1. End-to-End Process (ELI8, Step-by-Step)

This section describes a complete, practical workflow to (1) build the serverless-visible dataset and deployable model on a Windows x86 machine where privileged measurements are available, and then (2) validate the same deployment idea on an ARM Raspberry Pi. The key rule is:

> **At deployment time we never use PMCs.** We only use serverless-visible signals like *duration, CPU time, memory limit, RSS/peak RSS, I/O bytes, and cold/warm flag.*

1.1. Definitions and What We Measure

Invocation. One **invocation** is one run of a workload under a chosen memory tier and a chosen cold/warm condition, producing one record (features + label + metadata).

Two feature views.

- **Privileged view (lab-only):** Intel PCM / PMC-derived information used only for ground-truth labeling and for the teacher model (Windows x86).
- **Serverless view (deployable):** OS-visible signals available in serverless-like environments:
  - `duration_ms`: wall-clock runtime for the invocation.
  - `cpu_time_ms`: process CPU time (user + kernel if available).
  - `mem_limit_mb`: configured memory tier.
  - `rss_mb`, `peak_rss_mb`: resident set size and peak.
  - `io_read_bytes`, `io_write_bytes`: bytes read/written during invocation.
  - `cold_start`: 1 if cold, 0 if warm.
  - `concurrency`: number of concurrent invocations (if tested).
  - `queue_delay_ms`: time waiting before start (if you simulate a queue).

Labels. We recommend two labels:

- **Energy per invocation** (preferred): `energy_joules`.
- **EDP per invocation** (optional): `edp` = `energy_joules` × `duration_s`.

Latency SLO. Define one latency SLO (example):

$$\texttt{SLO\_ms} = 200ms$$

An invocation violates the SLO if `duration_ms` > `SLO_ms`.

1.2. Common Setup for Both Platforms

These steps ensure the Windows and Raspberry Pi experiments are comparable.

1.2.1 Step 0: Choose Workloads and a Repeatable Invocation Format

1. Pick workload groups: **CPU-bound**, **Memory-bound**, **Mixed**.
2. For each workload, define an **input size** (or request) so that:
   - One invocation lasts long enough to measure (e.g., 50–500 ms).
   - The workload is deterministic (same input, same output).
3. Wrap each workload in a **single command** that can be executed from a harness:

```
run_workload --name=W --input=I --out=...
```

4. Make sure each invocation prints or saves:
   - workload name, input id, start timestamp, end timestamp
   - success/failure (exit code)

1.2.2 Step 1: Define Memory Tiers (Your Control Knob)

1. Define a small set of memory tiers (example):

$$\mathcal{M} = \{128, 256, 512, 1024, 2048\}MB$$

2. These tiers will be the only options your controller can choose.
3. For each tier, ensure you can **enforce** the memory limit:
   - On Windows: use containers or job objects (recommended: Docker with memory limit).
   - On Raspberry Pi: use containers/cgroups (Docker memory limit).

1.2.3 Step 2: Define Cold vs Warm Starts

1. **Warm start** means the execution environment is reused (same container/process context reused).
2. **Cold start** means the environment is recreated:
   - new container instance, or
   - clear OS cache effects as much as feasible, and/or
   - idle timeout between invocations (optional).
3. In the harness, store `cold_start` $\in \{0, 1\}$ explicitly.

1.2.4 Step 3: Create a Simple Invocation Harness (Core Tool)

Your harness must do the same thing every time:

1. Set memory tier $m \in \mathcal{M}$.
2. Decide cold or warm.
3. Start measurement (timers + resource counters).
4. Execute the workload once.
5. Stop measurement and write one JSON record for the invocation.

   Each JSON record must include at least:

```
{
  "platform": "windows" or "rpi",
  "workload": "cpu_X" / "mem_Y" / "mix_Z",
  "run_id": "...",
  "invocation_id": "...",
  "mem_limit_mb": 512,
  "cold_start": 0,
  "duration_ms": ...,
  "cpu_time_ms": ...,
  "rss_mb": ...,
  "peak_rss_mb": ...,
  "io_read_bytes": ...,
  "io_write_bytes": ...,
  "concurrency": 1,
  "queue_delay_ms": 0,
  "energy_joules": ... (windows teacher-labeling),
  "edp": ...
}
```

Important detail: one record per invocation. Never store aggregated statistics only. Always store per-invocation records; you can compute averages later.

2. Process A: Windows x86 (Build + Train + Test + Closed-loop Controller)

2.1. A1. Windows Environment Preparation

1. Ensure Intel PCM is installed and working on Windows:
   - Verify `pcm-power` produces power/energy readings.

- Record PCM sampling interval used.

2. Ensure you can enforce memory tiers:
   - Recommended: Docker Desktop + Windows containers or Linux containers with memory limits.
   - Verify: a container started with `--memory` actually caps memory.
3. Ensure your harness can read:
   - wall-clock time (high resolution if possible),
   - per-process CPU time,
   - RSS/peak RSS,
   - bytes read/written (process or container).

2.2. A2. Collect Training Data (Windows)

2.2.1 A2.1 Decide an Experiment Grid

Choose:

- Workloads W (CPU, Memory, Mixed).
- Memory tiers $\mathcal{M}$.
- Cold-start ratio (example: 20% cold, 80% warm).
- Repetitions per configuration (example: 30–100 invocations).

Example grid. For each workload w, for each tier m, run:

$$N_{warm} = 80, \quad N_{cold} = 20$$

2.2.2 A2.2 Run the Harness and Log Everything

For each invocation:

1. (Optional) Apply concurrency and queue delay if this invocation is part of a concurrency experiment.
2. If cold start:
   - start a new container OR restart the runtime environment
3. Start timers and collect initial resource counters.
4. Run the workload exactly once.
5. Stop timers and collect final resource counters.
6. Collect PCM energy for the invocation window:
   - Option 1: run `pcm-power` continuously and compute delta energy over start/end timestamps.
   - Option 2: start PCM sampling before invocation and stop after (less ideal but simple).
7. Write one JSON record for the invocation.

Tiny but critical detail: time alignment. Make sure your start/end timestamps are aligned with PCM sampling. If PCM gives cumulative energy counters, always compute:

$$\Delta E = E_{end} - E_{start}$$

Never mix units (Joules vs mJ).

2.3. A3. Build Two Datasets From the Same Records

From the logged invocation records:

1. Create the **privileged dataset**:
   - Features: PCM/PMC-derived + any derived metrics.
   - Labels: `energy_joules` (and optional EDP).
2. Create the **serverless dataset**:
   - Features: {`duration_ms`, `cpu_time_ms`, `mem_limit_mb`, `rss_mb`, `peak_rss_mb`, `io_read_bytes`, `io_write_bytes`, `cold_start`, . . .
   - Labels: `energy_joules` for training (offline). At deployment, labels are not used.

2.4. A4. Data Splits (No Leakage)

1. Define `run_id` (one data-collection session).
2. Split by workload and run:
   - **Train**: some workloads/runs
   - **Validation**: different runs (same workloads ok)
   - **Test**: different workloads and different runs
3. Mandatory: include a **workload-shift test**:
   - Train on CPU + Mixed, test on Memory (and vice versa).

2.5. A5. Train Baselines on Serverless Features (Windows)

Goal. Show the simple model (MLP) vs strong tabular model (HistGBDT) gap on **serverless-only features**.

1. Train MLP on serverless dataset to predict `energy_joules`.
2. Train HistGBDT on serverless dataset to predict `energy_joules`.
3. Evaluate on test sets:
   - RMSE / MAE / MAPE,
   - error by workload class (CPU / Memory / Mixed),
   - generalization under workload shift.
4. Measure inference overhead:
   - average prediction time per sample (microseconds/milliseconds),
   - memory footprint of the model if possible.

2.6. A6. Train the Teacher Using Privileged Features (Windows)

Goal. Build the best accuracy "oracle-like" predictor using privileged features. This model will not be deployed.

1. Train HistGBDT (or best method) on privileged dataset.
2. Evaluate teacher accuracy on privileged test sets.
3. For every invocation record, compute and store:

$$\hat{y}^{teacher} = f_{teacher}(x^{privileged})$$

4. Save teacher predictions alongside each invocation record as `teacher_energy_pred`.

2.7. A7. Train the Deployable Hybrid Model (Distilled) (Windows)

Goal. Train a deployable model that uses only serverless-visible features but learns from:

- true energy labels, and
- teacher soft labels (`teacher_energy_pred`).

1. Input: serverless features only.
2. Targets:
   - ground truth energy y,
   - teacher prediction $\hat{y}^{teacher}$.
3. Use a combined loss (conceptually):

$$\mathcal{L} = \lambda \cdot \hat{y} - y^2 + (1 - \lambda) \cdot \hat{y} - \hat{y}^{teacher2}$$

4. Validate on validation split and pick λ that best improves workload-shift performance.
5. Measure inference overhead again and compare to MLP and HistGBDT.

2.8. A8. Build the Serverless Controller (Memory Tier Selection)

Inputs at runtime (serverless-visible only). `cpu_time`, `duration` (or partial signals if early), `mem_limit` candidate tiers, `cold_start`, RSS/I/O signals.

Controller objective. Choose memory tier $m \in \mathcal{M}$ that:

- meets latency SLO, and
- minimizes predicted energy (or EDP).

2.8.1 A8.1 How to Predict Latency and Energy for Each Tier

Because memory tier affects latency and energy, you need to predict both under each tier.

Practical approach:

1. Train two deployable models on serverless features:
   - $f_E(x, m)$: predicts energy at tier m,
   - $f_T(x, m)$: predicts latency at tier m.
2. At decision time, for each tier m:
   - build a feature vector with `mem_limit_mb = m`,
   - compute $\hat{E}_m = f_E(x, m)$,
   - compute $\hat{T}_m = f_T(x, m)$.

2.8.2 A8.2 Decision Rule

1. Define feasible tiers:
$$\mathcal{M}_{ok} = \{m \in \mathcal{M} \mid \hat{T}_m \le \texttt{SLO\_ms}\}$$
2. If $\mathcal{M}_{ok}$ is non-empty, choose:
$$m^\star = \arg\min_{m \in \mathcal{M}_{ok}} \hat{E}_m$$
3. If $\mathcal{M}_{ok}$ is empty, choose:
$$m^\star = \arg\min_{m \in \mathcal{M}} \hat{T}_m$$
   (i.e., the tier predicted to be fastest, to reduce violations.)

2.9. A9. Closed-loop Evaluation on Windows

Baselines. Compare your controller against:

- **Fixed tier:** always choose one tier (e.g., 512 MB).
- **Always-lowest:** always choose minimum tier (best energy, worst SLO).
- **Always-highest:** always choose maximum tier (best SLO, worst energy).
- **Oracle bound (offline only):** choose tier using true measured outcomes (not deployable).

Protocol.

1. Generate an invocation sequence:
   - mixture of CPU/Memory/Mixed,
   - include cold starts with fixed probability,
   - optionally include concurrency levels.
2. For each invocation:
   - controller selects $m^\star$,
   - run invocation at $m^\star$,
   - log actual duration, actual energy (PCM), actual SLO violation.

Report.

- Energy savings (%) vs baselines
- EDP savings (%) vs baselines
- SLO violation rate (%)
- Tail latency (p95, p99) of duration
- Stability under workload shift (performance per workload class)

3. Process B: Raspberry Pi ARM (Validation of Deployment Claims)

3.1. B1. Raspberry Pi Environment Preparation

1. Fix CPU frequency scaling if possible (to reduce noise):
   - choose a stable governor (performance or fixed frequency).
2. Manage thermals:

    - use a heatsink/fan if available,
    - log CPU temperature during experiments.
3. Install container support (recommended):
    - Docker on Raspberry Pi (or direct cgroups).
4. Verify memory limiting works:
    - run a memory-hungry test and confirm it is capped.

3.2. B2. Decide What "Validation" Means

We recommend validating three claims on Pi:

1. **Deployability:** the model runs with only serverless-visible signals.
2. **Low overhead:** inference time is small even on ARM.
3. **Controller behavior:** the memory-tier controller reduces energy proxy (or measured energy) while meeting SLO.

Energy measurement options on Pi.

- **Best: external power meter** (inline USB power meter or dedicated logger) to compute energy.
- **If not available:** validate using latency + predicted energy from your model (weaker but still useful).

3.3. B3. Port the Harness (Same JSON Schema)

1. Use the same harness steps as Windows:
    - set memory tier,
    - choose cold/warm,
    - measure duration, CPU time, RSS/peak RSS, I/O,
    - write one JSON record per invocation.
2. Keep the JSON keys identical to Windows so analysis scripts work unchanged.

3.4. B4. Two Validation Modes

3.4.1 Mode 1: "Train on Windows, Run on Pi" (Portability Test)

This is the strongest story if it works.

1. Export the deployable model trained on Windows serverless features.
2. On Pi, extract the same serverless features per invocation.
3. Run inference to predict energy/latency for each tier and select $m^{\star}$.
4. Evaluate:
    - SLO violation rate,
    - p95/p99 latency,
    - energy (if measured externally),
    - inference overhead.

Tiny detail: feature scaling/normalization. If your model expects normalized inputs:

- Save normalization parameters from Windows training,
- Apply the same normalization on Pi at runtime.

3.4.2 Mode 2: "Train on Pi, Run on Pi" (Deployment Feasibility Test)

This is easier and still publishable.

1. Collect a smaller Pi dataset with the same serverless features.
2. Train a lightweight deployable model on Pi data (or train on Windows using Pi data).
3. Evaluate accuracy (if you have energy measurement) or controller outcomes (latency + energy proxy).

3.5. B5. Closed-loop Controller Evaluation on Pi

Protocol.

1. Run the same closed-loop protocol:
   - sequence of invocations,
   - cold starts at fixed probability,
   - same memory tiers.
2. Compare the same baselines:
   - fixed tier,
   - always-lowest,
   - always-highest.
3. Report the same metrics:
   - SLO violations,
   - tail latency,
   - energy or energy proxy,
   - inference overhead.

Tiny detail: repeat runs to handle noise. On Pi, always repeat each configuration multiple times and report confidence intervals or at least mean $\pm$ std.

3.6. B6. What Figures/Tables to Produce (Minimal Set)

- **Fig. 1 (Windows):** MLP vs HistGBDT vs Hybrid accuracy on serverless features + inference overhead.
- **Fig. 2 (Windows):** Closed-loop controller energy/EDP savings vs SLO compliance (p95/p99).
- **Fig. 3 (Pi):** Inference overhead on ARM + controller SLO compliance; optionally energy savings if measured.
- **Table 1:** Workload-shift generalization (train CPU test Memory, etc.).
- **Table 2:** Controller outcomes across workload classes (CPU/Memory/Mixed).

3.7. B7. Checklist (Do Not Skip These)

1. Log one JSON record per invocation (never only averages).
2. Keep identical feature names and units across platforms.
3. Record units explicitly: ms, MB, bytes, Joules.
4. Split by run and by workload to avoid leakage.
5. Report tail latency (p95/p99), not just averages.
6. Always compare against simple baselines (fixed tiers).
7. Measure inference overhead on both Windows and Pi.
8. For cold starts, explicitly mark `cold_start` and use a consistent definition.