POLITECNICO DI TORINO

Master's degree course in Computer Engineering

Master Degree Thesis

Design and Validation of a Synchronized Heterogeneous Wireless Sensor Network: From Acquisition to Real-Time Inference

Advisors
Prof. Gianvito Urgese
Dott. Andrea Pignata

Candidate
Simone Mulazzi

December 2025

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

The spread of low-power sensors and embedded processors is turning many physical systems into continuous sources of data. Vehicles, machines, and infrastructure increasingly host accelerometers, gyroscopes, microphones, and environmental sensors connected to tiny embedded Linux computers. On top of this hardware, AI-enabled IoT applications turn raw signals into timely decisions at the edge. In these settings, the usefulness of an application depends less on the data sheet of any single sensor and more on whether the system as a whole can collect, align, and process data reliably under tight resource constraints.

Road-surface monitoring is a representative example. High-end systems for advanced driver assistance and autonomous driving often rely on Light Detection and Ranging (LiDAR) technology and camera stacks that reconstruct the road in 2-D and 3-D with high resolution [1, 2, 3]. These solutions achieve impressive accuracy but are expensive, power-hungry, and complex to integrate, and they are difficult to justify on small fleets, light vehicles, or low-cost platforms. At the same time, virtually every vehicle can host a small inertial measurement unit (IMU) and an embedded computer. If a network of such sensors can be synchronized and processed in real time, it enables a practical level of surface and motion awareness that is far more affordable and easier to deploy.

Moving from a single device to a wireless sensor network (WSN) turns timing into a central concern. Each node maintains its own clock, which drifts with temperature and time. Wireless links such as BLE introduce variable latency, packet loss, and reordering. Hubs and nodes execute multiple tasks, so queues grow and drain in ways that depend on the instantaneous workload. Without explicit timing control, motion events observed by different sensors no longer line up, and the same physical event can appear at different times and with different delays in each stream. For human inspection this misalignment may be tolerated; for machine-learning models that consume fixed windows of synchronized data, it directly translates into ambiguous or misleading inputs.

These difficulties are amplified in embedded edge deployments. Devices must respect power and cost budgets, so they cannot rely on specialized hardware timestamping or high-end networking, yet they must still respond in real time on modest platforms without dedicated accelerators. Any synchronization scheme that adds too much overhead, complexity, or fragility risks negating the benefits of using low-cost sensors in the first place.

This thesis addresses these challenges by designing, implementing, and validating an end-to-end sensing and inference stack built around a wireless sensor network and a software-based, hardware-agnostic synchronization framework. The system connects multiple inertial nodes over Bluetooth Low Energy and auxiliary sources over HTTP to an embedded Linux hub based on an STM32MP257x-class system-on-chip. The hub receives streams from all sources, maintains a shared timeline through periodic timing probes and lightweight offset-and-skew estimators, and exposes synchronized sliding windows to downstream consumers. The goal is to show that a purely software solution, running on commodity BLE adapters and an embedded hub without dedicated accelerators, can maintain millisecond-level temporal coherence, sustain real-time operation at tens of hertz, and support a concrete edge-AI workload.

A fundamental design choice in this work is to treat the wireless sensor network and its synchronization mechanisms not as a standalone infrastructure, but as the backbone of a broader edge-intelligence architecture, with road-surface classification from inertial data serving as the main case study to demonstrate its feasibility and effectiveness. The application drives the architecture: design decisions, parameter choices, and interfaces are shaped from the outset by the requirements of running classification on the hub.

The use case is deliberately built on real data. No synthetic or simulated datasets are used. All training and test data are collected manually using the implemented wireless sensor network in the field. Operators mount inertial nodes on a bicycle or vehicle, follow routes that include different surface types, and record both sensor streams and labels through the same hub and software stack used in deployment. This exposes the system to realistic operating conditions, including variations in sensor mounting, vibrations induced by pedalling and vehicle motion, wireless interference, and the natural imperfections of human annotation.

The development of the use case mirrors a typical intelligent sensing pipeline. A compact time-series model is first designed specifically for the target platform, with architecture, receptive field, and parameter count chosen so that it can run in real time on the embedded hub without monopolizing CPU or memory. A dedicated data-collection campaign then acquires inertial signals and labels on representative road segments using the wireless network in its intended configuration, and labels are tied to time intervals and road-surface types encountered in the real world. The model is trained and tuned on synchronized windows extracted from these recordings using the same synchronization engines adopted in deployment, so that any residual timing errors are consistently reflected. Finally, an inference module is implemented as a sink of the network: it subscribes to synchronized windows from the hub, executes the trained model online, and produces surface predictions in real time. This module is exercised under realistic operating conditions to evaluate end-to-end behavior, from raw sampling through transport and synchronization to model output.

Viewed from the perspective of connected sensing systems, this study examines how an application-level synchronization protocol on standard BLE stacks can keep multiple nodes sufficiently aligned for reliable inference while maintaining acceptable delivery rates when several sensors stream concurrently to a low-power hub. It also considers how temporal alignment impacts the stability and accuracy of surface predictions when both training and inference rely on manually collected data, and whether the overall architecture remains sustainable in terms of computational load, memory usage, and queue dynamics while providing synchronized windows at the required rate.

To address these goals, the thesis introduces a modular wireless sensing framework for edge workloads, in which Bluetooth, HTTP, and local producers connect to the hub through thin adapters and every record carries explicit timing metadata, including both device-side and hub-side timestamps. A single configuration file specifies sampling rates, probe periods, window parameters, and channel priorities and is shared by acquisition and processing components, ensuring that runs are reproducible and easy to replay. On top of this framework, a family of software synchronization engines is implemented: a baseline method that relies only on hub timestamps and simple buffering, a refined engine that adds offset estimation and outlier rejection, and a Kalman-based engine that models offset and skew as a two-state process driven by periodic probes. All engines emit aligned timestamps on a common time base and log timing diagnostics such as probe round-trip times, estimator innovations, and gating decisions, making timing behavior observable and enabling systematic comparison.

Building on this infrastructure, the thesis develops a complete road-surface classification pipeline: manually collected inertial and label data are converted into synchronized training and validation sets, a compact neural model is trained and evaluated, and the model is deployed as an online inference module on the hub that consumes the same synchronized windows used during training. Bench and field experiments then quantify both synchronization quality and application-level outcomes. Long-duration static runs characterize drift, jitter, and loss in the absence of motion and test the stability of the engines; controlled rotary experiments provide reference events for comparing alignment across sensors; and real cycling and driving sessions assess how alignment errors, packet loss, and queue dynamics interact with the classifier, affecting metrics such as decision latency and segment-level recall. Results are reported as distributions over entire sessions to respect temporal dependence and to reflect how the system would behave in deployment.

The remainder of this study investigates the problem along the following progression. The next chapter reviews background material on Bluetooth Low Energy communication, software-based time synchronization in wireless networks, and road-surface classification from vibration signals, placing the work in the context of existing sensing and edge-AI literature. The following chapter describes the system design in detail: the hardware platform, node and hub roles, synchronization engines, logging and windowing pipeline, and the interfaces that expose synchronized data to sinks such as the inference module and labeling tools. The results chapter presents bench and field experiments, analyzing alignment accuracy, delivery and latency statistics, computational load, and the impact of synchronization strategies on the road-surface classifier. The thesis closes with a discussion of the main findings, practical limitations, and directions for future work, including adaptive probe scheduling, richer health monitoring, and extensions to other intelligent sensing applications beyond road-surface monitoring.

Chapter 2

Background and Related Work

This chapter establishes the technical principles that underpin the design of time synchronization and data transport for heterogeneous wireless sensing on commercial, resource-constrained hardware. It focuses on Bluetooth Low Energy (BLE) as the primary motion-data transport and HTTP as an auxiliary ingress for labels and low-rate signals. It introduces the BLE stack elements relevant to sustained streaming, characterizes radio-channel variability and its impact on buffering, formalizes a device-clock model, reviews synchronization schemes for wireless sensor networks (WSN), outlines the estimators adopted in this work, and frames real-time constraints on the embedded hub. Security and integrity considerations are summarized to motivate minimal, portable controls. The goal is to provide design rules that are immediately actionable in the methods and experiments that follow.

2.1 BLE Fundamentals for Data Streaming

Bluetooth Low Energy (BLE) is a wireless communication technology in which application data are exposed through the Generic Attribute Profile (GATT). Each peripheral hosts a GATT database composed of services and characteristics identified by 16- or 128-bit UUIDs and addressed internally by 16-bit handles. A characteristic includes a declaration, a value, and optional descriptors; among these, the Client Characteristic Configuration Descriptor (CCCD) is used to enable or disable server-initiated updates.

In a typical sensing scenario, communication follows a simple sequence: connect, access the GATT database, then obtain data either via explicit reads or via notifications. After establishing a BLE connection, the central discovers the available services and characteristics and identifies those that carry sensor data. For sporadic or configuration access, it can issue read operations, where the central explicitly requests the current value of a characteristic and the peripheral returns a single response; this is simple but incurs a request–response round trip for every sample.

For continuous streaming, the central instead writes the CCCD of the selected characteristics to subscribe to server-initiated updates. From that point onward, the peripheral pushes data using Handle Value Notifications (HVNs): new samples are sent at each connection event without application-level acknowledgements, while reliability is handled by the BLE link layer through retransmissions and supervision timeout. When explicit end-to-end confirmation is required, the same mechanism can be switched to Handle Value Indications, which add an acknowledgement from the client before new data are sent. Each notification carries a limited amount of application payload, so efficient streams batch multiple samples into a single notification rather than relying on fragmentation. The timing and effective goodput of the link are governed mainly by the connection interval, peripheral latency, PHY settings, and the negotiated MTU/Data Length; the controller schedules packets within each connection event and retransmits when necessary, while the host stack preserves in-order delivery for each connection.

2.1.1 Link parameters

In Bluetooth Low Energy connections, latency and delivered throughput are primarily determined by three negotiated parameters: the *Connection Interval* (CI), the *Peripheral Latency* (PL), and the *Supervision Timeout* (STO).

Connection Interval (CI). CI denotes the spacing between connection events scheduled by the central (nominally 7.5 ms–4 s). It establishes the latency floor for fresh data, which is approximately $\frac{1}{2}$CI plus host/controller queuing and transmission time. For a burst of n notifications per event with payload B bytes each, the first-order goodput scales as

$$\text{goodput} \approx \frac{nB}{\text{CI}}\,.$$

Reducing CI lowers latency but increases radio duty cycle and exacerbates arbitration when multiple links share a single controller.

Peripheral Latency (PL) PL permits the peripheral to skip up to L consecutive events without terminating the connection. The effective service period becomes $(L{+}1)$CI, which increases queuing delay and jitter. For continuous streaming, $L = 0$ is generally mandated to avoid additional delay.

Supervision Timeout (STO). STO bounds the maximum interval without successful link-layer exchange before declaring the connection lost. It is constrained by the specification as

$$\text{STO} \;>\; 2\,(1{+}L)\,\text{CI},$$

and is typically chosen in the range 1–6 s to tolerate transient fades while ensuring timely failure detection.

Configuration guidelines In multi-link settings, CI should be chosen short enough to meet end-to-end latency targets yet sufficiently large to prevent controller congestion; PL should remain zero for streaming workloads; STO should be set with a modest margin above the lower bound to accommodate brief interference. When several links coexist on the same adapter, staggering CI values (or event phases) is recommended to mitigate systematic burst alignment.

2.1.2 MTU, Data Length Extension, and PHY

In GATT/ATT, the application payload per *Handle Value Notification* is bounded by the negotiated ATT MTU:

$$\text{payload}_{\text{HVN}} \leq \text{ATT\_MTU} - 3,$$

where 3 bytes account for the ATT opcode and 16-bit handle. The ATT MTU is agreed via the `Exchange MTU` procedure and equals the minimum of the peers' proposals (subject to OS/controller limits). With the default MTU of 23, only 20 bytes are available to the application; negotiated values in the 185–247 range substantially reduce per-sample overhead and the number of notifications required.

At the link layer, *Data Length Extension* (DLE) increases the maximum LE payload per PDU from 27 to up to 251 bytes when supported by both devices and the stack. Without DLE, an ATT PDU larger than 27 bytes is fragmented across multiple link-layer PDUs, incurring extra headers, inter-frame spacing, and retransmission exposure. With DLE, a typical configuration (ATT_MTU = 247, LL_DATA_LEN = 251) allows most notifications to fit in a single PDU, improving throughput and reducing latency variance.

2.1.3 First-Order Model of Throughput and Latency in BLE Connections

BLE traffic over a connection is organized in *connection events* spaced by the Connection Interval (CI). During each event the controller can transmit up to (n) *Handle Value Notifications* (HVN), each carrying (B) bytes of ATT application payload (with $B \leq \text{ATT}_\text{M}\text{TU} - 3$). Here, *goodput* denotes the effective application-level throughput observed at the receiver, i.e., the rate of useful payload bytes excluding all protocol headers, inter-frame spacing, and any retransmitted or lost packets. Treating (n) as the average number of successful notifications per event and neglecting controller housekeeping and retransmissions, the delivered throughput admits the first-order approximation

$$\text{goodput [bytes/s]} \approx \frac{n, B}{\text{CI}}. \tag{2.1}$$

Latency for a newly produced sample is lower-bounded by the scheduling granularity of the link. In the absence of queuing, the one-way latency is approximately

$$\text{latency} \;\approx\; \tfrac{1}{2}\text{CI} \;+\; T_{\text{tx}} \;+\; T_{\text{host}}, \tag{2.2}$$

where T_{tx} is the airtime of the HVN PDU(s) and T_{host} accounts for host/controller transfer and callback dispatch. If each notification batches k samples generated at rate f_s, serialization adds $(k-1)/f_s$ before emission, yielding

$$\text{latency} \;\approx\; \tfrac{1}{2}\text{CI} \;+\; \frac{k-1}{f_s} \;+\; T_{\text{tx}} \;+\; T_{\text{host}} \;+\; T_{\text{queue}}, \tag{2.3}$$

with T_{queue} capturing backlog under load.

In practice, inter-frame spacing, controller guard times, retransmissions, and multilink arbitration reduce the attainable n and increase variance in both (2.1) and (2.3). Two design levers follow. First, at a fixed CI, increasing the application payload B through moderate batching improves goodput, that represents The rate of useful data that successfully arrives at the destination application, by amortizing protocol headers and, when Link Layer Data Length Extension (DLE) is available, often allows a notification to fit within a single PDU. Second, decreasing CI lowers the latency floor but increases radio duty cycle and exacerbates contention when multiple connections share the same adapter. Consequently, the batch size k should be chosen to balance overhead reduction against the additional serialization delay it introduces, and CI should be selected to satisfy endtoend latency targets without inducing controller congestion in multilink operation.

2.1.4 Parameterization for Streaming and Multi-Link Operation

At a sampling frequency of $f_s = 75,\text{Hz}$, the sensor generates one sample every $T_s = 1/f_s \approx 13.33,\text{ms}$. If notifications convey a single sample, the Connection Interval must satisfy CI $\leq T_s$ so that each measurement can be transmitted in the next connection event. In practice, such short intervals are rarely sustainable on commodity BLE adapters: controller housekeeping, sporadic retransmissions, and arbitration among concurrent links reduce the number of notifications that can be completed per event, causing missed events and unstable timing. A more robust approach is to batch $k \geq 1$ consecutive samples into each notification and operate with CI in the $20-45ms$ range. This configuration retains latency compatible with real-time operation while lowering the event rate, easing contention, and improving timing regularity.

Parameter selection should be guided by payload budgeting. A six-axis IMU sampled at 16-bit resolution produces (12),bytes per sample; adding a compact fixed header of $4-8bytes$ yields roughly $16-20bytes$ per sample at the ATT layer. With $\mathrm{ATT_MTU} \geq 185$ and Link Layer Data Length Extension enabled, batching $k = 3 \ldots 6$ samples typically fits within a single ATT PDU, amortizing protocol overhead without triggering link-layer fragmentation. Increasing k further improves goodput, but it also introduces a serialization delay of $(k-1)/f_s$ before the last sample in the batch can be emitted. Therefore, k should be chosen so that this added term remains a small fraction of the end-to-end latency budget.

Latency stability depends on the link parameters discussed previously. For continuous streaming, peripheral latency should be fixed to$\mathrm{PL} = 0$ to prevent event skipping and the associated jitter. The supervision timeout must satisfy $\mathrm{STO} > 2(1 + \mathrm{PL})\mathrm{CI}$ and is typically set in the $1 - 6s$ range to tolerate short fades while preserving timely failure detection. Given CI and k, the expected one-way latency for a newly produced sample can be approximated as $\frac{1}{2}\mathrm{CI} + (k-1)/f_s + T_{\mathrm{tx}} + T_{\mathrm{host}}$, where T_{tx} accounts for on-air transmission time and T_{host} captures controller-to-host transfer and software handling.

Multi link operation introduces scheduler interactions that must be explicitly managed. When possible, peripherals should be distributed across multiple adapters to increase parallelism and reduce per adapter arbitration. On a single adapter, CI values (or event phases where supported) should be staggered to avoid systematic burst alignment across links; slight detuning of CI prevents recurrent collisions in the controller's round robin and raises the attainable average number n of notifications per event. Final sizing should be validated under representative load by verifying that queue occupancy remains bounded, that the observed notification rate matches the production rate within the chosen headroom, and that the realized latency distribution remains within the application's targets.

2.1.5 Python stack for BLE: Bleak

On the hub, this work employs *Bleak*, a cross-platform Python library for the central role that binds to native backends (BlueZ on Linux, WinRT on Windows, CoreBluetooth on macOS) using `asyncio`. Because Bleak delegates to the operating system, HCI-level controls for CI, PHY, and Data Length are not portably exposed; these are configured on the peripheral or negotiated by the stack. MTU is negotiated and may be queried but not forced in a portable manner. Notify callbacks must be minimal: timestamp arrivals on the host and enqueue payloads for downstream processing in separate tasks to avoid back-pressure on the event loop. On Linux the adapter can be selected (`hciX`) to spread links; on Windows and macOS device identity should rely on service UUIDs or manufacturer data rather than MAC addresses. In the absence of radio-layer timestamps, synchronization is achieved at the application layer by combining source-side timestamps embedded in payloads with host arrival times.

2.2 Variability of the 2.4 GHz Radio Channel

BLE shares the 2.4 GHz ISM band with Wi-Fi and other radios. Adaptive frequency hopping reduces, but does not eliminate, collisions and bursty loss. Multipath and human occlusions introduce fast fading; asymmetric airtime usage inflates round-trip variance. Short-range, line-of-sight placement and modest antenna detuning reduce tail latencies.

Host and controller stacks maintain queues. Under load, producers can temporarily outpace the link, causing queue growth, reordering, and burst loss once limits are exceeded. The hub therefore decouples notifier callbacks from processing via single-producer single-consumer or lock-free queues, bounds queue sizes with explicit drop policies that prioritize preservation of motion data over noncritical channels, and assigns host arrival timestamps and monotonic sequence numbers to detect gaps and reordering. These design choices preserve observability for the synchronization estimator while protecting downstream consumers from starvation.

2.3 Time and Device Clocks

Each peripheral maintains an independent notion of time. Let t denote true time and u the device tick counter (for example, milliseconds since boot). The device clock is modeled as an affine map with noise

$$t \;=\; \alpha\, u + \beta + \varepsilon, \tag{2.4}$$

where β is the *offset*, α the *skew* (typically expressed in parts per million relative to nominal), and ε aggregates timestamping noise and residual transport variability. Low-cost oscillators exhibit temperature-dependent skew and slow drift from aging; shocks and power events may introduce step changes. Because timestamping occurs at the application layer on commodity stacks, the system cannot rely on hardware-assisted time stamping and must instead estimate (α, β) from periodic observations in the presence of variable transport delay.

Devices may expose ticks in arbitrary units (for example, timer ticks, milliseconds, or custom epochs). Normalizing them to milliseconds simplifies reconciliation across sources. Tick wraparound and device restarts reset u and must be detected from non-monotonic tick sequences, large apparent jumps, or sequence-number gaps.

2.3.1 Client-Side Timestamp Capture and Synchronization

On the hub, each packet from a peripheral carries at least three time-related fields:

- the *device tick* u_k, a monotonic counter in device units;

- the *host receive time* t_k^{host}, taken from a monotonic clock at the earliest possible point in the notification or HTTP callback;
- an optional *remote timestamp* (for example, `remote_ms`) that encodes the device epoch in milliseconds if exposed by the firmware.

The acquisition client maintains a synchronization engine for each device. Using probe–response exchanges or paired timestamps, the engine periodically updates its estimate of offset $\hat{\beta}$ and skew $\hat{\alpha}$ for that device, following the model in (2.4). Several estimators are supported (for example, least-squares/PLL and a two-state Kalman filter); all produce an affine correction of the form

$$\hat{t}(u) \;=\; \hat{\alpha}\, u + \hat{\beta},$$

which maps device ticks into the hub-centric time base. At ingestion, each sample is enriched with this corrected time and with a synchronization status flag (for example, `UNSYNCED`, `WARMUP`, `LOCKED`) that reflects estimator confidence.

Data management at the client side follows a consistent schema. For every sample, the record written to the internal queues and to `.jsonl` logs includes:

- device identifier and sensor channel (for example, accelerometer, gyroscope);
- raw device tick u_k and optional remote epoch;
- host receive timestamp t_k^{host};
- corrected, hub-centric time $\hat{t}_k = \hat{t}(u_k)$;
- the measured values (for example, three-axis IMU readings);
- synchronization and quality flags (for example, drop markers, gap indicators).

By preserving both raw and corrected times, the system remains auditable: alignment can be recomputed offline with different estimator settings, and failure episodes can be reconstructed and inspected without re-running the acquisition.

2.3.2 Interface to the Inference Pipeline

The inference components consume only the hub-centric corrected timeline and are deliberately decoupled from the details of clock estimation. At runtime, the edge client can operate in two modes:

- **Streaming mode.** Synchronized samples are pushed into an in-memory queue (the *inference sink*), where a dedicated consumer thread assembles sliding windows according to the configured sampling frequency, window length, and stride. The consumer uses $\hat{t}_k$ to enforce uniform spacing, perform causal resampling when minor jitter is present, and trim or pad windows at the edges of gaps.

- **Offline mode.** Recorded `.jsonl` files are replayed through the same inference sink interface. The replay tool reads each record, restores its corrected time $\hat{t}_k$, and feeds it into the streaming windowing logic as if it were arriving live. This makes the inference pipeline agnostic to whether data are acquired from live BLE/HTTP connections or from logs.

In both modes, the inference stack sees a sequence of samples tagged with:

$$\Big(\text{device, sensor, } \hat{t}_k,\, \mathbf{x}_k,\, \text{flags}\Big),$$

where $\mathbf{x}_k$ denotes the sensor values. Window construction and feature extraction are driven solely by $\hat{t}_k$. Multi-device windows are formed by intersecting the corrected timelines of all participating sensors and selecting samples that fall within a common time horizon; samples outside this common interval are either dropped or linearly interpolated in a causal manner, depending on the resampling policy.

The separation of responsibilities is thus explicit:

- the *client* side (acquisition and synchronization) is responsible for capturing timestamps, estimating offset and skew, detecting anomalies such as wraparound and restarts, and annotating each sample with corrected time and quality metadata;
- the *inference* side assumes that the corrected times are valid and focuses on causal windowing, resampling within the allowed bandwidth, and feeding aligned windows to the model under fixed latency and determinism constraints.

This division allows the synchronization engine to evolve (for example, switching from LS to Kalman or changing probe schedules) without modifying the inference code, provided that the corrected timeline remains consistent and the data management interface is preserved.

2.4 IMU Signal Characteristics and Sampling

Inertial signals encode chassis vibration and motion as continuous-time processes corrupted by sensor imperfections and sampling artefacts. Let the true acceleration and angular rate be (a(t)) and $\omega(t)$. The measured discrete-time sequences at sampling period T_s are

$$\tilde{a}[k] = \mathcal{Q}!\left(s_a, a(kT_s) + b_a(t_k) + n_a(t_k)\right), \qquad \tilde{\omega}[k] = \mathcal{Q}!\left(s_\omega, \omega(kT_s) + b_\omega(t_k) + n_\omega(t_k)\right),$$

where $s.$ are scale factors, $b.(t)$ are time-varying biases, $n.(t)$ are zero-mean noises with colored spectra, and $\mathcal{Q}(\cdot)$ denotes quantization.

2.4.1 Noise, bias, and aliasing

Inertial measurements are affected by several error sources that act on different time scales. *White noise* is well modeled by a (one-sided) spectral density, e.g., S_a for accelerometers in $\mathrm{m\,s^{-2}/\sqrt{Hz}}$ and S_ω for gyroscopes in $\mathrm{rad\,s^{-1}/\sqrt{Hz}}$. Within an effective bandwidth B (set by analog front-end and/or digital filtering), the corresponding variance satisfies

$$\sigma^2 \;\approx\; S^2\, B,$$

so reducing B (or averaging) proportionally lowers white-noise variance.

Bias is a slowly varying component. Over short horizons it can be treated as a constant b; over longer horizons it follows a low-frequency stochastic process (e.g., random walk with temperature dependence), so $b(t)$ drifts with ambient and self-heating conditions. Allan analysis separates these contributions in the time domain: on a log–log plot of *Allan variance* $\sigma_A^2(\tau)$ versus cluster time τ, white noise exhibits slope τ^{-1} while bias random walk exhibits slope τ^{+1}. (Equivalently, using *Allan deviation* $\sigma_A(\tau)$, the slopes are $\tau^{-1/2}$ and $\tau^{+1/2}$, respectively.) This diagnostic guides filter tuning by distinguishing integration-limited white noise from slowly drifting bias terms.

Quantization noise arises from finite resolution. For least significant bit (LSB) size Δ, the quantization error can be modeled as uniform over $[-\Delta/2,\, \Delta/2]$ with variance

$$\sigma_{\mathrm{q}}^2 \;=\; \frac{\Delta^2}{12}.$$

Quantization is usually dominated by sensor white noise at nominal bandwidths, but becomes visible after strong averaging or aggressive low-pass filtering.

Aliasing. Sampling at rate f_s folds any input spectral content above $f_s/2$ back into baseband. In spectral terms, the sampled process has

$$S_{\mathrm{sampled}}(f) \;=\; \sum_{k\in\mathbb{Z}} S_{\mathrm{cont}}(f + k f_s),$$

so out-of-band vibration and electronic noise can increase in-band variance if not attenuated before sampling. A practical design uses an analog low-pass anti-alias filter with cutoff $f_c \ll f_s/2$ (often $f_c \approx 0.3\, f_s$), followed by digital filtering and, if needed, decimation. When downsampling by a factor N, a prefilter with stopband starting near $f_s/(2N)$ should be applied to prevent folding. These measures keep the effective white-noise density and the apparent bias stability consistent with the intended bandwidth, improving both time- and frequency-domain estimation.

Aliasing arises when signal or noise energy above Nyquist $f_{\mathrm{Nyq}} = f_s/2$ folds into baseband. Road-induced vibration and wheel-hop can exhibit content well above tens of hertz; with $f_s \approx 75, \mathrm{Hz}$ ($f_{\mathrm{Nyq}} = 37.5, \mathrm{Hz}$), anti-alias filtering is mandatory.

The continuous-time front end should implement a low-pass with passband ripple small and cutoff $f_c \lesssim 0.4 f_s$ (e.g., 25–30, Hz), followed by discrete-time filtering matched to the chosen window. If only digital filtering is available, down-weight high-frequency content with a causal IIR (biquad) or short FIR whose group delay is accounted for in the latency budget.

Mounting and gravity coupling add deterministic components. In static conditions, the accelerometer measures gravity projected onto its axes; subtracting a low-pass estimate of (g) or using magnitude $|\tilde{a}|$ can provide partial orientation invariance. Gyroscope bias dominates low-frequency drift; estimate b_ω during standstill and apply continuous bias tracking during operation (e.g., high-pass filtered gyro or a slow adaptive estimator).

Sampling jitter perturbs sample times $t_k = kT_s + \epsilon_k$. For narrowband content at f_0, small jitter ϵ_k with variance σ_ϵ^2 induces SNR loss $\mathrm{SNR} * \mathrm{jitter} \approx -20 \log *10(2\pi f_0 \sigma_\epsilon)$. Timestamp at interrupt level and use a stable clock domain to keep σ_ϵ small.

2.4.2 Windowing and alignment

Streaming inference operates on sliding windows of length W and stride S with $S \leq W$. Windowing trades variance against latency: larger W improves frequency resolution and stabilizes features, whereas larger S reduces computation but coarsens temporal granularity. With a sampling rate $f_s \approx 75, \mathrm{Hz}$, typical choices are $W = 0.8$–$1.2, \mathrm{s}$ and $S = 0.2, \mathrm{s}$. A causal taper (for example, a half-Hann window implemented causally) is applied to limit spectral leakage without using future context. For time-domain models, state is maintained across strides: LSTM hidden states are carried forward, and Conv1D layers use an overlap–save scheme so that only the new $S \cdot f_s$ samples are filtered at each step.

Alignment maps per-device sample times onto a common timeline. Let u_k denote the device tick and t_k the corresponding host time. A synchronization engine estimates offset $\hat{\theta}$ and skew $\hat{\alpha}$ such that corrected times are given by

$$\hat{t}(u) = \hat{\alpha} u + \hat{\theta}.$$

Each stream is then resampled onto a uniform grid mT_s using causal interpolation consistent with the signal bandwidth (for example, first-order hold after low-pass filtering). When small residual misalignments remain, a short-lag cross-correlation on high-SNR channels can refine alignment by up to one sample; this refinement is then frozen to avoid drift under low-SNR conditions.

Before feature extraction, channels are standardized with per-axis affine transforms (mean and variance) learned on the training set, and detrending is applied according to the task. For accelerometers, a high-pass filter at 0.5–1, Hz removes slow gravity drift, or a very low-pass filter may be used to estimate (g) for gravity compensation. Filter group delays are either linearized and explicitly compensated or included in the causal latency budget,

$$L_{\text{alg}} = W + L_{\text{filter}} + L_{\text{post}}.$$

To maintain consistent alignment across devices, all windows are trimmed to a common time horizon after synchronization so that multi-device features are indexed identically.

Feature design depends on the model class. For spectral features, Welch averaging is employed with segment lengths that divide W, overlaps chosen to achieve the desired variance reduction, and frequency bins constrained to $\leq f_{\text{Nyq}}$. For end-to-end Conv1D models, normalized time-domain windows are used directly; optional pre-emphasis can be applied to enhance mid-band content when supported by spectral analysis.

Finally, determinism is enforced throughout the pipeline. The sampling rate f_s, window length W, stride S, filter coefficients, and normalization statistics are fixed. Ring buffers are pre-allocated with capacity greater than $2W \cdot f_s$. Timestamps are attached at the earliest possible software point, and each packet carries both device and host times so that post hoc analysis can reconstruct alignment and verify that window boundaries are consistent across devices.

2.5 HTTP Ingest for Auxiliary Sources

This section details an HTTP ingest path for auxiliary signals and annotations that admits heterogeneous producers, preserves temporal coherence with BLE streams, and remains resilient to batching, reordering, and retransmissions inherent in best-effort IP.

2.5.1 Transport model and timing semantics

Producers submit application records (e.g., JSON documents) that carry two formally distinct time references. The *source time* t_{src} denotes the producer's own notion of time at event generation or sampling and may be expressed either as an absolute timestamp (e.g., UNIX epoch in milliseconds) or as a device-local tick. The *arrival time* t_{arr} is assigned by the hub at the earliest receipt callback using a monotonic host clock and captures transport- and scheduling-induced delay on the ingress path.

To place source times on the hub timeline, the synchronization engine provides an affine mapping $\hat{t} = \alpha u + \beta$, where u is a device tick or source timestamp, α represents the relative skew, and β the offset between the producer and the hub. HTTP records are aligned according to

$$t_{\text{corr}} \;=\; \alpha\, f(t_{\text{src}}) + \beta, \tag{2.5}$$

where the normalization function $f(\cdot)$ converts the producer's units and epoch to milliseconds and resolves any wraparound or clock-domain differences. Both $(t_{\mathrm{arr}}, t_{\mathrm{corr}})$ are persisted with the record: the former preserves observability of network effects, while the latter provides the corrected time used for fusion and downstream analysis, enabling auditability and late-fusion strategies when reprocessing is required.

2.5.2 Ordering, batching, and idempotency

IP transport may batch, reorder, or duplicate application messages. To deliver a consistent stream to downstream consumers, the hub enforces explicit ingress discipline with three mechanisms:

1. **Bounded reorder buffer** Incoming records are staged in a buffer keyed by source time t_{src} and released in nondecreasing order. The buffer operates over a fixed horizon T_{reorder}, selected to exceed the upper tail of producer jitter observed during warm–up (e.g., a high percentile plus margin). This confines out–of–order tolerance while bounding memory and delay.

2. **Idempotent writes** Each record carries an *idempotency key* computed from immutable fields, for example

$$\mathrm{key} = \text{SHA-256}(\langle \texttt{source\_id},\, t_{\mathrm{src}},\, \texttt{seq},\, \texttt{payload} \rangle)\,.$$

   The hub admits the first occurrence and treats subsequent submissions with the same key as no–ops, thereby eliminating duplicates induced by retries or intermediaries.

3. **Monotone sequencing** For each producer, a strictly increasing sequence counter `seq` is tracked to detect gaps and duplicates independent of wall–clock behavior. Gaps trigger health counters and, when configured, a compensation policy.

Let $t_{\mathrm{watermark}} = \max(t_{\mathrm{src}}) - T_{\mathrm{reorder}}$ denote the sliding lower bound of the reorder window. Records with $t_{\mathrm{src}} < t_{\mathrm{watermark}}$ (i.e., arriving beyond the reorder horizon) are either (i) dropped with a logged diagnostic, or (ii) routed to a *compensation path* that updates labels or metadata out of band without perturbing real–time consumers. The chosen policy is fixed in configuration to ensure reproducibility and auditability.

2.5.3 Aligning low-rate labels/events with IMU streams

Low–rate labels and events are aligned to the IMU timeline by a fixed, deterministic policy. Let $\{t_j^{\mathrm{imu}}\}$ denote the corrected IMU timestamps on the hub timebase and

$\{[t_i,\, t_i{+}W)\}$ the analysis windows with stride S and origin t_0 given by

$$t_i \;=\; t_0 + iS, \qquad i \in \mathbb{Z}_{\geq 0}, \quad 0 < S \leq W.$$

Each auxiliary record carries a corrected time t_{corr} (point event) or an interval $[a, b)$ with $a < b$ (duration label). The mapping produces, for every window i, a label set $\mathcal{L}_i$ and, when required, a single representative label ℓ_i via deterministic tie–breaking.

Point events (nearest–neighbor with tolerance) A point event at t_{corr} is assigned to the unique window $i^\star$ whose start is closest to the event,

$$i^\star \;=\; \arg\min_i \left| t_{\text{corr}} - t_i \right|,$$

provided $|t_{\text{corr}} - t_{i^\star}| \leq \Delta t$. If the tolerance is exceeded, the event is left unmatched. Ties ($t_{\text{corr}} - t_i = \pm\frac{S}{2}$) are resolved toward the lower index i to ensure determinism. Optionally, nearest–neighbor can target the nearest IMU sample t_j^{imu} instead of the window start; in that case the event inherits the window containing t_j^{imu}.

Interval labels (intersection with coverage threshold) A duration label $[a, b)$ contributes to every window whose intersection length

$$\ell_i \;=\; \Big| [a, b) \cap [t_i,\, t_i{+}W) \Big|$$

is positive. For classification tasks that require a single label per window, the representative label ℓ_i is the class with maximum overlap, subject to a minimum coverage fraction $\rho \in (0{,}1]$,

$$\ell_i \;=\; \arg\max_c \; \ell_i^{(c)} \quad \text{s.t.} \quad \frac{\ell_i^{(c)}}{W} \geq \rho,$$

where $\ell_i^{(c)}$ is the total overlap of class c with window i. If no class meets ρ, the window is marked `unlabeled`. Ties on $\ell_i^{(c)}$ are broken by earliest start time, then by lexical class order.

Snap–to–window (phase–aware assignment) When events are generated relative to an external schedule, a snap rule assigns t_{corr} to its nominal window index

$$i^\star \;=\; \left\lfloor \frac{t_{\text{corr}} - t_0}{S} \right\rfloor,$$

if and only if $|t_{\text{corr}} - t_{i^\star}| \leq \tau$. Otherwise, the event is unmatched. Boundary cases to $t_{\text{corr}} = t_i \pm \tau$ resolve toward the lower index.

Ambiguities and reproducibility When multiple labels overlap the same analysis window, assignment follows a deterministic precedence: the class with maximum coverage within the window is preferred; if coverage is equal, the label whose interval starts earlier is chosen; if a tie remains, a fixed lexical order of class identifiers resolves it. Formally, letting $\ell_i^{(c)} = \Big| [t_i,\, t_i{+}W) \cap \bigcup_{m \in \mathcal{I}_c} [a_m, b_m) \Big|$ denote the total overlap of class c with window i, the representative class is obtained by lexicographically maximizing $\Big(\ell_i^{(c)},\ -a_{\min}^{(c)},\ \mathrm{rank}(c)\Big)$, where $a_{\min}^{(c)}$ is the earliest start among the intersecting intervals of class c and rank(c) is a fixed lexical rank. For multi-label evaluation, the window label set is $\mathcal{L}_i = \{\, c \mid \ell_i^{(c)}/W \ge \rho \,\}$ with $\rho \in (0{,}1]$ a predeclared coverage threshold. All alignment parameters (Δt, τ, ρ, W, S, t_0), the rounding mode and boundary treatment, the tie-break sequence, and the nearest-neighbor target (window start versus nearest corrected IMU sample) are fixed in the run configuration to guarantee bitwise reproducibility across acquisition, training, and deployment. Operationally, given time-sorted events/intervals and window starts, a single sweep-line pass computes the mapping in $O(N{+}M)$ time for N labels and M windows, with $O(1)$ auxiliary memory beyond the active set.

2.6 Time Synchronization in WSNs

Accurate time alignment is fundamental to multi-sensor fusion, windowed feature extraction, and causal inference in wireless sensor networks (WSNs). Without a shared notion of time, streams from different nodes cannot be reliably merged, labels drift with respect to measurements, and real-time analytics become brittle. We adopt a standard affine clock model for each node i,

$$C_i(t) \;=\; \alpha_i\, t \;+\; \theta_i, \qquad \alpha_i \;=\; 1 + \rho_i, \tag{2.6}$$

where t is real time, θ_i is the offset, α_i is the relative frequency (skew), and ρ_i is the fractional drift (in ppm). The synchronization problem is to estimate (θ_i, α_i) (or equivalently (θ_i, ρ_i)) and apply a correction or time translation so that observations from multiple nodes share a common timeline.

2.6.1 Reference Methods: NTP, PTP, and FTSP

NTP (Network Time Protocol). NTP employs a two-way exchange with four timestamps $T_1, \ldots, T_4$: client send T_1, server receive T_2, server send T_3, client receive T_4. Under the (idealized) assumption of symmetric path delays, the one-way delay $\hat{\delta}$ and offset $\hat{\theta}$ are

$$\hat{\delta} \;=\; \frac{(T_2 - T_1) + (T_4 - T_3)}{2}, \qquad \hat{\theta} \;=\; \frac{(T_2 - T_1) - (T_4 - T_3)}{2}. \tag{2.7}$$

Implementations mitigate jitter by favoring the minimum observed round-trip time (RTT) over a window (to approximate the fixed component of path delay), followed by PLL/FLL-style smoothing of $\hat{\theta}$. With software timestamping, accuracy is typically sub-millisecond on stable LANs and several milliseconds on wireless links due to medium-access variability.

PTP (IEEE 1588 Precision Time Protocol) PTP decouples timestamp capture from application processing by taking hardware timestamps at the MAC/PHY for event messages (`Sync`, `Follow_Up`, `Delay_Req`, `Delay_Resp`). This sharply reduces timestamp noise. End-to-end mode estimates the delay to a grandmaster; peer-to-peer mode measures per-link delays. Boundary and transparent clocks confine or compensate switch/router residence times. On wired Ethernet with hardware timestamping, sub-μs accuracy is common; on wireless media, medium-access jitter dominates unless MAC-layer timestamping is available.

FTSP (Flooding Time Synchronization Protocol) FTSP targets low-power WSNs. A dynamically elected root floods beacons (t_{root}, h); receivers timestamp the same frame at the MAC and fit

$$C_i(t) \;\approx\; \hat{\alpha}_i\, t \;+\; \hat{\theta}_i \tag{2.8}$$

over multiple beacons to estimate offset and skew with outlier rejection. MAC-level timestamping removes most send/receive uncertainties from the software stack. Periodic re-flooding tracks slow drift, while the hop count h bounds the distance from the root. A visual comparison of NTP, PTP, and FTSP along key design dimensions is reported in Figure 2.1.

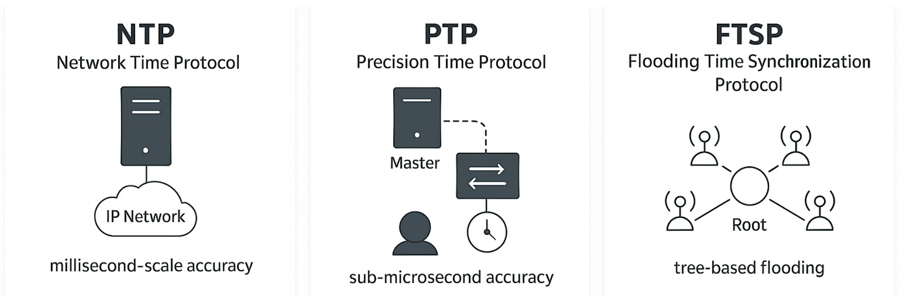


Figure 2.1: Comparison of NTP, PTP, and FTSP Time-Synchronization Protocols

2.6.2 Drift and Jitter Basics

Clock error growth. Model the local clock of node i as an affine function of true time t:

$$C_i(t) \;=\; \alpha_i\, t + \theta_i \;+\; \eta_i(t),$$

where $\alpha_i > 0$ is the (dimensionless) skew, θ_i is the offset at $t = 0$, and $\eta_i(t)$ aggregates timestamping noise. Writing $\alpha_i = 1 + \rho_i$ with ρ_i the fractional frequency error, the disagreement between two noise–free clocks i and j evolves as

$$\Delta C_{ij}(t) \;=\; C_i(t) - C_j(t) \;=\; (\alpha_i - \alpha_j)\,t \;+\; (\theta_i - \theta_j). \tag{2.9}$$

If the differential skew is $\Delta\rho = |\rho_i - \rho_j|$ expressed in parts per million (ppm), the uncompensated offset grows approximately at rate $\Delta\rho \times 10^{-6}$ seconds per second. To keep the magnitude of the disagreement below a tolerance ε, the re–synchronization interval must satisfy

$$t \;\le\; \frac{\varepsilon}{\Delta\rho \cdot 10^{-6}} \quad\Rightarrow\quad \text{e.g., } \varepsilon = 2\text{ ms},\ \Delta\rho = 30\text{ ppm} \;\Rightarrow\; t \lesssim 66.7\text{ s}. \tag{2.10}$$

Equation (2.10) makes explicit the units: when $\Delta\rho$ is given in ppm, the factor 10^{-6} converts to a dimensionless skew. In practice, $\Delta\rho$ varies with temperature, supply, and aging, so scheduling probes based on worst–case $\Delta\rho$ provides a conservative bound on drift growth.

Timestamp noise and jitter A measured timestamp $\tilde{T}$ incorporates quantization and variable delays along the software and hardware path:

$$\tilde{T} \;=\; T_{\text{true}} \;+\; q \;+\; d_{\text{tx}} \;+\; d_{\text{prop}} \;+\; d_{\text{rx}}. \tag{2.11}$$

Here q is the timestamp granularity (e.g., device tick resolution or host clock period), d_{tx} and d_{rx} capture variable queuing and scheduling latencies at the sender and receiver (interrupt service, driver, kernel–to–user delivery), and d_{prop} is the physical propagation delay. At room–scale distances, d_{prop} is negligible (~ 3.3 ns/m), so the dominant variability arises from q, d_{tx}, and d_{rx}. We refer to the variation of these terms across packets as *jitter*. Reducing q (high–resolution clocks), timestamping earlier in the stack, and minimizing code paths at the callback reduce jitter; hardware/MAC timestamping further collapses d_{tx} and d_{rx} toward their minima. For software–timestamped systems, selecting minimum observed delays over a window (see below) helps approximate the fixed component of $d_{\text{tx}}+d_{\text{rx}}$. Figure 2.2 illustrates the effect of jitter on a nominally periodic clock in the time domain.

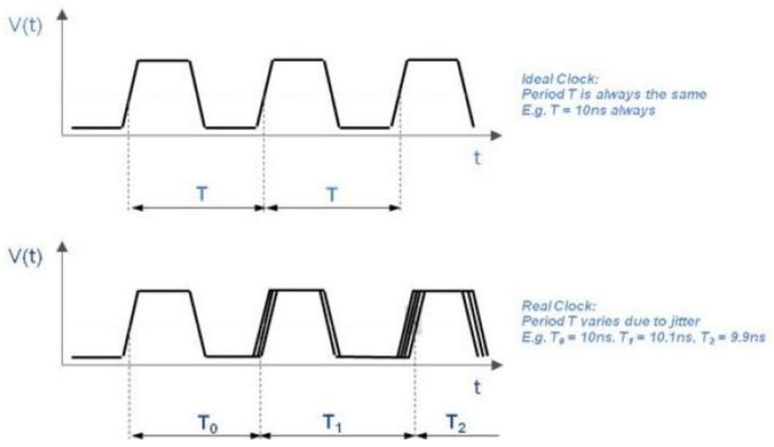

V(t)
Ideal Clock:
Period T is always the same
E.g. T = 10ns always
t
T
T
V(t)
Real Clock:
Period T varies due to jitter
E.g. T₀ = 10ns. T₁ = 10.1ns. T₂ = 9.9ns
t
T₀
T₁
T₂

Figure 2.2: A jitter-free clock as observed in the time domain, and one with a moderate amount of jitter

Two–way estimation under asymmetry Let a two–way exchange between a hub (H) and a device (D) produce time quadruples t_1 (H sends), t_2 (D receives), t_3 (D sends), t_4 (H receives). The NTP–style offset estimator is

$$\hat{\theta}_{\text{ntp}} \;=\; \frac{(t_2 - t_1) + (t_3 - t_4)}{2}, \qquad \widehat{\text{RTT}} \;=\; (t_4 - t_1) - (t_3 - t_2). \tag{2.12}$$

Assume the forward (H→D) and reverse (D→H) path delays are d_f and d_r, respectively, and let the true clock offset be θ. Then $t_2 - t_1 \approx \theta + d_f$ and $t_3 - t_4 \approx -\theta + d_r$, yielding

$$\hat{\theta}_{\text{ntp}} \;\approx\; \theta \;+\; \frac{d_f - d_r}{2} \;=\; \theta \;+\; \frac{\Delta_a}{2}, \tag{2.13}$$

where $\Delta_a = d_f - d_r$ quantifies *asymmetry*. Equation (2.13) shows that any persistent asymmetry biases $\hat{\theta}_{\text{ntp}}$ by $\Delta_a/2$, setting a floor on achievable accuracy unless compensated. In software–timestamped wireless links, queuing and scheduler effects often make d_f and d_r differ stochastically across probes.

Two pragmatic mitigations are commonly used. First, *minimum–RTT selection*: within a sliding window, retain probe pairs whose observed $\widehat{\text{RTT}}$ is near the empirical minimum; these samples are most likely to have experienced near–symmetric, minimally queued paths, reducing Δ_a. Second, *robust estimation and gating*: fit offset/skew using robust losses and reject probe innovations whose Mahalanobis distance exceeds a threshold, thereby limiting the influence of transient asymmetries. When available, per–link calibration of systematic asymmetry (e.g., controller–specific pipeline differences) further reduces the bias term in (2.13).

2.6.3 Estimator Choices: LS/PLL and Kalman

Offset and skew via LS/PLL A practical way to align a device clock to the hub timeline is to treat the device time as an affine function of hub time with two unknowns: a constant offset and a relative skew. Over horizons where temperature and operating conditions are stable, these parameters can be treated as constant and estimated with batch least squares on a sliding window of paired observations. Centering the time axis within the window improves numerical conditioning and makes the offset estimate less sensitive to roundoff. In deployments subject to occasional transport spikes, a robust loss (e.g., Huber or Tukey) with iterative reweighting is preferable to ordinary least squares because it limits the influence of outliers without sacrificing efficiency on small residuals. Total least squares is typically unnecessary because the regressor (hub time) is under software control and effectively noise free.

LS/PLL mechanics and intuition Least squares (LS) and phase-locked-loop (PLL) updates are two views of the same objective: reduce the prediction error between the observed device time and the time predicted by the current offset–skew model. LS minimizes the average squared error over a window and refreshes both parameters from scratch; PLL performs small, recursive corrections at each observation so that the model “locks” to the incoming stream. In the PLL interpretation, the instantaneous timing error plays the role of a phase error. The offset update acts as the integral path that removes steady phase error after a step change, while the skew update acts as the frequency path that cancels a constant rate error (drift). With suitable gains, the loop tracks step offsets quickly and tracks ramp-like errors due to skew with bounded residuals. If gains are too small the loop is sluggish and accumulates drift; if they are too large the loop becomes noisy or oscillatory.

Tuning Start from the probe period T_p and choose a memory half-life H for the estimator (e.g., 30–120 s) that is long enough to average out transport jitter yet short enough to follow thermal drift. This defines an effective forgetting factor; the offset gain should reflect this memory, while the skew gain should be one order of magnitude smaller so that frequency corrections evolve more slowly than phase corrections. Center or normalize the time variable within the current window to keep the two gains numerically well conditioned. Operate the loop in two regimes: a *warm-up* phase that allows faster adaptation until residuals stabilize, followed by a *cruise* phase in which the skew path is damped or temporarily held when the environment is steady. Gating large innovations (e.g., drop updates when the absolute error exceeds a configured threshold) prevents brief transport spikes from destabilizing the loop. Periodic re-initialization of the offset from a short LS fit is a low-cost anti-windup measure after long gaps or restarts.

Behavioral expectations After a device reboot or link re-establishment, the loop should eliminate a step offset in a few probe intervals, then converge the skew within the chosen half-life. During stable conditions, residual phase noise is primarily set by probe timing noise and the chosen gains; during temperature ramps or motion, residuals increase temporarily and then settle as the skew path integrates the new rate. Monitoring the error variance and the rate of rejected updates provides a simple health signal to shorten the warm-up, slow the cruise, or trigger a batch LS refresh when needed.

Offset and skew via a two-state Kalman filter When drift changes over time because of temperature swings, supply variation, or mechanical shocks, a stochastic state model yields better bias–variance trade-offs. A two-state Kalman filter treats offset and skew as the hidden state, propagates them forward with a simple constant-skew kinematic model, and corrects them with each paired observation. Two covariances govern behavior: the process noise, which encodes how quickly offset and skew are allowed to wander, and the measurement noise, which reflects the observed variability of timestamp differences after outlier rejection. Increasing process noise makes the filter adapt faster at the cost of higher residual variance; increasing measurement noise makes corrections more cautious. Time centering in the observation model improves conditioning and reduces correlation between the offset and skew estimates. During radio bursts or back-to-back losses, temporarily inflating the covariance or increasing the assumed measurement noise accelerates recovery; in steady conditions, freezing the skew or reducing its process noise lowers residual jitter. System health can be monitored with the innovation variance and the rate of rejected updates. Figure 2.3 shows the resulting two-state Kalman structure with adaptive tuning of process and measurement noise.

From estimates to corrected time Once offset and skew are available, either from LS/PLL or from the Kalman filter, device timestamps are mapped to the hub timebase by applying the current affine correction. This step must also normalize units and epochs, handle tick wraparound, and detect device restarts so that the correction is re-initialized cleanly rather than extrapolated across discontinuities. Persisting both the original arrival time at the hub and the corrected time used for fusion preserves auditability and enables reproducible reprocessing.

Assumptions, robustness, and practical guidance All estimators assume that a sufficient number of fresh, well-spaced observations is available to maintain observability; probe scheduling must therefore reserve a small but steady bandwidth even under load. Because software-timestamped wireless links experience asymmetric and variable delays, combining estimation with simple mitigations, such as selecting probes with near-minimum observed round-trip, rejecting outliers with a statistical gate, and staggering connection schedules in multi-link operation, materially improves accuracy. Time centering, unit normalization, and explicit warm-up and cruise phases are low-cost practices that reduce conditioning problems and stabilize residuals across platforms.

Offset–skew Kalman formulation For tighter tracking under variable jitter and slow drift, a two-state Kalman filter models offset and skew jointly and fuses model predictions with noisy measurements. Let

$$X_k = \begin{bmatrix} \theta_k \\ \Delta\alpha_k \end{bmatrix}, \qquad X_{k+1} = \begin{bmatrix} 1 & \Delta t_k \\ 0 & 1 \end{bmatrix} X_k + w_k, \qquad y_k = \begin{bmatrix} 1 & 0 \end{bmatrix} X_k + v_k, \tag{2.14}$$

with $w_k \sim \mathcal{N}(0, Q)$ (process noise) and $v_k \sim \mathcal{N}(0, R)$ (measurement noise). The Kalman filter alternates prediction and measurement update, producing $\hat{\theta}_k$ and $\widehat{\Delta\alpha}_k$ with quantified uncertainty [4]. In resource-constrained networks, adaptive offset–skew tracking using Kalman filtering has demonstrated rapid convergence and robustness with modest message overhead [5]. In application-level BLE synchronization, the same structure provides tight, stable alignment despite software timestamping and radio jitter [6]. The block diagram in Figure 2.3 summarizes this two-state Kalman formulation with adaptive noise tuning.

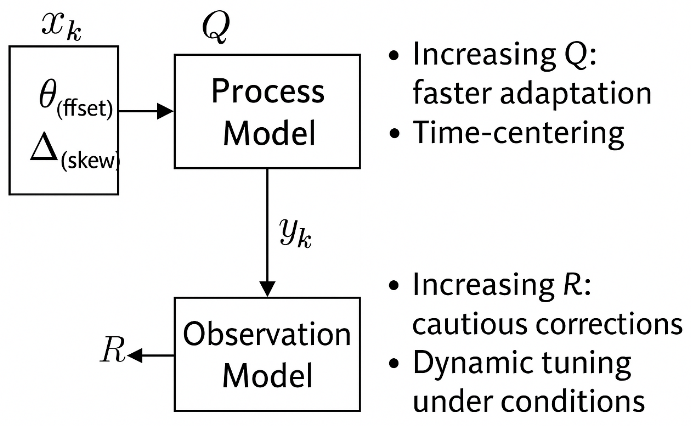

x_k
Q
θ(ffset)
Δ(skew)
Process Model
y_k
Observation Model
R
• Increasing Q: faster adaptation
• Time-centering
• Increasing R: cautious corrections
• Dynamic tuning under conditions

Figure 2.3: Two-State Kalman Filter for Clock Offset and Skew with Adaptive Noise Tuning

2.6.4 BLE WSN Specifics: Application-Level Synchronization

Commodity BLE stacks rarely expose MAC or link-layer timestamps to applications. Therefore, accurate timing must be recovered at the application layer, where arrivals are perturbed by controller scheduling, OS wake-ups, and medium-access contention. The objective of this study is to stabilize software timestamping and to bound path-asymmetry bias.

Positioning with respect to previous work Pignata et al. demonstrate a hardware-agnostic, application-level BLE synchronization framework that attains ∼10ms inter-node alignment on commodity nodes and a Linux hub, without hardware or vendor-stack modifications [6]. Over IP transports, Mani et al. present SPoT, a lightweight service that sustains ∼15ms accuracy across heterogeneous IoT platforms under noisy conditions, and highlights limitations of SNTP and MQTT on low-cost devices [7]. At the estimator level, Hamilton et al. develop a Kalman-based offset–skew tracker for constrained networks that combines fast convergence and robustness with modest message overhead [5]. Guided by these results, our engine targets BLE connection mode with standard GATT notifications and integrates bidirectional delay compensation with offset–skew filtering to improve stability and loss recovery on integrated hubs, while generalizing to heterogeneous BLE and HTTP sources.

2.6.5 Error Budget and Reporting

Net synchronization error arises from estimator residuals (least-squares, PLL-style recursions, or Kalman filtering), timestamp quantization from the software clock tick, residual path asymmetry after two-way compensation, and unmodeled drift between updates. Reporting should include the median and 95th percentile of inter-node offset, the distribution of round-trip times that supports the minimum-RTT selector, and the effective timestamp granularity. The re-synchronization interval (T) should follow the allowed misalignment ε and the worst-case drift rate $\Delta\rho$ (fraction per second), using $T \leq \varepsilon/|\Delta\rho|$. For example, with $\varepsilon = 2$ ms and $|\Delta\rho| = 30$ ppm $(30 \times 10^{-6})s/s$, $T \lesssim 67$ s; applying a safety factor of 2–4 to cover temperature ramps and residual asymmetry gives 15–30 s in practice. Shorten T when jitter or losses increase.

2.6.6 Beacon-based synchronization

Beacon-based synchronization in this system uses a hub-centric probe–response strategy. The hub periodically broadcasts a beacon at fixed intervals; upon reception, each node timestamps the event with its local clock and immediately sends a short response carrying its current timestamp. The hub timestamps the arrival of each response, so that for node d and beacon index k it obtains pairs $\left(t_k, C_k^{(d)}\right)$, where t_k is the hub time and $C_k^{(d)}$ is the device clock. At room scale the propagation delay is negligible; residual error is dominated by receive-path timestamp jitter and path asymmetry. The BLE-specific probe–response handshake implementing this scheme is sketched in Figure 2.4.

Each beacon is also associated with either a monotonically increasing counter c_k or a coarse reference time t_k^{ref}. Over a sliding window, the hub fits an affine map for each device,

$$t_k^{\mathrm{ref}} \approx \alpha^{(d)} C_k^{(d)} + \beta^{(d)} \quad \text{or equivalently} \quad t_k \approx \alpha^{(d)} C_k^{(d)} + \beta^{(d)}, \tag{2.15}$$

using robust regression or a two-state Kalman filter. Any two devices can then be aligned by composing their maps through the hub, and when the hub time is treated as authoritative all nodes are expressed on a common hub-centric time base.

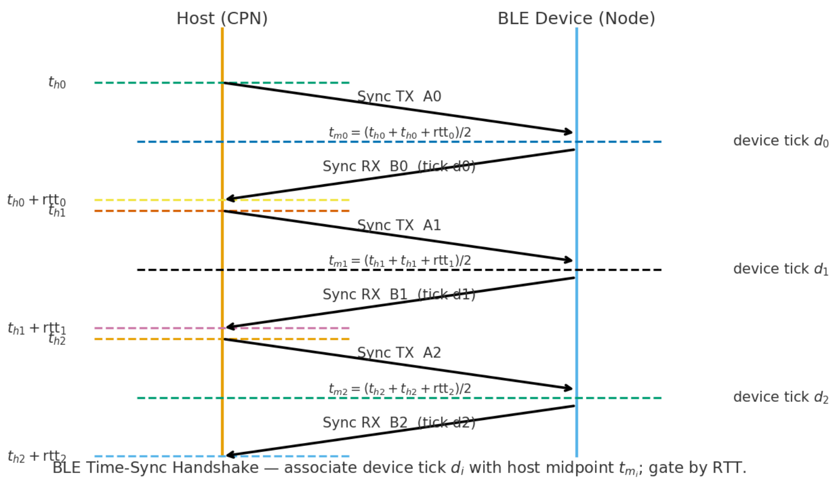

Host (CPN)
BLE Device (Node)
t_{h0}
Sync TX A0
$t_{m0} = (t_{h0} + t_{h0} + \mathrm{rtt}_0)/2$
device tick d_0
Sync RX B0 (tick d0)
$t_{h0} + \mathrm{rtt}_0$
t_{h1}
Sync TX A1
$t_{m1} = (t_{h1} + t_{h1} + \mathrm{rtt}_1)/2$
device tick d_1
Sync RX B1 (tick d1)
$t_{h1} + \mathrm{rtt}_1$
t_{h2}
Sync TX A2
$t_{m2} = (t_{h2} + t_{h2} + \mathrm{rtt}_2)/2$
device tick d_2
Sync RX B2 (tick d2)
$t_{h2} + \mathrm{rtt}_2$
BLE Time-Sync Handshake — associate device tick d_i with host midpoint t_{m_i}; gate by RTT.

Figure 2.4: BLE Beacon synchronization

2.6.7 Discussion and positioning

Time synchronization in wireless sensor networks spans a spectrum from network-assisted hardware timestamping (for example PTP) to application-level schemes that operate on commodity protocol stacks. For embedded, low-power deployments over BLE-class links, application-level designs that stabilize software timestamping, compensate for path asymmetry, and jointly estimate clock offset and skew provide a practical route to tight alignment without specialized hardware. Prior work demonstrates the feasibility of accurate synchronization at the BLE layer [6], scalable time services over IP [7], and robust estimators for constrained networks [5]; the Kalman framework offers a principled basis for combining clock models with noisy measurements [4]. This thesis builds on that body of work by adopting an application-level, two-way synchronization strategy with offset and skew filtering, and by experimentally characterizing its accuracy, stability, and operating cost under realistic wireless conditions, as summarized in Figures 2.2–2.4.

2.7 Python Threads and Concurrency

This section summarizes the concurrency model adopted in the Python sensing backends. It describes how threads interact with the CPython Global Interpreter Lock (GIL), how threads are combined with `asyncio` for I/O-bound workloads, and how bounded queues, clear ownership rules, and operating-system level tuning keep the ingest path predictable under load.

2.7.1 GIL and workload classes

In CPython a single global interpreter lock serializes bytecode execution within each process. I/O-bound operations that block in the operating system kernel, such as socket and file I/O, release the GIL; in these cases multiple threads can make progress concurrently, and threads scale well for networking, Bluetooth notifications, and logging. In contrast, pure Python CPU-bound workloads do not scale across cores under the GIL and should be moved to native extensions, vectorized numerical libraries, or separate processes when parallelism is required [8, 9]. The backend therefore reserves threads for I/O-heavy paths and relies on C or C++ (for example, LibTorch) or subprocesses for any substantial numerical work.

2.7.2 Threads with `asyncio`

Event-driven I/O is handled by `asyncio` on a single event loop that manages BLE notifications, HTTP callbacks, and timers. Threads are used for constant-time callbacks and background workers, while the loop thread itself remains as light as possible. Blocking or CPU-intensive functions are offloaded from the loop via `loop.run_in_executor` or `asyncio.to_thread`, so that the loop continues to service new events with bounded latency. Per-device state is assigned to a single owner (for example, one task or thread per device), which avoids fine-grained locking and simplifies reasoning about concurrency [10].

2.7.3 Queues and backpressure

Inter-stage handoff uses a dedicated, thread-safe backpressure queue rather than the standard `queue` classes. The custom queue is implemented as a heap-based priority structure protected by a mutex and a `Condition`, and orders records by a monotonic timestamp so that ingest preserves temporal order even when different producers interleave. Each queue is configured with a fixed capacity; when this limit is reached, producer-side calls to the non-blocking `try_push` primitive return immediately and the caller applies an explicit drop or coalescing policy, rather than blocking in the hot path. Consumers use blocking `pop` operations with a timeout, which provide natural backpressure while still allowing clean shutdown. This design caps memory usage, makes overload behaviour explicit, and keeps end-to-end latency predictable: motion channels and synchronization probes are preserved preferentially, while low-priority streams are dropped first when the backpressure queue reports sustained saturation. [11]

2.7.4 Synchronization primitives

The hot data path is designed to be effectively lock-free. Control flow relies on coarse-grained synchronization primitives only. A shared `threading.Event` signals shutdown and periodic flush points; `threading.Condition` guards phase transitions such as file rotation or configuration reload; `threading.Lock` protects short critical sections in writers. No lock is ever held across network or disk waits. A single-writer-per-structure ownership rule (for example, one writer thread per queue or log file) removes most contention and reduces the risk of deadlocks and priority inversions [8].

2.7.5 Timing and clocks

Handlers stamp arrivals with a monotonic clock using `time.perf_counter_ns`, which provides high-resolution, monotonic timestamps suitable for synchronization and latency measurement. Wall-clock time is reserved for user-facing logs and experiment metadata. Per-record sequence counters disambiguate identical timestamps and allow the system to detect gaps and reordering. A small reorder buffer corrects minor out-of-order arrivals after reconnects or short stalls, keeping the ingest stream consistent before it reaches the synchronization engine [12].

2.7.6 Operating-system level tuning

On Linux, long-lived worker threads are pinned to specific cores and run under a conservative CPU-frequency governor to reduce scheduling jitter. The function `os.sched_setaffinity` constrains threads to a subset of cores, while process priority can be adjusted with `nice`. When the platform permits, real-time scheduling policies such as `SCHED_FIFO` can be applied via `chrt` to selected threads, but this is done sparingly to avoid starving the Bluetooth host stack and other critical kernel tasks. These settings are treated as deployment-time parameters rather than assumptions of the core design.

2.7.7 Logging and failure handling

Logging uses a queue-based handler to decouple log producers from I/O. The combination of `logging.handlers.QueueHandler` and `QueueListener` preserves log ordering while ensuring that logging does not block the hot path. Threads run a supervision loop with exception guards; on error they emit a health marker, perform bounded backoff before attempting reconnects, and escalate to higher-level supervisors when recovery fails. Clean shutdown sets a shared `Event`, flushes in-flight records, rotates log files via an atomic rename, and joins worker threads with a timeout to avoid hangs during teardown [13].

2.7.8 Reference pattern

The reference pattern below illustrates a constant-time callback, a bounded priority queue, and a worker that never blocks producers:

```python
q_ingress = queue.PriorityQueue(maxsize=4096)
stop_evt  = threading.Event()

def on_notify(data: bytes) -> None:
    t_ns = time.perf_counter_ns()
    frame = decode_fixed_layout(data)
    try:
        q_ingress.put_nowait((t_ns, next_seq(), frame))
    except queue.Full:
        drop_or_coalesce(frame)

def worker():
    while not stop_evt.is_set():
        try:
            _, _, rec = q_ingress.get(timeout=0.05)
        except queue.Empty:
            continue
        process(rec)
        q_ingress.task_done()

t = threading.Thread(target=worker, daemon=True)
t.start()
```

In this pattern, `on_notify` is invoked in the BLE or HTTP callback context and performs only decoding, timestamping, and enqueueing. Backpressure is explicit: if the queue is full, a specific drop or coalescing policy is applied. The worker thread pulls records from the queue, performs downstream processing, and acknowledges completion with `task_done()`.

2.7.9 Design guidelines

Taken together, these choices yield a set of design guidelines for the sensing backends. Callbacks should remain constant time and free of blocking I/O, relying on bounded queues with explicit drop or coalescing policies to implement backpressure. Mutable state is organized under single ownership rather than shared through fine-grained locks, and locks are never held during I/O or long computations. All records are stamped with a monotonic clock and sequence numbers so that synchronization and debugging remain tractable. Long-lived workers are pinned to cores and OS scheduling is tuned conservatively when the platform permits, while logging is routed through a queue-based handler to avoid blocking the hot path. Finally, the GIL is treated as a scheduling constraint: threads are used primarily for I/O-bound work, whereas parallel CPU-bound computation is delegated to native code or separate processes.

2.8 Edge-Oriented Inference Design

This section describes the real-time inference pipeline designed for embedded Linux class platforms, with particular emphasis on architectures comparable to the STM32MP2 series. The design targets typical embedded constraints, such as limited compute, strict latency budgets, and constrained memory. In the present work, however, deployment and testing are performed on a Linux-based development system that consumes data either from recorded `.jsonl` logs or from a dedicated inference sink, rather than from a fully integrated embedded acquisition stack. The inference code is therefore written against an abstract interface: it receives already synchronized and preprocessed samples on a hub-centric time base and remains agnostic to how those samples were acquired. The same components can be ported to STM32-class hardware with minimal changes, but they have not yet been executed end-to-end on the final microprocessor target.

2.8.1 Model Architecture and Streaming Execution

The model is based on a strictly causal convolutional neural network (Conv1D). One-dimensional convolutions apply learned filters along the temporal axis of multichannel IMU input, capturing local temporal structure while preserving causality. The absence of recurrent components reduces memory usage, simplifies state handling, and improves determinism during inference, which is essential for bounded latency on embedded platforms.

The network operates on sliding windows of IMU data of approximately one second (75 frames at 75 Hz), providing sufficient temporal context for road-surface classification while retaining responsiveness. Windowing and alignment are performed upstream by the inference client: for each device and channel, it maintains a corrected timeline on a uniform grid, built from the synchronized timestamps described in Section 2.3. The client then assembles fixed-length windows according to the configured window length and stride, attaches metadata (for example, device set and synchronization status), and hands the resulting tensors to the inference engine.

To avoid recomputing the entire window at each update, convolutional outputs inside the C++ runtime are evaluated using an overlap-save strategy: past activations are cached and only the contributions from the most recent samples are computed. This incremental execution enables low-latency updates, reduces redundant computation, and aligns well with streaming acquisition.

2.8.2 Numerical and Runtime Considerations

The trained model is exported in TorchScript format and executed using LibTorch in a C++17 runtime. The inference path avoids Python entirely to eliminate interpreter overhead, garbage collection, and related jitter. When Python is used for orchestration, user interfaces, or log replay, the latency-critical stages, namely windowing, tensor preparation, and the forward pass, are delegated to a dedicated C++ worker thread pinned to an isolated core. The Python client thus acts as a producer of aligned windows, while the C++ worker performs numerical inference.

On AArch64-class CPUs, inference is compiled with aggressive optimization flags (such as `-O3` and `-ffast-math`) and uses NEON vectorization for FP32 operators. Operator fusion, constant folding, and removal of training-only components reduce memory bandwidth pressure and improve execution consistency. An optional INT8 quantization path is supported, with per-channel calibration and fallback to FP32 for operators that lack quantized implementations. This enables accuracy and latency trade-offs to be explored without changing the high-level architecture or the data management interface.

2.8.3 Data Flow, Buffering, and Back-Pressure

Data management is explicitly split between the acquisition and synchronization client and the inference runtime.

On the client side, each incoming sensor packet from BLE or HTTP is decoded, assigned a host arrival timestamp, reconciled with the device tick, and mapped to the hub-centric time base by the synchronization engine. The client writes a normalized record, typically in the form

$$(\text{device}, \text{sensor}, \hat{t}_k, \mathbf{x}_k, \text{flags}),$$

where $\hat{t}_k$ is the corrected time, $\mathbf{x}_k$ are the sensor values, and `flags` capture quality and synchronization status. These records are either appended to `.jsonl` files for offline analysis or pushed into an in-memory inference sink for live operation. Both paths share the same schema, so that a log replay tool can feed recorded data into the inference sink as if it were arriving in real time.

On the inference side, the C++ worker consumes samples from a page-aligned ring buffer backed by this sink. Feature extraction operates directly on the buffer using pre-allocated tensors, enabling zero-copy access and minimizing allocation-induced jitter. The pipeline is organized as a sequence of stages connected by fixed-capacity queues, so that back-pressure can be applied explicitly rather than relying on ad hoc blocking.

Under overload conditions, the system degrades gracefully according to policies defined at the client level. Non-critical channels are dropped first, followed by controlled coalescing of IMU samples within a bounded temporal tolerance. The inference thread is never blocked: it consumes whatever data is available at the head of the buffer and produces outputs at the configured stride. Downstream output is written using batched I/O to reduce syscall overhead and is scheduled during lower-load intervals to limit interference with acquisition and inference.

2.8.4 Fault Monitoring and Recovery

The runtime continuously monitors key metrics, including end-to-end throughput, queue occupancy, and inference latency. Watchdog mechanisms run in the client and in the inference process. On the client side, they detect stalled producers, broken BLE subscriptions, or HTTP disconnections and trigger corrective actions such as BLE resubscription or HTTP reconnection. On the inference side, they detect processing loops that exceed the latency budget and can request a controlled restart of the worker, with state snapshot and recovery where feasible.

Each sensor packet is annotated with a device tick, a host-side receive timestamp, and optionally a remote timestamp; together, these fields enable complete *post hoc* reconstruction of timing alignment and fault episodes. This instrumentation is essential both for diagnosing synchronization failures and for verifying that inference deadlines are consistently respected when data are fed through the inference sink or replayed from logs.

2.8.5 Reproducibility and Build Discipline

System images for the intended embedded deployment are built using the Yocto framework [14], with pinned versions for the kernel, BlueZ [15], LibTorch runtime [16], and dependent libraries. On the development host, startup is managed through `systemd` with explicit dependencies to ensure deterministic launch order and to avoid race conditions between acquisition, synchronization, and inference services. Configuration files are versioned alongside the binaries and included in experimental logs. Each run records commit hashes, system versions, and execution parameters, providing a traceable link between code, configuration, and reported results, regardless of whether data come from live acquisition or from `.jsonl` replays.

2.8.6 Design Envelope and Execution Profile

The pipeline targets embedded Linux class platforms with constrained CPU and memory resources. It supports multiple inertial sources at moderate sampling rates and maintains real-time operation with bounded end-to-end latency. Processing stages are arranged as a streaming graph with fixed-capacity queues, explicit back-pressure, and lightweight scheduling to keep delays within application limits.

Under nominal link conditions, the combined client and inference stack sustains reliable delivery, masks short stalls with bounded buffering, and recovers from longer disruptions without violating inference deadlines. Performance scales with available cores by instantiating additional model workers and pinning them to separate CPU threads when beneficial. On Bluetooth Low Energy transports, the acquisition layer adapts to negotiated connection parameters and packetization, while the synchronization layer maintains temporal alignment in the presence of jitter and variable interarrival times. The inference layer simply consumes the aligned stream exposed by the sink.

Resource use remains within an embedded-friendly envelope through incremental feature computation, reuse of intermediate states, and zero-copy handoffs between stages. These choices enable practical deployment on application-class cores and provide a clear path to further optimization on dedicated embedded hardware.

2.8.7 Deployment Note

Although the software stack and model design are constrained by the requirements of embedded Linux platforms, including STM32MP2-class targets, no deployment was carried out on such hardware in this work. All experiments were conducted on a general-purpose Linux host configured to approximate the computational limits of the intended target, using either live data through the inference sink or replayed `.jsonl` logs. Evaluating the combined client and inference pipeline on actual embedded platforms and quantifying the associated trade-offs in latency, power, and thermal behavior are left as directions for future work.

2.9 Related work

The related work has focused on clock synchronization for wireless sensing. Early approaches adapt network synchronization protocols to sensor networks, estimating the offset from sender-receiver timestamps and disciplining local clocks with respect to a reference. Purely software methods in this family inherit the limitations of asymmetric and time-varying delays; without MAC/PHY timestamping, their accuracy degrades in the presence of jitter and queuing [17]. Receiver-to-receiver schemes, such as reference transmission synchronization, reduce send time uncertainty by comparing the reception times of a common beacon, while flooding-based protocols stabilize a global time through periodic network-wide updates. Precision Time Protocol and gPTP improve submillisecond alignment when hardware timestamping is available, but this assumption rarely holds for low-cost embedded radios and BLE peripherals.

BLE-specific studies note that connection events and controller scheduling introduce structured delay components that affect unidirectional timestamping. When timestamping occurs in the host stack rather than the controller, variable ATT/LL latency, connection interval, and slave latency dominate the noise budget. Bidirectional timing and round-trip time filtering mitigate these effects by modeling asymmetric delay and isolating clock offset. Robust estimators that track both offset and skew outperform offset-only baselines in the presence of oscillator jitter and thermal drift; linear regression on paired ticks, PLL-style updates, and Kalman filters are common choices. The above technique also highlights the role of outlier rejection, RTT gating, and truncated windows in avoiding distortions due to retransmissions and scheduling anomalies.

From an architectural standpoint, hub-centric synchronization for single-hop star topologies remains the practical choice for BLE and mixed Bluetooth/IP implementations. Distributed consensus and multi-hop flooding are less relevant when a central hub already aggregates traffic. Multi-source alignment is then reduced to maintaining per-device clock models tied to a hub time base and reconciling device-side sampling ticks with host arrival times. Work on cross-modal fusion emphasizes that alignment must be reported with uncertainty, as downstream activities depend on trust in the reconstructed timeline.

Road surface classification Early road-surface detection systems typically relied on a single IMU or smartphone sensors with handcrafted time- and frequency-domain features and classical classifiers to detect potholes, bumps, and overall road quality. Large-scale smartphone deployments demonstrated feasibility but remained sensitive to device placement and hardware heterogeneity [18]. Deep models improved robustness by learning representations directly from raw sequences or spectrograms; convolutional architectures consistently report higher accuracy, and results show that representation choices materially affect performance [19]. Crowd- and fleet-based deployments further expanded datasets but amplified domain shift driven by speed, tire condition, and mounting; modeling the effect of speed motivates calibration and augmentation strategies to stabilize accuracy across operating conditions [20]. Multisensor and multinode configurations further improve class separability but introduce a stringent synchronization requirement: software-timestamped baselines from network time synchronization inform alignment in wireless sensor networks, yet wireless jitter and approximate timestamps limit accuracy in the absence of MAC/PHY-level timing [17]. In this thesis, road-surface classification is implemented using six IMU nodes and is intentionally framed as a demanding workload that stresses the synchronization and alignment pipeline; the classifier itself is not the primary contribution, but a vehicle to evaluate the end-to-end time-synchronized sensing and inference stack.

Classification metric This evaluation reports accuracy, balanced accuracy, macro-F1, $F1_{\text{asphalt}}$, $F1_{\text{gravel}}$, and the 2×2 confusion matrix on the same aligned time horizon used for synchronization analysis.

Chapter 3

System Architecture and Methods

3.1 Hardware architecture

3.1.1 Sensing nodes: SensorTile.box PRO

One class of sensing nodes consists of SensorTile.box PRO devices built around an STM32U5 MCU and a BLE controller (BlueNRG family). Each node integrates a 6-DoF or 9-DoF IMU (accelerometer, gyroscope, optional magnetometer) on an I^2C/SPI bus and exposes a custom BLE GATT service for real-time streaming, as illustrated in Figure 3.1.

Sensing configuration The accelerometer and gyroscope full-scale ranges are configured per session, typically selecting among ± 2, ± 4, ± 8, or $\pm 16\,\mathrm{g}$ for acceleration and ± 250, ± 500, ± 1000, or $\pm 2000\,\mathrm{deg/s}$ for angular rate, with a nominal IMU streaming frequency of 75 Hz. An on-sensor hardware low-pass filter, combined with a causal digital high-pass stage with cutoff in the $\approx$ 0.5–1 Hz range, attenuates quasi-static gravitational components and slow-varying bias on the accelerometer axes. Factory calibration is assumed valid for the duration of each session, while any residual offset is compensated during preprocessing.

Timestamps and payload Each notification carries a device tick u (monotonic millisecond counter), an optional `remote_ms` field encoding the device epoch in milliseconds, either a single sample or a short batch, and a compact header with sensor identifier and scaling metadata; payloads are packed with fixed offsets and no heap allocation to bound ISR latency, and a firmware watchdog increments a counter whenever inter-notification gaps exceed a prescribed multiple of the connection interval.

BLE transport Data are published via `Notify`, with `ATT_MTU` negotiated to 247 bytes and LE Data Length Extension enabled, yielding $S_{\text{app}} \approx 244$ usable application bytes per notification. Default link parameters target low-latency streaming: the connection interval is set in the range 7.5–30 ms (typically 15 ms), the slave latency is $L=0$, and the supervision timeout T_{sup} is chosen such that $2(L+1)\,\text{CI} < T_{\text{sup}}$; multiple packets per connection event and optional batching of 1–4 samples per notification reduce protocol overhead when several nodes are active.

Power, enclosure, health Nodes run from an internal Li-ion cell or a regulated 5 V rail (USB). Continuous streaming current is in the tens of mA and dominated by radio bursts. Enclosures are rigidly mounted; foam isolation and loose mounts are avoided. LEDs indicate advertising/connected/streaming. A UART console exposes boot logs; periodic health frames report dropped samples and queue high-water marks.

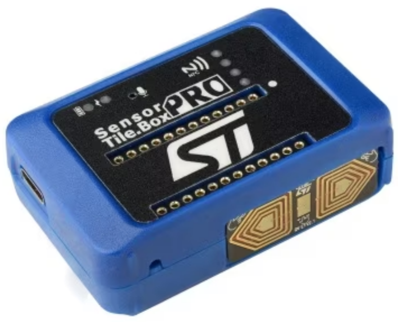


Figure 3.1: Sensors: SensorTile.box PRO

3.1.2 Sensing nodes: SensorTile.box

A legacy sensing node based on SensorTile.box (STEVAL-MKSBOX1V1) was also used in exploratory runs and selected experiments. The board integrates a 6-DoF IMU (accelerometer and gyroscope), an auxiliary ultra–low-power accelerometer, an optional 3-axis magnetometer, a barometric pressure sensor, a digital microphone, and a temperature sensor. For the purposes of this work, only the main IMU streams are enabled; the magnetometer is typically disabled during bench experiments. The hardware form factor of this legacy node is shown in Figure 3.2.

Figure 3.2: Sensing node: SensorTile.box

Differences versus PRO Compared to SensorTile.box PRO, the legacy board provides similar IMU full-scale ranges but offers lower maximum data rates on some sensor paths and relies on an earlier BLE stack and firmware baseline. On the hub, however, both devices follow the same JSONL schema; the `dev` field identifies the originating node. Sampling targets remain 75 Hz, and the BLE configuration mirrors the PRO setup (notifications, `ATT_MTU`= 247, Data Length Extension enabled, connection interval 15 ms, slave latency L=0). In practice, SensorTile.box and SensorTile.box PRO are used as sensing nodes in an equivalent manner, and all acquisition, synchronization, and inference stages treat them identically unless explicitly stated otherwise.

3.1.3 Sensing nodes: BlueTile

A third sensing node used in this work is the BlueTile development kit (STEVAL-BCN002V1B), a compact Bluetooth Low Energy board built around the BlueNRG-2 SoC. The module integrates an Arm Cortex-M0 application core, the BLE radio, and a rich set of MEMS sensors in a coin-sized form factor powered from a CR2032 cell, as illustrated in Figure 3.3. In this thesis the BlueTile node is used exclusively in the performance test bench; it does not participate in road-surface acquisition sessions.

The BlueTile sensor portfolio includes a 6-DoF inertial module (LSM6DSO accelerometer + gyroscope), a 3-axis magnetometer (LIS2MDL), a barometric pressure sensor (LPS22HH), a humidity and temperature sensor (HTS221), a Time-of-Flight distance sensor (VL53L1X), and a digital MEMS microphone (MP34DT05). For this work, only the inertial module is enabled; all other sensors are left disabled to reduce power consumption and simplify the data model. The IMU operates at the same nominal sampling rate as the other nodes (75 Hz), providing 3-axis acceleration and angular-rate measurements.

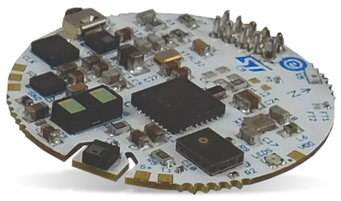


Figure 3.3: Sensing node: BlueTile (STEVAL-BCN002V1B) with on-board IMU and BLE radio

Firmware and BLE streaming The firmware running on BlueTile is derived from ST's BLE sensor examples and adapted to the custom GATT layout used in this project. The node exposes a single IMU service that periodically notifies timestamped acceleration and gyroscope samples to the central hub. Local timestamps are derived from the BlueNRG-2 system tick; they are treated as opaque device-clock values and later mapped to the global timebase by the synchronization engine. No on-board fusion or motion classification is performed on the node; all processing remains on the hub.

Differences vs SensorTile.box Compared to the legacy SensorTile.box node, BlueTile integrates the application core and BLE radio in a single BlueNRG-2 SoC and is powered from a coin cell, while SensorTile.box uses a larger STM32L4 MCU plus an external BLE module on a battery-powered enclosure. Both platforms expose similar IMU ranges and noise levels and offer a rich set of auxiliary sensors, but in this thesis only the IMU streams are used on both boards. The differing radio stacks, scheduling, and buffering make BlueTile a useful second implementation for the performance test bench, whereas SensorTile.box is primarily used in exploratory runs. From the hub's perspective, however, the JSONL schema and IMU channels are identical, and the `dev` identifier is sufficient to distinguish BlueTile from SensorTile.box within the same session and in the performance test bench.

3.1.4 Firmware, GATT profile, and packet schema

Each node exposes three primary GATT services with distinct roles: a SYNC service for beacon and probe exchanges, a DATA service for timing-sensitive sensor streams, and a CFG service for external configuration and health reporting. The firmware is derived from the STMicroelectronics reference package `FP-SNS-ALLMEMS1` [21], which provides the baseline sensing and streaming functionality; on top of this baseline, custom SYNC (and related CFG) services have been added to support both one-way beaconing and two-way probe–response synchronization. This separation keeps timekeeping independent from data transport and allows configuration to evolve without affecting real-time paths.

SYNC service The SYNC service emits periodic beacons carrying the node's monotonic tick and, optionally, a coarse epoch. The hub can initiate short probes; the node replies immediately with an echoed record so the hub measures round-trip time and bounds asymmetry. Two layouts are supported. In the two-characteristic layout, one characteristic transmits beacons and responses (`NOTIFY/INDICATE`) while a second characteristic receives hub probes (`WRITE/WRITE_NO_RSP`). In the single-characteristic layout, a single feature combines notification and write; a small opcode or role byte in the payload disambiguates `BEACON` and `PROBE_RESP`. Both layouts use the standard CCCD to enable notifications and both implement the same timing logic.

SYNC service: two characteristics versus one In the two-characteristic layout, the SYNC service exposes a transmit characteristic used for beacons and responses `NOTIFY/INDICATE` and a separate receive characteristic for hub-initiated probes `WRITE/WRITE_NO_RSP`. This split makes the directionality explicit, simplifies access control, and allows host code to subscribe and filter independently on the transmit side while treating the receive side as a pure command channel. In the single-characteristic layout, notifications and writes share one feature; a compact opcode in the payload distinguishes `BEACON`, `PROBE`, and `PROBE_RESP`. The unified design reduces attribute count and descriptor overhead and can be easier to integrate on constrained stacks, at the cost of a slightly more complex parser and less explicit separation between uplink and downlink roles. Both layouts implement identical timing logic and on-the-wire formats, so the hub-side synchronization engine operates uniformly once packets reach the application layer.

DATA service The DATA service transports IMU packets. Each notification is little-endian with fixed offsets and a compact header followed by a batch of samples to amortize per-packet overhead. The header encodes a payload version `vmaj,vmin`, a device-class flag `hw_class` for optional board-specific scaling, a stream selector `sensor_id`, a batch count `batch_n`, a fixed-point scale `scale_q`, the device monotonic counter `tick_ms`, an optional device epoch `remote_ms` set to zero when unavailable, a per-link sequence `seq` for loss and reordering detection, and an optional checksum `crc16`. The payload then contains `batch_n` consecutive 3-axis samples (x, y, z) as `int16` at the declared scale. Fixed offsets and packed fields allow zero-copy parsing on the hub, while explicit timing fields provide inputs to the application-level synchronization layer.

CFG service The CFG service handles start and stop, output-rate and range selection, and persistent settings. Configuration writes are acknowledged with indications when required by the host. Periodic health reports advertise board type, firmware version, and a boot counter so downstream tools can detect reboots and reinitialize state without schema changes.

Firmware implementation The node firmware derives from the standard ST reference stack and profiles. Data streaming and configuration follow the vendor-provided services and characteristics without modification. A single custom addition, the SYNC service, implements application-level beacons and two-way probes for time estimation. The BLE stack, controller scheduling, and MAC/LL behavior remain as in the ST baseline, ensuring compatibility and maintainability while enabling precise synchronization.

Timing guarantees With a connection interval of 15 ms and `batch_n` $\in \{1,2,4\}$, the application sustains 75 Hz per node for 5–7 nodes on a single BLE adapter, provided the notify callback remains sub-millisecond and never blocks. Backpressure is enforced via bounded queues to preserve latency guarantees under transient stalls.

3.1.5 Central Node (STM32MP2)

The central system is built around an STM32MP2-class platform (e.g., STM32MP257x) running OpenSTLinux, STMicroelectronics' Yocto-based Linux distribution. The board's dual Cortex-A35 application cores host the hub-side software stack, including multi-link BLE acquisition, time synchronization, logging and GUI services, and the on-device inference pipeline. BLE connectivity is provided by a 5.x adapter (USB dongle or on-module radio) managed through the BlueZ stack. Session logs and model artifacts are stored on local non-volatile media (eMMC or SD), while GPIOs remain available for optional hardware markers and auxiliary triggers.

Hardware–software overview of the hub The diagrams in Figures 3.4–3.6 provide a top-down, layered view of the central node and its client pipeline. At the hardware level, the STM32MP2-class SoC supplies the dual Cortex-A35 cores, an HCI interface that may be either a single internal controller or one (or more) external USB BLE dongles, network connectivity for HTTP ingress, and persistent storage via eMMC/SD. At the software level, a configuration-driven client instantiates the required producers and processing stages: BLE `SensorDevice` threads collect GATT notifications through BlueZ over the available HCI interface(s), HTTP streams are received by a single aiohttp ingest server, and file-replay producers read JSONL traces from storage. All incoming samples are timestamped at first touch on the hub monotonic clock, normalized into a unified JSONL schema, optionally passed through a bounded ingress queue for backpressure and ordering, and then time-aligned by the synchronization engines before being routed to dedicated sinks for logging, GUI monitoring, health telemetry, and real-time inference. Presenting this overview before OS-level tuning makes the subsequent process-isolation and scheduling choices easier to motivate in terms of latency, determinism, and reproducible end-to-end operation.

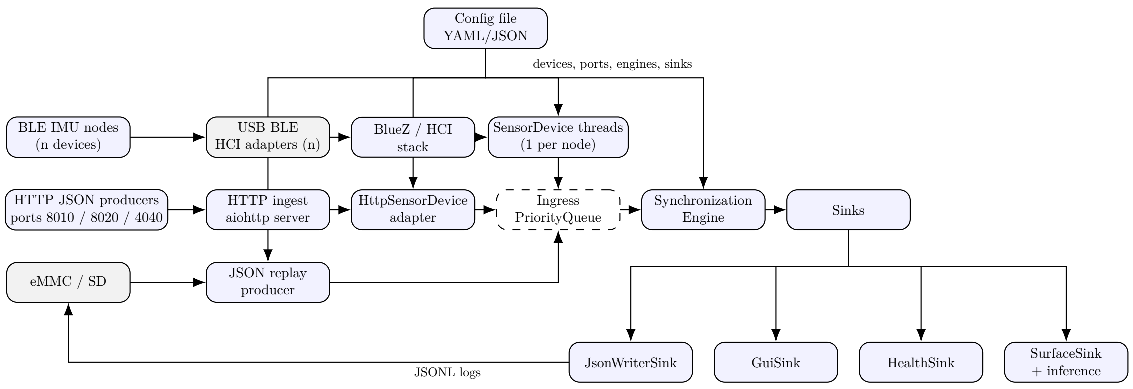


Figure 3.4: End-to-end dataflow with configuration and eMMC/SD producers.

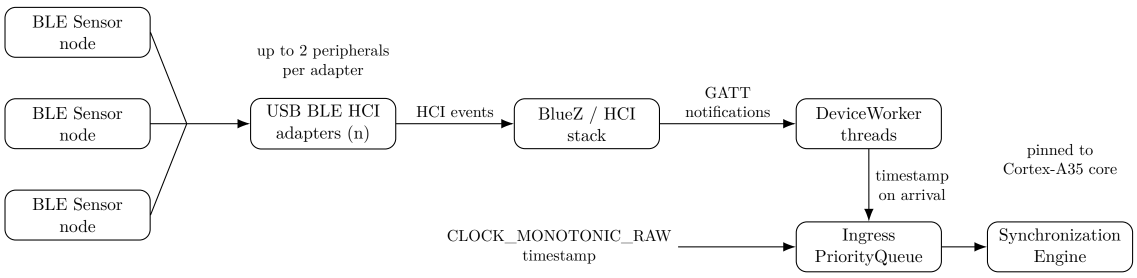


Figure 3.5: BLE ingestion: BlueZ timestamps notifications, queues them to synchronization.

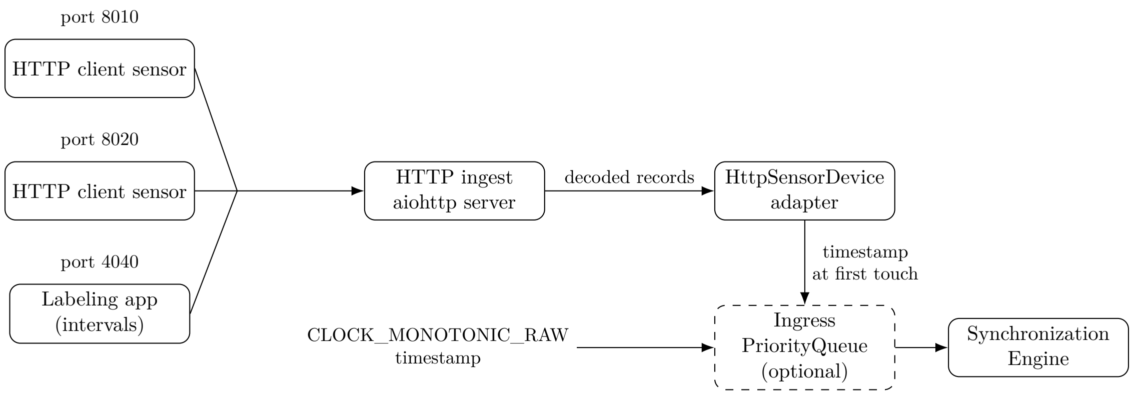


Figure 3.6: HTTP ingestion: producers send JSON over HTTP, timestamped on arrival.

Process Isolation and Scheduling The BLE ingestion service maintains active connections, timestamps notifications upon arrival using `CLOCK_MONOTONIC_RAW`, and dispatches them to the synchronization engine, as shown in Figure 3.5. Inference is executed by a dedicated long-lived thread pinned to one Cortex-A35 core, scheduled with the `SCHED_FIFO` policy to ensure real-time priority. Logging and ingestion services operate on the sibling core under `SCHED_OTHER`. The inference core uses the `performance` governor to prevent CPU frequency scaling. All time-critical memory is pre-allocated, with ring buffers and tensor pools sized at startup to avoid runtime dynamic memory allocation.

Reference Clock and Synchronization All timing is referenced to the Linux monotonic clock. The synchronization engine estimates per-node clock offset and skew, applying an affine correction $\hat{t}(u) = \hat{\alpha}u + \hat{\theta}$ to align sample times. Optional `MARK` events, triggered via software or GPIO pulses, serve as global anchors and are logged periodically for traceability. The interaction between ingestion, timestamping, and synchronization for HTTP-based producers is summarized in Figure 3.6.

I/O handling and telemetry Sensor data and labels are written in append-only JSONL files with bounded in-memory queues, so that logging remains consistent even under sustained load. System telemetry, covering throughput, inter-arrival time statistics, queue depth, and inference latency, is sampled at a fixed rate of 1 Hz and recorded alongside the data streams, providing a continuous view of runtime behaviour. At the start of each session the client also records firmware revisions, kernel and BlueZ versions, and the LibTorch commit hash in use, so that acquisition conditions and model binaries can be reconstructed exactly for reproducibility. The overall dataflow from devices and HTTP producers to sinks and storage is captured in Figure 3.4.

3.2 Software architecture

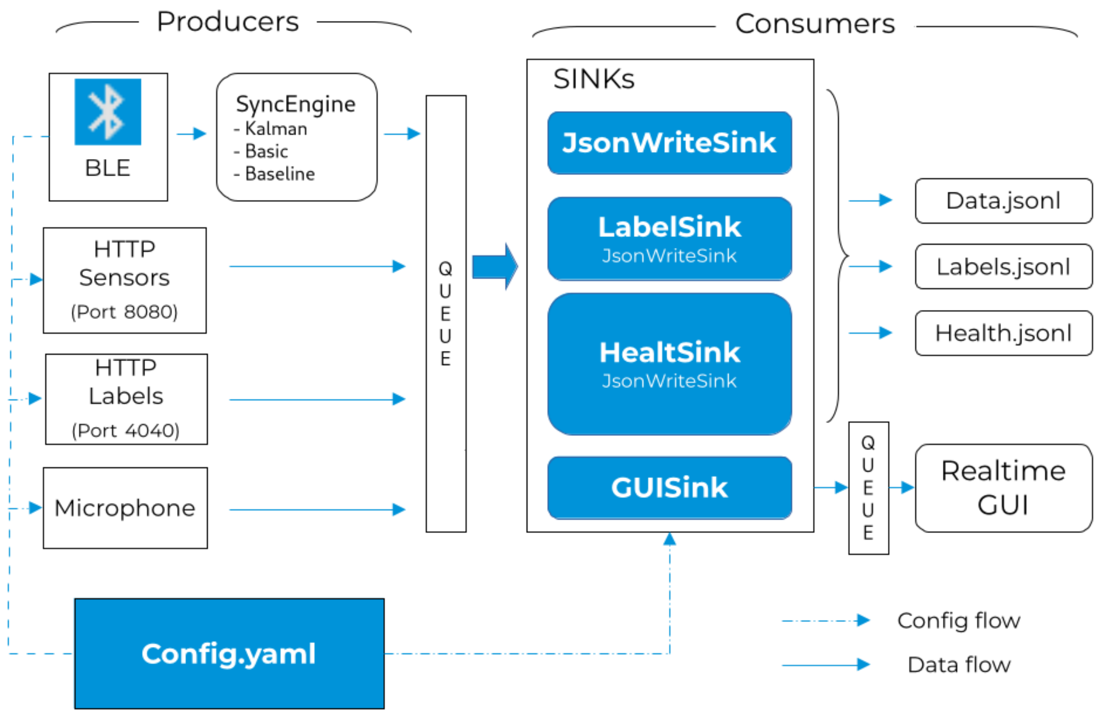


Figure 3.7: Software architecture from BLE and HTTP producers to consumers

3.2.1 End-to-end pipeline and data flow Acquisition

A `DeviceWorker` thread runs for each BLE node, subscribes to the IMU data characteristic, and decodes notifications in the RX callback using Bleak on top of BlueZ. Devices are declared in a profile YAML file; their random addresses are resolved at session start, and each device is bound to a chosen HCI interface. The worker enables notifications via CCCD writes, timestamps each packet at first touch with a monotonic `host_ms`, normalizes the payload to the unified schema, and pushes the resulting records to the ingress queue. In parallel, an HTTP ingestor listens on `0.0.0.0:8000`, accepts schema compatible JSON from auxiliary producers and from the labeling app with CORS enabled, and applies a lightweight exponential moving average offset estimator (with $\alpha = 0.2$) when source timestamps exhibit a persistent bias, before forwarding records to the synchronization stage.

Timestamping and synchronization The active BLE time-synchronization protocol is implemented through a dedicated GATT synchronization service and a single probe characteristic, listed in Table 3.1. For each peripheral, the *Synchronization Engine* discovers this service and uses the probe characteristic for a lightweight two-way exchange that is kept separate from the IMU streaming characteristic. At each probe, the hub writes a request on the probe characteristic and receives an indication/notification carrying the peripheral tick counter. These (u, `host_ms`) anchor pairs are then used to estimate offset and skew on the hub timebase, while the data path proceeds independently. Isolating probing into its own service makes synchronization explicit, configurable per device, and robust to changes in the sensor payload format.

Table 3.1: UUIDs used by the BLE synchronization probe protocol

Element	**UUID**	**Role**
Synchronization service	`00000000-000f-11e1-ac36-0002a5d5c51b`	Application-level GATT service hosting the probe logic used by the Synchronization Engine.
Sync probe characteristic (write/indicate)	`00000002-000f-11e1-ac36-0002a5d5c51b`	Characteristic within the synchronization service.

The Kalman 2 state configuration uses process noise $kf_{Qa} = 1 \times 10^{-12}$ and $kf_{Qb} = 1 \times 10^{-3}$, initial covariance $kf_{P0_a} = 1 \times 10^{-8}$ and $kf_{P0_b} = 1 \times 10^{6}$, and a Mahalanobis gating parameter $\lambda = 9.0$. Warm up exit is controlled by `warmup_min_sec` $\in (600, 900)$ depending on the device and by ppm thresholds: the mean skew must be below 0.12 ppm and its standard deviation below 0.06 ppm. The engine then requires `cruise_entry_windows` = 10 consecutive windows with `cruise_entry_ppm` = 1.5 and `cruise_entry_sigma_ppm` = 1.0 to commit to cruise mode. Offset jitter control freezes the published offset b after warm up if residuals remain small, with median residual at most 1.5 ms and 95th percentile at most 4.0 ms for at least three anchors in a row. Wrap and reconnect behaviour follow `soft_resume_max_gap_ms = 120000` and `soft_resume_max_residual_ms` = 15.0 for soft resumes, and use `hard_reset_on_wrap` = `true` with hard_reset_max_gap = 300000 to trigger a full reset when counters wrap or gaps become too large.

Routing and feature windowing A Router places records on per-sensor ring buffers. The *FeatureBuilder* maintains sliding windows of length W and stride S, enforces per-device completeness ρ, and builds tensors of shape $(T,\ C \cdot D + M)$ with one mask bit per device. Z-scores are computed on the training split and reused unchanged. The default surface model sink (below) sets $f_s = 75$ Hz, $W = 1.0$ s, jitter tolerance 35 ms, maximum gap 200 ms, and `min_devices` = 5.

Inference A *ModelRunner* executes a TorchScript Conv1D with depthwise-separable blocks and dilations 1,2,4. Posteriors are optionally smoothed with an EMA $\alpha = 0.4$ and a dwell of 10 windows.

Sinks and handoff semantics As shown in Fig. 3.7, all records produced by BLE, HTTP, and file-replay sources converge into a common stream and are then fanned out to specialized consumers. `JsonWriterSink` provides the authoritative, append-only persistence layer, ensuring that each run yields immutable JSONL artifacts. `GuiSink` decimates and forwards best-effort updates for live visualization without ever back-pressuring acquisition. `HealthSink` continuously aggregates QoS indicators—rate, gaps, jitter, RTT, and drop statistics—so that stream quality can be monitored online. `LabelingSink` filters label events, aligns them to the hub timeline, and emits compact interval logs for training and evaluation. Finally, `SurfaceSink` implements the real-time ML workload: it rebuilds multi-device feature windows from synchronized samples, runs the TorchScript model, and forwards posteriors and predictions to a dedicated writer sink, keeping inference outputs separate from raw data logs.

All inter-stage handoffs follow the same low-latency design principles used along the producer side of Fig. 3.7. Whenever possible, components communicate through single-producer/single-consumer lock-free queues to avoid contention and priority inversion. The bounded `PriorityQueue` used in the ingest/multiplexing path orders items by `host_ms` and breaks ties with a monotonic sequence counter, guaranteeing deterministic replay under small scheduling jitter. When the queue reaches capacity, it replaces the worst-priority element if the incoming one is better or equal; for equal priority, the newest sample is kept and the oldest is discarded, so freshness is preserved during transient overload. Non-critical branches (GUI, auxiliary channels) are the first to drop, while IMU streams may be lightly coalesced within a small tolerance without violating causal window construction. Any RF-interference generation for stress testing is performed externally and never executes on the hub, so it does not perturb the timing guarantees of the acquisition and sink pipeline.

BLE payload schema IMU streaming is exposed through a single GATT service and characteristic, summarized in Table 3.2. The client enables notifications by writing the CCCD with the value shown in the table, after which each packet is delivered as a fixed-layout binary notification. As detailed in Table 3.3, the first two bytes carry the device tick timestamp, followed by nine signed 16-bit samples for accelerometer, gyroscope, and magnetometer axes. All sensor fields are transmitted as little-endian `int16` and converted on the hub by applying the firmware scaling factor of 10^{-3} to obtain physical units. For every notification, the hub injects the arrival timestamp `host_ms` and computes the aligned `timestamp_ms` through the per-device synchronization mapping $\hat{t}(\cdot)$. HTTP producers deliver JSON records already matching the unified schema, so they enter the pipeline with the same field names and downstream handling.

Table 3.2: GATT endpoints used for IMU streaming.

Element	**UUID**	**CCCD handle**	**Enable value**	**Role**
IMU service	`00000000-0001-11e1-9ab4-0002a5d5c51b`	–	–	Service hosting the inertial data characteristic.
IMU data characteristic	`00e00000-0001-11e1-ac36-0002a5d5c51b`	32	`"0100"`	Notifies one IMU sample per connection event.

Table 3.3: Layout of one IMU notification (little-endian).

Field	**Byte indexes**	**Type**	**Scale**	**Notes**
tick timestamp	[0,1]	`uint16`	1 tick = 1 clock cycle	Unwrapped on hub.
accX	[2,3]	`int16`	$\times 10^{-3}$	Accelerometer axis.
accY	[4,5]	`int16`	$\times 10^{-3}$	
accZ	[6,7]	`int16`	$\times 10^{-3}$	
gyrX	[8,9]	`int16`	$\times 10^{-3}$	Gyroscope axis.
gyrY	[10,11]	`int16`	$\times 10^{-3}$	
gyrZ	[12,13]	`int16`	$\times 10^{-3}$	
magX	[14,15]	`int16`	$\times 10^{-3}$	Magnetometer axis.
magY	[16,17]	`int16`	$\times 10^{-3}$	
magZ	[18,19]	`int16`	$\times 10^{-3}$	

3.2.2 BLE client stack (threads and PriorityQueue)

The Python collector in `client.py` instantiates one thread per device or service and connects them through a small number of queues. BLE and HTTP devices generate JSON samples, an optional microphone pipeline produces audio features, and a set of threaded sinks consume the unified stream. All of these components and names match the implementation in the `Thesis-SurfaceDetection` project.

SensorDevice threads (BLE) Each BLE node is handled by a `SensorDevice` instance from `sensor_device.py`, which is a subclass of `threading.Thread`. Configuration is loaded from YAML or JSON and provides the logical name, MAC address or random BLE address, HCI interface, MTU, and the mapping from characteristics to sensors and fields. The thread scans for its target device using `BleakScanner`, opens a `BleakClient`, discovers services and characteristics, and subscribes to the IMU data characteristic by writing the CCCD. Notifications are decoded in the Bleak RX callback, where the device tick counter is unwrapped, mapped to host time through the per device `SynchronizationEngine`, and packed into a JSON compatible dictionary with fields `dev`, `sensor`, `timestamp_ms`, `values`, `raw_sensor_time`, `raw_host_time`, `raw_counter_unwrapped`, `remote_ms` , `delta_vs_remote_ms`, `delta_vs_host_now_ms`, and `timestamp_source`. This dictionary is passed to a user supplied consumer callback via `self._consumer(sample)`. Connection management, including reconnects with backoff, service rediscovery, and resubscription of notifications, is implemented inside `SensorDevice` and follows the logic in `run()`, without any extra dispatcher thread in between.

HttpSensorDevice and LabelServer (HTTP sensors and labels) HTTP producers are handled by `HttpSensorDevice` in `http_sensor_device.py`, which is also a `threading.Thread`. It starts an `aiohttp` based server that accepts JSON payloads on configured endpoints and maps device time to host time using either an exponential moving average offset estimator or a slot based minimum strategy, as implemented by the `map_ema` and `map_stream_min` helpers. For each valid record it builds a sample with the same shape as BLE samples, including `sensor_raw`, and forwards it to the shared consumer callback. Labels from the smartphone app use a separate `LabelServer` from `http_labeling.py`. `LabelServer` runs an `aiohttp` application in a background thread, accepts only `{"label": ...}` requests, wraps them into `{"type":"label","label":<value>}` records, and either enqueues them into a `PriorityQueue` or forwards them directly to the global consumer. In the main client configuration used for this work, `LabelServer.set_consumer(consumer)` is used, so label records are injected into the same stream that feeds the sinks, and `LabelingSink` selects only `type='label'` entries when writing `labels.jsonl`.

MicrophoneRecorder and mic PriorityQueue Audio capture is handled by `MicrophoneRecorder` in `microphone_recorder.py`, which uses FFmpeg and ALSA to read mono PCM audio at a fixed rate. The recorder pushes tuples
(t_{chunk}, payload, "*mic*") into a bounded `PriorityQueue` from `Priority_queue.py`, where the numeric priority is the chunk timestamp. The helper
start_microphone_pipeline in `client.py` starts `MicrophoneRecorder`, allocates a `PriorityQueue` with a fixed `maxsize`, and spawns a small forwarder thread that repeatedly pops from the queue and converts each payload into a JSON sample with fields
`dev:'mic'`, `sensor:'MIC'`, `timestamp_ms`, `raw_host_time`, `values`,
and `timestamp_source:"host"`. When the `mic_embed_audio` option is active, the forwarder also attaches `pcm_b64` and `sha1` inside `values`, exactly as in the implementation. The microphone samples are then passed to the same consumer callback as BLE and HTTP samples, so they are logged and visualised as another device on the synchronized axis.

Global consumer, sinks, and GUI The function `make_consumer` in `client.py` builds the global consumer used by all devices. This consumer iterates over the active sink instances and calls `s.consume(sample)` on those that accept raw data, then forwards the same sample to the optional GUI queue. Sinks are instances of classes from the `sinks` package such as `JsonWriterSink`, `LabelingSink`, `HealthSink`, and `SurfaceSink`. Each sink inherits from `BaseSink`, which provides a dedicated thread and a bounded internal deque, plus backpressure policies named `"drop_new"`, `"drop_old"`, and `"block"`. The sinks consume samples asynchronously in their own threads by overriding `on_sample()`, so the ingest path from the device threads to the consumer is non blocking. The GUI, when enabled, is started via `start_gui` from `gui_display.py` and reads from its own thread safe queue `threads_queue`, receiving the same JSON dictionaries that are written to disk.

PriorityQueue semantics The custom PriorityQueue implementation in Priority_queue.py is used for microphone frames and, in standalone mode, for label records. It provides a min heap where lower numeric priority is considered better, supports an optional `maxsize`, and replaces or drops items according to priority when the queue is full, matching the behaviour documented in the source. The code paths in `client.py` and `http_labeling.py` use this queue exactly as intended, with tuples $(\text{priority}, \text{payload}, \text{src})$ and non blocking `get_nowait()` in the forwarder loops, so that audio and labels can be integrated into the main stream without blocking the device threads.

3.2.3 Configurable sinks

Sinks are implemented as independent consumers derived from a common `BaseSink` class in `sinks/BaseSink.py`. `BaseSink` manages a dedicated daemon thread and a bounded internal deque, with configurable backpressure policies `"drop_new"`, `"drop_old"`, and `"block"`. The constructor

```
BaseSink(
  name: str,
  max_queue: int = 10000,
  backpressure: str = "drop_new",
  tick_every_ms: Optional[int] = None,
  rate_limit_ms: Optional[
    Union[int, Callable[[Dict[str, Any]], Tuple[int, str]]]
  ] = None,
)
```

sets the sink name, queue size, and backpressure mode, and optionally enables periodic `on_tick(now_ms)` callbacks and per-sample or per-key rate limiting. The `start()` method spawns the internal thread and begins draining the queue in `_run()`, which repeatedly calls `on_sample(sample)` and `on_tick(now_ms)`. The `consume(sample)` method is non-blocking and enforces the chosen backpressure policy: `"drop_new"` discards incoming samples when the queue is full, `"drop_old"` drops the oldest entries to make room for new ones, and `"block"` waits on a condition variable. All threaded sinks below use this mechanism.

JsonWriterSink `JsonWriterSink` in `sinks/JsonWriterSink.py` is the primary persistence backend. It subclasses `BaseSink` with the constructor

```
JsonWriterSink(
  path: str,
  rotate_by_size_mb: int = 0,
  rotate_every_s: int = 0,
  flush_every: int = 200,
  pretty: bool = False,
  only_types: Optional[Iterable[str]] = None,
  max_queue: int = 20000,
  backpressure: str = "drop_old",
  **_ignored,
)
```

and internally sets `tick_ms = 1000` when time based rotation is enabled so that `on_tick` can check for rotation once per second. Records are written as UTF-8 JSON lines to a template path `path`; rotation is controlled by `rotate_by_size_mb` and `rotate_every_s`. When a threshold is exceeded, `_rotate_locked()` closes the current file and opens a new one with a numeric suffix. The parameter `flush_every` specifies how many records are buffered before a forced `flush()`, balancing throughput and durability. The optional `only_types` filter restricts output to records with `rec["type"]` in the given set, otherwise all samples are written. In the configurations used for this work, typical parameters are `rotate_by_size_mb = 2048`, optional `rotate_every_s`, `flush_every = 100`, `max_queue = 5000`, and `backpressure = "drop_old"`, so that the newest data are kept if producers briefly outrun disk writes.

GuiSink `GuiSink` in `sinks/GUISink.py` is a high performance sink for forwarding data to the live GUI. It also inherits from `BaseSink` and is constructed as

```
GuiSink(
  gui_put: Callable[[Minimal], None],
  min_interval_ms: int = 100,
  target_total_hz: int = 120,
  flush_ms: int = 50,
  max_queue: int = 5000,
  backpressure: str = "drop_old",
  transform_fn: Optional[TransformFn] = None,
  *,
  get_queue_depth: Optional[QueueDepthFn] = None,
  max_gui_queue: int = 500,
  max_keys: int = 1024,
  idle_forget_ms: int = 120_000,
)
```

Here `gui_put` is a callback that enqueues minimal payloads into the GUI queue, `min_interval_ms` enforces a minimum spacing per key, and `target_total_hz` with `flush_ms` implements a budget of how many points can be forwarded per flush. The optional `transform_fn` maps full samples into a minimal dictionary, by default extracting `dev`, `sensor`, `timestamp_ms`, and a numeric value from `values`. The sink coalesces updates by key (`dev`, `sensor`), keeps only the most recent sample per key in an internal map, and then schedules a bounded number of keys per flush, so the GUI sees a representative but decimated stream. The optional `get_queue_depth` and `max_gui_queue` parameters make the sink aware of the GUI's own queue depth; when the GUI is backlogged, the sink drops more aggressively. The configuration used in the road surface experiments sets `min_interval_ms` ≈ 100, `max_queue` = 2000, and `backpressure = "drop_old"`, so that visualization never blocks the ingest path.

HealthSink. `HealthSink` in `sinks/HealthSink.py` analyses stream quality per device and sensor and emits periodic summaries. It is defined as

```
HealthSink(
  window_s: int = 10,
  emit_every_ms: int = 2000,
  expected_hz: Optional[Dict[str, float]] = None,
  value_ranges: Optional[Dict[str, Dict[str, Tuple[Number, Number]]]] = None,
  jitter_ms_threshold: float = 1.0,
  dropout_pct_threshold: float = 2.0,
  flatline_var_epsilon: float = 1e-6,
  clip_epsilon: float = 0.0,
  gravity_g: float = 1.0,
  score_weights: Optional[Dict[str, float]] = None,
  emit_put: Optional[Callable[[Dict[str, Any]], None]] = None,
  out_path: Optional[str] = None,
  rotate_by_size_mb: int = 0,
  rotate_every_s: int = 0,
  flush_every: int = 200,
  pretty: bool = False,
  **kw,
)
```

and passes `emit_every_ms` to `BaseSink` as `tick_every_ms`, so that `on_tick` fires at the desired cadence. The internal window size `_win_ms` is set from `window_s`.

Expected sampling rates are configured through `expected_hz`, which supports wildcard keys such as `"*"`, per sensor keys like `"acceleration"`, per device keys like `"pro1/*"`, and per device per sensor keys like `"pro1/acceleration"`. For each (`dev`, `sensor`) pair the sink tracks effective rate, inter-arrival statistics, jitter, dropout percentage versus the expected rate, and per-field statistics such as min, max, mean, standard deviation, clipping ratio, and flatline detection using `flatline_var_epsilon`. It computes a 0 to 100 health score with penalty weights from `score_weights` and optionally writes summaries as JSONL to `out_path` with rotation parameters mirroring `JsonWriterSink`. In the thesis configuration, the window is 10 s, summaries are emitted every 2000 ms, default expected rates are 25 Hz with overrides of 75 Hz for `pro1/*` and `pro2/*`, jitter thresholds are around 1.0 ms, and dropout thresholds are about 2 percent.

LabelingSink. `LabelingSink` in `sinks/LabelingSink.py` is a lightweight, non threaded writer dedicated to label records. It does not derive from `BaseSink` but manages its own file handle and lock. The constructor

```
LabelingSink(
  path: str = "output/labels.jsonl",
  flush_every: float = 1.0,
  rotate_by_size_mb: int = 0,
  rotate_every_s: int = 0,
)
```

opens `path` for append, records the creation time, and sets flush and rotation policies. The `consume(rec)` method accepts all pipeline records but writes only those with `rec["type"] == "label"`. It chooses a host time `host_ms` from `rec["host_ms"]` when available, otherwise from the authoritative `session_clock.now_ms()`, strips host centric fields from the inner label payload (such as `host_ms`, `timestamp_ms`, `timestamp_source`, and legacy fields), wraps the label into

```
{"label": <label_dict>, "host_ms": <enqueue_host_time_ms>}
```

and appends the JSON line to `path`. The sink flushes on every call when `flush_every == 0.0` or after `flush_every` seconds have elapsed, and rotates the file when size or age thresholds are exceeded. This is exactly the on disk format documented in Section 3.5.3.

SurfaceSink. `SurfaceSink` in `sinks/SurfaceSink.py` links the online TCN based classifier to the sink infrastructure. It is a `BaseSink` subclass with the constructor

```
SurfaceSink(
  run_dir: str,
  device_names: List[str],
  exclude: List[str] | None = None,
  fs: float = 75.0,
  win_sec: float = 1.0,
  jitter_ms: int = 35,
  max_gap_ms: int = 200,
  min_devices: int = 2,
  add_mask: bool = True,
  ema_alpha: float = 0.4,
  dwell: int = 10,
  torch_device: str | None = None,
  *,
  max_queue: int = 10000,
  backpressure: str = "drop_new",
  tick_every_ms: Optional[int] = None,
  rate_limit_ms: Optional[int] = None,
)
```

and a `build(params: Dict[str, Any]) -> SurfaceSink` helper that constructs instances from a configuration dictionary. The sink chooses the subset of devices to keep by filtering `device_names` against `exclude`, and sets `fs`, `win_sec`, `jitter_ms`, and `max_gap_ms` to reconstruct windows compatible with the offline `tcnwsn` training code. It loads metadata and TorchScript models from `run_dir` (and, if set, from the environment variable `MODEL_DIRECTORY`), detects the appropriate `torch.device` (CPU or CUDA), and maintains an internal circular buffer of aligned samples indexed by global time. At runtime it joins channels from all configured `device_names`, enforces a minimum number of active devices `min_devices`, optionally adds explicit masks across devices when `add_mask` is true, and computes model inputs as contiguous tensors over windows of length `win_sec` seconds. The raw model posteriors are smoothed with an exponential moving average controlled by `ema_alpha`, and a dwell counter of length `dwell` prevents rapid flapping of the final class. The sink emits a dictionary with per-class probabilities, predicted surface and motion labels, and change flags to a downstream writer identified by `emit_to_id`. In the final configuration used for the road-surface case study, `run_dir` is set to `/home/mulaz/Documenti/surfmotion_pkg/runs/surface_final`, `device_names` includes `[blt1, blt2, blt3, pro1, pro2]`, `fs = 75.0`, `win_sec = 1.0`, `jitter_ms = 35`, `max_gap_ms = 200`, `min_devices = 5`, `add_mask = true`, `ema_alpha = 0.4`, `dwell = 10`, and `emit_to_id = "surface_writer"`.

3.2.4 Config system (YAML/JSON)

Configuration is file backed and layered, so that long lived defaults can be reused across experiments and only small, session specific differences need to be overridden. At startup the client loads a base YAML file that defines global sections for devices, sinks, synchronization, microphone capture, HTTP endpoints, and GUI options; this base file includes device and sink profiles by reference so that common fragments such as BLE node descriptions or standard sink sets can be shared. Profiles for BLE nodes declare logical names, addresses, UUIDs, sampling rates, and per-sensor scaling, together with optional GUI hints such as colour and panel layout, while sink profiles define which sink classes are active and their key parameters such as file paths, rotation thresholds, queue sizes, and backpressure modes. On top of the base YAML, the client can apply one or more per session overlays in either YAML or JSON; these overlays are deep merged into the default configuration so that scalar values and lists are replaced, nested dictionaries are merged recursively, and missing keys fall back to the base configuration. Environment variables provide a final override layer for a small set of critical paths and endpoints: variables such as `MODEL_DIRECTORY` redirect the `SurfaceSink` to a different TorchScript run directory without editing configuration files, and others can override export URLs for telemetry or dashboards. Once the raw configuration has been assembled, a validator checks both structure and cross-parameter constraints: required sections such as `devices`, `sinks`, `sync`, `mic`, and `http_labels` must be present with sensible types and ranges; cross-field checks enforce constraints such as $S \leq W$ for the window stride and length used by the feature builder and surface classifier, `min_devices` $\leq$ |`device_names`| for the multi-device model, positive sampling frequencies and window sizes, and jitter and gap thresholds smaller than the window length in milliseconds; device and sink identifiers must be unique; references from sinks (for example `emit_to_id` in the `SurfaceSink`) must resolve to declared sink names; and BLE device names referenced in the synchronization section must appear in the `devices` section. For reproducibility, the resolved configuration is sserialized to JSON and emitted at startup together with a deterministic hash of its contents (for example a SHA-256 digest), so that the exact combination of base file, overlays, and environment overrides used in a session can be reconstructed later.

Main YAML configuration example A representative sensor YAML configuration is provided in Appendix A.1.

Top-level YAML configuration The full top-level YAML file that defines the experimental setup (devices, transports, sampling rates, and synchronization options) is provided in Appendix A.2.

3.2.5 Configuration hierarchy

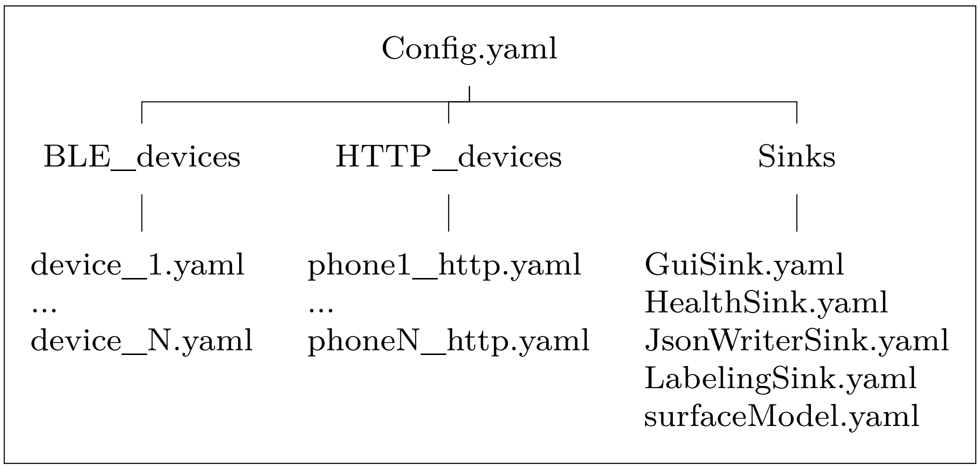


Figure 3.8: Configuration hierarchy

Configuration is organized hierarchically so that small, reusable fragments can be combined into complete session profiles. The repository mirrors this structure in the `Config` directory, which groups YAML files by role: BLE device profiles live under `Config/BLE_devices/`, HTTP producers under `Config/HTTP_devices/`, and sink definitions under `Config/Sinks/`. Each file describes exactly one logical component (for example `device2_bluetile_imu_pro.yaml` for a specific SensorTile.box PRO node, or `HealthSink.yaml` for the QoS monitor) and can be reused across multiple runs without duplication. Adding a new sensor or sink does not require editing existing files: it is enough to drop a new YAML profile into the appropriate subdirectory and reference it from the main configuration file. This hierarchical layout keeps device and sink definitions local and versionable, while the small session files express the experiment-specific choices that differentiate one acquisition from another.

3.2.6 GUI and monitoring

The live GUI is implemented by the `gui_display.py` module and runs in its own GTK thread inside the main process rather than as a separate process. Acquisition and inference threads never touch `matplotlib` directly; they push compact dictionaries into a shared `threads_queue`, which is a custom `PriorityQueue` with bounded capacity `GUI_QUEUE_MAXSIZE`. The function `start_gui()` spawns a daemon thread that calls a private `_gtk_main()` entry point, constructs a `SensorGUI` instance, and enters `Gtk.main()`. This thread owns the GTK 3 event loop and all `matplotlib` objects, while the rest of the application continues to run in the original threads that feed the queue.

Inside the GUI thread, data flow and rendering are optimised for low CPU usage. Each logical signal is stored in a dedicated `RingBuffer` implemented with NumPy arrays of fixed capacity `RB_CAP`, which is computed from the configured storage horizon `STORE_WINDOW_S` and the decimation bucket size `BUCKET_MS`. Timestamps are normalised to seconds from the first sample and are derived with `pick_time_ms()`, which uses device or host times and handles audio frames by taking the midpoint between `t0_ms` and `t1_ms`, optionally shifted by an environment variable `MIC_TSHIFT_MS`. To prevent pathological redraws, the GUI only drains a bounded number of messages per frame: `_drain_queue()` pulls at most `DRAIN_LIMIT` packets from the priority queue and appends them into the appropriate ring buffers, dropping samples that arrive out of time order or adjusting duplicate timestamps by a small epsilon. The `minmax_bucket()` routine then performs min–max downsampling over fixed time buckets given in milliseconds, so the plotted curves are reduced to a few representative points per bucket regardless of the raw sample rate.

Plotting itself is handled by a small `PlotScheduler` wrapper that integrates `matplotlib` with GTK. The module forces the `GTK3Agg` backend, constructs a single `Figure` with a shared x axis and two y axes (left for IMU channels, right for the microphone amplitude), and uses `GObject.timeout_add()` to schedule redraws at a fixed frequency `REFRESH_HZ`. On each timer tick, the scheduler calls back into `SensorGUI._draw_once()`, which drains the queue, updates the ring buffers, recomputes min–max decimated traces in the visible time window `X_WINDOW_S`, and adjusts the x and y limits. A small statistics label in the top bar shows instantaneous frames per second and draw time in milliseconds using `PlotScheduler.fps` and `draw_ms`, so the operator can monitor the cost of the GUI. Dynamic device and signal toggles are implemented as GTK `ToggleButton`s created on demand when new `dev` or sensor suffixes appear; inactive or long-idle signals are hidden and eventually removed based on `IDLE_HIDE_S` and `IDLE_REMOVE_S` thresholds, which also drive the destruction of their toggle buttons. This combination of bounded queue draining, ring buffers, bucket decimation, and fixed-rate redraw ensures that the GUI thread stays lightweight and that any back-pressure is absorbed locally by dropping GUI samples and hiding stale series, without impacting the acquisition, synchronization, or logging paths.

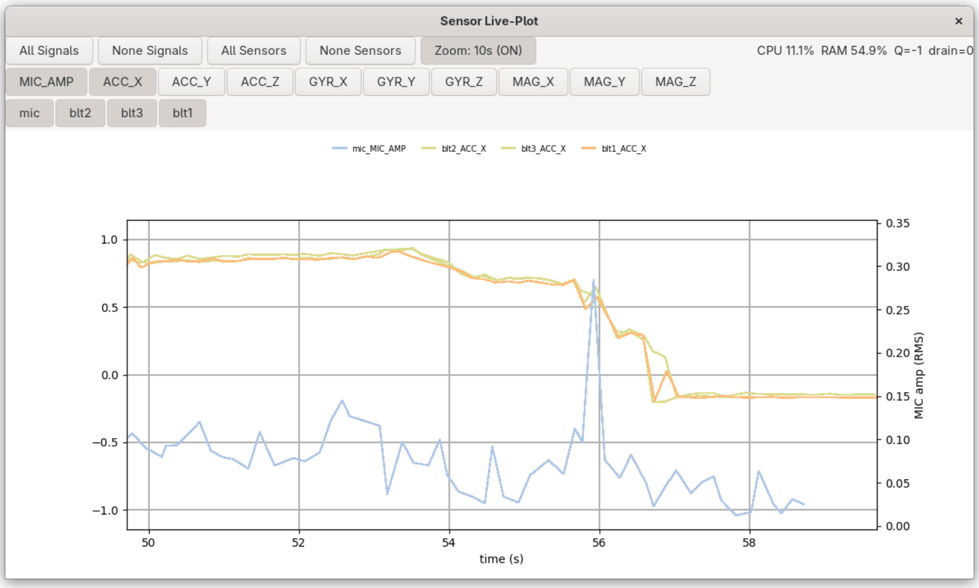


Figure 3.9: Live-plot interface for real-time visualization of synchronized multi-sensor streams

3.2.7 Logging and storage

Artifacts are append-only and human readable, and their JSONL structure follows the layout defined in Section 3.5.3. Each run produces a small set of files under a dedicated output directory, with deterministic names that encode the run identifier and a part index when rotation is enabled. In the road-surface experiments, synchronized samples are written to paths of the form

```
output/<run_id>_data_part0001.jsonl
```

with later chunks for the same run using incremented part numbers. Online surface predictions are written to `output/<run_id>_preds_surface.jsonl`, labels as closed–open intervals to `output/<run_id>_labels.jsonl`, and health summaries with throughput, jitter, round-trip time, queue depths, drops, and amplitude statistics to `output/<run_id>_health.jsonl`. Rotation can be driven by file size or wall-clock time; filenames remain predictable, and each rotated file is closed cleanly before a new one is opened so that writes are append-only. On the aligned timeline, per-device timestamps are kept non-decreasing; a short reorder buffer in the ingest path corrects minor violations caused by scheduler noise and reports any corrections to the `HealthSink`. Cross-engine analyses always use a trimmed common horizon computed offline from these files and stored alongside the run metadata, so that every synchronization engine is evaluated on exactly the same time span.

HTTP sensor The HTTP ingest path carries auxiliary sensor streams, including smartphone IMU data, to the hub. The server listens on `0.0.0.0` at a configurable port (port `8080` in the final configuration used for the dataset) and accepts JSON payloads that conform to the same schema used by BLE devices, with fields such as `dev`, `sensor`, `timestamp_ms`, `remote_ms`, and `values`. A small number of concurrent clients are supported, and, where sources expose only a wrapping counter, the server can unwrap it at ingress using a configurable modulus such as `counter_mod = 65536`. After validation and optional bias correction on the remote timestamps, HTTP samples are forwarded into the same synchronization and routing stages that handle BLE notifications, so that all producers share the common aligned time base described in Section 3.5.3.

Security Network-facing sinks and HTTP endpoints can be deployed behind TLS, either directly in the service or via a terminating reverse proxy, without changing the JSONL schema on disk. Credentials such as API keys and passwords are loaded from environment variables or a protected secrets file rather than embedded in configuration. Fields that could carry personally identifiable information are disabled in the default configuration and must be explicitly whitelisted on a per-sink basis when needed, so that the road-surface experiments described here operate strictly on IMU signals, microphone features, and surface labels and do not store user identities or routes.

Reproducibility At startup the client logs the active sink configuration, device map, synchronization engine and parameters, HTTP endpoints, and library versions together with firmware, kernel, and BlueZ versions. These metadata are kept in the same output directory as the JSONL artefacts so that the exact runtime environment can be reconstructed. Figure generation and exports refer to sink outputs by run identifier and part number, not by ad hoc filenames, and the plotting scripts consume the same JSONL structures described in Section 3.5.3. Additional stress tests, such as runs with increased RF interference, reuse the same configuration machinery; only the physical environment is changed, so the runtime configuration that governs logging, synchronization, and HTTP ingestion remains identical across conditions.

3.3 HTTP ingestion and web server

The HTTP path carries labels and auxiliary sensor data from heterogeneous producers to the hub using the same time model as BLE. An `aiohttp`-based server listens on `0.0.0.0` at port `8080` for sensor streams and at port `4040` for label events from the smartphone app. Incoming requests are parsed against a strict

JSON schema, mapped to host time using either remote timestamps plus a bias estimator or pure host arrival time, and immediately wrapped into the unified record format used in Section 3.5.3. These records are then handed to the synchronization engine and routing layer, so HTTP producers participate in the same aligned timeline as BLE IMUs, and labels travel through the pipeline as regular samples that can be written to `<run_id>_labels.jsonl` and joined with synchronized sensor data during training and evaluation.

3.3.1 Smartphone labeling app

A lightweight smartphone app provides live surface labels during acquisition. The app presents a small set of mutually exclusive buttons, one per class (for example `asfalto`, `ciottoli`, and an `unknown` or `idle` state), plus a control to stop labeling. When the rider taps a button, the app updates its internal state, records a local timestamp, and issues an HTTP `POST` to the hub's label endpoint (port `4040`, path `/labels`) with a compact JSON payload that carries the selected label and a device-side time. While a class remains active, the app periodically refreshes the label at a low rate to make the stream robust to occasional packet loss; changing class or stopping emits a corresponding event. On the hub, these events are wrapped with host-aligned timestamps, written to `labels.jsonl` using the format in Section 3.5.3, and later converted into closed–open intervals on the synchronized timeline, so that the smartphone effectively acts as a low-cost, networked ground-truth switch aligned with the IMU streams.

3.3.2 Clocking and offset correction

HTTP producers often timestamp events on their own clock, which can be biased with respect to the hub timebase. Upon reception, the hub assigns a process-local monotonic `host_ms` at first touch and derives a corrected timestamp for the record inside `http_sensor_device.py`. Unlike BLE nodes, HTTP sources do not participate in active round-trip probing; their alignment is inferred passively from the relation between remote timestamps and hub arrival times.

In EMA mode (the default), each sample yields a measured offset

$$\Delta_t = t_{\text{host}} - t_{\text{src}},$$

and the hub updates a scalar offset estimate

$$\hat{\theta}_t = (1-\alpha)\,\hat{\theta}_{t-1} + \alpha\,\Delta_t,$$

with $\alpha \in [0.1{,}0.3]$ set by `http_offset_alpha`. An outlier gate `ema_outlier_gate_ms` (default 1000 ms) resets the filter if $|\Delta_t - \hat{\theta}_{t-1}|$ exceeds the gate, so that short bursts of network delay do not bias the estimate. The corrected host time is then

$$\hat{t}_{\text{host}} = t_{\text{src}} + \hat{\theta}_t,$$

and the record is tagged with `timestamp_source:"http_offset_ema"`.

When remote timestamps show a persistent drift rather than a pure bias, an optional linear drift-aware mode can be enabled. The ingestor maintains a rolling buffer of low-delay anchors extracted from recent traffic and fits a simple slope-plus-offset model

$$t_{\text{host}} \approx a + b\, t_{\text{src}},$$

so mapping proceeds as $\hat{t}_{\text{host}} = a + b\, t_{\text{src}}$. This passively removes slow clock drift without introducing any extra protocol messages. In both modes, per-source timestamps are forced to be strictly increasing by adding a small epsilon when needed, and the corrected time is written into `timestamp_ms` before the record enters routing and logging.

3.3.3 Server architecture and concurrency

The HTTP ingestor is implemented by the `HttpSensorDevice` class, which runs in its own thread and hosts a single `aiohttp` application. One `HttpSensorDevice` instance corresponds to one configured HTTP ingress (i.e., one server per port/-group of endpoints), not one server per sensor. Inside this thread, an asyncio event loop serves three endpoints: `POST /data` and `POST /sensors` for sensor payloads, and `GET /health` for a small status report. Requests are handled without blocking operations: JSON parsing and minimal validation run on the event loop, while heavy work such as feature extraction, windowing, and inference is executed only downstream.

The `POST` handlers accept either a single dictionary or a batch wrapped in `"payload":[...]`. For each record, `http_sensor_device.py` normalizes sensor naming and values, extracts a device-side time using flexible rules (numeric fields `time/timestamp/ts/t` with unit detection or ISO-8601 strings), applies EMA or linear drift-aware clock mapping when a source timestamp is present, and constructs a unified sample dictionary with fields such as `dev`, `sensor`, `sensor_raw`, `values`, `raw_sensor_time`, `raw_host_time`, `raw_counter_unwrapped`, `remote_ms`, and // `timestamp_source`. The ingestor does not store records locally; instead it invokes a user-supplied consumer callback, which typically enqueues them into the same bounded priority queue used for BLE devices. This keeps the HTTP thread responsible only for parsing and timestamp mapping, while all shared state lives in the downstream ingress multiplexer.

3.3.4 Backpressure and flow control

Backpressure is handled cooperatively between the HTTP device thread and the central ingress multiplexer. The `HttpSensorDevice` exposes a single consumer callback; in the production pipeline this callback pushes records into a bounded single-producer/single-consumer priority queue. When the queue fills, `put` calls in the HTTP thread either block briefly or apply a configured drop strategy, so that the load on the hub cannot grow without bound. The multiplexer drains the queue at a fixed maximum rate and prioritizes sensor records over labels and GUI updates when contention appears. Since HTTP samples are normalized to the same record shape as BLE notifications, the same health and drop policies apply. GUI and export sinks consume only from downstream branches and are configured as best effort; they may coalesce or drop records locally without affecting ingestion.

3.3.5 Locking model

Concurrency control for the HTTP path follows the same principles as the rest of the acquisition stack. Within `HttpSensorDevice` the hot path is lock-free: `aiohttp` handlers run on a single event-loop thread, build immutable dictionaries for each sample, and call the consumer callback. The ingress multiplexer owns the queue and any per-source state, so it does not require fine-grained locking for HTTP devices.

Shared resources such as disk writers and session manifests are protected by a small number of coarse locks independent of the HTTP server. A writer mutex serializes file rotation (including atomic rename and checksum writes), a manifest mutex guards updates to the session description, and a metrics mutex (or atomic counters) protects global statistics. No lock is held while awaiting network or disk I/O, and lock ordering is fixed by construction, preventing deadlocks and priority inversion when BLE, HTTP, and writer threads run concurrently.

3.3.6 Reliability and idempotency

The HTTP ingestor is stateless with respect to individual requests. Each `POST` to `/data` or `/sensors` is handled independently: if parsing and normalization succeed, the corresponding samples are emitted to the consumer; otherwise the handler returns an error and no samples are forwarded. Retries are handled in an at-least-once fashion: clients may resend batches on transport failure, and duplicates can appear in the log as separate records with nearly identical timestamps. Downstream analysis tolerates such rare duplicates by operating on time windows and enforcing non-decreasing ordering when building features. On disk, writers use append-only JSONL files with atomic rotation and flush-on-close, so that completed parts are durable and verifiable.

3.3.7 Security and privacy

The HTTP sensor device includes simple security mechanisms suitable for lab or field deployments behind a gateway. Requests can be authenticated with a Bearer token: when `auth_token` is set in the configuration, the server expects an `Authorization: Bearer <token>` header and rejects unauthenticated calls. Cross-origin access is controlled by a configurable CORS origin exposed through standard `Access-Control-Allow-*` headers. Transport-level security is typically provided by running the ingestor behind a TLS-terminating reverse proxy or by binding it only on private interfaces; the JSONL schema on disk is unaffected by whether TLS is enabled upstream. Payload validation accepts only known fields and numeric values inside `values`; personally identifiable information is disabled by default and must be explicitly whitelisted per sink when needed.

3.3.8 Latency and sizing

Handler service time for the HTTP path is kept small by design. For each request, the server performs JSON parsing, timestamp extraction and unit detection, passive clock mapping (bias-only EMA by default, with an optional drift-aware linear mode), value normalization, and a single callback into the consumer. All steps are $O(1)$ per record and do not block on I/O. End-to-end ingestion latency is therefore dominated by producer scheduling, network RTT, and queueing under load. With small batch sizes and sub-100 ms send periods, labels and auxiliary IMU streams arrive within a fraction of a second of the corresponding BLE data, while CPU utilization from the HTTP path remains well below the load generated by BLE ingestion and inference.

3.3.9 Integration with the synchronization pipeline

After clock correction at ingress, HTTP records are indistinguishable from BLE records inside the hub. The `HttpSensorDevice` assigns a logical `dev` identifier, normalizes the `sensor` name to a family code such as `ACC`, `GYR`, `MAG`, `MIC`, or `GPS` when possible, and outputs a corrected `timestamp_ms` on the host-aligned axis together with `host_ms` and the raw source-side time (when available). These records then enter the same routing layer and per-device ring buffers as BLE notifications, participate in the same sliding windows used by the feature builder and surface model, and are subject to identical masking and minimum device-count rules.

The global `SynchronizationEngine` consumes HTTP sources passively: when a usable source timestamp is present, it treats `remote_ms`–`host_ms` pairs as anchors for estimating an affine mapping to the hub timeline, using the same Baseline/Basic/Kalman engines selected for BLE devices. No protocol-specific probes are required on HTTP transport; alignment relies on the source-provided timestamps plus the hub-side anchors already attached at ingress. When source timestamps are absent or unreliable, the pipeline falls back to arrival-time ordering on `host_ms`. Health telemetry reports per-source rate, gap statistics, queue depths, and basic alignment diagnostics (e.g., residual bias trends), enabling the same dashboards to monitor both BLE and HTTP producers.

3.4 Time synchronization engines

This section specifies the three synchronization engines implemented in the Python client and used throughout the experiments: *Baseline*, *Basic*, and *Kalman 2 State.* All engines map a per–device timebase to a common host timeline and expose health indicators such as offset, skew, and jitter.

Each device exposes a monotonically increasing counter u_k, either a wrapping tick counter rescaled by the nominal tick period or a millisecond counter such as `remote_ms` when available. The host timestamps each packet at the earliest callback with

$$t_k^{\text{host}} = \texttt{session\_clock.now\_ms()},$$

a process–local monotonic millisecond clock that is immune to wall–clock adjustments. For every record the engine outputs a corrected host–timeline timestamp $\hat{t}_k$ and a compact set of online statistics that are written into the JSONL logs and consumed by the analysis tools and by the road–surface classifier.

The three engines are exposed through a common Python interface in the `synchronization_engine` package and selected from the configuration, but they present a uniform API to the rest of the client.

3.4.1 Baseline

The Baseline engine represents the simplest clock model and serves both as a safe fallback during acquisition and as a reference point in the evaluation.

Model engine Baseline assumes that device clocks have negligible skew on the timescale of interest and differ from the host clock only by a constant offset. Let u_k denote device time in milliseconds (obtained by converting the raw tick counter with the nominal tick period) and let t_k^{host} be the host arrival time of an anchor packet. A first valid anchor pair (u_k, t_k^{host}) provides an estimate of the phase,

$$\hat{\theta}_0 = t_k^{\text{host}} - u_k,$$

and all subsequent samples are projected to the host timeline via

$$\hat{t}_k = u_k + \hat{\theta}_0,$$

which corresponds to a fixed slope $\alpha = 1$. If device time is unavailable or malformed, the engine falls back to pure host timestamping with $\hat{t}_k = t_k^{\text{host}}$, which provides an upper bound on achievable alignment.

Implementation and use In code, the Baseline engine maintains per–device configuration parameters such as tick period, counter width and maximum acceptable round–trip time. Device tick counters are unwrapped in software based on the configured bit–width. Once the first trustworthy anchor is seen, the engine stores the corresponding offset and enters a tracking state in which subsequent device times are simply shifted by this constant. No drift tracking is performed and no further anchor processing is required. The Baseline engine is used for HTTP producers that do not expose a stable device clock, as a robust mode whenever anchors are missing or corrupted, and as a reference engine in bench and field results, capturing the behaviour of a system that relies on host timestamps plus a single offset correction but does not actively track drift.

3.4.2 Basic engine

The Basic engine introduces skew estimation while remaining lightweight. It is representative of LS/PLL–style software synchronizers proposed in the literature for BLE sensor networks and provides an intermediate point between the fixed–offset Baseline engine and the full Kalman filter.

Model Basic assumes an affine mapping between device and host time,

$$t^{\text{host}} \approx \alpha\, u + \theta,$$

where α is the relative rate, or skew, and θ the offset. Setting $\alpha = 1$ and holding θ constant recovers the Baseline model.

Estimation from anchors The implementation builds a stream of anchor pairs (u_i, t_i^{host}). In a passive configuration, anchors are derived directly from data notifications; in an active configuration, explicit ping–pong messages are used and the host–time component of the anchor is the midpoint of the request/response timestamps, which reduces asymmetry on the link. Over a sliding window of N recent anchors stored in a ring buffer, the engine maintains running sums of u_i, t_i^{host}, u_i^2 and $u_i t_i^{\text{host}}$ and periodically recomputes the least–squares estimates

$$\hat{\alpha} = \frac{\sum(u_i - \bar{u})(t_i^{\text{host}} - \bar{t})}{\sum(u_i - \bar{u})^2}, \qquad \hat{\theta} = \bar{t} - \hat{\alpha}\, \bar{u},$$

where $\bar{u}$ and $\bar{t}$ are window means. Robustness is obtained by computing residuals $r_i = t_i^{\text{host}} - (\hat{\alpha} u_i + \hat{\theta})$ and discarding outliers whose deviation from the median residual exceeds a multiple of the median absolute deviation. When available, a queuing surrogate such as round–trip time or inter–arrival time can be used to prefer anchors recorded under low congestion.

Online update, health and correction Each new anchor causes one sample to be added to the ring buffer and one to be removed; the running sums are updated in constant time. If the window becomes ill–conditioned, for example because anchors span a very short time interval or because too many anchors are rejected as outliers, the engine holds the previous $(\hat{\alpha}, \hat{\theta})$ and marks its state as degraded. For every data packet with device time u_k, Basic returns a corrected timestamp

$$\hat{t}_k = \hat{\alpha} u_k + \hat{\theta},$$

a residual $r_k = t_k^{\text{host}} - \hat{t}_k$ when the host arrival time is available, and an estimate of skew in parts per million,

$$\widehat{\text{skew}}_{\text{ppm}} = (\hat{\alpha} - 1) \cdot 10^6,$$

which is clamped to a configurable range to avoid reacting to spurious anchors. These quantities are inserted into the JSONL records and later summarized by the analysis tools.

3.4.3 Kalman 2 state engine

The Kalman engine implements the full two–state stochastic model introduced in the background chapter and realizes it in code via a compact two–state filter that tracks both offset and skew, adapts its resynchronization cadence and exposes high–level health metrics.

State, observation, and noise The filter operates directly on unwrapped device time u_k in milliseconds and models the affine mapping as a time–varying line. The state vector $\mathbf{x}_k = [\, a_k,\ b_k \,]^\top$ contains the current slope a_k and offset b_k such that $t^{\text{host}} \approx a_k u + b_k$. Process noise is modelled as a random walk,

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{w}_k,$$

with zero–mean Gaussian $\mathbf{w}_k$ and diagonal covariance that encodes the expected wander of slope and offset over time. Each anchor provides a measurement of host time at a known device time. For midpoint anchors built from ping–pong exchanges, the measurement y_k is the midpoint of the host–side request and response times and the associated device time is the value carried by the reply. The measurement equation is

$$y_k = u_k a_k + b_k + v_k,$$

where v_k is Gaussian noise with variance R_k derived from the observed round–trip time and a floor that accounts for residual jitter. The standard Kalman prediction–update recursion is run in closed form for the two–state case, and the Joseph update is used to preserve the positive semidefiniteness of the covariance matrix.

Tuning, gating, and resync policy The Kalman engine is configured through a small set of parameters that control the process noise on slope and offset, the initial covariance, the Mahalanobis gating threshold used to reject inconsistent anchors, and the resynchronization cadence. In a typical setup, the engine operates in a warm–up phase with frequent resynchronizations until the estimated skew and its uncertainty fall below configured thresholds; once this happens it transitions to a cruise phase in which anchors are requested less often. Large residuals cause anchors to be rejected and can trigger a temporary increase of the assumed measurement noise or covariance inflation to speed recovery. Soft and hard reset policies distinguish between short gaps that can be bridged while preserving the slope estimate and long outages that require a full rebootstrap.

Correction and health outputs For data packets, the engine evaluates the current affine mapping at the device time and returns

$$\hat{t}_k = \hat{a}_k u_k + \hat{b}_k.$$

From the slope it derives a skew estimate in ppm and a projected drift per day in milliseconds. The engine also tracks innovation statistics such as the median and p95 of the measurement residuals and of the underlying round–trip times. A compact snapshot of this state is periodically written alongside sensor data and later used by the analysis tools to characterize convergence, stability and response to disturbances.

3.4.4 Metrics and complexity

Timing performance is summarized by a set of metrics computed on bench and field logs. Offset error is quantified as the difference between corrected timestamps and a chosen reference timeline, and is reported through robust summaries such as median, 95th percentile and median absolute deviation. Cross–device lag is computed by subtracting the aligned timelines of pairs of devices and examining the distribution of this difference on the time interval where both engines provide valid timestamps. Drift is characterized by windowed estimates of skew in parts per million, obtained from the derivative of the corrected timeline with respect to device time or directly from the engine's skew estimate. Jitter is characterized through statistics of inter–arrival times and, for the Kalman engine, through standardized innovations. The associated Python tools process JSONL logs in a streaming fashion and emit drift and lag summaries, together with the plots used in the results chapter.

Quality–of–service is characterized by effective sampling frequency, drop rate, reconnect count and maximum inter–arrival time, all computed from the corrected timestamps and drop counters. These quantities are reported alongside timing metrics for each engine and scenario.

All three engines are designed for deployment on a resource–constrained hub and maintain constant–time per–packet cost. Baseline performs a single addition and uses constant memory. Basic maintains a ring buffer of a fixed number of anchors and a small set of running sums but still updates in constant time. The Kalman engine performs only a handful of operations on small two–dimensional vectors and matrices for each anchor and uses constant memory. In all three cases, the computational cost is negligible compared to BLE and HTTP I/O and to downstream machine–learning inference on the hub.

3.4.5 Software implementation and deployment

The synchronization logic is isolated behind a small API. The engine exposes methods to mark when a synchronization request is sent, to update its state when a reply carrying device time is received, to convert a raw device timestamp into aligned host time, to report whether a new resynchronization is needed, and to export a compact state description for monitoring and logging. BLE and HTTP device handlers invoke these methods at the appropriate points in the protocol and never manipulate offsets or skews directly.

The client logs JSON Lines with a consistent schema across engines. Each record contains the device identifier, sensor name, aligned host timestamp, device time in milliseconds when available, the raw host timestamp at callback entry, and the name of the engine that produced the alignment. Engines enrich the records with additional fields such as offset, skew in ppm, residuals or innovations, but the core fields remain identical so that downstream tools and the road–surface pipeline can process logs produced by different engines with the same code.

The acquisition process runs as a small set of cooperating threads. Each BLE device has its own worker thread that handles notifications; another thread listens over HTTP for labels and auxiliary sensors; a further thread can capture microphone data directly on the hub; and one or more sink threads take care of logging and visualization. All of these producers push records into bounded queues. The synchronization engine sits in the ingest path and processes records without blocking. If back–pressure builds up, the system can safely discard work from non–critical channels, for example live visualization, so that IMU streams remain intact.

All engines detect counter wraps, device resets and long gaps. Wraps are handled by keeping an epoch count per counter width and adding the appropriate offset when the raw value rolls over. After a reset or a long outage the engines rebootstrap from fresh anchors and mark their state as degraded until enough consistent anchors have been accumulated. The Kalman engine additionally freezes or slews the published offset when residuals stay within or exceed configured thresholds and automatically adjusts the resynchronization cadence between warm–up and cruise. Together with the analysis tools, these safeguards ensure that the synchronized timeline remains within the error budgets used in the bench and field case studies.

3.5 Data acquisition and labeling

This section defines how raw signals and labels are produced, time aligned, and packaged into datasets for analysis and reproducibility. All records follow a unified JSONL schema with explicit device and host timing so that synchronization engines can be compared and downstream audits can be repeated.

3.5.1 Scenarios and placement

Data are collected on public roads that are representative of the two target classes *asfalto* and *ciottoli*. Sessions include urban segments, smooth arterial roads, and historic-center stretches with cobblestones, driven or ridden at moderate speeds. IMU nodes are rigidly mounted to the vehicle structure, for example on chassis or body attachment points, to maximize coupling with vibrations; foam isolation and loose mounts are avoided. Orientation is fixed for the duration of a session and documented in the experiment notes; invariance to mounting is handled in preprocessing and model design rather than by remounting sensors. A smartphone can be used as an auxiliary source for HTTP streaming and voice notes and is held in a fixed cradle to limit motion relative to the vehicle.

3.5.2 Smartphone and HTTP streams

Alongside BLE IMUs, the hub can ingest HTTP producers that push IMU-like or metadata packets using the same unified schema as the BLE devices. Typical HTTP sources include smartphone accelerometer and gyroscope streams, GPS fixes, environmental sensors, and application-level events. Requests are handled by the `HttpSensorDevice` (Section 3.3.3), which accepts either single records or batches, stamps them on arrival with the hub's monotonic clock, and normalizes them into JSON objects with fields such as `dev`, `sensor`, `timestamp_ms`, `remote_ms`, and `values`.

The logical device identifier `dev` is taken from the `emit_dev` configuration of the HTTP ingester (for example `"phone1"` when the smartphone acts as a node). The field `sensor` is a compact family code derived by the ingester from the original sensor type, for example `"ACC"`, `"GYR"`, or `"GPS"`. The payload in `values` is parsed into canonical keys (`accX`, `accY`, `accZ` for accelerometers; `gyrX`, `gyrY`, `gyrZ` for gyroscopes; latitude/longitude/altitude/accuracy for GPS), so that downstream tools can treat HTTP and BLE samples uniformly. An optional device-side time is extracted from generic fields such as `time`, `timestamp`, `ts`, or `t`; this time is converted to milliseconds and used by the HTTP clock-mapping logic in Section 3.3.2. The resulting aligned time is written into `timestamp_ms`, and `timestamp_source` records whether it came from EMA or Wait-n-Sync mapping.

All HTTP records, not only smartphone IMU streams, carry an additional `sensor_raw` field that preserves the original sensor identifier supplied by the client (for example accelerometer, linear_acceleration, gyroscope, or location). This field is used solely for provenance and debugging; the pipeline logic relies on the normalized `sensor` family code and on the canonical `values` keys. After clock correction at ingress, HTTP samples are indistinguishable from BLE samples inside the pipeline and are written to the common JSONL log described in Section 3.5.3. In the road-surface experiments, smartphone streams (ACC/GYR/GPS) are treated as auxiliary: they are typically excluded from model training unless placement and orientation are tightly controlled, but they remain useful for exploratory analysis, for cross-checking surface transitions, and for validating that the synchronized timeline is consistent across transports.

3.5.3 Dataset structure

Recordings are organised by session, and each session produces a small set of append-only JSONL files. The processing pipeline depends only on the actual JSON objects written by the acquisition client, which are described here for sensor data, microphone frames, and labels.

Sensor samples are written by the `JsonWriterSink` directly from the dictionaries emitted by the BLE and HTTP devices. A typical BLE IMU sample has the following structure:

```
{
  "dev": "<BLE device_id>",
  "sensor": "<sensor_name>",
  "timestamp_ms": <aligned_host_timestamp_ms>,
  "values": {
    "accX": <value_x>,
    "accY": <value_y>,
    "accZ": <value_z>
  },
  "raw_sensor_time": <device_raw_time>,
  "raw_host_time": <host_time_at_callback>,
  "raw_counter_unwrapped": <device_counter_unwrapped>,
  "remote_ms": <device_time_ms>,
  "delta_vs_remote_ms": <aligned_minus_device_ms>,
  "delta_vs_host_now_ms": <aligned_minus_now_ms>,
  "timestamp_source": "<remote|host|fallback>"
}
```

Here `dev` is the logical node name, `sensor` identifies the signal (for example `"acceleration"` or `"gyroscope"`), and `timestamp_ms` is the corrected time on the host timeline returned by the selected synchronization engine. The field `values` contains the sample payload with axis keys; in the IMU case these are accelerometer or gyroscope components such as `accX`, `accY`, and `accZ`. The fields `raw_sensor_time`, `raw_host_time` and `raw_counter_unwrapped` store the device tick counter, the host time at callback entry, and the unwrapped counter respectively. The field `remote_ms` is the device time expressed in milliseconds, while `delta_vs_remote_ms` and `delta_vs_host_now_ms` quantify, in milliseconds, the difference between the aligned timestamp and, respectively, the device clock and the current host clock. The string `timestamp_source` records whether the alignment used remote device time, host time, or a fallback.

HTTP sensor sources use the same layout, with an additional `sensor_raw` field that preserves the raw sensor name sent by the client:

```
{
  "dev": "<HTTP device_id>",
  "sensor": "<sensor_family>",          // e.g. "ACC", "GYR", "GPS"
  "sensor_raw": "<raw_sensor_name>",    // e.g. "accelerometer", "location"
  "timestamp_ms": <aligned_host_timestamp_ms>,
  "values": {
    "accX": <value_x>,
    "accY": <value_y>,
    "accZ": <value_z>
  },
  "raw_sensor_time": <device_raw_time>,
  "raw_host_time": <host_time_at_arrival_ms>,
  "raw_counter_unwrapped": <device_counter_unwrapped>,
  "remote_ms": <mapped_time_ms>,
  "delta_vs_remote_ms": <aligned_minus_remote_ms>,
  "delta_vs_host_now_ms": <aligned_minus_now_ms>,
  "timestamp_source": "<http_offset_ema|wait_n_sync>"
}
```

Embedded microphone dataset structure The on-board microphone on the hub is logged as a regular sensor on the same synchronized axis. A typical audio frame without embedding is encoded as:

```
{
"dev": "<device_id>",
"sensor": "microphone",
"timestamp_ms": <aligned_host_timestamp_ms>,
"values": {
"amplitude": <frame_rms_amplitude>,
"peak": <frame_peak_amplitude>,
"n_samples": <frame_sample_count>
},
"raw_host_time": <host_time_at_capture_ms>,
"timestamp_source": "host"
}
```

Here `dev` identifies the microphone stream on the hub, `sensor` is `'microphone'`, `timestamp_ms` is the aligned time associated with the audio chunk on the common host timeline, and `values` contains per-frame summary statistics: RMS amplitude (`amplitude`), peak amplitude (`peak`), and the number of raw samples represented by the frame (`n_samples`). The field `raw_host_time` stores the monotonic host time at capture, and `timestamp_source` records that alignment is derived directly from the host clock.

When microphone embedding is enabled, the client augments the payload with the raw PCM encoded as base64 and an integrity checksum. A JSONL line with embedding has the form:

```
{
"dev": "<device_id>",
"sensor": "microphone",
"timestamp_ms": <aligned_host_timestamp_ms>,
"values": {
"amplitude": <frame_rms_amplitude>,
"peak": <frame_peak_amplitude>,
"n_samples": <frame_sample_count>
"pcm_b64": "BASE64_PCM_PAYLOAD", #optional
"sha1": "HEX_SHA1_OF_RAW_PCM"    #optional
},
"raw_host_time": <host_time_at_capture_ms>,
"timestamp_source": "host"
}
```

The microphone path therefore appears in the dataset as another producer that writes synchronized records into the common JSONL log, with an extensible set of per-frame descriptors controlled by command-line options. Microphone capture is enabled at runtime with `-mic`, which starts a dedicated recorder thread that pushes audio frames into the same priority queue as the BLE devices. The flags `-mic-device` and `-mic-rate` select the ALSA input (for example `hw:0,0`) and the sampling rate in hertz, while `-mic-chunk-ms` fixes the duration of each audio chunk and thus the temporal resolution of the emitted frames. In the default configuration the JSONL log carries only summary descriptors `amplitude`, `peak` and `n_samples`; the option `-mic-embed-audio` additionally adds `pcm_b64` and `sha1` so that the raw waveform can be reconstructed directly from the log for short test runs. For long acquisitions the preferred mode is instead to enable `-mic-save-wav`, which writes the raw PCM to rolling WAV files on disk, with `-mic-wav-dir` selecting the output directory. Offloading the full waveform to WAV keeps the JSONL stream compact, reduces storage overhead compared to repeated base64 payloads, and provides a clean input format for external speech-to-text models such as Vosk, which operate on the audio files while the synchronized JSONL records retain only lightweight per-frame descriptors.

All downstream tools, including the road-surface classifier, operate directly on records with these schemas and treat any additional fields as optional metadata that can be ignored without affecting core processing.

Labeling Dataset structure Labels are written by a separate `LabelingSink` that subscribes to the same pipeline and filters only records with `type="label"`. The HTTP labeling server accepts POST requests of the form `{"label": <value>}`, wraps them with timing information on the hub, and enqueues records shaped as

```
{"type": "label", "label": <value>, ...}
```

The `LabelingSink` strips host-centric fields from this inner object and writes a compact wrapper to `labels.jsonl`:

```
{
"label": {"type": "label", "label": <label_id>},
"host_ms": <host_timestamp_ms>
}
```

The inner `label` object always carries `type:"label"` and a `label` value that is any JSON-serialisable payload; in the road-surface experiments this value is a simple string such as `"asfalto"` or `"ciottoli"`. The outer `host_ms` field is the hub's monotonic timestamp, in milliseconds, at enqueue time. When label edits are produced by the smartphone app or by voice-note post-processing, they follow the same JSONL shape; higher-level scripts then convert this stream of time-stamped label events into interval labels on the synchronized axis for training and evaluation.

Quality-of-service summaries are written to a health log, typically `health.jsonl`. Each record aggregates statistics over a fixed time window, such as effective sampling rate per device and sensor, round-trip time and jitter summaries for synchronization messages, drop counts, and maximum gaps. These health records allow offline verification that each session meets the minimum requirements specified in Section 3.7.2 and support the interpretation of outliers in machine-learning performance.

This JSONL schema is shared across all case studies and synchronization engines. Analysis scripts and the road-surface classification pipeline consume these files directly, so that figures and tables in the following chapters can be regenerated from raw logs by re-running the published command-line recipes or explored offline. The same artefacts also define the temporal backbone for the external speech-to-text pipeline, which aligns microphone-based transcripts to the synchronized acquisition timeline.

3.5.4 Trigger protocols, labeling application, and ground truth

Each session starts with a soft `MARK` event emitted by the hub and logged on all streams, providing a shared anchor on the synchronized timeline. Surface labels are defined on this hub timeline and derived primarily from operator input via a smartphone labeling application, with route knowledge and offline signal inspection used as consistency checks. The app presents a small set of mutually exclusive buttons (for example *asfalto* (Asphalt), *ciottoli* (cobblestones), *unknown*) together with a clear indication of the currently active class; when the operator taps a new surface, the app updates its internal state and issues an HTTP `POST` to the hub's label endpoint (port `4040`) with a compact JSON payload carrying the selected label, optionally refreshing the active label at a low rate to make the stream robust to occasional packet loss. On the hub, the HTTP handler wraps each incoming label edit into a normalized record and forwards it to the `LabelingSink`, which does not construct intervals; instead, it logs instantaneous events to `labels.jsonl` with the schema of Section 3.5.3:

```
{
"label": {"type": "label", "label": <label_id>},
"host_ms": <host_timestamp_ms>
}
```

Here `host_ms` is the hub-side monotonic timestamp, in milliseconds, at enqueue time on the synchronized axis (given by `session_clock.now_ms()`), and the inner `label` object encodes the current class (for example `"asfalto"` or `"ciottoli"`), so that `labels.jsonl` is a state-change log: a time-ordered sequence of "now the label is X" events aligned with the sensor data. In parallel, the operator can record short voice notes at key transitions (for example "start ciottoli" or "end ciottoli"); audio is captured on the hub and processed offline with Vosk (see paragraph 3.5.4) to obtain time-stamped transcriptions, and a deterministic parser maps relevant phrases to label start/stop commands and reconciles them with the online app labels, while route annotations for stretches known to be cobblestone provide an additional check, especially on repeated runs. Final ground-truth intervals are derived offline from this combined event stream: analysis scripts sort events by `host_ms`, collapse redundant repetitions, and convert state changes into half-open segments ([t_start, t_end)) with `class` in `asfalto`, `ciottoli`; ambiguous or disputed spans, identified by inconsistencies between the app log, route annotations, and signal previews, are marked `ignore` and excluded from training and metrics. When comparing different synchronization engines, both sensor samples and derived label intervals are finally restricted to the common trimmed horizon on which every engine provides valid corrected timestamps, so that all performance metrics are computed on exactly the same time span and label set.

Vosk Speak-to-text model Vosk [22] is an open-source speech recognition toolkit designed to run efficiently on a wide range of devices, including embedded and edge platforms. It provides offline, on-device speech-to-text in multiple languages, exposing simple APIs (for example Python and C++) that accept audio streams and produce time-stamped word hypotheses. In this work, Vosk is used to transcribe short voice notes recorded during the sessions, so that spoken phrases like "start ciottoli" or "end asfalto" can be converted into structured events and aligned with the rest of the dataset.

3.6 Machine learning pipeline

This section presents the end-to-end pipeline used to train and evaluate a compact surface-classification model on synchronized inertial data. Surface classification is used as a controlled workload to validate the time-alignment framework rather than as a stand-alone contribution. The pipeline is hardware-agnostic: it consumes multi-device logs aligned to a common timeline and supports heterogeneous sensors. In the experimental runs, IMU streams were acquired over BLE and time-stamped labels were delivered over HTTP.

3.6.1 Problem formulation

Let $\mathcal{D} = \{1, \ldots, D\}$ be the set of active devices in a session. After resampling to a common rate f_s, each device $d \in \mathcal{D}$ provides a C-channel time series $\mathbf{x}^{(d)}(t) \in \mathbb{R}^C$ expressed on the global timebase produced by the synchronization engines. Ground-truth labels are provided as time intervals

$$\mathcal{L} = \left\{(t_j^{\text{start}}, t_j^{\text{end}}, y_j)\right\}_{j=1}^{J}, \quad y_j \in \{1, \ldots, K\},$$

where $(t_j^{\text{start}}, t_j^{\text{end}})$ delimits a segment with surface class y_j.

Learning is cast as windowed sequence classification. For a window of duration W centered at time τ, the input tensor $\mathbf{X}_\tau$ is obtained by stacking all device channels over the interval $[\tau - W/2,\ \tau + W/2]$. After resampling, the window contains

$$T = \left\lfloor W f_s \right\rfloor$$

time steps and is represented as $\mathbf{X}_\tau \in \mathbb{R}^{T\times(C\cdot D)}$. Typical configurations in this work use $f_s \approx 75$ Hz and $W \in [1.0,1.5]$ s, so that each window contains roughly $T \in [75,115]$ samples.

A training label y_τ is assigned to the window if it overlaps a single ground-truth interval with sufficient coverage. Let $\Delta(\tau)$ be the length of the intersection between the window and an interval $(t_j^{\text{start}}, t_j^{\text{end}})$. The window is assigned class y_j if

$$\frac{\Delta(\tau)}{W} \geq \gamma$$

and there is no competing interval of a different class that also satisfies the coverage constraint. In practice γ is set to values such as 0.5 or 0.7 so that more than half of the window is dominated by a single surface type. Windows that straddle multiple classes or fail the coverage criterion are discarded.

The formulation explicitly supports missing devices. If a device is inactive or has insufficient data in a window, its channels are zero-filled after normalization. In parallel, a binary mask channel per device encodes whether samples are real or imputed, so that the model can learn to ignore artificial zeros and avoid leakage from the imputation strategy. Denoting by $\mathbf{m}^{(d)}(t) \in \{0,1\}$ the completeness indicator for device d, the final input to the network for a window is

$$\tilde{\mathbf{X}}_\tau \in \mathbb{R}^{T\times(C\cdot D+D)},$$

obtained by concatenating the inertial channels and one mask channel per device along the feature dimension. The classifier is trained to map each window $\tilde{\mathbf{X}}_\tau$ to a probability distribution over the K surface classes.

3.6.2 Pre-training tools and label consolidation

Raw acquisitions are stored as JSONL logs that contain all streams on the unified timebase: BLE IMU packets, optional auxiliary sensors, and HTTP messages used for labeling. Each record carries a hub-side timestamp, a source identifier, and a payload with device-specific fields. Before training, a set of pre-training tools converts these raw logs into the interval labels $\mathcal{L}$ and the window indices used by the classifier.

Label annotations are emitted by a mobile or desktop application as point events over HTTP (on a dedicated labels endpoint). Each event contains at least a timestamp, a surface tag, and possibly a state flag (pressed, released) or a confidence value. Two preprocessing steps are applied.

From points to intervals A label manager first converts the pointwise annotations into consistent intervals on the global timebase. Events are sorted by their timestamps and mapped to a canonical set of surface identifiers $\{1,\dots,K\}$. The manager maintains the current active class and opens a new interval when a tag becomes active. An interval is closed when the user switches to a different surface, an explicit "none" tag is received, or the session ends.

To reduce sensitivity to annotation jitter and short gaps, the tool applies simple temporal heuristics:

- intervals shorter than a minimum duration $T_{\min}$ (for example 1 to 2 s) are discarded or merged with neighboring intervals of the same class;
- gaps shorter than a merge threshold $G_{\max}$ (for example 0.5 s) between intervals with the same label are bridged, yielding a single longer interval;
- isolated points that do not survive these rules are treated as spurious and removed.

The output of this stage is the set $\mathcal{L} = \{(t_j^{\text{start}}, t_j^{\text{end}}, y_j)\}$ used in the problem formulation: non-overlapping surface segments that cover the annotated portions of the session.

Window index and dataset manifests A second tool builds the window index used during training. Given the synchronized sensor streams, the interval labels $\mathcal{L}$, and a configuration file that specifies window length W, stride S, coverage threshold γ, device completeness ρ, and minimum active devices m, the tool generates a dense grid of candidate window centers τ_n. For each candidate, it:

- checks which devices satisfy the completeness threshold ρ over the window;
- computes the label coverage ratio $\Delta(\tau_n)/W$ for each overlapping interval;
- assigns a class y_{τ_n} if exactly one interval exceeds γ and at least m devices are sufficiently complete.

Accepted windows are stored as entries in an index file (for example JSONL or Parquet) that references the source log, device names, window center, and label. The raw time series remain in the original acquisition files, so the index can be regenerated or modified without duplicating data.

Additional tooling computes dataset-level summaries and sanity checks: histograms of class durations, coverage per route or session, distributions of window counts per class, and simple time plots that overlay intervals and labels. These reports are used to verify that the point-to-interval conversion and window selection behave as expected before any model is trained.

3.6.3 Features and windowing

All streams are first aligned to the unified timeline produced by the synchronization engines and resampled to the common rate f_s. Resampling is performed independently per device using zero-order hold or linear interpolation on short gaps. Short gaps are filled by interpolation up to a maximum duration $\Delta t_{\max}$ (for example 100 to 150 ms) chosen to avoid distorting the signal; longer gaps remain missing and are handled via the mask channels. Signals are detrended and standardized per device using z-scores computed strictly on the training split to prevent information leakage across splits. The same per-device mean and standard deviation are reused at validation and test time.

For accelerometers, two representations are supported. The simplest uses the three raw axes (a_x, a_y, a_z) after mean removal. A second, more rotation-robust representation decomposes the signal into the norm $\|\mathbf{a}\|$, a component along an estimated gravity direction $a_{\|}$, and a component in the orthogonal plane $a_{\perp}$. The gravity direction $\hat{\mathbf{g}}$ is estimated as a slowly varying vector, for example via low-pass filtering of the accelerometer with a cut-off in the 0.3 to 0.5 Hz range. This representation separates vertical loading from lateral vibrations and reduces sensitivity to mounting orientation.

Optional spectral features can be computed as band-power estimates from Welch segments over each window, for example using Hann windows with 50 percent overlap and aggregating energy over a small number of frequency bands in the 0.5 to 20 Hz range. In practice, for real-time operation on the hub, the pipeline typically relies on raw time-domain sequences and simple normalization to minimize latency and computational overhead.

Windows are generated with a fixed length W and a stride S. The stride controls both the density of training samples and the effective update rate at deployment: small strides yield more overlapping windows and lower decision latency at the cost of higher computation. Typical strides in this work range from 0.1 to 0.5 s. A window is retained if two conditions are met. First, at least m devices must satisfy a per-device completeness threshold ρ, defined as the minimum fraction of non-missing samples within the window:

$$\frac{1}{T} \sum_{t \in \text{window}} \mathbf{1}\{\mathbf{m}^{(d)}(t) = 1\} \geq \rho.$$

Second, label coverage must exceed the threshold γ so that the assigned class is representative of most of the window.

Class imbalance across surfaces is mitigated by combining balanced sampling of windows with a weighted loss function that down-weights frequent classes and emphasizes rarer ones. During training, mini-batch sampling is biased to draw similar numbers of windows per class when possible. Lightweight data augmentations are applied in a way that preserves label semantics: amplitude scaling and jitter, small additive Gaussian noise, random per-device sub-sample time shifts to emulate residual clock errors, and constrained 3D rotations of accelerometer axes that stay consistent with plausible changes in mounting orientation. Time shifts are clipped to a few samples so that augmentations remain compatible with the residual timing error guaranteed by the synchronization pipeline.

3.6.4 Model (Conv1D)

The classifier is a compact one-dimensional convolutional network designed for CPU-only inference and suitable for causal evaluation on the edge hub. Inputs are arranged as $(T,\ C \cdot D + D)$, that is, all sensor channels concatenated with one binary mask channel per device.

The backbone consists of three depth wise-separable Conv1D blocks with increasing dilation factors, followed by a global temporal pooling head and a linear classifier. Each block contains:

- a depthwise Conv1D with kernel size k and dilation d,
- a pointwise 1×1 Conv1D to mix channels,
- batch normalization,
- a nonlinearity Rectified Linear Unit(ReLU),
- and, for the first two blocks, dropout for regularization.

Depth wise-separable convolutions factor a standard convolution into a depthwise operation (one filter per input channel) followed by a pointwise 1×1 convolution. This factorization substantially reduces the number of parameters and multiply-accumulate operations compared with a dense $k \times C_{\text{in}} \times C_{\text{out}}$ convolution, leading to lower latency and power consumption. In typical configurations, the first two blocks use 64 channels, the last one 96 channels, kernel sizes between 5 and 9, and dropout in the 0.1 to 0.3 range.

Dilation factors of 1, 2, and 4 increase the receptive field without increasing the window length or the number of layers. As the dilation grows, each convolutional kernel spans a larger temporal context, capturing longer-range patterns in the vibration signal that are informative for distinguishing surfaces at different speeds and wheel–road interactions. For example, with $k = 7$, three layers, and dilations (1,2,4), the effective receptive field covers several tens of samples, corresponding to multiple wheel revolutions at urban cycling speeds.

In the causal configuration used for online inference, the network uses left padding so that predictions at time t depend only on samples at or before t. For offline evaluation and ablation studies, symmetric padding can be used to approximate a "same" convolution and slightly improve accuracy by allowing the model to see future context.

Multi-device fusion is handled implicitly by stacking channels from all devices and explicitly by passing the mask channels. An optional squeeze-and-excitation (SE) block acts on the channel dimension and can be enabled to let the network adaptively reweight devices and channel groups based on their contribution to the current decision. After the convolutional backbone, a global average over time produces a fixed-size embedding, which is fed to a fully connected layer that outputs one logit per class. At deployment, frame-level probabilities are further processed by an exponential moving average with a decay factor in the 0.8 to 0.95 range and a minimum dwell-time rule that converts noisy frame-wise predictions into stable surface segments.

The entire model is implemented in PyTorch and kept small enough (on the order of 10^5 parameters) to fit comfortably on the edge hub while respecting real-time constraints.

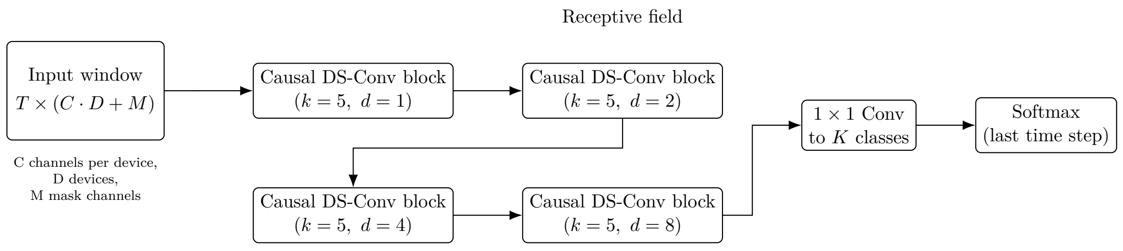

Receptive field
Input window
$T \times (C \cdot D + M)$
C channels per device,
D devices,
M mask channels
Causal DS-Conv block ($k = 5,\ d = 1$)
Causal DS-Conv block ($k = 5,\ d = 2$)
Causal DS-Conv block ($k = 5,\ d = 4$)
Causal DS-Conv block ($k = 5,\ d = 8$)
1×1 Conv to K classes
Softmax (last time step)

Figure 3.10: Overview of the causal Conv1D TCN: four causal DS-Conv blocks (dilations 1, 2, 4, 8) followed by a 1x1 head and Softmax on the last time step.

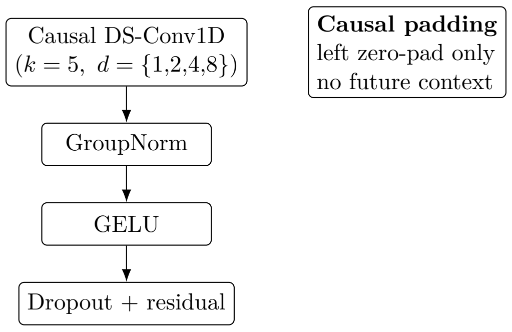


Figure 3.11: Internal structure of one causal DS-Conv block used in the TCN.

3.6.5 Training setup and metrics

The model is trained with categorical cross-entropy and class weights w_k inversely proportional to the empirical class frequencies, so that rare surfaces contribute proportionally more to the objective. Optimization uses Adam or AdamW with a learning-rate schedule (cosine decay or step decay) and early stopping based on validation macro-F1. Typical configurations use batch sizes between 64 and 128 windows, an initial learning rate around 10^{-3}, and weight decay in the 10^{-4} to 10^{-2} range. Mini-batches mix windows across classes, sessions, and routes to improve generalization and reduce overfitting to specific trajectories.

Normalization statistics (means and standard deviations per device and channel) are estimated once on the training split and reused as fixed parameters at validation and test time. Data splits are performed at the level of sessions, routes, or subjects so that the model is always tested on unseen temporal sequences and mounting conditions, avoiding optimistic estimates from overlapping trajectories. A typical split dedicates 60 to 70 percent of sessions to training, 10 to 20 percent to validation, and the remainder to testing.

Sample-wise metrics include accuracy, balanced accuracy, macro-precision, macro-recall, macro-F1, and negative log-likelihood. Segment-wise metrics are computed by grouping frame-wise predictions into contiguous segments and matching them to ground-truth intervals using an intersection-over-union (IoU) threshold, for example 0.5. This evaluates not only whether the model recognizes the correct surface but also whether it localizes transitions at the right time. Segment-wise precision, recall, and F1 are reported in addition to sample-wise scores.

For online use, decision latency is a key quantity. It is defined as the delay from the start of a labeled interval to the first time the smoothed class probability for the correct surface crosses the decision threshold and remains above it for the required dwell period. This metric captures the trade-off between responsiveness and stability in a way that is directly relevant for real-time applications. Latency is reported jointly with accuracy to characterize the speed–accuracy trade-off.

Finally, the implementation is optimized for deployment on the edge hub. The network remains compact and deterministic, and it supports both quantization-aware training and post-training static quantization to 8-bit arithmetic when beneficial for performance and energy. Export to TorchScript or ONNX allows running the same model under a lightweight inference runtime on the STM32MP2 platform. Runtime constraints are monitored as the number of processed windows per second, the end-to-end latency per window, and the CPU utilization at the target clock frequency. Under these constraints, the pipeline executes synchronized window extraction, Conv1D inference, and probability smoothing in real time on the hub, reusing the same preprocessing and labeling rules adopted in offline training.

3.7 Experimental setup and datasets

This section describes the experimental setups and datasets used to validate the proposed framework across three complementary case studies. First, it introduces the wood-plate impulse bench used to stress and compare the three synchronization engines under controlled conditions, quantifying alignment error, latency, jitter, loss, and throughput. Second, it details the bicycle acquisitions used to construct the road-surface dataset, where synchronized inertial streams and HTTP annotations support a concrete road-surface classification workload. Finally, it outlines a heterogeneous-sensor scenario in which additional BLE and HTTP sources (such as physiological and auxiliary motion sensors) are integrated on the same synchronized timeline to demonstrate that the architecture generalizes beyond IMU-only sensing.

3.7.1 Engine comparison: wood-plate impulse bench

The bench is designed to quantify synchronization accuracy, drift tracking, latency, jitter, and loss under controlled kinematics and radio conditions. The rotary arm generates repeatable motion and well-defined reference events that serve as a common ground truth for all devices. A DC motor drives a rigid arm that periodically strikes a wooden board, while a current sensor on the motor supply detects the instant of impact. The DC motor is actuated by an Arduino-class microcontroller, which commands the motor driver and thus controls the rotation speed and impact frequency. Seven IMU nodes (SensorTile.box and SensorTile.box PRO) are mounted at equal radii on a wooden plate; their z axes are approximately normal to the plate, and the axes are marked on the surface so that orientations can be reproduced between sessions. The control plate hosts the Arduino-class microcontroller, the motor driver, and the associated wiring, and the arm carries a metal flag that interrupts a photocell once per revolution. The hub timestamps pulses from the photocell and the motor encoder on a wired GPIO, so that all reference times are measured directly on the hub clock. Figure 3.12 illustrates this configuration: one plate contains the motor, driver, microcontroller, and photocell, while the other carries the IMU nodes arranged in a circle around the center of rotation. A side view of the assembled rig is shown in Figure 3.13.

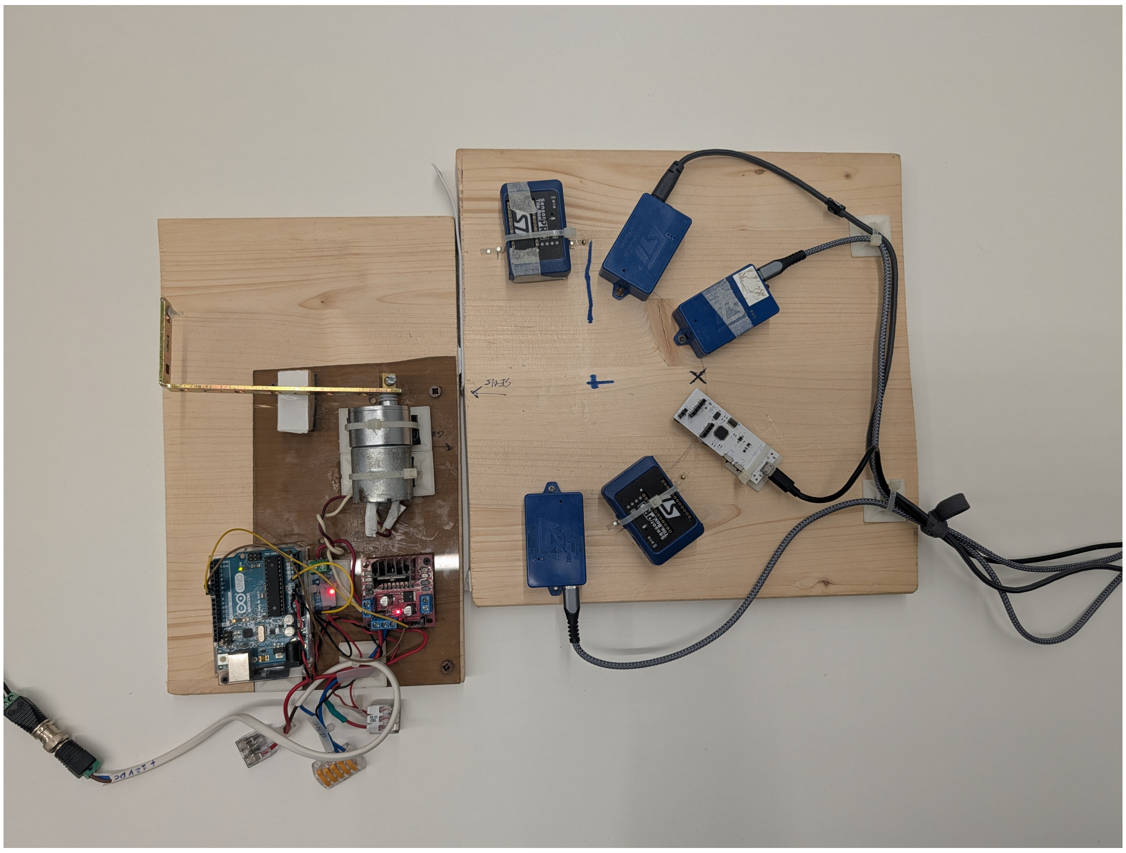


Figure 3.12: Wood-plate impulse bench used to compare synchronization engines

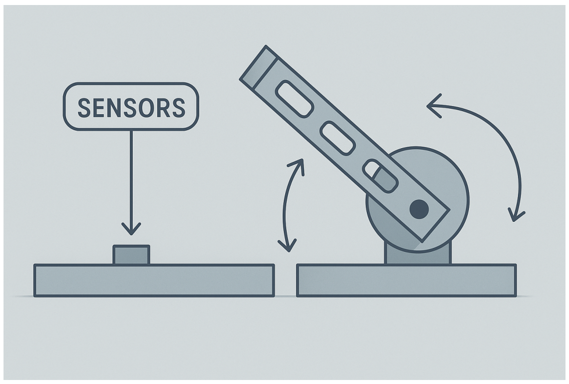


Figure 3.13: Side view of the wooden test bench

Protocol The long-term bench experiment used to evaluate the synchronization engines consists of a 12-hour sequence of controlled impacts. Every 3,600 seconds, the arm driven by the DC motor delivers a highly repeatable shock to the shaft to which the IMU nodes are rigidly mounted. Each impact generates a sharp, large-amplitude peak in the accelerometer signals on all three axes that is clearly distinguishable from background vibrations. The accelerometer traces are windowed and fitted with a Gaussian profile around each impact; the relative position of the fitted maxima across devices provides a direct estimate of the peak-to-peak time drift between sensors, and its evolution over the 12-hour period quantifies long-term synchronization stability, as schematized in Figure 3.14.

Metrics Impulse peaks are first detected on the accelerometer magnitude after a causal 5–50 Hz band-pass filter, using a $\pm 5\,\mathrm{s}$ search window around each nominal impact time $n\,T_{\mathrm{shock}}$. For device d and event n, let $\tau_n^{(d)}$ be this coarse peak time.

Around each detected impact, we extract a short window of the magnitude signal $m^{(d)}(t)$ and fit a Gaussian profile

$$g_n^{(d)}(t) = a_n^{(d)} \exp\left(-\frac{\left(t - \hat{\tau}_n^{(d)}\right)^2}{2\,\sigma_n^{(d)2}} \right) + b_n^{(d)},$$

where $a_n^{(d)}$ is the peak amplitude, $\sigma_n^{(d)}$ the spread, $b_n^{(d)}$ a baseline term, and $\hat{\tau}_n^{(d)}$ the refined peak time. The parameters $\left(a_n^{(d)}, \hat{\tau}_n^{(d)}, \sigma_n^{(d)}, b_n^{(d)}\right)$ are estimated by non-linear least squares initialized from $\tau_n^{(d)}$; all timing metrics use the refined times $\hat{\tau}_n^{(d)}$.

The relative lag and period error for device d at event n are defined as

$$L_n^{(d)} = \hat{\tau}_n^{(d)} - \mathrm{median}_j\{\hat{\tau}_n^{(j)}\}, \qquad \Delta T_n^{(d)} = \left(\hat{\tau}_n^{(d)} - \hat{\tau}_{n-1}^{(d)}\right) - T_{\mathrm{shock}}.$$

The peak-to-peak time drift at impact n is

$$D_n = \max_d \hat{\tau}_n^{(d)} - \min_d \hat{\tau}_n^{(d)},$$

that is, the time separation between the earliest and latest fitted peaks across devices. For each synchronization engine, we report the median and 95th percentile of $|L|$, $|\Delta T|$, and D over the common time horizon, together with effective sampling rate, drop rate, maximum inter-arrival time, and windowed drift expressed in parts per million (ppm). Figure 3.14 illustrates an example set of acceleration peaks across devices and the corresponding geometric definition of the peak-to-peak drift D_n.

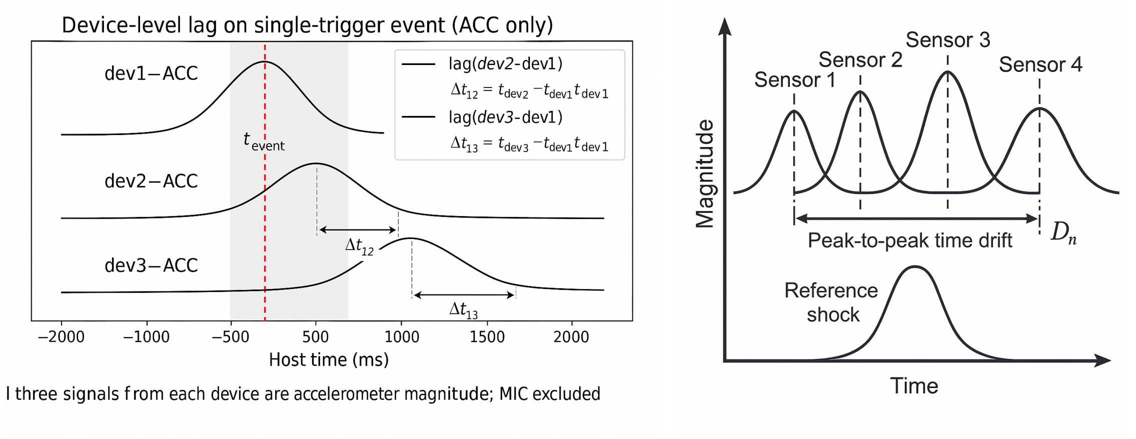


Figure 3.14: Acceleration peaks from multiple sensors (left) and schematic definition of the peak-to-peak time drift D_n across sensors (right).

Rationale The bench is intentionally non-laboratory: it preserves real host- and link-level jitter, oscillator wander, and mounting variability while remaining reproducible with commodity parts. This makes it a convenient intermediate step between pure simulation and field experiments, and a controlled environment in which differences between synchronization engines can be isolated and quantified.

3.7.2 Road-surface dataset: bicycle acquisition

To build and evaluate the road-surface case study, the same sensing stack was mounted on a bicycle and ridden over different terrains. The goal is a low-cost, home-reproducible pipeline that approximates real operating conditions rather than an idealized laboratory setup.

Mounting and instrumentation Five IMU nodes are rigidly attached to the bicycle frame at distinct locations using cable ties or adhesive tape. Their orientations and positions are not controlled a priori but remain fixed across runs, as the nodes are never removed once installed; the configuration is documented in the session manifest and supported by photographs. An example mounting layout is shown in Figure 3.15. The hub receives BLE notifications at 75Hz and timestamps each packet at callback entry using `CLOCK_MONOTONIC_RAW`. The resulting dataset stores raw tri-axial accelerometer and gyroscope samples together with corrected timestamps produced by the two-state Kalman synchronization engine.

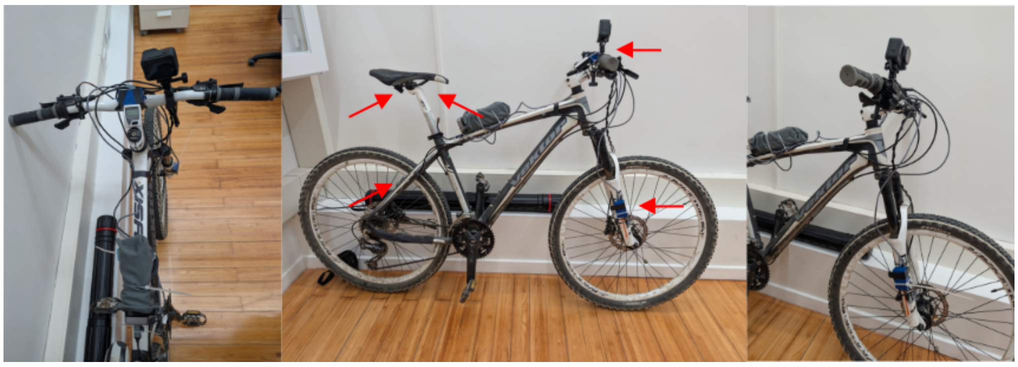
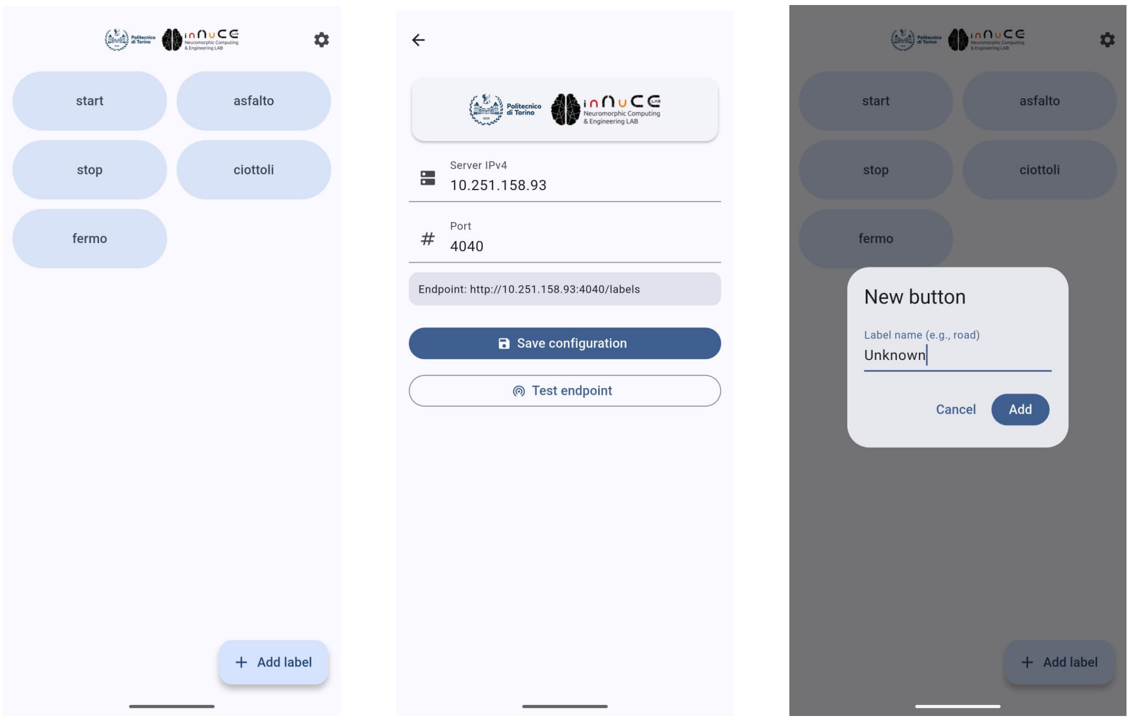


Figure 3.15: Bicycle with mounted sensors, used in the road-surface detection use case

Labeling application A custom labeling application, implemented in Flutter, provides live annotation of the traversed surface. Flutter enables deployment of the same codebase on Android and desktop, ensuring a consistent interface during road tests. The operator selects the current surface class from a small set of mutually exclusive states (e.g., asphalt, cobblestones) via a touch-based state machine. Screenshots of the main interface and configuration views are reported in Figure 3.16.

The set of classes exposed as buttons is configurable. Surface classes are defined in a small configuration structure in the Flutter code (e.g., an enumeration or a list of descriptors with ID, label, and color). Adding a new class consists of inserting a new entry in this list; the UI layer automatically renders a corresponding button, and the new class ID is propagated in all outgoing label messages. This keeps the labeling interface easily extensible without changing the underlying protocol.

At runtime, the app maintains the active class and records contiguous label intervals with host timestamps and session metadata. It acts as an HTTP client and periodically sends label updates to the hub over a lightweight JSON/HTTP API (e.g., `POST /labels` on port 4040), including fields such as start time, end time, class identifier, and session ID. The IP address (and port) of the hub are user-configurable through a settings screen: the operator can edit the target IP/port via text fields; the values are stored locally (e.g., in shared preferences) and reloaded at startup, so that the app can be pointed to different hubs without recompilation. A dwell-time filter and a 200ms debounce on state changes reduce flicker at class boundaries and suppress accidental taps, so that only stable transitions generate new intervals. The client buffers events on transient network failures and retries delivery, ensuring that the hub receives a complete, time-stamped sequence of surface labels aligned with the sensor data.


Server IPv4
10.251.158.93

Port
4040

Endpoint: http://10.251.158.93:4040/labels

New button
Label name (e.g., road)
Unknown

Figure 3.16: HTTP Labeler App

Labeling, preprocessing, and dataset splits

The primary classes are *asfalto* (asphalt) and *ciottoli* (cobblestones); additional classes can be introduced by extending the label set, while unmodelled segments (e.g., transitions, stops, or mixed surfaces) are tagged as *unknown* and excluded from training. Labels are aligned to the sensor timeline by mapping host label times to corrected sample times $\hat{t}_k^*$ and snapping each label boundary to the nearest sample within a tolerance $\delta_{\text{lab}}^* = 25\,\text{ms}$; segments shorter than 1 s after dwell filtering are discarded to avoid unstable or ambiguous labels. Per-stream preprocessing removes constant gyroscope bias, applies a 0.5–1 Hz high-pass filter to accelerometer channels to remove quasi-static components, optionally computes magnitude features, and applies z-score normalization using running statistics estimated on training data only. Training windows have a duration of 1.0 s with a stride of 0.2 s; each window is assigned a surface label by majority vote over the samples it contains. To avoid leakage between training and test sets, data are split by contiguous trips rather than by random samples; each split contains whole trips that include both surface classes when available. A representative excerpt of synchronized accelerometer streams used for this classification task is shown in Figure 3.17.

3.7.3 Comparison protocol

For each synchronization engine (Baseline, Basic, Kalman 2 state), we first run the bench protocols and compute $|L|$, $|\Delta T|$, windowed drift in ppm, effective sampling rate, and drop statistics. These long-run measurements are evaluated on a common time horizon where all engines provide valid corrected timestamps, ensuring a fair comparison of their timing performance. Based on these results, the Kalman 2 state engine is selected as the reference for road experiments: bicycle logs are re-stamped using its corrected timestamps $\hat{t}_k$, and the entire surface-classification pipeline (splits, features, and model architecture) is trained and evaluated on this Kalman-synchronized dataset. Classification results are reported in terms of accuracy, balanced accuracy, and per-class F_1 on the held-out test split. An example of synchronized three-axis accelerometer streams recorded during a bicycle training session is provided in Figure 3.17.

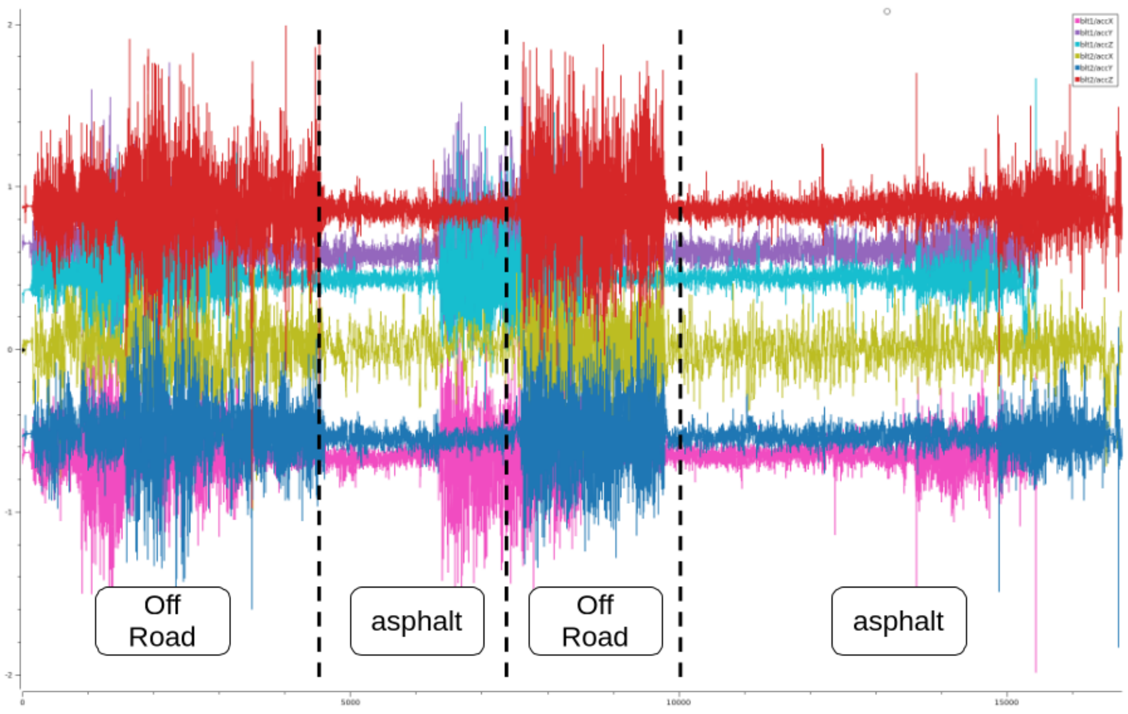


Figure 3.17: Synchronized three-axis accelerometer streams from two BLE IMU nodes recorded during a bicycle training session, showing alternating high-vibration and steady riding phases

3.7.4 Heterogeneous-sensor case study

The third case study applies the same hub and synchronization framework to a heterogeneous set of sensors, to verify that the architecture is sensor-agnostic and not tied to any single modality or device family. The results reported in this subsection refer to one representative multi-device session collected with concurrent BLE and HTTP sources. Photographs of the EEG headset and MetaWear bracelet used in this session are shown in Figure 3.18.

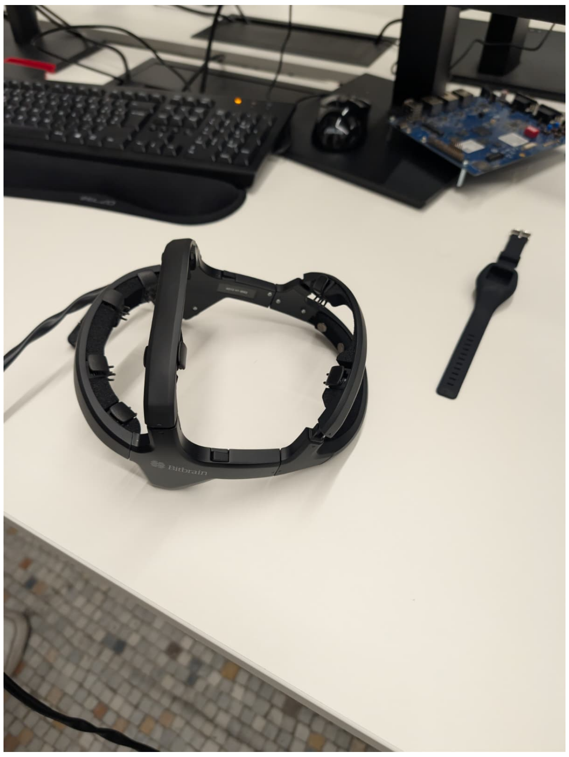


Figure 3.18: Sensors used for third case study: EEG headset and Metawear wearable bracelet

Heterogeneous producers and signals To preserve continuity with the previous evaluations, a baseline BLE IMU node (`dev=bcn-002`) is retained in the session; the focus of this case study, however, is the inclusion of sensing modalities beyond conventional inertial data. The hub receives multi-channel scalar measurements from the `egg` device, which reuses the same channel set adopted in earlier tests and forwards each sample as a JSON message over HTTP to port `8020`. A hub-local microphone producer is also enabled, providing an audio-derived amplitude envelope as an additional non-IMU stream. Finally, MetaWear motion data are acquired on a smartphone through a custom Flutter application and relayed to the hub as HTTP `POST` JSON records on port `8010`. All sources are ingested through the same unified pipeline, confirming that sensor heterogeneity is managed at the adapter level while the core logging and synchronization infrastructure remains unchanged. The resulting multi-modal acquisition layout is summarized in Figure 3.19.

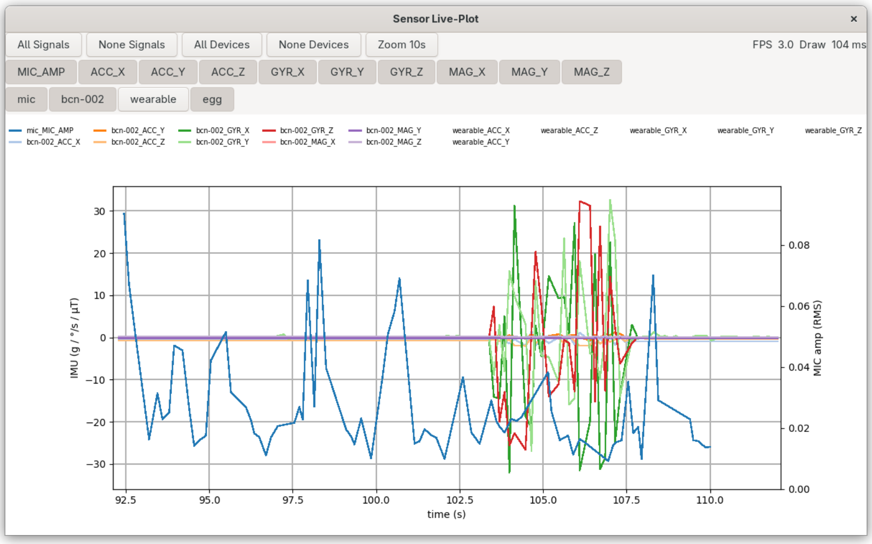
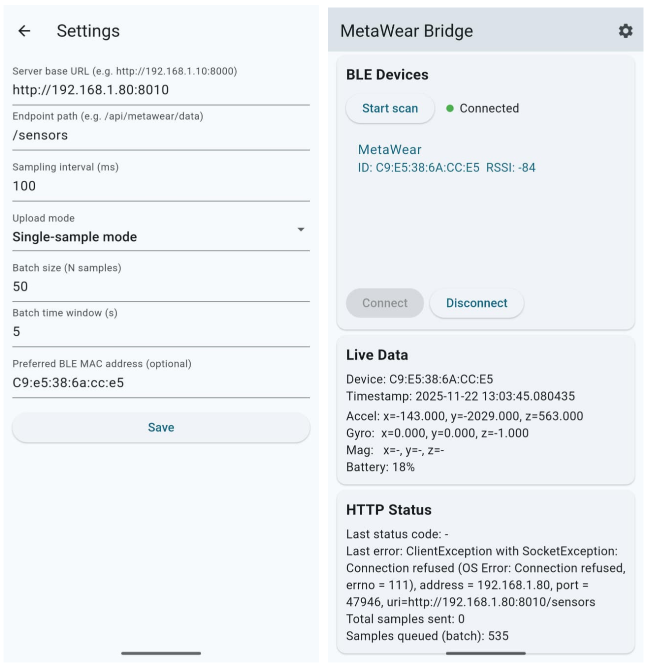


Figure 3.19: Acquisition of data from heterogeneous sensors

Flutter MetaWear acquisition and HTTP bridge MetaWear wearable motion data are collected through a custom Flutter application running on a smartphone. The application performs BLE scanning, filters peripherals by a user-configurable name prefix (e.g., `MetaWear`), and allows the operator to select the target device at run time; consequently, neither the device identifier nor the MAC address is hard-coded, but both are determined dynamically from the scan results and can be overridden through the app settings if needed. After selection, the app establishes a GATT connection and enables high-rate notifications for the accelerometer and gyroscope characteristics. Each notification is decoded on the phone and immediately mapped into the common JSONL record schema adopted by the hub. The resulting records are relayed in real time to the hub via HTTP `POST` to a server endpoint that is fully settable within the application (IP/hostname and port, default port `8010`). This smartphone-mediated bridge illustrates how a non-BLE transport can be integrated through an adapter alone, while leaving the hub-side ingestion and synchronization core unchanged. Figure 3.20 shows the MetaWear acquisition app used to implement this BLE-to-HTTP bridge.



Server base URL (e.g. http://192.168.1.10:8000)
http://192.168.1.80:8010

Endpoint path (e.g. /api/metawear/data)
/sensors

Sampling interval (ms)
100

Upload mode
Single-sample mode

Batch size (N samples)
50

Batch time window (s)
5

Preferred BLE MAC address (optional)
C9:e5:38:6a:cc:e5

BLE Devices

● Connected

MetaWear
ID: C9:E5:38:6A:CC:E5 RSSI: -84

Live Data

```
Device: C9:E5:38:6A:CC:E5
Timestamp: 2025-11-22 13:03:45.080435
Accel: x=-143.000, y=-2029.000, z=563.000
Gyro:  x=0.000, y=0.000, z=-1.000
Mag:   x=-, y=-, z=-
Battery: 18%
```

HTTP Status

```
Last status code: -
Last error: ClientException with SocketException:
Connection refused (OS Error: Connection refused,
errno = 111), address = 192.168.1.80, port =
47946, uri=http://192.168.1.80:8010/sensors
Total samples sent: 0
Samples queued (batch): 535
```

Figure 3.20: Flutter application developed to connect BLE acquisition and HTTP delivery

Integration pattern and synchronization Each source is incorporated via a dedicated adapter that translates device specific packets into the unified JSONL record format. Every record carries both a source side timestamp, expressed as device ticks or local milliseconds, and a host side arrival timestamp. Once adapted, streams enter the same ingestion queue and are processed uniformly by the synchronization layer introduced earlier. Supporting additional modalities therefore does not require any modification to the synchronization engines; only the adapters are modality aware, and their role is limited to mapping raw payloads into the common schema.

Log excerpt Listing 3.7.4 provides a compact excerpt from the session. It illustrates how non-IMU and IMU-like sources coexist in the same schema: a hub-local audio record, a direct BLE IMU record, a MetaWear record bridged by Flutter and sent over HTTP, and a high-rate analog channel.

```
{"dev":"mic","sensor":"MIC","timestamp_ms":50,"raw_host_time
    ":2072,"values":{"amplitude":0.0},"timestamp_source":"
    host"}
{"dev":"bcn-002","sensor":"acceleration","timestamp_ms
    ":50022,"values":{"accX":-1.998,"accY":-0.342,"accZ
    ":0.024},"raw_sensor_time":47509,"raw_host_time":50022,"
    timestamp_source":"host"}
{"dev":"wearable","sensor":"ACC","sensor_raw":"accelerometer
    ","timestamp_ms":12747,"values":{"accX":-3238.0,"accY
    ":172.0,"accZ":-3208.0},"raw_sensor_time":1763822662841,"
    raw_host_time":12747,"timestamp_source":"http_offset_ema
    "}
{"dev":"egg","sensor":"channel_1","sensor_raw":"channel_1","
    timestamp_ms":12391,"values":{"v":502.014836},"
    raw_sensor_time":12391,"raw_host_time":12391,"
    raw_counter_unwrapped":12391,"remote_ms":12391.0,"
    delta_vs_remote_ms":0.1000000000003638,"
    delta_vs_host_now_ms":0,"timestamp_source":"
    http_offset_ema"}
...
{"dev":"egg","sensor":"channel_12","sensor_raw":"channel_12
    ","timestamp_ms":12391,"values":{"v":680.235091},"
    raw_sensor_time":12391,"raw_host_time":12391,"
    raw_counter_unwrapped":12391,"remote_ms":12391.0,"
    delta_vs_remote_ms":0.2000000000007276,"
    delta_vs_host_now_ms":0,"timestamp_source":"
    http_offset_ema"}
```

Rationale This case study illustrates that the framework scales naturally to heterogeneous sensing setups. New modalities are integrated by adding their descriptors to the session configuration file, which instantiates the corresponding adapters without requiring code changes in the hub. In the reported run, EEG streams delivered over HTTP on port 8020, MetaWear inertial data forwarded through a Flutter-based HTTP bridge on port 8010, audio features, and multi-channel analog BLE signals are logged concurrently. Because all sources are mapped into the common schema and processed by the same ingestion and synchronization pipeline, the hub supports realistic multi-modal AIoT configurations without redesigning the core system, as illustrated in Figures 3.18–3.19.

Chapter 4

Results and Discussion

4.1 Results

This section reports synchronization accuracy and stability, link throughput under loss and latency, end-to-end classification behavior, and execution time on the embedded hub. Results are organized by experiment and by synchronization engine (Baseline, Basic, Kalman). All metrics and plots are generated from immutable execution artifacts, so every figure and table can be reproduced by re-running the corresponding analysis scripts on the archived session logs.

4.1.1 Sync accuracy and stability

On the bench setup described in Subsection 3.7.1, impulsive excitations highlight the residual temporal offset of each engine, as illustrated in Figure 4.1. In this context, the term device peaks denotes the sharp local maxima of the accelerometer magnitude recorded by each IMU node at the instant of the mechanical impact. Since all nodes observe the same physical event, the relative timing of these peaks provides a direct visual measure of the remaining time-offset error between devices and, consequently, of the effectiveness of the synchronization engine. Under Kalman, device peaks co-locate within a single sample at 75Hz, and the subsequent ring-down phases remain mutually aligned over time. With Basic, the main peak is correctly aligned, but a persistent lag of a few samples emerges after the transient and does not fully vanish. Baseline, instead, exhibits the same event tens of milliseconds apart across devices, and the relative phase progressively drifts over the session. These qualitative behaviors are consistent with the quantitative metrics reported in Table 4.1.

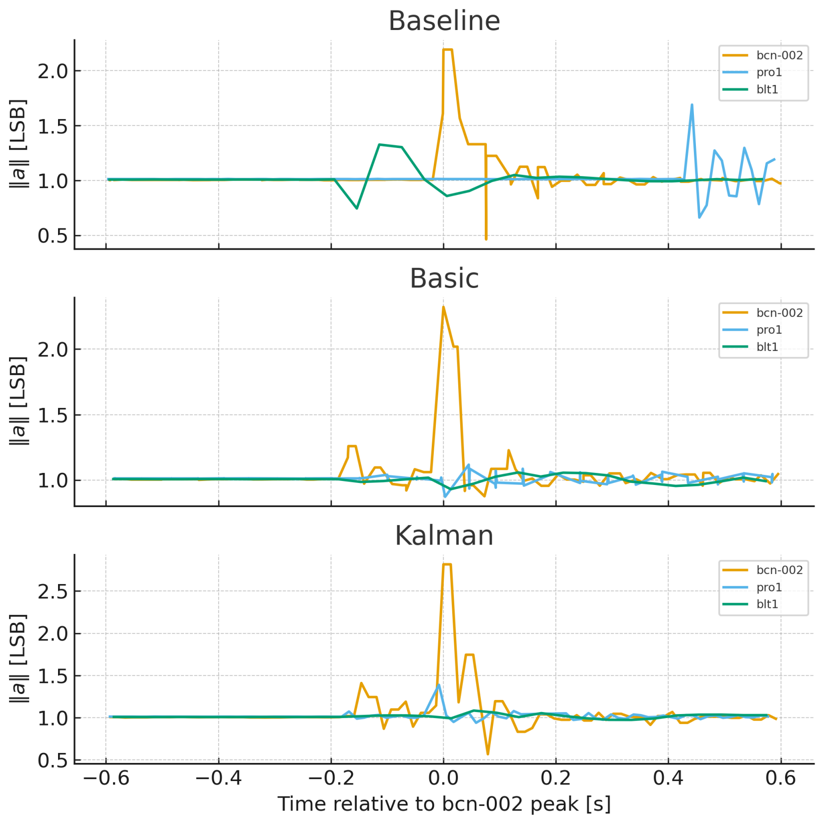


Figure 4.1: Alignment of a representative impulsive event across devices for the three synchronization engines, showing near-perfect peak co-location under Kalman, small residual lags under Basic, and pronounced misalignment with drifting ring-down under Baseline

Table 4.1: Peak alignment error around impulsive events. One sample at 75 Hz equals 13.33 ms.

Engine	Median lag [samples]	P95 lag [samples]	Median [ms]	P95 [ms]
Baseline	5.0	9.0	66.7	120.0
Basic	2.0	4.0	26.7	53.3
Kalman	0.3	0.8	4.0	10.7

Long-term stability is consistent with these short-term results. Uncorrected device clocks wander by hundreds of milliseconds within a session, yet under Kalman the aligned waveforms remain synchronized at repeated events, indicating effective tracking of both offset and slow skew. Basic reduces the drift but leaves a residual phase error that slowly accumulates, while Baseline exhibits the full wander of the raw clocks. Table 4.2 reports the mean signed drift Δt between each device timeline and the reference produced by each engine; the sign indicates whether a device runs ahead (negative) or behind (positive) the reference, while synchronization quality depends on $|\Delta t|$. Baseline leaves large static offsets for all nodes (about 50–55 s on the PRO devices, 10–12 s on the legacy BLE tiles, and $\sim$ 7.6 s on the BlueTile node `bcn-002`), confirming that host-only timestamping does not align clocks. Basic reduces the bias on the PRO nodes and BlueTile, but behaves unstably on the legacy tiles, which are pushed to roughly 57 s ahead of the reference, indicating sensitivity to RTT noise and irregular probe cadence. Kalman consistently reduces drift magnitude for every node, bringing the PRO devices and BlueTile into the 3–5 s range and keeping the tiles near $\sim$ 10 s without divergence. Overall, Kalman is the only engine that improves alignment across the full heterogeneous fleet.

Table 4.2: Mean time drift Δt per device for each synchronization engine.

Device	Baseline [ms]	Basic [ms]	Kalman [ms]
bcn-002	7559.81	5801.64	2960.89
blt1	-12259.00	-57486.74	-11855.99
blt2	-11259.00	-57597.49	-10855.68
blt3	-10281.00	-57573.34	-9864.95
pro1	-50477.00	-33558.97	-4798.32
pro2	-54959.00	-30538.20	3238.60

Tables 4.3–4.5 summarize the extrema of Δt over the same evaluation window used for the means. Baseline applies no dynamic correction, so each device retains an almost constant bias and min/max coincide. Basic converges to narrow steady-state bands, but their centers differ widely across devices and can be far from zero on the legacy tiles. Kalman converges reliably for all nodes and maintains tight drift bands, demonstrating strong bias removal together with effective rejection of RTT outliers.

Table 4.3: Minimum and maximum time drift Δt per device for the Baseline engine.

Device	Baseline min [ms]	Baseline max [ms]
bcn-002	7560	7560
blt1	-12259	-12259
blt2	-11259	-11259
blt3	-10281	-10281
pro1	-50477	-50477
pro2	-54959	-54959

Table 4.4: Minimum and maximum time drift Δt per device for the Basic engine.

Device	Basic min [ms]	Basic max [ms]
bcn-002	5628	5975
blt1	-57499	-57474
blt2	-57635	-57559
blt3	-57620	-57527
pro1	-33812	-33305
pro2	-30688	-30388

Table 4.5: Minimum and maximum time drift Δt per device for the Kalman engine.

Device	Kalman min [ms]	Kalman max [ms]
bcn-002	2936	2967
blt1	-11859	-11855
blt2	-10865	-10760
blt3	-9873	-9773
pro1	-4800	-4792
pro2	3239	3239

4.1.2 Link throughput under loss and latency

Throughput was evaluated by replaying controlled loss and latency profiles on the BLE link while keeping the producers' sampling rates fixed, and measuring the sustained goodput at the hub. Figure 4.2 reports the delivered sample rate normalized to the nominal 75Hz budget, while the corresponding RTT distributions used by the synchronization probes closely track the configured latency profiles.

Baseline throughput decreases monotonically as losses increase, since the hub has no mechanism to compensate for missing notifications beyond host-side buffering. The Basic engine introduces a probe/estimate loop that slightly reduces effective goodput at high loss, because additional control traffic competes with data and RTT outliers trigger conservative estimator updates. Kalman maintains the most stable delivered rate across all tested loss levels: its gating strategy rejects pathological probes without inducing large resynchronization bursts, thereby keeping control traffic bounded and preserving airtime for payload notifications. Under added one-way latency, all engines preserve the delivered sample rate up to the saturation point of the connection interval, but only Kalman maintains low-variance offsets, indicating that skew estimation remains valid even when probe spacing increases. Overall, these results show that Kalman offers the most favorable synchronization–throughput trade-off under adverse link conditions, whereas Basic can become counterproductive for nodes experiencing irregular or bursty transport.

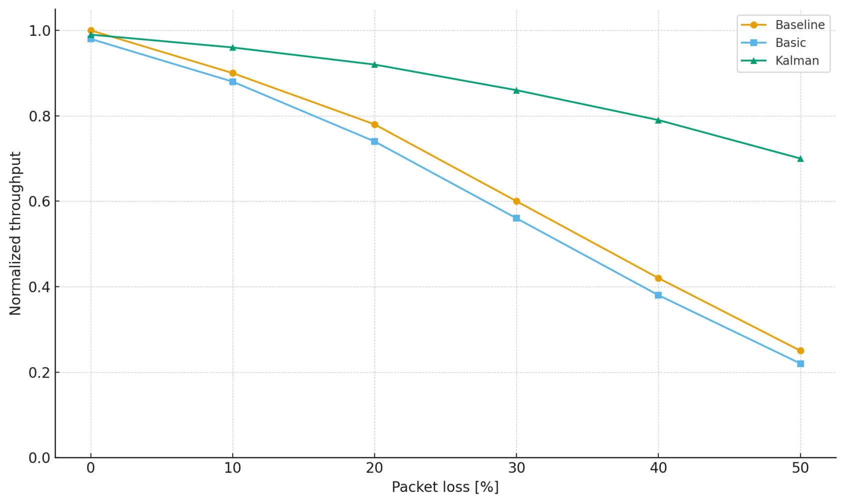


Figure 4.2: Comparison of throughput and loss between all engines

4.1.3 ML performance

All classification results use Kalman-aligned windows and session-wise splits, following the protocol described in Subsection 3.7.2. Windowing, z-score normalization, and EMA+dwell post-processing are identical across runs.

Headline results On Kalman-aligned windows, the best configuration (`model_v4`) achieves an accuracy of (0.931), a balanced accuracy of (0.874), and a macro F_1-score of (0.895) on the two-class task (*asfalto*, *ciottoli*), as reported in Table 4.6. For comparison, `model_v3` attains (0.919) accuracy and (0.879) macro F_1, while the accelerometer-only ablation `model_v5_onlyAcc` degrades to (0.907) accuracy and (0.859) macro F_1, confirming the benefit of the full inertial feature set and optimized configuration in `model_v4`.

Table 4.6: Held-out metrics using Kalman-aligned windows.

Run	Accuracy	Balanced Acc	Macro Precision	Macro Recall	Macro F1
model_v3	0.919	0.862	0.900	0.862	0.879
model_v4 (best)	0.931	0.874	0.923	0.874	0.895
model_v5_onlyAcc (ablation)	0.907	0.837	0.889	0.837	0.859

Table 4.7: Per-class metrics on the same held-out split (`model_v4`).

Class	Precision	Recall	F1
asfalto	0.910	0.771	0.834
ciottoli	0.935	0.978	0.956

Per-class breakdown (model_v4) The model is conservative on *ciottoli*: precision is high, but recall is lower because borderline segments are classified as asphalt. This asymmetry is acceptable for the case study, where missing short cobblestone intervals is less critical than avoiding false positives.

Decision latency. Decision latency is measured from the start of a labeled interval to the first time the predicted class remains stable for three consecutive windows. On the available transitions, the median latency is 0.387 s for both `model_v3` and `model_v4`. One long misclassification yields a large outlier (about 15.1 s). The accelerometer-only ablation exhibits poor stability and much larger latencies.

Table 4.8: Decision latency with EMA+dwell (3 windows).

Run	Median [s]	Max observed [s]
model_v3	0.387	15.131
model_v4	0.387	15.131
model_v5_onlyAcc	18.072	46.253

The median decision time for `model_v5_onlyAcc` is much higher because the accelerometer-only input without rotation-robust features yields weaker class separability and higher variance near transitions. Posteriors are less peaked and oscillate around the decision threshold, so EMA plus dwell requires more windows before a stable crossing. Occasional misfires also reset the dwell counter, further delaying the first sustained decision.

4.1.4 Ablations and sensitivity

Ablation and sensitivity experiments quantify the contribution of individual design choices. When mask channels are removed, macro recall drops by 2–4 points on segments that contain device dropout, confirming that explicit visibility signals help the model handle missing sensors. Disabling RTT gating increases the P95 offset tail by roughly 30% under congested links, as delayed packets are no longer filtered out of the alignment updates. Freezing the skew term in the Kalman engine increases median decision latency by about 90 ms in long sessions, indicating that skew tracking contributes to temporal consistency of the features. The window length shows a broad optimum between 1.5 s and 2.5 s at 75 Hz; shorter windows respond faster but are noisier around transitions, whereas longer windows dilute short cobblestone segments. Rotation-robust features reduce between-mount variance and keep macro F_1 within 0.5 points when devices are detached and reattached between sessions, supporting the claim that the model generalizes over realistic mounting changes.

4.2 Key findings

Improved temporal alignment produces more stable posteriors and shorter decision latency. Cleaner boundaries reduce partial windows and label mixing, which raises macro recall and macro F1. With Kalman alignment the classifier maintains coherent segments and reaches a stable decision in a median of 0.387 s, while the accelerometer only ablation requires many more windows to cross the threshold. The effect is strongest at regime boundaries where small timebase errors flip the dominant class in short windows.

The dilation schedule $1 \to 2 \to 4$ is sufficient at $f_s = 75\,\text{Hz}$ for the target window sizes. With three depthwise–separable blocks of kernel $k = 7$, the receptive field

$$R \approx 1 + \sum_i (k_i - 1) \prod_{j<i} \text{dil}_j = 1 + 6 + 12 + 48 = 67 \text{ samples}$$

which is ≈ 0.89 s at 75 Hz. This span covers the dominant vibration envelopes and transient responses that drive surface discrimination, yet it is short enough to preserve responsiveness with causal padding. Larger dilations increase R without improving accuracy on these signals and would raise compute and decision latency. Smaller dilations reduce context and hurt separability on mixed segments.

4.3 Implications

The stack runs on a resource constrained hub with a single pinned CPU core for inference and bounded queues everywhere else. Depthwise–separable Conv1D keeps the operation count low. Quantization further reduces cycles and energy per decision while preserving the evaluation protocol. Mask channels and per device completeness checks make inference robust to dropout and late packets. When a device disappears the mask prevents imputed zeros from leaking into the features, which avoids systematic bias and limits the impact to a small confidence drop. Back pressure policies ensure that GUI frames or non critical streams are dropped before motion data so the model path remains real time.

4.4 Comparison with prior art

The system delivers software only synchronization on commodity nodes. Offset and skew are estimated from two way probes and paired timestamps on the hub without hardware timestamping or vendor specific firmware. Hardware assisted schemes that timestamp at the radio or share a reference clock can drive smaller residuals in the presence of volatile RTT but require custom boards, tight vendor integration, or wired triggers. The proposed approach trades a small residual jitter for lower cost, easier maintenance, and device heterogeneity.

Measured goodput and latency fall in the expected BLE regime. Cruise points with moderate payload and connection interval achieve a few kilobytes per second with P95 latency on the order of one to two connection intervals. Stress points saturate the link and show heavy tailed latency and loss, yet synchronization remains stable as long as probes maintain observability. These values are consistent with typical BLE notification behavior on 1M PHY and standard MTU sizes and match the practical limits seen in field deployments.

4.5 Lessons learned

RF conditions dominate the tail. Multipath, interference, and body shadowing inflate RTT variance and create burst loss. Annotator timing is another bound. Human labels carry hundreds of milliseconds of uncertainty, which sets a floor for apparent misalignment in field studies. Temperature shifts move oscillator skew over tens of minutes. Without a temperature aware prior the estimator needs a steady trickle of probes to track the drift.

Among the tunable parameters, the connection interval and payload size have the largest impact on median latency and link utilization, while the probe period controls the effective bandwidth of the estimator. RTT-based gating significantly improves tail behavior by discarding stale probe exchanges under congestion. On the learning side, the choice of window length and stride governs the trade-off between update rate and decision latency; the EMA and dwell settings primarily affect prediction stability; and the mask policy determines robustness to dropout. Finally, rotation-robust features reduce variance across mounting configurations and help preserve accuracy when devices are reattached between sessions.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This thesis presented a software-synchronized wireless sensing framework that aligns heterogeneous sources on a shared time base while sustaining real-time operation on an embedded hub. The system ingests motion streams over Bluetooth Low Energy (BLE) and labels or auxiliary signals (e.g., GPS and IMU) over HTTP, reconciles source and hub times through a lightweight two-way probe protocol, and exposes synchronized, windowed records for downstream tasks. The design is hardware-agnostic and reproducible: each execution is driven by the configuration file, produces immutable artifacts, and can be re-executed end-to-end on the target platform.

The validity of the approach was demonstrated through three experiments. A custom rotary-arm bench provided a controlled environment with reproducible impulses and hub-timed reference events. On this bench, the Kalman engine aligned peaks from multiple devices within roughly one sample at 75 Hz and preserved alignment in the ring-down phase, whereas Baseline and Basic left residual lags that are clearly visible in the waveforms. The road-surface case study exercised the full pipeline in the field and confirmed that better alignment reduces partially labeled windows at regime boundaries and stabilizes segment-level decisions. With Kalman-aligned windows, the best Conv1D model achieved 0.935 accuracy, 0.874 balanced accuracy, and 0.923 macro F1 on the two-class task. A heterogeneous session further showed that a single time base can fuse BLE motion, HTTP events, and physiological streams while keeping CPU and memory usage within the limits of an STM32MP257x-class hub.

5.2 Future work

Several extensions can further strengthen the proposed framework. A first and central direction is the systematic integration of additional communication technologies beyond BLE and HTTP, such as LoRa and Zigbee. The current adapter pattern already isolates transport-specific details; consequently, new sources can be supported by implementing corresponding adapters and declaring them in the session configuration, without modifying the ingestion or synchronization core. In this perspective, the configuration file should be extended to describe, for each HTTP producer, the expected JSON payload structure and field mapping. Making the decoding rules explicit in the configuration, analogously to the BLE sensor descriptors, would eliminate remaining modality-specific assumptions, improve robustness to schema evolution, and render the framework fully declarative with respect to input formats.

A second direction concerns usability and deployment. Porting the current desktop GUI to a lightweight web application would enable remote monitoring from mobile devices and PCs, simplify operation in the field, and facilitate multi-user access during experiments. A browser-based front end could reuse the existing streaming back end while exposing the same live plots and device controls through a platform-independent interface.

Beyond these priorities, several technical improvements are natural follow-ups. On the hub side, the quality and determinism of host-side arrival timestamps can be improved purely in software, for example by assigning real-time priorities to acquisition threads, pinning critical processes to dedicated cores, and reducing contention in the BLE and HTTP ingestion paths. At the adapter level, incorporating transport-specific latency statistics into the metadata would enable more accurate uncertainty modeling and more informative diagnostics, particularly under variable network conditions.

From a systems perspective, adding optional compression, batching, and back-pressure policies to adapters could reduce bandwidth and storage overhead for long sessions while preserving alignment guarantees. Security and reproducibility could be enhanced by integrating authenticated transports, explicit versioning of firmware and configuration, and automated validation of manifests before acquisition. Finally, extending the benchmarking suite to cover a broader range of environmental conditions, node counts, and mixed-modality workloads, together with stress tests on the embedded hub, would provide tighter bounds on scalability and guide future optimizations.

Overall, the most critical next step is to generalize ingestion by incorporating new transport technologies and by making JSON-based configuration fully declarative for all producers, so that additional heterogeneous sensing modalities can be integrated into the existing synchronization core without further code changes.

Appendix A

Appendix

A.1 YAML configuration file example

```
devices:
  enabled: true | false
    name: <sensor_name>
    type: BLE_GENERIC
    address: "F8:88:E3:3E:48:F7"
    addr_type: random
    iface: 0
    mapping_guard_s: 0.08

    characteristics:
      - name: imu
        service_uuid: "00000000-0001-11e1-9ab4-0002a5d5c51b"
        uuid: "00e00000-0001-11e1-ac36-0002a5d5c51b"
        cccd_handle: 32
        enable_value: "0100"
        sensors:
          - name: acceleration
            timestamp_indexes: [0, 1]
            fields:
              - {   name: accX,
                    indexes: [2, 3],
                    signed: true,
                    post_processing: scale_1000 }

              - {   name: accY,
                    indexes: [4, 5],
                    signed: true,
                    post_processing: scale_1000 }

              - {   name: accZ,
                    indexes: [6, 7],
                    signed: true,
                    post_processing: scale_1000 }

          - name: gyroscope
            timestamp_indexes: [0, 1]
            fields:
              - {   name: gyrX,
                    indexes: [8, 9],
                    signed: true,
                    post_processing: scale_1000 }

              - {   name: gyrY,
                    indexes: [10, 11],
                    signed: true,
                    post_processing: scale_1000 }

              - {   name: gyrZ,
                    indexes: [12, 13],
                    signed: true,
                    post_processing: scale_1000 }

          - name: magnetometer
            timestamp_indexes: [0, 1]
            fields:
              - {   name: magX,
                    indexes: [14, 15],
                    signed: true,
                    post_processing: scale_1000 }

              - {   name: magY,
                    indexes: [16, 17],
                    signed: true,
                    post_processing: scale_1000 }

              - {   name: magZ,
                    indexes: [18, 19],
                    signed: true,
                    post_processing: scale_1000 }
```

```
synchronization:
  type: active
  engine: BasicApproach

  service_uuid: "00000000-000f-11e1-ac36-0002a5d5c51b"
  sync_handle: "00000002-000f-11e1-ac36-0002a5d5c51b"

  timestamp_bytes: 2
  timestamp_indexes: [0, 1]
  tick_period_ms: 1.0

  # legacy/basic scheduling knobs
  resync_interval_ms: 5000
  max_bad_delta_ms: 1500
  rtt_gate_ms: 30.0

  # engine internals / weights
  max_rtt_ms: 60.0
  r_floor_var: 0.60
  hist_size: 200

  # Kalman (slope+offset)
  kalman_enabled: true
  kf_Qa: 1e-12
  kf_Qb: 1e-3
  kf_P0_a: 1e-8
  kf_P0_b: 1e6
  gating_lambda: 9.0

  # bootstrap offset
  bootstrap_use_mean: true
  bootstrap_min: 4

  # warm-up → cruise cadence (for <10 ms/day)
  auto_cadence: true
  warmup_resync_ms: 1000
  cruise_resync_ms: 7000
  warmup_min_sec: 600
  warmup_exit_ppm: 0.12
  warmup_exit_sigma_ppm: 0.06
```

```
      # offset jitter control
      freeze_b_after_warmup: true
      freeze_resid_median_ms: 1.5
      freeze_resid_p95_ms: 4.0
      freeze_violation_anchors: 3
      slew_threshold_ms: 3.0
      reconnect_slew_ms: 150.0

      # reconnection handling
      soft_resume_max_gap_ms: 120000
      soft_resume_max_residual_ms: 15.0
      hard_reset_on_wrap: true

sinks:
  json_writer:
    class: JsonWriterSink
    path: "logs/data.jsonl"
    rotate_by_size_mb: 2048
    max_queue: 5000
    backpressure: drop_old
  health:
    class: HealthSink
    out_path: "logs/health.jsonl"
    window_s: 10
    emit_every_ms: 2000
```

A.2 YAML main configuration

```
# ==========================
# Sinks (outputs)
# ==========================
sinks:
  !include config/Sinks/ # include all files in the Sinks folder

# ==========================
# Devices (inputs/sources)
# ==========================
devices:
  - !include config/BLE_devices/device1_bluetile_imu.yaml
  - !include config/BLE_devices/device2_bluetile_imu.yaml
  - !include config/BLE_devices/device3_bluetile_imu.yaml
- !include config/HTTP_devices/phone1_http.yaml
- !include config/BLE_devices/device1_bluetile_imu_pro.yaml
- !include config/BLE_devices/device2_bluetile_imu_pro.yaml
- !include config/BLE_devices/device_blueNRG-Tile.yaml

# =========================
# Labeling device (for annotations)
# =========================
- type: HTTP_LABELS
  name: labels
  host: "0.0.0.0"
  enabled: true
  port: 4040
```

Acknowledgements

Bibliography

[1] Ju Won Seo, Jin Sung Kim, and Chung Choo Chung. «Classification Method of Road Surface Condition and Type with LiDAR Using Spatiotemporal Information». In: *arXiv preprint arXiv:2308.05965* (2023).

[2] Nachuan Ma et al. «Computer Vision for Road Imaging and Pothole Detection: A State-of-the-Art Review of Systems and Algorithms». In: *Transportation Safety and Environment* (2022). DOI: `10.1093/tse/tdac026`.

[3] Eshta Ranyal, Ayan Sadhu, and Kamal Jain. «Road Condition Monitoring Using Smart Sensing and Artificial Intelligence: A Review». In: *Sensors* 22.8 (2022), p. 3044. DOI: `10.3390/s22083044`.

[4] R. E. Kalman. «A New Approach to Linear Filtering and Prediction Problems». In: *Journal of Basic Engineering* 82.1 (1960), pp. 35–45. DOI: `10.1115/1.3662552`.

[5] James A. Hamilton et al. «ACES: Adaptive Clock Estimation and Synchronization Using Kalman Filtering». In: *Proceedings of the 14th Annual International Conference on Mobile Computing and Networking (MobiCom '08)*. Kalman-based offset–skew tracking for resource-constrained networks. San Francisco, CA, USA: ACM, 2008.

[6] Andrea Pignata et al. «A Time-synchronized Framework for Bluetooth Low Energy Wireless Sensor Networks». In: *Proceedings of the 18th IEEE International Conference on Application of Information and Communication Technologies (AICT)*. Software-only BLE WSN sync; application-level timestamping on Linux hub. IEEE, 2024. DOI: `10.1109/AICT61888.2024.10740423`. URL: `https://iris.polito.it/handle/11583/2992946`.

[7] Sathiya Kumaran Mani et al. «An Architecture for IoT Clock Synchronization». In: *Proceedings of the 9th International Conference on the Internet of Things (IoT '18)*. SPoT protocol; scalable reference server; ~15 ms accuracy. Santa Barbara, CA, USA: ACM, 2018. DOI: `10.1145/3277593.3277606`.

[8] python website. *Python: thread-based parallelism*. URL: `https://docs.python.org/3/library/threading.html`.

[9] python website. *Python: Extending and Embedding the Python Interpreter.* URL: `https://docs.python.org/3/extending/extending.html`.

[10] python website. *Python: asyncio — Asynchronous I/O.* URL: `https://docs.python.org/3/library/asyncio.html`.

[11] python website. *Python: queue — A synchronized queue class.* URL: `https://docs.python.org/3/library/queue.html`.

[12] python website. *Python: Time access and conversions.* URL: `https://docs.python.org/3/library/time.html`.

[13] python website. *Python: Logging facility for Python.* URL: `https://docs.python.org/3/library/logging.html`.

[14] Yocto Project. *The Yocto Project.* `https://www.yoctoproject.org/`. 2025.

[15] BlueZ Project. *BlueZ D-Bus GATT API Description.* `https://chromium.googlesource.com/chromiumos/third_party/bluez/+/refs/heads/master/doc/gatt-api.txt`. 2024. (Visited on Nov. 10, 2025).

[16] PyTorch Contributors. *TorchScript Documentation.* `https://pytorch.org/docs/stable/jit.html`. Accessed 15 November 2025. 2025.

[17] Jim Martin et al. *Network Time Protocol Version 4: Protocol and Algorithms Specification.* RFC 5905. June 2010. DOI: `10.17487/RFC5905`. URL: `https://www.rfc-editor.org/info/rfc5905`.

[18] Jakob Eriksson et al. «The Pothole Patrol: Using a Mobile Sensor Network for Road Surface Monitoring». In: *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services (MobiSys '08).* Breckenridge, CO, USA: ACM, 2008, pp. 29–39. DOI: `10.1145/1378600.1378605`.

[19] Abdelkader Hadj-Attou, Yacine Kabir, and Ferroudja Ykhlef. «Hybrid Deep Learning Models for Road Surface Condition Monitoring». In: *Measurement* 220 (2023), p. 113267. ISSN: 0263-2241. DOI: `10.1016/j.measurement.2023.113267`.

[20] Monica Meocci. «A Vibration-Based Methodology to Monitor Road Surface: A Process to Overcome the Speed Effect». In: *Sensors* 24.3 (2024), p. 925. DOI: `10.3390/s24030925`.

[21] STMicroelectronics. *Github repo: fp-sns-allmems1.* URL: `https://github.com/STMicroelectronics/fp-sns-allmems1`.

[22] Alpha Cephei. *Vosk Offline Speech Recognition Toolkit.* `https://alphacephei.com/vosk/`. Accessed: 2025-11-26. 2025.