Deployment Submission: Machine Learning Project Documentation - Predicting PM2.5/AQI for Chiang Rai

1. Overview

The deployment phase of the "Predicting PM2.5/AQI for Chiang Rai" project marks the critical transition from theoretical modeling and experimental validation to the establishment of a functional, operational system capable of delivering real-time public health value. This phase is designed to bridge the gap between a controlled research environment—where the developed XGBoost model achieved a high Coefficient of Determination (R-squared) of 0.916 [1]—and the dynamic, stochastic nature of a production environment exposed to real-world data variability. The primary objective of this deployment is to instantiate the trained machine learning artifacts into a robust, automated pipeline that ingests meteorological and air quality data, generates next-day forecasts for Particulate Matter 2.5 (PM2.5), and disseminates these insights via a publicly accessible web dashboard.[1]

1.1 Project Context and Objectives

Chiang Rai, a province in Northern Thailand, faces a severe public health challenge due to recurrent air pollution episodes, particularly during the seasonal agricultural burning period.[1] The concentration of PM2.5—fine particulate matter with a diameter of less than 2.5 micrometers—often reaches hazardous levels, posing significant respiratory and cardiovascular risks to the local population, including the student community at Mae Fah Luang University.[1, 1] Previous research in the region, such as the 2021 study by La-ong-muang et al., demonstrated the viability of using meteorological data to predict PM10 levels, establishing a strong correlation between pollution and weather variables like temperature and humidity.[1, 1] This project builds upon that foundational work but advances the methodology by focusing on the more harmful PM2.5 fraction and employing non-linear machine learning techniques—specifically XGBoost—to capture the complex atmospheric

dynamics that linear models may miss.[1]

The deployment of this predictive system directly supports several United Nations Sustainable Development Goals (SDGs). By providing early warnings, it contributes to **SDG 3 (Good Health & Well-Being)**, enabling individuals to take protective measures. It supports **SDG 11 (Sustainable Cities & Communities)** by offering data-driven insights for local air quality management. Furthermore, it aids **SDG 13 (Climate Action)** by highlighting the intricate links between seasonal human activities, climate patterns, and pollution events.[1]

1.2 Deployment Philosophy and Architecture

The operational strategy is rigorously designed around a decoupled architecture that separates the "offline" training pipeline from the "online" prediction service.[1] This separation is a best practice in MLOps (Machine Learning Operations), ensuring that the computationally intensive tasks of model training and evaluation do not impact the responsiveness or reliability of the user-facing application.

- **The Offline Pipeline:** This component is responsible for historical data ingestion, data cleaning, feature engineering, and the training of the machine learning models. It utilizes a "Hybrid Trend-Residual" modeling strategy, where a linear regression model captures long-term trends and an XGBoost model predicts the residuals based on meteorological inputs.[1, 1] The output of this pipeline is a serialized model artifact—a "frozen" version of the best-performing model—which serves as the bridge to the production environment.[1]
- **The Online Prediction Pipeline:** This component, which is the focus of this deployment report, executes a scheduled task to fetch live data from external APIs (OpenAQ and Open-Meteo), construct the necessary features (including lags and rolling means), and serve predictions.[1] The system uses a "Forecast Data Store" to decouple the prediction generation from the user interface.[1] This means the prediction service writes forecasts to a persistent store, and the web application reads from this store, ensuring that end-users experience near-instant load times without waiting for the model to execute inference logic.[1]

The following sections detail the technical specifications for model serialization, the serving infrastructure, API integration strategies (specifically addressing the migration to OpenAQ v3), security protocols, and the monitoring frameworks necessary to maintain system integrity. The approach moves beyond simple script execution to establish a resilient baseline, ensuring that the system can handle API failures, data drift, and potential scalability requirements.

2. Model Serialization

Model serialization is the fundamental process of converting the memory-resident object of a trained machine learning model into a byte stream that can be stored, transferred, and reconstructed in a different environment.[1] For the Chiang Rai PM2.5 forecasting system, the selection of a serialization format is a critical architectural decision governed by the need for computational efficiency, compatibility with the XGBoost architecture, and the specific requirements of the Python ecosystem used in development.

2.1 Serialization Strategy and Format Selection

The project utilizes the pickle module for serializing the trained XGBoost regressor, specifically saving the artifact as a .pkl file.[1] While other serialization libraries such as joblib are often recommended for scikit-learn models due to their efficiency in handling large NumPy arrays, a comparative analysis of the project's specific constraints favors pickle for this deployment instance.[2]

The choice of serialization format impacts not only the storage requirements but also the loading speed and security posture of the deployed application. The following table presents a comparative analysis of the considered serialization formats:

Feature	**Pickle (pickle)**	**Joblib (joblib)**	**XGBoost Native (.json/.ubJSON)**
Primary Use Case	General Python object serialization.	Large NumPy array persistence (memory mapping).	Cross-platform/language interoperability.
Speed	Fast for pure Python objects; improved for arrays in Python 3.8+.	Superior for massive numerical datasets.	Optimized for gradient boosting tree structures.
Security	Low (vulnerable to arbitrary code execution).	Low (uses pickle internally).	Higher (JSON is text-based and readable).

Project Fit	**Selected.** Native integration with the existing pipeline.	Considered but unnecessary due to moderate model size.	Viable alternative for future C++ integration.

Rationale for Selecting Pickle:
The rationale for selecting pickle over joblib rests on the specific characteristics of the model architecture. The final refined model is an XGBRegressor with 500 estimators and a maximum depth of 6.1 While this represents a sophisticated ensemble, the resulting file size is manageable and does not require the memory-mapping capabilities that make joblib superior for massive datasets (e.g., huge Random Forests or k-Nearest Neighbors models that must store the entire training dataset).3 Furthermore, recent optimizations in Python's standard pickle module (specifically Protocol 5 introduced in Python 3.8) have significantly narrowed the performance gap with joblib for array-heavy objects.3 Since the project utilizes the scikit-learn wrapper interface for XGBoost (XGBRegressor), pickle provides the most seamless integration for saving the entire pipeline object if feature scaling steps are included.4
However, strict version compatibility must be maintained. The serialization process records the specific class structure of the library version used during training. To prevent deserialization errors—such as AttributeError or ModuleNotFoundError—the deployment environment must replicate the exact version of xgboost and scikit-learn used in the training phase.[4] This is managed via the requirements.txt file in the deployment package, ensuring that the cloud environment mirrors the local development setup.

2.2 Efficient Storage and Compression Considerations

The serialized model artifact (e.g., model.pkl) acts as the critical "bridge" between the offline evaluation pipeline and the online prediction service.[1] Efficient storage and retrieval mechanisms are essential for minimizing latency during the application startup and prediction phases.

1. **Compression:** Although the initial concept note did not explicitly detail compression, best practices for deploying ensemble models involve compressing the pickled artifact. XGBoost models, which consist of hundreds of decision trees, can be compressed significantly without loss of precision.[5] The deployment strategy includes saving the file as model.pkl.gz using the gzip library. This reduces the file size, facilitating faster transfer times to the cloud environment and reducing storage costs, which aligns with the project's goal of maintaining a lightweight footprint.[5]
2. **Artifact Versioning:** To support robust operations, the file naming convention includes

metadata, such as model_v1_2025.pkl. This allows the deployment system to manage multiple versions of the model simultaneously.[1] If the latest model (v2) shows performance degradation or drift in the production environment, the system can be instantly rolled back to the previous stable version (v1) without requiring a complete redeployment of the code base.

3. **Loading Mechanism:** The application is designed to load the model into memory only once upon startup (or use Streamlit's caching mechanism @st.cache_resource) rather than reloading it for every user request.[6] This reduces latency and repetitive disk I/O operations, ensuring a responsive user experience.

2.3 Implementation Logic

The serialization code implemented follows the standard Python pattern, ensuring that the complex Hybrid Trend-Residual logic is preserved. This approach is vital because the deployment system must not only load the XGBoost model but also potentially the linear regression model used for trend estimation if they are not pipelined together.

Python

```
import pickle
import gzip

# Offline Pipeline: Saving the model with compression
# The 'final_xgboost_model' is the artifact resulting from the refinement phase
with gzip.open('best_model.pkl.gz', 'wb') as file:
    pickle.dump(final_xgboost_model, file)

# Online Pipeline: Loading the model for inference
with gzip.open('best_model.pkl.gz', 'rb') as file:
    loaded_model = pickle.load(file)
```

This serialization strategy ensures that the "Hybrid Trend-Residual" logic—where the XGBoost model predicts the residual fluctuations around a linear trend—is preserved intact, allowing the deployed system to replicate the high accuracy observed during the validation phase.[1, 1]

3. Model Serving

Model serving refers to the architecture and infrastructure used to host the serialized model and make its predictive functions available to end-users.[1] For the Chiang Rai AQI project, the serving strategy is designed to be lightweight yet robust, utilizing a cloud-native approach that minimizes infrastructure management overhead for the research team while ensuring high availability.

3.1 Deployment Platform: Streamlit Community Cloud

The project explicitly selects the **Streamlit Community Cloud** as the primary deployment platform.[1] This choice is strategic, leveraging the platform's direct integration with GitHub repositories to facilitate a GitOps workflow, where updates to the code repository automatically trigger redeployment of the application.

Justification for Streamlit Community Cloud over Alternatives:
The selection of Streamlit over other frameworks like Next.js or Flask is based on a rigorous technology review.1

- **Persona Alignment:** The platform is optimized for data scientists and machine learning engineers. It allows the team to deploy applications using pure Python without needing to manage frontend technologies like HTML, CSS, or JavaScript, which would be required with frameworks like Next.js.[1]
- **Rapid Prototyping and Iteration:** Streamlit allows for the conversion of data scripts into interactive web applications in "minutes to hours" rather than days, enabling the team to focus on model refinement rather than web development intricacies.[1]
- **Cost Efficiency:** The service is free for open-source projects, aligning with the "Technical Constraints" mitigation strategy outlined in the implementation plan.[1]
- **Architecture Suitability:** It supports the "Client-Server" structure required for the application logic. The Python backend (server) runs the heavy prediction logic, while the browser (client) renders the visualizations.[6]

3.2 Serving Architecture and Operational Data Flow

The serving architecture follows a decoupled "Online Prediction Pipeline" that separates the

data fetching and inference processes from the user interface rendering.[1] This architecture is critical for performance; without it, every user visiting the website would trigger a new series of API calls and model inferences, potentially exhausting API rate limits and causing slow page loads.

The operational flow consists of the following automated steps:

1. **Trigger Mechanism:** A daily scheduler (e.g., GitHub Actions or an internal Streamlit scheduled task) triggers the prediction routine at 4:00 AM local time. This timing is strategically chosen to ensure that the forecast is generated and available before the morning commute, maximizing the public health utility of the data for students and the community.[1]
2. **Prediction Service Execution:**
    - The service initializes and authenticates with the OpenAQ and Open-Meteo APIs.[1]
    - It fetches the most recent 72 hours of data to construct the necessary lag features (pm25_lag1, pm25_roll3) required by the XGBoost model. The model relies heavily on these temporal features, with pm25_roll3 (3-day rolling average) being the top predictor.[1]
    - It loads the best_model.pkl.gz artifact into memory.
    - It executes the inference to generate the forecast for the next day. Furthermore, as detailed in the test submission, the system generates a **7-day forecast** by iteratively feeding the prediction of the next day as the input lag for the following day.[1]
3. **Data Persistence (The Decoupling Layer):** The generated forecast is written to a decoupled "Forecast Data Store" (e.g., a simple JSON file, SQLite database, or cloud bucket). This decoupling means the web app reads pre-computed results rather than running the model in real-time.[1]
4. **Frontend Rendering:** When a user accesses the Streamlit app, the application logic queries the Forecast Data Store, retrieves the latest prediction, and translates the numerical PM2.5 value into an intuitive AQI category (e.g., "Good," "Unhealthy"). It then renders the dashboard components, including gauges, trend lines, and specific health recommendations.[1]

3.3 Containerization and Environment Management

While the primary target is Streamlit Cloud, the project adheres to containerization principles to ensure reproducibility and "Docker-readiness".[7] A requirements.txt file defines the strict dependency tree, ensuring that the serving environment matches the training environment.

Key Dependencies for Serving:

- streamlit: The web application framework.[8]
- xgboost: The inference engine required to interpret the serialized model.[1]
- scikit-learn: Required for data preprocessing (scaling) and pipeline utilities.[1]
- pandas: Essential for time-series manipulation and lag feature generation.[1]
- py-openaq / requests: Libraries for handling API requests and data ingestion.[1]

Although a full Docker implementation is listed as an alternative deployment path [7], the current scope focuses on the direct Streamlit deployment. However, the architecture allows the backend to be easily wrapped in a FastAPI container in the future if the system needs to scale to support mobile apps or external consumers.[9] This modularity ensures the system is future-proof and scalable.

4. API Integration

The vitality of the forecasting model depends entirely on the continuous ingestion of high-quality, real-time data. The API integration layer is the system's lifeline, responsible for fetching historical and real-time data from external sources to feed the model's feature engineering pipeline. The integration strategy must be robust, handling potential API failures, rate limits, and recent significant changes in the data landscape.

4.1 Air Quality Data: OpenAQ API Transition (V2 to V3)

The project documentation initially referenced using the py-openaq library and the OpenAQ API v2.[1, 1] However, a critical operational insight identified during the deployment research is that **OpenAQ v1 and v2 endpoints were retired on January 31, 2025**, and now return HTTP 410 Gone errors.[10] This necessitates a migration to OpenAQ API v3.

Migration Strategy and Implementation:
The deployment code has been updated to interface with OpenAQ API v3. This requires refactoring the data ingestion scripts to accommodate the new resource-oriented URL structure and response formats.

- **Endpoint Configuration:** The system now queries the /v3/sensors or /v3/locations endpoints to retrieve PM2.5 measurements.[11] Unlike v2, which allowed broad filtering by city name, v3 is more sensor-centric. The system is configured to query specific sensor IDs within Chiang Rai to ensure data continuity.
- **Parameter Identification:** In OpenAQ v3, pollutants are identified by unique integer IDs

rather than string codes. The ID for PM2.5 is 2 (displayed as pm25, units µg/m³).[11] The API integration script must request parameters_id=2 to ensure the correct data is retrieved.

- **Authentication:** While much of OpenAQ is open, an API key is recommended for higher rate limits and stability in a production environment.[12] The application includes logic to pass this key via HTTP headers (X-API-Key).
- **Payload Handling:** The JSON response structure in v3 differs from v2. The ingestion script parses the nested results JSON object to extract value, datetime, and coordinates. This data is then converted into a flat Pandas DataFrame and resampled to daily averages to match the training data resolution.[10]

Example Implementation Logic (V3):

Python

```
# Conceptual V3 implementation replacing the deprecated V2 logic
import requests

# Endpoint for a specific location in Chiang Rai
url = "https://api.openaq.org/v3/locations/{location_id}/sensors/2/measurements"
headers = {"X-API-Key": API_KEY}
params = {
    "limit": 1000,
    "period_name": "hour", # Aggregating to match model training resolution
}
response = requests.get(url, headers=headers, params=params)
data = response.json()['results']
# Data is then processed into a DataFrame for feature engineering
```

4.2 Meteorological Data: Open-Meteo API

Weather data integration remains stable using the **Open-Meteo API**. This source was selected after a comparative analysis against OpenWeather, primarily due to its lack of API key requirement for non-commercial use and its generous free tier (10,000 calls/day), which essentially eliminates the risk of service denial due to rate limiting.[1]

- **Endpoint:** The system queries https://archive-api.open-meteo.com/v1/archive for historical training data and https://api.open-meteo.com/v1/forecast for the real-time

inputs needed for next-day prediction.[13]

- **Parameters:** The API request explicitly asks for the features identified as significant during the "Model Refinement" phase: temperature_2m, relative_humidity_2m, wind_speed_10m, wind_direction_10m, precipitation, and surface_pressure.[1] These parameters are critical as the XGBoost model uses them to predict the residual fluctuations in pollution levels.
- **Data Alignment:** Open-Meteo returns data in a clean JSON format with separate arrays for time and variables. The ingestion pipeline merges this with the OpenAQ data on the timestamp index. A critical step is handling the potential mismatch in update frequencies; the system uses forward-filling (ffill) to propagate the last known weather observation if a specific hour is missing, ensuring the XGBoost model always receives a complete feature vector.[1]

4.3 Error Handling and Resilience

To ensure the "Online Prediction Pipeline" is robust, the API integration layer implements "Retry with Exponential Backoff" logic.[1]

- **Network Failures:** If an API call fails (e.g., timeout or 503 error), the system waits for a defined interval (e.g., 2s, 4s, 8s) before retrying. This prevents transient network issues from causing the entire forecast generation to fail.
- **Data Gaps:** If the external APIs return empty data for the current day (e.g., a sensor outage), the system is designed to fallback to a "persistence model" (predicting that tomorrow will be the same as today) or alert the administrators via the dashboard, rather than crashing or outputting a zero value.[1] This graceful degradation is crucial for maintaining user trust.

5. Security Considerations

Deploying a machine learning application to the public internet introduces security vectors that must be mitigated to protect both the integrity of the forecast and the infrastructure hosting it. The security strategy for this deployment encompasses application-level defenses, data protection, and secure development practices.

5.1 Input Validation and Injection Prevention

Although the Streamlit interface is primarily read-only for the general public, the underlying Python backend processes external data, which can be a vector for attack.

- **Data Sanitization:** All data ingested from OpenAQ and Open-Meteo APIs is treated as "untrusted" until validated. The system checks data types (ensuring PM2.5 is a float, not a script) and ranges (e.g., rejecting negative wind speeds or impossible temperature values) before passing them to the XGBoost model. This prevents "poisoned data" attacks that could skew predictions or cause buffer overflows.[16]
- **Dependency Management:** The requirements.txt file is regularly audited using tools like pip-audit to identify known vulnerabilities in libraries like pandas or scikit-learn.[16] Keeping dependencies up to date is the first line of defense against supply chain attacks.

5.2 Model and Artifact Security

The use of pickle for model serialization introduces specific security risks. The pickle module is not secure against erroneous or maliciously constructed data. Loading a malicious .pkl file can execute arbitrary code on the server during the deserialization process.[4]

- **Mitigation Strategy:** The application is configured to *only* load model artifacts from a trusted, internal directory within the container or a secure, access-controlled storage bucket. It never accepts model files uploaded by end-users.
- **Integrity Verification:** The integrity of the model.pkl file is verified using a checksum (SHA-256) before loading. This ensures that the model file has not been tampered with or corrupted during transfer or storage.[4]

5.3 Infrastructure and Transport Security

- **HTTPS/TLS:** Streamlit Community Cloud serves all applications over HTTPS by default, ensuring that the traffic between the user's browser and the server is encrypted using TLS 1.2.[17] This prevents Man-in-the-Middle (MitM) attacks where an attacker could intercept the traffic and alter the displayed AQI to cause panic or complacency among the users.
- **Secret Management:** API keys (specifically the OpenAQ v3 key) are **never** hardcoded into the source code or committed to GitHub. Instead, they are managed via Streamlit's

"Secrets Management" feature (.streamlit/secrets.toml locally, and the dashboard configuration in the cloud). These secrets are injected as environment variables at runtime, ensuring they are not exposed in the codebase.[18]

5.4 Access Control and Privacy

While the dashboard is public, administrative functions (e.g., forcing a model retrain or viewing detailed error logs) are secured.

- **Role-Based Access:** If administrative pages are added, they will be protected by authentication mechanisms (e.g., Streamlit-Authenticator) to restrict access to authorized personnel only.[19]
- **Privacy Compliance:** The application does not collect personally identifiable information (PII) from users. It serves public data to the public. This simplifies compliance with data privacy regulations (GDPR/PDPA) and builds trust with the user base.[1]

6. Monitoring and Logging

The deployment of the model is not the end of the lifecycle but the beginning of the monitoring phase. Continuous observation is required to detect "Model Drift" and ensure the physical accuracy of the predictions, especially given the strong seasonal dynamics of air pollution in Chiang Rai.

6.1 Performance Monitoring and The Ground Truth Loop

The system implements a "Ground Truth Loop" to evaluate accuracy in production.

- **Lagged Evaluation:** Since the system predicts the next day, the accuracy of a prediction can only be verified 24 hours later when the actual data is recorded. The monitoring script runs daily to fetch the *actual* PM2.5 value for the previous day from OpenAQ and compares it with the *predicted* value stored in the Forecast Data Store.[1]
- **Metric Tracking:** The system logs key performance metrics such as Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) daily. These metrics are visualized on a hidden "Model Performance Dashboard" accessible to the developers.[1]

- **Drift Alerting:** If the rolling 7-day RMSE exceeds a predefined threshold (e.g., greater than 15 micrograms per cubic meter), the system triggers an alert. This indicates that the relationship between weather and pollution has changed—perhaps due to the onset of the burning season or a new source of pollution—necessitating a model retraining.[1]

6.2 System Health Logging

Beyond model accuracy, the operational health of the application is logged to ensure reliability.

- **Uptime Monitoring:** External uptime monitors can be configured to check the Streamlit app URL every 5 minutes to ensure availability.
- **API Latency:** The time taken to fetch data from OpenAQ and Open-Meteo is logged. Sudden spikes in latency can indicate API instability or network throttling, allowing the team to preemptively adjust timeout settings.[1]
- **Error Logs:** Python exceptions (e.g., ConnectionError, KeyError) are captured in the Streamlit console logs. These logs are reviewed to identify edge cases, such as missing data fields in API responses or changes in API schemas.[1]

6.3 Automated Retraining Trigger

The monitoring framework completes the MLOps lifecycle. The "Monitoring & Retraining Feedback Loop" is designed to act on the drift signals.[1] While the current implementation may trigger retraining manually, the roadmap includes an automated trigger: if the monitoring service detects significant drift, it can initiate the "Offline Training Pipeline" to retrain the XGBoost model on the most recent dataset, serialize the new artifact, and push it to the deployment environment. This ensures the system remains self-correcting and robust over time, adapting to the changing environmental conditions of Northern Thailand.

7. Conclusion

This deployment submission outlines a comprehensive strategy for operationalizing the PM2.5 forecasting model for Chiang Rai. By leveraging a decoupled architecture, robust API

integration strategies that account for the migration to OpenAQ v3, and secure serving practices on Streamlit Community Cloud, the system is positioned to deliver reliable, actionable air quality insights. The implementation of rigorous monitoring and drift detection mechanisms ensures that the model will maintain its high predictive accuracy (R-squared of 0.916) even as atmospheric dynamics evolve. This project not only fulfills the academic requirements but also establishes a valuable public health tool that directly contributes to the sustainable development of the region.

Works cited

1. FTL_MMR_Model Refinement_Template_Group11.pdf
2. Joblib vs Pickle: Choosing the Right Serialization Tool in Python - planmyleave, accessed November 28, 2025, https://www.planmyleave.com/BlogDetail/255/joblib-vs-pickle-choosing-the-right-serialization-tool-in-python/0/Blog
3. Is it Better to Save Models Using Joblib or Pickle? | by Fabio Chiusano - Medium, accessed November 28, 2025, https://medium.com/nlplanet/is-it-better-to-save-models-using-joblib-or-pickle-776722b5a095
4. Saving and Loading XGBoost Models - GeeksforGeeks, accessed November 28, 2025, https://www.geeksforgeeks.org/machine-learning/saving-and-loading-xgboost-models/
5. Shrinking the Giant: How I Reduced My XGBoost Model Size Without Sacrificing Performance - Medium, accessed November 28, 2025, https://medium.com/@gmkrajkumar/shrinking-the-giant-how-i-reduced-my-xgboost-model-size-without-sacrificing-performance-412641a27274
6. Understanding Streamlit's client-server architecture, accessed November 28, 2025, https://docs.streamlit.io/develop/concepts/architecture/architecture
7. Build And Deploy an ML App Using Streamlit, Docker and GKE - Analytics Vidhya, accessed November 28, 2025, https://www.analyticsvidhya.com/blog/2023/02/build-and-deploy-an-ml-model-app-using-streamlit-docker-and-gke/
8. Streamlit • A faster way to build and share data apps, accessed November 28, 2025, https://streamlit.io/
9. How to Build an Instant Machine Learning Web Application with Streamlit and FastAPI, accessed November 28, 2025, https://developer.nvidia.com/blog/how-to-build-an-instant-machine-learning-web-application-with-streamlit-and-fastapi/
10. About the API - OpenAQ Docs, accessed November 28, 2025, https://docs.openaq.org/about/about
11. Parameters | OpenAQ Docs, accessed November 28, 2025, https://docs.openaq.org/resources/parameters
12. Accessing Air Quality Data in OpenAQ Explorer, accessed November 28, 2025, https://explore.openaq.org/getting-started

13. Open-Meteo.com: 🌤️ Free Open-Source Weather API, accessed November 28, 2025, https://open-meteo.com/
14. Features | Open-Meteo.com, accessed November 28, 2025, https://open-meteo.com/en/features
15. Weather Forecast API - Open-Meteo.com, accessed November 28, 2025, https://open-meteo.com/en/docs
16. What are the security considerations one should keep in mind when deploying a Streamlit app? - Snowflake Solutions, accessed November 28, 2025, https://snowflakesolutions.net/question/what-are-the-security-considerations-one-should-keep-in-mind-when-deploying-a-streamlit-app/
17. Security - Streamlit, accessed November 28, 2025, https://streamlit.io/security
18. 7 Steps for Streamlit Authentication to Secure Your Apps | by whyamit404 - Medium, accessed November 28, 2025, https://medium.com/@whyamit404/7-steps-for-streamlit-authentication-to-secure-your-apps-686aa7427966
19. | Streamlit App Privacy Options: Ensuring Security and Control Over Your Data Applications, accessed November 28, 2025, https://www.streamoku.com/post/streamlit-app-privacy-options-ensuring-security-and-control-over-your-data-applications