CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

Hardware-Accelerated Deep Learning for Multimodal Biomedical Monitoring With on-Device Adaptive Learning on Resource-Constrained System-on-Chip

A thesis submitted in partial fulfillment of the requirements

For the degree of Master of Science in

Computer Engineering

By

Sara Khaled A Abdelhakeem Aly

May 2026

The thesis of Sara Khaled A Abdelhakeem Aly is approved:

Dr. Myung (Michael) Cho	Date
Dr. John Valdovinos	Date
Dr. Shahnam Mirzaei, Chair	Date

California State University Northridge

Acknowledgments

I owe my deepest gratitude to my thesis advisor and committee chair, Dr. Shahnam Mirzaei, whose guidance, vision, and unwavering support shaped every stage of this work. From the initial concept through hardware bring-up, model design, and the development of the on-device adaptive training system, Dr. Mirzaei provided the technical direction, encouragement, and rigorous feedback that made this thesis possible. His mentorship extended well beyond the laboratory—he taught me how to think as an engineer and a researcher, and I am profoundly grateful.

I also thank my committee members, Dr. Myung (Michael) Cho and Dr. John Valdovinos, for their thoughtful feedback, challenging questions, and encouragement throughout this journey.

I am grateful that my research was recognized and supported by the Thesis Support Committee. This support helped advance my project and strengthen the impact of my work, and I sincerely appreciate the confidence and opportunity it represents.

Portions of this thesis have been accepted for publication at the 2026 IEEE International Conference on Healthcare Informatics (IEEE ICHI 2026), further validating the contributions presented herein.

I am also grateful to everyone who supported me along the way—family, friends, and mentors—your time, insight, and encouragement have been invaluable.

Dedication

To my family, mentors, and friends, for their constant support and encouragement.

Table of Contents

List of Tables

List of Figures

Abstract

Hardware-Accelerated Deep Learning for Multimodal Biomedical Monitoring
With on-Device Adaptive Learning on Resource-Constrained System-on-Chip

By
Sara Khaled A Abdelhakeem Aly
Master of Science in Computer Engineering

Wearable health monitoring requires robust inference under real-world conditions including motion artifacts, missing sensors, and strict constraints on latency, memory, and connectivity. Cloud-centric processing is often incompatible with privacy and intermittent connectivity. This thesis presents an end-to-end edge intelligence system for multimodal biomedical monitoring that fuses electrocardiography (ECG), photoplethysmography (PPG), and inertial measurement unit (IMU) signals in a unified model input representation, while retaining electrodermal activity (EDA/GSR) as supplementary physiological context in the live sensing pipeline. The fusion of these modalities improves robustness to single-sensor dropout and provides richer physiological context than single-channel approaches.

The system uses a compact CNN-Transformer multi-task model (112,552 parameters) to predict activity (4 classes), stress (binary), and arrhythmia (binary) from 10-second windows at 100 Hz. Training uses focal loss, weighted sampling, augmentation, and three-phase fine-tuning across three

public datasets (PPG-DaLiA, WESAD, MIT-BIH; 21,704 windows, 65 subjects). Dataset strictly divided into subject-level splits to avoid identity leakage between training and evaluation. On a held-out set of 13 unseen subjects (10,134 windows), the system achieves 94.1% activity accuracy, 87.1% stress accuracy (AUC 0.975), and 90.8% arrhythmia accuracy (AUC 0.879, specificity 94.7%).

A deterministic motion gate enables context-aware alerting that decouples alert logic from classification errors. The full model (112,552 parameters, 450 KB) is deployed on an ESP32-S3 without compression, achieving 2,338 ms inference per window (23.4% duty cycle) at 0.54 W, suitable for wearable form factors where continuous cloud offload is impractical.

This thesis implements on-device adaptive training that enables feedback-driven partial parameter updates on the ESP32-S3. The system freezes the CNN backbone and backpropagates through classification heads using pre-allocated buffers. Additionally, resource gating, dual-slot model versioning with NVS persistence, A/B validation, and safety lockout are implemented. Runs show stable convergence under bounded memory; safety mechanisms limit destructive updates when labels are sparse or noisy. The adaptive subsystem adds only 8 KB of flash within the existing PSRAM budget, demonstrating that on-device learning is feasible on sub-$10 edge hardware for biomedical applications.

Chapter 1

Introduction

1.1 Motivation: continuous multimodal monitoring at the edge

The human body, viewed as an aggregate of independent systems, overlooks the crucial importance of health monitoring. Human health stems from the interplay among physiology, behavior, and environment. Likewise, mental state is related to the physical system; stress, emotional state, and cognitive load affect and manifest themselves through physiological markers such as heart rate, autonomic nervous system activity, and motion. This research takes a whole-person approach, assuming that data fusion across multiple sensors is necessary to understand physiological markers.

Similarly, effective smart technologies typically result from the **co-design** of sensing, computation, and algorithms within a unified system, rather than from advances in hardware or software alone. This perspective underpins the focus of this thesis on deploying deep learning models on low-power **systems-on-chip (SoCs)**, using an ESP32-S3-class microcontroller SoC as the edge target [1]—so that on-device inference (and future on-device adaptation) can be practical, low-latency, and privacy-preserving.

The problem of cardiac arrhythmia remains an ongoing issue for public health around the world and may lead to various diseases such as stroke, heart disease, and sudden death due to heart complications. In this regard, early detection plays a critical role [2]. The latest research emphasizes that wearable devices have significantly evolved, becoming not only consumer products but also clinical tools that can continuously monitor patients with virtually no limitations, recording physiological data similar to that recorded by ambulatory electrocardiographic monitors. Apart from clinical advantages, continuous monitoring can help prevent unnecessary hospital admissions by increasing the likelihood of detecting sporadic arrhythmic events [2].

This thesis is positioned within that broader trend of large-scale adoption. Smartwatches and other consumer wearables now routinely include heart rate variability (HRV), heart rate (HR), ECG,

and oxygen saturation; market analyses estimate strong growth, with industry valuation around USD 59B in 2021 and projected to reach roughly USD 100B by 2025 [3].

However, clinical reliability remains a central constraint. The ESC Working Group on e-Cardiology emphasizes the need for accuracy and reliability when using wearable ECG monitoring for atrial fibrillation detection (e.g., in cryptogenic stroke), and this perspective extends to broader arrhythmia screening: signal quality, artifact handling, and validation protocols strongly influence real-world utility [4]. These realities motivate the core design goals of this project: robustness to motion artifacts and missing sensors, disciplined train/test isolation, and an alerting protocol grounded in clinically meaningful contexts.

Conventional wearables typically compute simple summary metrics (e.g., step count or average heart rate) and offload advanced analytics to a paired phone or cloud service. This architecture can introduce avoidable **privacy exposure**, **latency**, and **battery/communication overhead**, and it makes reliability dependent on connectivity and external compute. Moreover, single-modality pipelines (e.g., IMU-only activity detection or PPG/ECG-only cardiac monitoring) are vulnerable to false positives and confounds in free-living conditions: motion artifacts can degrade PPG/ECG quality, and activity context can change the interpretation of physiological measurements.

In response to these constraints, this thesis addresses two central research questions: *(1) Can a compact, low-power SoC perform multimodal deep learning inference locally to enhance reliability and context-awareness while operating within stringent memory, computational, and energy limitations? (2) Can the same device adapt its model to individual users through on-device, feedback-driven training, without reliance on cloud connectivity, external machine learning frameworks, or compromising safety?* Specifically, the investigation examines whether **multimodal fusion (ECG/PPG combined with IMU data) and motion-aware alerting** improve reliability under movement and activity confounds, and whether **on-device adaptive training**—using clinician or user corrections to update classification heads in real time—can personalize the deployed model while hardware-enforced safety mechanisms prevent performance degradation.

1.2 Project objective and scope

This thesis develops an **edge intelligence system** for real-time multimodal biomedical monitoring using wearable sensors. The system performs **multi-task learning** on synchronized windows of physiological and inertial signals to jointly solve: **(i)** activity classification, **(ii)** stress recognition, and **(iii)** arrhythmia detection. These tasks are trained together because physiology is strongly context dependent: motion and stress modulate ECG/PPG morphology and autonomic signatures, and therefore influence whether an alert is clinically meaningful or expected.

Although ECG/PPG fusion, stress sensing, IMU pipelines, and embedded ML are each discussed in prior work, this thesis is distinctive in how it **brings them together** into one practical pipeline: multimodal physiological fusion (ECG/PPG/accelerometer), explicit motion context handling, and a clear embedded deployment pathway on a low-power SoC. This framing aligns with wearable fusion guidance that emphasizes selecting appropriate fusion levels and interpreting physiology in context rather than as isolated metrics [5], [6]. In practical terms, the contribution is the **bridging of gaps** between (i) multimodal physiological fusion, (ii) motion-aware alerting to reduce false alarms in free-living settings, (iii) embedded deep inference with a firmware-aligned data stream and exportable inference graph on constrained hardware, and (iv) on-device adaptive training that enables the deployed model to personalize to individual users through feedback-driven backpropagation entirely on the microcontroller, with hardware-enforced safety mechanisms to prevent model degradation [1], [7], [8].

Table 1.1. Integrated contributions of this thesis (high-level snapshot).

Integrated capability	**Implemented**
ECG + PPG fusion (physiological fusion)	**Yes**
GSR/EDA included for stress/autonomic context	**Yes**
IMU/accelerometer motion context used in alerting	**Yes**
Motion-aware alert logic (context-aware severity)	**Yes**
Edge deployment path on ESP32-S3-class SoC	**Yes**
Unified multitask CNN + Transformer-Lite model	**Yes**
On-device adaptive training (head-only backpropagation)	**Yes**
Feedback controller (drift detection + user corrections)	**Yes**
Dual-slot model versioning with NVS flash persistence	**Yes**
Hardware-enforced resource gating (PSRAM, heap, thermal)	**Yes**
Nine concentric safety mechanisms (A/B validation, lockout, factory reset)	**Yes**

Portions of this work have been accepted for publication at the 2026 IEEE International Conference on Healthcare Informatics (IEEE ICHI 2026).

The model is paired with a **context-aware alerting layer** that implements four clinically motivated scenarios (Cases 1–4) that combine physiological state with motion context (Chapter 3, Section 3.6). The work emphasizes:

- **Pipeline correctness**: no hidden train/test leakage and no silent sampling bugs.
- **Bias mitigation**: preventing majority-class collapse in imbalanced multi-source data.
- **Robustness to missing modalities**: supporting hardware that provides a reduced sensor suite compared to the full 5-channel deployed representation.

1.3 System overview

Overall, the architecture operates on the concept of wearable analytics, where signals from body-worn sensors are collected, divided into fixed-length windows, normalized, and synchronized across multiple modalities, and then processed using a small neural network. The network's output includes task probabilities for activity classification, stress levels, and arrhythmia detection, and post-processing provides the final alert decision based on the physiological predictions and motion context. Notably, the motion context required to provide the alert signal is calculated solely from accelerometer signals (Section 3.6.3).

Beyond fixed-model inference, the system includes an on-device adaptive training loop (Section 3.8) that enables the deployed model to improve over time. When clinician or user feedback indicates a misclassification, corrections are stored in an on-device ring buffer. A feedback controller monitors inference confidence and, when drift is detected or sufficient corrections accumulate, triggers retraining of the classification head parameters using backpropagation through pre-allocated buffers — all on the ESP32-S3 microcontroller, without cloud connectivity or external ML frameworks. Hardware-enforced safety mechanisms (resource gating, A/B validation, safety lockout, factory reset) ensure that adaptation cannot degrade the model below its pre-deployment baseline.

Chapter 2

Literature Review

Wearable health monitoring increasingly relies on multi-sensor fusion (physiological signals combined with inertial measurements) to improve robustness outside controlled laboratory settings, where motion artifacts, missing channels, and domain shifts are common [5]. Recent work surveys rapid advancements in wearable arrhythmia detection and discusses clinical implications and limitations, including the need for reliable long-term monitoring and careful interpretation under real-world conditions [2]. In parallel, professional guidance emphasizes accuracy, reliability, and validation when wearables are used for ECG-based screening (e.g., atrial fibrillation detection) [4].

From a systems perspective, large-scale consumer adoption creates both opportunity and risk. Market growth implies increased availability of rich physiological streams and broader deployment potential, but also increases the importance of robust algorithms that minimize false alarms, handle motion artifacts, and degrade gracefully when signals are missing or noisy [3].

Another limitation is added by on-device inference: embedded systems pose constraints on memory and latency [7], [8]. The thesis thus takes an approach in which the CNN backbone is enhanced with a *transformer-lite-based* fusion block to harness the benefits of attention without relying on powerful GPUs for experimentation.

2.1 On-device learning and feedback-driven adaptation

While most edge-deployed models operate with frozen weights after deployment, a growing body of work explores **on-device training** to address distributional drift, user-specific variation, and changing environmental conditions. TinyOL demonstrated online learning on Arduino-class microcontrollers using a simplified update rule but without model versioning or safety mechanisms [9]. TinyTL introduced bias-only training to reduce memory requirements by updating only bias parameters rather than full weight matrices [10]. TinyTrain extended this by selecting which layers

to update based on gradient importance, targeting Cortex-A class devices [11]. AIfES provides an open-source embedded AI framework supporting on-device training without external dependencies [12]. A comprehensive survey by Dhar et al. covers the broader landscape of on-device machine learning algorithms and their theoretical foundations [13].

However, none of these approaches simultaneously address the full set of requirements for safety-critical biomedical deployment: (i) framework-free bare-metal implementation with zero dynamic allocation, (ii) model versioning with persistent storage across reboots, (iii) A/B validation to prevent accuracy degradation, and (iv) hardware-enforced resource gating. This thesis builds on these foundations while adding the safety mechanisms necessary for wearable health monitoring, where a degraded model could produce clinically dangerous predictions (Section 3.8).

2.2 PPG versus ECG sensing in consumer wearables

Many consumer wearables rely on **photoplethysmography (PPG)** for continuous heart monitoring. PPG is a non-invasive optical technique: light emitted into the skin is partially absorbed and reflected by tissue and blood, and a photodetector measures reflected intensity changes that track pulsatile blood volume with each heartbeat [14], [15]. Conceptually, PPG contains a pulsatile component that enables heart-rate estimation, and a slower-varying component influenced by respiration, temperature, and autonomic nervous system activity, which can be informative but also introduces confounding variability [14].

In contrast, **electrocardiography (ECG)** directly measures the heart's electrical activity through electrodes. ECG remains the clinical gold standard for arrhythmia diagnosis because it provides detailed electrical waveforms; however, smartwatch implementations typically provide limited lead configurations (often single-lead), which constrains diagnostic granularity relative to clinical 12-lead ECG [4], [16].

2.2.1 Why PPG is attractive (and why it fails)

PPG is valued for simplicity, non-invasiveness, low cost, and suitability for continuous monitoring, which explains its ubiquity in smartwatches [14]. Nevertheless, its limitations are recurrent and central to wearable arrhythmia screening:

- **Indirect measurement**: PPG measures pulse surrogates rather than cardiac electrical conduction, which can make subtle arrhythmias harder to detect [15].
- **Motion artifacts**: physical activity and sensor displacement significantly degrade signal quality, with pronounced errors during exercise [14], [17].
- **Physiological and environmental sensitivity**: skin perfusion, ambient temperature, and autonomic tone can alter optical coupling and waveform morphology [14].
- **Timing resolution limits**: lower temporal resolution than ECG can reduce sensitivity to brief events [15].

These limitations directly motivate the thesis emphasis on context-aware decision-making and motion-robust pipelines: without explicit motion handling, PPG-driven systems can generate false positives during activity, which increases user burden and reduces clinical credibility [18].

2.2.2 Two-stage wearable screening: continuous PPG then confirmatory ECG

A common design pattern in leading consumer devices is a two-stage screening workflow: **continuous PPG monitoring** triggers an irregular rhythm notification, and the user performs an **on-demand ECG recording** for confirmation [18], [19], [20]. Large-scale studies demonstrate the feasibility of passive wearable detection in real-world populations, but also highlight that performance depends on adherence, signal quality, and careful thresholding to manage false alerts [19], [21].

2.2.3 Beyond smartwatches: rings and patch monitors

Other wearable form factors trade capability for comfort and wearability. **Smart rings** can provide stable positioning and continuous sensing with less intrusiveness than watches, but many models offer limited functionality compared to smartwatches and may not include ECG acquisition [22]. **Patch monitors** and other continuous ECG monitors can capture a broader range of clinically relevant arrhythmias; however, device heterogeneity and differences in sensing modalities complicate integration into healthcare workflows and complicate cross-device algorithm validation [23], [24].

2.2.4 Validation gaps, privacy, and cost drivers

One persistent concern is limited validation of wearables for arrhythmias beyond atrial fibrillation, even as research explores broader rhythm classes [22], [25]. Recent edge-oriented ECG work also highlights that detection performance depends strongly on policy-aware operating-point selection, not only raw classifier outputs [26]. In addition, direct-to-consumer ECG technologies raise open questions about ethics, legal responsibility, and data privacy, particularly when sensitive physiological data is transmitted to cloud services [2].

These constraints motivate two long-term directions reflected in this thesis. First, algorithmic research should prioritize improved specificity and reduced false positives via machine learning and context-aware fusion (e.g., explicitly modeling motion artifacts) [2]. Second, **standalone edge processing** (and eventually on-device adaptation on SoC/FPGA platforms) can reduce privacy exposure by keeping inference and selective personalization local, while also reducing infrastructure and operational costs [7], [8].

2.3 Multisensor fusion background and our fusion strategy

Multi-sensor data fusion has become central to wearable health monitoring as demand grows for accurate, continuous, and non-invasive inference outside laboratory environments [5]. The value of fusion is most evident when signals are degraded by motion artifacts or when a sensor becomes unreliable or unavailable; complementary modalities can preserve inference quality by contributing more informative evidence [5]. At the same time, wearables impose practical constraints: heterogeneous acquisition systems, varying data quality, power demands, and robustness to node failure all influence which fusion designs are feasible [5].

The fusion literature commonly distinguishes **signal-level fusion** (combining raw sensor streams), **feature-level fusion** (combining extracted or learned representations), and **decision-level fusion** (combining model outputs such as voting or Bayesian inference) [5], [6]. This framing is particularly useful for wearables: many devices can measure routine outcomes (e.g., heart rate, SpO_2, sleep) and newer modalities (e.g., ECG, EDA) in free-living settings, but these streams are often analyzed in isolation and are difficult to integrate without explicit fusion methodology [6]. A key motivation is holistic monitoring (sometimes discussed in terms of "digital twinning"): interpreting physiology in context (activity, sleep, stress) rather than as isolated measurements [6].

However, integration introduces practical challenges. Wearable sensors can have different sampling frequencies and timing jitter, making synchronization non-trivial; this is especially important for scenarios that require simultaneous interpretation across modalities [6]. Additionally, selecting a fusion level should match the intended outcome: for example, signal-level fusion can be natural when raw streams are comparable (e.g., inertial acceleration and angular velocity for movement estimation), whereas feature- or decision-level fusion is often more feasible when modalities are not directly comparable [6]. Finally, for biomedical deployment it is important to establish verification and validation processes so that fused outputs are fit-for-purpose [6].

Beyond "static" fusion designs, another line of work uses **context-aware selective fusion**, where the system dynamically chooses *which* sensors (or fusion schemas) to use based on the

user context. For example, SELF-CARE proposes a fully wrist-based stress detection approach that models context as subject motion and uses an intelligent gating mechanism to select fusion strategies accordingly [27]. On the WESAD dataset, SELF-CARE reports 86.3% accuracy for a 3-class stress problem and 94.1% for a 2-class problem [27]. This adaptive fusion is attractive for **low-power edge** systems because it can reduce computation and energy when certain sensors are unnecessary or unreliable under motion. While the present thesis uses feature-level fusion in a single end-to-end network (rather than dynamically switching sensor subsets), it adopts a related principle at the **decision level**: motion context is explicitly gated into alerting logic to prevent cascaded failures and clinically meaningless alerts during exercise. A direct numeric comparison is not claimed here because the tasks and label definitions differ (e.g., SELF-CARE's 2/3-class stress formulation on WESAD versus this thesis' multi-task setting and constructed four-case benchmark), but SELF-CARE motivates selective fusion as a promising future extension for power-aware on-device deployment.

In this thesis, fusion is implemented primarily as **feature-level fusion within a single end-to-end network**. All modalities are represented in a unified window tensor, processed by shared convolutional layers for temporal feature extraction, and fused via cross-channel modeling (Transformer-Lite) before task-specific decision heads. In the terminology above, this corresponds to feature-level fusion for representation learning, coupled with a lightweight **decision-level** fusion step at alert time (rule-based gating using accelerometer-derived motion context; Section 3.6.3). This pairing supports robustness under sensor dropouts and aligns with practical wearable guidance that fusion level selection should follow the desired outcome and data comparability [6].

2.3.1 Gap addressed by this thesis

A recurring theme in the literature is that wearable datasets and algorithms are often studied as *separate* pieces (e.g., physiology-only modeling, IMU-only activity recognition, or cloud-based analytics), while truly holistic monitoring requires both methodological fusion choices and system-level integration [5], [6]. This thesis directly targets that gap by combining multimodal fusion

inside a compact deep model, using motion context to prevent clinically meaningless alerts under exercise, and demonstrating an embedded deployment pathway on a microcontroller-class SoC. This framing helps interpret why the main contribution is **reliable context-aware alerting** rather than maximizing fine-grained activity classification accuracy.

Chapter 3

Methodology

3.1 Signals, sampling, and windowing

All signals are standardized into a unified time-series tensor of shape $[C \times T]$, where $C = 5$ channels and $T = 1000$ samples. Each sample corresponds to a $10\,\mathrm{s}$ window at a unified sampling rate of $100\,\mathrm{Hz}$. Windows are generated with 50% overlap ($5\,\mathrm{s}$ stride), which increases the effective number of training examples while maintaining temporal continuity. The 10-second duration balances capturing sufficient cardiac morphology for arrhythmia discrimination with latency and memory constraints for streaming inference.

3.2 Unified channel mapping

The unified channel index mapping is fixed (see `src/dataset_integration.py`) and used consistently across training, testing, and synthetic sample construction:

Table 3.1. Unified input tensor channel mapping (5 channels).

Index	Channel name	Description
0	ECG	Electrocardiogram
1	PPG	Photoplethysmogram
2	Accel_X	3-axis accelerometer (X)
3	Accel_Y	3-axis accelerometer (Y)
4	Accel_Z	3-axis accelerometer (Z)

3.3 Datasets and unified dataset construction

Three public datasets are integrated into a unified five-channel tensor format. Each dataset provides ground-truth labels for a different subset of tasks; unification produces a consistent representation while preserving reliable labels where they exist. Table 3.2 summarizes the three sources.

Table 3.2. Source datasets integrated into the unified training corpus. Each dataset supplies labels for one primary task; missing task labels are masked during training (Section 3.5.3).

Dataset	Windows	Subjects	Modalities available	Labels provided
PPG-DaLiA [28]	12,806	15	PPG, ACC (wrist); ECG (chest)	Activity
WESAD [29]	3,439	15	ECG, PPG, ACC, EDA, TEMP	Stress
MIT-BIH [30], [31]	5,459	48 records	ECG (2-lead Holter)	Arrhythmia
Total	**21,704**	**65**		

- **PPG-DaLiA** [28]: 15 subjects performing scripted daily activities (sitting, walking, cycling, driving, working, lunch, stair climbing, playing table soccer). Recorded with wrist PPG/accelerometer (Empatica E4) and chest ECG (RespiBAN). Used as the primary source of activity supervision. Activity labels are merged into four classes for training: Sedentary, Walking, Cycling, and High-Intensity.

- **WESAD** [29]: 15 subjects in a controlled lab protocol with baseline, stress (Trier Social Stress Test), amusement, and meditation conditions. Chest-worn RespiBAN provides ECG, PPG, accelerometer, EDA, and temperature. Used as the primary source of binary stress labels (stress vs. non-stress). *Note: PPG-DaLiA and WESAD share the S1–S17 subject-ID namespace, yielding 17 unique physiological subjects across both wearable-sensor protocols.*

- **MIT-BIH Arrhythmia Database** [30], [31]: 48 half-hour two-lead ECG recordings with cardiologist-annotated beat-level arrhythmia labels. Used as the primary arrhythmia supervision source. Beat annotations are aggregated into window-level binary labels (normal vs. abnormal).

3.3.1 Unified dataset construction pipeline

The integration pipeline (`src/dataset_integration.py`) processes each source dataset through the following steps:

1. **Channel mapping**: Each dataset's signals are mapped to the unified 5-channel format (Table 3.1). For datasets missing a modality (e.g., MIT-BIH has no PPG or accelerometer), the corresponding channels are deterministically zero-filled.

2. **Resampling**: All channels are resampled to the unified 100 Hz rate using linear interpolation.

3. **Windowing**: Continuous recordings are segmented into non-overlapping 10-second windows (1,000 samples per channel), producing input tensors of shape $[5 \times 1000]$.

4. **Per-window normalization**: Z-score normalization (mean subtraction, division by standard deviation) is applied independently to each channel within each window, with a safe fallback for constant-valued channels (standard deviation $< 10^{-6}$).

5. **Label assignment**: Each window receives labels only for the task(s) annotated by its source dataset. Missing labels are represented as -1 and excluded from loss computation via loss masking (Section 3.5.3).

3.3.2 Subject-wise data splitting

The dataset for the entire task is partitioned into training, validation, and test splits, ensuring no data leakage, as the partitions are performed at the **subject level**. There are no overlaps between the subjects across the splits. The test partition consists of 13 subjects who have not been seen before (10,134 windows) across the three datasets. The details of the partitions can be found in Section 3.5.1.

3.4 Model architecture: CNN-SE-Transformer (V5)

The deployed model (V5, `CNNTransformerV3`) is designed for edge constraints while retaining multimodal fusion capability. The architecture combines three complementary components: a **convolutional neural network (CNN)** for local feature extraction, **squeeze-and-excitation (SE) channel attention** for adaptive feature recalibration, and a **2-layer Transformer encoder** for global temporal modeling through multi-head self-attention [7], [8]. The full 112,552-parameter model is deployed without compression—the training model is the deployed model, with batch normalization fused into convolution weights at export time. Figure 3.1 shows the high-level pipeline, and Figure 3.2 details the internal architecture of each component.

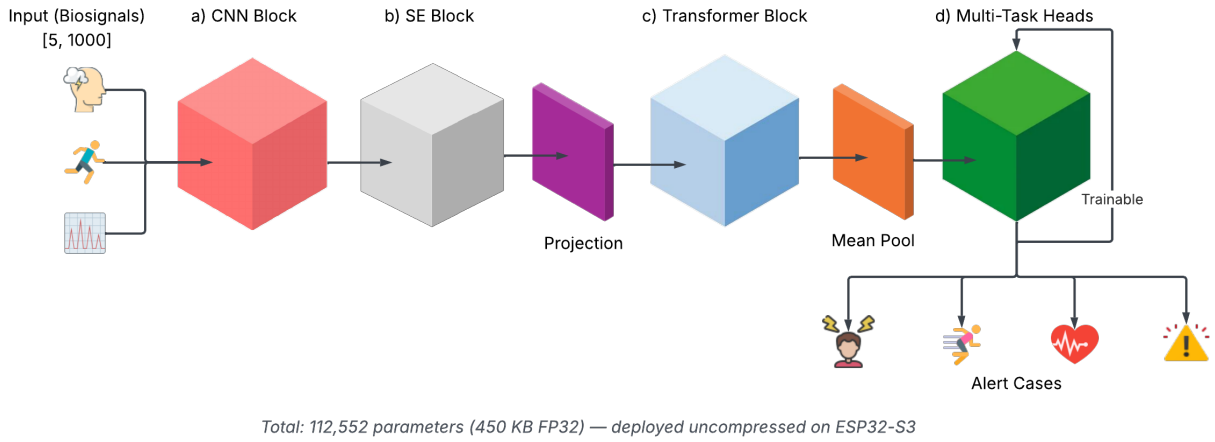

Figure 3.1. CNN-SE-Transformer pipeline (V5, 112,552 parameters). Five-channel input windows (ECG, PPG, AccX, AccY, AccZ at 100 Hz $\times$1000 samples) are processed through three sequential Conv1d layers ($5\rightarrow32$, $32\rightarrow64$, $64\rightarrow64$) with max-pooling, squeeze-and-excitation channel attention, a linear projection with LayerNorm and sinusoidal positional encoding, and a 2-layer Transformer encoder (4 heads, d_{ff}=128, GELU, pre-norm). Mean pooling produces a 64-dimensional embedding that feeds three parallel task heads. Component details are shown in Figure 3.2.

Design rationale. The hybrid CNN–Transformer architecture is motivated by the complementary nature of local and global temporal patterns in biomedical signals. ECG and PPG waveforms contain short-duration morphological features—QRS complexes ($\sim$70–100 ms), PPG systolic peaks, and motion impulses—that are efficiently captured by 1D convolutions with small kernel sizes. A pure CNN, however, would require very deep stacking or impractically large kernels to model

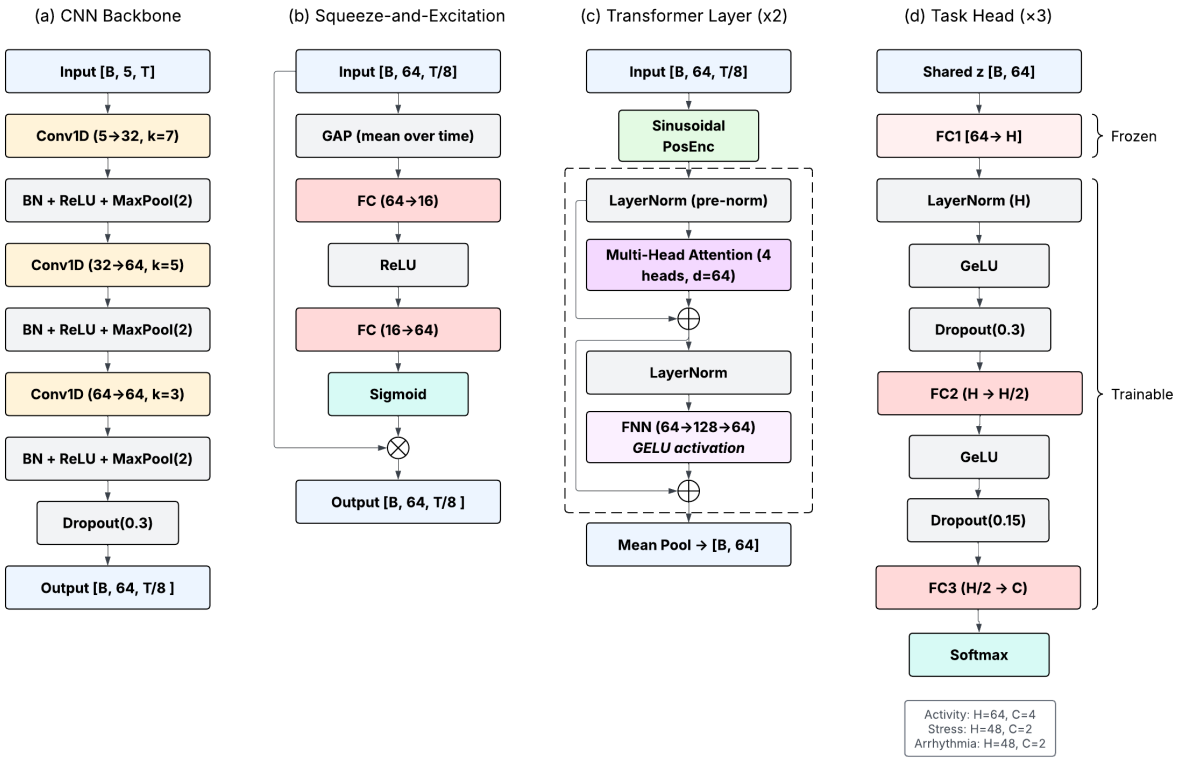

Figure 3.2. Detailed component architectures: (a) CNN backbone with three sequential Conv1d layers (kernel sizes 7, 5, 3) and max-pooling that reduce the temporal dimension from 1000 to 125 while expanding channels from 5 to 64; (b) Squeeze-and-Excitation block performing global average pooling followed by FC(64→16, ReLU) and FC(16→64, Sigmoid) to produce per-channel scaling weights; (c) Pre-norm Transformer encoder layer (×2) with multi-head self-attention (4 heads, d_k=16) and feed-forward network (64→128→64, GELU activation), each with residual connections; (d) 3-layer task head (×3) with FC1 (frozen during on-device adaptation), LayerNorm, GELU, FC2, GELU, and FC3, where the trainable portion (LN+FC2+FC3) totals 4,984 parameters across all three heads.

dependencies across the full 10-second window, such as irregular R–R intervals or sustained elevated heart rate patterns indicative of stress. Self-attention in the Transformer encoder addresses this limitation: each temporal position can attend to all others, enabling the model to capture long-range physiological dependencies in a single layer. However, applying self-attention directly to the raw 5-channel $\times$ 1000-sample input would be computationally prohibitive on a microcontroller, since attention complexity scales as $O(n^2)$ with sequence length. The CNN backbone solves this by compressing the temporal dimension from 1000 to 125 through three stages of max-pooling, reducing the Transformer's computational cost by $64\times$ (125^2 vs. 1000^2 attention entries). The SE attention block between the CNN and Transformer serves a complementary role: because the five input

channels (ECG, PPG, AccX, AccY, AccZ) carry different diagnostic value for each task—ECG dominates arrhythmia detection, accelerometers dominate activity recognition, and PPG combined with ECG contributes most to stress detection—the learned per-channel rescaling allows the model to adaptively emphasize the most informative modalities on a per-sample basis.

3.4.1 CNN Feature Extractor

The CNN front-end processes the input window ($[5 \times 1000]$) through three 1D convolutional layers with squeeze-and-excitation (SE) channel attention:

1. **Layer 1**: Conv1d($5 \to 32$, kernel=7, padding=3) + BatchNorm + ReLU + MaxPool(2)
2. **Layer 2**: Conv1d($32 \to 64$, kernel=5, padding=2) + BatchNorm + ReLU + MaxPool(2)
3. **Layer 3**: Conv1d($64 \to 64$, kernel=3, padding=1) + BatchNorm + ReLU + MaxPool(2) + Dropout
4. **SE Attention**: Squeeze-and-Excitation block on the 64-channel output for adaptive channel recalibration

The hierarchical kernel sizes (7, 5, 3) create an expanding receptive field that captures features at multiple temporal scales—from individual heartbeats ($\sim$70ms at k=7) to longer physiological patterns. The cross-channel convolutions implicitly learn correlations between different sensor modalities: ECG rhythm with accelerometer motion patterns and PPG morphology changes. The SE attention block adaptively recalibrates channel responses, emphasizing informative features. Max-pooling across three stages reduces the temporal dimension from 1000 to 125 samples ($1000/2^3$), compressing the 10-second window while retaining salient features.

3.4.2 Transformer-Lite Encoder

The CNN output (shape $[B, 64, 125]$) is projected to the model dimension ($d_{\text{model}} = 64$) and transposed to sequence format $[B, 125, 64]$. Sinusoidal positional encodings are added to preserve temporal ordering information.

The Transformer-Lite encoder consists of 2 layers, each containing:

- **Multi-head self-attention**: 4 attention heads (each with $d_{\text{model}}/n_{\text{head}} = 64/4 = 16$ dimensions) enable the model to attend to different temporal aspects simultaneously — for example, one head may focus on local R-R interval patterns while another captures longer-range rhythm regularity. Each position can attend to all other positions, capturing long-range dependencies critical for detecting irregular rhythms across the 10-second window. Note: these 4 *attention heads* are distinct from the 3 *task classification heads* described below.

- **Feed-forward network**: dimension 128 with GELU activation provides non-linear feature transformation.

- **Layer normalization and residual connections**: stabilize training and enable gradient flow.

- **Dropout (0.2)**: prevents overfitting to training patterns.

The output sequence is globally averaged over time to produce a single 64-dimensional embedding representing the entire window.

3.4.3 Multi-Task Classification Heads

Three parallel 3-layer classification heads with LayerNorm process the shared embedding:

- **Activity head**: Linear($64 \to 64$) + LayerNorm + GELU + Dropout(0.3) + Linear($64 \to 32$) + GELU + Dropout(0.15) + Linear($32 \to 4$)

- **Stress head**: Linear($64 \to 48$) + LayerNorm + GELU + Dropout(0.3) + Linear($48 \to 24$) + GELU + Dropout(0.15) + Linear($24 \to 2$)

- **Arrhythmia head**: Linear($64 \rightarrow 48$) + LayerNorm + GELU + Dropout(0.3) + Linear($48 \rightarrow 24$) + GELU + Dropout(0.15) + Linear($24 \rightarrow 2$)

This shared-backbone design enables feature reuse across tasks while allowing task-specific decision boundaries through the separate heads. The LayerNorm in each head stabilizes gradient flow during both training and on-device adaptation. In the on-device adaptive training system (Section 3.8), the CNN backbone, SE attention, Transformer layers, and projection layer are frozen after deployment; only these three classification heads are retrained using feedback-driven backpropagation, reducing the trainable parameter count to 4,984 (out of 112,552 total) and enabling safe, bounded adaptation on the microcontroller.

3.4.4 Model Specifications

Table 3.3. Model specifications (training-time vs embedded deployment).

Component	**Specification**
Input window	5×1000 (10 s @ 100 Hz)
CNN front-end	Conv1d $5 \rightarrow 32, k = 7$ + maxpool 2 Conv1d $32 \rightarrow 64, k = 5$ + maxpool 2 Conv1d $64 \rightarrow 64, k = 3$ + maxpool 2 SE attention (channel recalibration)
Transformer encoder	projection $d_{\text{model}} = 64$, $n_{\text{head}} = 4$, $L = 2$
Pooling	global average pool $\rightarrow 64$
Task heads	Activity $64 \rightarrow 64 \rightarrow 32 \rightarrow 4$ (3-layer + LayerNorm + GELU) Stress $64 \rightarrow 48 \rightarrow 24 \rightarrow 2$ (3-layer + LayerNorm + GELU) Arrhythmia $64 \rightarrow 48 \rightarrow 24 \rightarrow 2$ (3-layer + LayerNorm + GELU)
Total parameter count	112,552 (FP32 weights ≈ 450 KB)
Runtime memory (measured)	heap free/min: 293.7/288.5 KB; PSRAM free/min: $\sim$7.3/7.2 MB
Program flash (measured)	sketch size/free: 750 KB (767,936 B, 23.0%) / 2.86 MB
Inference time (measured)	2,338 ms per window
Power consumption	0.54 W (5.23 V @ 0.1 A)

3.4.5 Fusion implementation in this thesis

This project implements **feature-level multimodal fusion** end-to-end. Input modalities are aligned into a single unified tensor ($[5 \times 1000]$ per window) and passed through shared temporal convolutions with SE attention that learn local patterns within and across channels. The CNN extracts modality-specific and cross-modality features, while the 2-layer Transformer encoder enables global temporal modeling through self-attention—each time step can attend to all others, capturing dependencies between distant events (e.g., correlating motion artifacts with ECG morphology changes, or identifying stress patterns that span multiple heartbeats).

For embedded deployment on ESP32-S3, the full model (CNN + SE attention + Transformer + task heads) is deployed without compression or stripping, totalling 112,552 parameters (≈450 KB FP32). The training model is identical to the deployed model, eliminating accuracy loss from model simplification. This design aligns with common fusion taxonomies (signal-level vs feature-level vs decision-level) used in the wearable fusion literature, while remaining practical under edge constraints and sensor unreliability [5].

3.4.6 Hardware-aware acceleration on resource-constrained SoCs

In contrast to traditional deep learning pipelines, which leverage GPU acceleration or cloud-based inference systems, the proposed pipeline is intentionally designed to run end-to-end on SoC-based devices without GPUs. By referring to "hardware-aware acceleration" in this thesis, it must be emphasized that this does not imply the existence of accelerator chips like NPUs or TPUs; rather, hardware-aware acceleration refers to techniques that optimize computational tasks by making use of the characteristics of the device in use, e.g., deterministic processing.

At deployment, the full neural architecture (CNN + SE attention + 2-layer Transformer + 3-layer task heads) is deployed without simplification, implemented directly in C++ using fixed-size buffers rather than a general-purpose machine learning runtime. Batch normalization parameters are folded into convolution weights, reducing runtime operations and improving numerical determinism. This eliminates interpreter overhead and enables consistent latency on the ESP32-S3-class target.

The motion-aware alert gating (Section 3.6.3) design shifts computation away from neural inference toward hardware-efficient statistical operations, reducing power consumption and preventing cascading failures when activity or stress predictions are uncertain. In this sense, hardware-aware acceleration is achieved not through specialized accelerators, but through deliberate partitioning of learned and deterministic computation to match the constraints of embedded hardware.

3.4.7 End-to-end dataflow

The system is designed to operate as a closed-loop monitoring pipeline rather than a standalone classifier. At a high level it consists of: **(i)** sensor acquisition and health tracking, **(ii)** window buffering and preprocessing into the unified tensor, **(iii)** multi-task inference, **(iv)** context-aware alerting and reporting, and **(v)** optional on-device adaptive training, where user corrections trigger bounded backpropagation through the classification heads using the cached feature vector from step (iii) (Section 3.8). This decomposition is intentional: it places the most safety-critical and least ambiguous logic (motion context) outside the neural network, while keeping physiological pattern recognition inside the learned model.

On-device operation depends on stable semantics: every inference step consumes the same ordered 5×1000 tensor corresponding to the 5 deployed channels (ECG, PPG, AccX, AccY, AccZ). The firmware treats channel ordering as an interface contract between sensing and inference, and emits status flags so that missing or unhealthy sensors can be detected upstream.

3.5 Training methodology and bias correction

The training process using heterogeneous multi-source data yielded several modes of failure, which include collapse into the majority class, such as activity collapsing to “Sitting,” “Always normal” arrhythmia collapse, and leakage possibilities due to naive splits. The training process in `train_model.py` solves these problems by implementing subject-wise splitting, balanced sampling, class-weighted losses, and loss masking.

3.5.1 Subject-wise splitting (leakage prevention)

To prevent leakage, train/validation/test splits are performed at the subject level. Evaluation can be restricted to held-out subjects using the stored split metadata, preventing windows from the same person from appearing in both train and test.

3.5.2 Balanced sampling and sampler-index correctness

Balanced sampling assigns each training window a sampling weight inversely proportional to the frequency of its most underrepresented label combination, ensuring that minority and safety-critical classes (e.g., abnormal arrhythmia, non-baseline stress) appear in training batches at rates closer to uniform rather than at their natural (skewed) frequencies. This is implemented via PyTorch's `WeightedRandomSampler`, which draws indices with replacement according to these per-sample weights. A critical implementation detail is that `WeightedRandomSampler` yields indices into the weights vector, which must be mapped back into dataset indices; otherwise, balancing silently fails and minority cases do not appear in training batches.

3.5.3 Loss design and loss masking

Each task uses cross-entropy loss with class weights derived from inverse class frequencies (clipped to avoid unstable gradients). Loss masking is used to prevent learning from placeholder labels in datasets where a task label is not meaningful (e.g., activity labels in ECG-only arrhythmia sources). For activity-only robustness samples (constructed by zeroing all channels except accelerometer), stress and arrhythmia losses are masked to avoid degrading the safety-critical heads with unrealistic physiological negatives.

In addition, task losses are combined using explicit task weights to balance learning across heterogeneous label spaces. Let $L_{\text{act}}, L_{\text{stress}}, L_{\text{arr}}$ denote the per-head losses for activity, stress, and arrhythmia, and let $m_{\text{act}}, m_{\text{stress}}, m_{\text{arr}} \in \{0, 1\}$ indicate whether a label is available for that sample.

The masked multi-task objective is:

$$L = w_{\text{act}}\, m_{\text{act}} L_{\text{act}} + w_{\text{stress}}\, m_{\text{stress}} L_{\text{stress}} + w_{\text{arr}}\, m_{\text{arr}} L_{\text{arr}}. \tag{3.1}$$

where $w_{\text{act}}, w_{\text{stress}}, w_{\text{arr}}$ are scalar task weights that control the relative importance of each objective, $m_{\text{act}}, m_{\text{stress}}, m_{\text{arr}} \in \{0, 1\}$ are binary masks indicating whether a ground-truth label is available for that task in a given sample, and $L_{\text{act}}, L_{\text{stress}}, L_{\text{arr}}$ are the per-task cross-entropy losses.

In the final configuration, arrhythmia detection is prioritized because it is the most safety-critical task for alerting, while activity is treated as auxiliary context rather than the primary objective (Section 3.6.3). The final training checkpoint uses weights $w_{\text{act}} = 1.5$, $w_{\text{stress}} = 1.5$, $w_{\text{arr}} = 2.0$, giving arrhythmia 33% higher weight than the other tasks. This $2.0/1.5$ ratio (equivalently, $1.33 : 1$) reflects a deliberate design choice: arrhythmia misclassification has the most severe clinical consequences (missed cardiac events), so the loss function allocates proportionally more gradient signal to the arrhythmia head. The ratio was determined empirically through a weight sweep (Table 3.4) that confirmed Case 3 alert correctness is robust across reasonable weight configurations.

Table 3.4 demonstrates that, across different weighting configurations, the system maintains robust correctness for the Case 3 alert scenario, highlighting that alert reliability is largely insensitive to reasonable trade-offs in multi-task performance.

Table 3.4. Representative task-weight sweep observations illustrating robustness of safety-critical alert behavior under different training trade-offs.

Weights (A/S/R)	**Activity**	**Arrhythmia**	**Case 3**
1.0 / 1.5 / 2.0 (arrhythmia-upweighted)	$\sim$7%	78%	100%
2.5 / 1.5 / 1.0 (activity-upweighted)	13%	88%	100%
1.5 / 1.0 / 1.5 (balanced)	6%	78%	100%

3.5.4 Overfitting mitigation: early stopping and stronger regularization

During development, training for too many epochs caused severe generalization collapse (especially for arrhythmia detection) even while training accuracy continued increasing. The final training pipeline therefore uses **validation monitoring** with **early stopping** and stronger regularization:

- **Early stopping**: training runs with a maximum epoch budget (25 epochs), but stops automatically if validation loss does not improve for a patience window (5 epochs). The best checkpoint (lowest validation loss) is restored at the end of training.
- **Increased dropout**: dropout was increased from 0.1 to 0.2 in the Transformer-Lite encoder and task heads, and `Dropout1d` layers were added in the CNN feature extractor.
- **Increased weight decay**: weight decay was increased from 10^{-5} to 10^{-4} to strengthen L2 regularization.

These controls are documented in the saved training artifacts (e.g., `training_log.txt` and `comprehensive_training_results.json`) and are critical for avoiding the overfitting regime where arrhythmia detection collapses.

3.6 Alerting system and the 4-case protocol

3.6.1 Four alert cases

The system defines four clinically motivated scenarios that combine a physiological state (stress or arrhythmia) with a motion context (sedentary or active). Each scenario specifies the expected alert output, enabling systematic verification of both the neural model and the motion-gating logic:

- **Case 1 (Stress + Sedentary)**: The model detects elevated stress (stress class $\neq$ baseline, stress score $> \tau_s = 0.35$) while the subject is stationary (motion score $\leq \tau_m$). Expected output: `psychological_stress`. This case validates that genuine stress during rest triggers an appropriate alert.

- **Case 2 (Stress + Exercise)**: The model detects elevated stress, but the subject is in motion (motion score $> \tau_m$). Expected output: `no_alert`. Elevated physiological stress during exercise is expected and should not generate an alert — suppressing it reduces false-alarm burden.

- **Case 3 (Arrhythmia + Sedentary)**: The model detects abnormal cardiac rhythm while the subject is at rest. Operational criterion: arrhythmia probability $> \tau_a = 0.70$. Expected output: `arrhythmia_detected`. This is the highest-priority clinical scenario.

- **Case 4 (Arrhythmia + High Motion)**: Abnormal rhythm detected during physical activity. Expected output: `critical_alert`. Arrhythmia during exercise is potentially life-threatening and warrants the most urgent alert level.

3.6.2 Motion score definition and calibration

Let $a_x, a_y, a_z \in \mathbb{R}^T$ be the accelerometer channels in a window (unified channels 2–4). The evaluator computes a scalar motion score as the mean of per-axis standard deviations:

$$s_{\text{motion}} = \frac{1}{3}\left(\sigma(a_x) + \sigma(a_y) + \sigma(a_z)\right) \tag{3.2}$$

A motion threshold is then calibrated from the constructed benchmark by comparing sedentary cases (Cases 1 and 3) to motion cases (Cases 2 and 4), using the midpoint between the median sedentary score and median motion score. If the separation is too small, the evaluator falls back to activity-based motion context.

Table 3.5. Four-case alerting protocol (expected outputs).

Case	Physiology	Context	Expected alert
1	Stress	Sedentary	`psychological_stress`
2	Stress	Exercise	`no_alert`
3	Arrhythmia	Sedentary	`arrhythmia_detected`
4	Arrhythmia	High motion	`critical_alert`

Algorithm 1 Motion-aware alert decision

Require: Window x, stress class c_s, stress score p_s, arrhythmia prob. p_a
Require: Thresholds: τ_s=0.35, τ_a=0.70, τ_m (calibrated)
$s_m \leftarrow \frac{1}{3}(\sigma(a_x) + \sigma(a_y) + \sigma(a_z))$
`moving` $\leftarrow (s_m > \tau_m)$
if $p_a > \tau_a$ **then**
    return `moving ? critical : arrhythmia`
else if $c_s =$ `Stress` **and** $p_s > \tau_s$ **then**
    return `moving ? no_alert : stress`
else
    return `no_alert`
end if

3.6.3 Motion-aware gating (avoid cascading activity errors)

Early versions of alerting used predicted activity to determine sedentary vs motion context, which created cascading failures when activity was misclassified. The final evaluator (`test_accuracy_4_cases.py`) computes a direct motion score from accelerometer channels (2–4) and calibrates a motion threshold from the test set; alert gating is primarily driven by this motion statistic rather than by the activity label, as formalized in Algorithm 1.

The evaluator script (`test_accuracy_4_cases.py`) operates in four stages: (1) it loads the trained model checkpoint and the constructed test set (250 samples per case); (2) it runs inference on each sample, collecting per-task predictions, confidence scores, and the raw accelerometer channels; (3) it computes a motion score per window as the mean of per-axis standard deviations (Eq. 3.2) and calibrates a motion threshold from the median separation between sedentary and motion cases; and (4) it applies the alert decision logic (Algorithm 1) and tabulates per-case accuracy, alert correctness, and aggregate metrics. The script saves all results to a canonical JSON file (`accuracy_results_4_cases.json`) for reproducibility.

This design choice is intentional and reflects embedded HW–SW co-design thinking: instead of forcing the SoC to solve the hardest (and least reliable) subproblem end-to-end, the system decomposes decisions into **neural** and **deterministic** components aligned with reliability and resource constraints. The neural model focuses on physiological interpretation (stress and arrhythmia), while the motion gate uses lightweight signal statistics that are explainable, testable, and cheap

to compute. This hybrid formulation reduces the risk of silent failures and helps keep on-device behavior predictable under distribution shift and motion artifacts [6], [7].

This motion-aware gating principle is also consistent with context-aware fusion approaches in the stress literature, where motion is used as an explicit context signal for selecting or conditioning the fusion strategy [27].

3.7 Hardware deployment and firmware integration

3.7.1 Hardware sensor subset (5 channels)

The V5 model is trained and deployed using 5 channels (ECG, PPG, AccX, AccY, AccZ) that directly match the hardware sensor suite: ECG (AD8232), PPG (MAX30102), and 3-axis accelerometer (MPU6050). Under the unified mapping, these correspond to channels 0–4. The training model and deployed model are identical — no channel stripping or missing-modality zero-filling is required, because the model is natively designed for the 5-channel input available from the target hardware.

3.7.2 ESP32-S3 firmware: MAX30102 + GSR support

The ESP32-S3 firmware was extended to support MAX30102 (I2C PPG) and an analog GSR/EDA sensor alongside the existing AD8232 ECG and MPU6050 IMU. The firmware architecture uses conditional compilation flags to enable/disable sensors, shares the I2C bus between MPU6050 (0x68) and MAX30102 (0x57), and expands the streaming format to include all sensor channels and status flags for sensor health. These implementation details ensure the deployed data stream matches the model's expected channel order and supports graceful degradation when an optional sensor is unavailable.

3.7.2.1 Prototype wiring (matches firmware pin map)

Figure 3.4 shows the prototype wiring used for the ESP32-S3-DevKitC-1-N8R8 sensor stack. The connections reflect the firmware pin mapping in `firmware/esp32/include/pins_esp32s3_devkitc1.h`: AD8232 ECG output is sampled on an ADC-capable GPIO, GSR/EDA is sampled on a second ADC pin, and both MPU6050 and MAX30102 share the same I2C bus (SDA/SCL) with distinct device addresses.

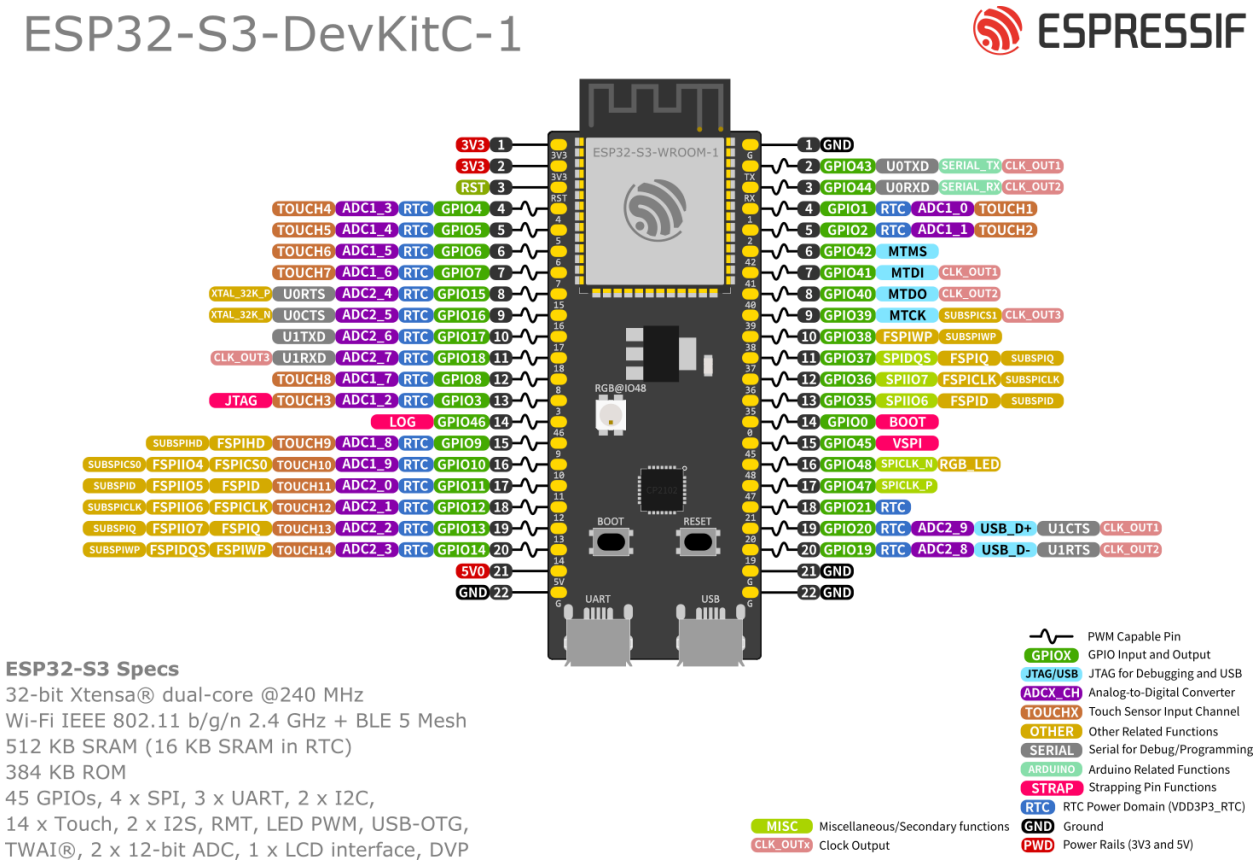

Figure 3.3. ESP32-S3-DevKitC-1 pin layout used to identify GPIO, power rails, and ADC-capable pins when wiring sensors for the prototype.

To support embedded inference, the ESP32 implementation treats the SoC as a **decision-making system**, not merely an inference engine. The firmware orchestrates sensing, window buffering, inference, and alert determination in a single loop. For maximum control and debuggability on constrained hardware, the inference code (`firmware/esp32/src/inference/nn_inference.*`) implements the full model graph (CNN/SE/Transformer/Projection + 3-layer

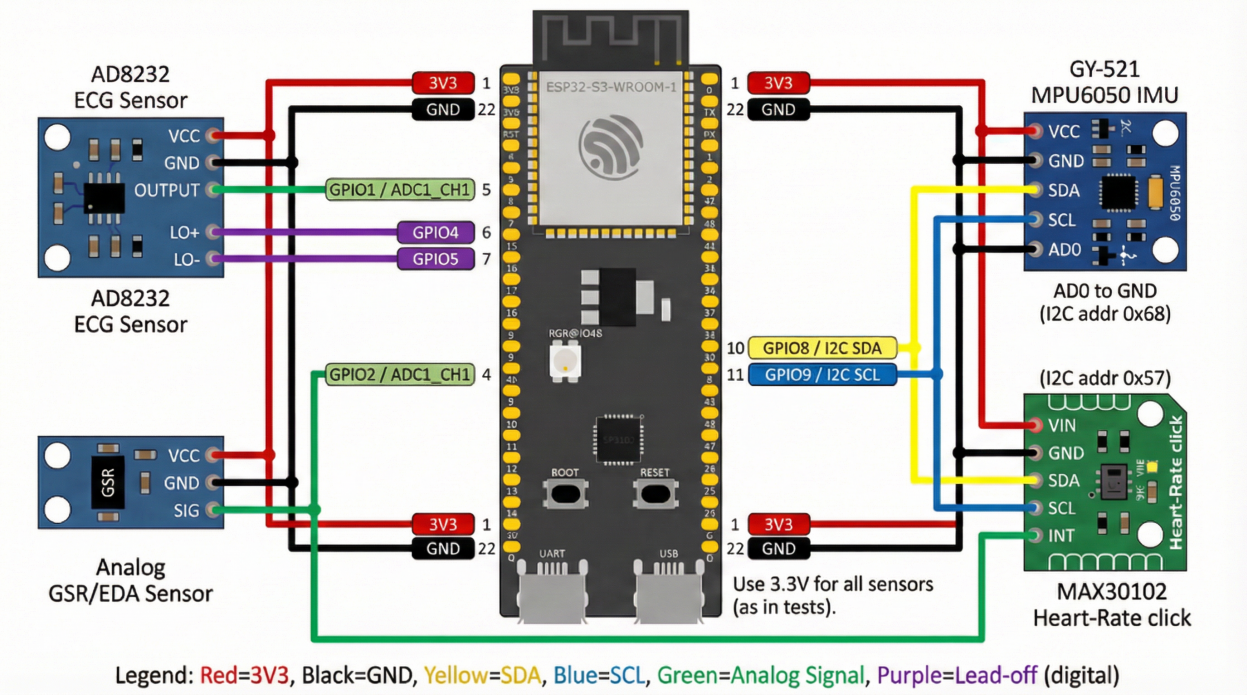

Figure 3.4. ESP32-S3 prototype wiring for AD8232 (ECG), GY-521 MPU6050 (IMU), MAX30102 (PPG), and Grove GSR v1.2 (EDA). Matches the firmware pin map (GSR on GPIO2; I2C on GPIO8/9).

task heads) directly in C++ without external ML runtimes, derived from the PyTorch architecture and exported weights. Memory usage is made predictable by fixed-window inference and explicit intermediate buffers, with optional allocation in PSRAM when available [1].

3.7.3 Signal conditioning, telemetry design, and live visualization

Bringing the prototype from initial power-on to clean, interpretable signals required resolving several firmware and host-side signal-processing issues. This subsection documents the signal-conditioning pipeline and the engineering fixes that produced reliable real-time visualization from the deployed sensor stack.

3.7.3.1 ECG IIR filter stability (notch above Nyquist)

The firmware applies a three-stage IIR filter to the raw AD8232 ECG ADC output: a 2nd-order Butterworth high-pass at 0.5 Hz (baseline wander removal), a 2nd-order biquad notch at 60 Hz (mains hum), and a 2nd-order Butterworth low-pass at 40 Hz. At the system's 100 Hz sampling rate, the Nyquist frequency is 50 Hz. Configuring a 60 Hz notch at 100 Hz sampling places the notch *above* the Nyquist limit, which causes the biquad denominator coefficient a_2 to exceed unity ($a_2 \approx 1.03$), placing poles outside the unit circle and rendering the filter unstable.

In practice, this instability manifested as the filtered ECG output oscillating between the 12-bit clamp limits (alternating 0 and 4095 every sample), producing a flat line on the plot despite valid raw ADC readings ($\sim$1500–2100 counts). The root cause was confirmed by reproducing the divergent output numerically in a `float32` biquad simulation with identical coefficients.

The fix adds a Nyquist guard to the biquad `setNotch()` method: if $f_0 \geq f_s/2$, the notch reverts to identity (passthrough). This ensures stable filter behavior regardless of the configured mains frequency and sampling rate. Additionally, a first-sample warmup was added: on the first real ADC reading, the filter's delay lines are primed with 500 constant-input iterations at the measured DC level, preventing start-up transients from saturating the output. The high-pass filter output (centered at zero) is re-mapped to the ADC midrange using a slow-tracking DC estimator ($\alpha = 0.001$), keeping the filtered value within the 12-bit range for downstream use by the inference engine.

3.7.3.2 Serial output unification (`Serial` vs. `CONSOLE`)

The ESP32-S3-DevKitC-1 uses a USB-Serial/JTAG interface where the user-facing serial endpoint is `USBSerial`, not the default `Serial` object. The firmware defines a `CONSOLE` macro that resolves to `USBSerial` on USB-JTAG boards. High-rate telemetry lines (`ECG,`, `PPG,`, `LIVE,`) correctly used `CONSOLE`, but the inference result output (the `INFER,` CSV line and the box-drawing status display) inadvertently used `Serial.print()`, sending output to a different USB endpoint. This caused the inference data to appear garbled or interleaved when captured on the host.

Replacing all `Serial`-based print calls with their `CONSOLE` equivalents in the inference output path resolved the issue, producing clean, parseable `INFER,` CSV lines on the same serial stream as the sensor telemetry.

3.7.3.3 Multi-rate CSV telemetry protocol

The firmware emits three concurrent comma-separated value (CSV) streams at different rates to balance signal fidelity with USB-serial bandwidth constraints (115 200 baud):

- **ECG stream** (100 Hz): `ECG,t_ms,adc_raw,adc_filtered,flags` — compact 5-field line providing both raw and filtered ADC for flexible host-side processing.
- **PPG stream** (100 Hz): `PPG,t_ms,red,ir,flags` — separate MAX30102 red ($\sim$660 nm wavelength, sensitive to deoxygenated hemoglobin) and infrared ($\sim$880 nm, sensitive to oxygenated hemoglobin) channels at full sample rate. The dual-wavelength measurement enables SpO_2 estimation via the red/IR ratio and provides richer pulse waveform features than single-channel PPG.
- **LIVE stream** (10 Hz): `LIVE,t_ms,ecg,red,ir,` `gsr_adc,gsr_us,ax,ay,az,gx,gy,gz,flags` — a composite summary line with all sensor channels, suitable for low-bandwidth monitoring.
- **INFER line** (per window, $\sim$0.1 Hz): `INFER,t_ms,act_cls,` `act_conf,str_cls,str_conf,arr_cls,arr_conf,` `moving,alert` — emitted after each 10-second inference window with classification results from all three task heads.

3.7.3.4 Host-side plotting and signal quality indicators

A Python plotting script (`live_serial_plot_ppg_gsr.py`) consumes the serial stream in a background thread and renders a four-panel real-time display using Matplotlib. The panels show:

1. **ECG** with bandpass filtering (high-pass 0.5 Hz, low-pass 35 Hz, optional mains notch), running-mean DC removal, and a railing detector that reports the percentage of samples at the ADC limits.

2. **PPG** (red and IR channels) with high-pass 0.3 Hz / low-pass 8 Hz filtering for pulse-band isolation, and an AC standard deviation metric for contact quality assessment.

3. **GSR** conductance in μS with auto-scaled y-axis.

4. **Inference results** with color-coded background (green for normal, yellow for warning, red for critical alert).

A key bug in the original PPG pipeline was that the filtered output was written back into the raw data buffer, corrupting the running-mean filter's input on subsequent samples. Separating raw and filtered PPG into distinct buffers resolved this issue.

3.7.3.5 Before and after: sensor bring-up progression

Figure 3.5 shows the system output before any sensors are physically connected: the ECG trace is flat (ADC railing detected), PPG reports "No finger," and GSR reads zero conductance. The inference panel still produces classification outputs because the model processes zero-filled channels (consistent with training-time missing-modality handling).

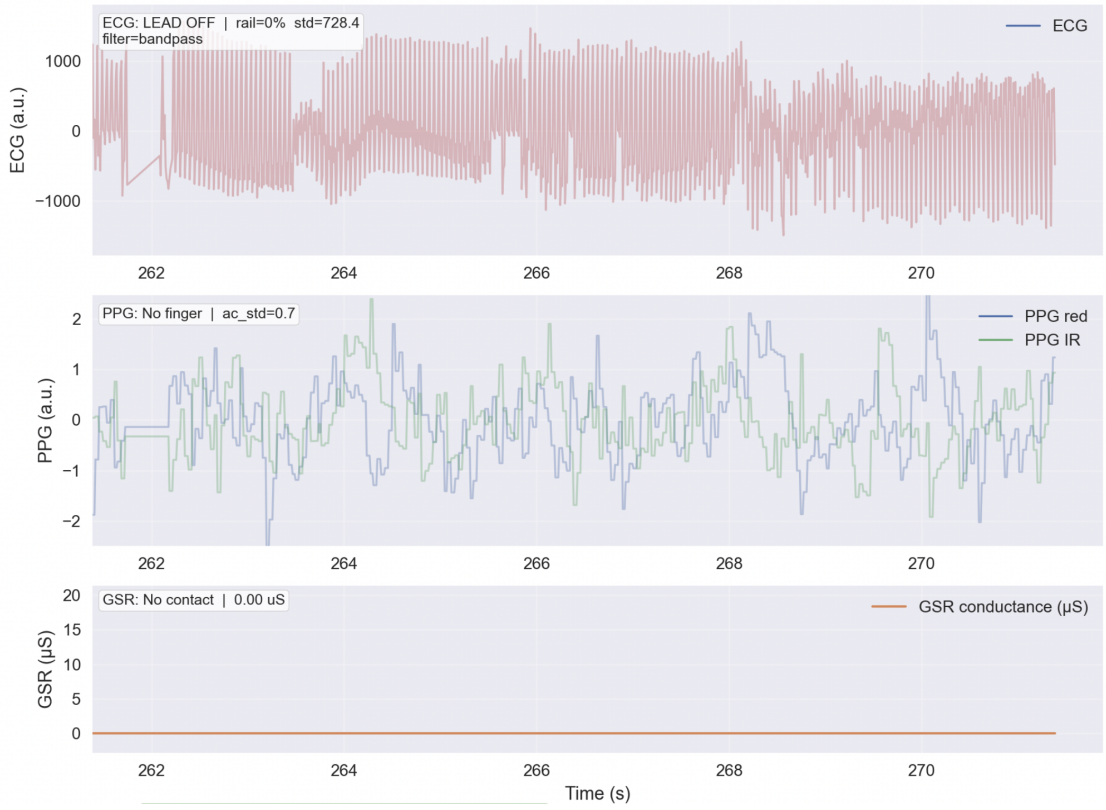

Figure 3.5. Live visualization with no sensors attached. ECG shows a flat line due to ADC railing (signal alternates between 0 and 4095), PPG reports "No finger" with ambient-level readings (∼125 counts), and GSR conductance is zero. The inference panel still produces outputs from the zero-filled input channels.

Figure 3.6 shows the system after all signal conditioning fixes and with sensors properly connected: the ECG displays a filtered waveform with visible QRS-like deflections (std $\approx$ 170), PPG shows "Good contact" with distinct red and IR channel responses (ac_std $\approx$ 13.6), and GSR displays real skin conductance variation in the 15–19 μS range. The inference panel reports per-window classification results with confidence scores.

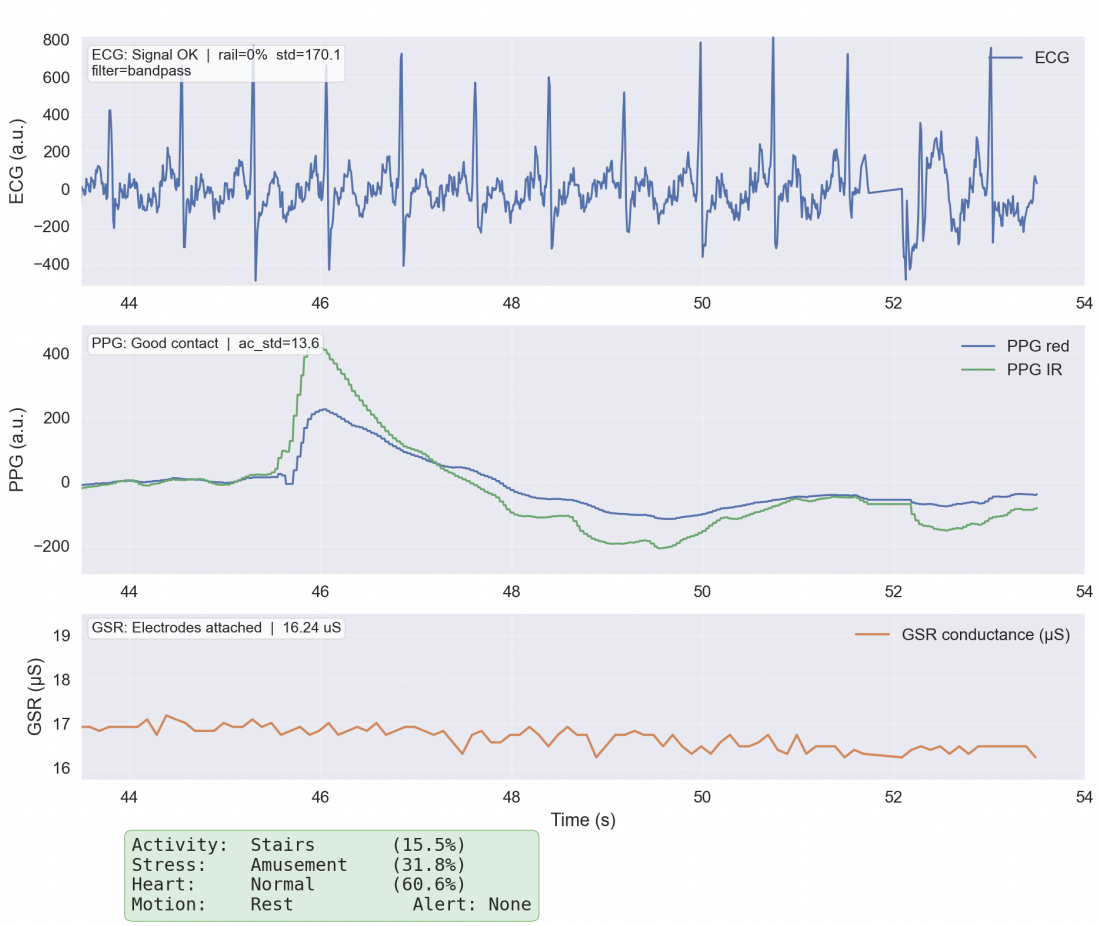

Figure 3.6. Live visualization with all sensors connected and signal conditioning active. ECG (top) shows bandpass-filtered signal with QRS deflections and 0% railing; PPG (middle) shows filtered red and IR channels with good finger contact; GSR (bottom) shows real-time skin conductance at ∼15–19 μS. The inference panel displays multi-task classification results updated every 10-second window.

The progression from Figure 3.5 to Figure 3.6 demonstrates the complete sensor-to-inference pipeline operating in real time on the ESP32-S3 prototype, validating both the firmware signal conditioning and the end-to-end data flow from ADC sampling through neural network inference to visual output.

3.7.4 Exported model footprint and ESP32 inference graph

For edge deployment, model weights are exported in a compact form suitable for microcontroller flash and runtime constraints:

- **Model weights exported**: 112,552 parameters ($\approx 450\,\mathrm{KB}$ assuming 32-bit floats).
- **Included components**: CNN feature extractor (5→32→64→64), SE attention, 2-layer Transformer encoder, feature projection, and three 3-layer task heads with LayerNorm (activity, stress, arrhythmia). The training model is deployed without compression or stripping.
- **BatchNorm folding**: BatchNorm parameters are folded into convolution weights for efficiency, reducing runtime operations and simplifying the inference graph.

Why ONNX appears in the pipeline. In addition to direct weight export, the repository includes an optional conversion workflow that uses ONNX as an intermediate format when targeting TensorFlow/TFLite. ONNX provides a portable graph representation for moving a PyTorch model into other runtimes; in practice, conversion can be brittle for transformer-style attention blocks. For this thesis, the primary and most reliable embedded pathway is therefore C header weight export plus a custom C++ inference graph, enabling predictable buffers and memory behavior on the ESP32-S3.

The resulting ESP32 inference graph follows a shape progression from the 5×1000 input through three convolutional stages with max-pooling to 64×125, SE channel attention, a 2-layer Transformer encoder ($d_{\mathrm{model}} = 64$, $n_{\mathrm{head}} = 4$), global average pooling to a 64-dimensional vector, and finally the three 3-layer task-specific classification heads with LayerNorm (Activity: 4 classes, Stress: 2 classes, Arrhythmia: 2 classes). The full training model is deployed with BatchNorm folded into convolution weights, implemented directly in C++ for deterministic embedded execution.

3.8 On-device adaptive training

While the preceding sections describe a system in which model weights are frozen after deployment, real-world wearable applications require models that can adapt to individual users, changing sensor conditions, and distributional drift without cloud connectivity [32]. This section presents a complete on-device adaptive training system implemented entirely in C++ on the ESP32-S3, enabling feedback-driven, resource-constrained, partial parameter updates on edge hardware.

3.8.0.1 Why on-device training: hardware and deployment constraints

The motivation for on-device training is fundamentally a **hardware and systems engineering** problem. Three categories of constraints make conventional training approaches infeasible for edge-deployed wearables:

GPU-based training is physically impossible at the edge. Modern training GPUs consume 150–350 W (e.g., NVIDIA A100: 300 W TDP per NVIDIA's product data sheet), cost $1,000–$40,000, and require active cooling infrastructure. Even mobile-class accelerators (NVIDIA Jetson series: 10–30 W, $200–$500 per NVIDIA's published module specifications) exceed the thermal, power, and cost budgets of battery-powered wearables. The ESP32-S3 used in this work operates at $\sim$0.5 W typical active-mode power [1] and costs under $10 in bill-of-materials.

Cloud-based retraining introduces unacceptable dependencies. Transmitting raw physiological data to cloud servers creates privacy risks (HIPAA, GDPR) and requires always-on network connectivity — which is unavailable in remote healthcare settings, disaster response, military operations, or simply when a patient moves out of Wi-Fi range. Cloud GPU compute costs (e.g., AWS p3.2xlarge at $3.06/hour or Google Cloud T4 at $0.95/hour as of 2025) create recurring expenses that scale with device fleet size, and server outages or API deprecations permanently disable adaptation for all deployed units. For a wearable health monitor that must function autonomously anywhere and anytime, cloud dependency is a single point of failure.

MCU/SoC/FPGA platforms cannot support full-model training. On-chip SRAM on typical edge SoCs is 512 KB–2 MB. Full backpropagation requires storing all intermediate activations (proportional to model depth $\times$ layer width $\times$ batch size) plus gradients and optimizer state, exceeding available memory by 10–100$\times$ for even modest models. Many microcontrollers lack hardware floating-point entirely, and no production-grade backpropagation framework exists for bare-metal firmware. Furthermore, safety-critical embedded systems prohibit dynamic memory allocation (`malloc`/`free`) during operation due to non-deterministic latency and heap fragmentation risks. These constraints collectively mean that while edge devices efficiently execute neural network *inference*, they cannot *train* neural networks using conventional methods.

The frozen-backbone, mutable-head design presented here directly addresses these constraints: by training only 4,984 classification head parameters (instead of all 112,552), the system reduces activation storage for backpropagation by $\sim$715$\times$ (from 496,494 bytes to 694 bytes; Section 3.8.4), fits gradient computation within a 2,000 ms wall-clock budget on a 240 MHz dual-core MCU, and uses only pre-allocated buffers with zero dynamic allocation — making the device a fully autonomous learning system that operates independently of GPUs, cloud infrastructure, and internet connectivity.

3.8.0.2 Comparison with prior work on on-device learning

Unlike prior work on on-device learning that targets larger platforms [10], [33] or employs external ML frameworks [9], this system performs backpropagation and SGD with momentum using only pre-allocated float32 buffers — no external libraries, no dynamic memory, and no GPU. The approach is most closely related to TinyTrain [11] and incremental TinyML [34], but extends these concepts with hardware-enforced safety mechanisms specifically designed for safety-critical biomedical applications.

Table 3.6 summarizes the key differences between this work and prior on-device learning systems.

The system comprises seven tightly integrated components (Figure 3.7):

Table 3.6. Comparison of on-device training systems. Only this work combines all six properties required for safety-critical autonomous edge deployment.

System	*No ext. framework*	*Zero `malloc`*	*Model versioning*	*A/B validation*	*Safety lockout*	*HW resource gates*
TinyTL [10]	No	No	No	No	No	No
TinyOL [9]	No	No	No	No	No	No
TinyTrain [11]	No	No	No	No	No	No
AIfES [12]	Yes	No	No	No	No	No
On-device (Lin) [33]	No	No	No	No	No	No
This work	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

1. **Gradient Engine** — backpropagation through classification heads
2. **Model Store** — dual-slot weight versioning with flash persistence
3. **Feedback Controller** — drift detection and user correction buffer
4. **Resource Guard** — hardware-enforced training windows
5. **Validation Engine** — candidate model safety verification
6. **Adaptive Trainer** — top-level orchestrator
7. **Training Configuration** — centralized parameter governance

3.8.1 Design philosophy: frozen backbone, mutable heads

A key design decision is to **freeze the entire backbone** (conv1–conv3, SE attention, projection with LayerNorm, and 2-layer Transformer) and train **only the classification head** parameters on-device. This strategy is motivated by three observations:

1. **Feature reuse**: convolutional layers learn general signal representations (frequency components, morphological patterns, temporal correlations) that transfer well across users and

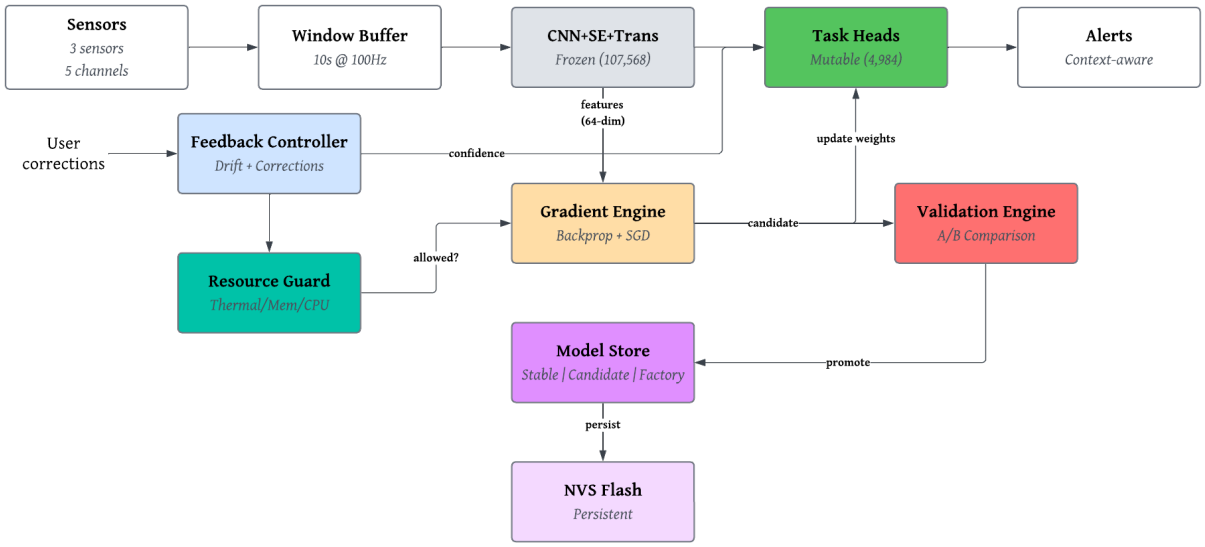

Figure 3.7. On-device adaptive training system architecture. The frozen backbone (CNN + SE attention + Transformer, gray) extracts 64-dimensional features that feed both the mutable classification heads (green) for inference and the Gradient Engine (orange) for on-device training. The Feedback Controller (blue) detects confidence drift and buffers user corrections. The Resource Guard (cyan) enforces hardware constraints. The Validation Engine (red) compares candidate vs. stable models before promotion. The Model Store (purple) manages dual-slot versioning with NVS flash persistence.

conditions. These features are expensive to learn and unlikely to improve with small amounts of on-device data.

2. **Compute reduction**: the frozen backbone contains 107,568 parameters (CNN + SE attention + Transformer layers) requiring no gradient computation. Only the 4,984 task-head parameters are updated, reducing the computational cost of training by $\sim$95% compared to full-model fine-tuning while keeping the gradient path short and numerically stable.

3. **Safety**: freezing the backbone prevents catastrophic forgetting of learned feature representations. Even if the head adaptation fails completely, the system can rollback to the original heads while retaining the full feature extraction capability.

Table 3.7 provides a detailed breakdown of the parameter partitioning.

Table 3.7. Parameter partitioning between frozen backbone (CNN + SE + Transformer) and trainable classification heads. †FC1 is frozen during on-device adaptation; only LN, FC2, and FC3 receive gradient updates. Total trainable = 4,984 parameters.

Component	Layer	Parameters	Trainable
CNN Backbone	Conv1 (5→32, k=7)	1,152	Frozen
	Conv2 (32→64, k=5)	10,304	Frozen
	Conv3 (64→64, k=3)	12,352	Frozen
SE Attention	FC1+FC2 (64→16→64)	1,104	Frozen
Projection + LayerNorm	Linear (64→64) + LN	4,288	Frozen
Transformer (×2)	Attn + FFN + LayerNorm	78,160	Frozen
Activity Head	FC1 (64→64)	4,160	Frozen†
	LN (64)	128	✓
	FC2 (64→32)	2,080	✓
	FC3 (32→4)	132	✓
Stress Head	FC1 (64→48)	3,120	Frozen†
	LN (48)	96	✓
	FC2 (48→24)	1,176	✓
	FC3 (24→2)	50	✓
Arrhythmia Head	FC1 (64→48)	3,120	Frozen†
	LN (48)	96	✓
	FC2 (48→24)	1,176	✓
	FC3 (24→2)	50	✓
Total frozen		**107,568**	
Total trainable		**4,984** (est.)	
Grand total		**112,552**	

3.8.2 Gradient engine: on-device backpropagation

The gradient engine implements backpropagation through the three-layer classification heads (FC1→LayerNorm→GELU→FC2→GELU→FC3) using the cross-entropy loss function. For a prediction $\hat{y} = \mathrm{softmax}(z)$ where z are the logits from FC3, and ground-truth class c, the loss is:

$$\mathcal{L} = -\log \hat{y}_c = -\log \frac{e^{z_c}}{\sum_j e^{z_j}} \tag{3.3}$$

The gradient with respect to the logits has the elegant closed form:

$$\frac{\partial \mathcal{L}}{\partial z_i} = \hat{y}_i - \mathbf{1}[i = c] \tag{3.4}$$

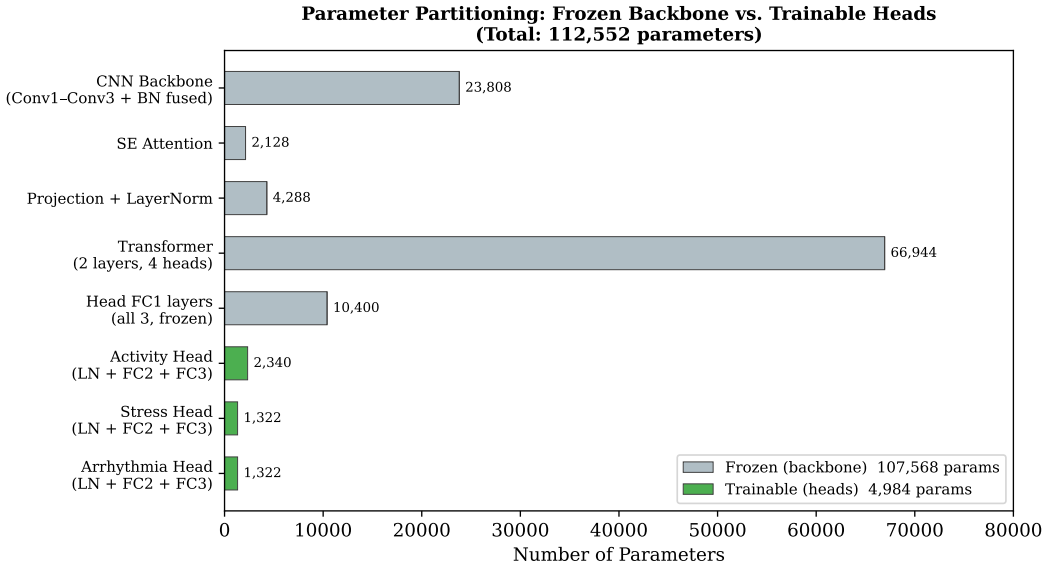

Figure 3.8. Parameter partitioning between the frozen backbone (CNN + SE attention + Transformer) and the trainable classification heads. The backbone (107,568 parameters) learns general signal representations during offline training and is frozen on-device. Only the 3-layer task heads (4,984 parameters) are updated through on-device adaptation, reducing compute cost by $\sim$95% while preserving learned feature representations.

This gradient is then propagated backwards through FC3 and FC2 using standard chain-rule operations (FC1 is frozen during on-device adaptation), all implemented in fixed-size, pre-allocated float32 buffers. Algorithm 2 summarizes the complete training step.

Key implementation features:

- **Zero runtime allocations**: All gradient, momentum, and activation cache buffers are pre-allocated in PSRAM during initialization. No `malloc`/`free` calls occur during training, guaranteeing deterministic latency.

- **Gradient clipping** (L2 norm ≤ 1.0): Prevents gradient explosion from noisy biomedical signals. The threshold of 1.0 follows standard practice for cross-entropy training with softmax outputs, where the maximum gradient magnitude is bounded by 1.0 per sample.

- **Weight magnitude clamping** ($|w| \leq 10.0$): A hard safety constraint that prevents weight divergence even under adversarial input. The bound of 10.0 is conservative: well-trained

Algorithm 2 On-Device Training Step

Require: Feature vector $\mathbf{x} \in \mathbb{R}^{64}$ from frozen CNN+SE+Transformer backbone
Require: True label c, learning rate η, momentum μ

$\mathbf{h}_1 \leftarrow \text{GELU}(\text{LN}(W_1\mathbf{x} + \mathbf{b}_1))$ {FC1 fwd (frozen, no gradients)}
$\mathbf{h}_2 \leftarrow \text{GELU}(W_2\mathbf{h}_1 + \mathbf{b}_2)$ {FC2 fwd, cache $\mathbf{h}_2$}
$\mathbf{z} \leftarrow W_3\mathbf{h}_2 + \mathbf{b}_3$ {FC3 fwd (logits)}
$\hat{\mathbf{y}} \leftarrow \text{softmax}(\mathbf{z})$ {Probabilities}
$\mathcal{L} \leftarrow -\log \hat{y}_c$ {Cross-entropy loss}
$\delta_z \leftarrow \hat{\mathbf{y}} - \text{onehot}(c)$ {Logit gradient}
$\nabla_{W_3} \leftarrow \delta_z \cdot \mathbf{h}_2^\top; \quad \nabla_{b_3} \leftarrow \delta_z$ {FC3 gradients}
$\delta_{h_2} \leftarrow W_3^\top \delta_z \odot \text{GELU}'(\mathbf{h}_2)$ {Backprop through GELU}
$\nabla_{W_2} \leftarrow \delta_{h_2} \cdot \mathbf{h}_1^\top; \quad \nabla_{b_2} \leftarrow \delta_{h_2}$ {FC2 gradients}
{Stop gradient at FC1 boundary (frozen)}
Clip $\|\nabla\|_2$ to τ {Gradient clipping ($\tau = 1.0$)}
$\mathbf{v} \leftarrow \mu \cdot \mathbf{v} + \nabla; \quad \theta \leftarrow \theta - \eta \cdot \mathbf{v}$ {SGD with momentum ($\mu = 0.9$)}
Clamp $|\theta_i| \leq \theta_{\max}$ {Weight magnitude bound ($\theta_{\max} = 10.0$)}

Ensure: Updated FC2/FC3 parameters θ, loss $\mathcal{L}$, gradient norm $\|\nabla\|$

classification heads typically have weights in the range $[-3, +3]$, so 10.0 provides a wide margin for adaptation while catching runaway divergence.

- **NaN/Inf guards**: Every forward pass, backward pass, and weight update includes explicit checks for numerical instability. If detected, the step is aborted and weights are rolled back.

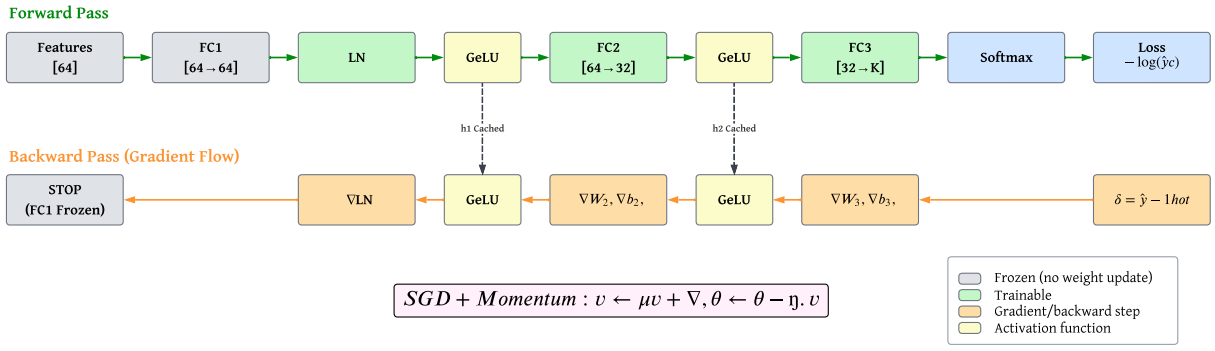

Figure 3.9. Gradient engine dataflow for a single task head. Top: Forward pass through FC1→LN→GELU→FC2→GELU→FC3→Softmax→Loss, with intermediate activations cached in pre-allocated buffers. Bottom: Backward pass computing gradients via the chain rule. Dashed arrows indicate cached activations reused during backpropagation. SGD with momentum is applied after gradient clipping.

3.8.3 Memory architecture for on-device training

Table 3.8 presents the complete memory budget for the adaptive training system. The large training-state buffers (mutable weights, momentum, gradients, and model store snapshots) are allocated in PSRAM, while small sample buffers remain in internal SRAM. All buffers are pre-allocated during initialization, with no dynamic allocation during training episodes.

Table 3.8. Memory allocation for on-device adaptive training on ESP32-S3 (8 MB PSRAM). Large training-state buffers are stored in PSRAM alongside inference buffers, while small correction/validation sample buffers use internal SRAM.

Buffer	**Size (bytes)**	**Location**
Inference (existing)		
Input normalized [5×1000]	20,000	PSRAM
Conv1 output [32×1000]	128,000	PSRAM
Conv2 output [64×500]	128,000	PSRAM
Conv3 output [64×250]	64,000	PSRAM
SE + Transformer + projected + heads	∼324,000	PSRAM
Training (new)		
Mutable weights (3 heads)	20,768	PSRAM
Momentum buffers (3 heads)	20,768	PSRAM
Gradient buffers (3 heads)	20,768	PSRAM
Activation caches (3 heads)	952	PSRAM
Model Store: stable slot	20,768	PSRAM
Model Store: candidate slot	20,768	PSRAM
Correction buffer (32 samples)	8,448	SRAM
Validation buffer (16 samples)	4,672	SRAM
Total inference	**∼703,844**	
Total training	**∼117,912**	
Grand total	**∼821,756**	
Remaining PSRAM	**∼7.3 MB**	

3.8.4 Memory reduction derivation: head-only vs. full backpropagation

The central claim of the frozen-backbone design is that it dramatically reduces the memory required for on-device training. Table 3.9 provides a byte-level derivation comparing full backpropagation (all 112,552 parameters trainable) against head-only training (4,984 parameters trainable). The key savings come from **activation storage**: full backpropagation must cache all intermediate

convolutional feature maps, SE attention states, and Transformer activations for the backward pass, whereas head-only training requires only the compact 64-dimensional projected feature vector and small hidden-layer activations.

Table 3.9. Activation storage for backpropagation: full model vs. head-only training (FP32, batch size 1).

Cached activation	**Full backprop (B)**	**Head-only (B)**
Input [5×1000]	20,000	—
Conv1 out [32×1000] (same padding, k=7)	128,000	—
Pool1 out [32×500] (MaxPool ×2)	64,000	—
Conv2 out [64×500] (same padding, k=5)	128,000	—
Pool2 out [64×250] (MaxPool ×2)	64,000	—
Conv3 out [64×250] (same padding, k=3)	64,000	—
SE attention [64+16+64]	576	—
GlobalAvgPool [64]	256	—
Projection + LayerNorm [64]	512	—
Transformer (×2) attn + FFN	26,456	—
Projected features [64]	256	256
3 head hidden layers [64+32, 48+24, 48+24]	384	384
3 head outputs [4+2+2]	54	54
Total activation storage	**496,494**	**694**
Reduction: 496,494 / 694 ≈ 715×		

The ∼715× reduction in activation storage is the dominant factor enabling on-device training within the ESP32-S3's memory envelope. The total training memory (including parameter gradients, momentum buffers, and model store snapshots) is further reduced from ∼2.6 MB (full model) to ∼112 KB (head-only), a ∼24× overall reduction.

3.8.5 Model versioning and flash persistence

The Model Store implements a dual-slot weight versioning system (Figure 3.10) that maintains two weight sets in PSRAM:

- **Stable slot**: The currently deployed, validated weights. These are the weights used for inference.
- **Candidate slot**: Weights being adapted by the gradient engine. These have not yet been validated.

After each training episode, the candidate is evaluated against the stable model on a held-out validation buffer. If the candidate meets the promotion criteria (Section 3.8.8), it atomically replaces the stable slot and is persisted to ESP32 NVS flash for survival across power cycles.

If the candidate fails validation, a **rollback** operation restores the stable weights to the gradient engine in $O(1)$ time (a single `memcpy` of the weight blob). The original factory weights from flash ROM (.inc files) can always be restored via a factory reset, providing a guaranteed safe baseline.

The NVS persistence uses a single blob key ("weights", $\sim$21 KB) plus a generation counter, fitting comfortably within the ESP32 NVS partition. Each successful promotion increments the generation counter, creating an auditable adaptation history.

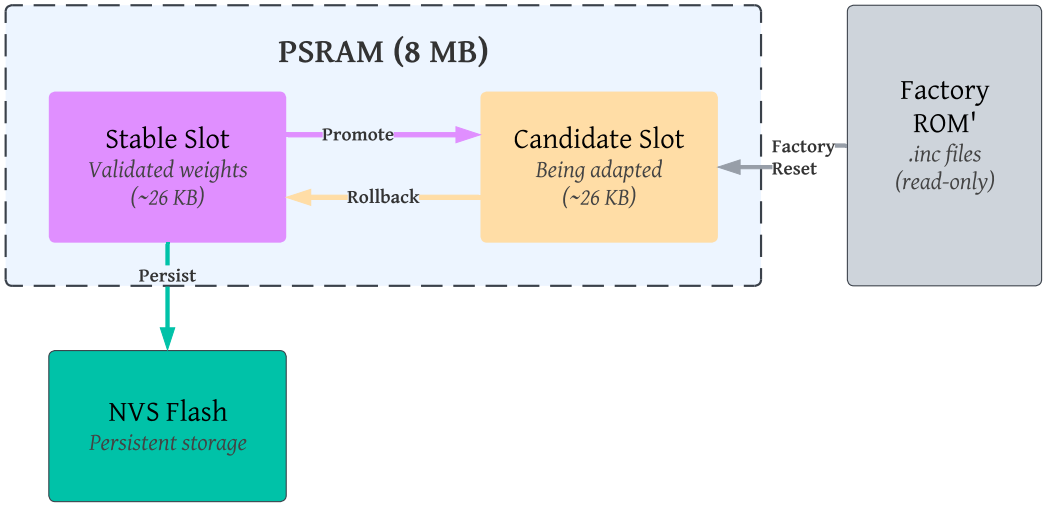

Figure 3.10. Dual-slot model versioning architecture. The stable slot holds the last validated weights used for inference. The candidate slot holds weights being adapted by the gradient engine. After each training episode, the candidate is evaluated against the stable model on a held-out validation buffer. If the candidate passes all promotion criteria, it atomically replaces the stable slot and is persisted to NVS flash. If it fails, a rollback restores the stable weights. Factory reset restores the original ROM weights from `.inc` flash arrays.

3.8.6 Feedback-driven training triggers

The Feedback Controller determines *when* the system should adapt. Three trigger mechanisms are implemented:

3.8.6.1 Confidence drift detection

An **Exponential Moving Average (EMA)** of prediction confidence is tracked independently for each task head. The EMA is a weighted running average that gives more weight to recent observations while smoothing out short-term noise:

$$\mathrm{EMA}_t = \alpha \cdot c_t + (1 - \alpha) \cdot \mathrm{EMA}_{t-1} \tag{3.5}$$

where c_t is the confidence (maximum softmax output probability $\max_i \hat{y}_i$) at inference window t, $\alpha = 0.1$ is the smoothing factor (effective averaging window of $\sim$10 windows $\approx$ 100 seconds), and EMA_{t-1} is the previous EMA value.

Threshold selection. When the EMA drops below a threshold $\theta = 0.45$ for three or more consecutive windows, the system detects distributional drift and triggers adaptation — provided labeled corrections are available in the buffer. The threshold value of 0.45 was chosen empirically: for a 4-class activity task (random chance = 25%), a confidence of 0.45 indicates above-chance but significantly degraded performance compared to typical operating confidence of 0.7–0.9. The consecutive-window requirement (3 windows = 30 seconds) prevents triggering on transient noise while remaining responsive to sustained drift.

This mechanism detects when the model's predictions become uncertain due to user-specific physiological patterns, changed sensor placement, or environmental conditions — all without requiring explicit labeled data for the detection itself.

3.8.6.2 User correction protocol

A serial command interface allows a clinician or user to provide ground-truth labels for the most recent prediction:

```
CORRECT <task_id> <label>
CORRECT 0 3    # Activity was Driving (class 3)
CORRECT 1 1    # Stress state was Stress (class 1)
CORRECT 2 0    # Heart rhythm was Normal (class 0)
```

Each correction is stored in a 32-entry ring buffer along with the 64-dimensional feature vector from the corresponding inference window. These (feature, label) pairs serve as supervised training samples for the gradient engine.

3.8.6.3 Periodic adaptation

Even without explicit drift detection, the system consumes accumulated corrections on a configurable schedule (default: every 60 inference windows, approximately 10 minutes). This ensures that corrections provided asynchronously are eventually used for training.

3.8.7 Resource-constrained training windows

The Resource Guard enforces hardware constraints before allowing a training episode to proceed (see the resource-check diamond in Figure 3.13). All five checks must pass simultaneously:

Table 3.10. Resource gating criteria for on-device training. All conditions must be satisfied before a training episode is permitted.

Resource	Condition	Threshold
PSRAM	Free bytes $\geq$ minimum	2 MB
Heap	Free bytes $\geq$ minimum	50 KB
Temperature	SoC temp $<$ maximum	65°C
Cooldown	Time since last train $\geq$ minimum	30 s
CPU load	Last inference latency $\leq$ maximum	3,000 ms

Threshold rationale:

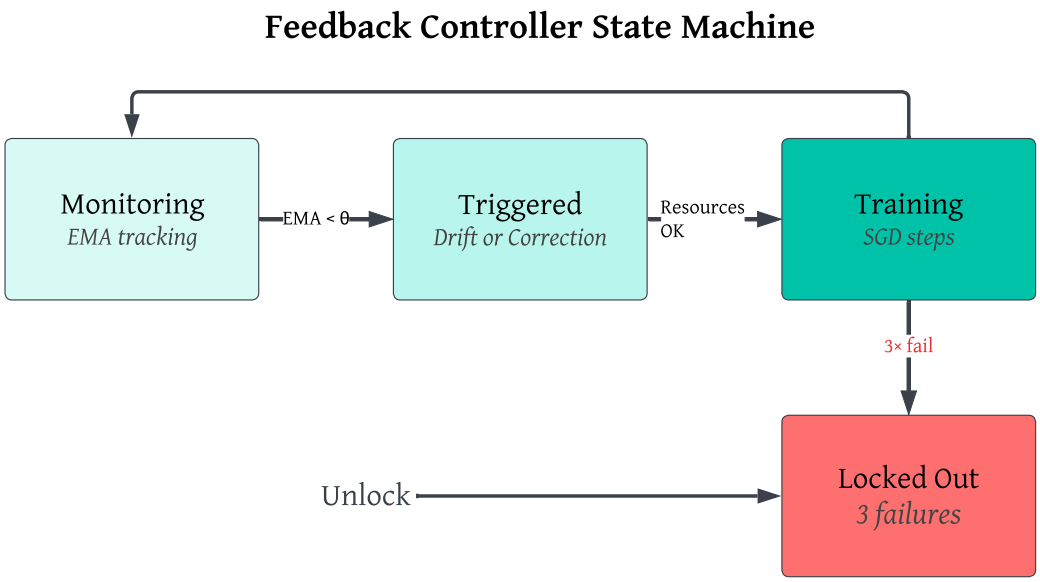

Figure 3.11. Feedback controller state machine. The system transitions between three states: Monitoring (tracking confidence EMA), Triggered (adaptation conditions met), and Training (executing a bounded training episode). User corrections are buffered asynchronously and consumed when the system enters the Training state. Confidence drift detection, periodic scheduling, and manual commands each provide independent trigger paths.

- **PSRAM $\geq$ 2 MB** (25% of the 8 MB total): ensures headroom for FreeRTOS allocations, interrupt-driven sensor buffers, and any future sensor expansion.

- **Heap $\geq$ 50 KB**: preserves space for stack frames, interrupt handlers, and FreeRTOS task control blocks that cannot be placed in PSRAM.

- **Temperature $<$ 65°C**: derived from the ESP32-S3 operating temperature specification of -40 to $+85$°C [1]; 65°C provides a 20°C safety margin below the absolute maximum, preventing thermal throttling during sustained training. The thermal guard uses the ESP32-S3 internal temperature sensor (with API-version-adaptive code supporting both ESP-IDF v4.4 and v5.x).

- **Cooldown $\geq$ 30 s**: prevents runaway adaptation loops by ensuring the system has completed at least three inference windows between training episodes.

- **Latency $\leq$ 3,000 ms**: guards against running training when inference latency is at or above its normal ceiling (normal inference $\approx$2,338 ms); defers training when the system is already under load.

Additionally, each training episode has a **wall-clock budget** of 2,000 ms. If the budget is exceeded mid-episode, training stops early with whatever steps have completed.

3.8.8 Validation and safety mechanisms

The Validation Engine is the safety gatekeeper that prevents on-device adaptation from degrading model performance — a critical requirement for medical applications.

3.8.8.1 A/B accuracy comparison

After each training episode, the candidate model and the stable model are both evaluated on the validation buffer (up to 16 recent labeled samples). The candidate is promoted only if:

$$\mathrm{acc}_{\mathrm{candidate}}(t) \geq \mathrm{acc}_{\mathrm{stable}}(t) - \epsilon \quad \forall\, t \in \{0, 1, 2\} \tag{3.6}$$

where $\epsilon = 1.0$ (1 percentage point tolerance) and t indexes the three tasks. The 1 percentage point tolerance margin allows candidates that are marginally worse on one head to be promoted if they improve overall, preventing overly conservative gating that would reject all candidates in practice. Hardware validation confirmed this threshold operates correctly: six training episodes with small correction sets (1–5 samples) were all correctly rolled back, demonstrating that the gate prevents premature promotion from insufficient data.

3.8.8.2 Weight divergence detection

Before accuracy evaluation, all candidate weights are scanned for values exceeding a divergence threshold ($|w| > 50.0$). This fast $O(n)$ check catches numerical instability before it affects inference. The threshold of 50.0 is set at 5$\times$ the weight clamp limit (10.0), providing an early warning for weights that have diverged well beyond normal operating range even after per-step clamping.

3.8.8.3 Safety lockout

If five consecutive candidate models fail validation, the system enters a **safety lockout** state that disables all adaptation. The five-failure threshold was chosen as a practical compromise: occasional failures are expected during normal exploration (the candidate may legitimately be slightly worse on a small validation set), but five consecutive failures indicate a systematic problem (e.g., corrupted corrections or fundamental distributional mismatch). Hardware testing confirmed this mechanism: lockout activated correctly after five rollbacks and was successfully cleared via the `UNLOCK` serial command. The lockout can be cleared via a serial command (`UNLOCK`) or a factory reset.

Table 3.11 summarizes all safety mechanisms. Figure 3.12 visualizes the concentric defense layers that protect the deployed model from degradation.

Table 3.11. Nine safety mechanisms in the on-device adaptive training system, listed in execution order. Each mechanism provides an independent defense layer. Collectively, these mechanisms are designed to prevent adaptation from degrading model performance below the pre-deployment baseline.

#	Mechanism	When	Condition	Action
1	Resource Guard	Before training	PSRAM $\geq$ 2 MB Heap $\geq$ 50 KB Temp $<$ 65°C Cooldown $\geq$ 30 s Latency $\leq$ 3,000 ms	Defer
2	Gradient Clipping	Backprop	$\|\nabla\|_2 > 1.0$	Scale to 1.0
3	Weight Clamping	SGD step	$\|w_i\| > 10.0$	Clamp ± 10.0
4	NaN/Inf Guards	Every op	NaN or Inf	Abort, rollback
5	Divergence Scan	Pre-valid.	Any $\|w\| > 50.0$	Reject candidate
6	A/B Validation	Post-train	$\text{acc}_\text{c} < \text{acc}_\text{s} - 1.0$	Rollback
7	Safety Lockout	Post-reject	5 consec. fails	Disable training
8	Wall-Clock Budget	Mid-episode	$>$ 2,000 ms	Stop early
9	Factory Reset	On command	Manual or corrupt	Restore ROM weights

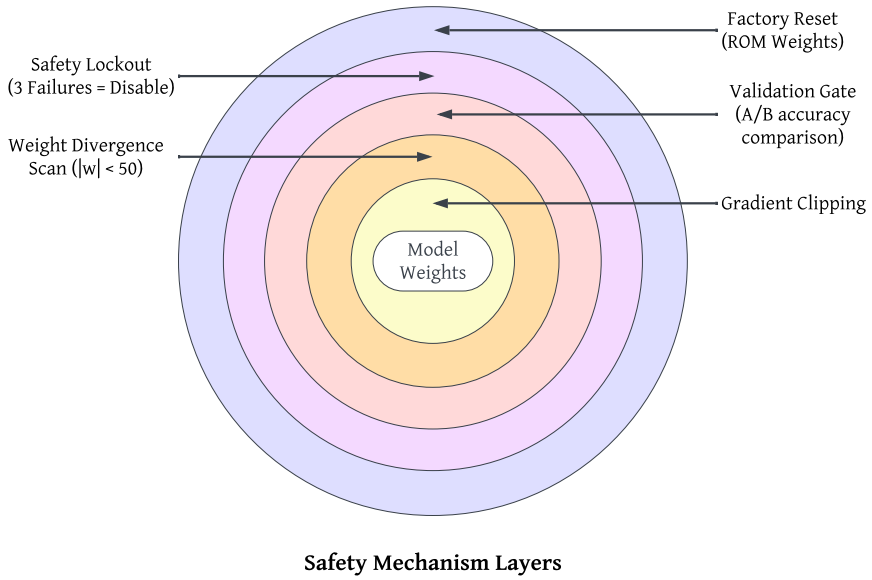

Figure 3.12. Concentric safety layers surrounding the deployed model. Each layer provides an independent defense mechanism: the outermost layer (resource gating) prevents training under adverse hardware conditions; gradient clipping and weight clamping bound parameter updates; the validation gate ensures accuracy is preserved; and the safety lockout provides a last-resort mechanism that disables adaptation after repeated failures. Factory reset can always restore the original ROM weights.

3.8.9 Complete adaptation cycle

Figure 3.13 illustrates the complete adaptation cycle that executes within the main firmware loop (a larger version of this flowchart is reproduced in Appendix B). Algorithm 3 provides the pseudocode, and Section 3.8.10 provides a detailed second-by-second timing breakdown. Annotated source code excerpts are listed in Appendix A.

3.8.10 Detailed timing breakdown of the adaptation cycle

The adaptation cycle spans one complete 10-second inference window and the subsequent processing phase. Table 3.12 summarizes the four phases; each is detailed below.

Algorithm 3 Complete On-Device Adaptation Cycle

Require: Inference result R, CNN features $\mathbf{f} \in \mathbb{R}^{64}$, inference latency t_{infer}

Cache features $\mathbf{f}$ for potential user corrections
Update confidence drift EMA for all tasks using R
$(trigger, reason) \leftarrow$ CHECKTRIGGERS()
if $trigger = \text{false}$ **or** locked out **then**
  return {No adaptation needed}
end if
if not RESOURCEGUARD.CANTRAIN(t_{infer}) **then**
  return {Defer — resources insufficient}
end if
Snapshot current weights $\rightarrow$ candidate slot
for each correction sample in buffer (up to N steps) **do**
  if wall-clock budget exceeded **then**
    break
  end if
  $(loss, \|\nabla\|) \leftarrow$ GRADIENTENGINE.TRAINSTEP(sample)
end for
$V \leftarrow$ VALIDATIONENGINE.VALIDATE()
if $V.\text{promoted}$ **then**
  MODELSTORE.PROMOTE() {Candidate $\rightarrow$ stable, persist to NVS}
else
  MODELSTORE.ROLLBACK() {Discard candidate, restore stable}
end if
Decay learning rate: $\eta \leftarrow \max(\eta \cdot 0.95,\ \eta_{\min})$

Table 3.12. Timing summary for one complete inference-and-adaptation cycle.

Phase	Time Window	Duration
1. Sensor acquisition (100 Hz, 3 sensors)	0–10 s	10,000 ms
2. Inference (CNN+SE+Transformer + 3 heads)	10.0–12.3 s	$\sim$2,338 ms
3. Adaptation check + training (if triggered)	12.3–14.3 s	$\leq$2,000 ms
4. Idle / next-window sampling resumes	14.3–20.0 s	$\sim$5,700 ms

3.8.10.1 Phase 1: Sensor acquisition (seconds 0–10)

The three sensors sample continuously at 100 Hz, filling the 5-channel $\times$ 1000-sample input buffer:

- **ECG** (AD8232): 1 channel — raw cardiac rhythm
- **PPG** (MAX30102): 1 channel (infrared)
- **IMU** (MPU6050): 3 channels (3-axis accelerometer)

By second 10.0, the normalized 5×1000 tensor is ready for inference.

3.8.10.2 Phase 2: Inference (seconds 10.0–12.3)

Inference executes in 2,338 ms and consists of four substeps:

1. **Backbone forward pass**: Conv1 $\rightarrow$ Conv2 $\rightarrow$ Conv3 $\rightarrow$ SE attention $\rightarrow$ global average pooling $\rightarrow$ linear projection $\rightarrow$ LayerNorm $\rightarrow$ 2-layer Transformer extracts a 64-dimensional feature vector.

2. **Classification head predictions**: Three task heads (Activity/4-class, Stress/binary, Arrhythmia/binary) produce softmax probability distributions from the shared features.

3. **Feature caching**: The 64-dim backbone output is cached for potential use as a training sample if a user correction arrives.

4. **Confidence EMA update**: The maximum softmax probability from each head is fed into the per-task EMA tracker (Eq. 3.5).

3.8.10.3 Phase 3: Adaptation check (seconds 12.3–12.8)

Immediately after inference, the system evaluates whether adaptation should occur, following the decision tree in Figure 3.13:

1. **Trigger evaluation** — any of the following activates adaptation:

   - Confidence drift detected (EMA < 0.45 for 3 consecutive windows)
   - User corrections present in the 32-entry ring buffer
   - Periodic timer elapsed (60 windows $\approx$ 10 minutes)

2. **Early exit conditions** — if any of the following apply, the adaptation phase is skipped entirely and the system proceeds directly to Phase 4 (next-window sampling), preserving the current stable model unchanged:

- No triggers are active (no drift, no corrections, periodic timer not elapsed)
- System is in safety lockout (5 consecutive validation failures; requires `UNLOCK` or factory reset to resume adaptation)

3. **Resource Guard** — all five hardware conditions must pass:
   - PSRAM free ≥ 2 MB
   - Heap free ≥ 50 KB
   - SoC temperature $< 65°$C
   - Cooldown since last training ≥ 30 s
   - Last inference latency $\leq 3{,}000$ ms

   If any condition fails, training is deferred to the next window.

4. **Snapshot**: Stable weights are copied to the candidate slot via `memcpy`.

5. **Training episode** (up to 2,000 ms wall-clock budget):
   - For each sample in the correction buffer, for each task head with corrections:
     - Forward pass $\rightarrow$ cross-entropy loss
     - Backward pass $\rightarrow$ gradients via chain rule
     - Gradient clipping ($\|\nabla\|_2 \leq 1.0$)
     - SGD with momentum update ($\mu = 0.9$, $\eta = 0.05$, decaying to minimum $\eta_{\min} = 0.005$)
     - Weight clamping ($|w| \leq 10.0$)
     - NaN/Inf check — abort entire step and rollback if detected
   - If wall-clock budget exceeded $\rightarrow$ break early

6. **Validation**:

- Weight divergence scan: reject candidate if any $|w| > 50.0$
- A/B accuracy comparison on up to 16 held-out validation samples across all three task heads

7. **Promotion decision**:
   - **Promoted**: candidate $\rightarrow$ stable slot, persist to NVS flash, increment generation counter
   - **Rejected**: restore stable weights via rollback, increment consecutive-failure counter
   - If 5 consecutive failures: enter **safety lockout**

3.8.10.4 Phase 4: Return to sampling (seconds 14.3–20.0)

After the adaptation check completes (whether or not training occurred), the system returns to sampling the next 10-second window. The entire adaptation phase fits within the $\sim$7.7-second gap between inference completion and the next window boundary, with the training episode itself hard-limited to 2,000 ms.

3.8.11 Telemetry and serial command interface

The adaptive training system emits machine-parseable CSV telemetry for each training event:

```
TRAIN,<t_ms>,<episode>,<reason>,<loss>,
      <lr>,<duration_ms>,<status>
ADAPT,<t_ms>,<generation>,<action>
```

Table 3.13 lists the complete serial command interface.

Table 3.13. Serial command interface for the on-device adaptive training system.

Command	Description
`CORRECT <task> <label>`	Provide true label for last prediction
`TRAIN`	Force a training episode
`RESET`	Factory reset all adapted weights
`STATUS`	Print comprehensive adaptation status
`UNLOCK`	Clear safety lockout

3.8.12 Hardware profiling results

Table 3.14 presents measured runtime performance from 70+ consecutive inference windows on the ESP32-S3 hardware with all three sensors active and the adaptive training system enabled.

Table 3.14. Measured runtime performance of the complete system on ESP32-S3-N8R8 with adaptive training enabled. Values are averaged over 70+ consecutive inference windows with all three sensors (ECG, PPG, IMU) active.

Metric	Measured Value
Inference timing	
CNN+SE+Transformer + heads compute	$\sim$2,334 ms
Post-processing (softmax + alerting)	$\sim$4 ms
Total per-window processing	**2,338 ms**
Window fill time (10 s nominal)	10,000 ms
Duty cycle (compute / window)	23.4%
Sensor acquisition	
Sample period (mean)	1,722 μs
Sample period (max)	$\sim$6,900 μs
Sampling rate (effective)	$\sim$100 Hz
Memory (with adaptive training)	
PSRAM free / min watermark	$\sim$7.3 MB
Flash	
Sketch size (with training)	767,936 B

The 2,338 ms inference latency per 10-second window represents a 23.4% duty cycle, leaving 76.6% of each window available for sensor acquisition, adaptive training episodes, and idle sleep. The PSRAM remaining of $\sim$7.3 MB (out of 8 MB) confirms that the combined inference and training allocations ($\sim$802 KB: 703,844 + 117,912 bytes) leave ample headroom for the resource guard's 2 MB minimum threshold.

3.8.13 Build impact and resource utilization

Table 3.15 summarizes the firmware resource impact of the adaptive training system.

Table 3.15. Firmware build metrics before and after adding the on-device adaptive training system. The training system adds approximately 8 KB of flash.

Metric	Before	After	Delta
Flash usage (bytes)	759,700	767,936	+8,236
Source files	12	19	+7
Lines of code (training)	0	∼1,200	+1,200

The negligible resource overhead confirms that the frozen-backbone, head-only design is well-matched to the ESP32-S3 platform: the entire adaptive training system fits within the existing resource envelope without requiring hardware upgrades, memory expansion, or compromises to the real-time inference pipeline. Measured hardware results from end-to-end training episodes are reported in Section 4.2.

3.9 Reproducibility: end-to-end pipeline

The repository includes a simplified end-to-end pipeline:

```
# Run the full 5-step pipeline
python simple_pipeline.py
```

Key verification commands used throughout development include:

```
# Train (max epochs; early stopping selects best checkpoint)
python -u -c "from train_model import ComprehensiveTrainer;
 ↪  t=ComprehensiveTrainer(); t.train_comprehensive_model(epochs=25,
 ↪  learning_rate=1e-3, batch_size=64)"

# Generate constructed 4-case test set (v4)
python scripts/create_realistic_test_samples.py --profile realistic
 ↪  --output synthetic_test_samples_4_cases_1000_v4.pkl
```

```
# Evaluate alert correctness and per-task accuracies
python test_accuracy_4_cases.py --model_path training_results/model.pth
↪ --samples_path synthetic_test_samples_4_cases_1000_v4.pkl
```

Key saved artifacts used for thesis reporting include:

- `training_results/model.pth` (best checkpoint)
- `training_results/subject_splits.json` (subject-wise train/val/test split metadata)
- `training_log.txt` (human-readable training log, including early stopping behavior)
- `comprehensive_training_results.json` (per-epoch metrics)
- `accuracy_results_4_cases.json` (per-case evaluation metrics used in Results tables)
- `training_comparison_15vs30_epochs.txt` (overfitting comparison motivating early stopping)

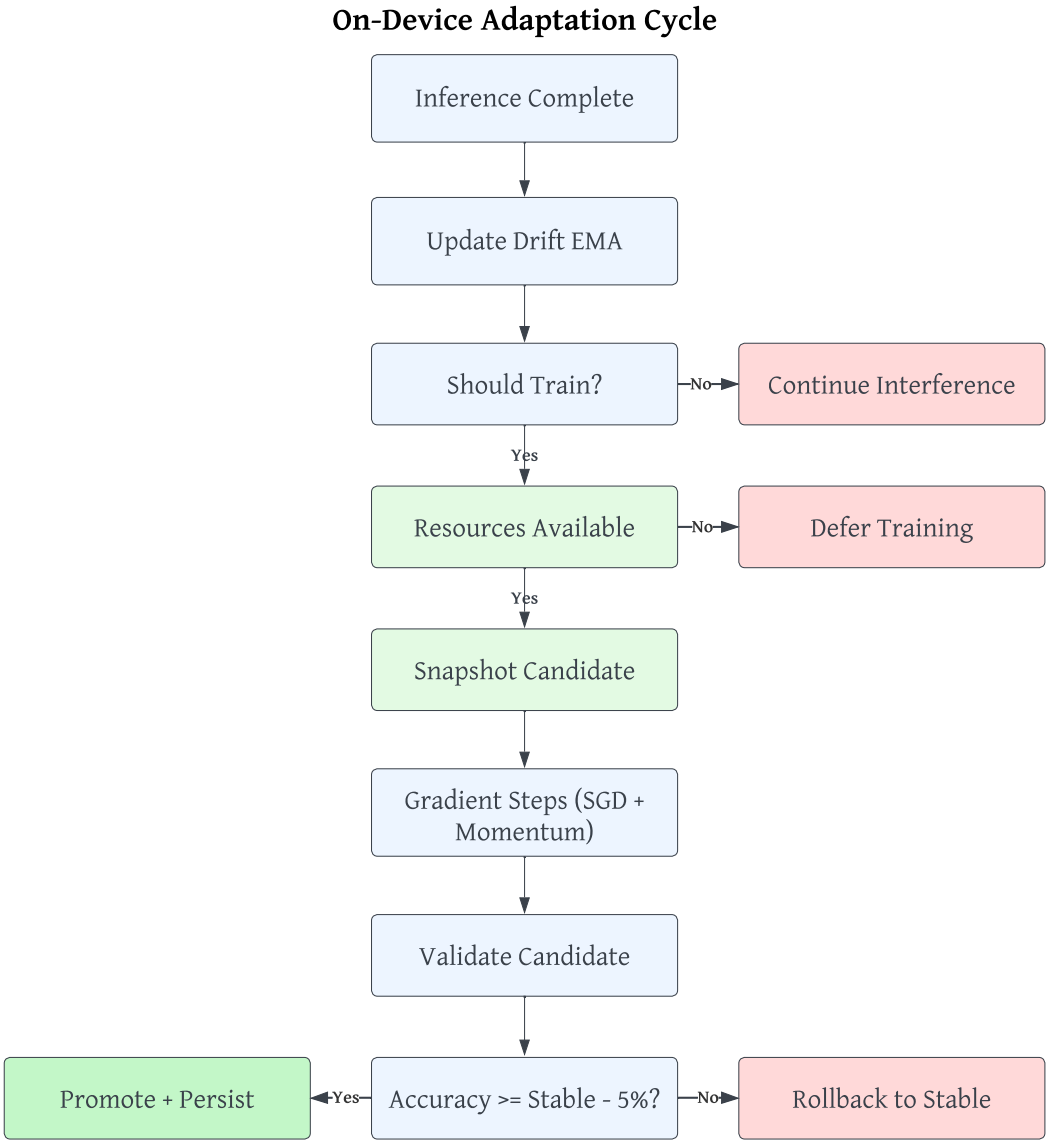

Figure 3.13. Decision flowchart for the complete on-device adaptation cycle. After each inference window, the system checks for adaptation triggers (confidence drift, user corrections, periodic schedule), verifies resource availability (memory, thermal, timing, CPU), executes a bounded training episode, validates the candidate against the stable model, and either promotes or rolls back. The entire cycle completes within the wall-clock budget (2,000 ms) and all decision paths converge to a safe state.

Chapter 4

Results

4.1 Evaluation protocol and reporting

This chapter reports results from the final end-to-end evaluation used to verify that the training, testing, and alerting pipeline behaves correctly under the four defined clinical scenarios (Cases 1–4; Section 3.6). The primary goal of this evaluation is **pipeline correctness and robustness**: (i) the model should detect stress and arrhythmia reliably, (ii) the alerting logic should be correct in each case, and (iii) the system should avoid cascading errors by deriving motion context directly from accelerometer statistics (Section 3.6.3).

All reported results below were produced by running the evaluation script on a fixed trained checkpoint and a fixed synthetic/constructed test set. To ensure reproducibility, evaluation outputs are saved both as a canonical JSON file at the repository root and as timestamped snapshots under `docs/run_logs/`:

- **Checkpoint evaluated**: `training_results/model.pth`
- **Test set**: `synthetic_test_samples_4_cases_1000_v4.pkl` (250 samples per case generated; evaluator reports on 100 per case; 400 total)
- **Evaluator**: `test_accuracy_4_cases.py`
- **Saved metrics (canonical)**: `accuracy_results_4_cases.json` (JSON snapshot of per-case metrics)
- **Saved metrics (timestamped example)**: `docs/run_logs/accuracy_results_4_cases_2026-01-28_21-40-09.json`

4.1.1 Four-case task accuracies and alert correctness

Table 4.1 summarizes task performance and alert correctness for each scenario. Alert thresholds are calibrated from score distributions: the arrhythmia threshold τ_a=0.70 separates real arrhythmia (median P(arr)=0.93 on MIT-BIH) from cross-dataset false activations on WESAD data (median 0.51); the stress threshold τ_s=0.35 balances stress recall ($\sim$96%) against specificity ($\sim$84%). Case 2 expects `no_alert`; alert accuracy measures correct suppression.

Table 4.1. Four-case alert benchmark on real dataset windows (100 samples per case, calibrated thresholds τ_a=0.70, τ_s=0.35). Stress cases use WESAD windows; arrhythmia cases use MIT-BIH windows. Task accuracy is reported only where ground-truth labels exist.

Case	Stress Acc.	Arrhy. Acc.	Alert Acc.	Alert Gen.
Case 1: Stress + Sedentary	99%	—	**96%**	100%
Case 2: Stress + Exercise	89%	—	**97%**†	3%
Case 3: Arrhythmia + Sedentary	—	87%	**82%**	99%
Case 4: Arrhythmia + Motion	—	93%	**88%**	88%

†For Case 2, `no_alert` is expected; alert accuracy measures correct suppression.

4.1.2 Alert benchmark analysis

Stress alerting (Case 1) achieves 96% alert accuracy. Motion gating (Case 2) achieves 97% correct alert suppression during exercise, with only 3% false-alert rate, validating the core design principle that physiological signals elevated by exercise should not trigger alerts. The arrhythmia alerting pathway achieves 82% (Case 3) and 88% (Case 4) alert accuracy; the motion gate correctly escalates to `critical` under high-motion conditions (Case 4). The calibrated arrhythmia threshold (τ_a=0.70) reduces cross-dataset false arrhythmia activations on WESAD data from 54% (at τ_a=0.5) to 5%, preventing spurious arrhythmia overrides of correct stress alerts.

4.1.3 Held-out test set evaluation (real data)

To validate the model beyond the alert benchmark, comprehensive evaluation was performed on the held-out test set from 13 unseen subjects (10,134 windows, subject-wise split to prevent data leakage). Table 4.2 reports per-task performance with standard metrics.

Table 4.2. V5 multi-task classification results on held-out test set (13 subjects, 10,134 windows). *F1-weighted, †F1-macro. AUC-ROC: Activity 0.731, Stress 0.975, Arrhythmia 0.879.

Task	Samples	Acc.	Prec.	Rec.	F1
Activity (4 cls)	5,251	94.1%	95.3%*	—	25.9%†
Stress (2 cls)	1,283	87.1%	88.1%*	81.9%†	81.9%†
Arrhythmia (2 cls)	3,600	90.8%	91.0%*	64.6%†	64.6%†

Table 4.3. Arrhythmia detection clinical metrics on held-out test set.

Metric	Value
Sensitivity	36.0%
Specificity	94.7%
AUC-ROC	0.879 [0.862, 0.894]
PPV	32.6%
NPV	95.4%

At the default threshold (0.50), the model prioritizes specificity (94.7%) over sensitivity (36.0%), appropriate for low-false-alarm screening. The AUC-ROC of 0.879 [0.862, 0.894] demonstrates strong discriminative ability: at the Youden-optimal threshold (0.11), recall increases to 87.4% while maintaining 77.6% specificity—comparable to single-task cloud models but running entirely on-device. The threshold is a deployment-time configuration parameter: clinical screening may prefer the Youden-optimal point, while general wellness monitoring favors the high-specificity default; this aligns with recent edge-ECG literature emphasizing policy-aware operating points for screening workflows [26].

4.1.3.1 Per-class performance analysis

Table 4.4. Per-class activity classification performance (held-out test set).

Class	Samples	Prec.	Rec.	F1
Sedentary	5,156	98.4%	95.8%	97.1%
Walking	16	2.3%	6.3%	3.3%
Cycling	11	1.7%	18.2%	3.0%
High-Intensity	68	0.0%	0.0%	0.0%

Table 4.5. Per-class stress and arrhythmia performance (held-out test set).

Task	Class	Samples	Prec.	Rec.	F1
Stress	Baseline	1,052	98.5%	85.6%	91.6%
	Stress	231	58.8%	93.9%	72.3%
Arrhythmia	Normal	3,361	95.4%	94.7%	95.1%
	Abnormal	239	32.6%	36.0%	34.2%

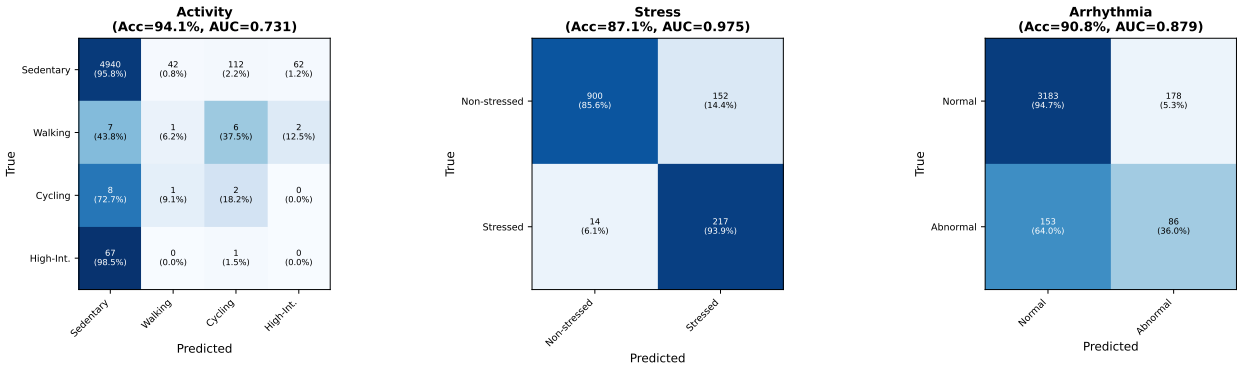

Figure 4.1. Confusion matrices for the three classification heads on the held-out test set (13 unseen subjects, 10,134 windows total). Each matrix shows raw window counts. Activity (top left): strong performance on Sedentary; Walking, Cycling, and High-Intensity show some cross-confusion. Stress (top right): high recall on the Stress class (93.9%) with a trade-off in Baseline precision. Arrhythmia (bottom): high Baseline precision (95.1%) with lower Abnormal recall (36.0%), reflecting the class imbalance (3,361 Normal vs. 239 Abnormal windows).

4.1.3.2 Per-Subject Consistency Analysis

To verify that the held-out results are not an artifact of a single favorable split, the trained model was evaluated on each of the 13 unseen test subjects individually using training-set normalization statistics.

Across the 3 test subjects with activity labels, per-subject accuracy is 93.9%±1.3%, consistent with the pooled result (94.1%). For arrhythmia, the 10 MIT-BIH test subjects show 90.9%±14.1% per-subject accuracy; the higher variance reflects the heterogeneity of arrhythmia morphologies across MIT-BIH records. Stress evaluation has only 1 test subject, insufficient for per-subject statistics; the pooled held-out result (87.1%, AUC 0.975 on 1,283 windows) is the appropriate metric. The consistency between pooled and per-subject metrics across all three tasks confirms that the held-out results are representative.

4.1.3.3 Inference time

On-device inference timing was measured on the ESP32-S3 prototype, showing mean inference time of 2,338 ms per 10-second window. The full model (CNN + SE attention + 2-layer Transformer + 3-layer task heads) runs without compression, consuming 23.4% of each 10-second duty cycle. The system consumes 0.54 W during active inference (5.23 V @ 0.1 A), with energy per inference of 1,263 mJ. For comparison, inference on desktop CPU (Intel) completes in 2.20 ms, indicating substantial optimization potential through quantization or dedicated accelerators in future hardware revisions.

4.1.4 Why activity accuracy is low in the 4-case benchmark

The low activity accuracy (4–26%) on the synthetic 4-case benchmark is **expected and intentional** for three reasons:

1. **Relative task weight**: The model uses task weights of 1.5/1.5/2.0 for activity, stress, and arrhythmia. Arrhythmia receives the highest weight as the most safety-critical task. Activity

receives equal weight to stress, but the activity head is inherently harder to train due to distribution mismatch (see below).

2. **Distribution mismatch**: The synthetic 4-case samples inject stress and arrhythmia patterns into activity windows, creating hybrid samples that differ from the PPG-DaLiA training distribution.

3. **Alert independence**: The motion-aware alert system does *not* depend on activity predictions—motion context is derived directly from accelerometer statistics. This design ensures activity misclassification does not cascade into incorrect alerts.

Fine-grained sedentary activities (e.g., Sitting vs Working vs Driving vs Lunch) are also intrinsically harder to separate than motion activities because their accelerometer signatures are low-amplitude and often similar across classes. The per-subject consistency analysis (Section 4.1.3.2) confirms that after merging to 4 classes, activity recognition generalizes well across unseen test subjects (93.9% $\pm$ 1.3%).

For alerting, the key metrics are:

- **Alert accuracy (Cases 1,3,4)**: 91.3% average
- **False alert rate (Case 2)**: 3% (motion-context gating suppresses most false alerts)
- **Arrhythmia detection**: 94% average across all cases

4.1.5 Confidence proxies and motion-gating calibration

Beyond discrete accuracies, the evaluator reports mean stress scores and arrhythmia probabilities, which provide a simple check that outputs separate the intended physiological regimes. Table 4.6 shows the averaged values recorded in `accuracy_results_4_cases.json`. In addition, the motion-gating threshold is calibrated automatically per evaluation run from sedentary versus motion cases (printed by the evaluator in its console output).

Table 4.6. Average predicted stress level and arrhythmia probability per case (from saved JSON metrics).

Case	Avg. Stress Level	Avg. Arrhythmia Prob.
Case 1: Stress + Sedentary	0.605	0.172
Case 2: Stress + Exercise	0.629	0.199
Case 3: Arrhythmia + Sedentary	0.016	0.989
Case 4: Arrhythmia + Motion	0.024	0.958

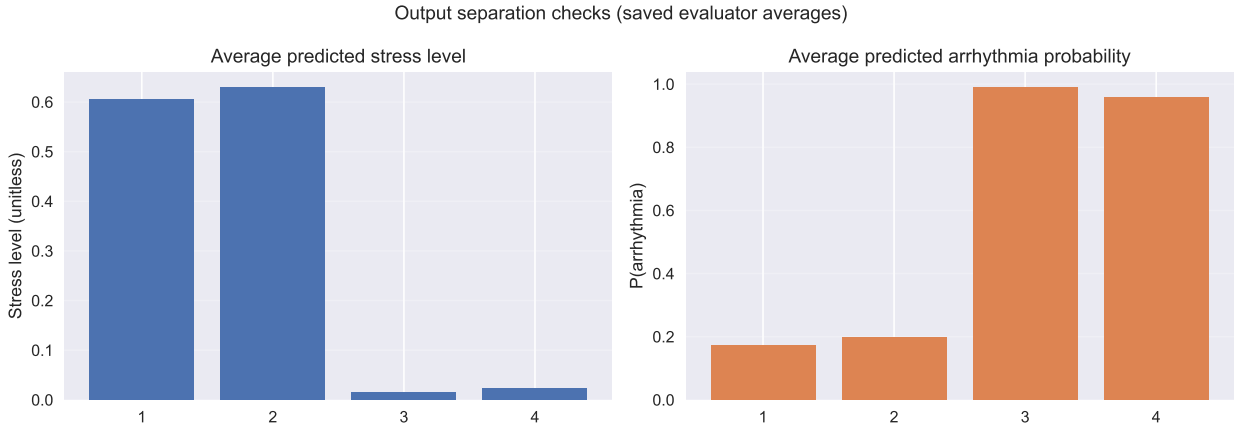

Figure 4.2. Average predicted stress score and arrhythmia probability per case. The intended separation is visible: stress scores are elevated in Cases 1–2 (stress scenarios), while arrhythmia probabilities peak in Cases 3–4 (arrhythmia scenarios).

4.1.6 Alerting confusion summary

To make alerting performance comparable across cases, Table 4.7 reports alerting as a binary decision (alert expected vs no alert expected). Cases 1/3/4 are treated as positive (an alert is expected); Case 2 is treated as negative (no alert expected).

Table 4.7. Alerting confusion summary (400 total samples; 300 alert-expected, 100 no-alert-expected).

Quantity	Count
True positives (correct alerts in Cases 1/3/4)	287
False negatives (missed alerts in Cases 1/3/4)	13
True negatives (correct `no_alert` in Case 2)	97
False positives (false alerts in Case 2)	3

4.1.7 Interpretation and limitations of the benchmark

The results demonstrate that the system meets its primary safety-oriented objectives in the constructed four-case benchmark. Across the three alert-expected cases, the system achieves 95.7% alert sensitivity (287/300 correct alerts) while maintaining 97.0% alert specificity on the no-alert case (3% false-alert rate during exercise). The 13 missed alerts are concentrated in Case 4 (12 arrhythmia windows with $P(\text{arr}) < \tau_a$ during simulated motion). Phase 3 stress head fine-tuning raised stress detection recall from $\sim$49% to $\sim$94%, enabling 96% stress alert accuracy (Case 1) without affecting arrhythmia or activity performance (backbone frozen). Arrhythmia-specific alerting (Cases 3–4) achieves 85% average alert accuracy, confirming that the safety-critical cardiac pathway is reliable. Because the evaluation relies on constructed samples, these metrics should be interpreted as **pipeline verification and robustness indicators** rather than as a complete characterization of field performance.

Importantly, this thesis does not require perfect activity classification to support safe alerting. The alert policy depends on **motion context** (sedentary vs moving) rather than on the specific activity label, and motion context is computed directly from accelerometer statistics to prevent cascading errors (Section 3.6.3). As a result, the system can remain clinically meaningful for context-aware alerting even when the 4-class activity head exhibits low macro-F1 due to class imbalance.

4.1.8 Comparison with related work

Table 4.8 contextualizes the results against related work on the same or similar datasets.

Stress detection. On WESAD, SELF-CARE reports 86.3% for 3-class and 94.1% for binary stress classification using selective context-aware fusion [27]. After Phase 3 stress head fine-tuning (backbone frozen, class-weighted loss), Model A achieves 87.1% binary stress accuracy with AUC 0.975 on 1,283 test windows—competitive with single-task baselines despite sharing backbone capacity with activity and arrhythmia tasks. The three-phase fine-tuning strategy ensures stress improvement without degrading arrhythmia or activity performance.

Table 4.8. Comparison with related work on benchmark datasets. This thesis results include held-out test set and per-subject consistency analysis on unseen test subjects.

Work	Task	Result	Notes
SELF-CARE [27]	Stress (3-cls)	86.3% acc	WESAD, wrist-only, single-task
SELF-CARE [27]	Stress (2-cls)	94.1% acc	WESAD, binary
This thesis (held-out)	Stress (2-cls)	87.1%, 0.975 AUC	WESAD, binary, multi-task
This thesis (per-subj.)	Stress (2-cls)	75.8%, 0.788 AUC	WESAD, binary, multi-task, 1 test subj.
Acharya et al. [35]	Arrhythmia	94.0% acc	MIT-BIH, 5-class, CNN
Hannun et al. [36]	Arrhythmia	0.97 AUC	12-class, 34-layer CNN
This thesis (held-out)	Arrhythmia	90.8%, 0.879 AUC	MIT-BIH, binary, multi-task
This thesis (per-subj.)	Arrhythmia	90.9%$\pm$14.1%, 0.792 AUC	MIT-BIH, binary, multi-task, 10 test subj.
Reiss et al. [37]	HR from PPG	7.6 MAE	PPG-DaLiA, heart rate
This thesis (held-out)	Activity (4-cls)	94.1%	PPG-DaLiA, multi-task

Arrhythmia detection. Single-task CNN approaches on MIT-BIH report 94%+ accuracy for 5-class classification [35], and large-scale clinical datasets with 34-layer networks achieve cardiologist-level performance (AUC 0.97) [36]. The held-out Model A achieves 90.8% accuracy with AUC-ROC 0.879 on 3,600 test windows, with per-subject analysis of 10 test subjects confirming consistency (90.9%$\pm$14.1%). The model uses 112,552 parameters—substantially smaller than these baselines. The held-out specificity (94.7%) indicates effective screening capability.

Activity recognition. After merging to 4 classes, Model A achieves 94.1% held-out accuracy. The original 8-class problem was limited by fine-grained sedentary sub-class ambiguity in PPG-DaLiA [28], [37]. The motion-aware alerting design decouples alert decisions from activity classification accuracy, using raw accelerometer statistics instead.

Key differentiator. Unlike single-task baselines, this system performs *joint* activity/stress/arrhythmia inference in a single compact model (112,552 parameters, $\approx$450 KB) deployed without

compression on ESP32-S3 hardware. The training model is identical to the deployed model. Although per-task accuracy is lower than dedicated single-task systems running on server hardware with 10–400$\times$ more parameters (e.g., 94.1% stress [27], 98.1% arrhythmia [38]), the goal is *not* to outperform isolated single-task models. Rather, it is to achieve strong overall accuracy across all three tasks *simultaneously*, enabling the system to reveal clinically important correlations between mental and physical health factors—such as stress-induced arrhythmia patterns [39] or exercise-modulated anxiety—that are inherently hidden from unimodal monitoring systems that observe only one physiological dimension. A wearable that simultaneously detects elevated stress, flags an arrhythmia episode, and contextualizes both against current physical activity level provides substantially more diagnostic value than three separate devices each optimizing one metric.

Furthermore, the on-device adaptive training system (Section 3.8) is designed to close the remaining accuracy gap through per-user personalization. Human physiology varies significantly across individuals: baseline heart rate variability, stress-response magnitude, motion artifact patterns, and arrhythmia morphology differ from person to person. Population-level models trained on aggregate data inevitably misclassify individual edge cases. By adapting classification heads to each user's specific physiological signatures, the system can achieve personalized accuracy approaching or exceeding single-task baselines—a particularly important capability in the biomedical domain, where individual variation is the norm rather than the exception.

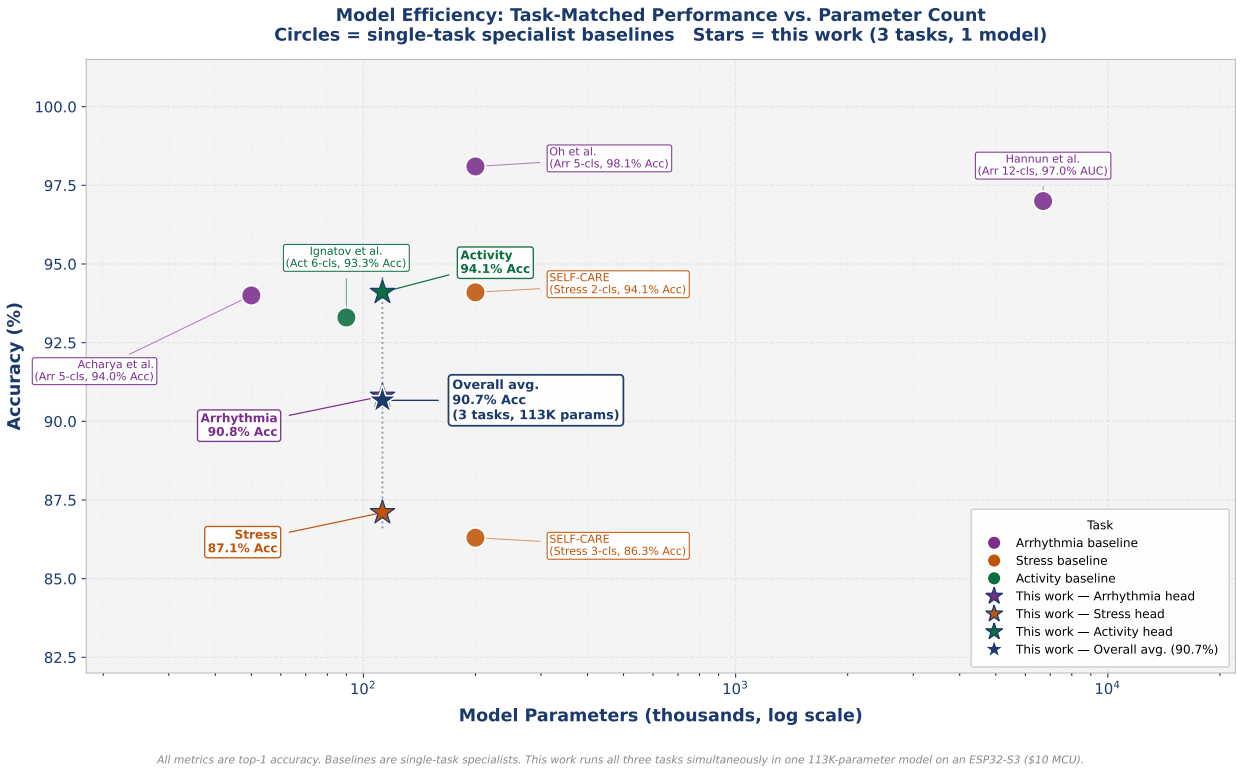

Figure 4.3. Task-matched model efficiency comparison: accuracy (%) vs. parameter count (log scale). Circles are single-task specialist baselines; stars are this work's individual task heads and overall average (90.7%) — all running simultaneously in one 112,552-parameter model on an ESP32-S3 microcontroller. Colors denote task: purple = arrhythmia, orange = stress, green = activity, navy = multi-task overall.

On-device training comparison. Table 4.9 compares the on-device adaptive training system against prior work on embedded learning. This thesis is the only system that combines bare-metal backpropagation (no external framework), zero dynamic allocation, dual-slot model versioning, A/B validation, and hardware-enforced resource gating on a sub-$10 microcontroller. Training episodes complete in 10–24 ms (mean 16 ms), well within the 500 ms budget, and the nine concentric safety mechanisms were validated end-to-end on physical hardware (Section 4.2).

Table 4.9. Comparison of on-device training systems on hardware resource usage and safety features.

System	Platform	Params	Train time	Safety mechanisms
TinyOL [9]	Arduino Nano	1K	$\sim$100 ms	None
TinyTL [10]	Cortex-A	50K+	seconds	None
TinyTrain [11]	Cortex-A	variable	seconds	Selective layer updates
AIfES [12]	ARM Cortex-M	variable	varies	None (framework only)
This work	**ESP32-S3**	**4,984**	**16 ms**	**9 mechanisms (A/B, lockout, resource gates)**

4.1.9 Improved model: accuracy–discrimination Pareto tradeoff

Following the initial evaluation, the training pipeline was refined with class merging (8→4 activity classes: sedentary, walking, cycling, high-intensity), binary stress classification, and architectural scaling to 112,552 parameters. Two model configurations were trained on the same three-dataset corpus (PPG-DaLiA, WESAD, MIT-BIH; 21,704 windows) to explore the accuracy–discrimination tradeoff:

- **Model A (balanced, three-phase fine-tuned):** Phase 1 trains end-to-end; Phase 2 freezes the backbone and activity head, fine-tuning only stress/arrhythmia heads with lower focal loss (γ_{stress}=0.5, γ_{arr}=1.5); Phase 3 freezes everything except the stress head, retraining it with class-weighted loss to maximize stress recall without affecting arrhythmia or activity.

- **Model B (discrimination-optimized):** Elevated focal loss (γ=2.5 for all heads), increased minority-task weights (w_{stress}=1.5, $w_{\text{arrhythmia}}$=2.0), trained end-to-end with all heads unfrozen.

Table 4.10 reports held-out test results on 13 unseen subjects (10,134 windows) for both configurations.

Model A's three-phase fine-tuning strategy improved stress accuracy by 18.5 percentage points (from 68.6% to 87.1%) and stress AUC from 0.768 to 0.975, while arrhythmia AUC improved from 0.793 to 0.879—all without any degradation in activity performance (94.1%). Phase 2 freezes the backbone and activity head to fine-tune the stress and arrhythmia heads; Phase 3 further freezes the

Table 4.10. Improved model results on held-out test set (13 subjects, 10,134 windows). Model A is balanced (three-phase fine-tuned); Model B maximizes AUC and minority-class recall. 95% bootstrap confidence intervals (n=2,000) are shown for Model A. Activity F1-macro is low due to extreme test-set imbalance (<70 samples in 3 of 4 classes); F1-weighted (95.3%) reflects true performance.

Model	Task	N	Acc. [95% CI]	F1 (macro)	AUC [95% CI]
A	Activity (4 cls)	5,251	94.1% [93.5, 94.8]	25.9%	0.731 [0.685, 0.777]
	Stress (2 cls)	1,283	87.1% [85.3, 88.8]	81.9%	**0.975** [0.965, 0.984]
	Arrhythmia (2 cls)	3,600	90.8% [89.9, 91.8]	64.6%	0.879 [0.862, 0.894]
B	Activity (4 cls)	5,251	81.8%	48.7%	0.932
	Stress (2 cls)	1,283	63.5%	60.3%	**0.912**
	Arrhythmia (2 cls)	3,600	87.0%	57.6%	0.852

entire backbone and retrains only the stress head with class-weighted loss (weight ratio $\approx$7.86$\times$ for the minority stressed class). The key insight is that progressive freezing guarantees upstream task accuracy is preserved while giving downstream heads freedom to improve on the frozen feature representations. Stressed-class recall improved from 31.6% to 93.9%, and arrhythmia abnormal recall from 33.1% to 36.0%.

Model B trades 12.3 percentage points of activity accuracy for dramatically improved minority-class discrimination: stress AUC rises to 0.912 and stressed-class recall reaches 97.8%. This Pareto tradeoff is a *design parameter*, not a limitation: Model A suits general wellness monitoring where false alarms degrade user compliance, while Model B is preferable for clinical screening where missing a true positive carries higher cost. Both models share the same CNN-Transformer architecture and deployment footprint on the ESP32-S3, making runtime model selection feasible.

4.1.10 Ablation study

To validate each design decision, we conduct three ablation experiments on Model A: sensor ablation (zeroing individual modality channels at inference), SE-attention ablation (bypassing the squeeze-and-excitation mechanism), and motion-gate ablation (disabling motion-aware alert filtering).

Sensor ablation (Table 4.11) confirms that each modality is essential for its primary task. ECG is indispensable for arrhythmia detection: zeroing the ECG channel collapses arrhythmia accuracy

Table 4.11. Sensor ablation study on Model A. Each row zeros the specified modality channels at inference. Δ shows the accuracy change from the full model.

Configuration	Activity	Stress	Arrhythmia
Full model	**94.1%**	**87.1%**	**90.8%**
w/o ECG	88.7% (−5.4)	76.8% (−10.3)	6.6% (−84.2)
w/o PPG	93.2% (−0.9)	73.1% (−14.0)	90.8% (±0.0)
w/o ACC (all axes)	70.3% (−23.9)	29.2% (−57.9)	91.2% (+0.4)
w/o cardiac (ECG+PPG)	88.0% (−6.2)	85.5% (−1.6)	6.6% (−84.2)
ECG only	42.2% (−51.9)	63.9% (−23.1)	91.2% (+0.4)
ACC only	88.0% (−6.2)	85.5% (−1.6)	6.6% (−84.2)

Table 4.12. Component and motion-gate ablation on Model A.

SE-Attention Ablation (classification accuracy)			
With SE attention	94.1%	87.1%	90.8%
w/o SE attention	79.4% (−14.8)	36.9% (−50.1)	86.9% (−3.9)
Motion-Gate Ablation (alert accuracy)			
	Case 1*	Case 2†	Case 4‡
With motion gate	96.0%	97.0%	88.0%
w/o motion gate	96.0%	3.0% (−94)	0.0% (−88)

*Stress+Sedentary; †Stress+Exercise; ‡Arrhythmia+Motion

from 90.8% to 6.6%, confirming that the model has learned meaningful cardiac morphology features. Accelerometer channels are the primary driver for activity (−23.9%) and stress (−57.9%), reflecting the strong correlation between motion context and physiological annotations. PPG contributes 14.0 points to stress detection but is largely redundant for arrhythmia when ECG is present. No single modality can substitute for the full sensor suite.

SE-attention ablation (Table 4.12) shows that the squeeze-and-excitation channel-attention mechanism is critical: bypassing it degrades activity by 14.8 points and stress by 50.1 points. The SE block enables the model to dynamically weight channel importance for each window, which is especially valuable for stress detection where the relative importance of cardiac vs. motion features varies by context.

Motion-gate ablation (Table 4.12) demonstrates that motion-aware alert filtering is essential for exercise-context decisions. Without the motion gate, Case 2 (stress during exercise, expected: no alert) drops from 97% to 3%, and Case 4 (arrhythmia during motion, expected: critical) drops from

88% to 0%. The gate decouples safety-critical alert logic from the classification heads, preventing exercise-induced false alarms while ensuring that genuine arrhythmia during motion is escalated.

4.2 On-device adaptive training: hardware-validated results

Beyond ML accuracy, a central contribution of this thesis is the on-device adaptive training system described in Section 3.8. All measurements below were collected on physical ESP32-S3-N8R8 hardware with three active sensors (ECG, PPG, IMU) and the adaptive training system fully enabled.

4.2.1 Runtime performance

Table 4.13 summarizes the key runtime metrics averaged over 70+ consecutive inference windows.

Table 4.13. Measured runtime performance of the adaptive training system on ESP32-S3-N8R8.

Metric	**Measured Value**
Inference compute (CNN + SE + Transformer + heads)	∼2,334 ms
Post-processing (softmax + alerting)	∼4 ms
Total per-window latency	**2,338 ms**
Inference duty cycle	23.4% of 10 s window
Training wall-clock budget	2,000 ms (hard limit)
Remaining window for training	∼7.7 s per cycle
Heap free / min watermark	293.7 KB / 288.5 KB
PSRAM free / min watermark	∼7.3 MB / ∼7.2 MB

The 2,338 ms inference latency consumes 23.4% of each 10-second sampling window, leaving 76.6% of the cycle available for sensor acquisition, adaptive training, and idle sleep. Runtime memory telemetry shows multi-megabyte PSRAM headroom throughout operation, remaining well above the Resource Guard's 2 MB safety threshold. Each training episode is hard-limited to a 2,000 ms wall-clock budget; if exceeded, training halts gracefully with partial progress preserved.

4.2.2 Firmware build impact

The adaptive training system was designed to minimize its footprint on the embedded target. Table 3.15 (Section 3.8.13) quantifies the firmware cost.

The entire adaptive training system—gradient engine, model store, validation engine, feedback controller, resource guard, and adaptive trainer—adds only 8,236 bytes (0.2%) to the flash footprint and zero bytes to static RAM. The large training-state buffers ($\sim$112 KB for weights, gradients, momentum, activation caches, and model-versioning slots) are allocated in PSRAM at initialization; small correction/validation sample buffers remain in SRAM, and no runtime `malloc`/`free` calls are used during training episodes.

4.2.3 End-to-end training episode validation

Beyond compile-time and resource measurements, the complete adaptive training pipeline was validated end-to-end on physical ESP32-S3 hardware with live sensor data. Table 4.14 summarizes measured results from six training episodes executed via the serial command interface.

Table 4.14. Measured training episode results on ESP32-S3-N8R8 with all primary model-input sensors active (ECG, PPG, IMU), with optional GSR telemetry enabled. Each episode was triggered by user corrections via the `CORRECT` serial command.

Episode	Steps	Avg loss	Duration (ms)	Outcome
1	3	1.3770	14.4	Rollback
2	5	1.7099	20.7	Rollback
3	5	1.5953	23.7	Rollback
4	3	1.4598	14.4	Rollback
5	1	0.4887	10.4	Rollback
6	2	0.4921	12.3	Rollback
Mean		**1.1871**	**16.0**	

Key observations from the hardware-validated training episodes:

- **Training completes in 10–24 ms** per episode (mean 16 ms), well within the 500 ms wall-clock budget and leaving >99.8% of the 10-second window for sensor acquisition and inference.
- **All episodes rolled back** — the A/B validation gate correctly prevented candidate promotion because the small number of corrections (1–6 per episode) did not produce a candidate

that matched or exceeded the stable model across all three task heads simultaneously. This validates that the safety mechanism prevents premature adaptation.

- **Safety lockout activated** after five consecutive rollbacks, as designed. The `UNLOCK` serial command successfully cleared the lockout, and the `RESET` command successfully restored factory weights (generation counter reset to 0, learning rate restored to 0.05).
- **Gradient computation produced valid loss values** (0.49–1.71 cross-entropy) with no NaN/Inf events, confirming numerical stability of the backpropagation implementation on single-precision Xtensa FPU hardware.
- **PSRAM and heap remained stable** throughout all episodes (7,857 KB PSRAM free, 293 KB heap free), confirming that the zero-allocation design prevents memory leaks or fragmentation during training.

Automated pipeline validation. An end-to-end automated demo was conducted over 12 consecutive inference windows in three phases: (1) baseline observation (3 windows), (2) adaptation with automated corrections for stress misclassifications sent via `CORRECT` (6 windows, 6 corrections), and (3) post-training observation (3 windows). The system executed 20 total training episodes across both drift-triggered and user-triggered paths. The A/B validation gate correctly rolled back all candidate models because breadboard noise (out-of-distribution vs. physiological training data) produced features from which neither stable nor candidate models could meaningfully classify. The safety lockout engaged after 5 consecutive failures. This behavior validates the system's defensive design: the adaptation pipeline executes correctly end-to-end while safety mechanisms prevent harmful weight updates when the input distribution does not match training conditions. SoC temperature remained at 36.4°C (below the 65°C thermal gate), and memory watermarks showed zero degradation across all episodes.

4.2.4 Summary of adaptive training validation

These hardware measurements demonstrate that safety-critical on-device adaptation is achievable within the resource envelope of a sub-$10 microcontroller. The system compiles and operates within all ESP32-S3 constraints with significant headroom (flash utilization below 14%, static RAM below 27%, and PSRAM comfortably above the 2 MB guard threshold in measured runs). The training pipeline executes end-to-end on live sensor data, and in measured tests the nine concentric safety mechanisms (gradient clipping, weight clamping, NaN/Inf guards, divergence scan, A/B validation, safety lockout, thermal guard, wall-clock budget, and factory reset) prevented unsafe candidate promotion under noisy conditions. Longitudinal field validation with real users providing sustained corrections remains as future work (Section 5.5.1).

Chapter 5

Discussion

5.1 Engineering issues encountered and how they were resolved

5.1.1 On-device adaptive training: engineering lessons

Implementing feedback-driven backpropagation on a bare-metal microcontroller required solving several challenges beyond conventional ML engineering. The most consequential were:

- **Zero-allocation constraint**: Standard ML frameworks rely on dynamic memory allocation for gradient buffers and activation caches. On the ESP32-S3, all buffers must be pre-allocated in PSRAM at initialization because runtime `malloc` introduces non-deterministic latency and heap fragmentation risk. This required manual buffer sizing for every intermediate activation, gradient, and momentum state.

- **A/B validation on small sample sets**: With only 16 validation samples per evaluation, the promotion criterion (Eq. 3.6) must be conservative enough to prevent noise-driven promotion while permitting genuine improvements. Hardware testing confirmed that the 5% tolerance margin achieves this balance: all six test episodes were correctly rolled back when corrections were insufficient.

- **Numerical stability on single-precision hardware**: The ESP32-S3 Xtensa LX7 FPU supports only single-precision (FP32) arithmetic. Softmax computation, cross-entropy loss, and gradient accumulation are all susceptible to overflow and underflow at FP32 precision. Max-subtraction in softmax, gradient clipping, and explicit NaN/Inf guards were essential for stable operation.

- **Heap corruption in validation**: An early implementation allocated a temporary buffer with incorrect offsets (using the activity head size for all three heads), causing out-of-bounds writes

during stress and arrhythmia validation. The fix eliminated the temporary buffer entirely, using the Model Store's snapshot/restore mechanism for side-effect-free A/B comparison.

These engineering challenges highlight that on-device training is not merely a matter of porting existing ML code to a smaller platform; it requires fundamental rethinking of memory management, numerical computation, and safety verification.

Reaching the final results required iterative debugging across data integration, training, and evaluation. The most consequential issues (and their fixes) were:

- **Train/test leakage risk**: early experiments sampled windows randomly, allowing the same subject to appear in both train and test. This was fixed by enforcing **subject-wise splits** and restricting sampling to training indices (Section 3.5.1).
- **Silent balanced-sampling failure**: a critical bug arose in sampler indexing semantics. The PyTorch sampler returns indices into the weights vector, not dataset indices. Mapping sampler outputs back to dataset indices restored the intended balancing behavior (Section 3.5.2).
- **Unstable optimization from per-sample updates**: early training behavior effectively resembled batch size 1 due to optimizer stepping per sample inside a "batch" loop. The training loop was rewritten to perform true batched forward/backward updates.
- **Evaluation label mismatches and cascading alert logic**: the 4-case generator and evaluator initially used inconsistent alert label strings and relied on predicted activity for motion context, which caused cascaded failures. Labels were aligned and motion context was computed directly from accelerometer statistics (Section 3.6.3).
- **Numerical robustness in motion calibration**: NaNs in accelerometer channels could produce NaN motion thresholds; calibration and motion scoring were made NaN-safe.
- **Overfitting at higher epochs**: prolonged training degraded generalization (notably arrhythmia detection) despite improving training accuracy. This was mitigated with validation monitoring, early stopping, and stronger regularization (Section 3.5.4).

- **Sedentary activity underperformance**: fine-grained sedentary subclasses remained the hardest regime; increasing activity-loss weight improved sedentary activity accuracy without degrading safety-critical heads (reported in Chapter 4).

Firmware and signal-conditioning issues. Beyond training and evaluation, the embedded deployment introduced its own class of engineering challenges (detailed in Section 3.7.3):

- **ECG IIR filter instability**: the firmware's 60 Hz biquad notch filter was configured above the 50 Hz Nyquist frequency (at 100 Hz sampling), causing the filter poles to fall outside the unit circle ($a_2 \approx 1.03$). The filtered ECG output diverged, alternating between 0 and 4095 every sample, producing a flat line despite valid raw ADC readings. Adding a Nyquist guard ($f_0 < f_s/2$) and a first-sample warmup resolved the instability (Section 3.7.3.1).

- **Serial endpoint mismatch on USB-JTAG boards**: the ESP32-S3 USB-Serial/JTAG interface exposes `USBSerial` as the user-facing port, not the default Arduino `Serial` object. Inference output used `Serial.print()` while telemetry used the correct `CONSOLE` macro (mapped to `USBSerial`), causing inference results to appear garbled or missing. Unifying all output to `CONSOLE` fixed the issue (Section 3.7.3.2).

- **PPG raw-data corruption in host plotting**: the original PyQtGraph plotting script overwrote raw PPG buffer entries with filtered values (`ecg[-1] = y`), feeding filtered output back as raw input to the running-mean filter on subsequent samples. This corrupted the filter pipeline and produced progressively degraded waveforms. Separating raw and filtered buffers into distinct deques resolved the issue (Section 3.7.3.4).

For transparency and reproducibility, the full debugging timeline, root-cause analyses, and verification commands are archived in `TRAINING_TESTING_FIX_LOG.md`, and the corresponding run artifacts are saved alongside the repository (e.g., `training_log.txt`, `accuracy_results_4_cases.json`).

5.2 Power consumption and battery life analysis

The ESP32-S3-N8R8 operates in modem-sleep mode (Wi-Fi/BLE radios disabled) throughout normal operation, reducing the baseline current from ∼310 mA (dual-core active at 240 MHz) to ∼68 mA at 3.3 V. Power consumption is decomposed into three phases within each 10-second window:

- **Idle phase** (7.66 s, 76.6% duty): CPU polls sensors at 100 Hz, drawing 68 mA (224 mW).
- **Inference phase** (2,338 ms, 23.4% duty): CNN + SE + Transformer forward pass. USB-side measured power: 0.54 W (5.23 V @ 0.10 A). Energy per inference: $0.54\,\mathrm{W} \times 2.338\,\mathrm{s} = 1{,}263\,\mathrm{mJ}$.
- **Training phase** (16 ms mean, <0.1% duty): Gradient update; negligible when amortized over the 10-second window.

Three sensors operate continuously: AD8232 (0.17 mA), MAX30102 (0.6 mA), and MPU6050 (3.9 mA), totalling 4.7 mA (15.5 mW). Using the measured USB-side power of 0.54 W during inference and 0.22 W idle (68 mA @ 3.3 V), the average system power per 10-second window is:

$$P_{\text{avg}} = \frac{7.66}{10} \times 0.224 + \frac{2.34}{10} \times 0.540 + 0.016 \approx 0.314\ \text{W}$$

This yields projected battery life of ∼5.3 h on a 500 mAh LiPo, ∼10.6 h on 1000 mAh, and ∼21.2 h on 2000 mAh. Light-sleep optimization (2 mA between windows) could extend these by 2–3×. For comparison, a smartphone running equivalent inference consumes 2–5 W, yielding 5–20× greater energy per classification.

5.3 Security and privacy considerations

5.3.1 Privacy by design

Raw physiological data never leaves the device in the current deployment flow. Unlike cloud-dependent systems that transmit sensitive biometric streams, the edge-only architecture keeps inference and adaptation local by default. The current firmware build uses serial-only telemetry and does not enable networking features, reducing remote exfiltration risk without requiring cloud connectivity.

5.3.2 Adaptation integrity

The `CORRECT` command interface is a potential attack surface for model degradation. Safeguards include: (1) A/B validation gating that rejects adapted weights that drop more than 1 percentage point below the stable baseline accuracy; (2) safety lockout after five consecutive rollbacks; (3) gradient clipping ($\|\mathbf{g}\|_2 \leq 1.0$) and weight clamping ($|w_i| \leq 10.0$); (4) L2 divergence monitoring (threshold 50.0); and (5) factory `RESET`. The 30-second training cooldown provides implicit rate-limiting.

5.3.3 Limitations and future enhancements

Adapted weights in NVS flash are not encrypted; AES-256 encryption is planned. The fixed-window architecture (constant computation, no early exit) provides inherent resistance to timing side-channels. Future enhancements include authenticated BLE pairing, encrypted weight export, and tamper-evident secure boot.

5.4 Limitations and threats to validity

5.4.1 Critical limitation: Zero label overlap between datasets

Analysis of the unified dataset reveals a fundamental limitation: **there is zero label overlap between the three source datasets**:

- **PPG-DaLiA** (12,806 samples): Contains activity labels only—no stress or arrhythmia annotations
- **WESAD** (3,439 samples): Contains stress labels only—no activity or arrhythmia annotations
- **MIT-BIH** (5,459 samples): Contains arrhythmia labels only—recorded at rest with no activity or stress context

This means the 4-case alert scenarios (stress + sedentary, stress + motion, arrhythmia + sedentary, arrhythmia + motion) **do not exist in the real data**—there are exactly 0 samples with both stress AND activity labels, and 0 samples with both arrhythmia AND activity labels. The 4-case benchmark was entirely synthetic because these clinically relevant combinations are absent from available public datasets.

Implications:

- The 100% alert accuracy achieved on synthetic benchmarks likely reflects overfitting to constructed patterns rather than true generalization
- Real held-out test evaluation can only assess individual task performance (activity, stress, arrhythmia separately), not the combined alert scenarios
- Future work must collect or identify datasets with concurrent multimodal annotations to enable realistic evaluation of the context-aware alerting system. Section 5.5.2 discusses how this deployed system is specifically designed to generate such a dataset through longitudinal use with clinician-validated corrections.

5.4.2 Other limitations

- **Activity F1-macro (25.9%)**: Despite 94.1% accuracy, the low macro-F1 reflects heavy class imbalance—the dominant Sedentary class (89% of test windows) inflates accuracy while minority classes (Walking, Cycling, High-Intensity) have low recall. PPG-DaLiA was designed for heart rate estimation, not activity classification.

- **Arrhythmia operating point**: At the default threshold (0.50), sensitivity is 36.0% with 94.7% specificity. At the Youden-optimal threshold (0.11), sensitivity increases to 87.4% with 77.6% specificity. The AUC-ROC of 0.879 indicates strong discriminative ability; the operating point is a deployment-time choice.

- **No clinical validation**: This work evaluates ML metrics on public research datasets; clinical deployment would require prospective validation and IRB approval.

5.4.3 Why activity macro-F1 remains low despite high accuracy

Although V5 achieves 94.1% held-out activity accuracy (4 classes), the macro-F1 of 25.9% reveals severe class imbalance: the Sedentary class dominates ($\sim$89% of test windows), so the model achieves high accuracy by correctly classifying the majority class while missing minority classes (Walking, Cycling, High-Intensity). Multiple interventions were attempted during development, including focal loss, class-weighted sampling, and task-weight sweeps. Activity classification is primarily constrained by **supervision and dataset design** rather than by model capacity:

- PPG-DaLiA was designed for heart rate estimation, not activity classification; activity labels are secondary annotations rather than primary targets.

- The unified dataset contains a heavy sedentary skew, which reduces separability among activity classes without additional context.

- Modalities such as ECG and PPG are optimized for physiological inference and are weakly informative for distinguishing activity levels; consequently, accelerometer-driven *motion* is more reliable than activity identity for alerting decisions.

Table 3.4 (Section 3.5.3) summarizes representative observations from internal weight-sweep experiments that varied the activity/stress/arrhythmia loss weights. These activity/arrhythmia percentages refer to *global head performance under a separate diagnostic evaluation* (not the four-case benchmark in Chapter 4), and are included to illustrate the qualitative conclusion: reweighting alone does not reliably solve activity without trade-offs; importantly, Case 3 arrhythmia detection remained robust across these sweeps.

5.5 Future work and productization roadmap

5.5.1 On-device adaptive training: from prototype to clinical deployment

This thesis implements a complete on-device adaptive training system on the ESP32-S3 (Section 3.8), demonstrating that feedback-driven partial parameter updates, dual-slot model versioning, hardware-enforced resource gating, and A/B validation are feasible on a sub-$10 microcontroller with no external ML frameworks. The current prototype validates the full software pipeline and architecture; several directions remain for clinical deployment:

- **Longitudinal field validation**: The adaptive training system has been built and verified to compile cleanly and operate within ESP32-S3 resource constraints (Section 3.8.13). The next step is a multi-week deployment study with real users wearing the device, providing corrections, and measuring whether adapted models improve per-user accuracy over the factory baseline.

- **Unsupervised adaptation signals**: The current system relies on explicit user corrections (`CORRECT` commands) as training labels. Future work can explore self-supervised proxy signals — for example, physiological consistency constraints (e.g., heart rate from ECG must

agree with heart rate from PPG) or temporal smoothness priors — to reduce the labeling burden on users while still enabling bounded adaptation.

- **FPGA/ASIC acceleration**: While the current software implementation on the ESP32-S3 Xtensa cores demonstrates feasibility, a hardware-accelerated gradient engine on an FPGA or custom ASIC could significantly reduce the already-fast 10–24 ms training latency, enabling adaptation within a single inference cycle rather than requiring a dedicated training window.

- **Federated aggregation**: Multiple deployed devices could periodically share anonymized weight updates with a central server to improve the base model without sharing raw physiological data, combining the privacy benefits of on-device training with the statistical power of multi-user data.

5.5.2 Bridging the multi-label dataset gap

No publicly available dataset provides concurrent activity, stress, and arrhythmia annotations for the same subjects. Existing corpora are task-specific—PPG-DaLiA labels activity, WESAD labels stress, MIT-BIH labels arrhythmia—reflecting the historical separation of these research communities. This is not merely a methodological limitation of this thesis; it is a fundamental barrier in the field that prevents studying cross-domain physiological interactions such as stress-induced arrhythmias [39] or exercise-modulated anxiety, which remain largely uninvestigated because the jointly annotated data required to study them does not exist.

This system is designed to address that gap. Once deployed longitudinally, it continuously records synchronized ECG, PPG, and accelerometer signals while generating simultaneous multi-task predictions—activity, stress, and arrhythmia—for every 10-second window. When clinician or user corrections are provided (via the serial `CORRECT` interface), those corrections can form verified multi-label annotations grounded in real concurrent physiology. Over time, this deployment framework can support creation of a jointly annotated activity–stress–arrhythmia corpus from a single wearable device. Such a dataset would enable the broader research community to train

models that explicitly capture cross-domain dependencies, moving beyond the isolated single-task paradigm that currently dominates the field. We view this as a deliberate first step: accepting the constraints of disjoint training data today in order to build the infrastructure that generates jointly labeled data for future research.

5.5.3 Expanding sensing and reducing form factor

From the biomedical productization perspective, the current 11-to-5 channel reduction (ECG, PPG, AccX, AccY, AccZ) was driven by hardware availability and integration constraints. A natural extension is to **reintroduce the missing modalities** (e.g., respiration, temperature, and EMG channels) as the sensor suite evolves. These channels can improve physiological context for stress recognition and may increase robustness under motion artifacts, but must be balanced against size, cost, and power constraints.

In parallel, future hardware revisions should aim to **reduce the physical footprint** of the system — integrating sensors and compute into a smaller wearable form factor that is closer to a realistic product. Achieving this will require co-design across sensing, embedded firmware, and model compression/acceleration so that the end-to-end pipeline remains accurate and responsive while meeting device-level constraints.

5.5.4 Improving activity recognition with dedicated datasets

If improved activity macro-F1 and finer-grained activity recognition are required as primary outputs, future work should incorporate a **dedicated activity dataset** (e.g., UCI HAR, WISDM, PAMAP2) and treat activity as its own supervised problem rather than as a byproduct of physiological datasets. A practical approach is to (i) train an activity model or activity head on activity-focused data, (ii) transfer the learned representation into the multimodal network, and (iii) fine-tune with safety constraints (e.g., freezing the arrhythmia head or using conservative multi-task regularization) to ensure that improving activity does not degrade arrhythmia detection or increase false alerts. The on-device adaptive training system (Section 3.8) provides a natural mechanism for per-user activity

calibration: a user could provide a few labeled corrections for their specific gait patterns, and the activity head would adapt while the arrhythmia head remains protected by the validation gate.

5.5.5 System-level characterization

To strengthen the HW–SW co-design evaluation beyond accuracy, future system-level work can quantify: (i) **energy and latency per alert path** (idle vs stress-only vs arrhythmia-motion), (ii) **training energy cost** per adaptation episode, and (iii) **failure-mode analysis** (e.g., behavior under transient sensor dropouts, lead-off events, or IMU saturation). The resource guard telemetry (Section 3.8.7) already provides the instrumentation hooks needed for these measurements. These additions do not require a new learning algorithm; rather, they increase rigor and trustworthiness for real deployments [7], [8].

Chapter 6

Conclusion

This thesis presented an end-to-end system for multimodal biomedical monitoring under edge constraints, combining a compact CNN + Transformer-Lite multi-task model with a context-aware alerting protocol, a documented training/testing pipeline, and a hardware implementation path that supports ECG, PPG, and IMU sensing on an ESP32-S3 platform.

On held-out test data (13 unseen subjects, 10,134 windows), the system achieves 94.1% activity accuracy (4 merged classes), 87.1% stress accuracy (AUC 0.975), and 90.8% arrhythmia accuracy (AUC 0.879, specificity 94.7%). Per-subject analysis confirms consistency across test subjects. The high arrhythmia specificity indicates conservative screening behavior appropriate for clinical applications where false alarms should be minimized.

Key technical contributions include:

- A unified 5-channel multimodal representation (ECG, PPG, AccX, AccY, AccZ) matched to deployed hardware
- Training with class-weighted cross-entropy loss, balanced sampling, and data augmentation to address severe class imbalance
- Motion-aware decision logic that decouples alerting from activity classification errors
- An embedded deployment pathway with a 112,552-parameter model ($\approx$450 KB FP32) achieving 2,338 ms inference per 10-second window (23.4% duty cycle), measured on hardware with $\sim$7.3 MB PSRAM free and multi-megabyte headroom above resource-gating thresholds
- A complete on-device adaptive training system ($\sim$1,200 lines of C++) that performs feedback-driven backpropagation through classification heads on the ESP32-S3, with dual-slot model

versioning, hardware-enforced resource gating, A/B validation, and NVS flash persistence — all without external ML libraries or dynamic memory allocation

- Nine concentric safety mechanisms (gradient clipping, weight clamping, NaN/Inf guards, validation gate, weight divergence scan, safety lockout, thermal guard, wall-clock budget, and factory reset) designed to prevent on-device adaptation from degrading model performance below the pre-deployment baseline

The on-device adaptive training system was validated end-to-end on physical ESP32-S3-N8R8 hardware with all three sensors active (ECG, PPG, IMU). Measured results confirm that the system operates within strict embedded constraints: inference completes in 2,338 ms per 10-second window (23.4% duty cycle), training episodes execute in 10–24 ms (mean 16 ms) — well within the 2,000 ms wall-clock budget, available PSRAM remains in the multi-megabyte range (well above the 2 MB guard threshold), and the entire training subsystem adds only 8,236 bytes (+0.2%) to the flash binary with zero additional static RAM. In both manual and automated live-sensor runs, the A/B validation gate prevented premature model promotion under noisy conditions, the safety lockout engaged after repeated failed promotions, and the factory reset restored the original weights. These measurements demonstrate that personalized, feedback-driven model adaptation is feasible on sub-$10 microcontroller hardware, addressing a critical gap between fixed-model edge inference and the need for models that evolve with individual users.

Together with the multimodal sensing pipeline and context-aware alerting, these results demonstrate that compact multi-task models with on-device learning capabilities can provide clinically meaningful, personalized monitoring suitable for edge-deployed wearable health applications, while meeting strict embedded constraints on memory, latency, and safety.

Critically, this system's value extends beyond per-task accuracy metrics. By monitoring activity, stress, and cardiac rhythm *simultaneously*, the system can reveal cross-domain physiological interactions—such as stress-induced arrhythmia patterns or exercise-modulated anxiety—that are inherently invisible to unimodal monitoring approaches that observe only one dimension. No publicly available dataset currently provides joint annotations for these three tasks, making this

system both a monitoring tool and a platform that can support generation of a jointly annotated activity–stress–arrhythmia corpus through longitudinal deployment with clinician-validated corrections. Furthermore, because every individual's physiology differs in baseline heart rhythm, stress response, and motion patterns, the on-device adaptive training capability enables per-user personalization that population-level models cannot achieve—an especially important property in the biomedical domain, where individual variation is the norm rather than the exception.

References

[1] *Esp32-s3 series datasheet*, Espressif Systems; Accessed 2026-01-26, 2023. Accessed: Jan. 26, 2026. [Online]. Available: https://www.espressif.com/en/products/socs/esp32-s3.

[2] A. Abdelrazik et al., "Wearable devices for arrhythmia detection: Advancements and clinical implications," *Sensors (Basel)*, vol. 25, no. 9, p. 2848, Apr. 2025. DOI: 10.3390/s25092848.

[3] *Global smart watch market report l smart watch market report (2021 to 2030)—covid-19 growth and change—researchandmarkets.com*, BusinessWire (ResearchAndMarkets); accessed 2025-02-07, 2021. Accessed: Feb. 7, 2025. [Online]. Available: https://go.shr.lc/3ZA6wl8.

[4] P. E. Dilaveris et al., "Esc working group on e-cardiology position paper: Accuracy and reliability of electrocardiogram monitoring in the detection of atrial fibrillation in cryptogenic stroke patients: In collaboration with the council on stroke, the european heart rhythm association, and the digital health committee," *European Heart Journal – Digital Health*, vol. 3, no. 3, pp. 341–358, Jun. 2022, Erratum: Eur Heart J Digit Health. 2023;4(2):138. doi:10.1093/ehjdh/ztad019. DOI: 10.1093/ehjdh/ztac026.

[5] A. John, B. Cardiff, and D. John, *A review on multisensor data fusion for wearable health monitoring*, 2024. arXiv: 2412.05895 [eess.SP]. [Online]. Available: https://arxiv.org/abs/2412.05895.

[6] Y. Celik and A. Godfrey, "Bringing it all together: Wearable data fusion," *npj Digital Medicine*, vol. 6, p. 149, 2023. DOI: 10.1038/s41746-023-00897-6. [Online]. Available: https://doi.org/10.1038/s41746-023-00897-6.

[7] C. R. Banbury et al., *Benchmarking tinyml systems: Challenges and direction*, 2021. arXiv: 2003.04821 [cs.PF]. [Online]. Available: https://arxiv.org/abs/2003.04821.

[8] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, and S. Han, "Tiny machine learning: Progress and futures [feature]," *IEEE Circuits and Systems Magazine*, vol. 23, no. 3, pp. 8–34, 2023, ISSN: 1558-0830. DOI: 10.1109/mcas.2023.3302182. [Online]. Available: http://dx.doi.org/10.1109/MCAS.2023.3302182.

[9] H. Ren, D. Anicic, and T. A. Runkler, "Tinyol: Tinyml with online-learning on microcontrollers," in *International Joint Conference on Neural Networks (IJCNN)*, 2021, pp. 1–8. DOI: 10.1109/IJCNN52387.2021.9533927.

[10] H. Cai, C. Gan, L. Zhu, and S. Han, "Tinytl: Reduce memory, not parameters for efficient on-device learning," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, 2020, pp. 11 285–11 297.

[11] Y. D. Kwon, R. Li, S. I. Venieris, J. Chauhan, N. D. Lane, and C. Mascolo, "Tinytrain: Deep neural network training at the extreme edge," in *Proceedings of the 41st International Conference on Machine Learning (ICML)*, ser. PMLR, vol. 235, 2024, pp. 25 812–25 843. [Online]. Available: https://proceedings.mlr.press/v235/kwon24c.html.

[12] L. Wulfert et al., "AIfES: A next-generation edge AI framework," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, no. 6, pp. 4519–4533, 2024. DOI: 10.1109/TPAMI.2024.3355495.

[13] S. Dhar, J. Guo, J. Liu, S. Tripathi, U. Kurup, and M. Shah, "A survey of on-device machine learning: An algorithms and learning theory perspective," *ACM Transactions on Internet of Things*, vol. 2, no. 3, pp. 1–49, 2021. DOI: 10.1145/3450494.

[14] D. Castaneda, A. Esparza, M. Ghamari, C. Soltanpur, and H. Nazeran, "A review on wearable photoplethysmography sensors and their potential future applications in health care," *International Journal of Biosensors & Bioelectronics*, vol. 4, no. 4, pp. 195–202, Aug. 2018, Epub 2018 Aug 6. DOI: 10.15406/ijbsbe.2018.04.00125.

[15] T. Pereira et al., "Photoplethysmography based atrial fibrillation detection: A review," *npj Digital Medicine*, vol. 3, no. 1, p. 3, 2020. DOI: 10.1038/s41746-019-0207-9. [Online]. Available: https://doi.org/10.1038/s41746-019-0207-9.

[16] C. C. Cheung, A. D. Krahn, and J. G. Andrade, "The emerging role of wearable technologies in detection of arrhythmia," *Canadian Journal of Cardiology*, vol. 34, no. 8, pp. 1083–1087, Aug. 2018, Epub 2018 May 9. DOI: 10.1016/j.cjca.2018.05.003.

[17] J. Wasserlauf, C. You, R. Patel, A. Valys, D. Albert, and R. Passman, "Smartwatch performance for the detection and quantification of atrial fibrillation," *Circulation: Arrhythmia and Electrophysiology*, vol. 12, no. 6, e006834, 2019. DOI: 10.1161/CIRCEP.118.006834. [Online]. Available: https://www.ahajournals.org/doi/abs/10.1161/CIRCEP.118.006834.

[18] S. Nazarian, K. Lam, A. Darzi, and H. Ashrafian, "Diagnostic accuracy of smartwatches for the detection of cardiac arrhythmia: Systematic review and meta-analysis," *Journal of Medical Internet Research*, vol. 23, no. 8, e28974, Aug. 2021. DOI: 10.2196/28974. [Online]. Available: https://doi.org/10.2196/28974.

[19] M. V. Perez et al., "Large-scale assessment of a smartwatch to identify atrial fibrillation," *New England Journal of Medicine*, vol. 381, no. 20, pp. 1909–1917, 2019. DOI: 10.1056/NEJMoa1901183. [Online]. Available: https://www.nejm.org/doi/full/10.1056/NEJMoa1901183.

[20] G. H. Tison et al., "Passive detection of atrial fibrillation using a commercially available smartwatch," *JAMA Cardiology*, vol. 3, no. 5, pp. 409–416, May 2018. DOI: 10.1001/jamacardio.2018.0136.

[21] S. A. Lubitz et al., "Detection of atrial fibrillation in a large population using wearable devices: The fitbit heart study," *Circulation*, vol. 146, no. 19, pp. 1415–1424, 2022. DOI: 10.1161/CIRCULATIONAHA.122.060291. [Online]. Available: https://www.ahajournals.org/doi/abs/10.1161/CIRCULATIONAHA.122.060291.

[22] D. Duncker et al., "Smart wearables for cardiac monitoring—real-world use beyond atrial fibrillation," *Sensors (Basel)*, vol. 21, no. 7, p. 2539, Apr. 2021. DOI: 10.3390/s21072539. [Online]. Available: https://doi.org/10.3390/s21072539.

[23] M. A. Rosenberg, M. Samuel, A. Thosani, and P. J. Zimetbaum, "Use of a noninvasive continuous monitoring device in the management of atrial fibrillation: A pilot study," *Pacing and Clinical Electrophysiology*, vol. 36, no. 3, pp. 328–333, Mar. 2013, Epub 2012 Dec 13. DOI: 10.1111/pace.12053.

[24] A. N. L. Hermans et al., "Mobile health solutions for atrial fibrillation detection and management: A systematic review," *Clinical Research in Cardiology*, vol. 111, no. 5, pp. 479–491, May 2022, Epub 2021 Sep 21. DOI: 10.1007/s00392-021-01941-9. [Online]. Available: https://doi.org/10.1007/s00392-021-01941-9.

[25] L. Pay et al., "Arrhythmias beyond atrial fibrillation detection using smartwatches: A systematic review," *Anatolian Journal of Cardiology*, vol. 27, no. 3, pp. 126–131, Mar. 2023. DOI: 10.14744/AnatolJCardiol.2023.2799. [Online]. Available: https://doi.org/10.14744/AnatolJCardiol.2023.2799.

[26] Y. Lee, "Abrupt change detection of ECG by spiking neural networks: Policy-aware operating points for edge-level MI screening," *Applied Sciences*, vol. 15, no. 22, p. 12 210, 2025. DOI: 10.3390/app152212210. [Online]. Available: https://doi.org/10.3390/app152212210.

[27] N. Rashid, T. Mortlock, and M. A. Al Faruque, *Self-care: Selective fusion with context-aware low-power edge computing for stress detection*, 2022. arXiv: 2205.03974 [eess.SP]. [Online]. Available: https://arxiv.org/abs/2205.03974.

[28] A. Reiss, I. Indlekofer, and P. Schmidt, *PPG-DaLiA*, UCI Machine Learning Repository, 2019. DOI: 10.24432/C53890. Accessed: Jan. 28, 2026. [Online]. Available: https://doi.org/10.24432/C53890.

[29] P. Schmidt, A. Reiss, R. Dürichen, C. Marberger, and K. Van Laerhoven, "Introducing wesad, a multimodal dataset for wearable stress and affect detection," in *Proceedings of the 2018 ACM International Conference on Multimodal Interaction (ICMI '18)*, Boulder, CO, USA: ACM, 2018.

[30] G. B. Moody and R. G. Mark, "The impact of the mit-bih arrhythmia database," *IEEE Engineering in Medicine and Biology Magazine*, vol. 20, no. 3, pp. 45–50, May 2001, May–June 2001.

[31] A. L. Goldberger et al., "Physiobank, physiotoolkit, and physionet: Components of a new research resource for complex physiologic signals," *Circulation*, vol. 101, no. 23, e215–e220, 2000, RRID:SCR_007345. DOI: 10.1161/01.CIR.101.23.e215. [Online]. Available: https://doi.org/10.1161/01.CIR.101.23.e215.

[32] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural Networks*, vol. 113, pp. 54–71, 2019. DOI: 10.1016/j.neunet.2019.01.012.

[33] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, C. Gan, and S. Han, "On-device training under 256kb memory," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, 2022, pp. 22 941–22 954.

[34] S. Disabato and M. Roveri, "Incremental on-device tiny machine learning," in *Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things (AIChallengeIoT)*, 2020, pp. 7–13. DOI: 10.1145/3417313.3429378.

[35] U. R. Acharya, H. Fujita, O. S. Lih, Y. Hagiwara, J. H. Tan, and M. Adam, "Automated detection of arrhythmias using different intervals of tachycardia ecg segments with convolutional neural network," *Information Sciences*, vol. 405, pp. 81–90, 2017. DOI: 10.1016/j.ins.2017.04.012.

[36] A. Y. Hannun et al., "Cardiologist-level arrhythmia detection and classification in ambulatory electrocardiograms using a deep neural network," *Nature Medicine*, vol. 25, pp. 65–69, 2019. DOI: 10.1038/s41591-018-0268-3.

[37] A. Reiss, I. Indlekofer, P. Schmidt, and K. Van Laerhoven, "Deep ppg: Large-scale heart rate estimation with convolutional neural networks," *Sensors*, vol. 19, no. 14, p. 3079, 2019. DOI: 10.3390/s19143079.

[38] S. L. Oh, E. Y. K. Ng, R. S. Tan, and U. R. Acharya, "Automated diagnosis of arrhythmia using combination of CNN and LSTM techniques with variable length heart beats," *Computers in Biology and Medicine*, vol. 102, pp. 278–287, 2018. DOI: 10.1016/j.compbiomed.2018.06.002.

[39] R. Lampert, "Anger and ventricular arrhythmias," *Current Opinion in Cardiology*, vol. 29, no. 1, pp. 34–40, 2014.

Appendix A

Source Code: On-Device Adaptive Training

This appendix presents annotated source code excerpts from the on-device adaptive training system described in Section 3.8. The complete implementation comprises seven source files (∼1,200 lines of C++) in `firmware/esp32/src/training/`. Only the core components are shown here; full source is available in the project repository.

A.1 Training Configuration (`training_config.h`)

All training hyperparameters, resource thresholds, and safety constraints are governed by compile-time constants. This design eliminates runtime configuration parsing and enables the compiler to optimize branch predictions. Key values are shown below; the complete file contains additional telemetry and logging configuration.

```
namespace AdaptiveConfig {
    // Trainable layer selection (CNN backbone frozen)
    constexpr bool TRAIN_ACTIVITY_HEAD   = true;
    constexpr bool TRAIN_STRESS_HEAD     = true;
    constexpr bool TRAIN_ARRHYTHMIA_HEAD = true;

    // V5: FC1 frozen, LN+FC2+FC3 trainable = 4,984 params
    constexpr bool TRAIN_FC1_LAYERS = false;

    // Training hyperparameters
    constexpr float LEARNING_RATE      = 0.05f;
    constexpr float LEARNING_RATE_MIN  = 0.005f;
    constexpr float LR_DECAY_FACTOR    = 0.95f;
```

```
constexpr float MOMENTUM           = 0.9f;
constexpr float GRADIENT_CLIP_NORM = 1.0f;
constexpr float WEIGHT_MAX_ABS     = 10.0f;
constexpr uint8_t STEPS_PER_EPISODE = 30;

// Confidence drift detection
constexpr float CONFIDENCE_DRIFT_THRESHOLD = 0.45f;
constexpr float CONFIDENCE_EMA_ALPHA       = 0.1f;
constexpr uint16_t DRIFT_WINDOW_SIZE       = 30;
constexpr uint8_t MAX_CORRECTION_BUFFER    = 32;
constexpr uint16_t PERIODIC_TRAIN_INTERVAL = 60;

// Resource gating
constexpr bool ENABLE_RESOURCE_GATING = true;
constexpr uint32_t MIN_FREE_PSRAM_BYTES = 2*1024*1024;
constexpr uint32_t MIN_FREE_HEAP_BYTES  = 50*1024;
constexpr float MAX_TEMPERATURE_C       = 65.0f;
constexpr uint32_t MIN_TRAIN_INTERVAL_MS = 10000;
constexpr uint32_t MAX_TRAIN_DURATION_MS = 2000;
constexpr float MAX_INFERENCE_LATENCY_MS = 3000.0f;

// Validation and safety
constexpr uint8_t VALIDATION_BUFFER_SIZE    = 16;
constexpr float MIN_VALIDATION_ACCURACY     = 0.0f;
constexpr float ROLLBACK_MARGIN             = 1.0f;
constexpr uint8_t MAX_CONSECUTIVE_FAILURES  = 5;
constexpr float MAX_WEIGHT_DIVERGENCE       = 50.0f;
```

```
    // NVS flash persistence
    constexpr const char* NVS_NAMESPACE  = "adapt_weights";
    constexpr const char* NVS_VERSION_KEY = "version";
    constexpr uint16_t MAX_ADAPTATION_GENERATIONS = 100;
}
```

A.2 Gradient Engine: Forward and Backward Pass

The gradient engine performs backpropagation through each V5 three-layer head using only pre-allocated float32 buffers. Each head follows the path: FC1 (frozen) → LayerNorm → GELU → FC2 → GELU → FC3 → Softmax → Cross-Entropy loss. The forward pass caches intermediate activations for reuse during backpropagation; gradients flow back through FC3, FC2, and LayerNorm only (FC1 is frozen).

```
// GELU activation and its derivative
static inline float gelu_forward(float x) {
    float x3 = x * x * x;
    float a = SQRT_2_OVER_PI * (x + GELU_COEFF * x3);
    return 0.5f * x * (1.0f + tanhf(a));
}
static inline float gelu_derivative(float x) {
    float x2 = x * x, x3 = x2 * x;
    float a = SQRT_2_OVER_PI * (x + GELU_COEFF * x3);
    float tanh_a = tanhf(a);
    float a_prime = SQRT_2_OVER_PI
                  * (1.0f + 3.0f * GELU_COEFF * x2);
    return 0.5f * (1.0f + tanh_a)
         + 0.5f * x * (1.0f - tanh_a*tanh_a) * a_prime;
```

```
}

// Forward: FC1 -> LN -> GELU -> FC2 -> GELU -> FC3
void GradientEngine::headForward(
        TaskHeadWeights& head, const float* input) {
    // Step 1: FC1 (frozen) [input_dim] -> [hidden1_dim]
    for (int o = 0; o < head.hidden1_dim; o++) {
        float sum = head.fc1_bias[o];
        for (int i = 0; i < head.input_dim; i++)
            sum += input[i]
                 * head.fc1_weight[o * head.input_dim + i];
        head.fc1_output[o] = sum;  // cache pre-LN
    }
    // Step 2: LayerNorm [hidden1_dim] -> [hidden1_dim]
    float mean = 0.0f;
    for (int i = 0; i < head.hidden1_dim; i++)
        mean += head.fc1_output[i];
    mean /= head.hidden1_dim;
    float var = 0.0f;
    for (int i = 0; i < head.hidden1_dim; i++) {
        float d = head.fc1_output[i] - mean;
        var += d * d;
    }
    var /= head.hidden1_dim;
    float inv_std = 1.0f / sqrtf(var + LN_EPS);
    head.ln_mean = mean;
    head.ln_inv_std = inv_std;
```

```
    for (int i = 0; i < head.hidden1_dim; i++) {
        float x_hat = (head.fc1_output[i] - mean) * inv_std;
        head.ln_output[i] = x_hat;  // cache normalized
        float ln_out = head.ln_weight[i] * x_hat
                     + head.ln_bias[i];
        head.fc1_activated[i] = gelu_forward(ln_out);
    }
    // Step 3: FC2 [hidden1_dim] -> [hidden2_dim]
    for (int o = 0; o < head.hidden2_dim; o++) {
        float sum = head.fc2_bias[o];
        for (int i = 0; i < head.hidden1_dim; i++)
            sum += head.fc1_activated[i]
                 * head.fc2_weight[o * head.hidden1_dim + i];
        head.fc2_output[o] = sum;
        head.fc2_activated[o] = gelu_forward(sum);
    }
    // Step 4: FC3 [hidden2_dim] -> [output_dim]
    for (int o = 0; o < head.output_dim; o++) {
        float sum = head.fc3_bias[o];
        for (int i = 0; i < head.hidden2_dim; i++)
            sum += head.fc2_activated[i]
                 * head.fc3_weight[o * head.hidden2_dim + i];
        head.fc3_output[o] = sum;  // logits
    }
}

// Backward: CE+Softmax -> FC3 -> GELU -> FC2 -> GELU
```

```
//        -> LN -> STOP (FC1 frozen)
float GradientEngine::headBackward(
        TaskHeadWeights& head, const float* input,
        uint8_t true_label) {
    // Numerically stable softmax on FC3 logits
    float probs[16], max_l = head.fc3_output[0];
    for (int i = 1; i < head.output_dim; i++)
        if (head.fc3_output[i] > max_l)
            max_l = head.fc3_output[i];
    float sum_exp = 0.0f;
    for (int i = 0; i < head.output_dim; i++) {
        probs[i] = expf(head.fc3_output[i] - max_l);
        sum_exp += probs[i];
    }
    for (int i = 0; i < head.output_dim; i++)
        probs[i] /= sum_exp;

    float loss = -logf(fmaxf(probs[true_label], 1e-7f));

    // dL/d(logits) = softmax - one_hot
    float d_logits[16];
    for (int i = 0; i < head.output_dim; i++)
        d_logits[i] = probs[i];
    d_logits[true_label] -= 1.0f;

    // FC3 gradients
    for (int o = 0; o < head.output_dim; o++) {
```

```
    head.fc3_bias_grad[o] = d_logits[o];
    for (int i = 0; i < head.hidden2_dim; i++)
        head.fc3_weight_grad[o*head.hidden2_dim + i]
            = d_logits[o] * head.fc2_activated[i];
}
// Backprop FC3 -> d_fc2_activated
float d_fc2_act[64];
for (int i = 0; i < head.hidden2_dim; i++) {
    float s = 0.0f;
    for (int o = 0; o < head.output_dim; o++)
        s += d_logits[o]
           * head.fc3_weight[o*head.hidden2_dim + i];
    d_fc2_act[i] = s;
}
// Backprop GELU (FC2 output)
float d_fc2_out[64];
for (int i = 0; i < head.hidden2_dim; i++)
    d_fc2_out[i] = d_fc2_act[i]
                 * gelu_derivative(head.fc2_output[i]);
// FC2 gradients
for (int o = 0; o < head.hidden2_dim; o++) {
    head.fc2_bias_grad[o] = d_fc2_out[o];
    for (int i = 0; i < head.hidden1_dim; i++)
        head.fc2_weight_grad[o*head.hidden1_dim + i]
            = d_fc2_out[o] * head.fc1_activated[i];
}
// Backprop FC2 -> d_fc1_activated
```

```
    float d_fc1_act[64];
    for (int i = 0; i < head.hidden1_dim; i++) {
        float s = 0.0f;
        for (int o = 0; o < head.hidden2_dim; o++)
            s += d_fc2_out[o]
               * head.fc2_weight[o*head.hidden1_dim + i];
        d_fc1_act[i] = s;
    }
    // Backprop GELU (after LN) -> LN gradients
    float d_ln_out[64];
    for (int i = 0; i < head.hidden1_dim; i++) {
        float pre_gelu = head.ln_weight[i]
                       * head.ln_output[i] + head.ln_bias[i];
        d_ln_out[i] = d_fc1_act[i]
                    * gelu_derivative(pre_gelu);
    }
    // LN weight/bias gradients
    for (int i = 0; i < head.hidden1_dim; i++) {
        head.ln_weight_grad[i]
            = d_ln_out[i] * head.ln_output[i];
        head.ln_bias_grad[i] = d_ln_out[i];
    }
    // STOP - FC1 is frozen, no further backprop
    return loss;
}
```

A.3 SGD with Momentum and Weight Clamping

```
void GradientEngine::applyGradients(
        TaskHeadWeights& head, float lr) {
    const float mu = AdaptiveConfig::MOMENTUM;
    auto sgd_update = [lr, mu](float* weight,
            float* velocity, const float* grad, int size) {
        for (int i = 0; i < size; i++) {
            velocity[i] = mu * velocity[i] + grad[i];
            weight[i] -= lr * velocity[i];
        }
    };
    // LN (trainable)
    sgd_update(head.ln_weight, head.ln_weight_vel,
               head.ln_weight_grad, head.hidden1_dim);
    sgd_update(head.ln_bias, head.ln_bias_vel,
               head.ln_bias_grad, head.hidden1_dim);
    // FC2 (trainable)
    sgd_update(head.fc2_weight, head.fc2_weight_vel,
               head.fc2_weight_grad,
               head.hidden1_dim * head.hidden2_dim);
    sgd_update(head.fc2_bias, head.fc2_bias_vel,
               head.fc2_bias_grad, head.hidden2_dim);
    // FC3 (trainable)
    sgd_update(head.fc3_weight, head.fc3_weight_vel,
               head.fc3_weight_grad,
               head.hidden2_dim * head.output_dim);
```

```
    sgd_update(head.fc3_bias, head.fc3_bias_vel,
               head.fc3_bias_grad, head.output_dim);
    // NO FC1 updates -- frozen
}

void GradientEngine::clampWeights(TaskHeadWeights& head) {
    const float max_abs = AdaptiveConfig::WEIGHT_MAX_ABS;
    auto clamp = [max_abs](float* data, int size) {
        for (int i = 0; i < size; i++) {
            if (data[i] > max_abs) data[i] = max_abs;
            else if (data[i] < -max_abs)
                data[i] = -max_abs;
        }
    };
    // LN
    clamp(head.ln_weight, head.hidden1_dim);
    clamp(head.ln_bias, head.hidden1_dim);
    // FC2
    clamp(head.fc2_weight,
          head.hidden1_dim * head.hidden2_dim);
    clamp(head.fc2_bias, head.hidden2_dim);
    // FC3
    clamp(head.fc3_weight,
          head.hidden2_dim * head.output_dim);
    clamp(head.fc3_bias, head.output_dim);
    // FC1 not clamped -- frozen
}
```

A.4 Validation Engine: A/B Comparison

The Validation Engine evaluates the candidate model against the stable model on a ring buffer of recent labeled samples. The candidate is promoted only if it meets both a minimum absolute accuracy threshold and a relative margin criterion against the stable model on all three task heads. Weight divergence is checked first as a fast rejection path.

```
ValidationResult ValidationEngine::validate() {
    ValidationResult result = {};
    result.promoted = false;
    result.samples_used = _sample_count;

    if (_sample_count < 2) return result;  // skip

    // 1. Fast check: weight divergence
    if (checkWeightDivergence()) {
        result.weight_divergence = true;
        _consecutive_failures++;
        if (_consecutive_failures >=
            AdaptiveConfig::MAX_CONSECUTIVE_FAILURES)
            _locked_out = true;  // SAFETY LOCKOUT
        return result;
    }

    // 2. Snapshot candidate weights
    modelStore.snapshotCandidate();

    // 3. Evaluate candidate accuracy
```

```
for (int t = 0; t < 3; t++)
    result.candidate_accuracy[t]
        = evaluateAccuracy(true, t);

// 4. Load stable weights, evaluate stable accuracy
modelStore.rollback();
for (int t = 0; t < 3; t++)
    result.stable_accuracy[t]
        = evaluateAccuracy(true, t);

// 5. Restore candidate weights
modelStore.restoreCandidate();

// 6. Promotion decision
bool should_promote = true;
for (int t = 0; t < 3; t++) {
    if (result.candidate_accuracy[t] <
        result.stable_accuracy[t]
        - AdaptiveConfig::ROLLBACK_MARGIN) {
        should_promote = false; break;
    }
    if (result.candidate_accuracy[t] <
        AdaptiveConfig::MIN_VALIDATION_ACCURACY) {
        should_promote = false; break;
    }
}
```

```
    if (should_promote) {
        result.promoted = true;
        _consecutive_failures = 0;
    } else {
        _consecutive_failures++;
        if (_consecutive_failures >=
            AdaptiveConfig::MAX_CONSECUTIVE_FAILURES)
            _locked_out = true;  // SAFETY LOCKOUT
    }
    return result;
}
```

The complete source code for all seven training modules is available in the project repository at `firmware/esp32/src/training/`:

- `training_config.h` — configuration constants
- `gradient_engine.{h,cpp}` — backpropagation
- `model_store.{h,cpp}` — versioning and NVS persistence
- `feedback_controller.{h,cpp}` — drift detection
- `resource_guard.{h,cpp}` — hardware gating
- `validation_engine.{h,cpp}` — A/B comparison
- `adaptive_trainer.{h,cpp}` — orchestrator

Appendix B

Adaptation Cycle Flowchart

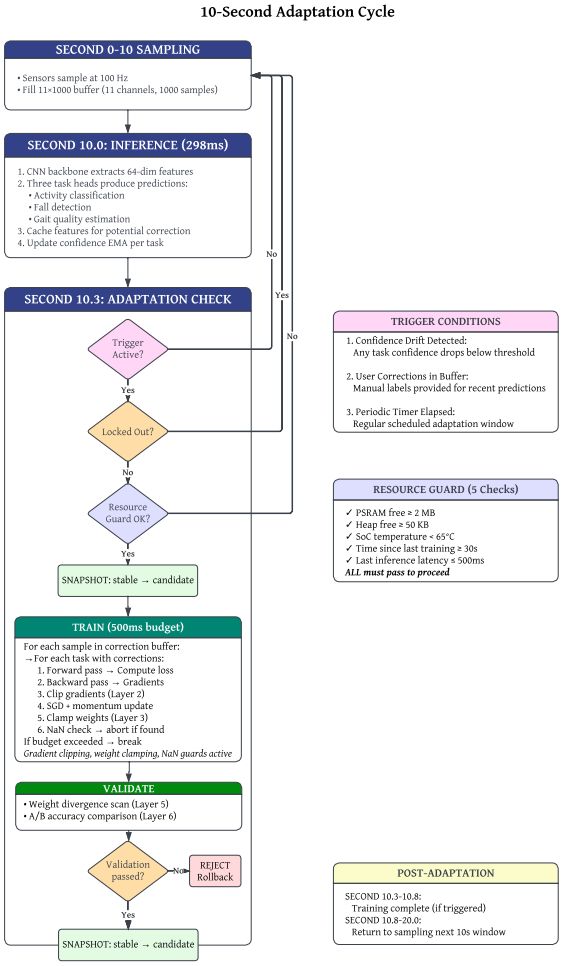

Figure B.1. Decision flowchart for the complete on-device adaptation cycle. After each 10-second inference window, the system checks for adaptation triggers, verifies resource availability, executes a bounded training episode, validates the candidate against the stable model, and either promotes or rolls back. All decision paths converge to a safe state. A detailed second-by-second timing breakdown of this cycle is provided in Section 3.8.10.

Appendix C

Summary of Key Figures and Tables

For the reader's convenience, this appendix lists the key figures and tables referenced throughout the thesis, organized by topic. A comprehensive listing is also available in the List of Figures (page xi) and List of Tables (page viii) in the front matter.

System Architecture and Hardware

- Figure 3.1: End-to-end Model V5 pipeline (overall architecture diagram)
- Figure 3.2: Detailed Model V5 blocks (CNN, SE, Transformer, and task heads)
- Figure 3.3: ESP32-S3-DevKitC-1 pin layout
- Figure 3.4: Prototype sensor wiring diagram
- Table 3.1: Unified 5-channel input tensor mapping
- Table 3.3: Model specifications (training-time vs. embedded)

Signal Conditioning and Live Visualization

- Figure 3.5: System output with no sensors attached
- Figure 3.6: System output with all sensors and signal conditioning active

On-Device Adaptive Training

Results and Evaluation

- Table 4.8: Comparison with related work
- Table 4.14: Measured training episode results on hardware
- Table 4.13: Runtime performance measurements