أكاديمية كاوست
KAUST ACADEMY

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Ovidio E-nose: TinyML for Scent Detection

Marco Zennaro, Erika Rosas, Erika Ronchin,
Yasser Albogami, Ajlan AlAjlan

mzennaro@ictp.it, esrosoli@upvnet.upv.es, eronchin@ictp.it,
yasser.albogami@kaust.edu.sa, ajlan.alajlan@kaust.edu.sa

Abstract

Truffles, specialty coffee, and premium spices are foods of high value with a complex identity that is easy to imitate. This combination makes them frequent targets of fraud: a lower value item is sold as a premium one, or an aged sample is sold as fresh. Verifying such a claim today requires either expert judgment or a laboratory technique such as gas chromatography, and neither option is available at a point of sale in the field. We introduce *Ovidio*, a portable electronic nose that reads a food item's scent and reports what it detects, without a laboratory. Ovidio combines an eleven channel gas sensor array and a BME680 environmental sensor, read by an Arduino UNO Q, into a complete scent classification pipeline that runs entirely on the device, with no dependence on a cloud connection. The system comprises *OvidioBench*, a public benchmark dataset of more than eighty two recording sessions; the **Ovidio Classic Model** (Legacy Model), a production classifier selected through a comparison of seven modeling methodologies and reaching a macro F1 score of 0.968 on a four class problem; a browser based workflow that lets a user retrain the classifier from newly recorded sessions without writing code; a dashboard that performs deterministic quality control; and a scent to speech stage that converts a classification into a spoken explanation grounded in the underlying sensor readings, produced by a language model and a speech synthesizer that both run on the device.

Date: 22 June 2026 – 22 August 2026
Code: https://github.com/YasserAlbogami/ovidio-enose

Contents

1 Introduction 2

2 Related Work 3

3 Hardware 3

3.1 Platform Evolution . . . 3
3.2 Prototype Evolution . . . 4
3.3 Hardware Architecture . . . 4
3.4 Firmware and RPC Surface . . . 5
3.5 Sensor Array Chemistry . . . 5
3.6 Reduced-Volume Sensing Chamber . . . 5
3.7 WIO Terminal Real-Time Display . . . 7

4 Data: OvidioBench 7

4.1 Data Collection Protocol . . . 7
4.2 IAEA Check Sessions . . . 8
4.3 Article Dataset . . . 8

5 Model Selection and Methodology 8

5.1 Model Evolution . . . 8
5.2 Methodology Comparison (Article Dataset) . . . 10
5.3 The Deployed Model: Ovidio Classic Model (Legacy Model) . . . 12

6 System Implementation 13

6.1 Interface Evolution . . . 13
6.2 Get Data . . . 13
6.3 Dataset . . . 13
6.4 Quality Control . . . 13
6.5 Live Scent Detection . . . 14
6.6 Status . . . 14

7 Train Model Page 14

8 Scent-to-Speech Pipeline 15

8.1 Pipeline Stages . . . 16
8.2 Language Model Selection . . . 16
8.3 TTS Model Selection . . . 16
8.4 End-to-End Resource and Latency Measurement . . . 18

9 Discussion 18

10 Limitations and Future Work 19

11 Conclusion 20

References 21

1 Introduction

This is the comprehensive final report for *Ovidio E-nose: TinyML for Scent Detection* (working title through the earlier reports: *Truffle-Hound AI: TinyML for Scent Detection*), covering the full internship period, 22 June 2026 – 22 August 2026. Three progress reports were produced over that period, each documenting the project as it stood at the time: a halfway report establishing the sensor array, the pivot from an ESP32/TinyML plan to a classical machine learning pipeline, and a first 3-class ambient/truffle/coffee_ground model; a second progress report documenting a reduced-volume chamber prototype, a data-volume-impact study, a text-to-speech engine benchmark, and a WIO Terminal display; and a final report documenting the settled system: OvidioBench, the systematically selected Ovidio Classic Model, the Train wizard, and the end-to-end Sniff & Speak pipeline. Rather than repeating only the settled end state, this report is written to show the full arc: what was built, tested, and learned at each stage, and how each stage's open questions were carried forward and resolved (or not) by the next.

Report	Period	Headline deliverables
Halfway report	Weeks 2–6	Sensor array + Arduino UNO Q pivot (ESP32/TinyML → classical ML on-device); binary and 4-class validation tasks; first 3-class ambient/truffle/coffee_ground model (macro-F1 = 0.936)
Second progress report	20–25 Jul 2026	Reduced-volume sensing chamber (150 mm → 70 mm height, 53.3 % volume cut); data-volume-impact study confirming no training-data cost; 4-engine TTS benchmark under a 2 GB budget; WIO Terminal real-time display
Final report	18 Jun – 25 Aug 2026	OvidioBench (82 + sessions); 7-methodology model-selection comparison; deployed Ovidio Classic Model (macro-F1 = 0.968); browser-based Train wizard; Quality Control dashboard; end-to-end Sniff & Speak pipeline (SLM + TTS)

Table 1 The three progress reports produced over the internship and their headline deliverables. This comprehensive report draws on all three.

The remainder of this report is organized as follows: Related Work (Section 2) situates the project against prior electronic-nose literature; Hardware (Section 3) traces the platform from its original ESP32 concept through the reduced-volume chamber prototype to the settled current design, and covers the WIO Terminal display; Data (Section 4) introduces OvidioBench; Model Selection and Methodology (Section 5) walks through every classification task attempted over the internship, illustrated with each task's own results, before covering the systematic comparison behind the deployed Ovidio Classic Model; System Implementation (Section 6) covers the deployed web application, including how its interface grew since the halfway report; the Train Model Page (Section 7) and the Scent-to-Speech Pipeline (Section 8) each get a dedicated section, the latter tracing the pipeline from a benchmarked TTS stage with no language-model component to a fully implemented sense-classify-explain-speak cycle; and Discussion, Limitations and Future Work, and Conclusion close the report.

2 Related Work

Electronic-nose research has repeatedly shown that MOS gas sensor arrays, paired with a pattern-recognition stage, can distinguish complex odor mixtures without the cost or immobility of laboratory instrumentation such as gas chromatography [1]. Sensor datasheets for the class of components this project uses, including the BME680/BME688 environmental sensor and comparable mixed-gas sensor packages, document the underlying chemiresistive sensing mechanism this array relies on [2, 3].

Two prior studies frame where this project sits methodologically. Sun et al. combine an 8-sensor Figaro TGS-series array with a CNN-LSTM classifier to identify spirits (vodka, tequila, gin, brandy, whiskey) at 98 % accuracy [4]; the pipeline's endpoint is a classification label, which is also where most published MOS-sensor electronic-nose work stops. Ovidio's scent-to-speech stage (Section 8) extends past that endpoint: the classifier's label and confidence become the input to a spoken, sensor-grounded justification, not the final output. Xiao et al. co-design a lightweight neural network (1,444 parameters) with a low-power FPGA accelerator for a 15-sensor MOS array classifying 10 industrial gases at 99.71 % accuracy and 1.7 W [5]; this result is the closest published precedent for the edge-deployment tradeoff this project makes repeatedly, trading model capacity against a hard memory and power ceiling, whether that ceiling applies to the classifier itself or, as in Section 8, to a locally hosted language model and speech synthesizer sharing the same 2 GB board. The general edge machine learning engineering workflow this project follows, moving inference from a cloud service to the device itself, is documented in Janapa Reddi's treatment of embedded machine learning systems [6].

3 Hardware

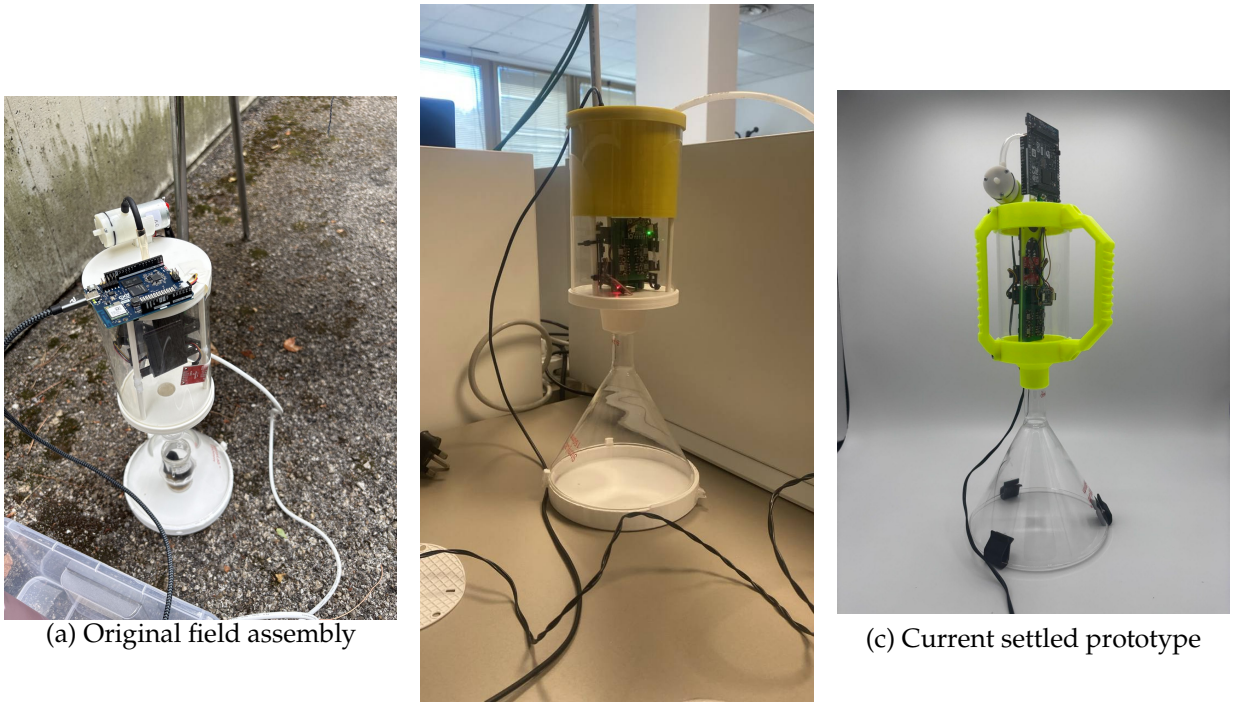
3.1 Platform Evolution

The project's reference hardware platform was originally BRIAN V1, an open-source electronic nose design built around a SparkFun MicroMod ESP32 (240 MHz dual-core Xtensa LX6, 520 KB SRAM), the same 11 DFRobot Fermion MEMS gas sensors and BME680 environmental sensor used today, and a planned software pipeline using Edge Impulse for cloud-assisted feature extraction and a TensorFlow-Lite-Micro model quantized back onto the ESP32, gated by a binary decision rule ($P(\text{truffle}) > 0.85$). Sensor gain on this design was fixed per channel, split between two settings (`GAIN_ONE`/`GAIN_SIXTEEN`).

Partway through the internship, the project pivoted to the **Arduino UNO Q** board used for every result in this report. The UNO Q is a dual-core device: a Zephyr-RTOS microcontroller side runs the sensor-reading sketch, and an embedded Linux side runs a Flask application that receives sensor data over Arduino's RouterBridge RPC channel and performs inference locally, in Python, using scikit-learn models rather than a TFLite Micro runtime. This traded the ESP32's tighter memory budget for the Linux side's ability to run classical machine learning models directly and to support in-app retraining on newly collected sessions, a better fit for a small, evolving dataset that could be iterated on far faster than a quantize-and-flash cycle. Sensor gain also became fully configurable per channel at runtime (default `GAIN_TWOTHIRDS`), persisted in `sensor_gains.json` on the device, replacing the original fixed two-setting scheme.

3.2 Prototype Evolution

The physical hardware went through three recognizable stages over the internship, shown in Figure 1: the original field-assembled sensor array on its first enclosure, documented in the halfway report; the first reduced-volume sensing chamber prototype (Section 3.6), introduced in the second progress report; and the current, settled prototype used for every result in this final report.



(a) Original field assembly

(b) First reduced-volume chamber

(c) Current settled prototype

Figure 1 Prototype evolution over the internship: (a) the assembled 11-sensor array and BME680 on its original enclosure (halfway report); (b) the first 70 mm reduced-volume chamber prototype (second progress report); (c) the current settled prototype, with the Arduino UNO Q and sample pump mounted above the chamber (final report).

3.3 Hardware Architecture

Layer	Component
Gas sensing	11× DFRobot Fermion MEMS sensors (HCHO, CH_4, VOC, Odor, EtOH, H_2S, NO_2, NH_3, CO, Smoke, H_2)
ADC	3× Adafruit ADS1015 (12-bit, I^2C addresses 0x48/0x49/0x4A)
Environmental	BME680 (temperature, pressure, humidity, gas resistance)
MCU / Linux	Arduino UNO Q: Zephyr-RTOS MCU side (`sketch.ino`) plus embedded Linux side (Flask app), connected by RouterBridge RPC
Storage	Session CSVs with a metadata sidecar, written by the Linux-side application; no SD card or BLE link

Table 2 Current hardware architecture.

Every gas channel is read through a programmable-gain amplifier. Because raw ADC counts depend on gain, they are not directly comparable across sessions recorded at different gain

settings; the pipeline instead converts every reading to gain-corrected `voltage_v` at ingestion time, which stays invariant to whichever gain a given sensor happened to use during a session. This conversion, not the raw counts, is the feature used throughout data collection and training (Section 9).

3.4 Firmware and RPC Surface

The Zephyr-side sketch owns everything below the RouterBridge boundary: I^2C communication with the three ADS1015 boards and the BME680, per-sensor gain state, and a status LED reflecting the current session phase. It exposes six RPC functions that the Linux side calls:

RPC function	Purpose
`read_sample`	Acquire one reading from all 11 gas channels + BME680, return as JSON
`get_gain_config`	Report the current per-sensor ADC gain configuration
`set_sensor_gain`	Set one sensor's gain (e.g. `"voc,2"`)
`set_all_sensor_gains`	Set every sensor to the same gain
`set_led_state`	Set the status LED on or off directly
`set_led_mode`	Drive the status LED to reflect the current app-side phase

Table 3 RPC surface exposed by the MCU-side sketch to the Linux-side Flask app.

3.5 Sensor Array Chemistry

All 11 gas channels are DFRobot Fermion metal-oxide-semiconductor (MOS) sensors: a heated SnO_2 thick-film element chemisorbs oxygen ions in clean air, raising its resistance; reducing gases lower that resistance by reacting with the chemisorbed oxygen, and this resistance change is the analog signal each ADC channel digitizes. Most channels are tuned toward one dominant target gas with narrow cross-sensitivity; three (VOC, Odor, Smoke) are deliberately broad-spectrum, responding across many overlapping compound classes at once, and are expected to respond most strongly, and least selectively, to a complex food scent such as truffle or coffee.

Their value in the array is not selective identification of one compound but a differently weighted response profile across overlapping classes; the classifier learns to recognize a combination of these non-selective responses as a scent fingerprint, rather than reading any single channel in isolation.

3.6 Reduced-Volume Sensing Chamber

The sensing chamber design introduced as a prototype in the second progress report is now the settled design used for all data collection described in this report. The chamber is treated as a right circular cylinder ($V = \pi r^2 h$); only the inner height h was changed, from 150 mm to 70 mm, at a constant 95 mm inner diameter, cutting internal volume from ≈1,063 cm^3 to ≈496 cm^3, a reduction of 53.3 %. Because the diameter did not change, volume scales linearly with height, so the percentage reduction equals the height reduction (80 mm of 150 mm).

A follow-up study reported in the second progress report confirmed that this reduction shortens real-world cleaning time for truffle samples (124–182 s, versus a 480 s cap consistently reached at the old volume) without reducing the volume of data available for model training, since the wide-format and windowed row counts a classifier trains on are set by fixed baseline/sample phase durations, not by physical chamber volume. A companion heuristic measurement put the sample/purge pump's output at 30.76 ± 1.25 cm^3/s (10 timed bag-fill runs); this figure was not

Sensor	Channel	Compound family / class	Selectivity
SEN0563	HCHO	Aldehydes (calibrated to formaldehyde, CH_2O)	Narrow
SEN0565	CH_4	Light alkanes / combustible hydrocarbons (calibrated to methane)	Narrow
SEN0566	**VOC**	Broad reducing organics: alcohols, aldehydes, ketones, aromatics, esters, terpenes	**Broad-spectrum**
SEN0571	**Odor**	Amines, H_2S, mercaptans/thiols, general VOCs (malodor-tuned)	**Broad-spectrum**
SEN0569	EtOH	Alcohols (calibrated to ethanol, C_2H_5OH)	Narrow
SEN0568	H_2S	Sulfur compounds (H_2S, cross to mercaptans)	Narrow-to-moderate
SEN0574	NO_2	Oxidizing nitrogen oxides (resistance *increases* here, opposite of the reducing-gas sensors)	Narrow
SEN0567	NH_3	Amines / ammonia	Narrow
SEN0564	CO	Combustion product (carbon monoxide)	Narrow
SEN0570	**Smoke**	Combustion/pyrolysis gases: CO, light hydrocarbons, H_2	**Broad-spectrum**
SEN0572	H_2	Small reducing gas (hydrogen)	Narrow

Table 4 Chemical family and selectivity of the 11 gas-sensing channels.

Table 5 Sensing-chamber dimensions and internal volume, old vs. current design.

Parameter	Old design	Current design
Inner diameter (mm)	95	95
Inner height, h (mm)	150	70
Inner radius, r (mm)	47.5	47.5
Cross-sectional area, πr^2 (cm^2)	70.88	70.88
Internal volume (cm^3 = mL)	1,063.2	496.2
Internal volume (L)	1.063	0.496

revisited during the reporting period covered here and carries the same caveats noted previously (uncorrected for rising back-pressure, and only 5 runs per open/closed condition).

3.7 WIO Terminal Real-Time Display

A Seeed Studio WIO Terminal was integrated during the second progress report's reporting period as an on-device, real-time display for sensor readings (Figure 2), giving a user a screen to read live sensor state from directly on the hardware rather than only through the Flask web interface documented in Section 6. As in the second progress report, this integration is described here only at the level supportable by the available photo; no further technical writeup of its data source, refresh rate, or connection to the RouterBridge/Flask pipeline was produced during the remainder of the internship (Section 10).



Figure 2 The WIO Terminal mounted on the device enclosure, used as a real-time display for sensor readings.

4 Data: OvidioBench

Over the course of the internship, data collection at ICTP's Fermi Building, Marconi Lab produced 82 + recording sessions spanning ambient air, truffle, coffee, black pepper, and coffee-variety samples, each session logging the 11-channel gas array alongside BME680 environmental readings at approximately 1 Hz. The dataset grew steadily over the internship: the halfway report's snapshot recorded 21 sessions, 41 samples, and 402,875 sensor rows across 10 labels, roughly a quarter of the final 82 +-session collection. This report names and formalizes that accumulated collection as **OvidioBench**: a public benchmark dataset for MOS-sensor scent classification, to be released on **KaggleHub** to coincide with the project's submission. The sessions recorded directly on the Arduino UNO Q device form the bulk of OvidioBench; two subsets within it were collected and vetted under a stricter, article-bound protocol and are described separately below.

4.1 Data Collection Protocol

Every session in OvidioBench follows the same fixed structure, bracketed by sensor warm-up and recovery periods: new sensors receive a 24–48 hour clean-air burn-in before first use; each session begins with a 30–60 minute warm-up followed by a 2–5 minute clean-air baseline recording; and 2–5 minutes of clean-air recovery are enforced between individual sniffs. This protocol is unchanged from the halfway report and applies uniformly across every class recorded.

4.2 IAEA Check Sessions

Two sessions, collected 5 August 2026, were recorded specifically to test the sensor array against a controlled reference: 3 repetitions of Arabica coffee and 3 repetitions of Robusta coffee, with ambient readings pooled from each session's own baseline phase. These sessions underlie the project's coffee-variety classifier (ambient/Arabica_coffee/Robusta_coffee).

4.3 Article Dataset

Twenty-five sessions (72 repetitions total, 18 repetitions per class) were collected 27–30 July 2026 under a single consistent indoor protocol (180 s baseline, 300 s sample, 300 s cleaning per repetition) across four classes: truffle, coffee, black pepper, and ambient. Each session was individually verified against its own metadata (phase history repetition counts, closed state, per-sample labels) before inclusion, per the project's authenticity requirement for article-bound data. This dataset is the training data behind the current production classifier, the Ovidio Classic Model (Legacy Model) (Section 5).

5 Model Selection and Methodology

5.1 Model Evolution

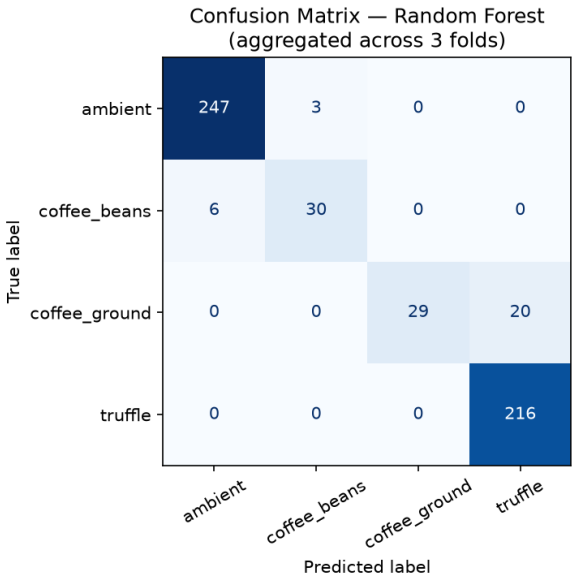
The classification task was not fixed at the project's outset; it evolved as more scent classes were collected and as confusions between classes were discovered during analysis. Table 6 summarizes the full progression, from the first end-to-end validation task through to the methodology comparison described in Section 5.2, and Figure 3 illustrates the four earliest tasks with their own confusion matrices, rather than numbers alone.

Task	Classes	Best model	Macro-F1
Binary	ambient / truffle	Random Forest	1.000
4-class	ambient / truffle / coffee_beans / coffee_ground	Random Forest	0.898
3-class	ambient / truffle / coffee_ground	Random Forest	0.936
Truffle-types	ambient / truffle / fresh_truffle	Random Forest	0.999*
Coffee-type	ambient / Arabica_coffee / Robusta_coffee	Random Forest	0.880
Article 4-class	ambient / truffle / coffee / black_pepper	L1 Logistic Regression	0.968

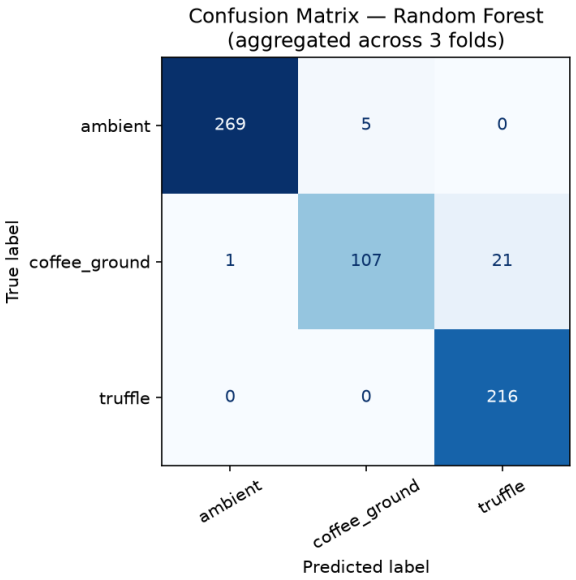
Table 6 Model evolution across the internship. *The truffle-types score is not taken at face value: with only a handful of sessions per class, `StratifiedGroupKFold` still leaves each fold's held-out data drawn from the same small set of recording days and gain configurations as its training data, so a near-perfect score is as likely to reflect session-level artifacts as genuine scent signal (Section 9).

The binary and early multi-class tasks validated the pipeline end to end and progressively added confusable classes (coffee as a distractor scent, then truffle freshness); the article 4-class task is the newest and most rigorously validated stage, since it is the first trained on a dataset large enough (25 sessions, 72 repetitions, 18 per class) for a systematic methodology comparison rather than a single fixed recipe, described in Section 5.2.

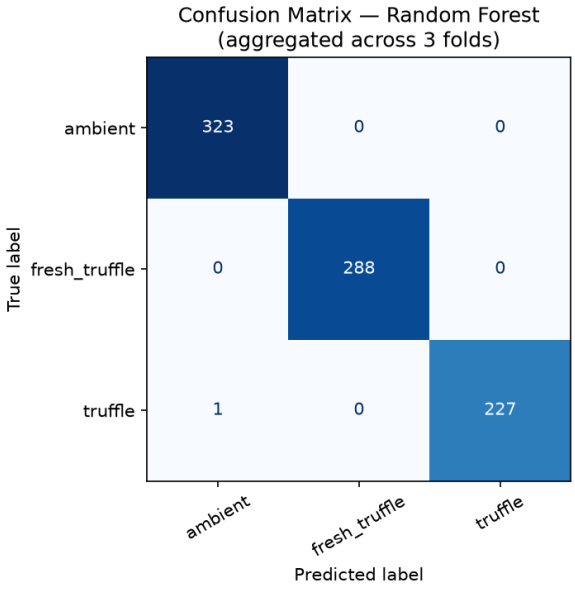
The truffle-types task's dataset was itself motivated by an earlier exploratory comparison between the project's original `truffle` sessions and a newly donated `fresh_truffle` sample, shown in Figure 4: a Mann-Whitney U / Cohen's d significance table across sensor channels, and a PCA projection showing the two groups separating into distinct clusters. This analysis, from the halfway-report period, is what first indicated the two truffle conditions were distinguishable and worth a dedicated classification task.



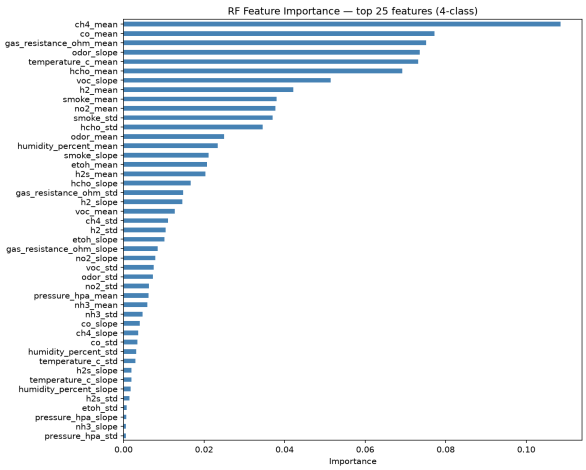
(a) 4-class (macro-F1 = 0.898)



(b) 3-class (macro-F1 = 0.936)

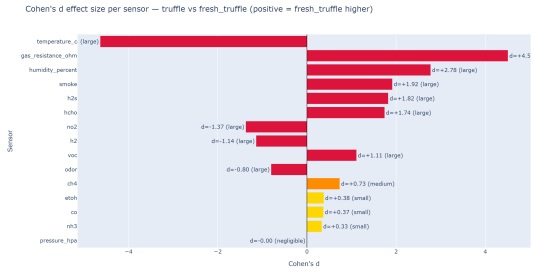


(c) Truffle-types (macro-F1 = 0.999*)

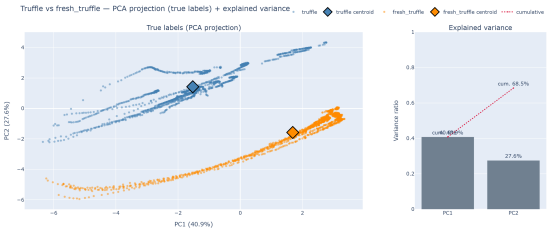


(d) 4-class Random Forest feature importance

Figure 3 Confusion matrices for the three earliest multi-class tasks, plus the 4-class task's Random Forest feature importance ranking. *See Table 6's caveat on the truffle-types result.



(a) Per-channel significance (Mann-Whitney U, Cohen's d)



(b) PCA projection, truffle vs. fresh_truffle

Figure 4 Exploratory truffle-vs-fresh_truffle comparison that motivated the truffle-types classification task: (a) per-sensor significance testing; (b) a PCA projection showing the two conditions separating into distinct clusters.

5.2 Methodology Comparison (Article Dataset)

Rather than extending the existing windowed-classical recipe directly, the article dataset was used to systematically compare seven modeling methodologies against one another before selecting a model for deployment.

Phase 1 (windowed classical). The existing recipe: 30 s windows, 15 s stride, mean/standard-deviation/slope per channel, then five classical models (Logistic Regression, KNN, Random Forest, SVM, Gradient Boosting) plus CatBoost, under `StratifiedGroupKFold` grouped by repetition. Logistic Regression led this phase (macro-F1 = 0.965). A regularization addendum then isolated the effect of penalty type on logistic regression specifically, fitting L1, L2, and elastic-net variants with the `saga` solver so the comparison varies penalty rather than solver: the L1-penalized (Lasso) variant scored highest (macro-F1 = 0.968 ± 0.042, accuracy = 0.970, balanced accuracy = 0.970), narrowly ahead of L2 (0.965) and elastic-net (0.961).

Phase 2a (row-wise). Classification directly on individual per-second readings, with no windowing at all. SVM led (macro-F1 = 0.968), the highest score of any 4-class methodology after Phase 2b. This result is treated with caution: grouped cross-validation groups by repetition, but it does not remove the temporal correlation between consecutive raw readings within the same repetition, so row-wise scores are expected to look optimistic relative to a genuinely new sample rather than reflecting a real improvement.

Phase 2b (per-repetition aggregate). Each repetition collapsed to one aggregated feature vector (mean/std/min/max/slope), giving 72 total examples. Random Forest scored highest overall across every methodology tested (macro-F1 = 0.973), but this methodology was not selected for deployment: it requires a full ≈300 s sample-measurement capture before any prediction is possible, conflicting with the fast, live-feedback positioning of the Live Scent Detection and Sniff & Speak pages (Sections 6, 8).

Phase 3 (PyTorch neural networks). Four architectures were trained on the same cross-validation folds as Phase 1, with early stopping on an inner `GroupShuffleSplit`: MLP-small, MLP-medium, CNN-small, and CNN-medium, operating on the windowed feature vectors and raw per-timestep voltage respectively. CNN-small led this phase (macro-F1 = 0.967), close to but not exceeding the classical Phase 1 baseline, consistent with the small-tabular-data regime this project operates in (Section 9).

Phase 4 (gradient-boosted trees). CatBoost, XGBoost, and LightGBM were trained on the windowed features as a dedicated boosting comparison, complementing the CatBoost runs already

included in Phases 1, 2a, and 2b. None exceeded the Phase 1 addendum's L1-penalized logistic regression.

Phase 5 (unsupervised embedding). A convolutional autoencoder learned a 16-dimensional embedding from raw per-timestep voltage, unsupervised, with the embedding then fed to a downstream Random Forest or CatBoost classifier. The best embedding pipeline (Embedding + CatBoost, macro-F1 = 0.916) underperformed the end-to-end Phase 3 CNN-medium (0.962), indicating the unsupervised embedding discards class-relevant signal that a supervised, end-to-end model retains.

Phase 6 (ambient-as-baseline delta). A reframing that drops `ambient` as an explicit class and instead classifies a per-repetition delta, sample-phase features minus that same repetition's own baseline-phase features, on the remaining three classes (truffle/coffee/black_pepper). Both a windowed-delta and a per-repetition-delta variant were tested; the per-repetition delta with CatBoost scored highest of any methodology in this comparison (macro-F1 = 0.983), but is not directly comparable to the 4-class methodologies above, since it solves a different, restricted problem and requires a baseline reading at inference time.

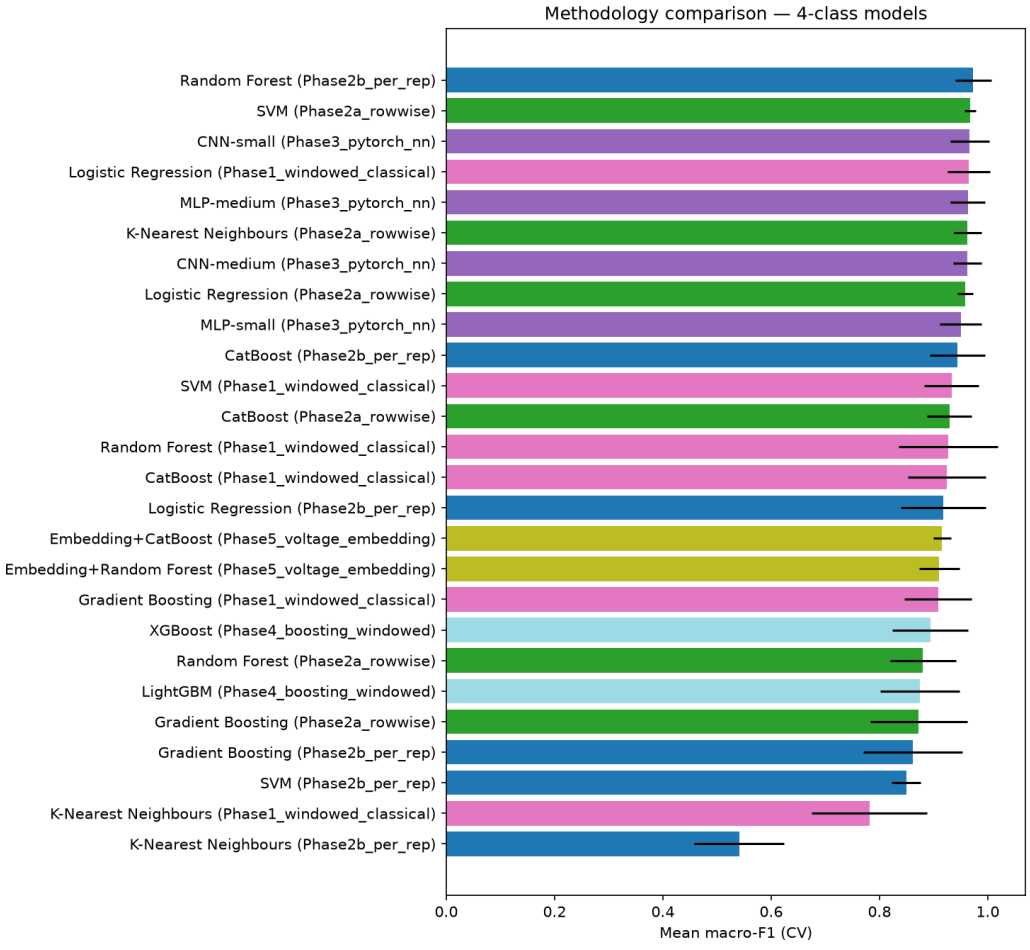


Figure 5 Cross-validated macro-F1 for every model/methodology combination tested, 4-class article dataset, ranked highest to lowest. Bar color groups results by methodology phase; error bars show the standard deviation across cross-validation folds.

5.3 The Deployed Model: Ovidio Classic Model (Legacy Model)

The L1-penalized (Lasso) logistic regression from the Phase 1 addendum was selected as the deployed model, referred to throughout the application and this report as the **Ovidio Classic Model (Legacy Model)**, for two reasons beyond its score: the L1 penalty drives uninformative sensor coefficients to zero, giving built-in feature selection and a more interpretable model, and it requires no new deployment infrastructure beyond the existing scikit-learn/joblib inference path already used by every other classifier in the system. It was chosen over the numerically higher-scoring Phase 2b Random Forest specifically because of the live-feedback constraint described above.

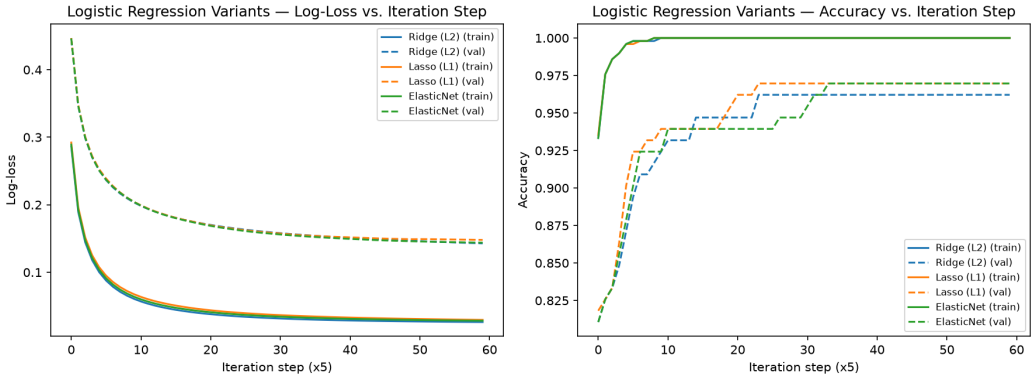


Figure 6 Training log-loss (left) and validation/training accuracy (right) against `saga` solver iteration for all three regularization variants compared in the Phase 1 addendum. Each curve is a genuine convergence trace, obtained by refitting the same estimator with `warm_start=True` at increasing iteration checkpoints. The Lasso (L1) curve (orange) is the deployed Ovidio Classic Model.

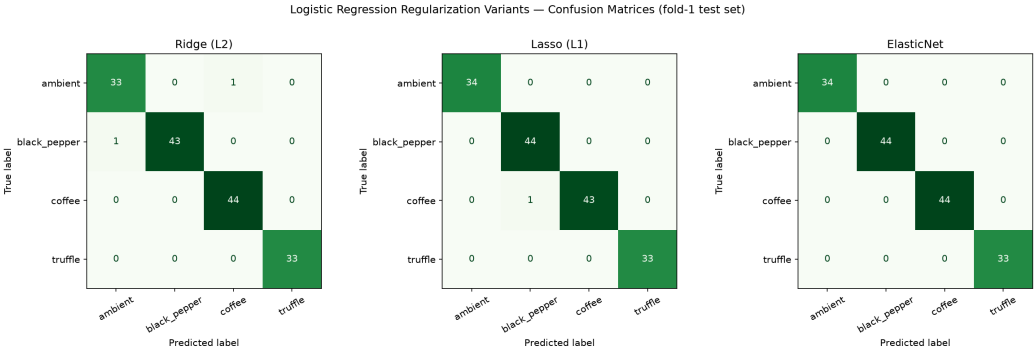


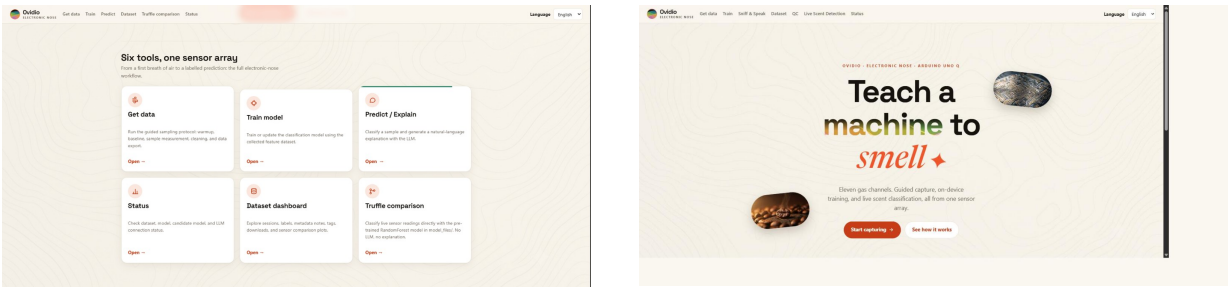
Figure 7 Fold-1 test-set confusion matrices for all three regularization variants. The center panel (Lasso, L1) is the deployed Ovidio Classic Model: 1 misclassification out of 155 test windows (a coffee window predicted as black_pepper), against 2 for Ridge (L2) and 0 for ElasticNet on this same fold.

6 System Implementation

The deployed application is a Flask web app (`python/app_factory.py`) built from six route blueprints (`collect`, `llm`, `dataset`, `train`, `classify`, `analysis`), exposed through seven top-level pages. Every classifier available to the application, including the Ovidio Classic Model (Legacy Model), is registered in a single lookup table (`python/classify/inference.py`) that maps a model key to its three-artifact bundle (model, scaler, label encoder); this same table is what the Train wizard (Section 7) writes new entries into when a user promotes a freshly trained model, making it selectable immediately, with no application restart. Screenshots in this section are captured from a local development instance running a small mock dataset, not from OvidioBench's full 82 +-session scale (Section 4); pages whose only informative content is a session count are described in text without a screenshot for this reason.

6.1 Interface Evolution

The web interface itself grew substantially over the internship. Figure 8 contrasts the halfway-report-era home page with the current one: the current interface adds the model-selectable Live Scent Detection dropdown, the Quality Control dashboard, and the Train wizard and Sniff & Speak pages described later in this section, none of which existed at the halfway mark.



(a) Home page, halfway report

(b) Home page, current

Figure 8 Interface evolution: (a) the application's home page as it stood at the halfway report; (b) the current home page.

6.2 Get Data

A guided acquisition protocol steps a user through sensor setup, warmup, sample configuration, baseline recording, sample measurement, and cleaning, live-plotting sensor values at each stage and writing the finished session to disk as a CSV file with a metadata sidecar.

6.3 Dataset

A dashboard for browsing recorded sessions: per-class row and repetition counts, label aliasing (joining spelling variants without editing the original CSV files), per-session metadata notes, selected-session CSV export, and sensor-by-sensor plotting and comparison across sessions.

6.4 Quality Control

A new dashboard applies deterministic checks to every recorded session rather than relying on a human catching a bad sample after the fact: a per-class clean/warning/error breakdown, a PCA-based class-separation plot showing which pairs of classes the array can and cannot tell apart, and a per-sensor response fingerprint (baseline-normalized net response per class) showing which channels drive that separation. Novelty and drift detection are code, not a language model

call; the design explicitly reserves any language-model role for narrating a verdict already reached deterministically, not for reaching it.

6.5 Live Scent Detection

A direct classification page (Figure 9) for a 30 s live capture against any registered classifier, selectable from a dropdown that lists every model in the registry: Live Scent Detection (ambient/black_pepper/coffee/truffle, the Ovidio Classic Model), Truffle Detection (ambient/coffee_ground/truffle), Truffle Freshness (ambient/fresh_truffle/truffle), Coffee Type (Arabica_coffee/Robusta_coffee/ambient), plus any model a user has promoted from the Train wizard.

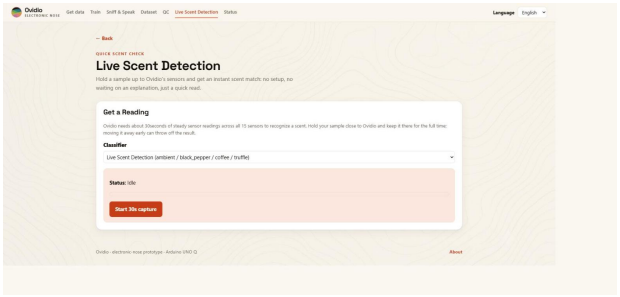


Figure 9 The Live Scent Detection page's model-selectable capture control.

6.6 Status

A single health dashboard (Figure 10) reporting dataset size, the currently promoted Train-wizard model (if any), the Sniff & Speak model and Piper voice readiness, and the language model endpoint's reachability, so a user can confirm the full pipeline is healthy without exercising every page individually.

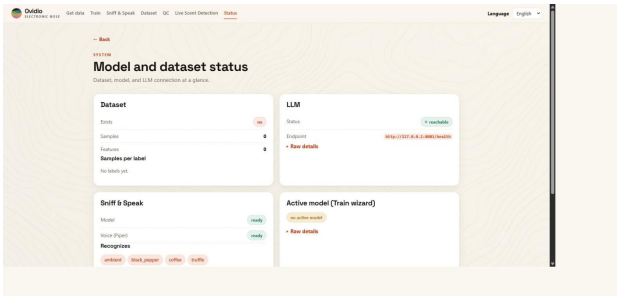


Figure 10 The Status page, reporting dataset, Sniff & Speak, and language-model health.

7 Train Model Page

The Train page (Figure 11) is a five-stage wizard that lets a user build and promote a new classifier from recorded sessions without writing code, implemented as a stateful pipeline (`python/train/pipeline.py`, `wizard_state.py`) behind a wizard API (`python/train/routes.py`). This wizard, and the browser-based retraining workflow it enables, did not exist at the halfway or second-progress-report stages; it is new to the final phase of the internship.

1. **Select** — the user picks at least two recorded sessions covering at least two labels from a session table (date, labels, repetition count).

2. **Load** — the selected sessions are loaded and reshaped into one wide-format dataframe, one row per second per sensor, reusing the project's `load_and_label()/to_wide()` functions.
3. **Explore the Data** — an in-browser exploratory data analysis stage computes class balance, a sensor correlation matrix, per-sensor boxplots and histograms, per-sensor time series, a 2D PCA projection, and a per-sample consistency heatmap.
4. **Prepare for Training** — the wide-format data is sliced into 30 s windows at 15 s stride (50 % overlap), and each window is reduced to a 45-dimensional feature vector (mean, standard deviation, and linear-regression slope for each of the 15 channels).
5. **Results** — six classical models are fit and ranked side by side under a group-aware train/test split (`GroupShuffleSplit`, so a session's windows never appear on both sides of the split): Logistic Regression, K-Nearest Neighbours, Random Forest, SVM, Gradient Boosting, and Lasso. Any resulting model can be named and promoted directly into the classifier registry used by Live Scent Detection, writing new model, scaler, and label-encoder files without an application restart.

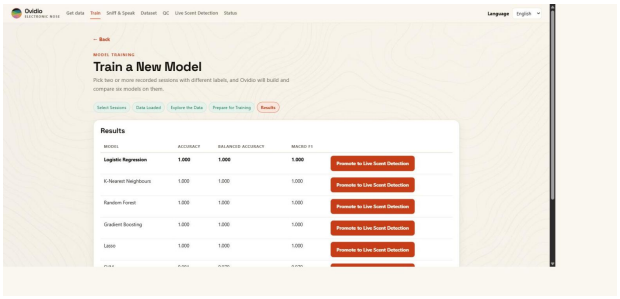


Figure 11 The Train wizard's Results stage, ranking six classical models on a demonstration run against a small local dataset; the near-perfect scores shown here reflect the size of that demonstration dataset rather than production performance, which is reported separately in Section 5.

8 Scent-to-Speech Pipeline

The second progress report benchmarked candidate text-to-speech engines in isolation, under a 2 GB memory budget matching the target Arduino UNO Q hardware, and identified Piper as the leading candidate; at that stage, the small-language-model (SLM) justification stage that would generate the spoken sentence's content did not exist yet, and the pipeline diagram from that report (Figure 12) shows TTS as the only implemented stage, with the SLM stage marked as planned future work. Since then, the full pipeline that benchmark was built toward has been implemented end to end and shipped as the application's Sniff & Speak page (Figure 13): pressing one button drives the device through an entire sense-classify-explain-speak cycle with no further input.

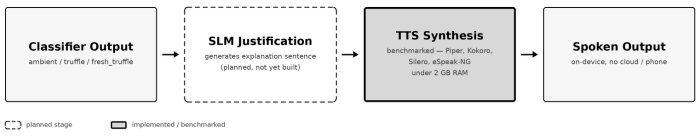


Figure 12 The scent-to-speech pipeline as it stood at the second progress report: classifier output feeds a future SLM justification stage (not yet built), which feeds the TTS synthesis stage actually benchmarked in that report, producing spoken output.

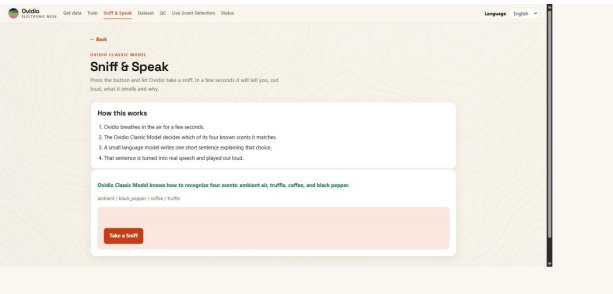


Figure 13 The Sniff & Speak page.

8.1 Pipeline Stages

1. **Gas Sensor Array** — the 11 gas channels plus BME680 environmental readings.
2. **Acquisition Window** — a 30 s capture window at 1 Hz, with saturated or missing readings forward-filled.
3. **Feature Extraction** — mean, standard deviation, and slope per channel, the same 45-dimensional feature vector used throughout the project.
4. **Classifier** — the Ovidio Classic Model (L1-penalized logistic regression, Section 5) returns a class label and its probability vector.
5. **Payload Builder** — the top 8 features by absolute value are packaged into a compact JSON payload.
6. **Few-shot Language Model** — Qwen2.5-0.5B-Instruct, served locally through `llama.cpp`'s `llama-server` over an OpenAI-compatible endpoint, writes a one-to-two-sentence justification grounded in the payload. This is the stage that did not exist at the second progress report.
7. **Speech Synthesis** — Piper (`en_US-lessac-medium` voice) renders that sentence to a WAV file.
8. **Client Browser** — the audio is played back automatically, with a manual play/stop control and a browser-native speech-synthesis fallback if server-side synthesis fails.

8.2 Language Model Selection

Qwen2.5-0.5B-Instruct was chosen specifically to fit inside the 2 GB on-device memory budget shared with every other running component. The model is not fine-tuned; steering happens entirely in context, through a fixed system prompt (spoken style, at most two sentences, no markdown, hedge under low confidence, cite at most one or two sensor clues, never alter the classifier's own probabilities) combined with five few-shot examples, one per class plus one deliberately low-confidence example, inserted as alternating user/assistant turns. The example matching the actual predicted class is always reordered to sit closest to the live request, exploiting the model's recency bias. Generation is constrained to at most 80 output tokens (English; slightly higher for Arabic and Italian), temperature 0.4, top-p 0.8.

8.3 TTS Model Selection

Four candidates were benchmarked under the same 2 GB constraint, each scored on cold-load time, real-time factor (synthesis time divided by audio duration), average synthesis time per sentence, and peak resident memory, using ten representative justification-style sentences as stand-in input.

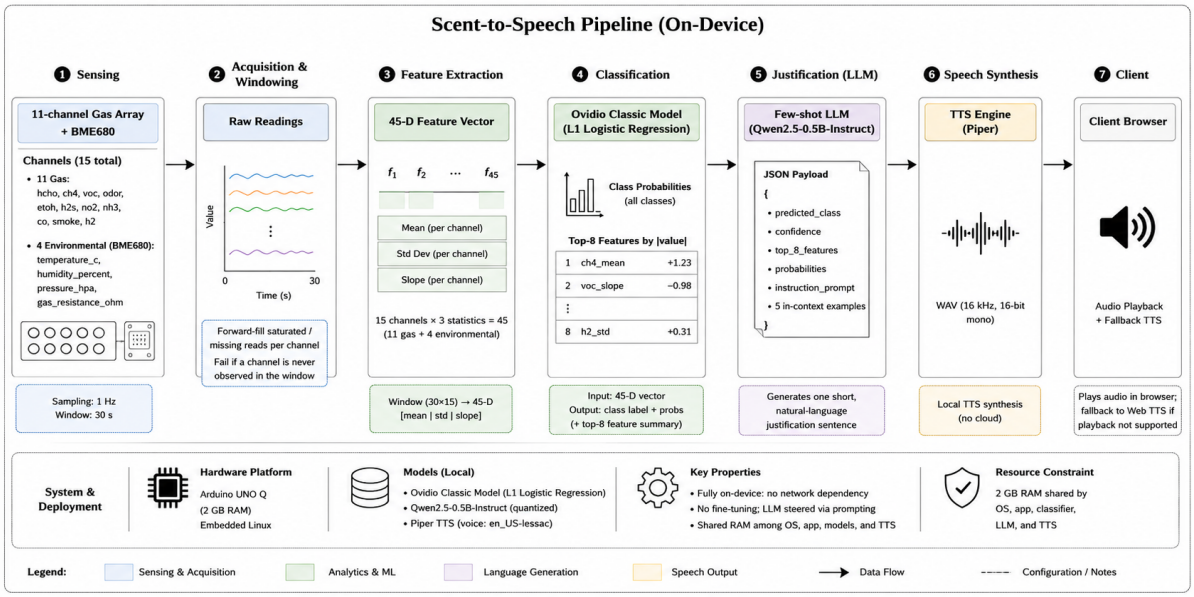


Figure 14 The completed scent-to-speech pipeline: sensing, acquisition and windowing, feature extraction, classification, justification (LLM), speech synthesis, and client playback, with the shared hardware platform, on-device models, key properties, and the 2 GB resource constraint summarized in the bottom band. Compare against Figure 12's pre-SLM version.

Engine	Status	Cold load (s)	RTF	Avg s/sentence	Peak RSS (MB)
Piper (`en_US-lessac-medium`)	ok	2.168	0.096	2.754	717.5
Silero (`v3_en`)	ok	0.949	0.355	2.558	881.5
Kokoro (82M, ONNX)	ok*	2.765	3.33	18.791	1039.0
eSpeak-NG (rule-based)	load_failed	—	—	—	—

Table 7 TTS engine benchmark, 2 GB memory cap, 10 test sentences.

*Kokoro's status is qualified rather than clean: across repeated `synthesize()` calls within a single live process, its peak resident memory climbed from 1,041 MB to 1,466 MB over just 10 additional calls, and once memory pressure crossed some threshold, per-sentence synthesis time collapsed from single digits to 30–50 s with no relationship to sentence length. Piper and Silero showed no such growth across all runs. Because the deployed TTS engine loads once and serves many inferences over its operating lifetime rather than a single call, a model whose memory grows with usage risks exhausting the shared 2 GB budget over time even when its first few calls look acceptable — this rules Kokoro out regardless of its otherwise competitive quality-per-parameter reputation. eSpeak-NG, a rule-based (non-neural) synthesizer included as a speed/memory floor, failed to load in the recorded benchmark run.

Piper was selected: it has the lowest cold-load time, the lowest average synthesis time per sentence, and a mid-range memory footprint with no growth over repeated calls. It is a VITS-based neural engine distributed with a lightweight C++/ONNX runtime, built specifically for Raspberry Pi-class boards with no GPU, the closest existing precedent to the Arduino UNO Q's own constraints. Two further candidates, Coqui TTS/XTTS and MeloTTS, were considered but not benchmarked: both target higher-quality multilingual synthesis with a correspondingly heavier dependency tree and larger models than a single-language justification sentence needs, and both remain candidates for a later round if the benchmarked shortlist shows an unacceptable quality gap once more memory headroom is known. A manual 1–5 listening-quality rating across all candidates was planned but has not yet been completed (Section 10), so Piper's selection rests on speed and memory alone at this stage.

8.4 End-to-End Resource and Latency Measurement

A dedicated benchmark harness measured the full language-model-plus-speech-synthesis round trip, with `llama-server` running the model under Q4_K_M quantization, CPU-only. Across 10 round trips, peak combined resident memory reached 1,058.7 MB (idle after both models load: 837.4 MB; drift between the first and last call: +6.4 MB, indicating no memory leak over this run length). Average language-model latency was 2.67 s (8.81 s on the first, cold call); average speech-synthesis latency was 0.79 s; average end-to-end latency was 3.46 s. Against an approximate 1.7 GB usable budget on the target board, this leaves roughly 640 MB of headroom. These figures were measured on an x86-64 development machine, not the board's ARM CPU, and are reported as a preliminary bound rather than a confirmed on-device measurement.

9 Discussion

The developments across the internship extend the project in two directions at once: outward, toward a durable, reusable artifact in OvidioBench, and inward, toward a system a non-technical user can operate, inspect, and retrain without touching code. OvidioBench formalizes 82+ sessions of the internship's own data collection into a named, citable benchmark, positioned for release on KaggleHub alongside the final submission; unlike a one-off internal dataset, this gives the sensor array's data a life beyond this specific project's model files.

The Train wizard and Quality Control dashboard, both new since the second progress report (Section 7), address a gap neither of the two earlier reports covered: a user with a laptop and a set of recorded sessions can now inspect data quality, retrain a classifier, and promote it into production, entirely inside the browser. This lowers the barrier between "collect more data" and "have a better model" from a manual notebook re-run to a five-step wizard.

Across every classification task in Table 6, and again across every phase of the article4class methodology comparison (Section 5.2), a classical model, tree ensemble or linear, has matched or

outperformed the deep-learning alternative trained on the same folds: no PyTorch architecture in Phase 3 exceeded the classical Phase 1/2b results, and the halfway report found the same pattern on its own tasks (Random Forest beating a 1D-CNN on every task it tried). This is consistent with the size and structure of the data collected throughout the project: a few hundred to a few thousand engineered feature vectors, drawn from 45 hand-crafted mean/std/slope statistics per channel, is a small-tabular-data regime where tree ensembles and regularized linear models are known to be competitive with, or better than, deep networks that would benefit more from larger sample counts. It also validates the pivot away from the original TensorFlow-Lite-Micro plan (Section 3): a full deep-learning deployment pipeline adds quantization and runtime complexity without a corresponding accuracy gain at this dataset size.

Converting raw ADC counts to gain-corrected `voltage_v` early in the pipeline (Section 3) was a necessary methodological fix, not a refinement: since gain settings changed across sessions and even across sensors within the same board, a model trained directly on raw counts would learn to key off gain configuration rather than scent. Using voltage instead is what allows sessions recorded at different gains, including every session pooled into OvidioBench, to be combined at all.

The drift between the local development copy and the physical device that the halfway report flagged as an open risk is now closed: a 2026-08-13 file-by-file comparison found identical bytes in every template, script, and route file (59/59 files) between the two, with the earlier drift fully reconciled.

The scent-to-speech pipeline and the WIO Terminal display (Sections 3.7, 8), first introduced together in the second progress report as two parallel extensions of the device's output side, one spoken and one visual, followed different paths afterward: the scent-to-speech pipeline was carried through to a fully implemented, benchmarked end-to-end system, while the WIO Terminal remained at its second-progress-report level of documentation for the rest of the internship. That asymmetry reflects where effort was directed, not a judgment on the WIO Terminal's usefulness. The scent-to-speech pipeline, now implemented end to end rather than benchmarked in isolation, is also where this project diverges most clearly from prior MOS-sensor electronic-nose work such as the spirits-identification pipeline of Sun et al. [4]: the classifier's label is a means to an explanation, not the final product. That choice carries the same edge-deployment tradeoff documented for lightweight classifiers on constrained hardware [5], now applied to a language model and a speech synthesizer sharing the same 2 GB budget as the classifier itself; Section 8's benchmark shows that budget holds on development hardware, with the open question of ARM-board parity still to be confirmed.

10 Limitations and Future Work

- **Scent-to-speech resource figures are not yet confirmed on-device.** Section 8's latency and memory measurements were taken on an x86-64 development machine; the board's ARM CPU has not yet been benchmarked directly, and this is the largest open item before the pipeline can be considered validated end to end.
- **Few-shot justification examples are English-only.** Spanish, Italian, and Arabic requests fall back to zero-shot prompting, with no measurement yet of how much that costs in explanation quality.
- **Speech synthesis offers a single voice.** Only the Piper `en_US-lessac-medium` voice is deployed; no listening-quality comparison against alternative voices or engines has been repeated since the second progress report's benchmark.

- **Two training paths coexist in the codebase.** A legacy, non-wizard training route (`python/train/trainer.py`, writing to `train_output/`) remains alongside the new Train wizard; consolidating onto the wizard path, or removing the legacy path, was not completed during the internship.
- **OvidioBench's exact composition has not been re-audited as a single artifact.** The IAEA and article subsets (Section 4) were individually vetted against their own session metadata; the remainder of the 82 + sessions has not yet been passed through the same per-session verification before the KaggleHub release.
- **Pump flow-rate estimate remains heuristic.** Unchanged since the second progress report: a controlled measurement with a flow meter and more runs per condition would replace the current bag-fill timing estimate.
- **WIO Terminal integration was never documented past the second progress report's level.** It is described in Section 3.7 only at the level supportable by the available photo; a fuller writeup of its data source, refresh rate, and connection to the RouterBridge/Flask pipeline, flagged as future work in the second progress report, was not produced during the remainder of the internship.

11 Conclusion

Across the internship, the project moved through three distinct stages, each documented by its own progress report and each building directly on the last. It began as a binary truffle/ambient classifier concept on ESP32/TinyML hardware and, by the halfway report, had pivoted to the Arduino UNO Q and validated a first defensible multi-class result (3-class macro-F1 = 0.936). The second progress report then extended the system's physical and output-side design without touching the classifier itself: a reduced-volume chamber confirmed not to cost any training data, a TTS engine benchmark, and a WIO Terminal display. The final phase closed the loop on both fronts: a systematically selected production classifier (the Ovidio Classic Model (Legacy Model), macro-F1 = 0.968 on a 4-class problem, chosen from a comparison of seven methodologies), a named, citable benchmark dataset (OvidioBench) ready for release on KaggleHub, a browser-based authoring pipeline for quality control and retraining, and a scent-to-speech stage that completed the SLM justification work the second progress report had only planned, turning a bare classification label into a spoken, sensor-grounded explanation. The remaining work before every part of this system is validated end to end is concentrated in one place: confirming the scent-to-speech pipeline's latency and memory figures on the board's own ARM hardware, the one measurement this report could not yet take on the target device itself.

References

[1] Amy Loutfi et al. "Electronic Noses for Food Quality: A Review". In: *Journal of Food Engineering* 144 (2015), pp. 103–111. DOI: 10.1016/j.jfoodeng.2014.07.019.

[2] Bosch Sensortec GmbH. *BME688: Low Power Gas, Pressure, Temperature & Humidity Sensor with AI*. Datasheet BST-BME688-DS000-03. Document revision 1.3. Bosch Sensortec GmbH, 2024. URL: https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme688-ds000.pdf.

[3] SGX Sensortech (Amphenol). *MiCS-6814: Compact Mixed Gas Sensor*. Datasheet 1143. Revision 8. SGX Sensortech / Amphenol Sensors. URL: https://www.sgxsensortech.com/content/uploads/2015/02/1143_Datasheet-MiCS-6814-rev-8.pdf.

[4] Jia-Tsung Sun et al. "Combining an Electronic Nose System with Artificial Intelligence Methods for Odor Identification of Spirits". In: *2024 10th International Conference on Applied System Innovation (ICASI)*. 2024, pp. 332–334. DOI: 10.1109/ICASI60819.2024.10547960.

[5] Lei Xiao et al. "Co-Design of Lightweight Neural Network and Low-Power Edge Computing Architecture for Intelligent Electronic Nose System". In: *IEEE Sensors Journal* 25.14 (2025), pp. 27288–27300. DOI: 10.1109/JSEN.2025.3569844.

[6] Vijay Janapa Reddi. *Introduction to Machine Learning Systems: Principles and Practices of Engineering Artificially Intelligent Systems*. Open-source web edition (matching the title cited in the project proposal) available at https://mlsysbook.ai/. Cited as the reference for the embedded/edge AI engineering workflow. MIT Press, 2026. ISBN: 9780262058889.