Baltic J. Modern Computing, Vol. 14 (2026), No. 4, 698-741
https://doi.org/10.22364/bjmc.2026.14.4.01

# From Runtime Inference to Algorithm Generation: Lessons Learned from Deploying Local LLMs in an Edge IoT Workflow

Ville PITKÄKANGAS

Vierimaantie Campus, Centria University of Applied Sciences, Vierimaantie 7, 84100 Ylivieska, Finland

ville.pitkakangas@centria.fi

ORCID 0000-0002-0847-2470

**Abstract:** The role of locally deployed large language models (LLMs) in edge Internet of Things systems remains unclear. This study examined runtime inference and offline code generation for temperature-based window-state determination. Gemma, Mistral, and GPT-OSS were evaluated as runtime inference engines, while GPT-OSS generated executable Python deployment logic from a truth-table specification. Latency, output-format inconsistency, and prediction-quality limitations constrained runtime inference. In synthetic analysis, an author-derived primary analytical rule achieved F1 = 0.892, compared with 0.894 for logistic regression. A human-extended analytical rule reproduced the synthetic reference boundary, serving as a consistency check rather than independent validation. In a closed-window measurement, the original GPT-OSS-generated implementation produced 173 false-positive samples; refinement removed them, but open-window performance was not evaluated. In a second measurement, refined predictions achieved F1 = 0.87 relative to temperature-derived proxy labels. GPT-OSS showed qualified value as an offline engineering assistant, while runtime inference remained impractical under the evaluated conditions.

**Keywords:** large language models; Internet of Things; sensor inference; edge computing; physical constraints; rule-based inference; window-state detection

## 1. Introduction

The integration of large language models (LLMs) into Internet of Things (IoT)-based monitoring systems has enabled new forms of sensor data interpretation, including natural-language reporting and high-level reasoning. However, LLMs operate primarily on statistical patterns rather than physical laws, raising concerns about their reliability in tasks requiring precise state inference.

Recent work has explored the use of LLMs for sensor data analysis in IoT and smart environments. LLM-based approaches can interpret multivariate sensor data and support natural-language interaction with sensor measurements, including zero-shot and few-shot sensor-understanding tasks (Li et al., 2024; Zhang et al., 2025). However, these approaches typically rely on cloud-based infrastructure and are not optimized for precise

physical-state inference. Deploying LLMs in IoT and edge environments introduces additional challenges related to latency, energy efficiency, and privacy. Prior studies propose layered architectures and optimization techniques, including quantization, retrieval-augmented generation, and chain-of-thought prompting, to improve reasoning performance (Kök et al., 2024; An et al., 2025). While effective, these methods generally assume significant computational resources. In smart building contexts, LLMs have been applied to tasks such as environmental monitoring and indoor condition analysis using multimodal inputs (Amangeldy et al., 2025; Arslan et al., 2025; He et al., 2025). However, their ability to reliably capture underlying physical relationships remains limited.

In parallel, large language models have increasingly been applied as software-development assistants capable of generating executable code from natural-language specifications. Recent surveys and empirical studies indicate that LLMs can support software engineering tasks ranging from code generation to interactive refinement workflows (Jiang et al., 2024; Fakhoury et al., 2024). However, limited evidence exists regarding the use of locally deployed LLMs for generating deterministic inference logic in resource-constrained edge-IoT systems.

Despite growing interest in LLM-assisted IoT systems, the practical role of local LLMs in resource-constrained edge environments remains unclear. In principle, local LLMs can be used in at least two ways. First, they may act as runtime reasoning engines that directly interpret sensor measurements and produce decisions. Second, they may function as offline engineering tools that generate executable algorithms subsequently deployed in deterministic inference pipelines. This distinction is important because the requirements of these roles differ substantially. Runtime inference emphasizes latency, resource efficiency, and operational robustness, whereas offline code generation emphasizes development support and algorithm synthesis. To date, limited empirical evidence exists comparing these alternatives in a practical edge-IoT setting.

This study began with an attempt to deploy local LLMs as runtime decision-making components in an edge-IoT system. In related LoLiPoP-IoT work, local-first processing and AI-assisted interpretation form part of a broader long-life sensing architecture in which validated data, traceability, and human oversight are prerequisites for operational AI use (Kalsi et al., 2026). Earlier project experience suggested that local LLMs could assist with sensor-report generation, but preliminary testing showed that direct runtime window-state inference was impractical under the tested hardware and application conditions because of excessive latency, inconsistent output-format compliance, and limitations in prediction quality. This finding motivated the subsequent shift from runtime inference to offline algorithm generation.

Rather than abandoning local LLMs entirely, the study was redesigned to investigate whether GPT-OSS could contribute value through offline algorithm generation. This transition motivated the comparison of the two roles examined in this paper.

This study addresses the following **research questions**:

- **RQ1.** Are freely usable local LLMs practical for direct real-time window-state inference on commodity hardware?
- **RQ2.** Can local LLMs instead be used effectively to generate algorithms for this task?
- **RQ3.** Does human refinement improve AI-generated algorithms?

The **main contributions** of this article are as follows:

- **MC1.** An empirical evaluation of local LLMs as runtime inference components under a constrained edge-IoT configuration.

- **MC2.** A documented case of GPT-OSS-assisted generation, operational integration, and human refinement of executable deployment logic.
- **MC3.** A controlled synthetic analysis of separate, author-derived analytical formulations and lightweight baseline classifiers.
- **MC4.** Practical lessons concerning implementation provenance, human code inspection, dataset-specific validation, and the evidentiary limits of synthetic and proxy-labeled evaluations.

The **novelty** of this work can be summarized as follows: Rather than proposing a new machine-learning model, this paper compares two roles for local LLMs in an edge IoT workflow: (1) runtime inference and (2) offline algorithm generation.

Figure 1 summarizes the overall study design, showing how each research question is addressed through a corresponding experiment and evaluation stage.

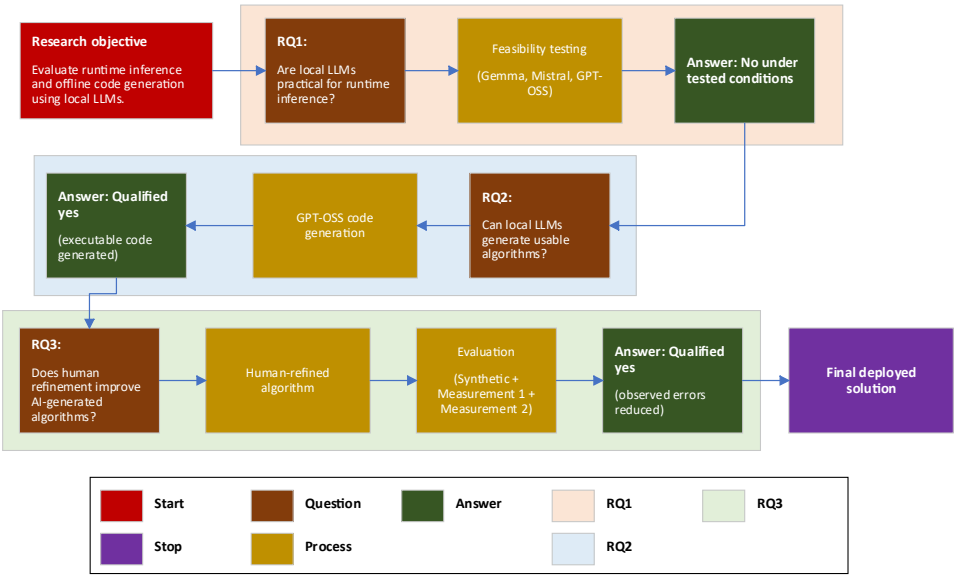


**Figure 1.** Research workflow and relationships between the research questions. Runtime inference (RQ1) was evaluated through feasibility testing and found impractical. The study subsequently shifted to offline code generation (RQ2) and human refinement (RQ3), resulting in the human-refined deployment implementation used in the operational workflow. The datasets used during evaluation (the synthetic dataset, Measurement 1, and Measurement 2) are described in Section 2.3. The RQ2 answer refers to executable code generation and operational integration, not to the author-derived synthetic-rule comparison. The RQ3 answer refers to suppression of the observed Measurement 1 false-positive mode; Measurement 2 provided supporting proxy-agreement evidence but did not permit a before-and-after comparison.

# 2. Materials and methods

## 2.1. Runtime-inference approach

The initial use case followed a local-first architecture for temperature sensor data ingestion, storage, processing, and interpretation. Sensor streams were normalized, stored in SQLite (Hipp, 2025), and analyzed in real time by a large language model (LLM), which returned an estimated window state (true = open, false = closed). Data transfer was handled via MQTT version 5 (Light, 2017) using the paho-mqtt client (v2.1.0) (Eclipse Foundation, 2024), while LLMs were hosted locally using Ollama (Ollama, 2025a) through Hypertext Transfer Protocol (HTTP) requests (Fielding and Reschke, 2014). Grafana (v12.2.1) (Grafana Labs, 2025) was used for visualization. All components were implemented in Python 3.10.11 (Python Software Foundation, 2023). The LLM was instructed to return the estimated window state in a simple JSON (Pezoa et al., 2016) format to minimize post-processing requirements.

Secure communication was ensured using Transport Layer Security (TLS) 1.2 (Dierks and Rescorla, 2008) with X.509 certificates (ITU-T, 2019) over Transmission Control Protocol (TCP) (IETF, 2022), complemented by username/password authentication.

To support modularity and contextual interpretation, the system defined allowed sensor names and optional enrichment settings. Each sensor name might map to multiple addresses; if unspecified, all associated addresses were accepted. Enrichment added metadata (e.g., sensor location such as near-window or room center) prior to processing.

Database maintenance enforced three constraints: (i) throttled sampling, (ii) bounded size, and (iii) removal of stale records. These mechanisms ensured timely analysis while preventing resource exhaustion. The system was executed on a laptop without a discrete GPU; detailed hardware and retained configuration information is provided in Section 2.5.

NumPy 2.1.2 (Harris et al., 2020), pandas 2.2.3 (pandas development team, 2020; McKinney, 2010), tabulate 0.9.0 (Astanin, 2022), and Matplotlib 3.10.7 (Hunter, 2007) were used for dataset generation and analysis. Visualization outputs include confusion matrices, validation heatmaps, and time-series plots.

Originally, the objective was to use the LLM to infer the window state in real time with sensor readings embedded in the prompt. The runtime-inference approach underwent several iterations intended to improve output consistency and latency. Unless otherwise noted, default inference parameters were used.

Mistral:7B (Jiang et al., 2023) was initially adopted for this approach. However, the model occasionally failed to follow the prompt faithfully, returning either free-form text or Python code instead of the expected JSON-formatted output indicating the window state.

To improve consistency, GPT-OSS:20B (OpenAI, 2025), which was also run locally, was adopted. This model reliably returned a JSON-formatted answer when prompted. Default inference settings were used, and thinking (reasoning) was initially set to “high” (according to the Ollama documentation (Ollama, 2025b), reasoning cannot be disabled for GPT-OSS, and three levels – “low,” “medium,” and “high” – are used instead), and “medium” and “low” were subsequently attempted. For debugging purposes, the prompt was extended to include a short explanatory sentence justifying the predicted state. This worked robustly during exploratory testing, generally yielding a plausible answer with a valid justification, and incurred minimal additional computation and runtime overhead.

To improve speed, a locally run Gemma3:1B (Gemma Team, 2025) was adopted and run with default inference settings.

Each model was evaluated separately in the same runtime-inference workflow; the models were substituted for one another and were never executed concurrently or connected in a processing chain.

Multiple configurations for the maximum database entry count per sensor (2, 6, 10) and timespan in minutes (1, 2, 5) were tested for optimal latency. The fastest response rate was obtained with the minimum values that the system supports (db_size=2, db_timespan_minutes=1).

Lastly, sensor enrichment settings were optimized by moving general information about measurements—such as room properties (air-conditioned, office), temperature unit (Celsius), and radio type (Bluetooth Low Energy)—to the prompt. This reduced redundant data sent to the LLM. The final enrichment settings were the same as in the revised approach explained in Section 2.2.

The final prompt used for all models in this approach was:

*"Can you estimate the window state from this JSON data? Use only temperature (Celsius) readings as indicators. Use only the latest measurement from each sensor. The sensor radios are Bluetooth Low Energy. The room is a typical, air-conditioned office room. Always give the answer in JSON format, with \"window_open\" as key and a boolean as value (true=open, false=closed). In addition, give a one-sentence, plain-text explanation/reasoning for the estimate. If you cannot give an estimate, assume the window state is false (closed)."*

The runtime-inference configuration described above was subsequently evaluated through exploratory feasibility testing, the results of which are presented in Section 3.1. It was ultimately rejected because it could not satisfy latency, output-format compliance, and prediction-quality requirements. Consequently, the system architecture was redesigned to use local LLMs for offline code generation rather than runtime decision making.

## 2.2. Offline code-generation approach

Following the runtime-inference experiments, the system was redesigned to use GPT-OSS for offline code generation instead of runtime decision making.

In the revised architecture shown in Figure 2, the LLM no longer participated in runtime decision making. Instead, GPT-OSS was used offline to synthesize deterministic Python code from a structured specification. The generated code was then executed locally as part of the operational inference pipeline. Both the generated and manually refined code are provided in Appendix F.

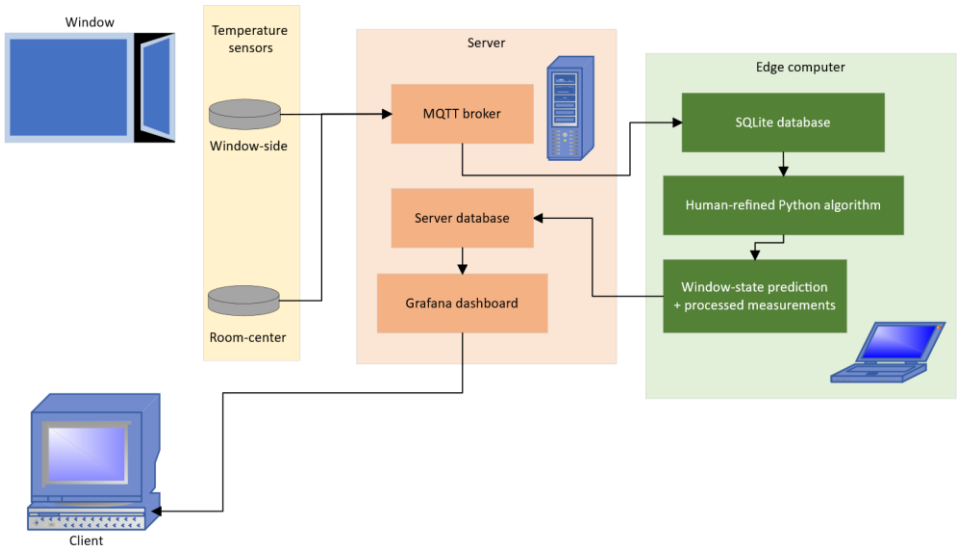


**Figure 2.** Final deployed system architecture. Temperature measurements are transmitted through an MQTT broker and stored in a temporary SQLite database on the edge computer. The human-refined Python algorithm operates on the stored measurements and generates window-state estimates. These estimates, together with additional processed measurement data, are subsequently transferred to the server-side infrastructure, where they are retained in a database and visualized through a Grafana dashboard accessible to clients. Unlike the runtime-inference architecture evaluated in RQ1, no LLM participates in operational inference.

The prompt used for code generation was designed to translate a structured truth table into deterministic inference logic. The prompt was: “Here's a table of window states, seasons, and temperatures. Based on this table, can you generate an algorithm to predict the window state and implement it in Python? Return only a ready-to-use Python code.”

A follow-up prompt was used: “Let's modify the code a bit: include both sensors and leave out season and A/C information. We assume it's always winter and the AC is permanently on.”

The prompt formulation used to generate the deployment implementation was intentionally kept simple to examine whether GPT-OSS could translate structured input into executable decision logic without task-specific fine-tuning. The GPT-OSS reasoning level (thinking parameter) was set to “high,” while, to the best of the retained records, other inference parameters remained at their Ollama defaults.

Distinct implementations were used in the synthetic and measurement-based analyses. The author-derived analytical formulations corresponding to Equations 1–2 were used in the synthetic evaluation, whereas the GPT-OSS-generated and human-refined deployment implementations documented in Appendix F were used in Measurement 1. Measurement 2 retained stored outputs generated during operation by the human-refined deployment-rule family. The analytical formulations are not GPT-OSS outputs and should not be interpreted as mathematical restatements or code-level representations of the Appendix F implementations.

Two author-derived analytical rules were evaluated using the synthetic dataset: a primary analytical rule containing an absolute-temperature-threshold condition and a hybrid analytical rule extending that formulation with a center-to-window temperature-difference clause.

Primary analytical rule (Equation 1):

$$\hat{y}_{\text{primary}} = \begin{cases} 1, & \text{if } \min(T_{\text{center}}, T_{\text{vent}}) \leq T_{\text{abs}}, \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

where $T_{\text{center}}$ and $T_{\text{vent}}$ are temperatures from the room-center and window-side sensors, respectively, and $T_{\text{abs}}$ is the absolute temperature threshold. Values 1 and 0 denote predicted open and closed window states, respectively.

Hybrid analytical rule (Equation 2):

$$\hat{y}_{\text{hybrid}} = 1 \Leftrightarrow [\min(T_{\text{center}}, T_{\text{vent}}) \leq T_{\text{abs}}] \vee [T_{\text{center}} - T_{\text{vent}} \geq \Delta T] \quad (2)$$

where $\Delta T$ is the center-to-window temperature-difference threshold in degrees Celsius. The hybrid analytical rule retains the absolute-threshold clause of Equation 1 and adds a temperature-difference clause. Candidate values of $T_{\text{abs}}$ and $\Delta T$ were examined through a validation-set parameter sweep.

The truth table in Table 1 was supplied to GPT-OSS as the primary specification for generating the deployment implementation documented in Appendix F, Listing 1. The author-derived analytical rules in Equations 1–2 were used separately in the synthetic evaluation and are not mathematical restatements or code-level representations of the deployment implementations documented in Appendix F.

**Table 1.** Truth table supplied to GPT-OSS for deployment-code generation. In the "Window open" and "AC on" columns, 1 denotes true (window open or AC on), while 0 denotes false (window closed or AC off). AC denotes air conditioning.

| Window open | AC on | Season | Temperature center | Temperature vent |
|---|---|---|---|---|
| 1 | 1 | Winter | 15 | 15 |
| 0 | 1 | Winter | 22 | 22 |
| 1 | 1 | Winter | 22 | 18 |
| 0 | 1 | Winter | 22 | 24 |
| 0 | 1 | Summer | Any | Any |

## 2.3. Datasets

For the LLM-as-code-generator approach, three datasets were used, each serving a distinct role:

- **Synthetic data:** controlled evaluation of the analytical rules and baseline methods
- **Measurement 1:** real-world analysis of failure modes and human refinement
- **Measurement 2:** agreement between stored operational predictions generated by the human-refined deployment implementation and post-hoc temperature-derived proxy labels

The synthetic data consisted of 10,000 independently generated samples (five seeds, 2,000 samples each) simulating winter conditions and did not model temporal

relationships between measurements. The dataset was generated using a Bernoulli process with p = 0.5, resulting in an approximately balanced distribution of open and closed window states. Synthetic reference labels for the set were defined as specified in Equation 3:

$$y_{\text{synthetic}} = 1 \Leftrightarrow \left[T_{\text{vent}} \leq T_{\text{center}} - \Delta_{\text{open}}\right] \vee \left[\min(T_{\text{center}}, T_{\text{vent}}) \leq T_{\text{abs}}\right] \quad (3)$$

Synthetic reference labels were generated using $T_{\text{abs}} = 20°\text{C}$ and $\Delta_{\text{open}} = 2°\text{C}$. Here, $\Delta_{\text{open}}$ is the center-to-window difference threshold parameter.

Additional details, including noise modeling and differences from the hybrid analytical rule (Equation 2), are provided in Appendices B and C.

Measurement 1 consists of a real-world time series containing 2,060 samples collected over 24 hours (5 February 2026) at approximately 30-second intervals. The window remained closed throughout the observation period. The dataset includes temperature readings from two sensors, predictions from the original and human-refined deployment implementations, and the directly known closed-window reference state. The human-refined deployment implementation used a 2°C center-to-window temperature-difference threshold. This deployment implementation is distinct from the hybrid analytical rule used in the synthetic evaluation. Measurement 1 used a BeeButton sensor positioned near the room center and a Ruuvi 8984 sensor positioned near the window.

Measurement 2 consists of a real-world time series containing 1,440 samples recorded between 12:00:00 and 23:59:59 on 25 February 2026. The available indicators suggest that the window was open for approximately 40–49 minutes. The temperature-derived proxy labels indicated a 40-minute interval from 14:16:00 to 14:56:00, whereas the stored deployment predictions indicated a 48-minute-30-second interval from 14:17:30 to 15:06:00. The manual notifications were approximately 43 minutes apart, occurring at approximately 14:27 and 15:10; these timestamps represent reporting times rather than exact physical transitions. Measurement 2 used a Ruuvi Air sensor for room-center measurements and a Ruuvi 8984 sensor near the window following a system upgrade. However, only the window-side temperature signal and stored predictions from the human-refined deployment implementation were preserved for post-hoc analysis. Because the room-center signal was unavailable, the dataset was used exclusively to assess agreement between the stored predictions and temperature-derived proxy labels; alternative methods requiring the missing signal could not be recomputed.

Temperature-derived proxy labels for Measurement 2 were derived using a threshold-based proxy, where the window state was labeled as open when the measured temperature fell below 18°C and closed otherwise, which reflects a clear separation between indoor baseline conditions (~21–22 °C) and reduced temperatures (~12–15 °C) observed during window opening. These labels are consistent with the reported window-opening events, providing temporally coarse qualitative corroboration of the proxy-label interpretation rather than exact sample-level validation.

To reduce noise, a sliding window of three samples was applied. The final label was set to open if at least one sample indicated an open state. The temperature-derived proxy labels were computed for Measurement 2 only.

Table 2 shows what data were preserved after the measurements; some data were lost during system upgrades between measurements and post-hoc analysis; these are denoted with a “No” in the table. All sensor signals required for runtime inference were available at the time of system operation. Consequently, the deployed system was able to generate the corresponding window-state predictions retained for later analysis. Note that the preserved Measurement 2 data contain only the window-side (vent) temperature signal.

Consequently, the primary analytical rule and fitted baseline classifiers that require the room-center temperature cannot be recomputed during post-hoc analysis. Furthermore, the system was upgraded between Measurements 1 and 2, during which the human-refined deployment implementation replaced the original deployment implementation as the operational window-state determination method.

**Table 2.** Data availability for post-hoc analysis. The availability columns refer to stored or recomputable algorithm predictions, not independently observed window states or reference labels.

| Dataset name | $T_{center}$ available | $T_{vent}$ available | Original deployment predictions available | Human-refined deployment predictions available | Supports recomputation of alternative methods |
|---|---|---|---|---|---|
| Synthetic | Yes | Yes | Not applicable | Not applicable | Yes |
| Measurement 1 | Yes | Yes | Yes | Yes | Yes |
| Measurement 2 | No | Yes | No | Yes | No |

For Measurement 2, predictions from the original deployment implementation are unavailable because the human-refined deployment implementation had already replaced it in the operational system at the time of data collection. Measurement 2 therefore retains only the predictions generated during operation by the human-refined deployment implementation. However, because the room-center temperature signal was not preserved following a sensor replacement and subsequent system upgrade, the primary analytical rule and fitted baseline classifiers requiring that signal cannot be evaluated retrospectively.

Consequently, the three datasets serve different purposes in the evaluation. The synthetic dataset enables controlled assessment of the complete rule logic, Measurement 1 evaluates behavior under real-world closed-window conditions, and Measurement 2 assesses agreement between stored predictions from the deployed human-refined implementation and temperature-derived proxy labels under real-world operating conditions.

## 2.4. Evaluation

For the synthetic analysis, agreement with the generated reference labels was summarized using precision, recall, F1 score, accuracy, and exact paired McNemar tests (McNemar, 1947). Measurement 1 was analyzed using sample-level false-positive counts and accuracy because it contained no open-window observations; consequently, positive-class precision, recall, and F1 were undefined. Measurement 2 was analyzed using precision, recall, and F1 relative to the temperature-derived proxy labels. For the synthetic analysis, logistic-regression and decision-tree classifiers implemented using scikit-learn (Pedregosa et al., 2011) were used as baseline methods. Their configurations and evaluation procedure are documented in Appendix H.

Synthetic datasets were used for parameter sweeps and controlled benchmarking. Real-world datasets were used to evaluate performance under practical conditions.

Runtime performance was measured using Python's time.perf_counter, and peak traced Python allocation was measured using tracemalloc. Benchmark scope differed by

analysis. The synthetic benchmark measured prediction over each 400-observation test partition, whereas the Measurement 1 benchmark covered CSV loading, timestamp alignment, preprocessing, prediction, and evaluation over 2,060 aligned observations. Peak traced Python allocation was measured using tracemalloc and does not represent total process resident memory.

The synthetic and real-world analyses used distinct rule implementations. The synthetic evaluation used the analytical implementations corresponding to Equations 1–2, whereas Measurement 1 used the original and human-refined deployment implementations documented in Appendix F. Measurement 2 used stored operational outputs from the human-refined deployment implementation. Table 3 identifies the implementation, workflow, and evidentiary purpose associated with each analysis.

**Table 3.** Provenance of the implementations, datasets, and evidence used in each analysis.

| Result or analysis | Implementation used | Decision condition or workflow | Evidentiary purpose |
|---|---|---|---|
| Synthetic primary analytical-rule evaluation | predict_primary(), the author-derived synthetic-harness implementation corresponding to Equation 1 | Predict open if $\min(T_{\text{center}}, T_{\text{vent}}) \leq T_{\text{abs}}$ | Agreement of the author-derived primary analytical rule with the generated synthetic reference labels |
| Synthetic hybrid analytical-rule evaluation | predict_hybrid(), the human-extended synthetic-harness implementation corresponding to Equation 2 | Predict open if $\min(T_{\text{center}}, T_{\text{vent}}) \leq T_{\text{abs}}$ or $T_{\text{center}} - T_{\text{vent}} \geq \Delta T$ | Consistency with the generated synthetic reference and effect of adding the temperature-difference clause |
| Synthetic parameter sweep | predict_hybrid(), corresponding to Equation 2 | Evaluate Equation 2 over candidate values of $T_{\text{abs}}$ and $\Delta T$ | Recovery of the parameter values defining the generated synthetic reference |
| Synthetic logistic-regression baseline | Logistic-regression implementation described in Appendix H | Center temperature, vent temperature, and center-to-vent difference; max_iter=1000; seeds 42–46 | Comparison of the analytical primary rule with a fitted linear classifier |
| Synthetic decision-tree baseline | Depth-3 decision-tree implementation described in Appendix H | Same three input features; max_depth=3; seeds 42–46 | Assessment of whether a simple nonlinear classifier can approximate the generated synthetic reference boundary |
| Measurement 1, original implementation | window_open_uncorrected(), corresponding to Appendix F Listing 1 | In winter, predict open if both rounded temperatures equal 15°C, or if $T_{\text{vent}} < T_{\text{center}}$ | Characterization of sample-level false-positive behavior during a closed-window episode |
| Measurement 1, human-refined implementation | window_open_corrected(), corresponding to Appendix F Listing 2 | In winter, predict open if both rounded temperatures equal 15°C, or if $T_{\text{center}} - T_{\text{vent}} \geq 2°\text{C}$ | Assessment of suppression of the identified false-positive mode |

| Result or analysis | Implementation used | Decision condition or workflow | Evidentiary purpose |
|---|---|---|---|
| Measurement 2 proxy-agreement analysis | Stored operational outputs generated by the human-refined deployment implementation | Compare retained predictions with post-hoc temperature-derived proxy labels; no post-hoc prediction recomputation | Agreement with temperature-derived proxy labels and temporally coarse corroboration from manual notifications |
| Runtime-inference feasibility study | Direct runtime LLM workflow described in Section 2.1 | Direct LLM interpretation of current sensor inputs | Feasibility under the tested hardware, model, and application conditions |

Consequently, the synthetic and real-world results should not be interpreted as evaluations of one universal primary-versus-hybrid rule pair. They concern distinct implementations and provide different forms of evidence.

## 2.5. Reproducibility considerations

**Notation.** Room-center and window-side temperatures are denoted by $T_{\text{center}}$ and $T_{\text{vent}}$, respectively. The absolute temperature threshold is denoted by $T_{\text{abs}}$. The hybrid analytical rule's center-to-window difference threshold is denoted by $\Delta T$, while the corresponding synthetic-generator parameter is denoted by $\Delta_{\text{open}}$. Python identifiers such as T_abs, dT, and open_delta are retained in verbatim code and data-column names. Appendix D retains the implementation-oriented labels and column names generated by the evaluation scripts to preserve traceability to the source code and CSV outputs. In these tables, primary and hybrid refer exclusively to the primary and hybrid analytical rules corresponding to Equations 1 and 2. The identifier T_abs corresponds to $T_{\text{abs}}$, while dT corresponds to $\Delta T$. The prefixes p_ and h_ denote results for the primary and hybrid analytical rules, respectively.

**Terminology.** This article distinguishes between two rule pairs. The terms primary analytical rule and hybrid analytical rule refer exclusively to the implementations corresponding to Equations 1 and 2 used in the synthetic evaluation. The terms original deployment implementation and human-refined deployment implementation refer to the GPT-OSS-generated and subsequently refined functions documented in Appendix F and evaluated using Measurement 1. Measurement 2 contains stored predictions generated during operation by the human-refined deployment-rule family. These analytical and deployment implementations should not be interpreted as interchangeable formulations of one universal rule pair.

**Hardware and model-execution environment.** Runtime-inference experiments were conducted on a Dell Latitude 7350 laptop running Microsoft Windows 11 Enterprise. The system was equipped with an Intel Core Ultra 5 135U processor operating at a reported base frequency of 1.6 GHz, with 12 physical cores and 14 logical processors, and 32 GB of installed RAM. The laptop used integrated Intel Graphics and had no discrete GPU. No GPU-specific Ollama configuration was applied.

**Ollama version and model settings.** The exact Ollama version and model-quantization metadata were not retained and could not be recovered during revision. An attempt to inspect the retained GPT-OSS installation during revision was unsuccessful because Ollama returned a tensor-size-overflow error. To the best of the retained records, model parameters were left at their Ollama defaults unless otherwise stated. For GPT-OSS, the reasoning level was an explicit exception: high, medium, and low reasoning levels were explored during feasibility testing, and the low setting produced the shortest reported response time. The reported timings should therefore be interpreted as observations from this specific hardware and software configuration rather than as general benchmarks of the evaluated models.

**Data and code availability.** The LLM-generated inference code and experimental parameters that were preserved are provided in the Appendices to support reproducibility. The tools for generating, analyzing, and visualizing the synthetic dataset are available from the author upon reasonable request. Exact reproduction of the latency values may not be possible because the complete software and model configuration was not preserved.

# 3. Results

## 3.1. Runtime inference feasibility (RQ1)

RQ1 investigates whether local LLMs can be used directly as runtime inference engines for window-state determination. This question was explored through feasibility testing using real sensor data and representative prompts. The objective was not to identify the best-performing model, but to determine whether the runtime-inference approach could satisfy practical requirements for output consistency, prediction quality, and response time. Mistral (Jiang et al., 2023), GPT-OSS (OpenAI, 2025), and Gemma (Gemma Team, 2025) were evaluated qualitatively with respect to these criteria.

Despite the methodological refinements described in Section 2, the overall approach remained unsuitable for real-time deployment. GPT-OSS:20B consistently produced correctly structured JSON output and plausible predictions, but required approximately 10 minutes to process inputs from two to three sensors and approximately 3 minutes when only a single sensor was used. Reducing the reasoning level shortened the runtime to approximately 25 seconds on average, but the latency remained too high for practical deployment. Alternative models were also tested. Mistral was not pursued further because of inconsistent behavior when following the required output format. Gemma3:1B achieved substantially lower latency but produced predictions considered implausible during manual inspection. Consequently, none of the evaluated runtime-inference configurations simultaneously satisfied the requirements for practical response time, output-format compliance, and prediction quality (Table 4).

**Table 4.** Summary of factors limiting practical deployment of the evaluated runtime-inference models.

| Model | Output compliance | Prediction quality | Practical assessment |
|---|---|---|---|
| Mistral:7B | Inconsistent | Not further evaluated | Rejected due to output inconsistency |
| GPT-OSS:20B | Consistent | Plausible | Rejected due to latency |
| Gemma3:1B | Generally acceptable | Implausible | Rejected due to prediction quality |

No evaluated model simultaneously satisfied the requirements for output-format compliance, prediction quality, and practical response time. These findings indicate that local LLMs were not practical for direct runtime inference under the tested conditions. Therefore, RQ1 is answered negatively. Consequently, the study was redirected toward offline code generation, which became the focus of RQ2 and RQ3.

## 3.2. LLM-assisted algorithm generation (RQ2)

RQ2 evaluates whether local LLMs can generate executable inference logic suitable for deployment in an edge-IoT environment. To complement the deployment evidence relevant to this question, the author-derived analytical rules were evaluated using the synthetic dataset, which provides a controlled environment with explicitly generated reference labels. The analytical rules were also compared against lightweight machine-learning baselines, namely logistic regression and decision tree classifiers.

The primary analytical rule occasionally produced false negatives when temperature differences indicated an open window but neither sensor value crossed the absolute threshold. The hybrid analytical rule eliminated these errors by introducing an explicit center-to-window temperature-difference clause. Both analytical rules executed in milliseconds per test batch and had similar peak traced Python allocations, indicating negligible computational requirements under the tested conditions. Across five seeds, the primary and hybrid analytical rules processed each 400-observation test partition in mean runtimes of 4.20 ms and 4.42 ms, respectively. Both had a mean peak traced Python allocation of 0.059 MB. The 0.22 ms difference between their mean batch runtimes was not considered operationally meaningful. These measurements cover prediction only and exclude synthetic-data generation, validation-set parameter selection, metric calculation, and plotting. Table 5 summarizes runtime and memory metrics; detailed per-seed tables appear in Appendix D.

**Table 5.** Prediction runtime and peak traced Python allocation for the analytical rules on the synthetic test partitions. Values are means across five seeds. CI denotes a confidence interval.

| Rule | Runtime, ms, mean (95% CI) | Peak traced allocation, MB, mean (95% CI) |
|---|---|---|
| Primary analytical rule | 4.20 (4.16 to 4.25) | 0.059 (0.058 to 0.059) |
| Hybrid analytical rule | 4.42 (4.07 to 4.77) | 0.059 (0.059 to 0.059) |

One unmeasured warm-up was followed by three measured repetitions for each seed. Runtime was measured using time.perf_counter, and memory was measured using tracemalloc. Memory values represent peak Python allocations during prediction and not total process resident memory. Data generation, validation-set parameter selection, metric calculation, and plotting were excluded. Both rules were benchmarked on the same 400-observation test partitions.

Figure 3 illustrates confusion matrices for the primary (left) and hybrid (right) analytical rules. The hybrid analytical rule eliminates the disagreements observed for the primary analytical rule, resulting in perfect agreement with the generated synthetic reference labels on the evaluated test partition. These patterns correspond to the observed improvement in recall from 0.806 to 1.000 and the F1 increase from 0.892 to 1.000, averaged across the five 2,000-sample datasets. The confusion matrices illustrate predictions on a representative test subset (400 samples), whereas aggregate metrics reported below are computed over all five seeds (n = 2,000 samples each).

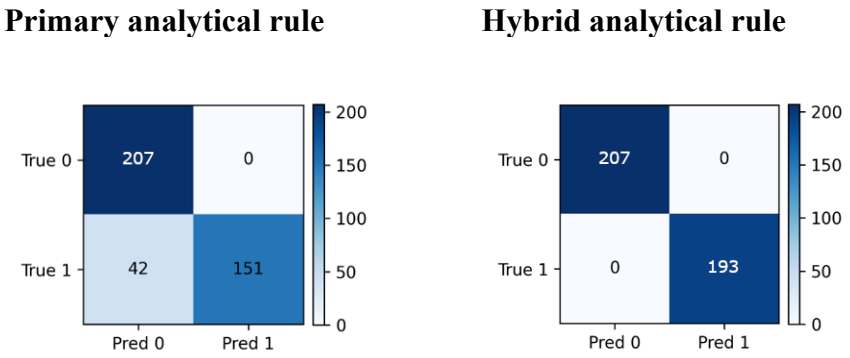


**Figure 3.** Confusion matrices for the primary (left) and hybrid (right) analytical rules evaluated on the synthetic dataset. The source set for these results is a 400-sample subset of a 2,000-sample dataset.

Agreement with the generated synthetic reference labels increased across all seeds: the primary analytical rule achieved F1 = 0.892 (95% CI: 0.880–0.905), while the hybrid analytical rule reached F1 = 1.000 with precision and recall of 1.000 (ΔF1 = +0.108). Table 6 summarizes these metrics. The result is expected because, when $T_{\mathrm{abs}} = 20.0°\mathrm{C}$

and $\Delta T = \Delta_{\text{open}} = 2.0°\text{C}$, the hybrid analytical rule's two decision clauses are equivalent to those used to generate the synthetic reference labels. The result therefore confirms consistency between the synthetic-data generator and the evaluated implementation and is not treated as independent evidence of predictive performance. The author-derived primary analytical rule achieved an F1 score of 0.892 relative to the generated synthetic reference labels, comparable to logistic regression at 0.894. This comparison characterizes the analytical formulation used in the synthetic evaluation and is not presented as direct evidence of GPT-OSS code-generation quality. The human-extended hybrid analytical rule achieved perfect agreement because it reproduced the synthetic generating boundary under the selected parameterization. This result is interpreted as a consistency and parameter-recovery check rather than as independent predictive validation. The two analytical rules differed on 194 test observations across the five seeds, all of which were matched by the hybrid analytical rule and missed by the primary analytical rule. The pooled exact McNemar comparison yielded $p<0.0001$, although the associated improvement should be interpreted within the generated synthetic specification.

**Table 6.** Summary of precision, recall, and F1 scores of the rules for the full synthetic dataset. CI denotes a 95% confidence interval.

| rule | F1 mean | F1 CI low | F1 CI high | Precision mean | Precision CI low | Precision CI high | Recall mean | Recall CI low | Recall CI high | Accuracy mean | Accuracy CI low | Accuracy CI high |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| primary analytical | 0.892 | 0.880 | 0.905 | 1.000 | 1.000 | 1.000 | 0.806 | 0.786 | 0.826 | 0.903 | 0.894 | 0.912 |
| hybrid analytical | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |

Figure 4 shows the validation F1 heatmap for the hybrid analytical rule across grid search parameters ($T_{\text{abs}} \in \{19, 20, 21\}$ and $\Delta T \in \{1.5, 2, 2.5\}$). The optimal configuration ($T_{\text{abs}} = 20°\text{C}$, $\Delta T = 2°\text{C}$) was selected for evaluation, as indicated by the highest F1 region. The parameter sweep recovered the same threshold values used in synthetic-data generation, $T_{\text{abs}} = 20.0°\text{C}$ and $\Delta T = 2.0°\text{C}$. Because these values define the synthetic decision boundary, the result should be interpreted as a consistency check showing that the optimization procedure correctly identifies the generating parameters rather than as an independent validation result.

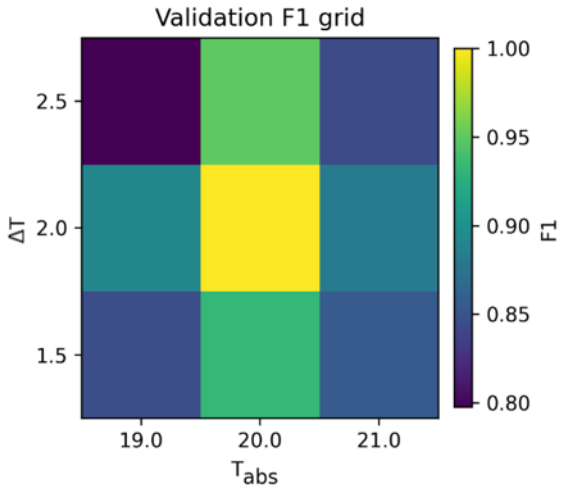


**Figure 4.** Validation F1 heatmap for the hybrid analytical rule across $T_{\text{abs}}$ and $\Delta T$. Values on both axes are expressed in degrees Celsius (°C).

To provide additional context, two lightweight baseline classifiers were evaluated against the synthetic reference labels. Logistic regression achieved a mean F1 score of 0.894, comparable to the primary analytical rule's mean F1 score of 0.892. A depth-3 decision tree achieved a mean F1 score of 0.996, indicating that a simple nonlinear classifier could closely approximate the deterministic synthetic reference boundary. The hybrid analytical rule achieved perfect agreement with the synthetic reference labels. These comparisons quantify agreement with the generated synthetic specification and should not be interpreted as validation against independently observed physical window states. However, because the hybrid analytical rule and the synthetic-data generator have equivalent decision boundaries when $T_{\text{abs}} = 20.0°\text{C}$ and $\Delta T = \Delta_{\text{open}} = 2.0°\text{C}$, its F1 score of 1.000 is interpreted as a consistency check rather than as evidence that it predictively outperformed the baseline classifiers. Consequently, the comparison between the primary analytical rule and logistic regression characterizes the behavior of the author-derived analytical formulation rather than GPT-OSS code-generation quality. The author-derived primary analytical rule achieved synthetic-reference agreement comparable to that of a fitted linear classifier despite not being fitted to the synthetic observations. The baseline classifiers were evaluated only on the synthetic dataset because Measurement 2 lacks the room-center temperature signal required for recomputation and post-hoc model evaluation. Table 7 reports mean test-set metrics across five independently generated 2,000-observation datasets and summarizes agreement between the four methods and the generated synthetic reference labels. The depth-3 decision tree's F1 of 0.996 indicates that the synthetic task is largely governed by simple deterministic threshold relationships. The results therefore do not establish an advantage of LLM-generated rules over conventional lightweight methods. Prediction runtime for each 400-observation test batch was measured over three repeated calls after model fitting. The reported memory values

represent peak Python allocations traced during prediction and exclude model fitting and total process memory.

**Table 7.** Precision, recall, and F1 scores of the primary analytical rule, hybrid analytical rule, and machine-learning baselines on the synthetic dataset.

| Method | Precision | Recall | F1 |
|---|---|---|---|
| Primary analytical rule | 1.000 | 0.806 | 0.892 |
| Logistic regression | 0.897 | 0.897 | 0.894 |
| Decision tree | 0.996 | 0.997 | 0.996 |
| Hybrid analytical rule | 1.000 | 1.000 | 1.000 |

The author-derived primary analytical rule achieved an F1 score of 0.892 relative to the generated synthetic reference labels, comparable to logistic regression at 0.894. This comparison characterizes the analytical formulation used in the synthetic evaluation and is not presented as direct evidence of GPT-OSS code-generation quality. The human-extended hybrid analytical rule achieved perfect agreement because it reproduced the synthetic generating boundary under the selected parameterization. This result is interpreted as a consistency and parameter-recovery check rather than as independent evidence of predictive superiority.

GPT-OSS generated the executable deployment implementation documented in Appendix F, Listing 1. This implementation was integrated into the operational workflow and, as shown in Section 3.3, produced 173 sample-level false-positive detections during Measurement 1. The generated implementation therefore demonstrated technical executability and operational integration but required human refinement to suppress the observed false-positive mode. The human-refined deployment-rule family subsequently generated the stored operational predictions evaluated in Measurement 2. RQ2 is therefore answered positively but with important qualifications: GPT-OSS generated executable and operationally integrated code, but because the original generated implementation was evaluated only during a closed-window episode, the study does not demonstrate that the generated code itself performed the complete open-versus-closed window-state detection task.

Collectively, the synthetic results indicate that simple deterministic rules and lightweight fitted methods can closely approximate the generated synthetic reference structure. Broader implications for LLM-assisted algorithm generation, implementation provenance, and human refinement are discussed in Section 4. Detailed per-seed evaluation results, including performance and resource metrics, McNemar statistics, and parameter-sweep results, are provided in Appendices D and E.

## 3.3. Effect of human refinement (RQ3)

RQ3 examines whether human refinement improved the behavior of LLM-generated inference logic under the conditions evaluated in this study. Measurement 1 and Measurement 2 were used because they provide real-world operating conditions not fully represented in the synthetic dataset. The primary objective is to assess the effects of human refinement on the deployment implementation under the observed real-world conditions.

Machine-learning baselines were not evaluated on Measurement 1 because the dataset contains only closed-window observations. Consequently, no meaningful conventional binary classification task can be constructed from the available labels. Similarly, machine-learning baselines could not be evaluated on Measurement 2 because the room-center temperature signal required for recomputation and post-hoc model training was not preserved.

Measurement 1 represents a real-world time series in which the window remained closed throughout the observation period. It was used to characterize false-positive behavior and assess whether human refinement could suppress spurious open-window predictions. Because no open-window cases occurred, this dataset does not evaluate open-window sensitivity, and positive-class precision, recall, and F1 are undefined.

Measurement 2 provides a separate real-world time series containing both proxy-open and proxy-closed periods. It was used to assess agreement between the stored predictions from the deployed human-refined implementation and a temperature-derived proxy label under conditions not represented in Measurement 1.

Measurement 1 contained 2,060 aligned observations, all representing the closed-window state. The original GPT-OSS-generated deployment implementation produced 173 sample-level false-positive detections, all occurring between 06:19 and 08:39, rather than occurring throughout the 24-hour measurement. The human-refined deployment implementation, which added a minimum 2°C center-to-window temperature-difference condition while retaining the exact 15/15 exception, produced no false-positive samples. The time-series plot of this dataset and its evaluation results are depicted in Figure 5, and the corresponding confusion matrices are shown in Figure 6. Because the dataset contains no open-window observations, the refined implementation's accuracy of 1.000 represents the absence of false-positive samples during this episode and should not be interpreted as complete window-state detection performance. Because no actual open-window observations occurred, this result does not establish open-window sensitivity. Instead, the measurement identifies a temporally concentrated false-positive mode in the original deployment implementation and shows that the added 2°C constraint suppressed that mode under the observed conditions.

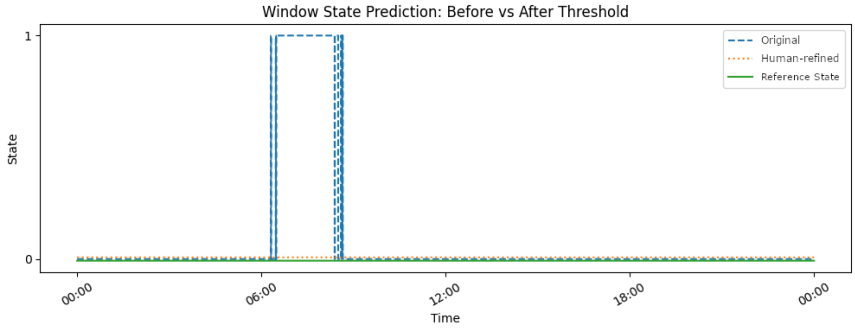


**Figure 5.** Window-state outputs from the original and human-refined deployment implementations during Measurement 1. The reference state remained closed throughout the measurement. The original implementation produced 173 false-positive samples concentrated between 06:19 and 08:39, whereas the human-refined implementation produced no false-positive samples. Measurement 1 therefore evaluates suppression of a specific false-positive mode and does not evaluate sensitivity to actual window-opening events.

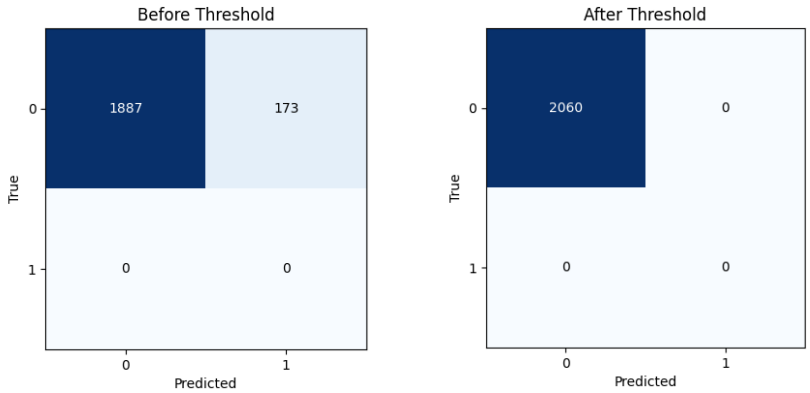


**Figure 6.** Confusion matrices for Measurement 1. The original GPT-OSS-generated deployment implementation produced 173 sample-level false-positive detections during the closed-window episode (left), whereas the human-refined deployment implementation produced none (right). Because the dataset contains no open-window observations, the figure characterizes false-positive suppression rather than positive-class detection performance.

An always-closed reference likewise produced no errors because every observation represented the closed-window state. The refined implementation therefore cannot be distinguished from this trivial reference using Measurement 1 alone. The meaningful result is narrower: the human-added condition suppressed the sample-level false-positive behavior exhibited by the original deployment implementation during the observed episode. Measurement 1 contains only negative cases; therefore, accuracy should be interpreted with caution. An always-closed classifier would also achieve perfect accuracy. Therefore, the principal result is the reduction of false-positive samples from 173 to 0.

Runtime and peak traced Python allocation were evaluated for the complete Measurement 1 pipeline using an initial unmeasured warm-up followed by three measured runs. The measured pipeline included CSV loading, timestamp alignment, preprocessing, prediction, and evaluation. Table 8 summarizes the results.

**Table 8.** Runtime, peak traced Python allocation, and false-positive sample count of the complete Measurement 1 pipeline. Values are means across three measured runs after an initial unmeasured warm-up.

| **Metric** | **Original deployment implementation** | **Human-refined deployment implementation** |
|---|---|---|
| Runtime (s) | 0.0394 ± 0.0020 | 0.0376 ± 0.0002 |
| Peak traced Python allocation (MB) | 0.72 ± 0.00 | 0.72 ± 0.00 |
| Accuracy | 0.916 | 1.000 |
| False-positive samples | 173 | 0 |

**Note.** The pipeline measurements include data loading, timestamp alignment, preprocessing, prediction, and evaluation over 2,060 aligned observations. Peak traced Python allocation was measured using tracemalloc and does not represent total process resident memory. Because Measurement 1 contains only closed-window observations, an always-closed reference would also achieve an accuracy of 1.000. Accuracy therefore indicates only the absence of sample-level false positives and does not measure open-window detection performance.

The two Measurement 1 implementations had similar computational requirements. The original and human-refined complete pipelines required mean runtimes of 0.0394 s and 0.0376 s, respectively, and both had a mean peak traced Python allocation of 0.72 MB. The small runtime difference was not considered operationally meaningful. These values are not directly comparable with the synthetic prediction-only benchmark because the Measurement 1 benchmark included the complete pipeline over 2,060 observations, whereas the synthetic benchmark measured prediction only over 400-observation test partitions.

Collectively, the Measurement 1 results show that the human-added 2°C condition suppressed the false-positive mode observed in the original deployment implementation during this closed-window episode, without a meaningful increase in computational cost. However, because the dataset contains no open-window observations and the always-closed reference also produced no errors, Measurement 1 does not establish open-window sensitivity or superiority over the trivial reference.

Measurement 2 evaluates agreement between deployed predictions and a temperature-derived proxy label. The reported metrics should not be interpreted as direct validation against independently observed window states. Table 9 compares the opening and closing indications obtained from the stored predictions, proxy labels, and manual notifications.

**Table 9.** Opening and closing indications from the deployed predictions, temperature-derived proxy labels, and manual notifications for Measurement 2.

| Evidence source | Opening indication | Closing indication |
|---|---|---|
| Deployed prediction | 14:17:30 | 15:06:00 |
| Temperature-derived proxy | 14:16:00 | 14:56:00 |
| Manual notification | approximately 14:27 | approximately 15:10 |

The inferred interval lengths therefore differed by up to approximately 8.5 minutes across the three evidence sources.

The manual notifications establish that an opening and subsequent closing occurred, but their timestamps indicate when the events were reported rather than necessarily when the physical transitions occurred. They are therefore used as temporally coarse independent corroboration, not as exact sample-level labels.

For Measurement 2, the stored predictions generated during operation by the human-refined deployment implementation were compared with the temperature-derived proxy labels defined in Section 2.3. No additional parameter adjustment was performed. Figure 7 shows the window-side temperature measurements, the temperature-derived proxy labels, and the corresponding stored predictions.

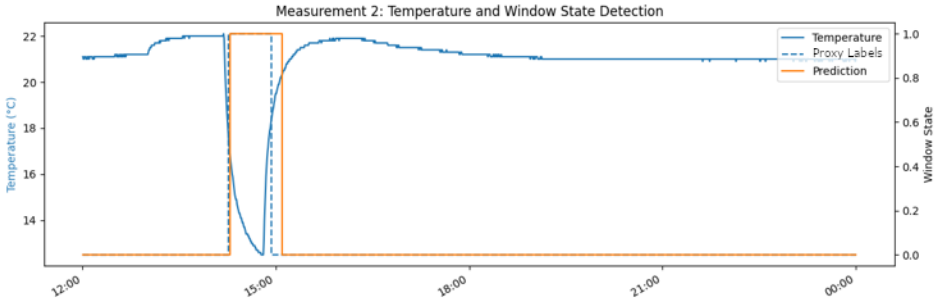


**Figure 7.** Time-series comparison of measured window-side temperature ("Temperature"), temperature-derived proxy labels ("Proxy Labels"), and stored predictions from the human-refined deployment implementation ("Prediction") for Measurement 2. The prediction changes to open later than the proxy labels and changes back to closed later than the proxy labels, producing false-negative disagreements near the opening transition and false-positive disagreements near the closing transition. The prediction and proxy series use different temperature-based decision procedures, and the proxy labels additionally incorporate a three-sample temporal filter.

The stored predictions were evaluated against the temperature-derived proxy labels without additional parameter adjustment. The resulting confusion matrix is shown in Figure 8.

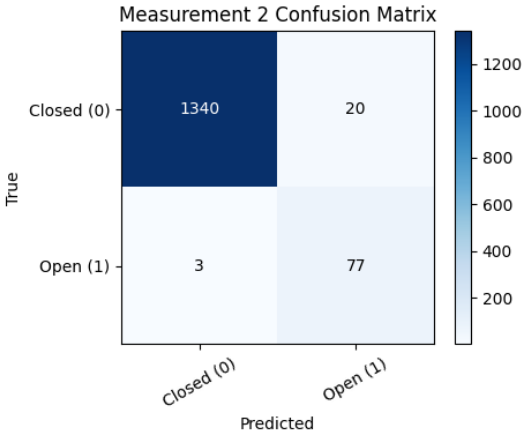


**Figure 8.** Confusion matrix comparing the stored predictions from the human-refined deployment implementation with the temperature-derived proxy labels for Measurement 2. The reference labels are temperature-derived proxies rather than independently observed sample-level window states.

Relative to the temperature-derived proxy labels, the stored predictions produced three false-negative and 20 false-positive disagreements, yielding precision of 0.79, recall of 0.96, and F1 score of 0.87. Relative to the temperature-derived proxy labels, the stored predictions achieved high recall and only three false-negative disagreements. These values

quantify proxy agreement and should not be interpreted as independent open-window detection accuracy. An always-closed reference would achieve accuracy of 0.944 because of the predominance of proxy-closed samples, but recall and F1 for the proxy-open class would both be zero. The deployed predictions therefore captured the proxy-open period in a way that the majority-state reference did not.

The three false-negative disagreements occurred at the beginning of the proxy-open interval, between the proxy transition at 14:16:00 and the deployed-prediction transition at 14:17:30. The false-positive disagreements were concentrated near the end of the interval, between the proxy transition to closed at 14:56:00 and the deployed-prediction transition at 15:06:00. These timing differences may reflect differences in threshold logic, temporal filtering, thermal response, and the temperature signals used by the two procedures. Consequently, the stored predictions and proxy labels need not change state at exactly the same time, and the disagreements cannot be attributed solely to deployment-rule failure or solely to sensing dynamics.

These findings illustrate how differences in sensing, threshold logic, and temporal processing can affect proxy-agreement metrics.

Human refinement changed the observed behavior of the GPT-OSS-generated deployment implementation in Measurement 1: the human-added 2°C condition suppressed the observed sample-level false-positive mode. Measurement 2 provides proxy-agreement evidence only for the human-refined deployment-rule family and does not permit a before-and-after comparison. Separately, adding a temperature-difference clause to the author-derived analytical formulation increased agreement with the generated synthetic reference labels, although the resulting perfect agreement reflects equivalence with the generating boundary. RQ3 is therefore answered positively within the limited deployment conditions and evidence available.

## 3.4. Summary of research-question outcomes

The overall outcomes of the study are summarized in Table 10.

**Table 10.** Summary of research-question outcomes and supporting evidence.

| Research question | Answer | Key evidence |
|---|---|---|
| RQ1 | No, under the tested conditions | GPT-OSS latency exceeded deployment requirements; the other models showed output-format-compliance or prediction-quality limitations. |
| RQ2 | Qualified yes | GPT-OSS generated executable code that entered the operational workflow, but the original implementation required human refinement and was evaluated only on closed-window data. The synthetic comparison used separate author-derived rules. |
| RQ3 | Yes, within the evaluated conditions | Human refinement reduced the 173 false-positive detections in Measurement 1 to zero. Measurement 2 provided proxy-agreement evidence but not a before-and-after comparison. |

The results provide answers to the three research questions introduced in Section 1. These answers are necessarily qualified by the hardware, datasets, implementations, and operating conditions evaluated in this study. Runtime inference using local LLMs was found to be impractical under the tested conditions because no evaluated model simultaneously satisfied the requirements for output-format compliance, prediction quality, and response time. Although GPT-OSS produced the most reliable outputs, its latency remained too high for operational use.

The results provide a qualified positive answer to RQ2. GPT-OSS generated executable Python deployment logic that was integrated into the operational workflow and formed the basis of the human-refined deployment implementation. However, the original generated implementation required human correction and was evaluated only during the closed-window Measurement 1 episode. Separately, the author-derived primary analytical rule achieved an F1 score of 0.892 relative to the generated synthetic reference labels, comparable to logistic regression at 0.894. This comparison characterizes the analytical formulation used in the synthetic evaluation and does not directly measure GPT-OSS code-generation quality. The hybrid analytical rule's perfect agreement is interpreted as a consistency and parameter-recovery result because it reproduced the synthetic generating boundary under the selected parameterization. The decision-tree result further indicates that the generated synthetic reference structure can be closely approximated using a simple nonlinear classifier. Together, the analytical-rule and baseline results indicate that simple, interpretable methods can closely approximate the deterministic structure of the generated synthetic reference labels.

In Measurement 1, the human-added 2°C condition suppressed the sample-level false-positive mode observed in the original GPT-OSS-generated deployment implementation, without a meaningful increase in computational cost. Measurement 2 provides additional proxy-agreement evidence for the human-refined deployment-rule family but does not permit a direct before-and-after comparison. Separately, adding a temperature-difference clause to the author-derived analytical formulation increased agreement with the generated synthetic reference labels, although its perfect agreement is interpreted as a consistency result rather than independent predictive validation.

Taken together, these findings suggest that local LLMs may be more valuable as offline engineering assistants than as runtime reasoning engines in resource-constrained environments. Rather than directly performing operational inference, GPT-OSS provided practical value as an offline engineering assistant by generating executable logic that could subsequently be refined by human experts and deployed as deterministic code.

# 4. Discussion and limitations

## 4.1. Interpretation of findings

A key finding is the substantial computational gap between runtime LLM inference and deterministic inference logic. Under the tested laptop configuration and for the evaluated window-state task, the local LLMs required tens of seconds to several minutes to produce predictions, whereas the deterministic rule-based implementations executed in milliseconds. This difference motivated the architectural transition from runtime reasoning to offline code generation.

The baseline comparisons provide additional context for interpreting the analytical rules. The author-derived primary analytical rule achieved an F1 score of 0.892, comparable to the logistic-regression baseline at 0.894. This comparison indicates that the analytical formulation approximated much of the generated synthetic reference structure without being fitted to the synthetic observations; it is not interpreted as evidence of GPT-OSS code-generation quality. The depth-3 decision tree achieved an F1 score of 0.996, showing that a simple nonlinear classifier could closely recover the deterministic synthetic reference boundary. The hybrid analytical rule achieved perfect agreement because it reproduced that boundary under the selected parameterization; its result is therefore interpreted as a consistency check rather than as evidence of superiority over the fitted baselines. Collectively, these findings indicate that the synthetic task is largely governed by deterministic threshold relationships and can be addressed effectively using simple, interpretable methods.

The outcomes also indicate that the selected 2°C difference threshold suppressed the observed false-positive mode in Measurement 1, although broader transferability was not evaluated. Controlled simulation may nevertheless provide a useful environment for implementation checking and early-stage failure analysis when independently labeled datasets are unavailable, provided that synthetic agreement is not treated as external validation.

Residual disagreements in Measurement 2 occurred near the opening and closing transitions. The stored predictions changed to open later than the temperature-derived proxy labels, producing three false-negative disagreements, and changed back to closed later than the proxy labels, producing false-positive disagreements. These timing differences may reflect differences in threshold logic, temporal filtering, thermal response, and the temperature signals used by the two procedures. They cannot be attributed solely to deployment-rule failure or solely to sensing dynamics.

This observation also has implications for evaluation. When labels are derived from proxy signals rather than direct observations, some discrepancies may reflect physical delays and sensing characteristics rather than classification errors. Performance metrics should therefore be interpreted in the context of both sensor behavior and environmental dynamics.

The runtime-inference experiments also highlight a more fundamental challenge of applying LLMs to physically grounded inference tasks. The generated code and runtime outputs did not demonstrate an inherent understanding of the underlying physical processes; effective deployment still required explicit domain constraints, provenance tracking, validation against observed behavior, and human refinement.

The observed effect of the human-added temperature-difference condition is consistent with evidence from other sensor-based applications that physical knowledge and domain-specific constraints can strengthen AI-assisted solutions. In photovoltaic power forecasting, Gjylapi et al. (2026) found that physically informed constraints improved predictive performance and operational robustness.

Additionally, the results demonstrate that useful outcomes may be obtained from LLM-assisted development even when direct runtime inference proves impractical. Although none of the evaluated models satisfied the requirements for operational deployment as real-time reasoning engines, GPT-OSS generated executable deployment logic that was integrated into the operational workflow and formed the basis of the human-refined implementation. This workflow suggests that local LLMs may provide value in

resource-constrained environments by supporting inspectable software development and algorithm synthesis rather than participating directly in online reasoning.

Overall, the findings suggest that local LLMs may provide greater value as software-generation assistants than as runtime reasoning engines in resource-constrained edge-IoT systems.

Beyond answering the research questions, the study provides several practical observations (Section 4.2) regarding the deployment of local LLMs in resource-constrained IoT environments. These observations may help practitioners select appropriate roles for LLMs in similar systems.

## 4.2. Lessons learned

**1. Local LLMs may provide greater value as engineering assistants than as runtime reasoning engines.**

In the tested hardware and application, direct LLM-based reasoning introduced unacceptable latency and, for some models, limitations in output compliance or prediction quality. However, GPT-OSS provided qualified value when used offline to generate executable deployment code. Before adopting LLMs for operational inference, practitioners should evaluate whether the latency, output-format compliance, and prediction-quality requirements of the intended deployment can realistically be satisfied.

**2. AI-generated code should be reviewed, validated, and refined by human experts.**

The GPT-OSS-generated deployment implementation captured useful elements of the intended logic but required human review and refinement. In Measurement 1, the original deployment implementation exhibited a sample-level false-positive mode that was suppressed by the human-added temperature-difference condition. Human expertise was required to identify failure modes, interpret sensor behavior, and introduce a temperature-difference constraint that improved robustness. The final solution therefore combined AI assistance with domain knowledge rather than replacing the engineer.

**3. Simple, interpretable algorithms can be effective for narrowly defined IoT tasks.**

The analytical rules and deployment implementations remained computationally inexpensive and interpretable. The depth-3 decision-tree baseline also closely approximated the synthetic reference boundary, suggesting that complex runtime AI may be unnecessary when a narrowly defined task is governed by simple deterministic relationships.

**4. Early feasibility studies can prevent unnecessary development effort.**

The runtime-inference approach was evaluated before extensive integration work had been completed. This revealed practical limitations early in the development process and enabled a redesign before significant additional effort had been invested. Small-scale

feasibility studies may therefore be valuable when assessing emerging AI technologies for edge deployments.

**5. Negative results can support better engineering decisions.**

The original runtime-inference approach did not satisfy practical deployment requirements. However, identifying its limitations led directly to a more effective architecture. Publishing such findings may help practitioners avoid unproductive design paths while encouraging alternative uses of the same technology.

## 4.3. Limitations

Several limitations should be considered when interpreting the results.

First, the runtime-inference evaluation was conducted as an exploratory feasibility study rather than a controlled benchmarking experiment. Consequently, observations regarding latency, output-format compliance, and prediction quality should be interpreted qualitatively rather than as definitive comparisons between models.

Second, the study evaluated only a small number of local LLMs and a single deployment environment. Different hardware platforms, model variants, quantization strategies, or future generations of LLMs may yield different results.

Third, Measurement 2 lacked the room-center temperature signal following a sensor replacement and subsequent system upgrade. As a result, the primary analytical rule and fitted baseline classifiers requiring the room-center signal could not be recomputed during post-hoc analysis. Although the dataset remains useful for assessing agreement between the stored deployed predictions and the temperature-derived proxy labels, it does not provide independent sample-level validation against directly observed window states, nor does it support the same range of analyses as the synthetic dataset or Measurement 1. This experience also highlights the importance of preserving raw measurement data before modifying operational systems, particularly when future post-hoc analyses are anticipated.

Fourth, the evaluation focused on a winter deployment scenario in which air conditioning remained continuously active. Consequently, the generated rules and corresponding performance results should not be assumed to generalize directly to other seasons, climates, or building configurations. In environments with reduced or reversed temperature gradients, additional contextual information or adaptive thresholds may be required.

Fifth, the synthetic dataset comprised independently generated samples and did not model temporal relationships between measurements, whereas Measurements 1 and 2 were real-world time series. Consequently, the synthetic evaluation cannot fully capture temporal effects such as thermal lag, transition periods, or delayed sensor responses observed in operational deployments.

Finally, the real-world evaluation was limited to a small number of measurements and sensing configurations, and a single binary classification task (window-state determination). Although two different sensor configurations were evaluated, broader cross-device, cross-building, and cross-application generalization remains unverified. Additional deployments across multiple buildings, room types, seasons, climates, sensing tasks, and sensor combinations would be required to assess the generality of the findings and strengthen their external validity.

### 4.4. Future work

Future research could evaluate the proposed workflow using additional edge-deployable LLMs, alternative prompting strategies, and more diverse sensing environments. Multi-site studies covering different seasons, sensor configurations, room characteristics, and building types would help assess the generality of the approach. Another research direction is the incorporation of time-series anomaly detection into the analysis pipeline. It would also be valuable to investigate whether iterative human-in-the-loop or automated refinement techniques can further improve LLM-generated deployment logic while preserving their interpretability and low computational cost. Finally, comparative studies involving newer local LLMs and different edge-computing platforms could help determine how broadly the observed design pattern applies.

## 5. Conclusions

This study investigated two alternative roles for locally deployed large language models (LLMs) in an edge-IoT workflow for window-state determination: direct runtime inference and offline code generation. The motivation was the limited evidence regarding the practical use of local LLMs in resource-constrained edge environments.

The research questions introduced in Section 1 can now be answered as follows:

**RQ1. Are freely usable local LLMs practical for direct real-time window-state inference on commodity hardware?**

Answer: **Under the tested conditions, no.** Exploratory feasibility testing showed that none of the evaluated models simultaneously satisfied the requirements for output-format compliance, prediction quality, and practical response time.

**RQ2. Can local LLMs instead be used effectively to generate algorithms for this task?**

Answer: **Qualified yes within the evaluated conditions.** GPT-OSS generated executable Python decision logic that was integrated into the operational workflow and formed the basis of the human-refined deployment implementation. However, because the original generated implementation was evaluated only during a closed-window episode, the study does not demonstrate that the generated code itself performed the complete open-versus-closed window-state detection task.

**RQ3. Does human refinement improve AI-generated algorithms?**

Answer: **Yes, within the evaluated conditions.** Human refinement of the GPT-OSS-generated deployment implementation suppressed the observed sample-level false-positive mode in Measurement 1, reducing 173 false-positive detections to zero. Measurement 2 provides proxy-agreement evidence for the human-refined deployment-rule family but does not support a direct before-and-after comparison. Separately, adding a temperature-difference clause to the author-derived analytical formulation increased agreement with the generated synthetic reference labels, but this synthetic comparison does not evaluate refinement of GPT-OSS-generated code.

In addition to answering the research questions, **the study produced four practical takeaways for the deployment of local LLMs in resource-constrained edge-IoT systems:**

1. Runtime LLM reasoning was not practical for the evaluated edge-IoT deployment.

2. GPT-OSS generated executable deployment code that was integrated into the operational workflow.
3. AI-generated code should be treated as an initial solution requiring validation and review.
4. In the evaluated analyses, simple and interpretable rules supported low-cost inference, while human refinement suppressed one observed deployment failure mode.

Overall, the findings suggest that local LLMs may provide greater value as offline engineering assistants than as runtime reasoning engines in resource-constrained edge environments. In the evaluated workflow, GPT-OSS generated executable deployment logic that was subsequently inspected, refined, and deployed as deterministic, auditable code. This result supports a qualified design pattern in which LLM-assisted code generation complements rather than replaces conventional engineering, with provenance tracking, domain expertise, and dataset-specific validation remaining essential.

## Abbreviations

AI – Artificial Intelligence

HTTP – Hypertext Transfer Protocol

IoT – Internet of Things

JSON – JavaScript Object Notation

LLM – Large Language Model

MQTT – Message Queue Telemetry Transport (historical expansion of MQTT)

TCP – Transmission Control Protocol

TLS – Transport Layer Security

## Acknowledgments

The author acknowledges the contributions of colleagues from multiple teams involved in sensor system development and data collection, which supported the experimental validation presented in this work.

This work was supported by the Chips JU Project LoLiPoP-IoT (Long Life Power Platforms for Internet of Things), www.lolipop-iot.eu, Grant Agreement No. 101112286, jointly funded by the Chips Joint Undertaking and national public authorities.

Generative AI tools, including GPT-5, were used during manuscript preparation for language refinement and editorial support. The author takes full responsibility for the content of this article.

# References


Amangeldy, B., Tasmurzayev, N., Imankulov, T., Bajgarayeva, Z., Izmailov, N., Riza, T., Abdukarimov, A., Mukazhan, M., Zhumagulov, B. (2025). *AI-powered building ecosystems: A narrative mapping review on the integration of digital twins and LLMs for proactive comfort, IEQ, and energy management*, Sensors, **25**(17), 5265. DOI: 10.3390/s25175265.

An, T., Zhu, Y., Zou, H., Yang, J. (2025). *IoT-LLM: Enhancing real-world IoT task reasoning with large language models*, preprint, available at https://arxiv.org/abs/2410.02429.

Arslan, M., Munawar, S., Mahdijoubi, L., Manu, P. (2025). *Monitoring indoor environmental conditions in office buildings using a sustainable agentic RAG-LLM system*, Energy Build., **347, Part A**, 116276. DOI: 10.1016/j.enbuild.2025.116276.

Astanin, S. (2022). *tabulate, version 0.9.0*, available at https://pypi.org/project/tabulate/.

Dierks, T., Rescorla, E. (2008). *The transport layer security (TLS) protocol version 1.2, RFC 5246*, Internet Engineering Task Force. DOI: 10.17487/RFC5246.

Eclipse Foundation (2024). *Eclipse Paho MQTT Python client library, version 2.1.0*, available at https://pypi.org/project/paho-mqtt/.

Fakhoury, S., Naik, A., Sakkas, G., Chakraborty, S., Lahiri, S. (2024). *LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation*, IEEE Trans. Softw. Eng., **50**(9), 2254–2268. DOI: 10.1109/TSE.2024.3428972.

Fielding, R., Reschke, J. (2014). *Hypertext transfer protocol (HTTP/1.1): Message syntax and routing, RFC 7230*, Internet Engineering Task Force. DOI: 10.17487/RFC7230.

Gemma Team (2025). *Gemma 3 technical report*, preprint, available at https://arxiv.org/abs/2503.19786.

Gjylapi, D., Hyso, A., Denaj, A. (2026). *Hybrid Physics-Informed Temporal Attention Model with External Physical Constraints for PV Power Forecasting*, Baltic J. Modern Computing, **14**(2), 356–375. DOI: 10.22364/bjmc.2026.14.2.05.

Grafana Labs (2025). *Grafana, version 12.2.1*, available at https://grafana.com/.

Harris, C. R., Millman, J. K., van der Walt, S. J., Gomers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., Del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., Oliphant, T. E. (2020). *Array programming with NumPy*, Nature, **585**, 357–362. DOI: 10.1038/s41586-020-2649-2.

He, L., Kanagaratnam, S., Zanandreis, L. A., Deng, M. (2025). *Driven robotic indoor air quality monitoring: Advances in natural language interaction and decision-making*, ICRA Construction Robotics Workshop, available at https://construction-robots.github.io/papers/85.pdf.

Hipp, R. D. (2025). *SQLite*, available at https://www.sqlite.org/index.html.

Hunter, J. D. (2007). *Matplotlib: A 2D graphics environment*, Comput. Sci. Eng., **9**(3), 90–95. DOI: 10.1109/MCSE.2007.55.

IETF (2022). *Transmission control protocol (TCP)*, RFC 9293. DOI: 10.17487/RFC9293.

ITU-T (2019). *Information technology—Open systems interconnection—The directory: Public-key and attribute certificate frameworks*, Rec. ITU-T X.509, available at https://www.itu.int/rec/T-REC-X.509-201910-I.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L. R., Lachaux, M.-A., Stock, P., Le Scao, T., Lavril, T., Wang, T., Lacroix, T., El Sayed, W. (2023). *Mistral 7B*, preprint, available at https://arxiv.org/abs/2310.06825.

Jiang, J., Wang, F., Shen, J., Kim, S., Kim, S. (2024). *A Survey on Large Language Models for Code Generation*, preprint, available at https://arxiv.org/abs/2406.00515.

Kalsi, G. S., Tuomaala, T., Pitkäkangas, V., Pahkasalo, M., Hartikainen, M. (2026). *Long-Life IoT Sensing and AI-Supported Analytics for Smart District Heating: Lessons from the LoLiPoP-IoT Project*, J. Artif. Intell. Data Anal., **1**(2), 1–8. DOI: 10.67927/JAIDataAnalytics/2026(1)112.

Kök, İ., Demirci, O., Özdemir, S. (2024). *When IoT meet LLMs: Applications and challenges*, preprint, available at https://arxiv.org/abs/2411.17722.

Li, Z., Deldari, S., Chen, L., Xue, H., Salim, F. D. (2024). *SensorLLM: Aligning large language models with motion sensors for human activity recognition*, preprint, available at https://arxiv.org/abs/2410.10624.

Light, R. A. (2017). *Mosquitto: Server and client implementation of the MQTT protocol*, J. Open Source Softw., **2**(13), 265. DOI: 10.21105/joss.00265.

McKinney, W. (2010). *Data structures for statistical computing in Python*, in Proc. 9th Python in Science Conf. (28 Jun, 2010), pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

McNemar, Q. (1947). *Note on the sampling error of the difference between correlated proportions or percentages*, Psychometrika, **12**(2), 153–157. DOI: 10.1007/BF02295996.

Ollama (2025a). *Ollama: Local deployment platform for large language models*, available at https://ollama.com.

Ollama (2025b). *Thinking - Ollama*, available at https://docs.ollama.com/capabilities/thinking.

OpenAI (2025). *gpt-oss-120b & gpt-oss-20b Model Card*, preprint, available at https://arxiv.org/abs/2508.10925.

pandas development team (2020). *pandas*, available at https://pandas.pydata.org/. DOI: 10.5281/zenodo.3509134.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, É. (2011). *Scikit-learn: Machine Learning in Python*, J. Mach. Learn. Res., **12**(85), 2825–2830.

Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M., Vrgoč, D. (2016). Foundations of JSON Schema, *Proc. 25th Int. Conf. World Wide Web*, pp. 263–273. DOI: 10.1145/2872427.2883029.

Python Software Foundation (2023). *Python language reference, version 3.10.11*, available at https://www.python.org/downloads/release/python-31011/.

Zhang, Y., Ayush, K., Qiao, S., Heydari, A. A., Narayanswamy, G., Xu, M. A., Metwally, A. A., Xu, S., Garrison, J., Xu, X., Althoff, T., Liu, Y., Kohli, P., Zhan, J., Malhotra, M., Patel, S., Mascolo, C., Liu, X., McDuff, D., Yang, Y. (2025). *SensorLM: Learning the Language of Wearable Sensors*, preprint, available at https://arxiv.org/abs/2506.09108.

# Appendices

The appendices provide additional implementation details, evaluation results, and supporting materials that complement the experiments presented in the main text.

# Appendix A. Runtime-inference configuration

The runtime-inference approach described in Section 2.1 was evaluated using locally deployed LLMs hosted through Ollama. The objective was to determine whether local LLMs could directly infer window state from sensor measurements in real time.

**Model configurations**

Three models were evaluated:
- Mistral:7B
- GPT-OSS:20B
- Gemma3:1B

Unless otherwise stated, default inference parameters were used. GPT-OSS provided three reasoning levels ("low", "medium", and "high"), all of which were explored during feasibility testing. The lowest available reasoning level produced the shortest observed response times and was therefore used in the final runtime-inference experiments.

**Database settings**

Several configurations for the number of retained database entries and analysis time window were evaluated (Table 11). In these settings, db_size denotes the maximum number of retained entries, whereas db_timespan_minutes denotes the maximum time span between the earliest and latest retained measurements.

**Table 11.** Runtime-inference parameters and tested values.

| Parameter | Tested values |
|---|---|
| db_size | 2, 6, 10 |
| db_timespan_minutes | 1, 2, 5 |

The fastest observed response times were obtained using:
- db_size = 2
- db_timespan_minutes = 1

This combination therefore became the preferred configuration.

**Sensor enrichment configuration**

To reduce redundant information in the LLM input, general sensor metadata was moved from individual measurements to the prompt. The following contextual assumptions were supplied as prompt information:

- Room type: typical office
- Air conditioning: enabled
- Temperature unit: Celsius
- Radio technology: Bluetooth Low Energy

The same enrichment strategy was later retained in the offline code-generation workflow.

**Final runtime-inference prompt**

The following prompt was used in the final runtime-inference experiments:

"Can you estimate the window state from this JSON data? Use only temperature (Celsius) readings as indicators. Use only the latest measurement from each sensor. The sensor radios are Bluetooth Low Energy. The room is a typical, air-conditioned office room. Always give the answer in JSON format, with \"window_open\" as key and a boolean as value (true=open, false=closed). In addition, give a one-sentence, plain-text explanation/reasoning for the estimate. If you cannot give an estimate, assume the window state is false (closed)."

**Test dataset**

Runtime inference was evaluated using a small synthetic test setup consisting of two simulated sensors (Table 12):

**Table 12.** Simulated-sensor names and descriptions.

| Sensor | Description |
|---|---|
| Simulated Ruuvi F1F1 | Room-center sensor |
| Simulated Ruuvi F1F2 | Window-side sensor |

The purpose of this configuration was to assess response formatting, prediction quality, and practical latency before proceeding to larger-scale experiments.

# Appendix B. Synthetic reference-label generation

Section 2.3 provides an overview of the synthetic dataset used in the study. This appendix describes the synthetic-label generation process in greater detail.

Synthetic reference labels were assigned using a winter-like thermal model. Each sample consists of two temperature readings:

- **center** (room interior)
- **vent** (near window or vent)

The synthetic reference label (gt) is assigned using the following deterministic rule (Equation 4):

$$\text{open} \Leftrightarrow \left(vent \leq center - \Delta_{\text{open}}\right) \vee \left(\min(center, vent) \leq T_{\text{abs}}\right) \quad (4)$$

where:

- $\Delta_{\text{open}} = 2.0°\text{C}$ (center-to-window temperature-difference threshold)
- $T_{\text{abs}} = 20.0°\text{C}$ (absolute threshold)

This rule captures two plausible indicators of an open window:

1. A strong temperature gradient between center and vent.
2. Extremely low temperature at either sensor.

Noise is added to both readings to mimic sensor variability (details provided in Appendix C). The dataset was stratified to maintain approximately equal proportions of open and closed states.

The hybrid analytical rule and the synthetic-data generator become algebraically equivalent when $\Delta T = \Delta_{\text{open}} = 2.0°\text{C}$ and $T_{\text{abs}} = 20.0°\text{C}$ in both formulations. Consequently, perfect performance in this synthetic setting should be interpreted as a consistency check rather than independent validation of predictive performance.

# Appendix C. Synthetic-data noise model

Noise was introduced during synthetic-data generation to simulate plausible sensor variability rather than extreme anomalies. Therefore, noise levels and parameters were chosen to reflect plausible deviations for indoor IoT sensors.

Noise was explicitly added in the synthetic data generator as follows (Equations 5 and 6):

$$T_{\text{center}}^{\text{noisy}} = T_{\text{center}}^{\text{base}} + \varepsilon_{\text{center}} \quad (5)$$

$$T_{\text{vent}}^{\text{noisy}} = T_{\text{vent}}^{\text{base}} + \varepsilon_{\text{vent}} \quad (6)$$

where

$$\varepsilon_{\text{center}} \sim \text{N}(0, 0.5^2)$$
$$\varepsilon_{\text{vent}} \sim \text{N}(0, 0.5^2)$$

Noise was applied after generating base temperatures for center and vent sensors. The purpose of adding noise was to mimic sensor inaccuracies and environmental fluctuations. The noise terms are Gaussian with zero mean and standard deviation σ = 0.5°C, and they are applied independently to the two sensor values.

## Appendix D. Per-seed evaluation results

Appendix D retains the Python-style variable and column names produced by the evaluation scripts to preserve traceability between the reported tables, source code, and generated CSV files. In these tables, T_abs corresponds to the mathematical notation $T_{\mathrm{abs}}$, and dT corresponds to $\Delta T$. The prefixes p_ and h_ identify results for the primary analytical rule and hybrid analytical rule, respectively. Mathematical notation is standardized in the main text, equations, and figure captions, whereas implementation-oriented identifiers are retained in this appendix and in verbatim code.

This appendix contains evaluation tables for the synthetic dataset used in the study.

What each column represents in the tables is listed below.

**General Columns**

- **seed:** Random seed used for synthetic data generation and train/validation/test split.
- **rule:** Analytical rule evaluated on the synthetic dataset:
    - primary = primary analytical rule corresponding to Equation 1, using the absolute-temperature threshold only;
    - hybrid = hybrid analytical rule corresponding to Equation 2, retaining the absolute-temperature threshold and adding the center-to-window temperature-difference clause.
- **T_abs:** Absolute temperature threshold (°C) used in the rule.
- **dT:** Python-style identifier for the center-to-window temperature-difference threshold $\Delta T$(°C) used by the hybrid analytical rule; blank for the primary analytical rule. The reported dT value for each seed was selected using the validation-set parameter sweep.

**Columns beginning with p_ denote metrics for the primary analytical rule corresponding to Equation 1.**

- **p_tp:** Number of observations for which both the synthetic reference label and the primary analytical rule indicated an open state.
- **p_tn:** Number of observations for which both the synthetic reference label and the primary analytical rule indicated a closed state.

- **p_fp:** Number of observations for which the primary analytical rule indicated an open state while the synthetic reference label indicated a closed state.
- **p_fn:** Number of observations for which the primary analytical rule indicated a closed state while the synthetic reference label indicated an open state.
- **p_precision:** Precision = TP / (TP + FP).
- **p_recall:** Recall = TP / (TP + FN).
- **p_f1:** F1 score = harmonic mean of precision and recall.
- **p_accuracy:** Accuracy = (TP + TN) / total.
- **p_runtime_s:** Mean prediction runtime across three measured repetitions over the 400-observation test partition.
- **p_runtime_std_s:** Standard deviation of prediction runtime across the three measured repetitions.
- **p_peak_traced_mem_mb:** Mean peak Python allocation traced during prediction across the three measured repetitions, in megabytes. This value does not represent total process resident memory.
- **p_peak_traced_mem_std_mb:** Standard deviation of the three peak traced-allocation measurements.

**Columns beginning with h_ denote metrics for the hybrid analytical rule corresponding to Equation 2. The TP, TN, FP, FN, precision, recall, F1, and accuracy categories are defined relative to the same generated synthetic reference labels as the p_ columns.**

The corresponding hybrid-rule columns use the same definitions:

- **h_tp, h_tn, h_fp, h_fn, h_precision, h_recall, h_f1, h_accuracy, h_runtime_s, h_runtime_std_s, h_peak_traced_mem_mb, h_peak_traced_mem_std_mb.**

**McNemar Test Columns**

- **mcnemar_b:** Number of test observations for which the hybrid analytical rule agreed with the synthetic reference label and the primary analytical rule did not.
- **mcnemar_c:** Number of test observations for which the primary analytical rule agreed with the synthetic reference label and the hybrid analytical rule did not.
- **mcnemar_p:** Exact two-sided p-value from McNemar's test comparing the paired correctness outcomes of the primary and hybrid analytical rules on the synthetic test partition.

The tables (Table 13–15) themselves are included below.

**Table 13.** Per-seed test-set evaluation results for the primary analytical rule. Runtime values are means across three measured prediction calls following one unmeasured warm-up. Peak traced memory denotes mean peak Python allocation during prediction.

| seed | rule | T_abs | dT | p_tp | p_tn | p_fp | p_fn | p_ precision | p_ recall | p_ f1 | p_ accuracy | p_runtime _s | p_runtime_std _s | p_peak_traced _mem _mb | p_peak_traced _mem_std _mb |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 42 | primary | 20 | | 151 | 207 | 0 | 42 | 1 | 0.782 | 0.878 | 0.895 | 0.00418 | 7.32566e-05 | 0.0593 | 0.0011 |
| 43 | primary | 20 | | 156 | 200 | 0 | 44 | 1 | 0.78 | 0.876 | 0.89 | 0.00426 | 1.2303e-05 | 0.0586 | 1.3214e-05 |
| 44 | primary | 20 | | 167 | 196 | 0 | 37 | 1 | 0.819 | 0.900 | 0.908 | 0.00422 | 0.0002 | 0.0586 | 1.3214e-05 |
| 45 | primary | 20 | | 172 | 192 | 0 | 36 | 1 | 0.827 | 0.905 | 0.91 | 0.00418 | 0.00016 | 0.0586 | 1.3214e-05 |
| 46 | primary | 20 | | 162 | 203 | 0 | 35 | 1 | 0.822 | 0.903 | 0.912 | 0.00419 | 6.50942e-05 | 0.0586 | 1.3214e-05 |

**Table 14.** Per-seed test-set evaluation results for the hybrid analytical rule. Runtime values are means across three measured prediction calls following one unmeasured warm-up. Peak traced memory denotes mean peak Python allocation during prediction.

| seed | Rule | T_abs | dT | h_tp | h_tn | h_fp | h_fn | h_ precision | h_ recall | h_ f1 | h_ accuracy | h_runtime _s | h_runtime_std _s | h_peak_traced _mem _mb | h_peak_traced _mem_std _mb |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 42 | hybrid | 20 | 2 | 193 | 207 | 0 | 0 | 1 | 1 | 1 | 1 | 0.00439 | 0.00031 | 0.0587 | 2.055e-05 |
| 43 | hybrid | 20 | 2 | 200 | 200 | 0 | 0 | 1 | 1 | 1 | 1 | 0.00422 | 0.00015 | 0.0587 | 1.3214e-05 |
| 44 | hybrid | 20 | 2 | 204 | 196 | 0 | 0 | 1 | 1 | 1 | 1 | 0.0049 | 0.00076 | 0.0587 | 1.3214e-05 |
| 45 | hybrid | 20 | 2 | 208 | 192 | 0 | 0 | 1 | 1 | 1 | 1 | 0.00432 | 0.00014 | 0.0587 | 1.3214e-05 |
| 46 | hybrid | 20 | 2 | 197 | 203 | 0 | 0 | 1 | 1 | 1 | 1 | 0.00425 | 0.00016 | 0.0587 | 1.3214e-05 |

**Table 15.** Per-seed McNemar discordant-pair counts and exact two-sided p-values comparing the primary and hybrid analytical rules on the synthetic test partitions.

| seed | mcnemar_b | mcnemar_c | mcnemar_p |
|---|---|---|---|
| 42 | 42 | 0 | 4.547e-13 |
| 43 | 44 | 0 | 1.137e-13 |
| 44 | 37 | 0 | 1.455e-11 |
| 45 | 36 | 0 | 2.91e-11 |
| 46 | 35 | 0 | 5.821e-11 |

## Appendix E. Parameter-sweep results

This appendix contains the top 10 threshold configurations ($T_{\mathrm{abs}}$, $\Delta T$) for the window-state analyzer, based on F1 score. The first table (Table 16) shows the average across seeds; the second table (Table 17) shows results for the first seed. The synthetic dataset

was used. The optimal configuration selected for evaluation was $T_{\mathrm{abs}}$ = 20°C and $\Delta T$ = 2°C.

The manuscript uses $T_{\mathrm{abs}}$ and $\Delta T$ for the absolute and temperature-difference thresholds, respectively. The corresponding Python identifiers in the evaluation code and raw CSV outputs are T_abs and dT.

**Table 16.** Average parameter-sweep results across all seeds.

| Absolute threshold, $T_{\mathrm{abs}}$(°C) | Difference threshold, $\Delta T$(°C) | Mean validation F1 |
|---|---|---|
| 20.000 | 2.000 | 1.000 |
| 20.000 | 2.500 | 0.951 |
| 20.000 | 1.500 | 0.940 |
| 19.000 | 2.000 | 0.888 |
| 21.000 | 2.000 | 0.874 |
| 19.000 | 1.500 | 0.857 |
| 21.000 | 1.500 | 0.848 |
| 21.000 | 2.500 | 0.844 |
| 19.000 | 2.500 | 0.779 |

**Table 17.** Parameter-sweep results for the first seed.

| Absolute threshold, $T_{\mathrm{abs}}$(°C) | Difference threshold, $\Delta T$(°C) | Validation F1 |
|---|---|---|
| 20.000 | 2.000 | 1.000 |
| 20.000 | 2.500 | 0.949 |
| 20.000 | 1.500 | 0.933 |
| 19.000 | 2.000 | 0.890 |
| 21.000 | 2.000 | 0.882 |
| 19.000 | 1.500 | 0.856 |
| 21.000 | 1.500 | 0.845 |
| 21.000 | 2.500 | 0.844 |
| 19.000 | 2.500 | 0.797 |

# Appendix F. GPT-OSS-generated and human-refined deployment code

Listings 1–2 reproduce the retained GPT-OSS-generated deployment implementation and its human-refined version verbatim, including the original spelling, comments, annotations, and documentation. Listing 1 was generated by GPT-OSS from the truth-table specification in Table 1 and the follow-up prompt described in Section 2.2. Listing 2 was produced by adding the human-specified 2°C minimum center-to-window temperature-difference condition to the retained GPT-OSS-generated implementation. This condition is the only functional difference between the two listings. Both implementations target Python 3.10.11.

**Relationship to the analytical rules.** Listings 1–2 are distinct from the author-derived analytical formulations and synthetic-harness implementations corresponding to Equations 1–2. Equation 1 and predict_primary() constitute the author-derived primary analytical rule and its implementation, while Equation 2 and predict_hybrid() constitute the human-extended hybrid analytical rule and its implementation. These analytical rules were used in the synthetic-data evaluation, whereas the deployment implementations reproduced here were used in the Measurement 1 analysis. Equations 1–2 should therefore not be interpreted as GPT-OSS outputs or as mathematical restatements, code-level representations, or equivalent versions of Listings 1–2. The stored Measurement 2 predictions were generated during operation by the human-refined deployment-rule family.

**Implementation limitations.** The listings are reproduced as retained implementation records rather than as corrected revision-stage code. The exact 15/15 condition appears to reproduce one truth-table example literally and should not be interpreted as a general low-temperature rule. The integer annotations describe the rounded values supplied to the Measurement 1 functions but do not preserve the floating-point precision of the original sensor measurements. The ac_on argument was retained in the function interface but was not referenced in the decision logic and therefore provided no contextual adaptation. Although the Measurement 1 evaluation called both deployment functions with ac_on=0, this value had no effect on the reported predictions. The generated code also retained an explicit season parameter and a summer branch despite the follow-up instruction to assume winter operation and permanently active air conditioning. These interface and prompt-implementation discrepancies are retained in the listings and documented as part of the implementation audit. Finally, Listing 2 retains the original docstring, which does not document the human-added 2°C condition. Collectively, these limitations illustrate literal generalization from the truth-table example, incomplete adherence to the follow-up prompt, and incomplete documentation after human refinement. They also reinforce the need for human inspection of LLM-generated code before deployment.

**Listing 1.** Original GPT-OSS-generated rule.

```
"""
window_state.py
Decision logic for whether a window should be open (1) or closed (0).
The logic was derived from the truth table you supplied.
Typical usage:
  >>> from window_state import window_open
```

```
    >>> window_open(ac_on=1, season="Winter", temp_center=22, temp_vent=18)
    1
    """
    def window_open(ac_on: int, season: str, temp_center: int, temp_vent: int) -> int:
        """
        Return 1 if the window should be open, 0 otherwise.
        Parameters
        ----------
        ac_on : int
            1 if the AC is on, 0 otherwise (currently unused – kept for API compatibility).
        season : str
            "Winter" or "Summer" (case-sensitive, as in the truth table).
        temp_center : int
            Temperature measured at the centre of the room (°C).
        temp_vent : int
            Temperature measured at the vent (°C).
        Returns
        -------
        int
            1 for open, 0 for closed.
        Notes
        -----
        * In Winter: the window is open when the vent temperature is lower than
          the centre temperature, *or* when both temperatures are exactly 15°C.
        * In Summer: the window is always closed.
        The implementation is deliberately simple so it can be easily extended
        (e.g., to handle AC off or other seasons).
        """
        # 1. Summer → closed
        if season == "Summer":
            return 0
        # 2. Winter → evaluate temperature difference
        if season == "Winter":
            # special exception for 15/15
            if temp_center == 15 and temp_vent == 15:
                return 1
            # general rule: open if vent < centre
            return 1 if temp_vent < temp_center else 0
        # 3. Unknown season → default to closed (safe default)
        return 0
```

Listing 2 reproduces the human-refined implementation used in the Measurement 1 analysis. The refinement suppresses an open-window prediction when the window-side temperature is lower than the room-center temperature but their rounded difference is less than 2°C. Apart from the retained 15/15 exception, the refined winter rule therefore predicts open only when the room-center temperature exceeds the window-side temperature by at least 2°C. The original docstring was not updated when this condition was introduced and is preserved here as part of the implementation record.

**Listing 2.** Human-refined rule.

```
"""
window_state.py
Decision logic for whether a window should be open (1) or closed (0).
The logic was derived from the truth table you supplied.
Typical usage:
   >>> from window_state import window_open
   >>> window_open(ac_on=1, season="Winter", temp_center=22, temp_vent=18)
   1
"""
def window_open(ac_on: int, season: str, temp_center: int, temp_vent: int) -> int:
   """
   Return 1 if the window should be open, 0 otherwise.
   Parameters
   ----------
   ac_on : int
      1 if the AC is on, 0 otherwise (currently unused – kept for API compatibility).
   season : str
      "Winter" or "Summer" (case-sensitive, as in the truth table).
   temp_center : int
      Temperature measured at the centre of the room (°C).
   temp_vent : int
      Temperature measured at the vent (°C).
   Returns
   -------
   int
      1 for open, 0 for closed.
   Notes
   -----
   * In Winter: the window is open when the vent temperature is lower than
     the centre temperature, *or* when both temperatures are exactly 15°C.
   * In Summer: the window is always closed.
   The implementation is deliberately simple so it can be easily extended
   (e.g., to handle AC off or other seasons).
   """
   # 1. Summer → closed
   if season == "Summer":
      return 0
   # 2. Winter → evaluate temperature difference
   if season == "Winter":
      # special exception for 15/15
      if temp_center == 15 and temp_vent == 15:
         return 1
      # HUMAN REFINEMENT (close if temp diff too small even if vent < centre)
      if temp_vent < temp_center and temp_center-temp_vent < 2:
         return 0
```

```
    # general rule: open if vent < centre
    return 1 if temp_vent < temp_center else 0
# 3. Unknown season → default to closed (safe default)
return 0
```

## Appendix G. Evaluation workflow and implementation details

Python-style identifiers are retained in this appendix to preserve traceability to the implementation. The identifiers T_abs, dT, and open_delta correspond to $T_{\mathrm{abs}}$, $\Delta T$, and $\Delta_{\mathrm{open}}$, respectively.

**window_state_eval.py**

Purpose:

- synthetic-data generation and reference-label construction
- author-derived primary analytical-rule evaluation
- human-extended hybrid analytical-rule evaluation
- validation-set parameter sweep
- McNemar test on paired synthetic-test outcomes
- confidence-interval calculation
- confusion-matrix generation
- validation F1 heatmap generation
- prediction-runtime and peak traced-allocation benchmarking

**Analytical-rule implementations.** The functions predict_primary() and predict_hybrid() directly implement the decision conditions defined in Equations 1 and 2, respectively, without additional decision clauses. predict_primary() is the author-derived synthetic-harness implementation of the primary analytical rule, whereas predict_hybrid() is the human-extended synthetic-harness implementation of the hybrid analytical rule. Neither function is a retained GPT-OSS output or a code-level representation of the deployment implementations reproduced in Appendix F.

Inputs:

1. center_mean
2. vent_mean

3. noise_std
4. open_delta
5. T_abs
6. dT
7. seed

Outputs:

1. per-seed test metrics for both analytical rules
2. aggregate metrics across seeds
3. selected hybrid-rule parameter values
4. validation F1 heatmap
5. representative test-set confusion matrices
6. McNemar discordant-pair counts and exact p-values
7. prediction-runtime measurements
8. peak Python allocations traced during prediction

---

**measurement1_eval.py**

Purpose:

- original versus human-refined deployment-implementation comparison
- sample-level false-positive analysis
- complete-pipeline runtime benchmarking
- complete-pipeline peak traced-allocation benchmarking
- confusion-matrix generation
- time-series visualization

---

**measurement2_preprocess.py**

Purpose:

- merge window-side temperature observations and stored deployment predictions
- create analysis dataset

---

**measurement2_eval.py**

Purpose:

- derive temperature-based proxy labels from window-side temperature values
- calculate precision / recall / F1
- generate confusion matrix
- create time-series figures

---

**baseline_eval.py**

Purpose:

- logistic regression baseline
- decision tree baseline
- prediction-runtime and peak traced-allocation benchmarking

## Appendix H. Baseline classifier configuration

**Evaluation procedure.** Logistic regression and a depth-3 decision tree were evaluated across five independently generated synthetic datasets using seeds 42–46. Each dataset contained 2,000 observations. For each seed, 60% of the observations were used for model fitting, 20% were reserved to maintain the same partition structure as the analytical-rule evaluation but were not used by the baseline models, and the final 20% formed a 400-observation test partition. Hyperparameters were fixed in advance and no validation-based model selection was performed.

Both classifiers used room-center temperature, window-side temperature, and their difference as input features. Models were fitted once per seed. Prediction runtime and peak traced Python allocation were then measured over three repeated calls to the complete test partition. Model fitting, synthetic-data generation, feature construction, and metric calculation were excluded from these measurements. Performance metrics were calculated from the deterministic predictions produced for the test partition and averaged across the five seeds.

**Input features**

Both baselines used the same three input variables:

- Room-center temperature (center)
- Window-side temperature (vent)
- Temperature difference (center − vent)

**Logistic regression**

Configuration:
- Model: LogisticRegression
- max_iter = 1000
- random_state = seed

All other parameters used library defaults.

**Decision tree**

Configuration:
- Model: DecisionTreeClassifier
- max_depth = 3
- random_state = seed

All other parameters used library defaults.

Received August 11, 2026, accepted November 17, 2026