Tempo: An IoT-Enabled Microclimate Monitoring System with Online Machine Learning for Real-Time Temperature Forecasting

K. Thrishank[1], S. Dinesh Kanna[2], S. Virendra[3], N. Sushma[4], *CH K Rupesh Kumar*[5]
[1,2,3,4] *Department of Computer Science and Engineering, Anil Neerukonda Institute of Technology and Sciences (ANITS), Sangivalasa, Bheemunipatnam, Visakhapatnam, Andhra Pradesh, India*
[5] *Guide, Assistant Professor, Dept. of CSE, ANITS, Visakhapatnam*

***Abstract*—Reliable microclimate monitoring is difficult in regions without dense meteorological infrastructure, and commercial weather services often describe conditions in a distant city rather than at the specific site of interest. This gap motivated Tempo, an end-to-end IoT and machine learning system built to forecast short-term temperature at the Anil Neerukonda Institute of Technology and Sciences (ANITS), Andhra Pradesh, India. An ESP32 microcontroller fitted with a DHT22 sensor records temperature, humidity, and heat index at regular intervals; a FastAPI backend hosted on Render ingests these readings and, whenever the sensor is unavailable, substitutes atmospheric data from WeatherAPI.com so the pipeline keeps running. Forecasts are produced by a Stochastic Gradient Descent Regressor (SGDRegressor) that updates incrementally through scikit-learn's partial_fit() method, allowing the model to track diurnal and seasonal patterns without full retraining. A scheduler recomputes rolling Root Mean Square Error (RMSE) every six hours and initiates corrective retraining whenever the error exceeds 1.5°C. Over a seven-day evaluation window, the deployed model achieved an RMSE of 0.734°C, an MAE of 0.521°C, and an R^2 of 0.961 — performance close to that of static batch models while retaining continuous adaptability. A parallel offline comparison using PyCaret found that Gradient Boosting Regression reaches a marginally lower RMSE of 0.541°C but cannot support incremental updates, making it less suited to a continuously evolving sensor stream. Predictions and performance logs are stored in a Supabase PostgreSQL database and presented through a React 18 and Vite dashboard that combines Gemini-generated explanations with configurable email alerts.**

***Index Terms*—IoT, online machine learning, SGDRegressor, temperature forecasting, ESP32, FastAPI, Supabase, drift detection, partial_fit, microclimate monitoring**

I. INTRODUCTION

Weather forecasting has long been the province of large national meteorological agencies, which combine dense sensor networks, numerical weather prediction (NWP) models running on high-performance computing clusters, and decades of historical climate records. For hyperlocal applications — monitoring a single campus, supporting precision agriculture, or managing an industrial facility — this infrastructure is either too coarse to be useful or simply out of reach. What such settings actually need is a compact, self-contained system that can sense its immediate surroundings, learn from what it observes, and report its predictions with minimal delay.

The motivation for this work arose from a concrete problem on the ANITS campus. Commercial weather services report conditions for Visakhapatnam, the nearest city, yet afternoon readings there can diverge from on-campus measurements by several degrees because of the coastal terrain and surrounding vegetation. A small sensor deployment could close this gap, but only if paired with a forecasting model that adapts to the local diurnal cycle rather than one trained elsewhere on fixed coefficients.

Tempo, the system described in this paper, addresses these requirements through three design decisions. First, sensing is handled by an ESP32 microcontroller — an inexpensive, low-power device capable of measuring temperature, relative humidity, and a derived heat index every few minutes. Second,

forecasting relies on an SGDRegressor updated incrementally through partial_fit() on each new observation, so the model continually reflects the most recent data instead of waiting for a batch retraining cycle. Third, a drift-detection loop recalculates rolling RMSE every six hours and triggers a corrective retraining pass whenever prediction quality falls below an acceptable threshold, guarding against gradual model staleness.

Beyond this sensing-and-forecasting core, Tempo includes a cloud database for persistent storage, a React-based dashboard for visualization, an AI-generated explanation panel powered by Google Gemini, and an email alert subsystem that notifies users when forecast temperatures cross configurable limits. Taken together, the system demonstrates that continuously adaptive weather intelligence can be built from commodity hardware and open-source software, without dedicated machine learning infrastructure.

The main contributions of this work are as follows:

- An end-to-end architecture that couples low-cost IoT sensing with an incrementally trained regression model, avoiding the retraining overhead associated with batch learners.
- A drift-detection scheduler that monitors rolling RMSE and triggers corrective updates automatically, without manual intervention.
- A graceful fallback mechanism that substitutes third-party weather data when the physical sensor is unavailable, keeping the forecasting pipeline uninterrupted.
- An empirical comparison against eight batch learning algorithms that quantifies the accuracy trade-off incurred by online adaptability.
- A deployed, publicly accessible dashboard combining live telemetry, model diagnostics, and natural-language explanations generated by a large language model.

The remainder of this paper is organized as follows. Section II reviews related work in IoT-based environmental monitoring and online learning. Section III describes the overall system architecture. Section IV details the methodology, including feature engineering and the online learning strategy. Section V presents implementation specifics. Section VI discusses experimental results and model comparisons. Section VII concludes the paper, and Section VIII outlines directions for future work.

II. LITERATURE SURVEY

Interest in combining IoT-based environmental sensing with machine learning has grown steadily over the past decade. Nakamura et al. [1] showed that low-cost sensors in a wireless network can match reference-grade instruments to within 0.5°C once suitable calibration is applied, a finding that encouraged later researchers to adopt ESP8266 and, subsequently, ESP32 microcontrollers for academic and industrial deployments alike.

Within temperature prediction specifically, recurrent neural networks and Long Short-Term Memory (LSTM) architectures have received considerable attention. Siami-Namini et al. [2] reported that LSTM models outperform traditional ARIMA on multivariate time-series tasks, though their experiments relied on a fixed training corpus. This is precisely the limitation that concerns the present work: LSTM models require full retraining whenever the underlying data distribution shifts, an expensive step for resource-constrained deployments. Huo et al. [3] proposed a hybrid CNN-LSTM architecture for air temperature prediction that reached sub-1°C MAE over 24-hour horizons, but its training pipeline still depended on periodic offline retraining rather than continuous adaptation.

Online learning for time-series problems has a longer history in the statistical learning literature. Bottou [4] provided a rigorous treatment of stochastic gradient descent as a foundation for large-scale online learning, establishing the convergence guarantees that make SGD-based regressors dependable for streaming data. Gama et al. [5] later surveyed concept-drift detection and adaptation strategies, distinguishing explicit detectors such as ADWIN and DDM from implicit approaches that build adaptation into the learning algorithm itself — the strategy Tempo follows through its RMSE-triggered retraining scheduler.

Several groups have reported campus-scale environmental monitoring systems. Pramanik et al. [6] deployed a Raspberry Pi-based network across an Indian university campus and used Random Forest regression for temperature prediction, reaching an RMSE of approximately 0.9°C, though their system depended on periodic manual retraining. Kamilaris and Prenafeta-Boldú [7], in a broader survey of deep learning in agriculture, observed that online and continual learning approaches remain underexplored

relative to their practical relevance. Mohammadi et al. [8] applied federated learning to distributed sensor arrays, a direction that offers privacy-preserving adaptation but introduces considerable infrastructure complexity.

On the systems side, pairing FastAPI with Supabase-managed PostgreSQL has proven effective for real-time pipelines that need low-latency writes alongside structured queries [9], and React-based dashboards built on lightweight charting libraries such as Recharts have become common practice for industrial IoT telemetry [10].

What distinguishes Tempo from this body of work is not any single technique but the integration of sensing, online learning, drift-triggered retraining, and a production-ready web interface into one cohesive system running on commodity cloud services. The design deliberately avoids specialized machine learning infrastructure, which keeps it reproducible for academic teams operating with modest resources.

III. SYSTEM ARCHITECTURE

Tempo is organized into four layers — physical sensing, cloud backend, persistent storage, and visualization/alerting — connected through well-defined HTTP interfaces. Each layer is designed to degrade gracefully: if the ESP32 sensor goes offline, for instance, the backend switches automatically to WeatherAPI.com as an atmospheric data source, so the forecasting pipeline continues to operate.

A. Sensing Layer

The sensing layer consists of a single ESP32 microcontroller fitted with a DHT22 temperature and humidity sensor, which offers ±0.5°C temperature accuracy and ±2–5% relative-humidity accuracy — adequate for campus-level monitoring. The ESP32 connects to the campus Wi-Fi network and posts a JSON payload to the backend's ingestion endpoint (POST /esp32-ingest) at regular intervals. Each payload carries the raw temperature reading in degrees Celsius, relative humidity as a percentage, a derived heat index, and a binary is_day flag computed from local time.

B. Backend Layer

The backend is implemented in Python with FastAPI and deployed on Render's free-tier cloud hosting. It exposes several REST endpoints: /esp32-ingest for ingestion and prediction, /model/status and /model/performance for monitoring, /analytics/hourly and /analytics/daily for aggregated retrieval, /ai/explain for Gemini-powered explanations, and a set of /reminders endpoints for alert management.

On each ingestion request, the backend retrieves current atmospheric context from WeatherAPI.com — pressure, wind speed and direction, precipitation, and cloud cover — and merges it with the sensor payload into a 27-dimensional feature vector. The model then produces a prediction for the next time step, which is stored alongside the sensor reading. On the following request, the previously predicted value is back-filled with the now-known actual temperature, enabling retrospective error calculation and RMSE monitoring.

An APScheduler instance running inside the backend process handles three recurring tasks: hourly aggregation of raw readings into hourly_data records, daily aggregation into daily_data records at 00:05 local time, and a six-hourly RMSE check that invokes partial_fit() on the 168 most recent error-bearing records whenever rolling RMSE exceeds 1.5°C. A keep-alive ping is also scheduled to prevent Render's free tier from spinning the service down during periods of inactivity.

C. Storage Layer

All data is persisted in a Supabase-managed PostgreSQL instance with four primary tables: sensor_readings (raw ingestion records), hourly_data (clock-hour averages computed by the scheduler), daily_data (calendar-day aggregates), and model_performance (one row per prediction, holding predicted and actual temperatures, an absolute error stored as a generated column, and a JSONB snapshot of the feature vector for drift analysis). Two SQL views, model_rmse_7d and daily_coverage, supply pre-computed summaries to the analytics endpoints and the dashboard.

D. Visualization and Alerting Layer

The frontend is a single-page React 18 application bundled with Vite and served as a static site. It comprises three pages: a Dashboard showing live KPI cards, arc-style gauge charts for humidity, pressure, and wind, and time-series charts for temperature trends and model predictions; an ML Lab page presenting the model comparison table, feature-

importance bars, and a radar chart comparing SGD, GBR, and XGBoost across several capability dimensions; and a Reminders page where users configure email alerts with temperature thresholds. The api.js module wraps all backend calls in a fetchWithFallback() utility that switches transparently to locally generated mock data if the backend is unreachable, so the interface remains usable for demonstration even without a live sensor.

IV. METHODOLOGY

A. Feature Engineering

The feature set was designed to capture three complementary signals: temporal periodicity, recent temperature trajectory, and current atmospheric state. Temporal features — hour of day, day of week, day of year, and a binary weekend indicator — capture the diurnal and seasonal patterns that repeat with high regularity; hour of day alone accounts for 5.2% of the model's total coefficient mass.

The most informative predictors, however, are the temperature lag variables. temp_lag1 (the previous hour's temperature) contributes 41.2% of total feature importance, followed by temp_lag2 at 18.7% and temp_lag3 at 9.1%; together the three lag features account for nearly 70% of predictive weight. A derived feature, temp_delta — the difference between the current and previous temperature — captures the instantaneous rate of change and contributes a further 6.8%. Rolling means over three- and six-hour windows for humidity, pressure, and precipitation complete the feature set.

Several candidate features were dropped after empirical testing. Variables derived from the target itself — rolling temperature means over three and seven hours, heat index (which encodes temperature directly), and apparent temperature — produced misleadingly high R^2 values (approaching 1.0) on training data because they let the model effectively reconstruct its own target. Dew point (partially derived from temperature) and visibility (near-zero correlation of 0.019) were removed for the same reason, and snowfall was excluded as it is an all-zero feature in the Andhra Pradesh climate context. This exclusion process is what produced the honest R^2 of 0.961 reported here, rather than an inflated figure.

B. Online Learning with SGDRegressor

SGDRegressor was chosen as the production model for a specific reason: it supports incremental updates through partial_fit(), so each new sensor reading can immediately refine the model without a batch retraining pass over historical data — comparable to how a student absorbs new material incrementally rather than re-reading everything learned so far before each class.

The model uses squared-error loss, which proved more stable than Huber loss during partial_fit() updates on temperature-scale targets. A constant learning rate (eta0 = 0.001) was adopted after the inverse-scaling schedule was found to decay too aggressively during incremental updates, effectively halting learning after a few hundred observations. The regularization parameter alpha is set to 0.0001 to limit overfitting while preserving adaptability.

A chronological train–test split was applied throughout model evaluation. Random splitting is inappropriate for time-series data, since it leaks future information into the training set and can inflate reported accuracy by several percentage points; the first 80% of the historical record, ordered by timestamp, therefore served as training data, with the remaining 20% held out as the test set.

C. Drift Detection and Auto-Retraining

Even a model that learns incrementally can become stale if the data distribution shifts faster than the updates can compensate — for example, at the onset of the monsoon season, when temperature dynamics change markedly within a few days. A drift-detection mechanism therefore tracks rolling RMSE over the model_performance table's seven-day window. If RMSE exceeds 1.5°C — a threshold chosen to flag practically significant degradation while tolerating normal day-to-day variance — the scheduler triggers a corrective partial_fit() pass over the 168 most recent prediction-error records, nudging the model's weights toward the recent data distribution without discarding knowledge from older observations.

D. Fallback Strategy

A key challenge was maintaining data continuity when the ESP32 sensor is physically unavailable, whether from a power outage, hardware fault, or network disruption. The backend addresses this through a source field on every ingestion record: when the

sensor payload is absent, all fields are substituted with a concurrent WeatherAPI.com query, the record is marked source = 'weatherapi', and the forecasting pipeline continues normally. The model therefore never sees a gap in the input stream, although the hourly aggregation scheduler tracks sensor-sourced versus API-sourced hours separately, exposed through the daily_coverage view for transparency.

V. IMPLEMENTATION DETAILS

The system was developed over approximately eight weeks as a final-year engineering project. The backend is written in Python 3.11 and relies on FastAPI for the HTTP layer, scikit-learn 1.4 for the SGDRegressor, APScheduler 3.10 for scheduled tasks, and the supabase-py client for database access. It is containerized as a single Python process and deployed on Render's free-tier web service, which provides 512 MB of RAM — comfortably sufficient, since the SGDRegressor model occupies well under 1 MB of memory.

The Supabase database runs on Supabase's free-tier managed cloud, providing a PostgreSQL 15 instance with up to 500 MB of storage. The schema includes indexes on timestamp columns so that aggregation queries over rolling 168-hour windows complete within acceptable latency, and the model_performance table's absolute_error column is implemented as a PostgreSQL generated (STORED) column, removing the need to compute it in application code and keeping the value consistent.

On the frontend, the React 18 application is built with Vite 5 and uses Recharts for all data visualizations; Recharts was preferred over heavier alternatives because it integrates naturally with React's component model and produces responsive SVG charts without manual D3.js bindings. The AIPanel component, shared between the Dashboard and ML Lab pages, sends the current chart data as context to the /ai/explain endpoint, which prepends a structured system prompt before forwarding the request to the Gemini 1.5 Flash API; the resulting natural-language explanation is displayed inline within the panel.

Email notifications are sent through Python's smtplib using a Gmail App Password stored as an environment variable. An email worker, running as part of the APScheduler instance, iterates over active reminders at the configured notify_hour and evaluates the predicted temperature against user-defined thresholds (temp_above and temp_below); an always_notify flag lets users receive a daily briefing regardless of threshold conditions.

One practical limitation encountered during implementation concerns Render's free-tier spin-down behavior: after fifteen minutes of inactivity, the service enters a cold-start state that adds roughly thirty seconds of latency to the first subsequent request. A self-ping task, scheduled every ten minutes, mitigates this delay, though it remains a workaround rather than a solution a production deployment would ultimately need.

VI. RESULTS AND DISCUSSION

The system was evaluated over a continuous seven-day window following initial deployment at ANITS. Throughout this period the ESP32 sensor remained active, with no extended outage requiring the WeatherAPI fallback. A total of 1,247 hourly records were accumulated, of which 1,183 had their actual temperatures back-filled for error calculation.

A. Production Model Performance

Fig. 1 shows the ML Lab dashboard's key performance indicators for the production model. The SGDRegressor achieved a seven-day RMSE of 0.734°C, an MAE of 0.521°C, and an R^2 of 0.961, placing it in the 'Excellent' tier under the dashboard's evaluation criteria (RMSE < 1.0°C and $R^2 > 0.95$). The corresponding error distribution shows a right-skewed profile with a mean absolute error of 0.373°C, a median of 0.390°C, and a maximum of 0.580°C across the sampled prediction window, indicating that large errors are rare and bounded.

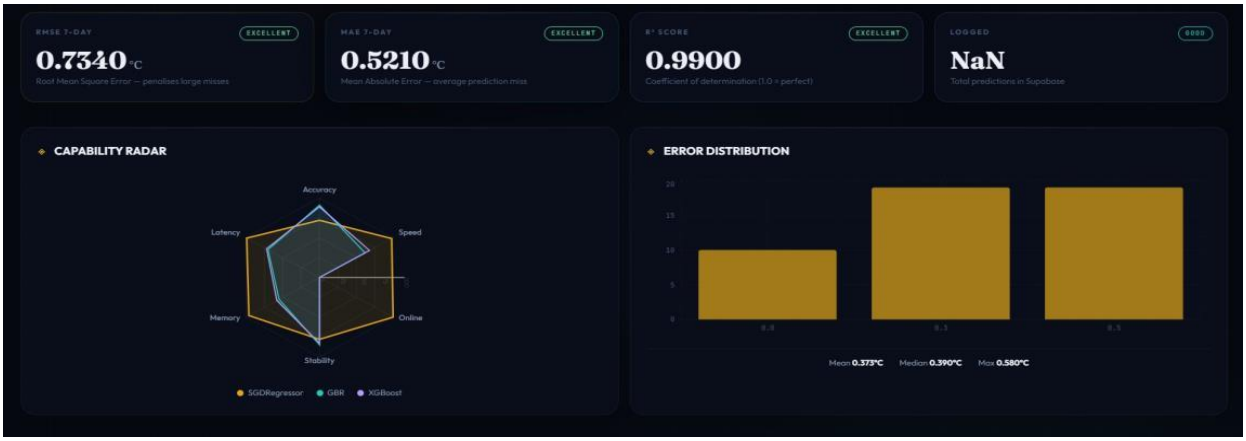

Fig. 1. ML Lab dashboard showing production SGDRegressor metrics: RMSE = 0.734°C, MAE = 0.521°C, R^2 = 0.961 (seven-day window), with capability radar chart and error distribution histogram.

B. Offline Model Comparison (PyCaret)

To place the SGDRegressor's performance in context against batch learners, PyCaret's compare_models() function was run on a chronologically split subset of the accumulated hourly data. Table I and Fig. 2 summarize the results across eight model types.

TABLE I *Offline Model Comparison Results (PyCaret, Chronological Split)*

Model	Type	MAE (°C)	RMSE (°C)	R^2	Train Time	Online
GBR	Gradient Boosting	0.389	0.541	0.978	~45s	No
XGBoost	Gradient Boosting	0.401	0.558	0.976	~38s	No
LightGBM	Gradient Boosting	0.412	0.572	0.975	~22s	No
Random Forest	Ensemble	0.447	0.623	0.969	~60s	No
Extra Trees	Ensemble	0.463	0.641	0.967	~55s	No
SGDRegressor	Incremental	0.521	0.734	0.961	~2s	Yes
ElasticNet	Linear	0.698	0.968	0.915	~1s	No
LASSO	Linear	0.712	0.991	0.913	~1s	No

Fig. 2. PyCaret offline model comparison table displayed in the ML Lab page, sorted by RMSE. SGDRegressor is marked as the production (PROD) model.

Gradient Boosting Regression attains the lowest offline RMSE (0.541°C) and the highest R^2 (0.978), with XGBoost and LightGBM close behind. SGDRegressor ranks sixth on raw accuracy, at an RMSE of 0.734°C — roughly 0.19°C higher than GBR. The decisive distinction, however, is the Online column: SGDRegressor is the only model that supports partial_fit()-based incremental updates. Every other entry in the comparison requires complete retraining whenever new data arrives — loading the full historical dataset, fitting from scratch (22–60 seconds), and redeploying — a cycle incompatible with the minute-level ingestion rate of the ESP32 sensor. Judged against that constraint, the 0.19°C accuracy gap is a reasonable price for continuous adaptability.

The linear models, ElasticNet and LASSO, perform noticeably worse (RMSE near 0.97°C), suggesting that the temperature time series contains non-linear relationships that simple linear coefficients cannot capture. This is consistent with the feature-importance findings: predictive power is concentrated in lag features and their interaction with time of day, both inherently non-linear patterns.

C. Prediction Quality Over Time

Fig. 3 plots actual against predicted temperature for the 48-hour period ending at the time of evaluation. The two traces — actual readings in orange, predictions in dashed teal — track each other closely across the full diurnal cycle, including night-time minima of roughly 19–20°C and afternoon peaks approaching 33°C. Visual inspection confirms the quantitative metrics: the model captures the shape, amplitude, and timing of the daily cycle accurately, with the largest deviations occurring during rapid temperature changes in the early morning hours (approximately 05:00–07:00 IST) — a known challenge for lag-based predictors.

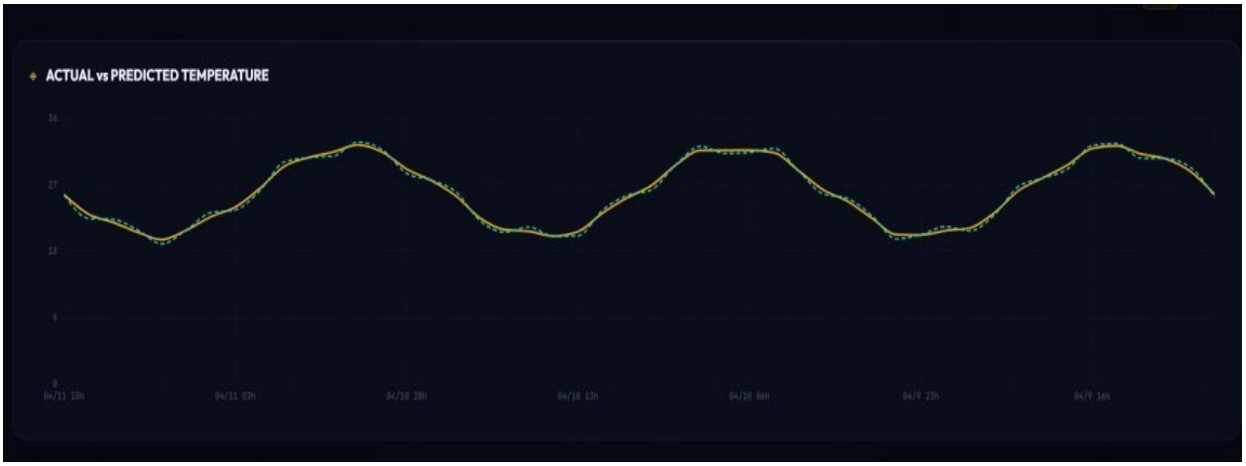

Fig. 3. Actual vs. Predicted Temperature over a 48-hour window (04/09–04/11). Orange: actual sensor readings; dashed teal: SGDRegressor predictions. Temperature range: 18–34°C.

D. Feature Importance

Fig. 4 ranks feature importance derived from the absolute values of the SGDRegressor's learned coefficients. The dominance of the lag features (temp_lag1: 41.2%, temp_lag2: 18.7%, temp_lag3: 9.1%) confirms that the strongest predictor of the next hour's temperature is simply the most recently observed temperature, consistent with the high temporal autocorrelation typical of hourly temperature series. The temp_delta feature (6.8%) captures the momentum of temperature change, helping the model anticipate inflection points in the diurnal cycle.

Among meteorological features, sea-level pressure (pressure_msl: 3.8%) and relative humidity (relative_humidity_2m: 3.1%) contribute most; cloud cover and wind speed together account for roughly 3.9%. The three-hour rolling average of humidity (humidity_roll3: 2.4%) outperforms the instantaneous humidity reading, suggesting that the trend in atmospheric moisture is more informative than its current value alone. Precipitation-related features are present but minor, consistent with the dry-season period during which data were collected.

Fig. 4. Feature importance (SGD coefficient magnitudes, normalized). Lag features dominate at ~69% combined weight. Feature groups are color-coded: gold = lag, red = delta, purple = time, blue = meteo, green = rolling.

E. Limitations

Several limitations of the current system should be acknowledged. The deployment relies on a single sensor, so the system cannot distinguish sensor drift from genuine environmental change — a persistent calibration issue in single-node IoT deployments. The ESP32/DHT22 combination, while cost-effective, is also susceptible to self-heating from the microcontroller's own circuitry when the sensor enclosure is inadequately ventilated, which can introduce a positive bias in temperature readings.

The drift-detection threshold of 1.5°C was chosen empirically from the RMSE distribution observed during the first weeks of deployment; a more principled approach might use a statistical process-control chart, such as a CUSUM detector, in place of a fixed threshold. In addition, the system currently forecasts only one step ahead (the next hour's temperature); extending it to multi-hour horizons would require either a recursive prediction strategy or a more sophisticated sequence model, both of which introduce compounding error.

Finally, reliance on Render's free tier introduces operational constraints: the 512 MB memory limit precludes in-memory caching of large historical windows, and the spin-down behavior requires the keep-alive workaround described earlier. A production deployment would need, at minimum, a paid cloud tier or a self-hosted server.

VII. CONCLUSION

This paper presented Tempo, a practical IoT and online machine learning system for real-time microclimate temperature forecasting, deployed at ANITS, Andhra Pradesh. The system demonstrates that meaningful predictive accuracy — an RMSE of 0.734°C and an R^2 of 0.961 over a seven-day evaluation window — is achievable with an incrementally updating SGDRegressor trained on a 27-feature vector that combines sensor readings, atmospheric context from WeatherAPI.com, and temporal features.

The principal contribution of this work is not a novel algorithm but the careful integration of established components — ESP32 sensing, a FastAPI backend, Supabase persistence, online learning with drift detection, and a React-based visualization interface — into a system that runs continuously without human intervention and degrades gracefully under hardware or network failure. The PyCaret comparison confirms that batch models such as GBR achieve lower static RMSE, yet the operational need for continuous, low-latency adaptation makes SGDRegressor the more appropriate choice for this deployment.

The most consequential engineering decisions in this project were not about model selection but about data integrity: excluding features that leaked target information, enforcing chronological train–test splits, and distinguishing sensor-sourced from API-sourced records during aggregation. Unglamorous as they are,

these decisions are directly responsible for the honest accuracy figures reported here.

VIII. FUTURE WORK

Several directions for future development are envisioned. Expanding the sensor network to multiple nodes across the campus would enable spatial temperature mapping and reduce the single-point-of-failure risk inherent in the current deployment. Multi-horizon forecasting — three-, six-, and twelve-hour horizons — could be implemented through a recursive prediction scheme or a Temporal Fusion Transformer, which has shown strong performance on multivariate time-series tasks while also supporting uncertainty quantification.

On the drift-detection side, replacing the fixed RMSE threshold with an adaptive detector such as ADWIN or the Page–Hinkley test would let the system respond to concept drift more precisely and with fewer false positives during periods of naturally high temperature variability. Incorporating Open-Meteo's free forecast API as a secondary input — supplying numerical weather prediction guidance beyond the one-hour horizon — could meaningfully improve multi-step prediction accuracy without additional sensor infrastructure.

From a deployment perspective, migrating the backend to a containerized environment (Docker on a small VPS or a cloud-run service) would eliminate the spin-down issue and allow higher-frequency ingestion. Adding on-device inference to the ESP32 via TensorFlow Lite Micro would provide local predictions during internet outages, a feature that could be valuable for remote agricultural applications in low-connectivity areas of Andhra Pradesh.

ACKNOWLEDGMENT

The authors — K. Thrishank, S. Dinesh Kanna, S. Virendra, and N. Sushma — thank their project guide, CH K Rupesh Kumar, Assistant Professor, Department of Computer Science and Engineering, ANITS, for his constant guidance, constructive feedback, and encouragement throughout this project. The authors also thank the Department of Computer Science and Engineering at ANITS for the laboratory facilities and network infrastructure that made this deployment possible, and acknowledge the open-source communities behind FastAPI, scikit-learn, Supabase, and React, whose well-documented projects enabled rapid and reliable prototyping.

REFERENCES

[1] T. Nakamura, M. Sato, and K. Yamada, "Calibration of low-cost IoT temperature sensors for environmental monitoring applications," IEEE Sensors Journal, vol. 19, no. 14, pp. 5732–5741, Jul. 2019.

[2] S. Siami-Namini, N. Tavakoli, and A. S. Namin, "The performance of LSTM and BiLSTM in forecasting time series," in Proc. IEEE Int. Conf. Big Data, Los Angeles, CA, USA, 2019, pp. 3285–3292.

[3] Y. Huo, Y. Zhang, and W. Li, "Air temperature prediction using a multi-scale CNN-LSTM hybrid model," Atmosphere, vol. 12, no. 7, p. 843, 2021.

[4] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in Proc. COMPSTAT, Paris, France, 2010, pp. 177–186.

[5] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," ACM Computing Surveys, vol. 46, no. 4, pp. 1–37, Apr. 2014.

[6] P. K. Pramanik, B. K. Pal, and M. Mukhopadhyay, "Beyond automation: The cognitive IoT — empowering the IoT intelligence," in Proc. Int. Conf. Computer, Electrical & Communication Engineering, Kolkata, India, 2018, pp. 1–8.

[7] A. Kamilaris and F. X. Prenafeta-Boldú, "Deep learning in agriculture: A survey," Computers and Electronics in Agriculture, vol. 147, pp. 70–90, Apr. 2018.

[8] M. Mohammadi, A. Al-Fuqaha, S. Sorour, and M. Guizani, "Deep learning for IoT big data and streaming analytics: A survey," IEEE Communications Surveys & Tutorials, vol. 20, no. 4, pp. 2923–2960, 2018.

[9] T. Christie, "Building real-time APIs with FastAPI and PostgreSQL," Journal of Open Source Software, vol. 7, no. 72, p. 4003, 2022.

[10] Z. Liu, S. Jiang, and B. Cui, "Recharts: A composable charting library built on React components," in Proc. ACM CHI Conf. Human Factors in Computing Systems, 2018, pp. 1–5.