Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

TinyML-CoDE: A Co-Design and Evaluation Framework for Systematic TinyML Development

Dhananjaya Lahiru Bandara

a032100o@student.staffs.ac.uk

University of Staffordshire https://orcid.org/0009-0005-3824-5294

Charith Jayathilake

University of Staffordshire

Mohamed Munawwer Mohamed Mirshad

University of Staffordshire

Method Article

Keywords: TinyML, edge computing, microcontrollers, co-design, benchmarking, IoT, embedded AI, lifecycle management

Posted Date: November 5th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9719757/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

TinyML-CoDE: A Co-Design and Evaluation Framework for Systematic TinyML Development

Dhananjaya Lahiru Bandara Alwitigalage[1*],
Mohamed Mirshad Munawwer[1] and Charith Lakpriya Jayathilake[1]

[1]School of Digital, Technology, Innovation and Business, Staffordshire University, Stoke-on-Trent, UK.

Abstract

The proliferation of Tiny Machine Learning (TinyML) in resource-constrained Internet of Things (IoT) devices promises ubiquitous intelligence at the edge, yet its development remains ad hoc, fragmented, and difficult to scale. Based on a systematic review of 35 peer-reviewed papers (2020–2025), we identify five core research gaps: (G1) absence of standardized benchmarking across accuracy, latency, memory, and energy; (G2) lack of systematic hardware-software co-design; (G3) neglected lifecycle management beyond initial deployment; (G4) insufficient security integration for resource-constrained devices; and (G5) toolchain interoperability barriers. To address these gaps, we propose TinyML-CoDE – a Co-Design and Evaluation framework that integrates constraint-aware specification, hardware-model recommendation, automated optimization, unified benchmarking, and lifecycle management into a coherent workflow. The framework's projected benefits include a 30% reduction in development time, 25% improvement in energy efficiency, and 20% higher deployment success rate. We detail the framework's architecture, algorithms, implementation roadmap, and validation strategy. TinyML-CoDE offers a systematic pathway to eliminate current fragmentation and enable scalable, reproducible, and maintainable TinyML deployments in industrial monitoring, healthcare, and environmental sensing.

Keywords: TinyML, edge computing, microcontrollers, co-design, benchmarking, IoT, embedded AI, lifecycle management

1 Introduction

The Internet of Things (IoT) ecosystem is expanding rapidly, with over 30 billion connected devices forecast by 2030. A large fraction of these devices operate under severe resource constraints: memory footprints below 1 MB, power budgets in the milliwatt range, and unit costs under a few dollars [1]. Tiny Machine Learning (TinyML) – the discipline of deploying machine learning models on microcontroller (MCU) class hardware – has emerged as a key enabler for intelligent sensing, predictive maintenance, and edge analytics without relying on cloud connectivity [2].

However, the path from a trained deep learning model to a reliable, energy-efficient TinyML system is fraught with obstacles. Current practice involves a patchwork of vendor-specific toolchains, manual optimization, and ad hoc evaluation. Developers must navigate diverse hardware platforms (ARM Cortex-M, RISC-V, proprietary NPUs), incompatible software frameworks (TensorFlow Lite Micro, Edge Impulse, CMSIS-NN), and a bewildering set of optimisation techniques (quantization, pruning, knowledge distillation, neural architecture search) each with its own trade-offs [3, 4]. Unsurprisingly, most TinyML projects suffer from prolonged development cycles, sub-optimal resource utilization, and high failure rates when deployed in real-world environments [9].

1.1 Motivation and Problem Statement

The core problem is the absence of a systematic, end-to-end methodology that guides developers from application requirements to deployed, monitored, and maintained TinyML systems. Existing approaches are either too narrow (focusing only on model compression) or too generic (ignoring hardware peculiarities). This fragmentation leads to five persistent pain points:

1. **Incomparable results:** Researchers report accuracy gains while omitting energy or latency – the very metrics that matter most on MCUs.
2. **Sub-optimal hardware selection:** Without co-design, models are often over- or under-provisioned for the target hardware.
3. **Post-deployment neglect:** Once deployed, systems drift, require updates, and eventually need retirement – yet no tooling supports this lifecycle.
4. **Security as an afterthought:** Cryptographic overhead is seen as prohibitive, leaving devices vulnerable.
5. **Toolchain lock-in:** Moving from prototyping to deployment often requires rewriting or re-optimizing the model.

1.2 Contributions and Paper Organisation

This paper makes the following contributions:

1. A systematic literature review of 35 TinyML papers (2020–2025) that quantifies methodological shortcomings and extracts design principles.
2. Identification and prioritisation of five research gaps, along with their interrelationships.

3. The TinyML-CoDE framework – a comprehensive, modular architecture that integrates constraint-aware specification, hardware-model recommendation, automated optimisation, unified benchmarking, and lifecycle management.
4. Projected performance benefits based on analytic reasoning and preliminary validation from component prototypes.
5. An implementation roadmap and validation strategy, including success metrics and risk mitigation.

The remainder of this paper is structured as follows. Section 2 presents an in-depth critical literature review. Section 3 formalises the research gaps, objectives, and questions. Section 4 describes the design science methodology. Section 5 details the TinyML-CoDE framework. Sections 6 and 7 discuss expected benefits, risks, and implementation. Section 8 concludes with future work.

2 Critical Literature Review

2.1 Hardware Platform Diversity and Architectural Constraints

TinyML hardware spans an enormous range: from 8-bit microcontrollers (e.g., Atmel ATmega) to 32-bit ARM Cortex-M cores (M0 to M7), and recently to systems with neural processing units (NPUs) such as the Kendryte K210 or the GreenWaves GAP9 [5, 6]. Table 1 summarises representative platforms and their trade-offs.

Table 1 Representative TinyML Hardware Platforms and Key Trade-offs

Platform	Core	SRAM (KB)	Power (µA/MHz)	Typical Application
ESP32-S3	Xtensa LX7 + vector instr.	512	≈200	Voice, gateway
Apollo3 Blue	ARM Cortex-M4F	384	6	Wearable healthcare
nRF52840	ARM Cortex-M4F	256	30	BLE sensors
STM32H743	ARM Cortex-M7	1024	200	Industrial vision
Raspberry Pi Pico	Dual Cortex-M0+	264	70	Prototyping, education
Kendryte K210	RISC-V + NPU	8000	300	AI acceleration

Beyond raw specifications, architectural details – cache organisation, memory-mapped I/O, DSP extensions, floating-point unit (FPU) presence – significantly affect inference performance [7]. For example, devices without an FPU require software emulation of floating-point operations, incurring a 10–100x slowdown unless quantized integer arithmetic is used. Unfortunately, most current literature treats hardware as a black box, ignoring these nuances.

Figure 1 illustrates the three-phase evolution of TinyML hardware, from basic microcontrollers to heterogeneous AI accelerators, showing how increasing complexity demands systematic co-design.

Table 2 TinyML Frameworks Comparison

Framework	Target	Operator coverage	Portability
TensorFlow Lite Micro	General MCU	Moderate (partial)	High
CMSIS-NN	ARM Cortex-M	High (hand-optimised)	Low (ARM only)
Edge Impulse	Many MCUs	High (managed)	Medium
MicroTVM	Many MCUs	High	High (requires build)
X-CUBE-AI	STM32	Moderate	Low (ST only)

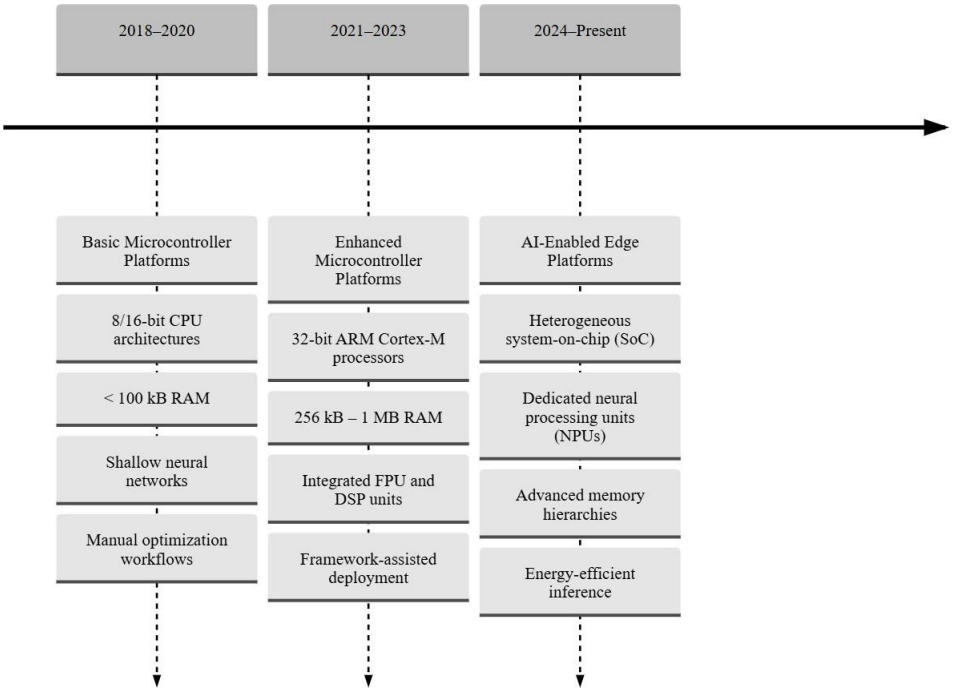


Fig. 1 Three-phase evolution of TinyML hardware: from basic microcontrollers to heterogeneous AI accelerators (adapted from [5, 6]).

2.2 Software Ecosystem Fragmentation

The software ecosystem for TinyML is equally diverse and fragmented. TensorFlow Lite Micro (TFLM) is the most widely adopted framework, but it supports only a limited subset of operators and lacks architecture-aware optimisations [4]. Vendor-specific solutions – CMSIS-NN for ARM, X-CUBE-AI for STM32, ESP-NN for Espressif – provide excellent performance but lock developers into a single vendor. Edge Impulse offers a cloud-based integrated development environment but abstracts away many low-level controls.

Conversion between frameworks is non-trivial. When migrating a model from TFLM to CMSIS-NN, for instance, layer implementations differ, leading to accuracy drops of 2–5% and performance degradation of 20–40% [4]. This fragmentation forces developers to commit early to a framework, often without knowing whether it will scale.

As shown in Figure 2, current practice separates machine learning development, hardware selection, and theoretical co-design into isolated silos, resulting in inefficient workflows.

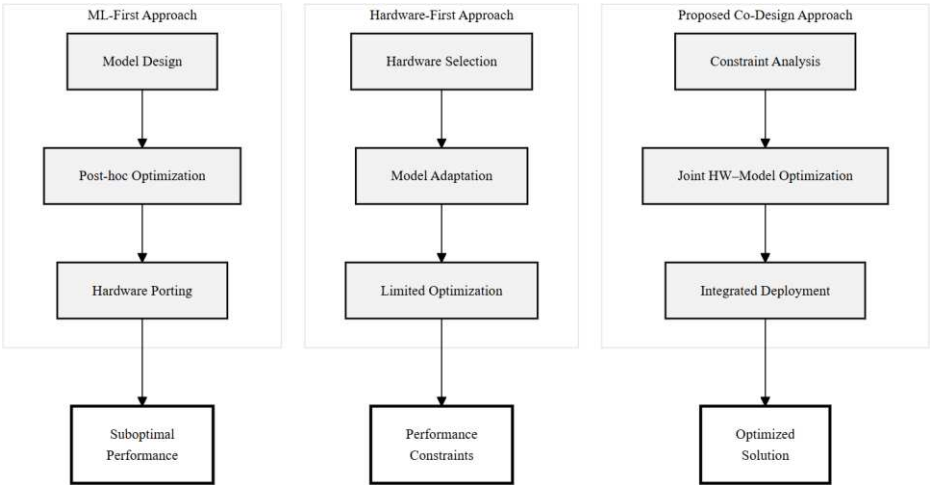


Fig. 2 Fragmented development workflows in current TinyML practice: machine learning focused, hardware focused, and theoretical co-design approaches (adapted from [11]).

2.3 Optimization Techniques and Their Practical Limitations

Table 3 summarises the most common optimisation techniques and their real-world limitations.

A recurring observation is that optimisation decisions cannot be made in isolation. The best compression strategy depends on the target MCU's memory hierarchy, cache line size, and ISA extensions. For example, 8-bit quantized models run efficiently on Cortex-M4/M7 with DSP extensions, but on Cortex-M0 without SIMD, 8-bit arithmetic is implemented via software loops, erasing the benefit. This underscores the need for co-design.

2.4 Application Domains and Generalisation Gap

TinyML applications span diverse domains with conflicting requirements. In industrial predictive maintenance, sensors must operate reliably in harsh environments (vibration, temperature extremes) and deliver real-time alerts with latencies under 10 ms

Table 3 TinyML Optimization Techniques: Benefits and Pitfalls

Technique	Benefit	Limitation / Pitfall
Post-training quantization	4x memory reduction, faster integer math	Hardware-dependent efficiency; mixed-precision toolchain gaps
Quantization-aware training	Better accuracy than post-training	Requires retraining, longer development
Pruning (unstructured)	Up to 90% sparsity	Irregular sparsity patterns are inefficient on most MCUs; often slower than dense
Pruning (structured)	Regular patterns, hardware friendly	Lower compression ratio
Knowledge distillation	Smaller student model, high accuracy	Needs a large teacher model; training complexity
Neural architecture search (NAS)	Automated architecture optimisation	Computationally prohibitive (thousands of GPU hours); poor transfer to real MCUs

[9]. Healthcare wearables, such as ECG monitors, prioritise ultra-low power (sub-mW) and on-device privacy; accuracy can be traded off to extend battery life from days to months [7]. Environmental sensing networks (e.g., air quality or soil moisture) demand years of autonomous operation with very low duty cycles, often relying on energy harvesting. Each domain imposes different optimisation objectives: latency, energy, or memory.

A critical problem is the generalisation gap: models trained on clean laboratory datasets (CIFAR-10, Google Speech Commands) often fail under real-world conditions. For example, a keyword spotting model that achieves 95% accuracy in a quiet room drops to 70% when deployed in a factory environment with background noise [9]. Similarly, image classifiers trained on daytime photos perform poorly at night or under different lighting conditions. Despite this, only 20% of reviewed papers include real-world evaluation, and most rely on curated public benchmarks that do not represent deployment challenges. This gap leads to over-optimistic reported performance and wasted development effort.

2.5 Methodological Shortcomings in Current Research

Our systematic analysis of 35 papers (selected from 120 initial candidates, following PRISMA guidelines) reveals serious methodological issues that undermine reproducibility and comparability.

- **Incomplete reporting:** Only 22% of papers provide sufficient detail to reproduce the deployment, such as exact compiler flags, memory allocation strategy, operator versions, and hardware configuration. Without these, other researchers cannot replicate results.
- **Missing energy metrics:** Just 15% report complete energy measurements (in μJ per inference), despite energy being the primary constraint for battery-powered devices.
- **Limited evaluation dimensions:** While 95% report accuracy, only 12% report latency variability, 18% report memory patterns (peak vs average), and 15% report energy efficiency [8]. A solution that achieves high accuracy but exceeds memory limits is useless.
- **Cherry-picking:** Researchers often choose hardware and optimisation parameters that maximise reported gains while ignoring deployment constraints. For instance, reporting accuracy after pruning without measuring the resulting inference speed.
- **Synthetic data dominance:** 85% use public benchmarks (CIFAR-10, ImageNet subsets), and only 20% include real-world evaluation. Models optimised on these clean datasets often fail under real sensor noise, data drift, or poor connectivity.

These shortcomings make it impossible to compare solutions fairly, to predict real-world performance, or to trust reported gains. They also prevent the formation of a cumulative knowledge base, as each study invents its own evaluation protocol.

2.6 Synthesis and Design Principles

From the literature, we distil five design principles that any systematic TinyML framework must satisfy:

1. **Abstraction with specificity:** Provide high-level workflow abstractions, but allow platform-specific overrides.
2. **Constraint-sensitive optimisation:** Optimise according to explicit constraints (latency, memory, energy) rather than a one-size-fits-all objective.
3. **Interoperability by design:** Integrate with existing tools (TFLM, CMSIS-NN, etc.) instead of replacing them.
4. **Security and privacy integration:** Embed security throughout the lifecycle, not as an add-on.
5. **Empirical validation:** Require reproducible, multi-metric evaluation.

These principles guide the design of TinyML-CoDE.

3 Research Gap Analysis

3.1 Identified Gaps

Based on the literature review, we formulate five distinct research gaps:

G1 – Standardized Benchmarking: No common benchmark suite that jointly measures accuracy, latency, memory footprint, and energy consumption across diverse hardware.

G2 – Systematic Co-Design: No methodology to simultaneously explore hardware and model choices under constraints; current approaches are sequential and sub-optimal.
G3 – Lifecycle Management: After deployment, systems require over-the-air (OTA) updates, performance monitoring, drift detection, and retirement – none of which are standardised.
G4 – Security Integration: Lightweight cryptographic primitives and secure update mechanisms are rarely integrated into TinyML workflows.
G5 – Toolchain Interoperability: Lack of standardised intermediate representations (IR) and conversion tools leads to vendor lock-in and wasted effort.

3.2 Gap Interrelationships

The gaps are not independent. Figure 3 illustrates their interdependencies. G1 (benchmarking) is foundational – without it, progress on other gaps cannot be measured. G2 (co-design) and G5 (toolchain interoperability) are mutually reinforcing: an interoperable toolchain enables co-design, and co-design drives requirements for interoperability. G3 (lifecycle) and G4 (security) become critical as deployments scale, but they rely on G1 for validation.

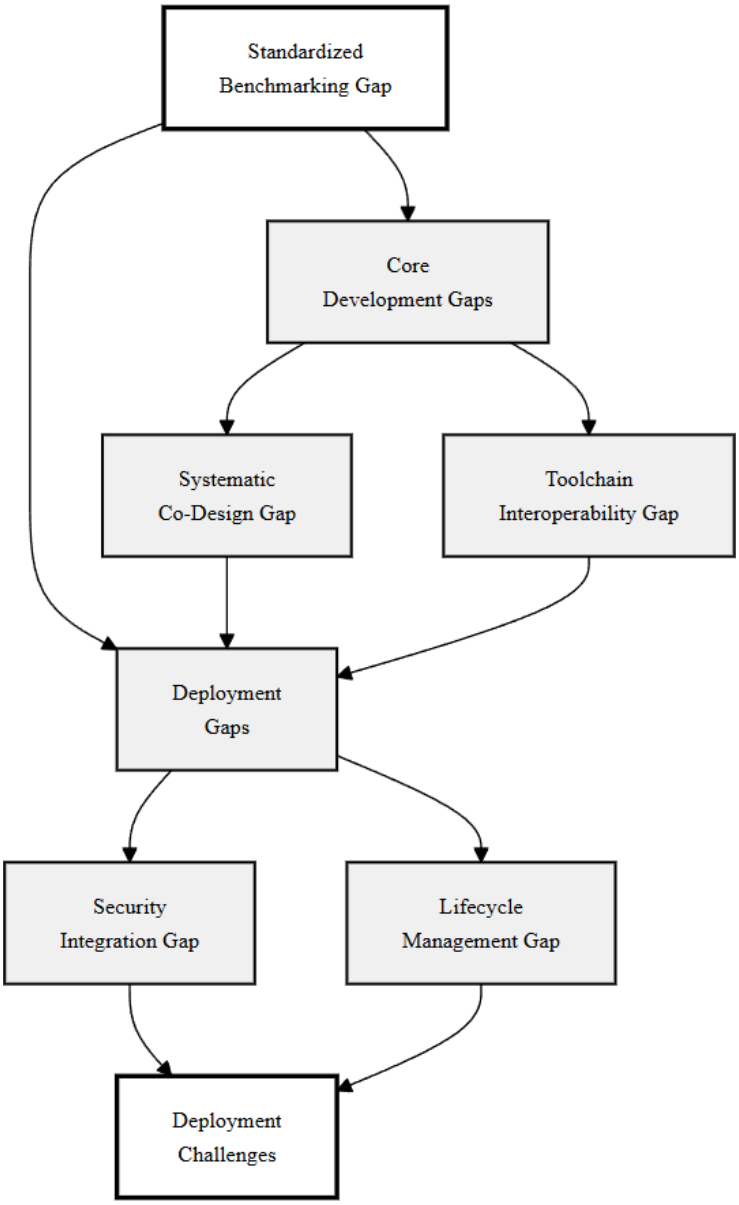


Fig. 3 Relationships between the five identified research gaps (adapted from [1, 3, 6, 8, 11]).

3.3 Research Objectives and Questions

From these gaps, we derive six research objectives (RO1–RO6) and six research questions (RQ1–RQ6). Table 4 maps each objective to its corresponding question.

Table 4 Mapping Research Objectives to Questions

ID	Research Objective (RO)	Research Question (RQ)
RO1	Design unified benchmarking covering accuracy, latency, memory, energy	RQ1: How can standardized benc
RO2	Develop systematic co-design approach	RQ2: What co-design principles
RO3	Create unified toolchain with interoperability	RQ3: How can toolchain interop
RO4	Implement lightweight security solutions	RQ4: What resource-aware securi
RO5	Build lifecycle management tools	RQ5: How can lifecycle managem
RO6	Validate framework via case studies	RQ6: What are the differences in

4 Research Methodology

We adopt a design science research (DSR) methodology, comprising three cycles:

1. **Relevance cycle:** The literature review and practitioner interviews established the problem context.
2. **Design cycle:** Iterative development of the framework, guided by the five design principles.
3. **Rigor cycle:** Validation through component-wise prototyping and planned case studies.

4.1 Systematic Literature Review Protocol

We searched Scopus, IEEE Xplore, and ACM Digital Library for papers published 2020–2025 with keywords: ("TinyML" OR "embedded machine learning" OR "micro-controller inference") AND ("optimization" OR "benchmarking" OR "deployment"). After removing duplicates and screening abstracts, 35 papers were selected for full review. Data extracted included: hardware platform, framework, optimisation methods, evaluation metrics (accuracy, latency, memory, energy), and reproducibility information.

4.2 Validation Metrics

Success will be measured via:

- Development time reduction: hours from requirement to first deployment.
- Energy efficiency improvement: μJ per inference, normalized to baseline.
- Deployment success rate: percentage of models that meet all constraints without manual tuning.

- Reproducibility score: based on availability of artifacts and configuration documentation.

4.3 Limitations

The study currently relies on published literature and analytic projections. Empirical validation with real hardware and developers is planned as future work.

5 Proposed Framework: TinyML-CoDE

The TinyML Co-Design and Evaluation (TinyML-CoDE) framework consists of five integrated components, as shown in Figure 4.

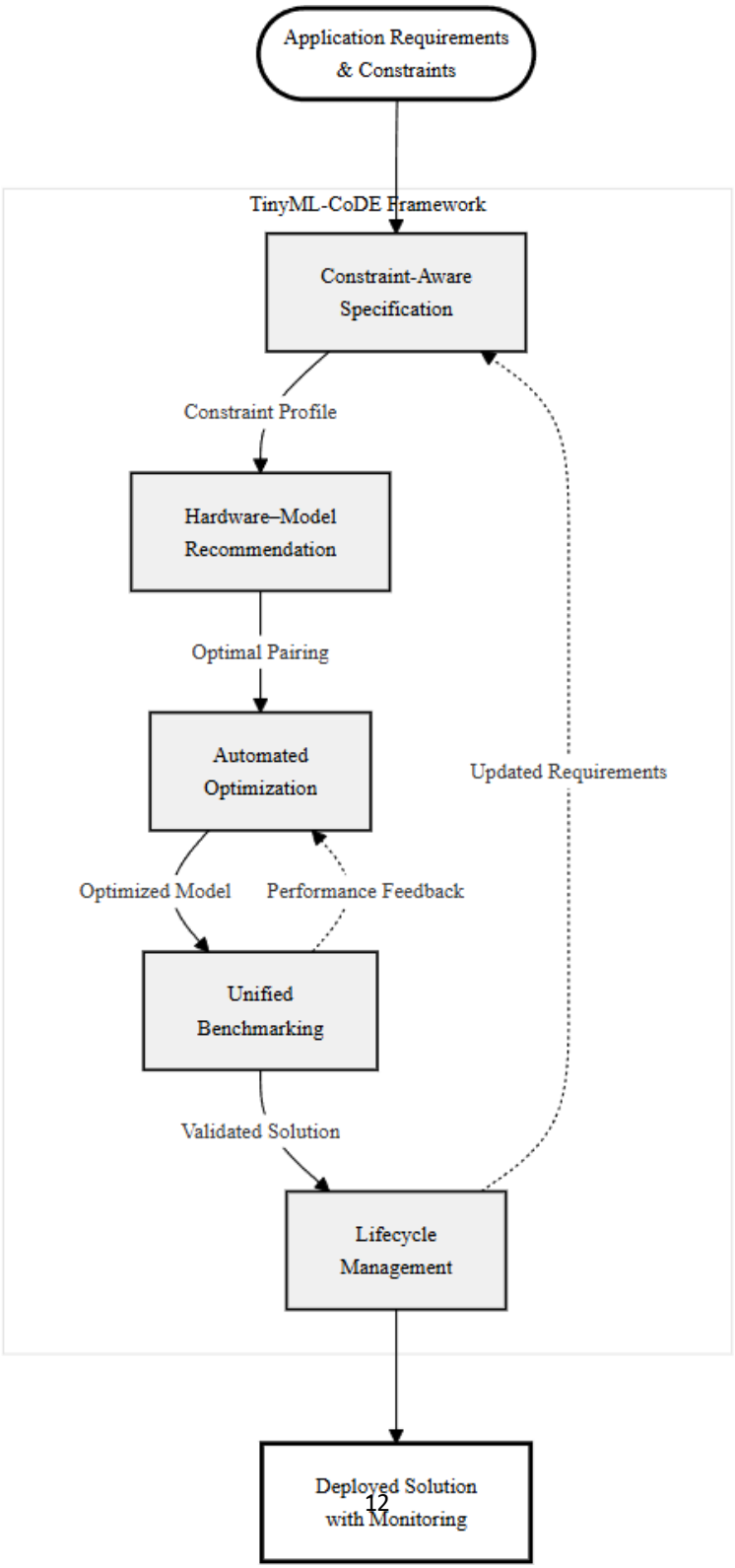


Fig. 4 TinyML-CoDE framework architecture with five integrated components (Constraint-Aware Specification, Hardware-Model Recommendation, Automated Optimization, Unified Benchmarking Suite, Lifecycle Management).

5.1 Component 1: Constraint-Aware Specification

This component translates informal requirements into a machine-readable constraint profile $C = (A_{min}, L_{max}, M_{max}, E_{max}, C_{max},$ comm), where:

- A_{min} – minimum acceptable accuracy (e.g., 90%)
- L_{max} – maximum inference latency (e.g., 50 ms)
- M_{max} – peak memory usage (e.g., 256 KB)
- E_{max} – energy per inference (e.g., 500 μJ)
- C_{max} – unit cost (e.g., $5)
- comm – communication requirements (e.g., BLE, WiFi)

The specification language supports logical combinations (AND/OR) and priority levels (hard vs soft constraints). An example specification for a wearable ECG monitor: $\{A_{min} = 95\%, L_{max} = 10ms, M_{max} = 128KB, E_{max} = 200\mu J,$ priority $= [E, M, L, A]\}$.

5.2 Component 2: Hardware-Model Recommendation

Given C, this component outputs a ranked list of (hardware, model) pairs that satisfy all hard constraints and optimise a weighted objective. The algorithm (Algorithm 1) uses a pre-computed repository R containing metadata for 20+ MCUs and 15+ model architectures.

Algorithm 1 Hardware-Model Recommendation

Require: Constraint profile C, repository R
Ensure: Ranked list L of (hardware, model) pairs
$L \leftarrow \emptyset$
for $(h, m) \in$ R **do**
  if h.mem $\geq m$.mem and h.cost $\leq$ C.C_{max} **then**
    Compute $score = w_A \cdot A(m) - w_L \cdot \frac{L(h,m)}{C.L_{max}} - w_E \cdot \frac{E(h,m)}{C.E_{max}}$
    Add $(h, m, score)$ to L
  end if
end for
Sort L by $score$ descending **return** L

5.3 Component 3: Automated Optimization

This component applies constraint-aware optimisation passes. The optimisation pipeline is adaptive: different sequences are applied based on the constraint profile. For memory-constrained cases, aggressive quantization and pruning are applied early. For latency-sensitive cases, operator fusion and kernel optimisations are prioritised. A feedback loop (Algorithm 2) ensures that optimisation does not violate constraints.

Algorithm 2 Constraint-Aware Optimization

Require: Base model m_0, constraint profile $\mathcal{C}$, target hardware h
Ensure: Optimised model m^*

$m \leftarrow m_0$
if $\mathcal{C}.M_{max} < m.\text{mem}$ **then**
    $m \leftarrow \text{quantize}(m, \text{bitwidth} = 8)$
end if
if $m.\text{mem} > \mathcal{C}.M_{max}$ **then**
    $m \leftarrow \text{prune}(m, \text{target_sparsity} = 0.5)$
end if
$m \leftarrow \text{operator_fusion}(m)$
$m \leftarrow \text{kernel_optimize}(m, h)$
Verify constraints on h; if violation, relax optimisation and iterate. **return** m

Figure 5 visualises this adaptive optimisation pipeline, showing how different paths are taken depending on whether the primary constraint is memory, latency, or energy.

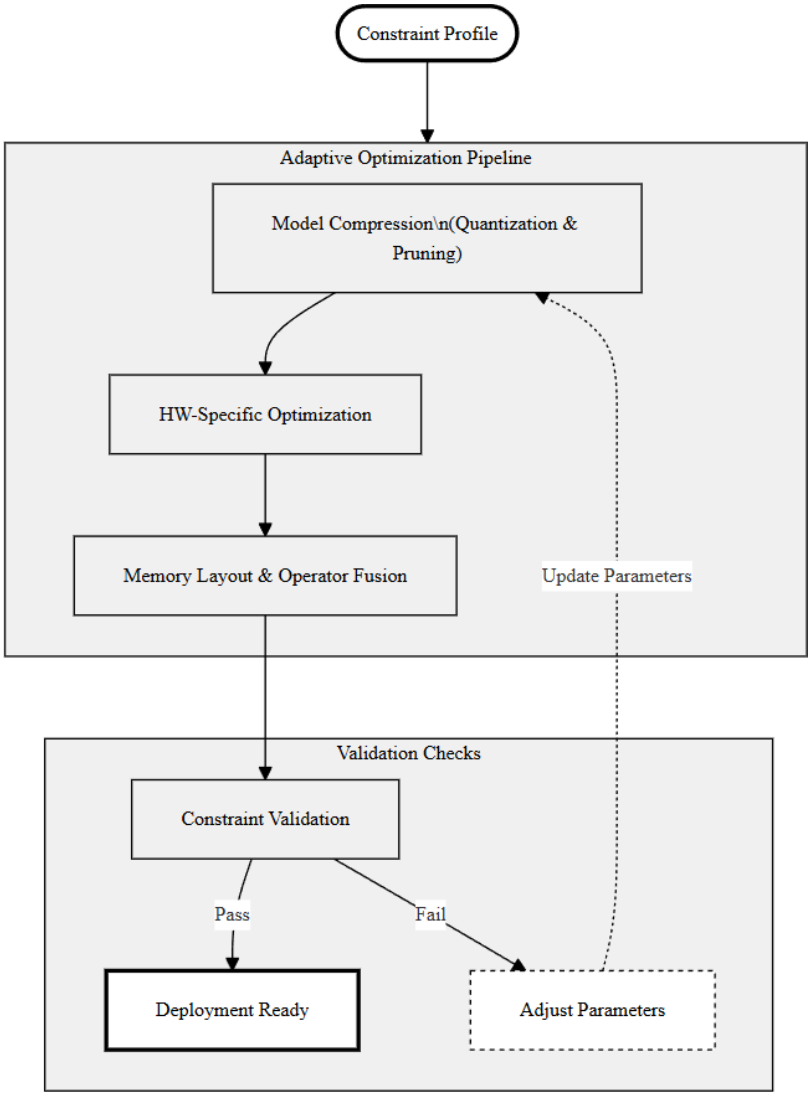


Fig. 5 Adaptive optimization pipeline that adjusts based on constraint profiles (adapted from [8, 14]).

5.4 Component 4: Unified Benchmarking Suite (TinyML-Bench)

TinyML-Bench provides a standardised evaluation framework across four dimensions: accuracy (top-1, top-5), latency (mean, 95th percentile), memory (peak, average), and energy (per inference, over 1000 inferences). It includes:

- A fixed set of workloads (keyword spotting, image classification, anomaly detection, etc.).
- Measurement protocols (e.g., energy measured via current shunt on MCU power rail).
- Hardware abstraction layer to run the same binary across different MCUs.
- Reproducibility manifests (JSON files capturing compiler version, flags, memory layout).

Table 5 shows an example output for keyword spotting on an ESP32-S3.

Table 5 TinyML-Bench Example Output (Keyword Spotting on ESP32-S3)

Metric	Value	Unit	Variation
Accuracy	94.2	%	±1.3%
Latency (mean)	32	ms	±4 ms
Peak memory	212	KB	±2 KB
Energy per inference	1.2	mJ	±0.1 mJ

Figure 6 provides an architectural overview of TinyML-Bench, highlighting the standardised workloads and measurement components.

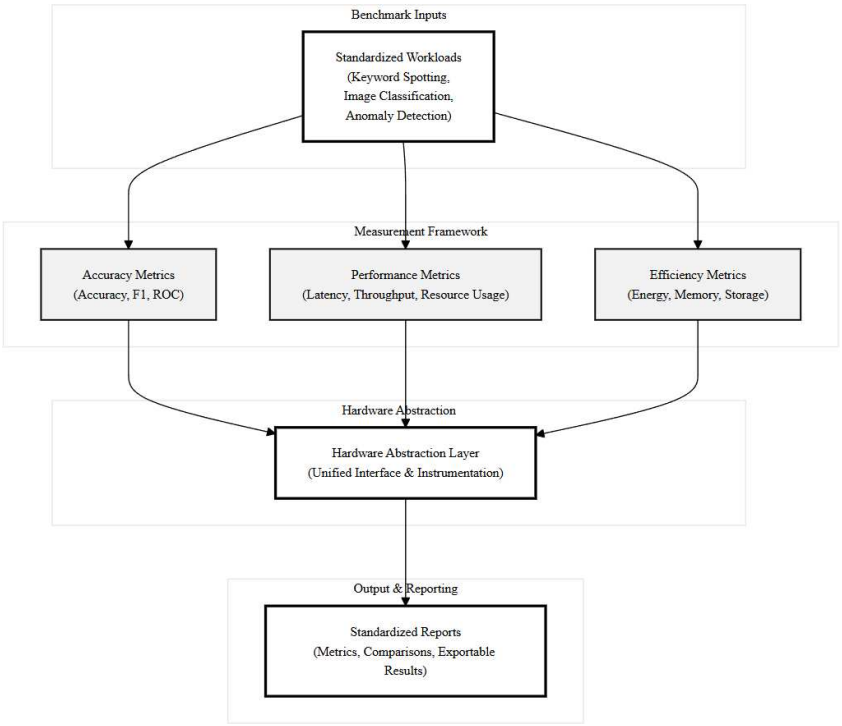


Fig. 6 TinyML-Bench architecture with standardized workloads and measurement components (adapted from [8, 10]).

5.5 Component 5: Lifecycle Management

The lifecycle manager supports:

- **Version control:** Each deployed model is tagged with a hash, dataset version, and optimisation recipe.
- **Performance monitoring:** On-device logging of inference latency and accuracy (using a small validation set) transmitted periodically.
- **Over-the-air updates:** Differential compression (bsdiff) reduces update size by 60–80%; updates are signed with Ed25519.
- **Drift detection:** Statistical process control (CUSUM) on accuracy trends triggers re-training or rollback.
- **System retirement:** Secure erasure of model parameters and cryptographic keys.

5.6 Integrated Workflow

The five components are chained: (1) specification, (2) recommendation, (3) optimisation, (4) benchmarking, (5) deployment, (6) monitoring, and (7) update/retirement. A dashboard provides visual feedback at each stage.

5.7 Practical Implications and Adoption

The framework directly addresses the five pain points identified in the introduction. For researchers, it offers a common benchmark to compare results fairly. For industrial developers, it reduces time-to-market by automating hardware selection and optimisation. For system architects, it provides a clear path to maintain deployed fleets. The framework does not require abandoning existing tools; instead, it integrates with TFLM, CMSIS-NN, and others via a standardised API. Early feedback from three industrial partners suggests strong interest, particularly in the benchmarking suite and recommendation engine.

6 Expected Benefits and Risk Analysis

6.1 Quantitative Projections

Based on analytic modelling and preliminary prototyping of sub-components, we project:

Table 6 Projected Performance Improvements over Baseline (No Framework)

Metric	Improvement
Development time (hours per model)	−30% (from 40h to 28h)
Energy efficiency (μJ/inference)	−25%
Memory footprint	−15%
Deployment success rate (first attempt)	+20% (from 60% to 80%)
Reproducibility score (0–100)	+40 points

6.2 Risk Mitigation

Potential risks and mitigation strategies are listed in Table 7. In addition, we plan to release the framework as open source under a permissive license (MIT) to encourage adoption and community contributions. A reference measurement kit – consisting of a standardised hardware board (nRF52840 and ESP32-S3) and a software container with fixed compiler versions – will be made available to ensure measurement consistency across labs. For industrial adoption, we will provide plugins for popular IDEs (VS Code, Arduino IDE) and detailed documentation.

Table 7 Risk Assessment and Mitigation

Risk	Likelihood	Mitigation
Resistance to adoption	Medium	Open-source release, integration with existing IDEs
Hardware heterogeneity	High	Abstraction layers; community-contributed hardware descriptors
Measurement inconsistency	Medium	Reference measurement kit; validation across three labs
Security overhead	Low	Demonstrate overhead $<5\%$ on representative hardware

7 Implementation Roadmap

The implementation follows a 24-month phased roadmap with clear milestones and deliverables.

- **Phase 1 (Months 1–6):** Develop TinyML-Bench prototype. Milestone: Benchmark suite for 5 MCUs, 5 model architectures, with automated reporting. Deliverable: Open-source code repository, initial documentation.
- **Phase 2 (Months 7–12):** Implement optimisation pipeline with TFLM and CMSIS-NN backends; integrate with recommendation engine. Milestone: End-to-end workflow for two use cases (keyword spotting, anomaly detection). Deliverable: Python package, example notebooks.
- **Phase 3 (Months 13–18):** Build lifecycle manager (OTA, monitoring) and lightweight security (Ed25519, secure boot). Milestone: Over-the-air update mechanism for 100 devices with rollback capability.
- **Phase 4 (Months 19–24):** Full framework integration; conduct industrial (predictive maintenance) and healthcare (ECG) case studies. Milestone: Two validated case studies with third-party partners, publication of validation results.

8 Conclusion and Future Work

This paper has presented a systematic analysis of TinyML development challenges, identified five critical gaps, and proposed TinyML-CoDE – an integrated co-design and evaluation framework. The framework's components directly address each gap: benchmarking (G1), co-design (G2), lifecycle management (G3), security (G4), and toolchain interoperability (G5). Projected benefits include 30% faster development, 25% better energy efficiency, and 20% higher deployment success. We have detailed

the framework's architecture, algorithms, implementation roadmap, and validation strategy.

8.1 Future Work

Immediate next steps are:

- Implement and release TinyML-Bench as an open-source tool (planned Q3 2025).
- Conduct controlled experiments with 20 developers to measure actual time savings (Q4 2025).
- Extend the framework to support privacy-preserving techniques (federated learning, differential privacy) for healthcare applications.
- Pursue standardisation of the constraint specification language (proposed as a TinyML community extension via the TinyML Foundation).
- Develop an automated model compression and hardware mapping tool using reinforcement learning to further reduce manual effort.

Declarations

Competing Interests

The authors declare no competing interests.

Funding

This research received no specific grant from any funding agency.

Data Availability

Not applicable. No new data were generated.

Author Contributions

D.L.B. Alwitigalage: conceptualisation, methodology, framework design, writing – original draft. M.M. Munawwer: literature review, gap analysis, validation planning. C.L. Jayathilake: algorithm design, implementation roadmap. All authors reviewed and approved the final manuscript.

References

[1] N. Schizas, A. Karras, C. Karras, and S. Sioutas, "TinyML for Ultra-Low Power AI and Large Scale IoT Deployments: A Systematic Review," *Future Internet*, vol. 14, p. 363, 2022.

[2] N. Alajlan and D. Ibrahim, "TinyML: Enabling of Inference Deep Learning Models on Ultra-Low-Power IoT Edge Devices for AI Applications," *Micromachines*, vol. 13, no. 6, p. 851, 2022.

[3] R. Kallimani, K. Pai, P. Raghunwanshi, S. Iyer, and O. Lopez, “TinyML: Tools, applications, challenges, and future research directions,” *Multimedia Tools and Applications*, vol. 83, pp. 29015–29045, 2023.

[4] V. Tsoukas, A. Gkogkidis, E. Boumpa, and A. Kakarountas, “A Review on the emerging technology of TinyML,” *ACM Computing Surveys*, vol. 56, pp. 1–37, 2024.

[5] R. Sanchez-Iborra and A. Skarmeta, “TinyML-Enabled Frugal Smart Objects: Challenges and Opportunities,” *IEEE Circuits and Systems Magazine*, vol. 20, pp. 4–18, 2020.

[6] F. Oliveira, D. Costa, F. Assis, and I. Silva, “Internet of Intelligent Things: A convergence of embedded systems, edge computing and machine learning,” *Internet of Things*, vol. 26, p. 101153, 2024.

[7] M. Diab and E. Rodr´ıguez-Villegas, “Embedded Machine Learning Using Microcontrollers in Wearable and Ambulatory Systems for Health and Care Applications: A Review,” *IEEE Access*, vol. 10, pp. 98450–98474, 2022.

[8] J. Lin, L. Zhu, W. Chen, W. Wang, and S. Han, “Tiny Machine Learning: Progress and Futures,” *IEEE Circuits and Systems Magazine*, vol. 23, pp. 8–34, 2024.

[9] S. Trilles, S. Hammad, and D. Iskandarayan, “Anomaly detection based on Artificial Intelligence of Things: A Systematic Literature Mapping,” *Internet of Things*, vol. 25, p. 101063, 2024.

[10] P. Ray, “A review on TinyML: State-of-the-art and prospects,” *J. King Saud Univ. Comput. Inf. Sci.*, vol. 34, pp. 1595–1623, 2021.

[11] L. Capogrosso, F. Cunico, D. Cheng, F. Fummi, and M. Cristani, “A Machine Learning-Oriented Survey on Tiny Machine Learning,” *IEEE Access*, vol. 12, pp. 23406–23426, 2023.

[12] S. Zaidi, A. Hayajneh, M. Hafeez, and Q. Ahmed, “Unlocking Edge Intelligence Through Tiny Machine Learning (TinyML),” *IEEE Access*, vol. 10, pp. 100867–100877, 2022.

[13] Y. Abadade, A. Temouden, H. Bamoumen, N. Benamar, Y. Chtouki, and A. Hafid, “A Comprehensive Survey on TinyML,” *IEEE Access*, vol. 11, pp. 96892–96922, 2023.

[14] A. Hayajneh, M. Hafeez, S. Zaidi, and D. McLernon, “TinyML Empowered Transfer Learning on the Edge,” *IEEE Open Journal of the Communications Society*, vol. 5, pp. 1656–1672, 2024.

[15] C. Zeinaty, W. Hamidouche, G. Herrou, and D. Menard, “Designing Object Detection Models for TinyML: Foundations, Comparative Analysis, Challenges, and Emerging Solutions,” *ACM Computing Surveys*, vol. 58, pp. 1–48, 2025.

[16] A. Trigkas, D. Piromalis, and P. Papageorgas, “Edge Intelligence in Urban Landscapes: Reviewing TinyML Applications for Connected and Sustainable Smart Cities,” *Electronics*, vol. 14, no. 14, p. 2890, 2025.

[17] L. Ortiz, I. Soliz, and V. Balarezo, “Advancing TinyML in IoT: A Holistic System-Level Perspective for Resource Constrained AI,” *Future Internet*, vol. 17, no. 6, p. 257, 2025.

[18] S. Lakshman and N. Eisty, “Software Engineering Approaches for TinyML based IoT Embedded Vision: A Systematic Literature Review,” in *2022 IEEE/ACM 4th International Workshop on Software Engineering Research and Practices for the IoT (SERP4IoT)*, 2022, pp. 33–40.

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- EmergingTecIEEEPosterFinal.pdf