference engine and C for the application's core logic.

C vs. C++ for Embedded TinyML
Criteria
Key Strengths
Key Weaknesses
Memory Management
Suitability for TinyML
Recommendation

3. The End-to-End TinyML Workflow: From Data to Deployment

3.1 Data Acquisition and Preprocessing: The On-Device Funnel

The TinyML development process begins not with model training, but with a clear problem definition and a comprehensive plan for data acquisition.[7] Raw sensor data is often unsuitable for direct inference.[1] For TinyML applications, preprocessing must occur on the device itself to

minimize the amount of data that needs to be stored or processed by the model, thereby conserving memory and power.[1]

For time-series data, a common format in both predictive maintenance and environmental monitoring, several preprocessing techniques are essential. Data must first be normalized to a standard range to improve model performance.[20] Noise reduction techniques, such as data smoothing using a Simple Moving Average (SMA), can eliminate random fluctuations and reveal underlying patterns.[20] A more advanced method for multi-sensor systems involves projecting a sensor's signal onto its neighbors to suppress sensor-specific noise or "glitches" while preserving the signal of interest.[21]

The most critical step, however, is on-device feature extraction. Instead of feeding a neural network raw, noisy data, it is more efficient to extract meaningful features that a model can easily learn. For vibration analysis, a Fast Fourier Transform (FFT) converts time-domain sensor readings into the frequency domain.[22] The model then analyzes this frequency spectrum, where anomalies are often more apparent, drastically reducing the model's computational burden.[22] The ESP32-S3's dedicated ESP-DSP library is specifically designed to accelerate these computationally intensive digital signal processing (DSP) tasks, making on-device FFT a practical reality.[10] The practice of curating and processing data at the edge transforms the development paradigm from simply collecting raw data to actively extracting insights on the device itself, reducing network latency, power consumption, and data privacy risks.[1]

3.2 Model Design and Training: Tailoring AI for the Constrained Edge

The choice of model architecture for a TinyML application is dictated by the problem and the available computational resources.[1] For the use cases of predictive maintenance and environmental monitoring, a distinction must be made between supervised and unsupervised learning approaches. Supervised learning requires a dataset with labeled examples of both normal and abnormal conditions.[24] This is ideal for classification tasks where the goal is to identify a specific known fault.[25]

However, in many real-world scenarios, it is difficult to collect a sufficient number of data points for all possible failure or anomaly events.[24] This is where unsupervised learning becomes particularly valuable. Anomaly detection models are trained exclusively on data representing "normal" behavior. The model learns to recognize this pattern and then flags any new data that significantly deviates from it.[24] For this purpose, autoencoders are a highly effective neural network architecture.[24] An autoencoder consists of an encoder that compresses input data into a smaller representation and a decoder that reconstructs the data

from that compressed form.[24] During inference, the reconstruction quality serves as an anomaly score; a high error indicates that the input data does not conform to the learned pattern and is therefore anomalous.[24]

3.3 Model Optimization for Constraints: The Art of the Trade-Off

Given the ESP32's limited memory, a trained model is too large for on-device deployment without significant optimization.[1] The most critical optimization technique is quantization, which reduces the memory footprint and computational requirements of a model.[3] This process involves converting model parameters and activations from 32-bit floating-point numbers to lower-precision integers, such as 8-bit or even 4-bit values.[3] This drastically reduces the model's size and enables faster, more energy-efficient integer-based operations on the microcontroller.[3] While uniform quantization is the most common approach due to its compatibility with hardware accelerators, other methods like non-uniform quantization can offer higher accuracy in low-precision regimes.[29]

Another essential technique is pruning, which removes redundant or less relevant connections from a neural network.[3] This reduces both model size and computational complexity, often with minimal impact on accuracy.[3] Structured pruning, which removes entire network structures, is particularly beneficial for embedded systems because it preserves the regular network structure, which is more efficient for hardware implementation.[26]

The process of optimization is an iterative one, balancing model size, inference speed, and accuracy.[12] While quantization can yield up to a 90% size reduction and a 3.5x speed increase, this may come at the cost of a slight drop in accuracy.[30] The ESP32’s limited 520 KiB of SRAM is the primary constraint that necessitates these aggressive optimization techniques.[9] The goal is to find the optimal point where the model is small and fast enough to run on the target hardware while retaining sufficient performance for the application.

3.4 The Deployment Toolchain: Bridging the Gap

The final step in the workflow is deploying the optimized model onto the ESP32. This is facilitated by specialized software toolchains that bridge the gap between model development on a PC and on-device execution. TensorFlow Lite for Microcontrollers (TFLM) is a key framework in this process.[2] It is a lightweight C++ library designed to run on devices with

a few kilobytes of RAM. The workflow involves training a model in TensorFlow, converting it to the

.tflite format, and then using a tool like xxd to convert the .tflite file into a C array.[9] This array is then included in the ESP32's firmware, where the TFLM interpreter can load and execute the model.[9]

Edge Impulse is a web-based platform that automates much of this process, from data collection to model deployment.[8] It provides a user-friendly interface to build and train models and then exports a complete C++ or Arduino library for the ESP32.[14] Edge Impulse has a symbiotic relationship with Espressif's hardware, as it automatically includes and leverages ESP-DSP acceleration for faster digital signal processing on compatible ESP32 variants.[14] In addition to these frameworks, C code can be further optimized at the compiler level by setting the optimization flag to

-O2 and by moving frequently executed code to Instruction RAM (IRAM) to prevent performance bottlenecks caused by cache misses.[28]

4. Case Study 1: TinyML for Predictive Maintenance

4.1 The Business Case for PdM

Predictive maintenance (PdM) is a highly valuable application of TinyML in industrial environments.[4] It moves beyond traditional reactive maintenance, where equipment is repaired after a failure, and preventive maintenance, which relies on fixed schedules and can be wasteful.[1] By using sensors to monitor the real-time condition of equipment, PdM models can predict when a failure is likely to occur.[1] This allows for condition-based maintenance, preventing unexpected downtime and significantly reducing repair and replacement costs.[4] TinyML is a game-changer in this field because it enables real-time anomaly detection directly on the sensor device, which is crucial for applications that require immediate feedback and cannot afford the latency of cloud-based systems.[31]

4.2 Sensor Selection and Data Modalities

A TinyML-based PdM system relies on data from sensors that can provide insight into a machine's operational health.[35] Vibration and temperature are two of the most critical indicators.[33] Vibration sensors, often piezoelectric, measure changes in vibration patterns that can signal bearing issues, misalignment, or loose parts.[33] Temperature sensors can detect abnormal heat buildup, a common sign of impending failure. For a TinyML system to work, these sensors must be interfaced with the ESP32 using appropriate protocols, such as I^2C or SPI, or through its analog-to-digital converter (ADC).[36]

A crucial component of this application is on-device signal processing. The raw time-series data from a vibration sensor is less valuable than its frequency spectrum.[22] The ESP32 can perform a Fast Fourier Transform (FFT) on the raw data to convert it into the frequency domain, a computationally intensive task that is accelerated by the ESP32-S3's DSP instructions.[22] The resulting frequency spectrum can then be fed into a lightweight model to identify anomalous patterns. The ability to perform this feature extraction on the device itself is critical, as it reduces the data payload and makes real-time inference possible.

The TinyML approach also facilitates the creation of "set and forget" industrial monitoring systems. Its low-power design allows devices to operate for extended periods on battery or solar power, reducing the maintenance overhead for the sensors themselves.[27] This enables their deployment in remote or difficult-to-reach locations, expanding the reach of predictive maintenance beyond traditional wired factory settings.[33]

Predictive Maintenance Sensor Matrix
Sensor Type
Vibration
Temperature
Pressure
Acoustic

5. Case Study 2: TinyML for Environmental Monitoring

5.1 On-Device Sensing for a Safer World

TinyML for environmental monitoring enables real-time, localized detection of critical conditions.[4] This approach moves beyond simple threshold-based alarms, which can be prone to false positives, to intelligently analyze complex data patterns.[24] It is particularly valuable for detecting gas leaks, air quality degradation, and other environmental anomalies.[4] By processing data on-device, these systems offer crucial advantages: low latency for immediate alerts, enhanced data privacy, and the ability to operate autonomously in remote areas without a network connection.[8]

5.2 Sensor Suite and System Integration

An effective environmental monitoring system often employs a suite of sensors to provide a comprehensive picture of the environment.[36] A typical setup might include gas sensors to detect smoke or hazardous compounds, temperature and humidity sensors, and light or soil moisture sensors.[36] The ESP32 serves as the central processing unit, managing data from these various sensors through its GPIO, ADC, or I²C pins.[36] A key benefit of TinyML in this context is the ability to perform sensor fusion, combining data from multiple sensors to achieve more robust and accurate predictions.[37] For instance, a model can learn to correlate temperature and humidity readings with gas sensor data to confirm a gas leak and reduce false alarms.[24]

Environmental Monitoring Sensor Suite
Sensor Type
Gas / Air Quality
Temperature / Humidity
Soil Moisture

Water Level
Light Intensity

5.3 Model Implementation

Unsupervised anomaly detection is an ideal approach for environmental monitoring.[24] A model, such as an autoencoder, is trained solely on data collected during "normal" environmental conditions.[24] This allows the system to learn the complex, natural patterns of the environment, such as the daily rise and fall of temperature or the fluctuation of humidity.[24] During deployment, the ESP32's firmware continuously feeds new sensor readings through the model. If the resulting reconstruction error exceeds a pre-defined threshold, it flags the data as anomalous, triggering an alert via a local buzzer, an LED, or a notification over Wi-Fi or BLE.[8]

5.4 Addressing the Challenges of Drift and Calibration

A significant challenge with low-cost environmental sensors is long-term drift, where sensor readings gradually become inaccurate over time due to environmental stress, aging, or thermal effects.[40] This makes traditional, one-time factory calibration insufficient for long-term deployments.[40]

The ESP32's on-device intelligence offers a solution to this problem. Instead of manual recalibration, a TinyML model can be used to continuously learn and compensate for sensor drift by correlating it with other stable readings or analyzing the data’s temporal patterns.[40] This approach leverages the universal approximation capabilities of machine learning to create a data-driven solution that adapts to the sensor's changing behavior over its lifetime.[40] One proposed technique uses a Temporal Convolutional Neural Network (TCNN) with a fast Hadamard transform, which is computationally lightweight and can effectively separate slowly-varying drift components from the actual signal.[41] This advancement in sensor technology, enabled by on-device intelligence, allows for the creation of self-calibrating and intelligent sensors that can operate autonomously for extended periods, solving a major barrier to the widespread adoption of low-cost sensor networks.

6. The Path Forward: Unlocking New Potential

The analysis of TinyML on the ESP32 reveals a complex interplay between hardware, software, and programming practice. The central challenge remains a critical trade-off between model size, inference speed, and accuracy, all of which are constrained by the ESP32's limited SRAM.[9] A pragmatic, hybrid C/C++ development approach is the most effective way to navigate this landscape, allowing engineers to leverage powerful C++-based AI toolchains while maintaining the low-level control and efficiency of C for core application logic.[9]

The field is poised for continued growth as new hardware and software trends emerge. The ESP32-P4, with its custom AI and vector instructions, points toward a future where microcontrollers are no longer general-purpose but are purpose-built for AI workloads.[5] The software ecosystem is also evolving, with automated model optimization tools and more efficient neural network operations that simplify the development process and enable more complex applications, such as real-time computer vision for quality control and low-power audio analysis for security.[32]

7. Conclusion: The Foundation of an Intelligent Edge

This report demonstrates that TinyML, deployed on the ESP32 family of microcontrollers, is a mature and viable technology stack for high-impact applications in predictive maintenance and environmental monitoring. The combination of hardware evolution—particularly the introduction of AI acceleration in variants like the ESP32-S3 and ESP32-P4—a robust software ecosystem anchored by frameworks like TensorFlow Lite for Microcontrollers and Edge Impulse, and a pragmatic development methodology that blends the strengths of C and C++ provides a powerful foundation. By moving processing to the edge, these systems unlock critical advantages in latency, privacy, and energy efficiency. They also enable new capabilities, such as the "set and forget" industrial sensor and the self-calibrating environmental monitor, that were previously unachievable. The ESP32 is a central component in the creation of a new generation of intelligent, autonomous, and efficient edge devices that are poised to transform industries and enhance safety.

Works cited

1. A Primer for tinyML Predictive Maintenance: Input and ... - DTU Orbit, accessed on September 7, 2025,

https://orbit.dtu.dk/files/274322308/A_Primer_for_tinyML_Predictive_Maintenance_Input_and_Model_Optimisation.pdf
2. Introduction to TinyML on Microcontrollers: Bringing AI to the Edge - ThinkRobotics.com, accessed on September 7, 2025, https://thinkrobotics.com/blogs/learn/introduction-to-tinyml-on-microcontrollers-bringing-ai-to-the-edge
3. tinyML - MATLAB & Simulink - MathWorks, accessed on September 7, 2025, https://www.mathworks.com/discovery/tinyml.html
4. What is TinyML? Tiny Machine Learning - GeeksforGeeks, accessed on September 7, 2025, https://www.geeksforgeeks.org/machine-learning/what-is-tinyml-tiny-machine-learning/
5. ESP32 - Wikipedia, accessed on September 7, 2025, https://en.wikipedia.org/wiki/ESP32
6. A Guide for the ESP32 Microcontroller Series - DigiKey, accessed on September 7, 2025, https://www.digikey.com/en/maker/blogs/2024/a-guide-for-the-esp32-microcontroller-series
7. TinyML: A Game-Changer in Edge AI with On-Device Use Cases | by Kim Boher | Medium, accessed on September 7, 2025, https://medium.com/@kimber.labs0101/tinyml-a-game-changer-in-edge-ai-and-use-cases-for-on-device-ai-02b1b58e47ab
8. A TinyML-based system for gas leakage detection | Request PDF - ResearchGate, accessed on September 7, 2025, https://www.researchgate.net/publication/361986779_A_TinyML-based_system_for_gas_leakage_detection
9. TensorFlow Lite On ESP32 – OpenELAB Technology Ltd., accessed on September 7, 2025, https://openelab.io/blogs/learn/tensorflow-lite-on-esp32
10. ESP32-S3 Wi-Fi & BLE 5 SoC | Espressif Systems, accessed on September 7, 2025, https://www.espressif.com/en/products/socs/esp32-s3
11. ESP32-P4 High-performance SoC - Espressif Systems, accessed on September 7, 2025, https://www.espressif.com/en/products/socs/esp32-p4
12. A Workflow For TinyML Development - Cadence System Analysis, accessed on September 7, 2025, https://resources.system-analysis.cadence.com/blog/a-workflow-for-tinyml-development
13. ESP32-S3 for Edge AI | SIMD Vector Instructions Boost Dot Product Performance, accessed on September 7, 2025, https://saludpcb.com/esp32-s3-edge-ai-simd/
14. Espressif ESP-EYE - Edge Impulse Documentation, accessed on September 7, 2025, https://docs.edgeimpulse.com/docs/edge-ai-hardware/mcu/espressif-esp32
15. C++ for embedded development: Pros and cons - Incredibuild, accessed on September 7, 2025, https://www.incredibuild.com/blog/c-for-embedded-development-pros-and-cons

16. edgeimpulse/example-standalone-inferencing-espressif-esp32: Builds and runs an exported impulse locally (ESP IDF) - GitHub, accessed on September 7, 2025, https://github.com/edgeimpulse/example-standalone-inferencing-espressif-esp32
17. C++ for Embedded: Advantages, Disadvantages, and Myths - Qt, accessed on September 7, 2025, https://www.qt.io/blog/c-for-embedded-advantages-disadvantages-and-myths
18. Complete Guide To TinyML Deployments: Optimize Machine Learning For Microcontrollers, accessed on September 7, 2025, https://troylendman.com/complete-guide-to-tinyml-deployments-optimize-machine-learning-for-microcontrollers/
19. Easy TinyML on ESP32 and Arduino - Hackster.io, accessed on September 7, 2025, https://www.hackster.io/news/easy-tinyml-on-esp32-and-arduino-a9dbc509f26c
20. Data Preprocessing Techniques for Handling Time Series data for Environmental Science Studies - International Journal of Engineering Trends and Technology, accessed on September 7, 2025, https://ijettjournal.org/assets/Volume-69/Issue-5/IJETT-V69I5P227.pdf
21. Sensor noise suppression - PMC, accessed on September 7, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC2253211/
22. ESP32 shows FFT wave to identify phonemes, musical notes, rhythms - YouTube, accessed on September 7, 2025, https://www.youtube.com/watch?v=pwi9sBT3_-E
23. WEBSPECTOR - a Web Based FFT Spectrum Analyzer With ESP32 : 5 Steps - Instructables, accessed on September 7, 2025, https://www.instructables.com/WEBSPECTOR-a-Web-Based-Spectrum-Analyzer-With-ESP3/
24. Environmental Anomaly Detection with Microcontrollers and TinyML - Embedded Explorer, accessed on September 7, 2025, https://embeddedexplorer.com/environmental-anomaly-detection-with-microcontrollers-and-tinyml/
25. A Holistic Review of the TinyML Stack for Predictive Maintenance - DTU Orbit, accessed on September 7, 2025, https://orbit.dtu.dk/files/383607666/A_Holistic_Review_of_the_TinyML_Stack_for_Predictive_Maintenance.pdf
26. Optimization Strategies for Low-Power AI Models on Embedded Devices | Applied and Computational Engineering, accessed on September 7, 2025, https://www.ewadirect.com/proceedings/ace/article/view/20598
27. How can we optimize AI and ML algorithms to reduce energy consumption and improve sustainability in computing? | ResearchGate, accessed on September 7, 2025, https://www.researchgate.net/post/How_can_we_optimize_AI_and_ML_algorithms_to_reduce_energy_consumption_and_improve_sustainability_in_computing
28. Speed Optimization - ESP32 - — ESP-IDF Programming Guide v5.5.1 documentation, accessed on September 7, 2025, https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-guides/performance/speed.html

29. Quantized Neural Networks for Microcontrollers: A Comprehensive Review of Methods, Platforms, and Applications - arXiv, accessed on September 7, 2025, https://arxiv.org/html/2508.15008v2
30. EFFICIENT DEPLOYMENT OF MACHINE LEARNING MODELS ON MICROCONTROLLERS: A COMPARATIVE STUDY OF QUANTIZATION AND PRUNING STRATEGIES. - Blucher Engineering Proceedings, accessed on September 7, 2025, https://www.proceedings.blucher.com.br/article-details/efficient-deployment-of-machine-learning-models-on-microcontrollers-a-comparative-study-of-quantization-and-pruning-strategies-38886
31. This Arduino-Powered TinyML Project Uses an Edge Impulse Model ..., accessed on September 7, 2025, https://www.hackster.io/news/this-arduino-powered-tinyml-project-uses-an-edge-impulse-model-to-listen-out-for-motor-failure-d167e4347629
32. TinyML Case Studies - Seeed Studio, accessed on September 7, 2025, https://files.seeedstudio.com/wiki/K1100-quick-start/TinyML-Case-Studies.pdf
33. Measuring Vibration for Predictive Maintenance | Pruftechnik, accessed on September 7, 2025, https://www.pruftechnik.com/measuring-vibration-for-predictive-maintenance/
34. Predictive Machine Maintenance Using Tiny ML - IJRASET, accessed on September 7, 2025, https://www.ijraset.com/research-paper/predictive-machine-maintenance-using-tiny-ml
35. Sensor Maintenance Dataset - Kaggle, accessed on September 7, 2025, https://www.kaggle.com/datasets/ziya07/sensor-maintenance-dataset
36. ESP32-Based Environmental Monitoring System with Multiple ..., accessed on September 7, 2025, https://docs.cirkitdesigner.com/project/published/41bd9302-4a49-49ca-9997-fe857a75b83a/esp32-based-environmental-monitoring-system
37. A Multi-Modal IoT Node for Energy-Efficient Environmental Monitoring with Edge AI Processing † - † thanks - arXiv, accessed on September 7, 2025, https://arxiv.org/html/2507.14165v2
38. TinyML Projects with the ESP32 WROVER Kit and Common Arduino Sensors, accessed on September 7, 2025, https://embeddedexplorer.com/tinyml-projects-with-the-esp32-wrover-kit-and-common-arduino-sensors/
39. ESP32-Based Environmental Monitoring System - How-to Guide and Editable Circuit, accessed on September 7, 2025, https://docs.cirkitdesigner.com/project/published/2e5030d1-8fa5-4a3e-9079-bed7b152134d/esp32-based-environmental-monitoring-system
40. Tiny Machine Learning Zoo for Long-Term Compensation of Pressure Sensor Drifts - MDPI, accessed on September 7, 2025, https://www.mdpi.com/2079-9292/12/23/4819
41. TinyML-Based Real-Time Drift Compensation for Gas Sensors Using Spectral–Temporal Neural Networks - MDPI, accessed on September 7, 2025,

https://www.mdpi.com/2227-9040/13/7/223
42. TinyML - Should models include preprocessing blocks to be ported safely to MCUs? - Reddit, accessed on September 7, 2025, https://www.reddit.com/r/embedded/comments/1ii487k/tinyml_should_models_include_preprocessing_blocks/