FertiLuna: a privacy-preserving, on-device machine-learning system for menstrual-cycle analysis

A dual-model (vision + forest) architecture
shipped entirely to the web browser

Bijou Kachungunu
bijoukach2000@gmail.com

Gabriel Mougard
gabriel.mougard@gmail.com

We describe the design, training, and deployment of FertiLuna, a fertility-data analysis tool whose entire machine-learning stack runs inside the user's web browser. Two complementary model families work together: a depthwise-separable convolutional vision network *that digitizes chart screenshots, and a calibrated* random-forest classifier *paired with an isolation forest for out-of-distribution detection. Both are trained on physiologically grounded synthetic data, exported to ONNX, and served as same-origin static assets that are cached and version-controlled in the browser. No account is created, and no cycle datum ever leaves the device.*

Technical report • June 15, 2026

Source, PlantUML diagrams, and figures accompany this document.

Abstract

Health applications that handle reproductive data sit at an awkward crossroads: machine learning offers real analytical value, but the underlying records are intimately personal. FertiLuna resolves that tension by moving *all* inference to the client. Models are downloaded once, validated against a cryptographic checksum, cached in IndexedDB, and executed with ONNX Runtime Web compiled to WebAssembly. The system combines two stages with complementary inductive biases. The *vision* stage is a deterministic classical computer-vision pipeline (OpenCV-style segmentation, multi-axis OCR with RANSAC line fitting, and geometric marker detection), aided by a small PaddleOCR text recognizer. It reads a screenshot of a fertility-app chart and recovers per-day basal-body-temperature (BBT) and luteinizing-hormone (LH) values, together with the chart's temperature unit and its calendar table, removing the need for manual transcription. We also built a convolutional-network alternative that we deliberately did not ship, and we explain why the deterministic pipeline was preferred. The *forest* stage is a single prefit random forest with Platt (sigmoid) probability calibration, paired with an isolation forest. It maps a fixed thirty-dimensional feature vector to one of five clinically meaningful cycle labels, with a statistically honest confidence gate. We detail the pipeline stages, the classifier and its calibration, the ONNX export and the Python↔TypeScript parity contracts, and the on-device shipping pipeline. On a held-out synthetic test set the classifier reaches 88.56 % accuracy with a log loss of 0.329. The vision pipeline correctly auto-detects the temperature unit and the LH source on four real Premom screenshots spanning two languages, both units, and a mix of sparse and dense chart densities.

Contents

1 Introduction

1.1 Motivation and design constraints

Fertility-awareness methods such as the sympto-thermal method [1] infer the timing and quality of ovulation from two daily signals: basal body temperature (BBT) and, optionally, luteinizing hormone (LH) measured by ovulation test strips. The data are intimate, and mishandling them carries real harm. The European General Data Protection Regulation [2] classifies such records as a special category of personal data, which motivates an architecture in which the data are never transmitted, stored remotely, or associated with an account.

FertiLuna takes the strongest possible reading of that constraint: *the server never sees a single data point*. There is no account, no cookie that tracks the user, and no upload. The only network traffic beyond the page itself is a one-time, cacheable download of public, non-personal model files. Every act of inference (chart digitization and cycle classification alike) runs locally in the browser. This is not a privacy feature bolted onto a server-centric design; it is the organizing principle from which the rest of the system follows.

1.2 The two-stage thesis

A naïve approach would attempt a single end-to-end model from raw pixels (a screenshot) to a clinical label. We deliberately decompose the problem into two stages with distinct inductive biases:

1. A **vision stage** (Section 4) that solves a *perception* problem: turning an image of a chart into structured, per-day numerical series. We solve it with a *deterministic classical computer-vision pipeline* (color segmentation, multi-axis OCR with robust line fitting, and geometric marker detection) rather than a neural network, for the auditability, structured side-output, and zero-training-data reasons detailed in Section 4.2.
2. A **forest stage** (Section 3) that solves a *reasoning* problem: turning a small, hand-engineered, clinically interpretable feature vector into a calibrated decision. Tree ensembles excel on tabular features, are robust to missing data, are inexpensive to evaluate, and (crucially for the browser) serialize to small ONNX graphs.

The decomposition has three practical virtues. First, each stage can be developed, validated, versioned, and shipped independently. Second, the user gets an auditable intermediate representation: the digitized table can be inspected and corrected before classification. Third, the perception stage is optional. A user who types numbers directly invokes only the forest stage.

1.3 Contributions

This report contributes a complete, reproducible account of three things: (i) a deterministic, OpenCV-based chart-digitization pipeline (color segmentation, multi-axis OCR-plus-RANSAC tick reading, geometric marker detection, and calendar-table extraction), validated on real Premom screenshots, together with a reasoned comparison against a neural baseline we built but did not ship; (ii) a calibration-aware random-forest design that keeps the exported ONNX graph to a single tree ensemble while preserving a statistically meaningful confidence threshold; (iii) an on-device shipping pipeline built on ONNX Runtime Web, with a checksum-versioned IndexedDB cache and an enforced Python↔TypeScript parity contract.

2 System overview

Figure 1 summarizes the full lifecycle. Offline, in Python, a synthetic cycle generator feeds the forest training pipeline (scikit-learn), which exports ONNX artifacts accompanied by JSON manifests that record the input/output contract, normalization constants, evaluation metrics, and SHA-256 checksums. The classical vision pipeline requires no training; its only model asset is a pre-built PaddleOCR recognizer.

A build script copies all artifacts into the static-asset directory of an Astro application deployed as a Cloudflare Worker. At run time the browser fetches each model once, verifies its checksum, caches it in IndexedDB, and executes it through ONNX Runtime Web on the WebAssembly backend, while the vision geometry runs as pure client-side TypeScript.

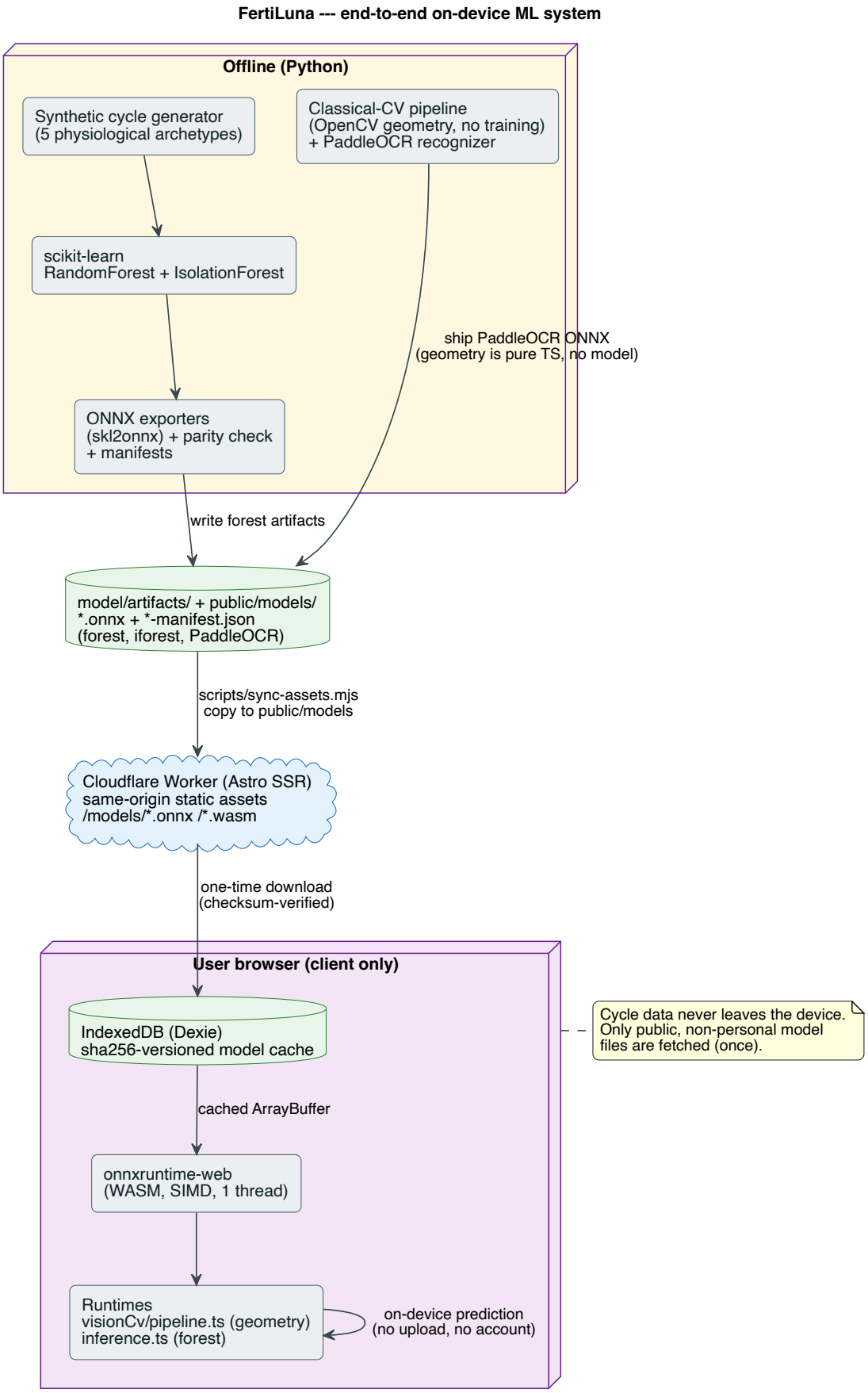

Figure 1: End-to-end system architecture. Training is offline; serving is a same-origin static-asset delivery; all inference is client-side. The dashed boundary around the browser denotes the privacy perimeter: cycle data never crosses it.

2.1 Technology stack

The application is written in TypeScript and Astro and deployed on Cloudflare Workers (server-side rendering for the page shell only). Browser inference uses `onnxruntime-web` on the WASM backend with a single thread and SIMD enabled, which avoids the cross-origin-isolation (`COOP/COEP`) headers that multi-threaded WASM would require. The model cache is built on Dexie (a typed wrapper over IndexedDB). The Python side uses NumPy, pandas, and scikit-learn for the forest family [3]; PyTorch for the vision network; and `skl2onnx`/`torch.onnx` for export [4, 5].

3 The forest model: calibrated cycle classification

3.1 Problem formulation

Let a cycle be represented by two length-D vectors ($D = 35$ days), $\boldsymbol{t} \in (\mathbb{R} \cup \{\text{NAN}\})^D$ of temperatures and $\boldsymbol{\ell}$ of LH values, where NAN marks a missing observation. A deterministic feature map ϕ produces a fixed vector $\boldsymbol{x} = \phi(\boldsymbol{t}, \boldsymbol{\ell}) \in \mathbb{R}^{30}$, and a classifier assigns one of five labels:

$$\mathcal{Y} = \{\text{ovulation\_confirmee},\ \text{ovulation\_douteuse},\ \text{anovulation},\ \text{phase\_luteale\_courte},\ \text{donnees\_insuffisantes}\}.$$

These correspond, respectively, to a confirmed ovulation, a doubtful thermal shift, an anovulatory cycle, a short luteal phase, and insufficient data.

3.2 Feature engineering

The thirty features (Table 1) are scalar, NaN-safe summaries of the two curves. The design philosophy is to hand the model the same quantities a clinician would inspect, so that the learned function refines domain knowledge rather than rediscovering it from scratch. The cornerstone is a hard-coded implementation of the SENSIPLAN 3-over-6 ovulation rule: the first day d at which three consecutive temperatures strictly exceed the maximum of the preceding six, with the third exceeding that maximum by at least 0.2 °C, marks the last follicular day.

```python
def _detect_ovulation_day_sensiplan(temps):
    n = len(temps)
    for d in range(6, n - 2):
        prior  = temps[d - 6 : d]
        rising = temps[d : d + 3]
        prior_max = float(np.nanmax(prior))
        if not np.all(rising > prior_max):
            continue
        if rising[2] - prior_max < 0.20:        # >= 0.2 C confirmation
            continue
        return d - 1                            # last follicular day (0-indexed)
    return -1
```

Listing 1: The SENSIPLAN 3-over-6 detector, the anchor feature for follicular/luteal partitioning (from `features.py`).

Table 1: The thirty-dimensional feature vector $\phi(\boldsymbol{t}, \boldsymbol{\ell})$. Indices match `constants.py`. All entries are scalars; missing days contribute NAN that is handled by NaN-safe reductions.

#	Feature	#	Feature	#	Feature
0	`n_temp_observed`	10	`follicular_mean`	20	`lh_peak_value`
1	`n_lh_observed`	11	`follicular_std`	21	`lh_peak_count`
2	`missing_rate_temp`	12	`luteal_mean`	22	`days_lh_peak_to_rise`
3	`missing_rate_lh`	13	`luteal_std`	23	`follicular_length`
4	`temp_mean_overall`	14	`thermal_rise_amplitude`	24	`luteal_length`
5	`temp_std_overall`	15	`rise_steepness_max`	25	`slope_pre_ovulation`
6	`temp_min`	16	`plateau_days_above_base`	26	`slope_post_ovulation`
7	`temp_max`	17	`n_consecutive_high_days`	27	`fraction_days_below_mean`
8	`temp_range`	18	`post_rise_dip_count`	28	`longest_run_above_mean`
9	`estimated_ovulation_day`	19	`lh_peak_day`	29	`longest_run_below_mean`

3.3 Synthetic data generation

No real labeled corpus exists at the scale required, and collecting one would contradict the privacy thesis. We therefore generate $N = 50\,000$ physiologically grounded cycles from five archetypes with prior weights $(0.45, 0.15, 0.15, 0.15, 0.10)$ for {normal, doubtful, short-luteal, anovulation, insufficient}. Each archetype samples plausible parameters (follicular and luteal lengths, thermal-rise amplitude and duration, baseline temperature, measurement noise), and the curve is then corrupted with realistic artifacts (fever bumps, late-measurement spikes) and missing data. The LH surge is placed 12 to 36 hours before ovulation, with optional spurious peaks that emulate PCOS-like patterns. Labels are assigned by the same clinical rules the model is meant to internalize, so the classifier learns a noise-robust version of an explicit decision procedure.

3.4 Model and calibration

The classifier is a `RandomForestClassifier` [6] with 120 trees, maximum depth 12, minimum leaf size 12, $\sqrt{\cdot}$ feature subsampling, and balanced class weights. Raw random-forest probabilities are poorly calibrated. They cluster near the extremes, which would make a fixed confidence threshold meaningless. We therefore apply Platt (sigmoid) calibration [7, 8].

A subtle but important engineering decision concerns *how* calibration is fitted. The default `CalibratedClassifierCV(cv=5)` trains five separate forests, which would serialize to roughly 250 MB of ONNX. That is a non-starter for a browser download. Instead we train a *single* forest on a "fit" split and calibrate it on a held-out "calibration" split using a frozen (prefit) estimator. The exported graph thus contains exactly one tree ensemble (5.6 MB) while still yielding the calibrated probabilities the confidence gate relies upon.

```
base_rf = RandomForestClassifier(n_estimators=120, max_depth=12,
                                 min_samples_leaf=12, max_features="sqrt",
                                 class_weight="balanced", random_state=seed)
base_rf.fit(X_fit, y_fit)                          # ONE forest
clf = CalibratedClassifierCV(estimator=FrozenEstimator(base_rf),
                             method="sigmoid")    # Platt scaling only
clf.fit(X_calib, y_calib)
```

Listing 2: Single prefit forest, calibrated on a held-out split (from `train.py`).

At inference time the predicted label is the arg max of the calibrated probabilities, *unless* the maximum probability falls below the gate $\tau = 0.60$, in which case the label is overridden to DONNEES_INSUFFISANTES:

$$\hat{y} = \begin{cases} \arg\max_k p_k & \text{if } \max_k p_k \geq \tau, \\ \text{donnees\_insuffisantes} & \text{otherwise.} \end{cases} \tag{1}$$

3.5 Out-of-distribution detection

A classifier trained on synthetic archetypes will happily produce confident, yet meaningless, outputs on genuinely anomalous curves. As an advisory backstop we fit an isolation forest [9] with 150 estimators and contamination 0.10 on the same features, and store the $5^{\text{th}}/50^{\text{th}}/95^{\text{th}}$ percentiles of its decision-function on the training distribution. At run time a raw anomaly score s is mapped to a 0–100 "unusualness" percentile by piecewise-linear interpolation through those anchors (lower $s \Rightarrow$ more anomalous), and surfaced in the UI rather than used to alter the label.

3.6 Training and export pipeline

Figure 2 renders the complete pipeline. After export, `skl2onnx` [5] produces graphs pinned to opset 17 (and `ai.onnx.ml` 3, the version supported by ONNX Runtime Web). A parity check asserts that the maximum absolute difference between scikit-learn and ONNX Runtime probabilities is below 10^{-3}; in practice it is $\approx 1.2 \times 10^{-7}$.

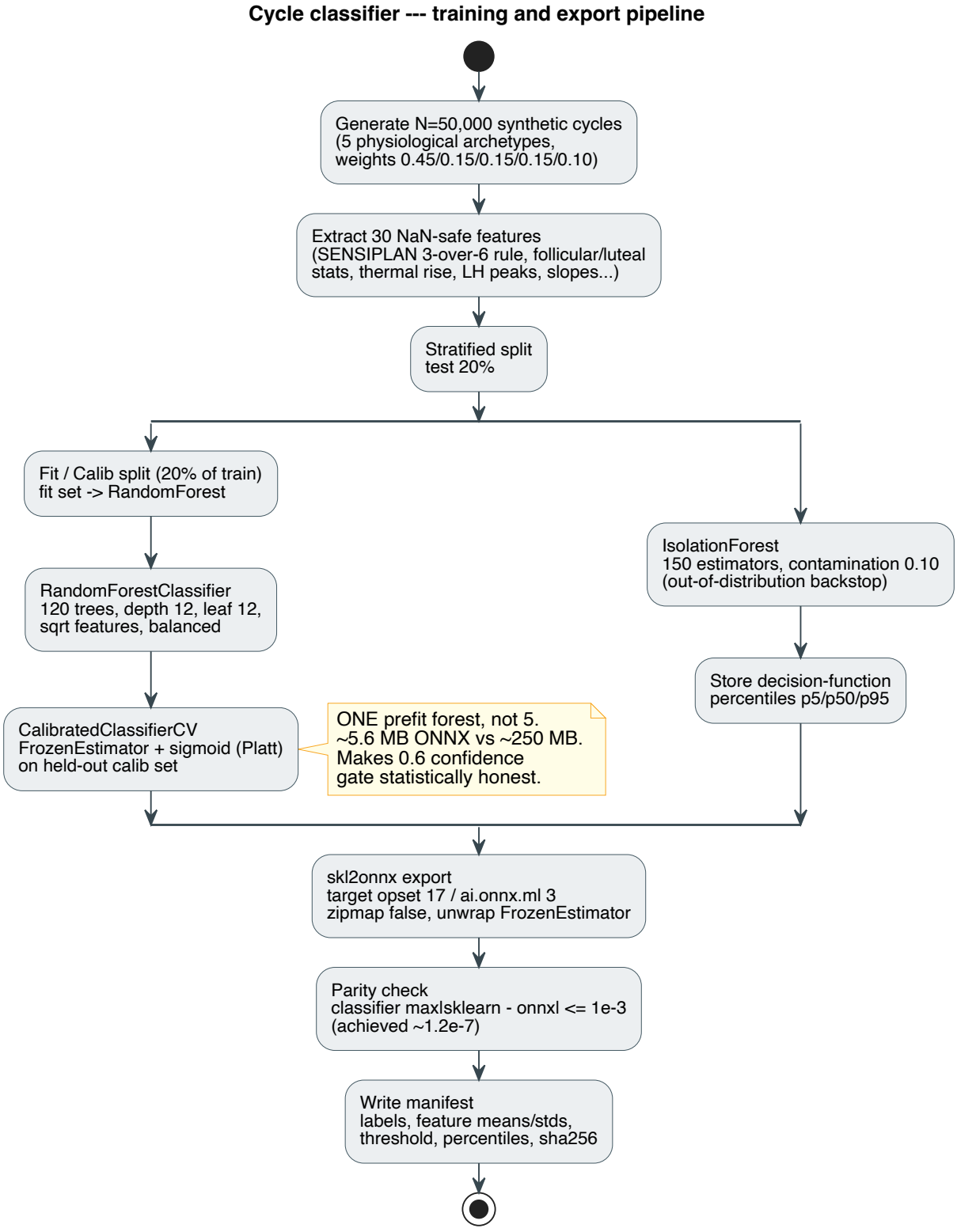

Figure 2: Forest training and export pipeline: synthetic generation, feature extraction, a single prefit forest with Platt calibration, an isolation-forest OOD backstop, and ONNX export with an enforced numerical-parity contract.

3.7 Results

On a held-out synthetic test set of 10 000 cycles the calibrated classifier reaches an accuracy of 88.56 % and a log loss of 0.329. Per-class metrics appear in Table 2; the confusion matrix is visualized in Figure 3. As expected, the main confusions live between *confirmed* and *doubtful* ovulation and between *doubtful* and *short-luteal*. These are boundaries that are genuinely fuzzy in clinical practice. The data-sufficiency and anovulation classes, on the other hand, are recovered cleanly.

Table 2: Per-class performance of the calibrated forest on the synthetic test set (n = 10 000).

Class	Precision	Recall	F_1	Support
ovulation_confirmee	0.957	0.932	0.944	4474
ovulation_douteuse	0.764	0.784	0.774	1495
anovulation	0.876	0.877	0.876	1507
phase_luteale_courte	0.784	0.828	0.805	1526
donnees_insuffisantes	0.939	0.933	0.936	998
Accuracy		**0.8856**	(log loss **0.329**)	

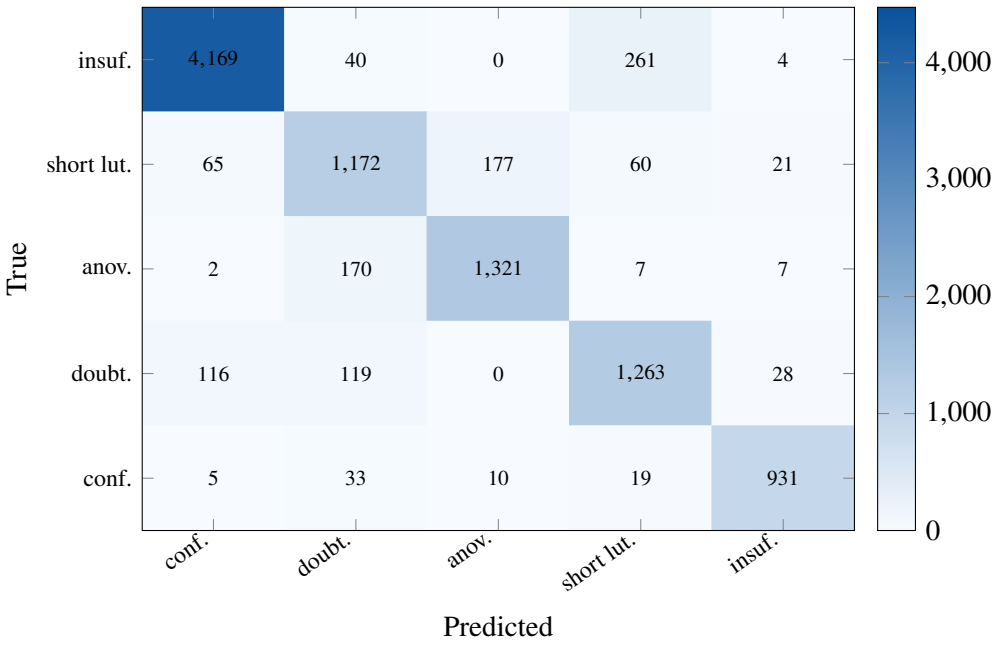

Figure 3: Confusion matrix of the calibrated forest (rows: true class top-to-bottom follows the legend order; counts annotated). The dominant off-diagonal mass lies between adjacent clinical categories.

4 The vision stage: reading charts with classical computer vision

4.1 Problem statement

Many users arrive with a screenshot from an existing fertility app rather than a table of numbers. The vision stage recovers, from a single RGB image, two per-day series (basal body temperature, BBT, and the LH or ovulation-test signal), together with the temperature unit of the BBT axis (Celsius or Fahrenheit) and an auxiliary parse of the calendar table beneath the chart. Its output is exactly the structured representation that the forest classifier of Section 3 consumes, so the two stages compose cleanly.

4.2 Why a deterministic pipeline rather than a neural network

We implemented *two* solutions to this problem: a convolutional neural network (described, for completeness, in Section 4.8) and a *classical computer-vision pipeline* built on OpenCV. **The deployed vision stage is the classical pipeline.** The neural network is kept as a research baseline but is not shipped. The decision rests on four properties that matter more than raw accuracy for a privacy-first, on-device health tool:

1. **Determinism and auditability.** Every decision in the pipeline is an explicit, inspectable rule with a named constant (an HSV band, a coverage threshold, a RANSAC tolerance). There are no learned weights to reason about, no opaque failure modes, and the same input always yields the same output. For a tool that informs reproductive decisions, an auditable extraction is preferable to a marginally more accurate black box.
2. **Structured side-output for free.** Because the pipeline reasons about the image geometrically, it also recovers the axis tick *values*, the day-cell grid, the detected markers, and the bottom calendar table (cycle-day, days-past-ovulation, intercourse, cervical-mucus, hCG). The CNN produces only the two series and the unit; the table would require a separate model.
3. **No training data, no sim-to-real gap.** The CNN must be trained on procedurally rendered charts and then bridged to real app screenshots. The classical pipeline reads the real screenshot directly, so there is no distribution shift to manage.
4. **Smaller and simpler to ship.** The only model asset the pipeline needs is a ~9 MB PaddleOCR text-recognition network used to read axis labels [10]; the geometry is pure arithmetic. The CNN would add a 16.6 MB download with no table output.

The one genuine cost is engineering effort: the pipeline is a sequence of hand-tuned stages, whereas the CNN is a single trained artifact. We judged that cost worthwhile, given the auditability and side-output benefits.

4.3 Pipeline architecture

The pipeline (Figure 4) is an ordered sequence of nine stages implemented in Python with OpenCV and NumPy, plus a PaddleOCR PP-OCRv3 recognizer exported to ONNX for reading axis tick labels. The same stages are mirrored, op for op, in a pure-TypeScript browser port (Section 4.7) so that the production tool runs entirely client-side. Each stage consumes the working image and the accumulated state and contributes to a single `ChartResult`.

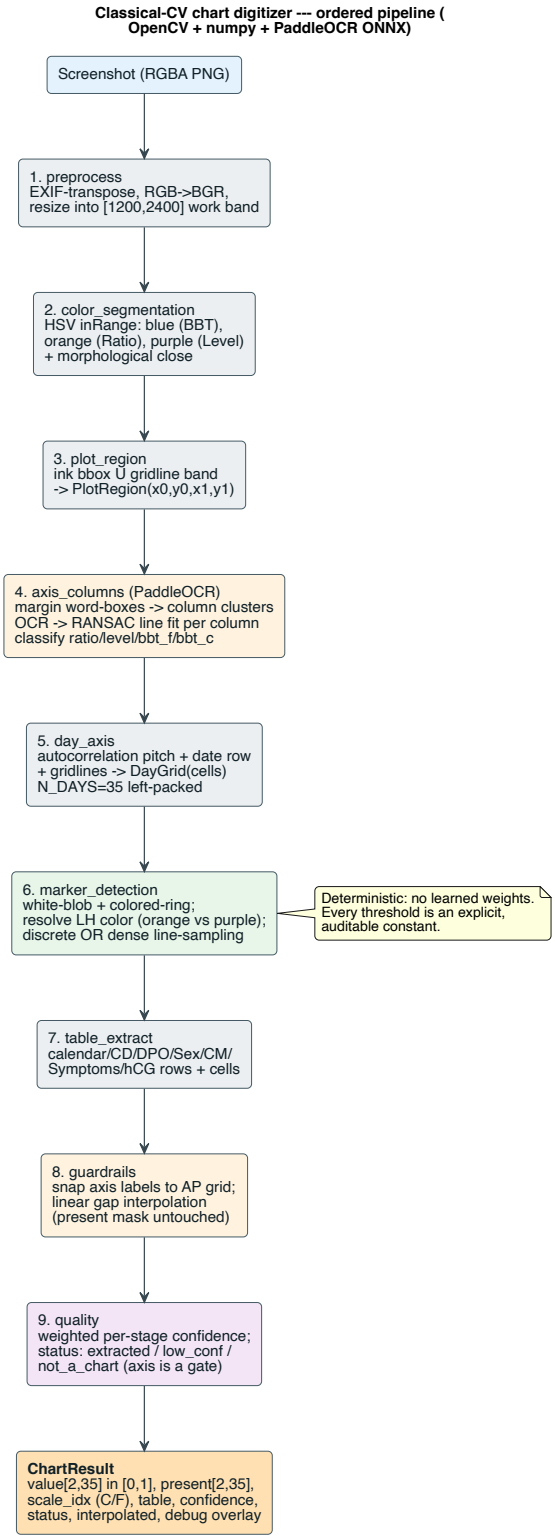

Figure 4: The classical-CV chart digitizer: nine ordered, deterministic stages from raw screenshot to `ChartResult`. The only learned component is a small PaddleOCR network that reads numeric axis labels; all geometry is explicit arithmetic.

4.4 Key stages in detail

Color segmentation (stage 2). The chart's series are distinguished by color, so the pipeline converts to HSV and applies deliberately *non-overlapping* `inRange` bands: blue for BBT, orange for the LH "Ratio"

line, and purple for the LH "Level" line. The non-overlap is a design choice: it guarantees that the purple distractor cannot leak into the blue BBT mask. A small morphological close bridges the gaps between line strokes.

Multi-axis tick reading via OCR + RANSAC (stage 4). This is the heart of unit detection and de-normalization. Premom-style charts carry up to three stacked y-axes (a Ratio axis, a Level axis, and the BBT axis). The pipeline detects text word-boxes in the left and right margins, clusters them into vertical *columns* by their x-center, OCRs each box with PaddleOCR, and fits a line value $= a + b\,y$ to each column by RANSAC. RANSAC keeps the fit robust to a stray calendar date that leaks into the margin. Each fitted column is then classified by matching its top/bottom value span against known axis profiles:

$$\text{ratio}\ (1.9{\to}0.1), \quad \text{level}\ (95{\to}5), \quad \text{bbt}_\text{F}\ (99.5{\to}95), \quad \text{bbt}_\text{C}\ (37.4{\to}35.6).$$

Crucially, the temperature unit is read *directly* from which BBT column exists, rather than guessed. If a column fits the $[95, 99.5]$ Fahrenheit profile, the scale is Fahrenheit with full confidence. The inverse of the fitted line, value_at$(y) = a + b\,y$, later maps each detected marker's pixel position to a physical value.

Day-cell grid (stage 5). The horizontal day axis is recovered, in priority order, by clustering the date-row digits, by detecting vertical gridlines, or, failing those, by taking the dominant pitch of the column-density autocorrelation and cross-checking it against the marker spacing. The grid is constrained to a plausible pitch and coverage, and tiled to $N = 35$ cells anchored at the leftmost marker. Charts that show more than 35 days are left-packed and flagged `truncated`.

Marker detection and LH-source resolution (stage 6). Data points are small white blobs ringed by a colored stroke. The detector finds white connected components inside the plot and, for each, sweeps candidate ring radii and measures the fraction of an angular sample that falls on each color mask; the best-covered color assigns the blob to a series. A dedicated step resolves whether the LH signal is carried by the orange "Ratio" line or the purple "Level" line by comparing discrete-marker counts and ink continuity. For dense, near-continuous curves the detector switches from discrete markers to per-cell line sampling, reading the ink centroid in each day column while excluding the flat BBT threshold line.

Guardrails (stage 8). Two repairs run before the result is emitted. First, OCR labels are snapped onto the inferred arithmetic tick grid (the canonical fix $[36, \texttt{"5"}, 37] \to [36, 36.5, 37]$). Second, short gaps in each series are linearly interpolated, but the `present` mask is *never* modified, so a synthesised day can never be mistaken for a measured one. This separation is a deliberate medical-safety property.

Quality gating (stage 9). A weighted confidence in $[0, 1]$ is computed from per-stage signals, and the fitted BBT axis acts as a gate: with no trustworthy axis the marker contribution is heavily discounted and the status is capped at *low confidence*; with no axis and fewer than two points the image is rejected as *not a chart*.

4.5 The `ChartResult` contract

The pipeline emits a single dataclass whose numeric core is identical to the schema the neural alternative would produce, so downstream code is agnostic to which vision stage ran (Table 3).

Table 3: The principal fields of `ChartResult`. The numeric core (`value`, `present`, `scale_idx`) matches the contract of the neural alternative; the remaining fields are side-outputs unique to the geometric pipeline.

Field	Type	Meaning
`value`	(2, 35) float32	per-series value, normalized to [0, 1] within its axis range
`present`	(2, 35) float32	measured-day mask $\in \{0, 1\}$ (never inferred)
`scale_idx`	int	0 = Celsius, 1 = Fahrenheit
`scale_label`, `scale_confidence`	str, float	detected unit and its confidence
`table`	dict	calendar / CD / DPO / Sex / CM / Symptoms / hCG rows and cells
`visible_days`, `truncated`	int, bool	estimated days shown; flag when > 35
`confidence`, `status`	float, str	overall quality and {extracted, low_confidence, not_a_chart}
`interpolated`	(2, 35) float32	1.0 on synthesised days (kept separate from `present`)
`debug`	dict	masks, plot box, grid, axis columns, markers (drives the overlay)

4.6 Results on real Premom screenshots

We evaluate the pipeline on four real, in-the-wild Premom screenshots that differ in language, temperature unit, chart density, and LH source. The pipeline is run through its command-line *assess* mode, which renders a diagnostic overlay and emits the decoded JSON:

```
python -m fertiluna_vision_cv.cli assess \
    --images ../real-screen-1.png ../real-screen-2.png \
             ../real-screen-3.png ../real-screen-4.png \
    --out /tmp/cv-assess
# -> {stem}_overlay.png + {stem}_decoded.json + index.html
```

Listing 3: Producing the diagnostic overlays from real screenshots (run from `model/`).

Table 4 summarizes the per-screenshot outcome. The unit is auto-detected correctly in every case (Celsius for the German screenshot, Fahrenheit for the others), the LH source is correctly resolved between the orange and purple lines, and the two long charts ($\approx$ 42 visible days) are correctly flagged as truncated.

Table 4: Classical-CV pipeline on four real Premom screenshots. Counts are detected (present) days out of the 35-day window; *vis.* is the estimated number of days the chart actually shows.

Screenshot	Description	Scale (conf.)	LH source	BBT	LH	Vis. / trunc.	Status (conf.)
real-screen-1	English, sparse	F (1.00)	orange “Ratio”	16/35	15/35	18 / no	extracted (0.87)
real-screen-2	German, sparse	C (0.80)	orange “Ratio”	19/35	19/35	25 / no	extracted (0.71)
real-screen-3	dense ($\approx$42 d)	F (1.00)	purple “Level”	34/35	35/35	42 / yes	extracted (0.83)
real-screen-4	dense + LH peak	F (1.00)	purple “Level”	35/35	35/35	42 / yes	extracted (0.82)

The diagnostic overlay drawn by the pipeline (`debug_overlay.py`) is the clearest way to see what each stage recovered. It draws the detected plot region (green), the axis tick boxes labeled with their OCR’d values and classified kind (ratio / level / bbt), the day-cell grid (numbered 1–35), the BBT markers (blue rings) and LH markers (orange rings), and the extracted calendar table, with a per-day decoded value strip beneath. Figures 5 and 6 show two contrasting cases.

Figure 5: Overlay for `real-screen-1` (English, sparse, Fahrenheit auto-detected). Blue rings are detected BBT points; orange rings are the LH "Ratio" line; the green box is the plot region; the numbered ticks at the bottom are the day-cell grid; the right-margin boxes are the OCR'd Fahrenheit BBT axis ($\geq 95 \ldots \geq 99.5$); the bottom rows are the recovered calendar table (Mar dates, CD, DPO, Sex hearts, CM). The two-row strip at the very bottom is the de-normalized output (°F for BBT, ratio for LH).

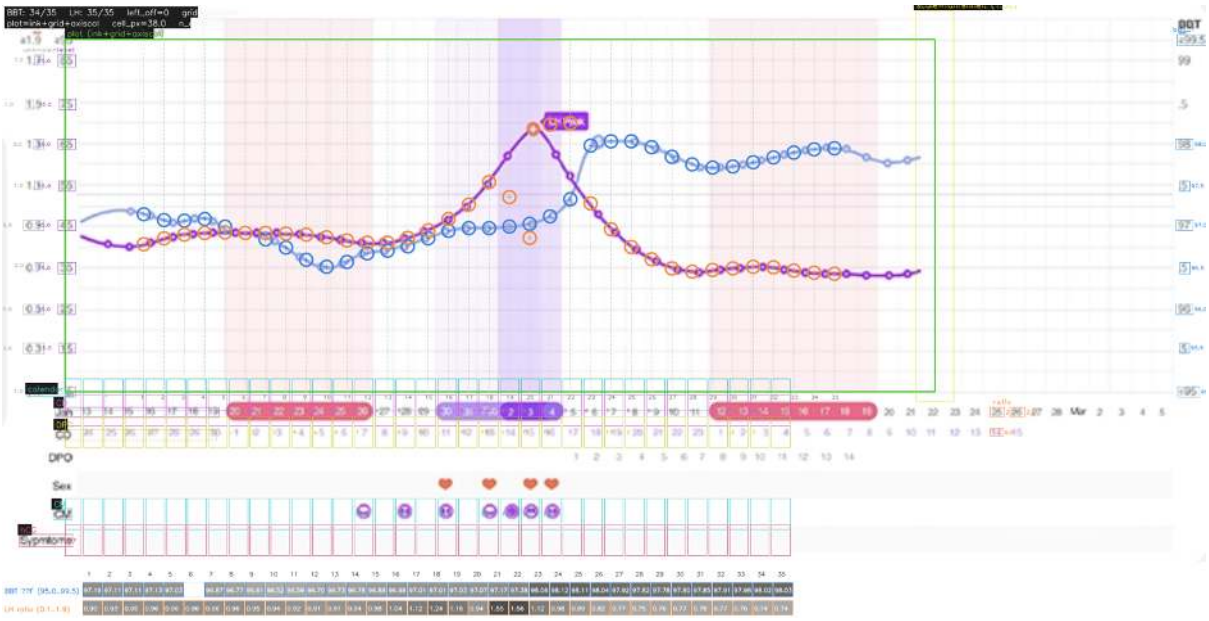

Figure 6: Overlay for `real-screen-3`, a dense $\approx$42-day chart whose LH signal is the *purple* "Level" line. Here the detector switches from discrete markers to per-cell line sampling: the densely-packed blue and orange rings track the continuous curves cell by cell, the chart is left-packed into the 35-day window and flagged `truncated`, and the purple Level line is correctly chosen as the LH source over the orange Ratio line.

4.7 The browser port

The production tool runs the pipeline entirely in the browser. The only two OpenCV primitives that are not pure arithmetic, connected-component labeling and morphological dilation, are reimplemented in pure TypeScript (an eight-connectivity flood-fill with bounding-box and centroid statistics, and a separable rectangular structuring element). As a result, **no OpenCV.js or WASM build of OpenCV is required**; the only model asset is the PaddleOCR ONNX, which is executed by ONNX Runtime Web exactly like the forest models (Section 5). The TypeScript port emits a `ChartResultCv` type that mirrors the Python `ChartResult` field for field, and is parity-tested against the Python reference.

4.8 A neural alternative we explored but did not ship

For completeness we record the convolutional network we built and evaluated before settling on the classical pipeline. It is a compact, fully convolutional encoder in the MobileNet family [11, 12]: a strided stem and six depthwise-separable blocks downsampling only to $/8$, feeding three decode heads (Figure 7). Its distinctive element is a per-series *soft-argmax over height* [13]. A 1×1 convolution produces one heatmap channel per series, a softmax over the height dimension yields a vertical probability distribution, and the expected vertical coordinate *is* the predicted value,

$$\boldsymbol{p} = \underset{h}{\mathrm{softmax}}(\mathrm{heat}/\tau)\,, \qquad \mathrm{value}_{s,w} = \sum_{i=1}^{h} p_{s,i,w}\, y_i \in [0,1], \tag{2}$$

which makes "value = vertical position" true by construction. A presence head (height-pooled) and a unit-classification head complete the network. Trained on procedurally rendered charts via a two-stage sim-to-real curriculum (generic pre-training, then a Premom-specific fine-tune) with a masked-Huber + BCE + cross-entropy loss, the v2 network (4.13M parameters, 16.6 MB ONNX) reached a present-day value MAE of 0.022 of the axis span and a presence F_1 of 0.934 on synthetic validation.

Despite these respectable numbers, we did *not* deploy it, for the reasons in Section 4.2: it offers no auditability, produces no calendar table, and carries a sim-to-real risk that the classical pipeline avoids by construction. We include its architecture here only as a documented baseline.

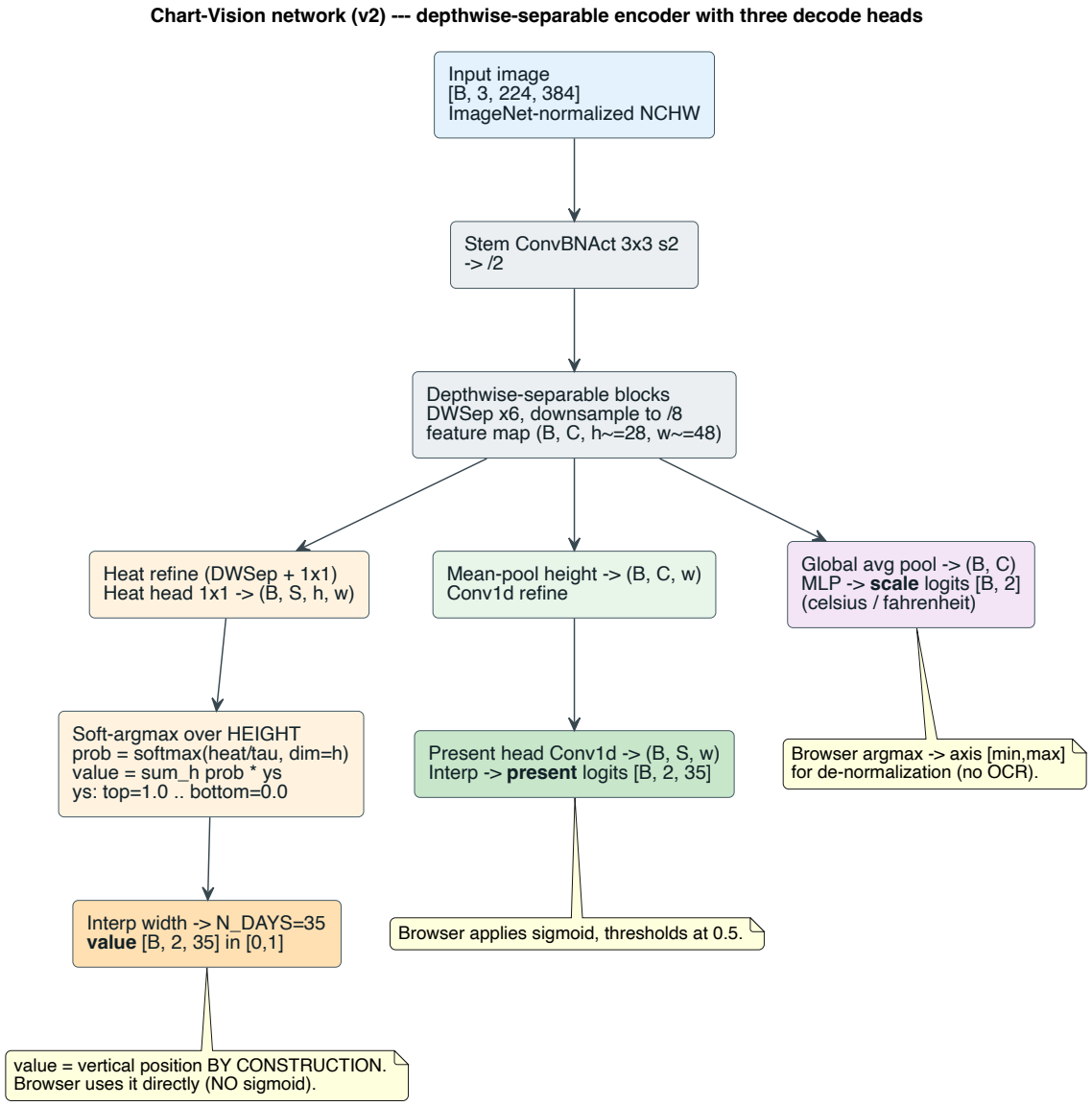

Figure 7: The *unshipped* neural baseline: a depthwise-separable encoder feeding a soft-argmax value head, a presence head, and a unit-classification head. Retained for reference; the deployed vision stage is the classical pipeline of Figure 4.

4.9 A hybrid digitization router

The classical pipeline is the default of three on-device routes for reading a screenshot (Figure 8). The deterministic CV pipeline runs first; the manual column-scan calibrator gives the user a fully transparent fallback; and the neural baseline of Section 4.8 remains available as an experimental option. All routes converge on the same per-day series structure and feed the forest classifier.

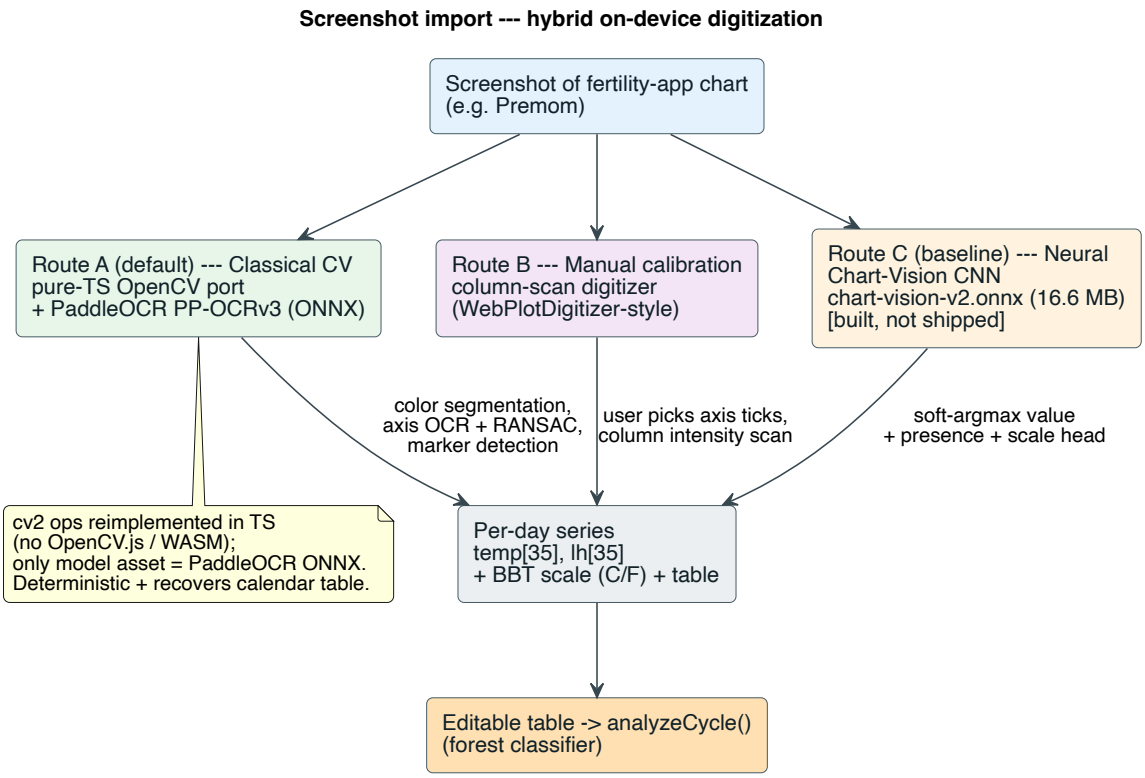

Figure 8: Hybrid on-device digitization. The deployed default is the deterministic classical-CV pipeline (+ PaddleOCR); a manual calibrator and the neural baseline are alternative routes. All produce the same structured series consumed by the forest classifier.

5 Shipping models on-device

5.1 ONNX as the lingua franca

Every model (the forest, the isolation forest, the vision network, and the OCR) is exported to ONNX [4] and executed by ONNX Runtime Web on the WebAssembly backend [14]. ONNX decouples the training framework (scikit-learn, PyTorch) from the deployment runtime and provides a single, auditable artifact whose operator set can be pinned to exactly what the browser runtime supports.

5.2 From artifact to asset

Exported `.onnx` files and their JSON manifests are written to the `model/artifacts/` directory. A build step (`scripts/sync-assets.mjs`) copies them into the application's `public/models/` directory, from which the Cloudflare Worker serves them as same-origin static assets. The WASM runtime binary is emitted by the bundler with a content hash and served from the same origin, so no cross-origin CDN is involved and no special isolation headers are required.

5.3 Versioned, checksum-validated caching

The heavy ONNX files would be wasteful to re-download on every visit, so they are cached in IndexedDB through Dexie. The cache is content-addressed and versioned: each blob is keyed by `$version:$fileName` and validated against the SHA-256 recorded in the manifest. On a version bump or a checksum mismatch the stale entry is evicted and the file refetched; older versions of a file are purged to bound storage. After the first successful load the application is fully offline-capable.

```ts
const cached = await db().modelFiles.get(key);   // key = `${version}:${fileName}`
if (cached && cached.sha256 === sha256) return cached.data;   // hit
if (cached) await db().modelFiles.delete(key);                // stale -> evict
const data = await fetchWithProgress(url, fileName, expectedBytes, onProgress);
if (await sha256Hex(data) === sha256) {           // integrity gate
  await db().modelFiles.where("fileName").equals(fileName)
            .and(r => r.version !== version).delete();          // purge old versions
  await db().modelFiles.put({ key, version, fileName, sha256, data, ... });
}
```

Listing 4: Checksum-keyed cache lookup and eviction (from `modelCache.ts`).

5.4 Inference lifecycle

Figure 9 traces the full sequence for both a cold start (cache miss) and a warm start (cache hit). On a cold start the runtime fetches the manifest, streams the model with progress reporting, verifies the checksum, persists it, and constructs an inference session. On a warm start it loads directly from IndexedDB with no network access. Inference itself (feature extraction, the classifier forward pass, the confidence gate of Equation (1), and the OOD percentile) runs entirely in WASM.

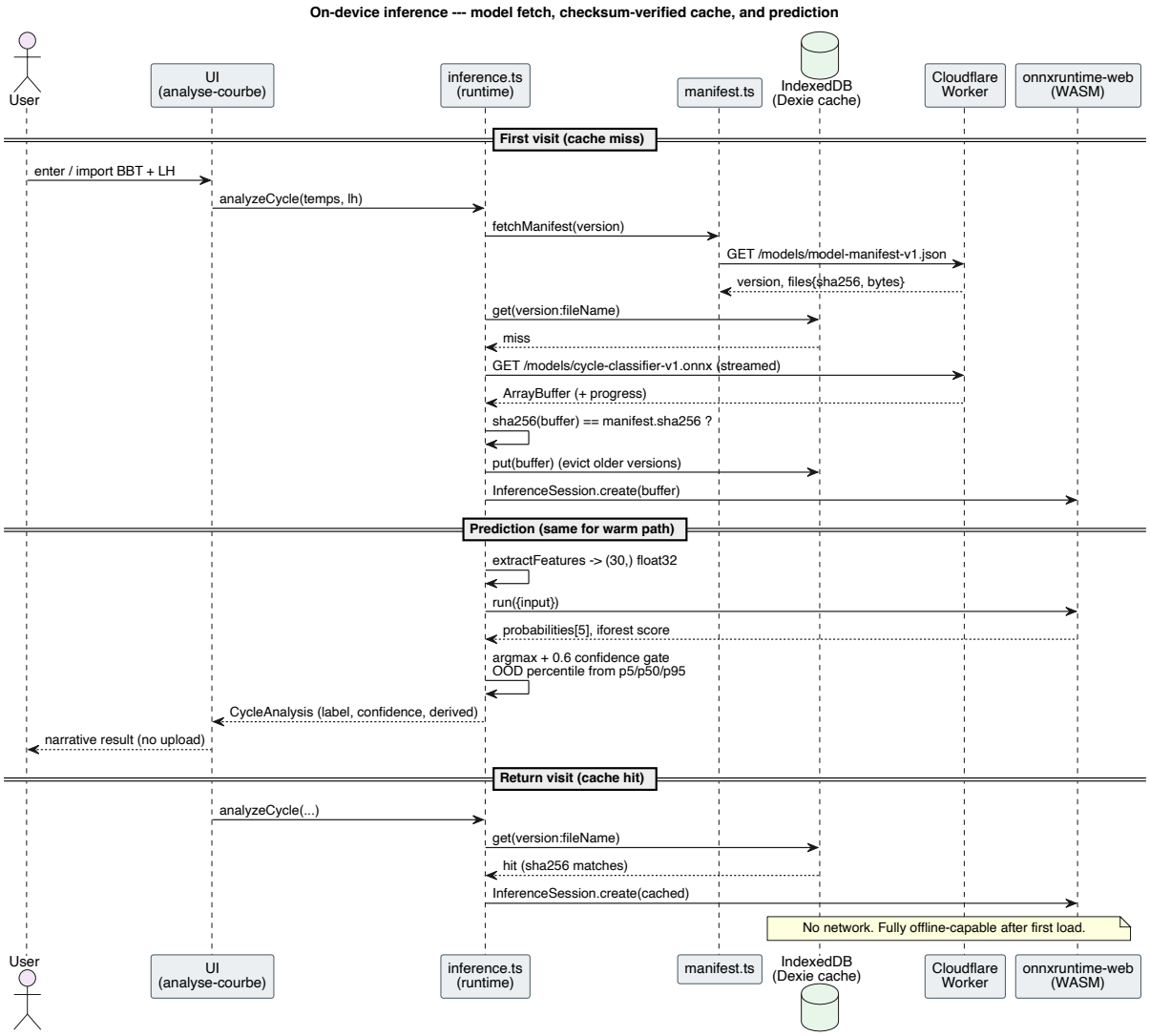

Figure 9: On-device inference lifecycle. The cold path (cache miss) verifies a checksum and persists the model; the warm path is network-free. No cycle datum is ever transmitted.

5.5 The parity contract

Feature extraction is implemented twice: canonically in Python (`features.py`) for training, and in TypeScript (`features.ts`) for the browser. Any divergence would silently degrade accuracy, because the model was trained on the Python features but evaluated on the TypeScript ones. We therefore treat numerical parity as a hard contract enforced by a fixture-based test: the Python implementation emits reference vectors that the TypeScript implementation must reproduce exactly.

Property 1 (Cross-language feature parity)**.** *For every input pair* $(\boldsymbol{t}, \boldsymbol{\ell})$ *in the fixture set,* $\phi_{py}(\boldsymbol{t}, \boldsymbol{\ell}) = \phi_{ts}(\boldsymbol{t}, \boldsymbol{\ell})$ *within floating-point tolerance. The export likewise guarantees* $\left\| \Pr_{sklearn} - \Pr_{onnx} \right\|_{\infty} < 10^{-3}$ *for the classifier. The vision pipeline likewise carries a Python↔TypeScript parity contract: the browser port reproduces the reference* `ChartResult` *of the Python implementation field for field, verified by fixtures.*

6 Privacy and threat model

The system's privacy posture follows directly from its architecture rather than from policy promises. There is no account and no server-side data store; the only egress is the one-time download of public model files. Consequently:

- **Confidentiality at rest and in transit.** Cycle data exist only in the page's memory and (optionally) the user's own local storage; they are never serialized to a network request.
- **No linkage.** Without an account or tracking identifier, analyses cannot be associated with an individual across sessions or devices.
- **Integrity of models.** The SHA-256 gate ensures that a tampered or corrupted model file is rejected rather than executed, providing a basic supply-chain integrity check for the cached artifacts.
- **Regulatory alignment.** By never processing personal data on the server, the design sidesteps the bulk of the obligations that GDPR [2] attaches to special-category data.

The residual trust assumption is the integrity of the delivered application code and WASM runtime, which is identical to that of any web page, and which subresource integrity and the same-origin policy mitigate.

7 Discussion and limitations

Determinism over learned perception. The deployed vision stage is a deterministic pipeline of hand-tuned stages rather than a learned model. Its main risk is therefore brittleness to chart idioms it was not tuned for: a new app's color palette, axis layout, or marker style may require new HSV bands or axis profiles. We accept this in exchange for auditability, a structured calendar-table side-output, and the absence of any sim-to-real gap. The neural baseline of Section 4.8, trained with domain randomization and a Premom-specific fine-tune, remains available should a learned route ever be preferred. Broadening either approach to more app idioms is the natural next step.

Synthetic-only training of the forest. The forest is trained without real labeled data. Its labels are by construction consistent with the clinical rules, so the principal risk is distribution shift between synthetic archetypes and real cycles. The isolation-forest backstop mitigates this risk, but does not eliminate it.

Confusion between adjacent classes. The largest classification errors fall between clinically adjacent categories (confirmed vs. doubtful ovulation), which reflects genuine ambiguity rather than model failure. The calibrated probabilities and the confidence gate of Equation (1) convert this ambiguity into an explicit, user-visible uncertainty rather than a falsely confident label.

Client compute budget. A single-thread WASM runtime keeps deployment simple but caps throughput. The geometric vision pipeline and the modest forest are both chosen to stay within an interactive latency budget on commodity devices; multi-threaded WASM or WebGPU execution providers are a forward-looking optimization.

8 Conclusion

FertiLuna shows that a non-trivial machine-learning product, pairing a deterministic computer-vision perception stage with a calibrated tabular reasoning model, can be delivered *entirely* on-device in the web browser without compromising analytical quality. The two-stage decomposition aligns each task with an appropriate inductive bias: a transparent, auditable geometric pipeline reads the chart (and recovers its calendar table as a side-output), while a synthetic-trained, probability-calibrated forest renders the clinical judgment. Choosing classical computer vision over a learned model for perception trades a little engineering effort for determinism, a richer structured output, and the elimination of any sim-to-real gap. We argue this is the right trade for a privacy-first health tool. The ONNX-plus-IndexedDB shipping pipeline turns the remaining model assets into versioned, integrity-checked, offline-capable assets, and the enforced cross-language parity contract closes the loop between the Python reference and the browser runtime. Together, these choices yield a system whose privacy guarantees are structural, not merely contractual.

9 References

[1] Petra Frank-Herrmann, J. Heil, C. Gnoth, E. Toledo, et al. “Real-Time Fertility Awareness: The Sympto-Thermal Method”. In: *Human Reproduction* 22.5 (2007), pp. 1310–1319.

[2] European Parliament and Council. *Regulation (EU) 2016/679 (General Data Protection Regulation)*. Official Journal of the European Union L119. 2016.

[3] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[4] ONNX Community. *ONNX: Open Neural Network Exchange*. https://onnx.ai. 2017.

[5] ONNX Community. *sklearn-onnx: Convert scikit-learn models to ONNX*. https://onnx.ai/sklearn-onnx/. 2019.

[6] Leo Breiman. “Random Forests”. In: *Machine Learning* 45.1 (2001), pp. 5–32.

[7] John Platt. “Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods”. In: *Advances in Large Margin Classifiers*. MIT Press, 1999, pp. 61–74.

[8] Alexandru Niculescu-Mizil and Rich Caruana. “Predicting Good Probabilities with Supervised Learning”. In: *Proceedings of the 22nd International Conference on Machine Learning (ICML)*. 2005, pp. 625–632.

[9] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. “Isolation Forest”. In: *2008 Eighth IEEE International Conference on Data Mining*. IEEE. 2008, pp. 413–422.

[10] Yuning Du, Chenxia Li, Ruoyu Guo, Xiaoting Yin, Weiwei Liu, et al. “PP-OCR: A Practical Ultra Lightweight OCR System”. In: *arXiv preprint arXiv:2009.09941*. 2020.

[11] Andrew G. Howard et al. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”. In: *arXiv preprint arXiv:1704.04861*. 2017.

[12] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. “MobileNetV2: Inverted Residuals and Linear Bottlenecks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018, pp. 4510–4520.

[13] Aiden Nibali, Zhen He, Stuart Morgan, and Luke Prendergast. “Numerical Coordinate Regression with Convolutional Neural Networks”. In: *arXiv preprint arXiv:1801.07372* (2018).

[14] Microsoft. *ONNX Runtime Web*. https://onnxruntime.ai/docs/tutorials/web/. 2021.