TinyML on ESP32 for Predictive Maintenance and Environmental Monitoring in Semiconductor Manufacturing

By VigneshKumar Bagavathsingh

Email: vigneshkumarb@bravetux.com ic19939@gmail.com

Executive Summary: TinyML enables running machine learning on low-power microcontrollers (MCUs). In semiconductor fabs, deploying TinyML on ESP32 boards (which offer Wi-Fi/Bluetooth connectivity and low cost) can support *predictive maintenance* (PM) and *environmental monitoring* (EM) without relying on cloud inference. In PM, vibration and temperature sensors on tools like pumps or wafer handlers can feed TinyML models that flag anomalies (e.g. bearing wear) before failure[1][2]. In EM, gas and particulate sensors detect leaks or contamination (VOC, HF, particulate levels) via pattern-recognition models (an "electronic nose" approach)[3][4]. Toolkits like TensorFlow Lite for Microcontrollers (TFLM) and Edge Impulse automate training and C code generation for ESP32. Decision-makers gain higher equipment uptime, lower scrap, and safety compliance; for example, one AI vibration-monitoring retrofit in a fab cut unscheduled downtime by 72% (saving ~$4.3 M/year)[5]. This report details sensor choices, data preprocessing, model design (anomaly vs classification), and ESP32 deployment with C code examples. We discuss frameworks, performance trade-offs (latency, power, accuracy), and alternatives (STM32, RP2040, etc.) to help engineers implement TinyML solutions effectively.

TinyML and ESP32 Overview

TinyML refers to machine learning models optimized for microcontrollers. ESP32 MCUs (dual-core 240 MHz, ~520 KB SRAM, built-in Wi-Fi/BLE) are popular for TinyML because they balance compute and connectivity[6]. TensorFlow Lite Micro (TFLM) is an open-source framework that runs inference on devices with only kilobytes of RAM; its core runtime can fit in ~16 KB on a Cortex-M3 and is officially ported to ESP32[6]. Commercial platforms like Edge Impulse provide a GUI/data-flow environment, automated feature extraction (e.g. spectral features from time-series) and an "EON" compiler that compresses models (typically 25–55% less RAM and ~35% less flash than raw TFLite)[7][8]. Models are trained off-chip (Python/TF) or via embedded data collection, then quantized (usually to 8-bit) and converted to a C array (via `xxd`). The resulting code is compiled into the ESP32 firmware, along with TFLM or Edge-Impulse runtime libraries, enabling on-device inference without dynamic memory.

ESP32 development can use the Arduino framework or Espressif's native ESP-IDF. In Arduino, for example, one adds the TinyML library and includes the converted model

header (e.g. `#include "model_data.h"`). A typical ESP32 TinyML sketch uses `tflite::MicroInterpreter` to load the model and allocate a "tensor arena" buffer. Data from sensors (ADC or I^2C) is placed into the interpreter's input tensor, `Invoke()` is called, and the output tensor yields a classification or anomaly score[9][10]. For instance, the code might look like:

```
#include <TensorFlowLite.h>
#include "model_data.h" // TFLite model in a C array

// TensorFlow Lite setup (error reporter, interpreter, etc.)
tflite::MicroErrorReporter micro_error_reporter;
const tflite::Model* model = tflite::GetModel(model_data);
static tflite::AllOpsResolver resolver;
constexpr int kTensorArenaSize = 2048;
static uint8_t tensor_arena[kTensorArenaSize];
static tflite::MicroInterpreter interpreter(model, resolver, tensor_arena,
kTensorArenaSize, &micro_error_reporter);

void setup() {
    Serial.begin(115200);
    interpreter.AllocateTensors();  // Prepare memory for input/output
}

void loop() {
    // Read sensors into input array
    float sensor_values[kInputSize] = { read_x(), read_y(), read_z() }; //
e.g., accelerometer axes
    float* input = interpreter.input(0)->data.f;
    for(int i=0; i<kInputSize; i++) {
        input[i] = sensor_values[i];
    }
    // Run inference
    if (interpreter.Invoke() != kTfLiteOk) {
        Serial.println("Model invoke failed");
        return;
    }
    // Process output
    float* output = interpreter.output(0)->data.f;
    int predicted = argmax(output, kOutputSize);
    Serial.println(predicted);
    delay(100);
}
```

Such code (adapted from tutorials[9][10]) illustrates the main steps: include TFLM, load the model, feed input, invoke, and read results. For anomaly detection models, instead of `argmax`, one might check if the output "anomaly score" exceeds a threshold. All allocation is static to avoid dynamic memory.

1. Predictive Maintenance Use Case

Sensors and Data Collection

In semiconductor fabs, equipment like **pumps, compressors, motors, and wafer-handling robots** are critical. Sensors commonly used include **vibration accelerometers** (to catch misalignment or bearing wear), **temperature probes** (overheating bearings, motors), **pressure transducers**, and **acoustic or current sensors**[2]. For example, piezoelectric or MEMS accelerometers (from vendors like PCB Piezotronics, STMicro, or Analog Devices) are used in etching and wafer-handling tools; PCB notes that accelerometers measure equipment vibration to ensure uniform etching[1]. Likewise, condition-monitoring blogs highlight that pumps in fabs are instrumented with vibration, pressure, and temperature sensors to detect bearing faults or leaks early[2]. Robotic arms may use magnetic or encoder sensors for position, but also monitor motor current/vibration. These signals are sampled (often at 100–1000 Hz for vibration) and may be processed with filters or FFT. Data preprocessing (e.g. windowing, FFT magnitude, or time-domain features like RMS) is usually done on the MCU before inference to reduce noise and input size.

Key sensor examples: - **Accelerometers (vibration):** MEMS 3-axis (e.g. Analog Devices ADXL345, Bosch BMA280) or piezoelectric types. These are read via I^2C/SPI and give X, Y, Z acceleration. - **Temperature:** Digital sensors (Dallas DS18B20) or thermocouples (MAX6675 interface) monitor bearing/motor heat. - **Pressure:** MEMS pressure sensors (Bosch BMP/BME series) monitor hydraulic/pneumatic lines. - **Current:** Hall-effect or shunt sensors track motor current draw. - **Acoustic:** Microphones can detect unusual sounds.

Model Selection (Anomaly vs Classification)

Predictive maintenance can use **supervised classification** if fault modes are known (e.g. normal vs “imbalance” vs “misalignment” states) or **anomaly detection** if only normal operation data is labeled. On a microcontroller, anomaly detection (unsupervised) is common: for instance, training a model only on “normal” vibration patterns so that deviations trigger an alert. Edge Impulse demonstrates using a K-means or autoencoder anomaly model on motor data: they trained a model on normal motor power data and flagged any out-of-distribution behavior[11]. In one example, a TinyML K-means anomaly detector running on a microcontroller caught unusual current spikes in a BLDC motor[11]. Alternatively, a small neural network (fully-connected or CNN) can classify a few discrete fault classes if a labeled dataset exists (e.g. “normal”, “stop”, “block”, “imbalanced” in a rotating device[12][13]).

Model complexity must be tiny: typically 1–3 layers with a few dozen neurons or small CNNs (for spectral input). Quantization to INT8 is essential to fit memory. Frameworks allow automatic generation: e.g., TensorFlow’s TFLiteConverter quantizes and outputs a C

array. For anomaly detection, Edge Impulse can auto-generate a threshold-based model after PCA/K-means.

Implementation on ESP32 (C Code)

Once a model is trained and converted, deployment on ESP32 follows the earlier code pattern. The sensor loop often looks like:

```
// Example: Acquire vibration data and run anomaly detection
float vib_data[64];
for(int i=0; i<64; i++) {
    vib_data[i] = read_accel(); // block until a window of 64 samples is
filled
}
float* input = interpreter.input(0)->data.f;
for(int i=0; i<64; i++) input[i] = vib_data[i];
interpreter.Invoke();
float anomaly_score = interpreter.output(0)->data.f[0];
if (anomaly_score > threshold) {
    // Flag maintenance alert
}
```

In practice, one might run an FFT on `vib_data` before feeding it. All code is in C/C++ using the TFLM C++ API (under the hood it's C). TensorFlow Lite Micro uses fixed buffers (the "tensor_arena" in the example) sized at compile time; for example a keyword-spotting model might only need a few KB[14].

Deployment and Communication

The ESP32 can send alerts/results via Wi-Fi to a factory dashboard or cloud. For reliability, one may implement retries or store logs in flash. OTA firmware updates are possible if tools are in place, allowing models to be updated in the field. Critical safety processes might require the MCU to fail safe (e.g. shut off motor) if an error is detected.

Benefits and ROI

Predictive maintenance yields **higher equipment uptime and efficiency**. Industry reports note that improved tool reliability can raise equipment availability by ~15% (translating to similar OEE gains)[15]. In practice, AI-driven monitoring has shown dramatic savings: one semiconductor fab cut its unplanned downtime by 72%, saving \$4.3M/year, by adding an ML-based vibration monitor to a critical machine[5]. More broadly, predictive maintenance can reduce maintenance costs by 20–30% and extend equipment life[16]. Decision-makers should compare this to the low cost of an ESP32+sensor module (tens of dollars) versus the high cost of tool downtime or scrap. The performance advantage is near-real-time fault detection (single-digit-millisecond inference) with no monthly cloud fees, and all data privacy is local.

Primary considerations: PD TinyML trade-offs include sensor sampling rate vs power (higher sample rates = more CPU usage) and model complexity vs memory. Using an anomaly model means no need to label faults, but may trigger false positives if normal conditions drift. Performance can be tested on target hardware – e.g., a small classification model on 3-axis accelerometer data often runs in <10 ms inference on an ESP32 at 240 MHz.

2. Environmental Monitoring Use Case

Sensors and Integration

Semiconductor fabs require ultra-clean environments. Environmental monitoring TinyML systems can detect **air quality issues** (particulates, humidity), **gas leaks** (e.g. toxic or flammable gases), or even **water/soil conditions** around the plant. Typical sensors include:

- **Gas sensors:** Metal-oxide semiconductor (MOX) sensors for VOCs (CO, NO_2, NH_3, H_2S, etc.), electrochemical sensors for CO, O_2, etc. For example, multi-channel MEMS gas sensor modules (with arrays of four different MOX elements) can profile complex gas mixtures[17]. An "e-nose" using such an array can identify different volatile organic compounds by pattern recognition[3]. Semiconductor sensor vendors (e.g., SGX Sensortech, Bosch BME680/BME688) make small gas/IAQ sensors often used with MCUs.
- **Particulate sensors:** Optical particle counters (laser scattering) measure dust particles for cleanroom monitoring. These typically communicate via UART/I^2C.
- **Humidity and temperature:** Sensors like SHT3x or BME280 monitor cleanroom conditions (tight humidity/temp control is vital).
- **Airflow/pressure:** Differential pressure sensors ensure positive/negative room pressure.
- **Specialty gas:** Many fabs use gases like hydrogen fluoride (HF) or ammonia. Dedicated toxic gas detectors exist (though usually in industrial control systems, not easily interfaced to hobby MCUs).

All sensors interface via analog or digital (I^2C/SPI) lines. An ESP32 can poll them at modest rates (e.g. 1 Hz or slower for gas sensors). Preprocessing might include calibration and drift compensation. Patterns of sensor readings can feed a TinyML model: e.g., use multiple gas sensor channels as a feature vector.

Model Strategies

Environmental TinyML can use **classification** to identify specific events (e.g. "CO leak", "VOC rise", "normal") or **anomaly detection** to warn of any deviation. For example, the Edge Impulse blog "Sniffing Out Industrial Chemicals" shows a TinyML model distinguishing six industrial chemical fumes (acetone, hexane, etc.) with ~100% accuracy[17] using a 4-channel gas sensor array. That model was trained on 15-second

sensor traces and compressed with Edge Impulse's EON compiler before deployment. In a fab scenario, one might train on "clean air" versus "contaminated" samples. The model size can remain small (e.g. a few thousand weights) and quantized to fit the ESP32.

A simple threshold logic (e.g. "if any sensor > limit, alarm") is an alternative, but TinyML offers pattern recognition across sensors. For instance, classifying specific gas types or combined indexes (like an Air Quality Index). Code on ESP32 would read the sensors (e.g. analog voltages from gas sensor heaters) into an input array and run the model, similar to the PD example.

Deployment Example

Suppose we use a 4-gas MEMS sensor (as in[17]). The ESP32 code loop might be:

```
float gas_vals[4];
for(int i=0; i<4; i++) {
    gas_vals[i] = analogRead(gas_pin[i]); // or I2C read for digital sensor
}
float* input = interpreter.input(0)->data.f;
for(int i=0; i<4; i++) input[i] = gas_vals[i];
interpreter.Invoke();
float* out = interpreter.output(0)->data.f;
int gas_type = argmax(out, numClasses);
// Map gas_type to actual condition (e.g. 0=normal,1=leak)
```

Integration: Many TinyML projects (DFRobot, Seeed) pair an ESP32 with a Grove gas sensor or an environmental shield. Libraries for specific sensors (I^2C/SPI drivers) are used in setup(). The model header is loaded as before. Edge Impulse can even generate an Arduino library for the trained model to simplify coding.

Cleanroom & Safety Use Cases

In fabs, environmental TinyML could continuously monitor: - **Air quality:** Detect unexpected VOC spikes that could indicate solvent leaks or off-gassing. - **Gas leaks:** Identify traces of dangerous gases (e.g. PH_3 in doping, or HCl/HF in wet etch stations) quickly. The system could send an alarm well before normal alarms would trigger, enabling preventive action. - **Particle breaches:** If combined with a particle sensor, flag a burst of particulates (though real cleanroom monitors are highly specialized). - **HVAC faults:** Using temperature/humidity/airflow, detect failures in cleanroom conditioning. - **Soil/water:** If outside environment matters (e.g. groundwater monitoring for leaks), sensors like pH or moisture sensors can similarly feed TinyML to detect contamination trends.

TinyML models here improve responsiveness: inference is local (no network delay) and the ESP32 can operate autonomously even if connectivity is lost. Because air/gas sensors often produce slow-changing signals, the inference frequency can be low (seconds or minutes), conserving power.

Benefits

TinyML-based monitoring can prevent costly contamination incidents and ensure regulatory compliance. Semiconductor gas leaks (toxic or flammable) can shut down fab lines; early detection via intelligent sensing can avert this. The “electronic nose” approach enables identifying complex gas mixtures that single-sensor alarms would miss. Crucially, these systems run on battery-powered or 24/7 edge devices, so they don’t rely on cloud or MES systems. A side benefit is that all raw sensor data never leaves the fab (data privacy).

A recent market survey notes the global semiconductor gas sensor market is growing ~8–9%/year as fabs invest in more monitoring[18]. The integration of TinyML further justifies sensors: it adds real “intelligence” and ROIs. Decision-makers can quantify savings in avoided downtime (e.g. 1 hour of downtime on a 300 mm line can cost \$100K+), and in regulatory penalties averted. A cited example: adding smart sensors to plant utilities can cut maintenance labor by ~12% while boosting uptime[19], illustrating how predictive strategies (applied to environment too) are already driving cost reduction.

3. Tools, Frameworks, and Implementation Details

Frameworks: The dominant open-source tool is **TensorFlow Lite for Microcontrollers**. This provides C++ APIs (in a few .h/.cpp files) to run inference, with support for standard layers (conv, dense, softmax, etc.). Espressif maintains an official *esp-tflite-micro* library; examples show how to integrate `tflite::GetModel`, `MicroInterpreter`, etc.[20][21]. Another approach is **Edge Impulse**, a cloud/web platform: one collects data (ESP32 can stream via Edge Impulse’s data forwarder), trains a model in the browser, then downloads an Arduino-compatible SDK or C library for the model. Edge Impulse adds optimization (its EON compiler) and a library that abstracts sensor handling, but under the hood it also uses TFLM.

Other frameworks include **STM32Cube.AI** (ST’s tool that generates C code for STM32 MCUs) and **NXP eIQ**. These target more powerful MCUs. MicroTVM or uTensor are alternative open toolkits, but less common in industry. All of these follow a similar pattern: convert a trained model into optimized fixed-size code for the MCU.

C Programming on ESP32: While examples often use Arduino (C++), one can use straight ESP-IDF (which is essentially C). The core inference code is valid C/C++. An important step is model conversion: after training in Python/TensorFlow, run the TFLiteConverter and then a script like `xxd -i` to produce a `.h` with a byte array (as shown in[22]). This array is included in the project. The code in `setup()` initializes the interpreter and allocates the tensor arena, as in[23]. In `loop()`, read sensors and copy values into `interpreter->input(0)->data.f[i]`[9]. After `Invoke()`, access `interpreter->output(0)->data.f`. Utility functions like finding the max index in the output (or thresholding an anomaly score) are then implemented in C.

Edge Impulse Example: Edge Impulse automates this: after deploying a model, you can export an Arduino sketch that includes all inference code. Under the hood it does

essentially the same steps (model header, interpreter, etc.). The DFRobot tutorial confirms that TensorFlow Lite Micro supports ESP32[6], and Edge Impulse can generate code that runs on ESP32 with lower memory footprint thanks to EON[7].

Development Workflow: A recommended flow is: 1. **Data Collection:** Use the ESP32+sensor to gather representative data for each use case (store to SD or send to PC). 2. **Model Training:** In Python or Edge Impulse, preprocess (normalize, window, compute FFT if needed) and train a small network. 3. **Model Optimization:** Quantize weights/activations to 8-bit. Use Edge Impulse's validation or TensorFlow's tools to ensure accuracy holds. 4. **Code Generation:** Convert to TFLite, then C array. Prepare an ESP32 project including TensorFlow Lite Micro library (via ESP-IDF `idf.py add-dependency` or Arduino Library Manager). 5. **Coding:** Write C code to interface sensors, feed data into the model, and act on the output (e.g. log or flag alarm). 6. **Testing on Hardware:** Check inference results against known inputs (print outputs to serial). Debug any memory issues (tensor arena too small etc.). 7. **Deployment:** Flash to ESP32 modules placed on equipment. Implement long-term logging or cloud sync if needed.

This mirrors embedded ML best practices; many resources (e.g. Digikey TinyML tutorials) describe these steps in detail[24][21].

4. Performance Trade-offs and Best Practices

Memory: The ESP32's ~520 KB SRAM (and typically a few MB of flash) is limited. Models must be very compact. For example, the sine-wave demo in Espressif's guide set a tensor arena of only 2000 bytes[25]. In practice, you might allocate 10–20 KB for a small network. Any model and tensor arena together must fit under RAM – as the Espressif guide warns, exceeding this causes failures[14]. To stay within limits, fully **quantize** models to INT8 (reduce memory 4× vs float) and consider pruning. Quantization also speeds up inference and lowers power[26]. Indeed, surveys note that quantizing embedded models "reduces memory usage, accelerates computation, and decreases power consumption"[26]. Edge Impulse's EON compiler automates some of this.

Latency: TinyML inference on ESP32 is very fast for small models (typically 1–10 ms). However, large models *will* be slow or impossible. A cautionary report noted that a ~2 MB YOLO model on an ESP32-S3 (with only ~300 KB free SRAM) took ~60 seconds per inference because it spilled into external PSRAM[27]. In contrast, a similar model on PC took ~50 ms. This illustrates that once a model can't fit in fast on-chip memory, latency explodes[27]. Thus, **keep models small**: often <50 KB. Models with heavy ops (large convolution layers, big images) are unrealistic. Instead use 1D or very small 2D convnets. Always measure on-device inference time to validate real-time requirements.

Power: Active ESP32 operation (with radio and CPU) consumes on the order of tens of milliamps. Typical Wi-Fi/BLE plus ML can draw ~40–60 mA[28]. In deep-sleep (Wi-Fi off), ESP32 can go down to <0.1 mA on a well-designed board[29]. In TinyML applications, it is common to wake the chip on a timer or interrupt, take readings, run the model, send a result, then return to deep sleep. This duty-cycling yields very low average power. (For

example, an ESP32 board measured ~48 mA active vs ~0.05 mA in sleep[29].) The exact savings depend on hardware. Engineers should disable unused peripherals and reduce clock speed if possible. Quantized inference also uses simpler integer math, which is somewhat more energy-efficient.

Accuracy: Quantization and model compression can slightly degrade accuracy. Use calibration or quantization-aware training if full precision drop is unacceptable. Also, account for sensor noise and drift: for PM, regularly re-train models with new data, and for EM, recalibrate gas sensors periodically. Edge Impulse's workflow (data cleaning, validation charts, etc.) helps ensure robustness.

Wireless Trade-offs: ESP32's built-in Wi-Fi is a key advantage (no extra modules needed). However, transmitting large datasets is expensive power-wise. With TinyML, one typically sends only concise results (e.g. "anomaly detected") or periodic logs, not raw data. If always-on connectivity isn't needed, consider using ESP32's 802.11n hardware off. Also ensure security: use HTTPS or secure MQTT for any cloud comms, and consider SSL certificates to protect ML models and data.

Deployment Challenges: Common issues include memory allocation errors (often fixed by reducing model size or increasing `kTensorArenaSize`), and ensuring all TensorFlow operations used in the model are supported by TFLM's op resolver. Testing edge cases is critical. Also, maintainability: plan how to update the model when needed (e.g. via OTA or swapping an SD card). Document the model's performance (false-positive/negative rates) for review. Finally, ensure the ESP32 code properly handles cases where sensor readings fail or time out (to avoid inference on garbage data).

5. Alternatives and Platform Comparison

While ESP32 is a strong TinyML platform (integrated radio, low cost), alternatives exist:

- **ARM Cortex-M MCUs:** Many STM32 (STMicro) and NXP i.MX RT chips can run TFLite Micro or vendor tools (STM32Cube.AI) for TinyML. These often have more SRAM or dedicated DSP (e.g. STM32H7 has 1 MB+ RAM), enabling slightly larger models. However, they usually lack built-in Wi-Fi; adding a module raises cost/size. DFRobot notes that TensorFlow Lite Micro targets ARM Cortex-M processors, and an official ESP32 port is provided[6].

- **Raspberry Pi Pico W (RP2040):** The RP2040 dual-M0+ chip can run lightweight ML (e.g. with TensorFlow Lite Micro), and the Pico W adds Wi-Fi. It has 264 KB SRAM. Its clock is slower (133 MHz) than ESP32, but for very simple models it can suffice. It's a low-cost alternative, but peripherals (I/O, ADC quality) differ.

- **Raspberry Pi Zero / Compute Module:** Technically not a microcontroller, but a Linux SBC. It can run full TensorFlow or PyTorch Lite, handling larger models and more sensors. The trade-off is much higher power (~150 mA idle), larger size, and

OS complexity. It's overkill for simple TinyML but useful if you need camera-based ML.

- **Dedicated AI MCUs:** Chips like the Syntiant NDP101 or GreenWaves GAP8 are emerging for audio/TinyML tasks with extremely low power. They have built-in NPUs but no general-purpose I/O; pairing them with an ESP32 as a host is possible.

- **Comparison:** The ESP32 stands out for rapid prototyping due to Arduino libraries and community examples. It offers a sweet spot: enough horsepower for small nets, plus connectivity. Its main limit is memory; if that's too restrictive, a higher-end MCU or a small Linux board may be chosen. For instance, ST's new STM32U5 series (Cortex-M33) has low-power modes and could run TinyML with very low standby current.

In summary, choose ESP32 when Wi-Fi and low-cost form-factor are needed and model size is modest. Use alternatives only if you need much larger models, ultra-low idle power, or special sensors (e.g. built-in camera). Table:

- *ESP32:* ~240 MHz dual-core, Wi-Fi/BLE, ~520 KB SRAM; widely supported by TensorFlow Lite Micro[6].
- *STM32H7:* ~480 MHz single-core, no radio, >1 MB SRAM; supported via STM32Cube.AI.
- *RP2040:* 133 MHz dual-core, Wi-Fi on Pico W, 264 KB SRAM.
- *RPi Zero W:* 1 GHz ARM11, 512 MB RAM, Linux OS, higher power.

6. Conclusion

Implementing TinyML on ESP32 for semiconductor-industry use cases combines edge intelligence with cost-effective hardware. For predictive maintenance, TinyML enables real-time anomaly detection on vibration and temperature data, helping avoid costly downtime[5][30]. For environmental monitoring, ESP32-based gas-sensing systems can continuously surveil fab air, detecting hazardous leaks or contamination (an "electronic nose" in the cleanroom)[3][4]. Key strategic benefits include higher yield, safety compliance, and reduced maintenance cost, all at low device cost.

Engineers can leverage TensorFlow Lite Micro or Edge Impulse to compress models into C code, and integrate with common sensors via I^2C/SPI. They must balance latency, accuracy, and power: models should be quantized and small, sensors polled at an appropriate rate, and the system duty-cycled into deep sleep when idle. Decision-makers will appreciate the tangible ROI: sensors + ESP32 modules cost a few tens of dollars, yet real-world deployments have cut maintenance events by >70%[5] and significantly boosted OEE[15]. Compared to alternative controllers, the ESP32's built-in wireless and strong community support give it an edge for rapid TinyML development. In the fast-evolving fab environment, TinyML on microcontrollers offers a pragmatic path to smarter, data-driven operations with minimal infrastructure change.

References: Findings and recommendations are supported by industrial reports and technical sources[1][3][17][30][5][14][6][9][26][2], among others. These include application notes on fab sensors, TinyML implementation guides, and case studies. Each citation 【cursor†L..-L..】 corresponds to detailed information used.

---

[1] Sensors for Semiconductor Manufacturing | PCB Piezotronics

https://www.pcb.com/applications/test-and-measurement/semiconductor-manufacturing

[2] Precision in Semiconductor Manufacturing: The Role of Condition Monitoring and Level Sensing | Balluff

https://www.balluff.com/en-us/blog/precision-in-semiconductor-manufacturing-the-role-of-condition-monitoring-and-level-sensing

[3] [18] Semiconductor Gas Sensors for Environmental Monitoring

https://www.azosensors.com/article.aspx?ArticleID=2765

[4] TinyML-ESP32 Integration: Cutting-Edge and Smart Home Automation Applications Overview - DFRobot

https://www.dfrobot.com/blog-13902.html?srsltid=AfmBOoo8XFlHKwAC0IkRokZJ7Lqig0bq5HR1es2MNqobae5d82URTWBL

[5] [16] [19] AI Predictive Maintenance Cuts Costs by Millions

https://www.flexsin.com/blog/how-ai-can-help-drive-seven-figure-cost-reductions-with-predictive-maintenance/

[6] [7] Top 8 TinyML Frameworks for Makers | Tiny Machine Learning Tools & Platforms - DFRobot

https://www.dfrobot.com/blog-13921.html?srsltid=AfmBOoot3DsZeouWmkyeLRkY20rOuzwGMnapsPsH-QvunUHvDbLYvd7S

[8] [17] Sniffing Out Industrial Chemicals Using tinyML

https://www.edgeimpulse.com/blog/sniffing-out-industrial-chemicals-using-tinyml/

[9] [10] [13] [20] [23] TinyML with ESP32 Tutorial | Microcontroller Tutorials

https://www.teachmemicro.com/tinyml-with-esp32-tutorial/

[11] Brushless DC Motor Anomaly Detection - Edge Impulse Documentation

https://docs.edgeimpulse.com/projects/expert-network/brushless-dc-motor-anomaly-detection

[12] Perform Predictive Maintenance on Rotating Device Using ESP32 Board, ThingSpeak, and Machine Learning - MATLAB & Simulink

https://www.mathworks.com/help/matlab/supportpkg/predictive-maintenance-rotating-device-esp32-ML-example.html

[14] [21] [22] [24] [25] TensorFlow Lite On ESP32 – OpenELAB Technology Ltd.

https://openelab.io/blogs/learn/tensorflow-lite-on-esp32?srsltid=AfmBOooke9BoXWwpZsUf1b_lHf1XscZfFSXaque2ZRgzWBc1U5kSTYeD

[15] [30] Using Predictive Maintenance To Boost IC Manufacturing Efficiency

https://semiengineering.com/using-predictive-maintenance-to-boost-ic-manufacturing-efficiency/

[26] Quantization For Embedded Systems

https://www.meegle.com/en_us/topics/quantization/quantization-for-embedded-systems

[27] Using a large model on a ESP32 S3 N16R8 (TFMIC-37) · Issue #94 · espressif/esp-tflite-micro · GitHub

https://github.com/espressif/esp-tflite-micro/issues/94

[28] [29] ESP32 Deep Sleep Tutorial : 7 Steps - Instructables

https://www.instructables.com/ESP32-Deep-Sleep-Tutorial/