What Every Engineer Must Know About AI

No Hype. No Vagueness. Just Engineering.

A structured journey from first principles to physical deployment of AI for ECE, EEE, CSE, and Mechanical engineers.

Rev1.0 June 2026

Contents

SECTION 01

Foundations

01

AI, ML, Deep Learning, Model, Training vs Inference

What is Artificial Intelligence?

A simple definition of AI:
AI is any engineered system that exhibits intelligent behavior — either through rules written explicitly by a human, or through patterns learned automatically from data.

	Rule-Based AI	Learning-Based AI
Rules created by	Human engineer — written explicitly	Machine — Rules discovered from data
Does it learn?	No — follows fixed instructions programmed	Yes — during training
Needs data to build?	No	Yes — large amounts of data needed
Examples	Kalman Filter, Fuzzy Logic, PID logic	Neural networks, GPT-4, image recognition

One important truth:

Rule-Based AI is still genuine AI — it exhibits intelligent behaviour without learning.

However, when people say 'AI' today they almost always mean Learning-Based AI specifically.

Fact – Rule-based AI has been there for ages

In fact, "If-Then-Else logic" was the first form of AI ever implemented — expert systems in the 1960s and 70s were essentially large collections of If-Then-Else rules encoding human expert knowledge.

Implementation	Complexity	Example
If-Then-Else logic	Simplest — threshold decisions	If temperature > 85°C then trigger alarm
State machines	Structured rule sequences	Fault detection across operating modes
PID with decision logic	Control + mode switching	Motor control with fault response
Fuzzy Logic	Handles imprecise inputs	HVAC comfort control
Kalman Filter	Optimal estimation under noise	Drone stabilization, GPS smoothing
Expert systems	Large rule knowledge bases	Medical diagnosis, Fault free analysis

All of these share the same fundamental property — **a human engineer wrote every rule explicitly.** The system follows those rules precisely. It does not learn. It does not adapt. It does not improve from experience.

The progression from If-Then-Else to Kalman Filter is purely one of mathematical sophistication — not a different category. They are all Rule-Based AI.

Learning-based AI — End-to-End Journey

Every learning-based AI system — simple or complex — follows the same end-to-end journey:

RAW DATA → ALGORITHM → TRAINING → MODEL → INFERENCE ENGINE → DECISION

Step	What it means
Raw Data	Measurements, signals, images, text — anything the real world generates
Algorithm	The mathematical method that defines how the machine will learn from that Raw data
Training	The process of running the algorithm on data until the machine learns accurately
Model	The stored result of training — a file of learned numbers, ready to be deployed
Inference Engine	The program that loads the model and applies it to new incoming data
Decision	The physical output — classify, detect, predict, alert, control

Each step has a distinct location, distinct hardware, and distinct engineering ownership — covered in the next slide.

Machine Learning based AI Implementation Process — End to End

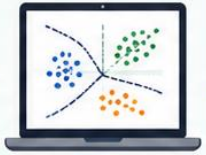
1 Instrument the System

Collect structured sensor data over time.

2 Label the Data

Label the data — Normal / Fault / Degraded or continuous measurement targets.

3 Select Algorithm

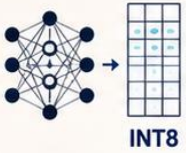


- Decision Tree
- Random Forest
- SVM
- Shallow Neural Network

4 Train on Workstation or Cloud

Train the model and validate accuracy against held-out data.

5 Quantize Model

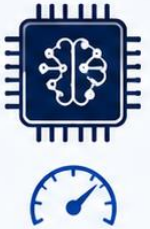


Compress the trained model to INT8 for efficient embedded deployment.

6 Deploy to MCU

Deploy via OTA, JTAG, DFU, or any supported update mechanism.

7 Inference Engine Runs on MCU



Decision made in microseconds on every new reading.

8 Retrain Periodically

As new field data is collected...

9 Retrain and Redeploy Improved Model

Use new field data to retrain, re-validate and deploy improved model continuously.

- Adaptive Improvement
- Higher Accuracy over Time
- Efficient Embedded Performance
- Lower Bandwidth & Compute Cost
- Secure & Reliable Deployment

The Hardware Reality — Where Each Step Lives

RAW DATA → ALGORITHM → TRAINING → MODEL → INFERENCE ENGINE → DECISION

Step	Location / Origin
Raw Data	Comes from Field — sensors, equipment, environment
Algorithm	Workstation / cloud — written / often chosen by ML engineer
Training	Wherever compute power is available — PC, workstation, cloud
Model	Originates where training runs — frozen model deployed to the embedded systems (for making inferences) via OTA, JTAG, DFU, or any supported update mechanism
Inference Engine	Runs on target hardware embedded system — MCU, MPU, Single Board Computer, edge module
Decision	Taken at the Edge device (Embedded system) — where action is needed

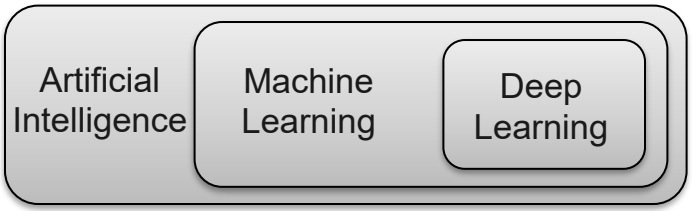
Deployment Type	Hardware Example	RAM required	Learning on Device?
TinyML	STM32 Cortex-M4 MCU	KBs	No — frozen model deployed
Edge AI	Raspberry Pi / ESP32-S3	MBs	No — frozen model deployed
Advanced Edge AI	Jetson Orin / Apple M4	GBs	Frozen / continuously learning

Where Does Machine Learning Fit?

AI is the broad goal — any system that exhibits intelligent behaviour.
Machine Learning is one specific way of achieving that goal.

Level	Name	What it means
Broadest	Artificial Intelligence	Any system performing tasks requiring human-like intelligence
Subset of AI	Machine Learning	Subset of AI where systems are designed to automatically learn patterns from data using one of many algorithms
Subset of ML	Deep Learning	ML using large multi-layer neural networks specifically

Visualised:



$$DL \subset ML \subset AI$$

When someone says 'AI' today they almost always mean the ML subset — and often the Deep Learning subset within that. Knowing which level is being referred to is the first act of terminological precision.

What is Machine Learning?

Machine Learning is a subset of AI where systems are designed to automatically learn patterns from data using one of many algorithms.

Three elements are always present:

Element	What it is	Analogy
Data	The experience the machine learns from — sensor signals, images, measurements, text	Textbooks and examples a student studies
Algorithm	The mathematical method that defines exactly how the machine learns from that data	The study method the student uses
Model	The stored knowledge produced after training is complete — a file of numbers encoding what was learned	The knowledgeable engineer after years of experience

DATA + ALGORITHM → TRAINING → MODEL

ML is not a single technique — it is a category containing many different learning approaches each suited to different problems.

The number of algorithms is not fixed — new ones are published continuously and many proprietary ones are never published.

What is a Model — Really?

After training, what you have is a model — the most important output of the entire ML process. The model sits exactly at the boundary between cloud and edge:

`TRAINING (cloud)  → MODEL  →  INFERENCE ENGINE (edge device)`

What a model physically is:

A model is a large file containing numbers — called weights and biases — organised in layers. These numbers encode everything the machine learned from the data.

System	Approximate weights	Deployed where
GPT-4 (cloud LLM)	~1.8 trillion	Cloud servers
BERT (language model)	~110 million	Server / edge server
MobileNet (image, mobile)	~4 million	Smartphone / Raspberry Pi
Keyword spotting model	~50,000	STM32 Cortex-M4

The model is not executable code. It is data — like filter coefficients stored in flash on a DSP. The inference engine reads the weights and performs calculations on new inputs.

The Model File — What Is Really Inside

After training completes — the result is saved as a model file. To an embedded firmware engineer this file has a direct and familiar parallel.

Content	What it is	Embedded parallel
Architecture definition	How many layers, how they connect, what operation each layer performs	A struct definition — describes the shape of the data
Weights and biases	The actual learned numbers — one large array per layer	A const array in flash — coefficients stored at known addresses

What the numbers look like — before quantization:

```
Layer 1 weights:  [ 0.00732,  -0.01423,   0.00089,  -0.00341 ... ]
Layer 1 biases:   [ 0.12400,  -0.05600,   0.08900             ... ]
Layer 2 weights:  [-0.02341,   0.00912,  -0.01876,   0.00445  ... ]
Layer 2 biases:   [ 0.03200,   0.01100,  -0.02400             ... ]
```

Nothing else. No executable code. No logic. Just numbers.

The model file is data — not a program. The program that reads it and uses it is the inference engine.

The Model File — Size Reality Across the Spectrum

The same concept — weights and biases stored as numbers — scales from a microcontroller to a cloud server. Only the count and precision change.

Model	Weights	FP32 size	INT8 after quantization	Deployed where
GPT-4 (cloud LLM)	~1.8 trillion	~7.2 TB	~1.8 TB	Cloud servers
MobileNet V2 (image)	~3.4 million	~13 MB	~3.4 MB	Smartphone / Pi
Keyword spotting	~500,000	~2 MB	~500 KB	STM32 Cortex-M4
Anomaly detector	~50,000	~200 KB	~50 KB	STM32 Cortex-M4

Key insight for embedded engineers:

The anomaly detector model — 50,000 weights at INT8 — occupies 50KB of flash. That is within the budget of any STM32 Cortex-M4 with 1MB flash. The weight array sits alongside your application code exactly like any other const data.

The model does not grow at runtime. It is static data in flash — it never moves to RAM.

The Model File — Quantization Step by Step

Quantization converts each 32-bit float weight to an 8-bit integer — reducing memory by 4x. The mathematics are identical to Q-format fixed point on a C2000 DSP.

```
FP32 weight = 0.00732841

Step 1 — Find range of all weights in the layer:
         min = -0.08431    max = 0.09124

Step 2 — Compute scale factor:
         scale = (max - min) / 255  =  0.000688

Step 3 — Quantize:
         INT8 = round((0.00732841 - (-0.08431)) / 0.000688)
              = round(133.2)  =  133

Step 4 — Store as one byte:
         weight = 133   ← 1 byte instead of 4
```

STM32Cube.AI performs all four steps automatically — you import the trained model and it generates the quantized C array ready to compile into your firmware.

The Model File — What Arrives on the STM32

STM32Cube.AI generates a C array you include in your project — exactly like any other const data. This is the model.

```
/* Auto-generated by STM32Cube.AI                          */
/* Do not edit — regenerated on each model update          */

const uint8_t network_weights[] = {
    133, 67, 129, 114, 201, 88, 156, 43,
    178, 92, 107, 245,  31, 189, 77, 203,
    /* ... continues for tens of thousands of bytes ...    */
};

const uint8_t network_biases[] = {
    159, 82, 141,  97, 220,  73,
    /* ...                                                 */
};
```

This is not code. It is data.

It sits in flash exactly like a lookup table or a filter coefficient array. The inference engine reads it. When an OTA update arrives — only this array changes. The inference engine code does not change.

The Inference Engine — What It Is

The inference engine is the C program that reads the weight array from flash and computes a decision from new sensor input. It is the only executable code in the AI system.

The analogy every embedded firmware engineer immediately understands:

DSP FIR filter	Neural network inference
Filter coefficients in flash — const array	Model weights in flash — const array
New ADC sample arrives	New sensor feature vector arrives
Multiply sample by each coefficient	Multiply input by each weight
Accumulate results — MAC	Accumulate results — MAC
Apply gain / scaling	Apply activation function
Output filtered value	Output classification or prediction

If you have implemented a FIR filter in C on an MCU — you have already implemented the core operation of neural network inference.

The inference engine is a generalisation of that exact computation.

The Inference Engine — One Layer in C

A single neural network layer is a multiply-accumulate loop followed by an activation function. This is the complete C implementation:

```
void run_layer(
    const float *input,    /* sensor feature vector        */
    const float *weights,  /* weight array from flash      */
    const float *biases,   /* bias array from flash        */
    float       *output,   /* result to next layer         */
    int          n_in,     /* number of input features     */
    int          n_out)    /* number of output neurons     */
{
    for (int i = 0; i < n_out; i++)
    {
        float sum = biases[i];       /* start with bias     */
        for (int j = 0; j < n_in; j++)
        {
            sum += input[j] * weights[i * n_in + j]; /* MAC */
        }
        output[i] = (sum > 0.0f) ? sum : 0.0f; /* ReLU    */
    }
}
```

The Inference Engine — What Each Line Does

Every line of the inference engine maps directly to a concept familiar to embedded firmware engineers:

Code line	Operation	Embedded parallel
sum = biases[i]	Load bias — starting offset for this neuron	DC offset — initialise accumulator
sum += input[j] * weights[...]	Multiply input by weight — accumulate	FIR filter MAC — multiply sample by coefficient
(sum > 0.0f) ? sum : 0.0f	ReLU activation — clip negative to zero	Hard threshold — comparator with zero

The inner loop is the only mathematical operation a neural network performs:

```
sum += input[j] * weights[i * n_in + j];  /* this one line */
```

Repeated across millions of weights across multiple layers. That is inference. Nothing more, nothing less.

This is why AI accelerators and NPUs exist — they are hardware designed to execute this one MAC operation on thousands of weight-input pairs simultaneously, in parallel, at minimal power.

The Inference Engine — Running the Full Network

A complete network is the same layer function called repeatedly — output of each layer becomes input of the next:

```
void run_inference(
    const float *sensor_features,  /* 14 features from sensors */
    float       *decision)         /* [Normal, Warning, Fault] */
{
    float layer1_out[8];   /* intermediate — RAM only          */
    float layer2_out[4];   /* intermediate — RAM only          */

    run_layer(sensor_features, weights_l1, biases_l1,
              layer1_out, 14, 8);  /* 14 inputs → 8 neurons    */

    run_layer(layer1_out,      weights_l2, biases_l2,
              layer2_out, 8, 4);   /* 8 inputs  → 4 neurons    */

    run_layer(layer2_out,      weights_l3, biases_l3,
              decision,  4, 3);   /* 4 inputs  → 3 outputs     */
}
```

weights_l1, weights_l2, weights_l3 are the const arrays in flash generated by STM32Cube.AI. They never move to RAM. The layer buffers are the only RAM used.

The Inference Engine — The MCU Main Loop

The complete embedded AI system — from sensor reading to intelligent decision — in a main loop any firmware engineer recognises:

```
void main_loop(void)
{
    float features[14];  /* sensor feature vector            */
    float decision[3];   /* [Normal, Warning, Fault]         */

    while (1)
    {
        /* 1. Read sensors — your existing embedded code     */
        read_sensors(features);

        /* 2. Run inference — reads weights from flash       */
        run_inference(features, decision);

        /* 3. Act on result                                  */
        if      (decision[2] > 0.8f) trigger_alarm();
        else if (decision[1] > 0.6f) set_warning_flag();

        /* 4. Sleep until next sample                        */
        delay_ms(100);
    }
}
```

The Inference Engine — INT8 Version for MCU

On Cortex-M4 without FPU or with tight power budget — the INT8 quantized version runs. CMSIS-NN provides this. The concept is identical to Q-format on C2000:

```
/* q7_t = int8_t — identical concept to Q7 format on C2000 */
void run_layer_int8(
    const q7_t  *input,    /* INT8 input features             */
    const q7_t  *weights,  /* INT8 weights from flash         */
    const q31_t *biases,   /* INT32 — wider accumulator       */
    q7_t        *output,   /* INT8 output to next layer       */
    int n_in, int n_out, int shift) /* shift = Q-format scale */
{
    for (int i = 0; i < n_out; i++) {
        q31_t sum = biases[i];       /* 32-bit — no overflow  */
        for (int j = 0; j < n_in; j++)
            sum += (q31_t)input[j] * weights[i*n_in+j]; /* MAC*/
        sum >>= shift;               /* rescale — like IQmath  */
        output[i] = (q7_t)MAX(0, MIN(127, sum)); /* ReLU+clip */
    }
}
```

sum >>= shift is identical to IQtoF() in IQmath — rescaling the fixed point result back to the correct range.

INT8 Inference — The C2000 IQmath Parallel

The mathematics of INT8 neural network inference and Q-format fixed point on C2000 are not similar — they are identical. If you understood IQmath, you already understand CMSIS-NN.

Concept	C2000 IQmath	CMSIS-NN INT8 inference
Data type	q7_t — Q7 fixed point integer	q7_t — INT8 weight and activation
Multiply	IQmpy(a, b) — fixed point multiply	(q31_t)input[j] * weights[i*n_in+j]
Accumulator	q31_t — wider type to prevent overflow	q31_t sum — same reason
Rescale result	IQtoF(x) or right shift by Q factor	sum >>= shift — right shift by scale factor
Encode float	_IQ(0.00732) — float to Q format	Quantization step — float weight to INT8
Library	IQmath.lib	CMSIS-NN / STM32Cube.AI

The engineering discipline is identical. The only difference is the application — filtering a signal versus classifying a sensor pattern.

An embedded engineer who has worked with IQmath already has the mental model for understanding INT8 inference completely.

Model File and Inference Engine — Complete Picture

```
CLOUD / WORKSTATION
  Train → Quantize FP32→INT8 → Generate C arrays → Flash via JTAG/OTA/DFU
          ↓
STM32 FLASH
  ├── network_weights[]    ← const data — never executes
  ├── network_biases[]     ← const data — never executes
  └── inference engine     ← compiled C — reads the arrays
STM32 RAM
  ├── layer_buf[8]         ← intermediate calculation only
  └── features[14]         ← sensor input
MAIN LOOP
  read_sensors() → run_inference() → decision → action
```

Three sentences to carry out of this room:

The model is a const array in flash.

The engine is a MAC loop in C.

OTA updates the array — not the engine.

Edge AI — One Term, One Precise Meaning

Edge AI means running AI inference on a device local to where data is generated — without sending data to a remote server for processing. The word "Edge" describes where the inference happens — not what hardware runs it.

The hardware that can run Edge AI inference:

Hardware	Example	Edge AI?
Standard MCU — no NPU	STM32 Cortex-M4	Yes
MCU with DSP instructions	Cortex-M4F, M7	Yes
MCU with NPU	STM32N6 — Cortex-M55 + Ethos U55	Yes
DSP	TI C2000, C6000	Yes
FPGA	Xilinx, Intel — inference in hardware logic	Yes
Edge AI module	NVIDIA Jetson Orin	Yes
Cloud GPU	NVIDIA A100 — data centre	No — not edge

Three properties that always define Edge AI:

- **Local inference** — model runs on the device, not in the cloud
- **Data stays at source** — raw data never leaves the device, only decisions do
- **Autonomous operation** — device makes decisions without network dependency

Training vs Inference — A Critical Distinction

Two completely separate phases exist in every AI system. Confusing them leads to wrong hardware selection, wrong architecture, wrong design.

	Training	Inference
When	Once or periodically	Continuously on deployed system
What happens	Weights adjusted to reduce error	Frozen model applied to new input
Duration	Hours to months	Microseconds to milliseconds
Hardware needed	GPU clusters, TPUs, workstations	MCU, DSP, edge AI chip, NPU
Model changes?	Yes — weights update constantly	No — model is completely frozen
Who does it	ML engineer on powerful hardware	The deployed embedded system

One rule that never changes: Training always happens where compute is abundant. Inference happens where the decision is needed.

Training vs Inference — The Edge AI Reality

This is the standard production workflow for MCU-based Edge AI systems:

```
Cloud / Workstation
  ├── Collect sensor data from field
  ├── Train model — validate accuracy
  ├── Quantize: 32-bit float → INT8
  └── Package weights into firmware binary

          ↓  Deploy via OTA, JTAG, DFU, or supported mechanism

STM32 Cortex-M4 / Edge Device
  ├── Receives new model weights in flash
  ├── Inference engine reads frozen weights
  ├── Sensor data arrives continuously
  ├── Decision made in microseconds
  └── Model never changes until next deployment
```

The device does not learn. It applies what was already learned — until a new model arrives via the next deployment.

Six Ways a Machine Can Learn

ML is not one method. The learning approach chosen depends on what data is available and what problem needs solving.

Type	How it learns	Physical world example
Supervised	Learns from labeled examples — input paired with correct answer	Fault classification: vibration data labeled Normal / Warning / Fault
Unsupervised	Finds hidden patterns — no labels provided	Grouping machines by behavior without prior knowledge of failure modes
Semi-supervised	Mix of labeled and unlabeled data	Medical imaging where only some scans are annotated by doctors
Self-supervised	Creates its own labels from the data itself	GPT-4 — predicts next word using the text itself as the label
Reinforcement	Learns by trial, error, reward and punishment	Robot arm learning to assemble a component through repeated attempts
Transfer Learning	Reuses knowledge from one task and applies to another	Image model trained on millions of photos — adapted to detect weld cracks

Choosing the right learning type is an engineering decision — driven by what data you have, not by preference.

Choosing the Right learning Type

The right learning type is determined by the data available — not by preference or trend.

Data situation	Learning type to use
Have labeled data for every input?	Supervised learning
Have data but no labels?	Unsupervised learning
Have some labeled, most unlabeled?	Semi-supervised learning
Have vast raw data, no labels needed?	Self-supervised learning
Learning through interaction with environment?	Reinforcement learning
Have a related trained model already?	Transfer learning

Two important boundaries:

- Each learning type can use different mathematical algorithms — the choice of algorithm within each type is a separate engineering decision
- The number of algorithms is not fixed — researchers publish new ones continuously and many proprietary ones are never published at all

ML Algorithms — Industrial and Embedded Systems

SEGMENT 1 OF 4 — INDUSTRIAL / EMBEDDED SYSTEMS | MACHINE LEARNING APPROACH

Algorithm	Learning type	When to use	Embedded application example
Random Forest	Supervised	Structured sensor data — tabular features — robust to noise and outliers	Machine fault classification from vibration and temperature feature vector
Gradient Boosting — XGBoost	Supervised	Tabular features — highest accuracy on structured data — slightly heavier than Random Forest	Remaining useful life prediction from sensor degradation features
Support Vector Machine — SVM	Supervised	Small to medium labeled dataset — clear boundary between classes	Normal vs fault classification on bearing vibration features
Decision Tree	Supervised	Explainability required — rules must be auditable — safety critical context	Fault diagnosis where maintenance team must understand every decision
K-Nearest Neighbours — KNN	Supervised	Very simple baseline — small dataset — low latency not critical	Early fault detection prototype before moving to heavier model
Isolation Forest	Unsupervised	Anomaly detection — no fault labels available — deploy immediately on normal data	Machine health monitoring — Phase 1 — detect any deviation from normal
One-Class SVM	Unsupervised	Learn boundary of normal — flag anything outside — lightweight	Anomaly detection on STM32 — small feature vector — no fault data needed
Autoencoder — shallow	Unsupervised	Multi-sensor anomaly detection — captures non-linear relationships between features	14-feature vector anomaly detector — quantized INT8 — runs on Cortex-M4
Linear Regression	Supervised	Continuous output — predict a value not a class — simple relationship	Battery SoC estimation from voltage and temperature — initial baseline
Gaussian Process	Supervised	Small dataset — uncertainty estimate needed alongside prediction	RUL prediction with confidence interval — critical safety application

DL Algorithms — Industrial and Embedded Systems

SEGMENT 1 OF 4 — INDUSTRIAL / EMBEDDED SYSTEMS | DEEP LEARNING APPROACH

Algorithm	Learning type	When to use	Embedded application example
LSTM — Long Short-Term Memory	Supervised	Sequential sensor data — current state depends on history — battery, motor dynamics	Battery SoC estimation — LSTM learns voltage response history — Cortex-M4/M7
1D CNN — Convolutional Neural Net	Supervised	Raw waveform input — extract local temporal patterns — faster than LSTM	Bearing fault detection from raw vibration waveform — STM32 with Ethos NPU
Temporal Convolutional Network — TCN	Supervised	Long sequence dependencies — faster training than LSTM — parallelisable	Predictive maintenance from long sensor history — edge server deployment
Autoencoder — deep	Unsupervised	Complex multi-sensor anomaly — captures deep non-linear relationships	Phase 1 anomaly detection — 14 sensors — quantized — Cortex-M55 + Ethos
Variational Autoencoder — VAE	Unsupervised	Anomaly detection with uncertainty — generates synthetic fault data to augment training	Rare fault augmentation — supplement limited labeled fault dataset
Transformer — small	Supervised	Long range temporal dependencies — sensor fusion across multiple channels	Multi-sensor industrial anomaly — Jetson Orin deployment — larger model
GRU — Gated Recurrent Unit	Supervised	Sequence data — lighter than LSTM — similar accuracy — better for constrained hardware	RUL estimation on edge server — lighter than LSTM — faster inference

ML Algorithms — Audio and Speech

SEGMENT 2 OF 4 — AUDIO AND SPEECH | MACHINE LEARNING APPROACH

For audio — standard ML algorithms do not operate on raw waveforms. They operate on extracted features — MFCCs, spectral centroids, zero crossing rate — computed from the audio by signal processing first.

Algorithm	Learning type	When to use	Audio application example
Gaussian Mixture Model — GMM	Unsupervised	Model statistical distribution of audio features — speaker or sound class	Industrial sound anomaly — learn normal machine acoustic signature — no labels
Hidden Markov Model — HMM	Supervised	Sequential audio features — temporal structure matters — classic speech approach	Simple voice command recognition — small vocabulary — MCU deployment
Random Forest on MFCCs	Supervised	Labeled audio clips — MFCC features extracted — moderate dataset	Industrial sound classification — grinding vs normal — bearing acoustic fault
SVM on MFCCs	Supervised	Small labeled audio dataset — good generalisation — explainable boundary	Keyword detection — binary — wake word present or absent — small dataset
K-Means clustering	Unsupervised	Group audio clips by acoustic similarity — no labels — exploratory	Cluster machine operating modes by acoustic signature — before labeling

MFCCs — Mel Frequency Cepstral Coefficients — are the standard feature extraction for audio ML. They compress the spectral content of an audio frame into 13 to 40 numbers that capture perceptually relevant patterns.

DL Algorithms — Audio and Speech

SEGMENT 2 OF 4 — AUDIO AND SPEECH | DEEP LEARNING APPROACH

DL for audio typically converts the waveform to a spectrogram — a 2D time-frequency image — then applies image classification or sequence models. The network learns its own features.

Algorithm	Learning type	When to use	Audio application example
CNN on spectrogram	Supervised	Keyword spotting — audio classification — spectrogram treated as image — proven on MCU	Wake word detection — Hey Device — STM32 Cortex-M4 — 50KB INT8 model
DS-CNN — Depthwise Separable CNN	Supervised	Keyword spotting optimised for MCU — 5x fewer parameters than standard CNN	Keyword spotting — Google Speech Commands dataset — Cortex-M4 / ESP32-S3
CRNN — CNN + RNN combined	Supervised	Audio with temporal context — sound event detection — continuous audio stream	Industrial sound event detection — Jetson or Raspberry Pi deployment
RNN / LSTM on MFCC sequence	Supervised	Sequential audio features — temporal dependencies across frames	Continuous voice command recognition — small vocabulary industrial control
Transformer — audio	Supervised	Large dataset — complex audio — state of the art accuracy — larger model	Industrial acoustic anomaly detection — edge server — Jetson Orin
Autoencoder on spectrogram	Unsupervised	Audio anomaly detection — no fault labels — learn normal acoustic pattern	Machine acoustic health — learn normal running sound — flag any deviation

ML Algorithms — Vision and Imaging

SEGMENT 3 OF 4 — VISION AND IMAGING | MACHINE LEARNING APPROACH

Standard ML on images requires hand-crafted feature extraction — traditional computer vision methods — before the ML algorithm. This was the standard approach before deep learning made direct pixel-to-decision feasible.

Algorithm	Feature extraction	When to use	Vision application example
SVM on HOG features	Histogram of Oriented Gradients	Object detection — person, vehicle — limited compute — no GPU available	Person present / absent — fixed camera — STM32 with Ethos NPU
Random Forest on colour histograms	Colour and texture statistics	Simple appearance classification — controlled lighting — limited dataset	Product colour or surface quality check — industrial line — constrained hardware
SVM on Haar features	Haar wavelet features — OpenCV	Face or object detection — real time — very limited compute	Presence detection — face detection — legacy embedded vision systems
K-Means on pixel statistics	Statistical region features	Unsupervised segmentation — find regions of interest — no labels	Thermal image segmentation — find hotspots — group temperature regions
PCA + SVM	PCA dimensionality reduction	High dimensional image features — reduce then classify — small dataset	Thermal camera anomaly — PCA finds dominant patterns — SVM classifies

For most modern embedded vision applications — DL approaches on the next slide now outperform traditional ML approaches and run efficiently on MCUs with NPU support.

DL Algorithms — Vision and Imaging

SEGMENT 3 OF 4 — VISION AND IMAGING | DEEP LEARNING APPROACH

Algorithm	Learning type	When to use	Embedded vision application example
MobileNet V2 / V3	Supervised	Image classification on MCU — specifically designed for constrained hardware	Defect present / absent — weld inspection — STM32N6 with Ethos U55
EfficientNet-Lite	Supervised	Higher accuracy than MobileNet — still deployable on edge — more compute	Component presence verification — PCB inspection — Raspberry Pi / Jetson
YOLO — You Only Look Once — Nano/Tiny	Supervised	Real time object detection — bounding box + class — edge optimised variants	Person detection, vehicle counting, assembly verification — Jetson Orin
SSD — Single Shot Detector	Supervised	Object detection — faster than YOLO on some hardware — good accuracy tradeoff	Industrial object detection — Jetson or powerful edge module
U-Net — lightweight	Supervised	Image segmentation — pixel level classification — find exact defect region	Crack or defect segmentation — thermal map hotspot isolation — Jetson
CNN Autoencoder on images	Unsupervised	Visual anomaly detection — no defect labels — learn normal appearance	Surface inspection — learn normal texture — flag any visual deviation
MobileNet + Transfer Learning	Supervised — transfer	Limited labeled data — reuse ImageNet features — fine tune on own images	Custom defect classifier — 50 to 200 labeled images — Cortex-M55 + Ethos

ML Algorithms — Sequence and Time Series

SEGMENT 4 OF 4 — SEQUENCE AND TIME SERIES | MACHINE LEARNING APPROACH

Time series ML uses statistical and window-based features computed from sensor history — trends, slopes, rolling statistics — rather than raw sequence values directly.

Algorithm	Learning type	When to use	Physical system application example
Random Forest on window features	Supervised	Sensor trend classification — rolling mean, slope, variance as features — robust	Degradation stage classification — early, moderate, severe — from trend features
Gradient Boosting — XGBoost	Supervised	RUL regression — highest accuracy on engineered trend features	Remaining useful life prediction — battery, bearing — from degradation trajectory
ARIMA — statistical model	Unsupervised / forecasting	Univariate time series forecasting — stationary signal — no external features	Energy consumption forecasting — predict next hour load — single sensor
Isolation Forest on windows	Unsupervised	Anomaly detection on sensor windows — flag abnormal trends — no fault labels	Process anomaly detection — flag unusual trend before fault threshold crossed
Linear Regression on features	Supervised	Simple continuous prediction — linear relationship — explainable output	Simple degradation trend extrapolation — when will threshold be crossed
Prophet — Facebook	Supervised / forecasting	Seasonal time series — known periodicity — interpretable components	Energy or load forecasting with daily and weekly patterns — edge server

DL Algorithms — Sequence and Time Series

SEGMENT 4 OF 4 — SEQUENCE AND TIME SERIES | DEEP LEARNING APPROACH

Algorithm	Learning type	When to use	Physical system application example
LSTM	Supervised	Sequential sensor data — current output depends on history — battery, motor dynamics	Battery SoC / SoH — learns voltage response to current history — Cortex-M4/M7
GRU — Gated Recurrent Unit	Supervised	Similar to LSTM — fewer parameters — better for constrained hardware	RUL estimation — lighter than LSTM — faster inference — edge server
1D CNN on raw sequence	Supervised	Local temporal patterns — faster training than RNN — parallelisable	Bearing RUL from raw vibration sequence — STM32 with Ethos NPU
TCN — Temporal Convolutional Net	Supervised	Long range temporal dependencies — parallelisable — often outperforms LSTM	Multi-sensor process monitoring — long history — Jetson deployment
Transformer — time series	Supervised	Long sequence — complex inter-sensor relationships — large dataset available	Multi-machine fleet health — cloud or powerful edge server — large model
LSTM Autoencoder	Unsupervised	Sequential anomaly detection — learn normal temporal pattern — flag deviation	Process anomaly in sensor stream — no labels — detect unusual sequence pattern
N-BEATS / N-HiTS	Supervised / forecasting	Pure time series forecasting — no external features needed — strong baseline	Energy demand forecasting — production scheduling — edge server or cloud

Choosing the Right Algorithm — Decision Guide

The right algorithm is the simplest one that solves the problem within the hardware and data constraints — not the most sophisticated one available.

Start here — your situation	Recommended algorithm	Why
Structured sensor features — labeled data — need explainability	Decision Tree or Random Forest	Auditable rules — maintenance team can understand every decision
Structured sensor features — labeled data — need best accuracy	Gradient Boosting — XGBoost	Consistently highest accuracy on tabular sensor data
No fault labels — deploy now on normal data	Isolation Forest or shallow Autoencoder	Unsupervised — only needs normal data — immediate deployment
Raw vibration or audio waveform — labeled data	1D CNN	Learns temporal patterns from raw signal — efficient on MCU with NPU
Camera or thermal image — labeled data	MobileNet + Transfer Learning	Proven on embedded vision — tiny model — Cortex-M55 + Ethos U55
Sequential sensor data — history matters	LSTM or GRU	Captures temporal dependencies — battery, motor, process dynamics
Very limited labeled data — related public dataset exists	Transfer Learning + fine tuning	Reuse public dataset knowledge — fine tune on your small dataset
Safety critical — decision must be fully explainable	Decision Tree or Rule-Based AI	Full auditability — every decision traceable to a specific condition

Recommended Books and Online Resources

Books:

Title	Authors	Why valuable for physical systems engineers
Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow	Aurélien Géron — O'Reilly	Best practical ML/DL reference — algorithm by algorithm with code — engineer friendly
TinyML	Pete Warden, Daniel Situnayake — O'Reilly	ML specifically on microcontrollers — STM32, Arduino, deployment — directly relevant
Deep Learning	Goodfellow, Bengio, Courville	Definitive theoretical reference — free online at deeplearningbook.org
Machine Learning Engineering	Andriy Burkov	Deploying ML in production — bridges research and engineering practice

Online resources:

Resource	URL	Best for
Scikit-learn documentation	scikit-learn.org	Best ML algorithm reference — examples for every algorithm
Edge Impulse documentation	docs.edgeimpulse.com	End to end TinyML — STM32, Arduino, Raspberry Pi to deployment
TensorFlow tutorials	tensorflow.org/tutorials	DL with deployment focus — TFLite for edge
Papers With Code	paperswithcode.com	Every algorithm with benchmarks, papers, and code
Deep Learning Book — free	deeplearningbook.org	Full text of Goodfellow et al. — free online

SECTION 03

Three Approaches to AI

03

Rule-Based AI, ML Based AI, Deep Learning Based AI

Three Approaches to AI on an Embedded System

Any embedded system using one of these three approaches to make intelligent decisions can be called an AI system.

Approach	When to Use	Examples
1 – Rule-Based AI	Problem is mathematically tractable – system behaviour can be modelled with equations or logical rules	Kalman Filter, Fuzzy Logic, If-Then-Else, PID with decision logic
2 – Machine Learning based AI	Input is structured and measurable, but relationships are too complex or non-linear for hand-crafted rules	Decision Tree, Random Forest, SVM, shallow Neural Networks
3 – Deep Learning based AI	Input is high dimensional and unstructured – images, audio, video – where standard ML algorithms cannot extract meaningful patterns	CNN, RNN, Transformer

The choice is driven by the nature of the input, the complexity of the decision, the hardware available, and the amount of data that can be collected.

All three are valid AI implementations. None is superior — the right choice is always the simplest approach that solves the problem within the hardware constraints.

Approach 1 — Rule-Based AI

The engineer writes the intelligence explicitly as mathematical or logical rules. The system follows those rules precisely. No data collection, no training, no model file.

When is this the right choice?

Condition	Detail
Rules are known and stable	System behaviour can be described mathematically or logically
Hardware is severely constrained	No memory headroom for a model file
Explainability is critical	Every decision must be traceable to a specific rule

Common implementations:

Implementation	What it does	Typical application
If-Then-Else logic	Simplest — threshold decisions	Over-temperature shutdown, pressure alarms
PID with decision logic	Control + intelligent mode switching	Motor speed control with fault detection
Kalman Filter	Optimal state estimation under sensor noise	Drone stabilisation, GPS smoothing
Fuzzy Logic	Handles imprecise or uncertain inputs using linguistic rules	Washing machine control, HVAC comfort control

When system complexity grows beyond what a human can model — that is precisely where Approach 2 begins.

Approach 2 — Machine Learning Based AI

When system behaviour is too complex for hand-crafted rules — but the input data is structured and measurable — Machine Learning discovers the patterns the engineer cannot write explicitly.

Structured data in the embedded world:

- Vibration amplitude at specific frequencies — from an accelerometer
- Motor current signature — from a current sensor
- Temperature over time — from a thermocouple
- Pressure, flow rate, RPM — from industrial sensors

Common embedded applications:

Application	Structured Input	Intelligent Decision
Machine health monitoring	Vibration, current, temperature features	Normal / Warning / Fault
Predictive maintenance	Motor current signature features	Remaining useful life estimate
Battery management	Cell voltage, current, temperature	State of Charge / Health
Anomaly detection	Any sensor feature stream	Normal / Anomaly flag

When input becomes unstructured — images, audio, video — standard ML algorithms cannot extract meaningful patterns. That is where Approach 3 begins.

Approach 3 — Deep Learning Based AI

Deep Learning is a specialised subset of Machine Learning that uses multi-layer neural networks to automatically learn patterns from high dimensional unstructured input — such as images, audio, and video.

Where it fits:

`Artificial Intelligence → Machine Learning → Deep Learning ← Approach 3`

The one sentence that defines when Approach 3 is needed:

When the input to the system is too high dimensional and unstructured for a human engineer to extract meaningful features by hand — and too complex for standard ML algorithms to learn from directly.

Structured Input — Approach 2	Unstructured Input — Approach 3
Temperature — one number	Image — thousands to millions of pixel values
Vibration RMS — one number	Audio waveform — thousands of samples per second
FFT amplitude — one number	Video frame — millions of values per frame
14 feature vector — engineered by human	Raw thermal image — spatial temperature map

Why Standard ML Cannot Handle Unstructured Input

A standard ML algorithm sees a flat list of numbers. It cannot reason about spatial relationships, temporal patterns, or hierarchical structure within the input.

Input	Values per sample	What gets lost with flat ML
Grayscale image 320×240	76,800	Spatial relationships between pixels
Colour image 640×480	921,600	Shape, colour gradients, object structure
Audio — 1 second	16,000 to 44,100	Temporal patterns across time
Thermal camera frame	19,200 to 307,200	Spatial temperature gradients

Deep Learning preserves and exploits these relationships. Standard ML discards them.

How a Deep Neural Network Processes Input

Each layer learns increasingly abstract representations — automatically, from data alone. The engineer does not define these representations.

```
Raw input
  ↓  Layer 1 — learns edges and basic patterns
  ↓  Layer 2 — learns shapes and combinations
  ↓  Layer 3 — learns parts and structure
  ↓  Layer 4 — learns complex relationships
  ↓
Output — classification or detection result
```

The critical difference from Approach 2:

Approach 2 — ML	Approach 3 — Deep Learning
Engineer extracts features by hand	Network discovers features automatically
14 numbers fed to algorithm	Raw pixels or samples fed directly
Feature quality depends on engineer	Feature quality improves with more data

Deep Learning Architectures — Right Tool for Right Input

Different architectures are designed for different input types. Choosing the right one is an engineering decision.

Architecture	Designed for	Embedded application example
CNN — Convolutional Neural Network	Images — spatial pattern recognition	Thermal camera defect detection
1D CNN	Raw waveforms — time series	Vibration based fault detection
RNN / LSTM	Sequences — temporal patterns	Predictive maintenance trends
Autoencoder	Anomaly detection — unsupervised	Machine health on STM32
MobileNet	Images on constrained hardware	Person detection on Cortex-M55 + Ethos
Transformer	Language and voice commands	Voice control on edge device

MobileNet and similar lightweight architectures are specifically designed for embedded deployment — accuracy balanced against memory and compute constraints.

When is Deep Learning the Right Choice?

Condition	Implication
Input is an image or video	CNN is the right starting point
Input is raw audio or complex time series	1D CNN or RNN
Input is voice command or natural language	Transformer based architecture
Feature extraction by hand is not feasible	Deep Learning learns features automatically
Large labeled dataset is available	Deep Learning has enough data to generalise
Hardware has NPU or GPU	Deep Learning runs efficiently

The honest data reality:

Approach	Labeled examples typically needed
Rule-Based AI	None — rules written by engineer
ML Based AI — Approach 2	Hundreds to low thousands
Deep Learning Based AI — Approach 3	Thousands to tens of thousands

More powerful — but more hungry for data. Transfer Learning is the practical path when large datasets cannot be collected.

Approach 3 — What Changes and What Does Not

What changes:

- Input is raw and unstructured — images, audio, waveforms
- Model architecture is deeper — more layers, more weights
- Training data requirement is significantly higher
- Hardware needs NPU or GPU for efficient inference
- Model file is larger — more flash memory required

What does NOT change:

- Training always on workstation or cloud — never on MCU
- Model is frozen after training — inference only on edge
- Quantization mandatory — INT8 for embedded deployment
- Deployment via OTA, JTAG, DFU, or supported mechanism
- Inference engine runs on target — CMSIS-NN, TFLite Micro

The end-to-end journey remains identical across all three approaches. What changes is the complexity of the model and the nature of the input it handles.

SECTION 03-A

ML Based AI – Case Study

BMS Case Study — ML Based AI Implementation

ML Based AI — Battery Management System Case Study

A BMS is a compelling example where ML Based AI offers measurable advantages over the Rule-Based approaches — particularly as battery chemistry, age, and operating conditions introduce complexities that are difficult to model mathematically.

BMS Function	Traditional Approach	ML Based AI Approach
State of Charge (SoC)	Kalman Filter variants — EKF, UKF	Neural network trained on charge/discharge cycles
State of Health (SoH)	Empirical degradation models	ML learning degradation patterns from real field data
Remaining Useful Life	Physics based ageing models	ML trained on historical battery ageing datasets
Fault Detection	Threshold rules and Kalman residuals	ML anomaly detection on cell voltage and temperature

The emerging direction — Hybrid Architecture:

Component	Approach	Role
Real-time SoC estimation	Kalman Filter — Rule-Based AI	Mathematical rigour, explainability, certifiability
Battery model parameter adaptation	ML Based AI	Learns how parameters shift with age and temperature
Fault pattern detection	ML Based AI	Detects subtle signatures too complex for threshold rules

The BMS is one of the few embedded systems where all three AI approaches can coexist — often in the same product.

BMS Case Study — ML Based AI Implementation

ML Based AI for BMS learns the relationships between measurable signals and true battery state directly from data — without requiring an explicit electrochemical model.

The Data Inputs to ML:

Input feature	Physical meaning	Why it matters
Cell voltage	Instantaneous terminal voltage	Primary SoC indicator
Pack current	Charge / discharge rate	Affects internal resistance and heat generation
Cell temperature	Local thermal condition	Dramatically affects electrochemical behaviour
dV/dt	Rate of voltage change	Captures dynamic response — diffusion effects
Cumulative Ah	Coulomb count since last known state	Integrates current history
Cycle count	Number of completed charge cycles	Primary ageing indicator
Time in service	Calendar age	Captures storage degradation
Voltage spread	Max cell voltage − min cell voltage	Imbalance indicator — early fault signal
Temperature spread	Max cell temp − min cell temp	Thermal non-uniformity — fault indicator

What ML Algorithm is Typically Used:

Function	Algorithm	Why this choice
SoC estimation	LSTM — Long Short Term Memory	Battery is a dynamic system — current state depends on history. LSTM captures temporal dependencies naturally
SoH estimation	Random Forest or Gradient Boosting	Tabular input — cycle count, capacity fade metrics. No temporal dependency needed
RUL prediction	Gradient Boosting or shallow LSTM	Regression on degradation trajectory — tabular + trend
Anomaly / fault detection	Autoencoder or Isolation Forest	Unsupervised — learns normal cell behaviour, flags deviation
Cell balancing optimisation	Reinforcement Learning	Learns optimal balancing strategy from interaction with pack

Why LSTM for SoC — The Key Insight:

A battery is not a static system. The current SoC depends not just on the current voltage reading — it depends on what has happened over the past seconds and minutes — recent current pulses, relaxation time, thermal history.

Simple ML (Random Forest):
Input: [V, I, T] → SoC estimate
Problem: ignores history — same V, I, T in different conditions
gives same answer — often wrong

LSTM:
Input: sequence of [V, I, T] over past N seconds → SoC estimate
Advantage: learns how voltage responds over time
accounts for recent current history
accounts for relaxation dynamics

This is why LSTM — a recurrent architecture designed specifically for sequences — outperforms static ML algorithms for SoC estimation.

BMS Case Study — The Training Data and Deployment Reality

What Training Data Is Needed:

Data type	How collected	Volume needed
Charge / discharge cycles	Laboratory cycling at multiple temperatures	Hundreds of cycles minimum
Temperature sweep data	Cycling at -20°C, 0°C, 25°C, 45°C	At each temperature
Ageing data	Accelerated cycling to capture degradation	Ideally 20 to 80% SoH range
Reference SoC	Precision coulomb counter or reference method	Labels for every sample
Fault scenarios	Controlled cell degradation or simulation	For fault detection model

BMS Case Study — The Training Data and Deployment Reality

The Honest Data Challenge:

Data type	How collected
Ageing data is slow to collect	Real ageing data takes months to years — accelerated ageing introduces artefacts
Temperature range coverage	Laboratory testing at every operating temperature is time consuming
Reference labeling	True SoC ground truth requires precision reference equipment
Chemistry specificity	Model trained on NMC cells does not transfer directly to LFP cells
Transfer to production cells	Laboratory characterisation cells may not match production cell behaviour exactly

BMS Case Study — The Training Data and Deployment Reality

Deployment on Embedded Hardware:

Data type	How collected
LSTM model — SoC	Quantized INT8 — runs on Cortex-M4 or M7 — inference every 100ms
Random Forest — SoH	Lightweight — runs on Cortex-M4 — inference every charge cycle
Autoencoder — fault detection	Small INT8 model — runs continuously on Cortex-M4
Model update mechanism	OTA, JTAG, DFU, or any supported update mechanism
Retraining trigger	Accuracy drift detected — field data collected — retrain on workstation

The Hybrid Architecture — Best of Both:

The most robust production BMS combines both approaches:

Kalman Filter
- **Real time SoC estimation — deterministic, certifiable**
- **Uses battery model parameters**
- **Fast — runs every millisecond**

↑ **parameters updated by**

LSTM / ML layer
- **Learns how model parameters drift with age and temperature**
- **Updates R0, OCV curve parameters periodically**
- **Runs less frequently — every few seconds**
- **Makes Kalman Filter self-adapting over battery life**

The Kalman Filter provides mathematical rigour and functional safety certifiability. The ML layer provides adaptability over the battery service life. Neither alone is as powerful as both together.

BMS Case Study — Comparison at a Glance

Aspect	Traditional — Kalman Filter	ML Based AI	Hybrid
SoC accuracy — new battery	High	High	Highest
SoC accuracy — aged battery	Degrades — fixed model	Maintained — adapts	Maintained
Data needed to build	Battery characterisation tests	Hundreds of labeled cycles	Both
Explainability	Full — traceable to equations	Limited — weights opaque	Partial
Functional safety path	Well established	Evolving — challenging	Possible with frozen ML
Model update after deployment	Not applicable	OTA retraining	OTA for ML layer
Compute on MCU	Very low	Low to medium	Medium
Handles new cell chemistry	Needs reparametrisation	Needs retraining	Needs both

The BMS is one of the most compelling embedded AI applications precisely because the problem — estimating hidden internal state from noisy external measurements on an ageing electrochemical system — is exactly the class of problem where ML offers measurable, quantifiable improvement over traditional approaches.

SECTION 03-B

03

DL Based AI – Case Study

Deep Learning Based AI — Keyword Spotting Case Study

DL Based AI — Keyword spotting Case Study

Keyword spotting is the task of detecting a specific spoken word or phrase in a continuous audio stream — in real time, on a battery powered embedded device, without cloud connectivity.
This is a problem that cannot be solved with Rule-Based AI or standard ML. The input is raw audio — an unstructured waveform with no fixed feature positions — where the pattern of interest is buried in noise, varies with speaker, accent, and environment, and must be detected in milliseconds..

Why Approach 1 and Approach 2 Cannot Solve This:

Approach	Why it fails for keyword spotting
Rule-Based AI	No human can write rules for what "Hey Device" sounds like across all speakers, accents, distances, and noise levels
ML Based AI — Approach 2	Feature extraction by hand loses the temporal and spectral patterns that distinguish one word from another
Deep Learning — Approach 3	Learns spectral and temporal patterns directly from thousands of audio examples — the only practical approach

The Input — Why It Is Unstructured

A one second audio clip sampled at 16kHz contains 16,000 raw amplitude values. No two utterances of the same word produce the same waveform. The pattern exists in the relationship between values across time and frequency — not in any single measurable number.

Raw audio waveform → 16,000 values per second

↓

Convert to spectrogram → time-frequency representation

↓

Feed to CNN → learns patterns in the spectrogram image

↓

Output → Keyword detected / Not detected

The Deep Learning Pipeline:

Step	What happens	Where
Audio capture	Microphone samples at 16kHz continuously	On device
Windowing	Sliding 1 second window extracted	On device
Spectrogram	FFT converts window to time-frequency image	On device
CNN inference	Trained model classifies spectrogram	On device — INT8
Decision	Keyword detected → trigger action	On device
Latency	End to end — under 10 milliseconds	On device

The Model — What Was Trained:

The CNN was trained on thousands of labeled audio examples:

- Positive examples — many speakers saying the target keyword
- Negative examples — other words, background noise, silence

After training — quantized to INT8 — the model fits in approximately 50KB of flash and runs inference in under 5ms on a Cortex-M4.

Hardware Reality:

Hardware	Model size	Inference Time	Power
STM32 Cortex-M4 — no NPU	~50KB INT8	~8ms	~5mW active
STM32 Cortex-M55 + Ethos U55	~200KB INT8	~1ms	~2mW active
ESP32-S3 with vector instructions	~100KB INT8	~3ms	~4mW active

What Makes This Deep Learning and Not ML:

- The CNN does not work on a hand-crafted feature vector.
- It works directly on the spectrogram — a 2D image of time versus frequency.
- The convolutional layers learn which spectral and temporal patterns distinguish the keyword from everything else.
- No human defined those patterns.
- The network discovered them from thousands of training examples.

This is precisely the class of problem Deep Learning was designed for — high dimensional input where the meaningful patterns cannot be written as rules or extracted as simple features by a human engine

The Contrast With the BMS Case Study

	BMS – Approach 2	Keyword Spotting – Approach 3
Input	Cell voltage, current, temperature, cycle count — structured, physically measurable	16,000 raw audio samples per second — unstructured waveform
Feature extraction	Not needed — measured values are directly meaningful	Not needed — CNN learns features automatically from spectrogram
Why no manual features needed	Each input already carries known physical meaning	No human can define what spectral patterns distinguish one word from another
Algorithm	Random Forest, SVM, or shallow Neural Network	Convolutional Neural Network — CNN
Training data needed	Hundreds of labeled charge/discharge cycles	Thousands of labeled audio clips — multiple speakers, noise conditions
Model size on MCU	~20KB INT8	~50KB INT8
Inference time	~1ms	~5 to 8ms
DL Need	No, as the input is already structured	Yes as the raw audio has no structure

Both run on the same STM32 Cortex-M4. Both use a frozen INT8 model deployed via OTA, JTAG, DFU, or any supported update mechanism. The difference is not the hardware or the deployment — it is the nature of the input and what the model had to learn

SECTION 02

The AI Toolchain

02

Frameworks, Model file, Inference engine, Quantization

ML Frameworks — What They Are and Why They Exist

An ML framework is a software library that provides the complete environment for building, training, evaluating, and exporting models.

What a framework provides:

- Pre-built learning algorithm implementations — neural network layers, loss functions, optimisers
- Automatic gradient computation — the mathematics of training handled automatically
- GPU acceleration — training on NVIDIA hardware without writing CUDA code
- Tools to build, train, validate, and export models to deployment formats

Framework	Created by	Primary use	Known for
PyTorch	Meta (Facebook)	Research and production	Most used in research today — flexible and intuitive
TensorFlow	Google	Production deployment	Strong mobile and edge export tools — TFLite
Keras	Google	Simplified interface	Beginner friendly — runs on top of TensorFlow
JAX	Google	High performance research	Functional style — very fast on TPUs

For embedded engineers: the framework is used only during training on the workstation or cloud. It never runs on the MCU. The MCU runs the inference engine.

From Framework to Edge Device — The Conversion Chain

A model trained in PyTorch or TensorFlow cannot run directly on an MCU. It must be converted and compressed through a chain of steps.

```
Train in PyTorch / TensorFlow  (32-bit float, full precision)
        ↓
Export to interchange format  (.onnx or .tflite)
        ↓
Quantize  (32-bit float weights → INT8 integers)
        ↓
Convert to target format  (CMSIS-NN, STM32Cube.AI, TFLite Micro)
        ↓
Generate C weight array  (sits in flash as const data)
        ↓
Compile inference engine for target ISA  (ARM Cortex-M, RISC-V...)
        ↓
Flash to device  (OTA, JTAG, DFU, or supported mechanism)
```

STM32Cube.AI automates steps 3 through 6 for STM32 targets specifically — reducing the chain to import, convert, and flash.

The Model File — What Is Inside

A model file contains two things:

- Architecture definition — how many layers, how they connect, what operations each layer performs
- Weights and biases — the actual learned numbers, one array per layer

Why so many file formats?

Format	Used for	Created by
.pt / .pth	PyTorch training and research	Meta
.onnx	Cross-platform interchange — framework agnostic	Microsoft / community
.tflite	Mobile and embedded deployment	Google
.gguf	Compressed LLM edge deployment	Open source
.safetensors	Safe fast loading — modern standard	HuggingFace

The field is young and fragmented — like embedded formats before Intel HEX and Motorola SREC became standards. Consolidation will come.

The Inference Engine — A C/C++ Program

The inference engine is the software that loads model weights and performs calculations on new input data to produce a decision.

What it does — in four steps:

- Load model weights from flash or storage into RAM
- Take new input — sensor data, image, audio sample
- Perform matrix multiplication layer by layer — the core MAC operation
- Produce output — classification result, anomaly score, prediction

---

The core operation — repeated billions of times:

```
Matrix multiplication  +  Activation function  →  repeated per layer
```

This is multiply-accumulate — MAC — the same operation DSPs are optimised for. AI accelerators and NPUs exist specifically to perform MACs at extreme speed and efficiency.

The inference engine is ISA dependent — it must be compiled for the target processor. The model weights are not ISA dependent — they are universal data.

Inference Engines — Which One for Which Hardware

Engine	Language	Target hardware	Created by
CMSIS-NN	C	ARM Cortex-M MCUs	ARM
STM32Cube.AI	C	STM32 family specifically	STMicroelectronics
TFLite Micro	C++	Any MCU — portable	Google
ESP-DL	C++	ESP32 / ESP32-S3	Espressif
ONNX Runtime	C++	Cross-platform	Microsoft
TensorRT	C++	NVIDIA GPU / Jetson	NVIDIA
llama.cpp	C/C++	CPU edge — LLM inference	Open source

Same model weights — different inference engine — different target processor.

The weights are universal data. The engine is ISA-specific compiled code. This is why the same trained model can be deployed to an STM32, an ESP32, a Raspberry Pi, or a Jetson — using a different engine for each.

Quantization — The Cloud to Edge Bridge

Models trained in the cloud use 32-bit floating point numbers — high precision, high memory, high compute. Embedded devices have kilobytes of RAM and no FPU. A direct deployment is impossible. Quantization solves this.

Property	Cloud — training	Edge — after quantization
Weight format	32-bit float (FP32)	8-bit integer (INT8)
Memory per weight	4 bytes	1 byte — 4x reduction
Compute needed	FPU or GPU	Integer MAC — no FPU needed
Accuracy	Full	Slight reduction — acceptable for most applications
Speed on MCU	Not feasible	Runs efficiently on bare metal C

The result: a model that was trained on a cloud GPU with terabytes of RAM can run on a microcontroller with 256KB of RAM.

Quantization — DSP Fixed Point Thinking Applied to AI

Engineers who have worked with TI C2000 DSP and IQmath libraries already understand quantization completely. The concept is identical.

Concept	C2000 DSP world	AI quantization world
The problem	No FPU — must do float operations in fixed point	MCU has no FPU — must compress float weights to run efficiently
The solution	Scale float to Q-format fixed point integer	Scale FP32 weights to INT8 integers
The library	IQmath library handles scaling and arithmetic	CMSIS-NN / STM32Cube.AI handle quantized MAC operations
Precision trade-off	Small — managed by Q-format selection	Small — managed by calibration during quantization
The result	Efficient DSP operation without FPU	Efficient MCU inference without FPU

The mathematics are equivalent. The engineering discipline is identical. Quantization is fixed-point DSP thinking applied to AI weights.

If you understood IQmath on C2000 — you already understand AI quantization.

SECTION 04

The Hardware Spectrum

04

TinyML, Edge AI, NPU, Ethos NPU, OTA Model Update

The Hardware Spectrum — Where Does Your System Sit?

Every AI implementation sits somewhere on a hardware spectrum — from a microcontroller with kilobytes of RAM to a cloud GPU cluster with terabytes of memory.

Most constrained ⟷ Most capable

STM32 MCU → ESP32 → Raspberry Pi → Jetson → Cloud GPU
(TinyML) (TinyML) (Edge AI) (Edge AI+) (Training)

TinyML	Edge AI	Edge AI+	Training
Inference only	Inference only	Inference +	Training +
KBs of RAM	MBs of RAM	learning	large inference

The governing engineering question:

Where does the decision need to be made — and what are the power, latency, privacy, and connectivity constraints at that location?

The answer to that question determines your position on the spectrum — not the capability of the algorithm.

The Hardware Spectrum — Specifications at Each Level

Hardware	Core	RAM	AI Framework	Role
STM32 Cortex-M4	180 MHz ARM	256 KB	CMSIS-NN, Cube.AI	TinyML inference
STM32 Cortex-M55 + Ethos U55	480 MHz + NPU	512 KB	Cube.AI, TFLite Micro	TinyML + NPU acceleration
ESP32-S3	240 MHz Xtensa	512 KB	ESP-DL, TFLite Micro	TinyML inference
Raspberry Pi 5	2.4 GHz ARM A76	4 to 8 GB	TFLite, ONNX Runtime	Edge AI inference
NVIDIA Jetson Orin	12-core ARM + GPU	8 to 64 GB	TensorRT, PyTorch	Edge AI — vision, LLM
Apple M4	ARM + Neural Engine	16 to 128 GB	CoreML, PyTorch	Edge AI + fine tuning
Cloud GPU — NVIDIA A100	6912 CUDA cores	80 GB VRAM	PyTorch, TensorFlow	Full model training

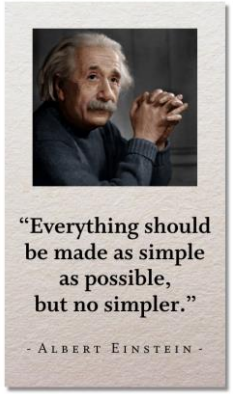
TOPS — Tera Operations Per Second — is the standard measure of AI inference throughput. The Ethos U55 delivers 256 TOPS per watt — designed specifically for battery powered embedded inference.

Check this Video from ARM : tinyML Development with Tensorflow Lite using CMSIS-NN and Ethos-U55
(Done on Hardware Arduino nano 33 BLE sense + Arducam mini 2MP+ camera)

Choosing Your Position on the Hardware Spectrum

The right hardware is the least capable hardware that meets all the system requirements — not the most powerful available.

Requirement	Points toward
Battery powered — years of operation	STM32 MCU — microamp sleep current
Sub-millisecond inference latency	MCU with NPU — Cortex-M55 + Ethos
Image or video inference — low power	Jetson Orin NX — power optimised GPU
Full Linux OS needed	Raspberry Pi or Jetson
LLM inference at edge	Jetson Orin or Apple M4
Maximum privacy — no data leaves device	MCU or edge module — any level



Start with requirements. Let requirements choose the hardware. Never let hardware availability drive the AI architecture.

TinyML — AI on Microcontrollers

TinyML is the practice of running ML inference on microcontrollers — devices with kilobytes of RAM, no operating system, and milliwatt to microwatt power budgets.

Constraint	Typical limit	Implication
RAM	16 KB to 512 KB	Model activations must fit during inference
Flash	256 KB to 2 MB	Model weights and inference engine must fit
Power	Microwatts to milliwatts	Inference must complete quickly — then sleep
Clock speed	16 MHz to 480 MHz	No GPU — pure integer MAC operations
OS	None — bare metal or RTOS	No memory management — static allocation only

What these constraints demand:

- INT8 quantization — mandatory
- Model architecture must be lightweight by design
- Inference engine in bare metal C — no dynamic allocation

TinyML is not a simplified version of AI. It is a discipline — applying the full rigour of embedded systems engineering to ML deployment.

TinyML — What It Can and Cannot Do

What TinyML can do today:

Application	Input	Hardware example
Keyword spotting	Microphone audio	STM32 Cortex-M4
Gesture recognition	IMU — accelerometer + gyro	Arduino Nano 33 BLE
Anomaly detection	Sensor feature vector	STM32 Cortex-M4
Machine health monitoring	Vibration + temperature features	STM32 Cortex-M4 / M7
Predictive maintenance	Current + vibration features	STM32 Cortex-M4

What TinyML cannot do:

Limitation	Reason
Large image recognition	Insufficient RAM for activations
Language models	Orders of magnitude too large
On-device learning	No compute or memory headroom
Video inference	Bandwidth and memory far exceed limits

The boundary of what TinyML can do moves outward every year — driven by better quantization, more efficient architectures, and MCUs with dedicated NPUs.

TinyML — Tools and Ecosystem

Tool	Created by	What it does
STM32Cube.AI	STMicroelectronics	Converts trained model to optimised C code for STM32 — generates weight arrays and inference engine automatically
CMSIS-NN	ARM	Low level C library of optimised neural network operators for Cortex-M — foundation most MCU inference engines build on
TensorFlow Lite Micro	Google	Lightweight inference engine — portable across any MCU — no OS dependency
Edge Impulse	Edge Impulse Inc	End to end TinyML platform — data collection, training, quantization, deployment — browser based
NanoEdge AI	STMicroelectronics	Automated anomaly detection library for STM32 — minimal ML knowledge needed
ESP-DL	Espressif	Inference engine optimised for ESP32-S3 vector instructions

STM32Cube.AI and Edge Impulse together cover the full workflow from training to deployment — without requiring deep knowledge of quantization internals.

Your Edge AI Journey: An Inside Look at the Edge Impulse Platform and Process STM32Cube.AI - STMicroelectronics - STM32 AI

Edge AI — Precise Definition

Edge AI means running AI inference on a device local to where data is generated — without sending data to a remote server for processing.

Three properties that define Edge AI:

Property	What it means
Local inference	The model runs on the device — not in the cloud
Data stays at the source	Raw data never leaves the device — only decisions do
Autonomous operation	Device makes decisions without network dependency

Edge AI vs Cloud AI:

	Edge AI	Cloud AI
Where inference runs	Local device	Remote data centre
Latency	Microseconds to milliseconds	100ms to seconds
Privacy	Data stays on device	Data leaves device
Connectivity needed	Optional or none	Always required
Model update	OTA, JTAG, DFU, or supported mechanism	Instant — server side

Edge AI is not a specific hardware class — it is a deployment philosophy.

Edge AI — What the Term Really Means in Practice

Edge AI is one of the most loosely used terms in the industry. Precise engineers must distinguish between what is marketed and what is actually happening.

What is said	What it actually means
"Running AI on the edge"	Almost always — frozen model inference only
"Edge AI chip"	A chip optimised for inference — not training
"Machine learning on MCU"	Inference using a pre-trained model — no learning
"AI enabled device"	A device running a frozen quantized model
"On-device learning"	Genuine weight updates on device — rare, needs powerful hardware

The precise statement an engineer should use:

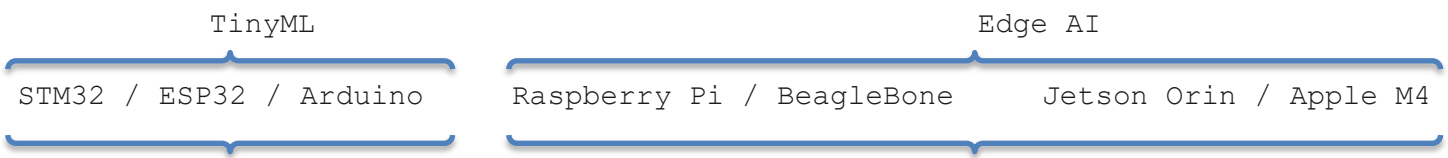
"Running inference using a pre-trained quantized model on an embedded processor, with model updates delivered via OTA, JTAG, DFU, or any supported update mechanism."

That one sentence leaves zero ambiguity about what the system is actually doing.

TinyML and Edge AI — Where Each Sits on the Spectrum

TinyML is a subset of Edge AI — constrained to microcontrollers specifically. Every TinyML deployment is Edge AI. Not every Edge AI deployment is TinyML.

	TinyML	Edge AI
Hardware	Microcontrollers — MCU only	MCU, SBC, edge module, edge server
RAM	KBs	KBs to GBs
OS	Bare metal or RTOS	Bare metal, RTOS, or full Linux
Model size	KB range — heavily quantized	KB to MB range
Power budget	Microwatts to milliwatts	Milliwatts to watts
On-device learning	Not feasible	Possible on high end edge hardware



NPU — Why a Dedicated AI Accelerator Exists

A general purpose CPU can run inference — but inefficiently. The core operation of every neural network inference is one thing repeated billions of times:

```
output = (input × weight) + bias          ← Multiply Accumulate — MAC
```

Repeated billions of times per inference. This is all a neural network does at its mathematical core.

Why a CPU is inefficient for inference:

CPU characteristic	Problem for inference
Designed for general purpose sequential tasks	MAC operations are massively parallel — not sequential
Fetches instructions and data separately	Inference is data intensive — memory bandwidth is the bottleneck
Integer and float units shared with all tasks	Inference competes with other system tasks for compute

An NPU — Neural Processing Unit — is a dedicated hardware block containing thousands of MAC units operating in parallel — designed exclusively for the matrix multiplication operations that neural network inference requires.

NPU — Key Players on the Embedded Spectrum

NPU	Found in	Performance	Designed for
ARM Ethos-U55	STM32N6, Cortex-M55 MCUs	256 TOPS/W	TinyML — ultra low power inference
ARM Ethos-U65	Higher end Cortex-M55 SoCs	512 TOPS/W	TinyML — higher throughput
ARM Ethos-N78	Mobile SoCs — Cortex-A	1-10 TOPS	Mobile edge AI inference
Apple Neural Engine	iPhone, iPad, Mac M-series	38 TOPS	On device AI — vision and language
NVIDIA GPU — Jetson	Jetson Orin, Jetson AGX	275 TOPS	Edge AI — vision, LLM inference
Google Edge TPU	Coral Dev Board	4 TOPS	TFLite model inference at edge

Term	Meaning
TOPS	Tera Operations Per Second — 10^{12} MAC operations per second
TOPS/W	TOPS per Watt — efficiency metric — critical for battery devices

For battery powered embedded systems — TOPS/W matters more than raw TOPS.

ARM Ethos NPU — Inside the STM32 Ecosystem

The STM32N6 — launched in 2024 — is STMicroelectronics' first MCU with an integrated ARM Ethos-U55 NPU. Hardware accelerated inference now available in the STM32 family.

Parameter	STM32N6 Specification
CPU	ARM Cortex-M55 — 800 MHz
NPU	ARM Ethos-U55 — 256 TOPS/W
RAM	4.2 MB on-chip SRAM
Flash	External — up to 256 MB
Camera interface	DCMI — direct camera input

CPU only vs CPU + Ethos NPU — inference comparison:

Model	Cortex-M55 CPU only	Cortex-M55 + Ethos U55
MobileNet V2 image classification	~500ms	~5ms
Keyword spotting	~10ms	~1ms
Anomaly detection — small autoencoder	~2ms	~0.2ms

Vision AI that was previously only feasible on Jetson class hardware now runs on a microcontroller.

OTA Model Update — The Production Workflow

In a deployed embedded AI system — the model is never static forever. As operating conditions change and more data is collected — the model can be retrained and redeployed.

Trigger	Example
Performance degradation	Model accuracy drops as new machine variants are added to fleet
New fault type discovered	New failure mode identified in field — model retrained to include it
More labeled data available	Phase 1 anomaly detector ready to become Phase 2 fault classifier

	OTA Model Update	On-Device Learning
Where retraining happens	Workstation or cloud	On the device itself
What arrives on device	New firmware with updated weight array	Updated weights computed locally
Model frozen after update?	Yes — until next OTA	No — weights change continuously
Feasible on STM32 class MCU?	Yes	No

OTA model update is not on-device learning. The device never trains. It receives the result of training done elsewhere.

OTA Model Update — End to End Workflow

```
CLOUD / WORKSTATION
  ├── Collect new field data
  ├── Retrain model on updated dataset
  ├── Validate — compare against previous version
  ├── Quantize — INT8
  ├── Generate updated C weight array
  └── Package into new firmware binary

          ↓  Deliver via OTA, JTAG, DFU, or supported mechanism

EDGE DEVICE — BOOTLOADER
  ├── Validates firmware integrity — CRC or cryptographic signature
  └── Writes new firmware to flash — reboots

EDGE DEVICE — RUNNING
  ├── Inference engine loads new weights from flash
  └── Sensor data → inference → decision continues
```

The weight array is just data in flash — like any const array in embedded C. Updating it is no different from updating any other firmware component.

SECTION 05

Deployment Realities

Frozen model, Continuous learning, Decision framework

Frozen Model Applications — The Majority of Embedded AI

Most embedded AI systems in production today use a frozen model — inference only. Intelligence is delivered through periodic model redeployment — not continuous learning.

Application	Input	Decision	Hardware
Machine health monitoring	Vibration + temperature features	Normal / Warning / Fault	STM32 Cortex-M4
Predictive maintenance	Motor current signature features	Remaining useful life	STM32 Cortex-M4 / M7
Keyword spotting	Microphone audio	Wake word detected / not	STM32 Cortex-M4
Gesture recognition	IMU — accelerometer + gyro	Gesture class	ESP32-S3
Battery state estimation	Cell voltage, current, temperature	SoC / SoH	STM32 Cortex-M4
Anomaly detection	Any sensor feature stream	Normal / Anomaly	TinyML class MCU
Presence detection	Low resolution camera	Person present / absent	Cortex-M55 + Ethos

Every one of these applications runs a frozen quantized model on an MCU — inference only — model improvements delivered via OTA, JTAG, DFU, or any supported update mechanism.

Continuous Learning — When the Model Must Keep Improving

A small but important class of applications requires the model to keep learning after deployment. The world changes around the device — and the model must adapt to remain accurate.

Condition	Why frozen model is insufficient
User specific adaptation	Every person's voice, behaviour, usage pattern is unique — generic frozen model is less accurate
Environment changes continuously	Acoustic environment, lighting, temperature — frozen model drifts in new conditions
New patterns emerge over time	New fault types, new language — frozen model has no knowledge of them
Privacy prevents cloud data transfer	Data cannot be sent for retraining — learning must happen locally

	Frozen Model	Continuous Learning
Weights after deployment	Fixed forever	Update on device continuously
Intelligence delivery	Via OTA / JTAG / DFU	Grows from local experience
Same input → same decision?	Always	Improves over time

Continuous learning is not a better version of frozen model deployment — it is a fundamentally different architecture with fundamentally different hardware requirements.

Continuous Learning — Applications and Hardware Reality

Application	Why learning must continue	Minimum hardware
Voice recognition — personalised	Must adapt to one person's voice, accent, environment	Smartphone DSP class
Predictive text keyboard	Learns individual writing style over time	Smartphone CPU
Adaptive robotics	Adapts to mechanical wear and environment	Jetson class or above
Federated learning node	Privacy — data cannot leave device	Powerful edge server
Autonomous vehicle perception	Must handle new road conditions continuously	Custom AI SoC

Hardware reality:

Capability	TinyML MCU	Smartphone	Jetson Orin
Frozen model inference	Yes	Yes	Yes
On-device fine tuning	No	Limited	Yes
Full on-device training	No	No	Limited

On-device learning on an STM32 class device is not feasible today.

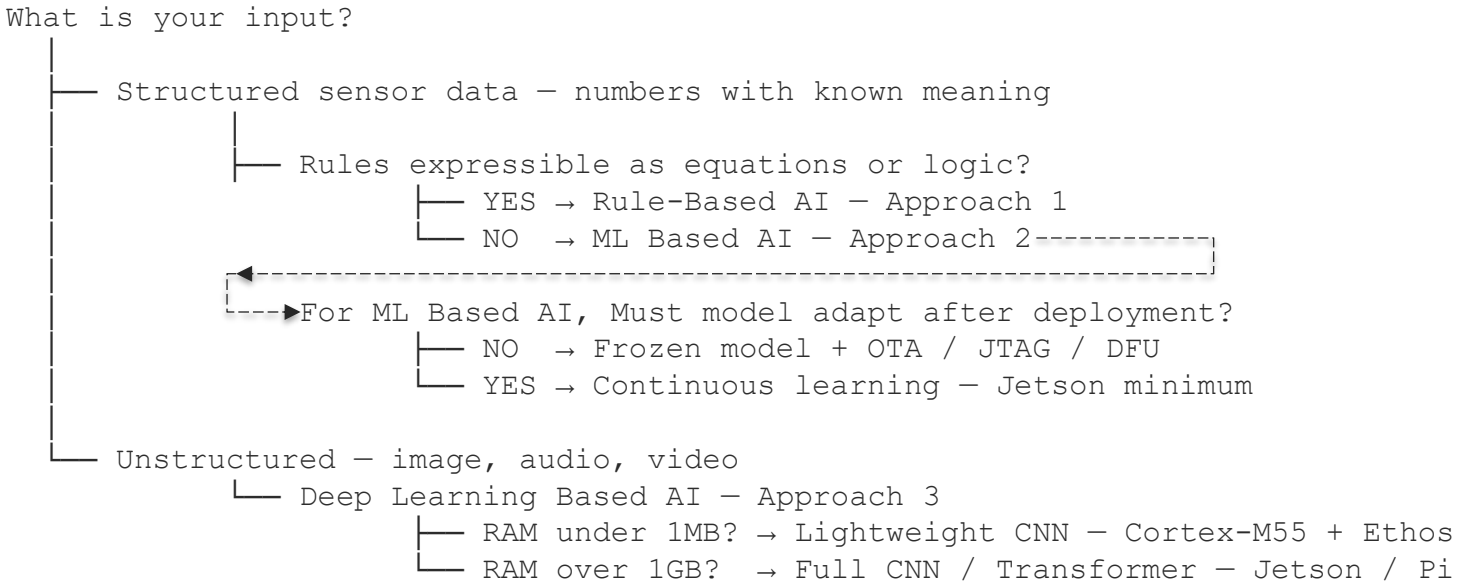
Choosing the Right Deployment Approach

Every embedded AI project begins with the same set of engineering questions. The answers determine the approach — not the other way around.

Engineering Question	If Yes	If No
Must decision happen locally?	Edge AI — TinyML or edge module	Cloud inference acceptable
Is RAM under 1 MB?	TinyML — INT8 quantization mandatory	Edge SBC or module
Is input unstructured — image, audio?	Deep Learning — Approach 3	ML or Rule-Based AI
Input structured, relationships complex?	ML Based AI — Approach 2	Rule-Based AI — Approach 1
Can rules be written explicitly?	Rule-Based AI — Approach 1	ML or Deep Learning
Must model adapt after deployment?	Continuous learning — Jetson minimum	Frozen model + OTA update
Is functional safety certification needed?	Rule-Based or frozen model	Continuous learning — challenging

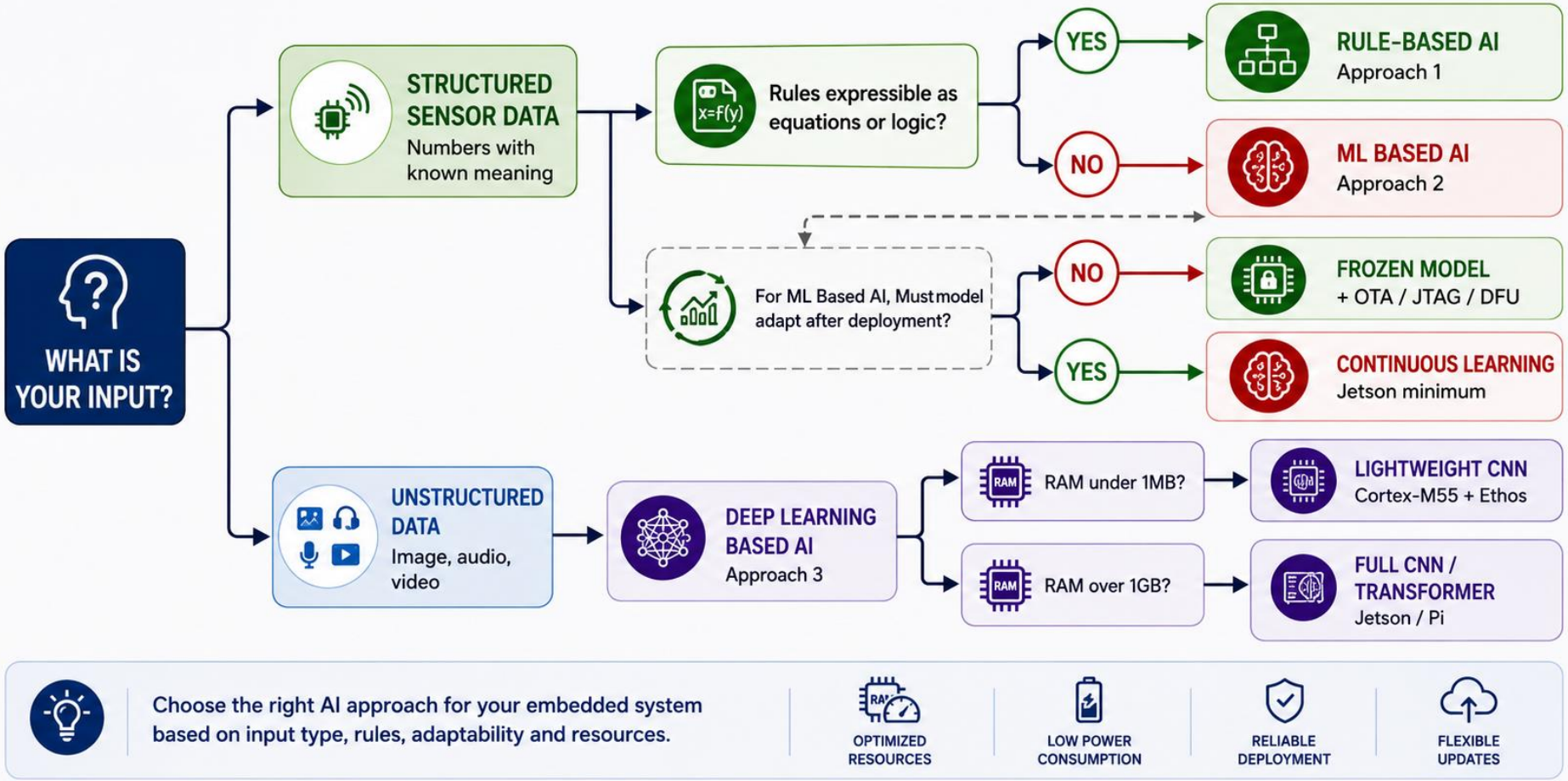
Start with requirements. Let requirements choose the approach. Let the approach choose the hardware. Let the hardware choose the framework.

The Deployment Decision — One View



Every embedded AI decision traces a path through this map. The path is determined by the problem — not by preference, trend, or tool availability.

EMBEDDED AI IMPLEMENTATION DECISION MAKING



SECTION 06

Terminology Discipline

Terminology map, Precision in context, The engineer's standard

The AI Terminology Map — Every Key Term Precisely Placed

Term	Precise definition	Common misuse
Artificial Intelligence	Any system performing tasks requiring human-like intelligence — rule-based or learning-based	Used only for learning-based systems
Machine Learning	Subset of AI where systems automatically learn patterns from data using one of many algorithms	Used interchangeably with AI
Deep Learning	Subset of ML using large multi-layer neural networks for unstructured high dimensional input	Used interchangeably with ML
Algorithm	Used at three levels — generic method / specific learning technique / mathematical operator in engine	Used without specifying which level
Model	File of learned weights and biases — data, not executable code	Confused with the inference engine
Training	Process of adjusting weights on powerful hardware until accuracy is acceptable	Confused with inference
Inference	Applying frozen model to new input on edge device — no weights change	Called running AI without specifying frozen
Quantization	Compressing FP32 weights to INT8 — equivalent to Q-format fixed point on DSP	Confused with model compression generally
TinyML	ML inference on MCUs — kilobytes of RAM, INT8, bare metal or RTOS	Applied to any small AI system
Edge AI	AI inference on local device — data stays at source — no cloud dependency	Applied even when cloud processing involved
NPU	Dedicated hardware for parallel MAC operations — not general compute	Called AI chip loosely
OTA update	Firmware update delivering new model weights — not on-device learning	Model assumed static forever

Terminology — Context Determines the Meaning

The same word means different things at different levels of the conversation. Knowing which level is active is the mark of a precise engineer.

The word "Model" — three different meanings in one project:

Context	What "model" means
Business discussion	"Our AI model detects faults early" — means the entire AI system
ML engineering discussion	"The model has 4 million weights" — means the trained weight file
Embedded discussion	"The model fits in 512KB of flash" — means the quantized INT8 weight array

The word "Learning" — two completely different meanings:

Context	What "learning" means
Training phase	"The model is learning from data" — weights are actively being updated on powerful hardware
Deployment phase	"The device is learning" — almost always incorrect — device is doing inference only

Precise engineers qualify every ambiguous term with its context. Imprecise engineers assume the listener shares their mental model — and are surprised when they do not.

The Engineer's Standard — Three Principles

A word is not just a label. In engineering it carries a precise definition, a set of assumptions, a boundary of applicability, and an implied context. When that word drifts — errors propagate silently.

01 — Precision Matches Phase

Concept phase allows broader terms. Design phase demands precise terms. Implementation mandates exact terms. Adapt the depth — never reduce the precision.

02 — Terminology is a Contract

Two engineers using the same word with the same meaning share zero ambiguity. When a word drifts — errors propagate silently. The smarter the engineer — the more dangerous the drift.

03 — Push Back on Vagueness

Always question imprecise words — from a colleague, a document, a datasheet, or an AI system.

Discomfort with vague terms is not pedantry. It is professional discipline.

Closing Summary — The Complete Picture

Approach	When	Example
Rule-Based AI	Rules expressible as equations or logic	Kalman Filter, Fuzzy Logic
ML Based AI	Structured data — complex relationships	Machine health monitoring
Deep Learning Based AI	Unstructured input — images, audio	Thermal camera defect detection

One journey — always:

RAW DATA → ALGORITHM → TRAINING → MODEL → INFERENCE ENGINE → DECISION

One rule — never changes:

Training happens where compute is abundant. Inference happens where the decision is needed.

One standard — nonnegotiable:

Precise terminology is not pedantry. It is the foundation of flawless engineering.

The hardware is just the beginning. The data discipline is everything. The terminology is the contract.