Deployment Submission

Project Title: Dengue Fever Weekly Risk Alert System for Thailand

Team: Group 8

Member:

1. Sue Sha Htunn (sueshahtunn2002@gmail.com)
2. Mya Moe Wai (myamoewai2002@gmail.com)
3. Shwe Sin Phoo (shwesinphoo2431@gmail.com)
4. Kyaw Soe Lwin (klwin7@my.smccd.edu)
5. Phyo Zay Yar Kyaw (kophyozayarkyaw@gmail.com)

1. Overview

In the deployment phase, I took the dengue risk prediction model that was trained and evaluated in previous steps and prepared it so it can be used consistently outside of a local environment. The main goals were:

- make the final model and preprocessing steps reusable through a single interface
- expose the model both as Python functions (for notebooks / scripts) and as an interactive dashboard
- design a simple API structure that could later be deployed as a web service

To achieve this, I exported the trained models and preprocessing objects to disk, wrapped them in a reusable `DengueRiskPredictor` class (deployment script), and connected that to a Streamlit dashboard for interactive exploration. I also drafted a Flask-style REST API design for future integration into external systems.

2. Model Serialization

The best-performing models from the training phase (Step 7) were:

- XGBoost (primary model for deployment)
- LightGBM
- Random Forest
- Logistic Regression (baseline)

I serialized all trained models using `joblib`, which is well suited for scikit-learn style objects and gradient boosting models. The following artifacts are saved as `.pkl` files:

- `xgboost_model.pkl` – main production model for deployment
- `lightgbm_model.pkl`, `random_forest_model.pkl`, `logistic_regression_model.pkl` – alternative models for comparison / backup
- `feature_scaler.pkl` – `StandardScaler` used to normalize features for linear models
- `province_label_encoder.pkl` – `LabelEncoder` mapping province names to integer codes

Using `joblib.dump` keeps loading time short and preserves all model parameters exactly as trained. The files are stored in the project directory so they can be loaded by both the deployment script and the Streamlit app. In a real production setting, these files could be versioned (for example `xgboost_model_v1.pkl`, `v2.pkl` etc.) and stored in a model registry or object storage, but for this capstone they are kept locally in the repository.

3. Model Serving

To serve predictions, I implemented a reusable Python class:

```
class DengueRiskPredictor:
    ...
```

This class is responsible for:

- loading the serialized XGBoost model, scaler, and province encoder
- recreating the same feature engineering pipeline used in Step 6 (lagged features, rolling means, seasonality features, interactions)
- accepting new input data (current weather and province) and returning a risk category and probabilities

The serving logic is currently used in two main ways:

1. **Programmatic use**
    - The class can be imported in a Python script.
    - Functions like `predict()` and `predict_weekly_risk()` let me pass in a `DataFrame` or a simple `dict` and get back a structured prediction result.
2. **Interactive dashboard (Streamlit)**
    - I built a Streamlit app (`Step 11_Streamlit Dashboard for Dengue Risk.py`) that loads the same serialized model and uses the predictor to generate predictions.
    - The dashboard allows users to select a province, choose a date, and adjust weather inputs (temperature, rainfall, humidity, pressure), then it visualizes predicted risk levels and probabilities.

For this assignment, the model is served on my local machine inside the `capstone-climate` Conda environment. In a future production scenario, the same code could be containerized (for example with Docker) and deployed to a cloud VM or platform service, with the predictor loaded once per process and reused for each request.

4. API Integration

Although I did not fully deploy a live web API, I designed an example REST API endpoint using Flask to show how the model would be integrated into an application. The idea is:

An endpoint `/predict` accepts a JSON payload such as:

```
 {
  "province": "Bangkok",
  "temperature": 31.5,
  "precipitation": 85.0,
  "humidity": 78.0,
  "pressure": 1010.0
}
```

- Inside the endpoint, the server:
    - validates the input
    - calls `DengueRiskPredictor.predict_weekly_risk(province, weather_dict)`
    - returns a JSON response containing:
        - predicted risk category (`"Low"`, `"Medium"`, or `"High"`)
        - probability for each class
        - a simple alert label (`"LOW RISK"`, `"MODERATE ALERT"`, or `"HIGH ALERT"`)

The response structure is designed to be easy for other systems to consume, such as a hospital dashboard, regional public health website, or mobile app. This API design can be implemented directly by plugging the existing deployment class into a Flask (or FastAPI) service when the project is ready for real deployment.

5. Security Considerations

The current project uses **aggregated weekly provincial data**, not individual patient records, which reduces privacy risks. However, if this system were exposed as a real service, I would still apply several security measures:

- **Transport security**
    - Use HTTPS for all API calls to prevent data from being read or modified in transit.

- **Authentication and authorization**
    - Protect the `/predict` endpoint with API keys or tokens so only authorized systems (for example hospitals or health departments) can access it.
    - Store secrets (API keys, database credentials) in environment variables or a secrets manager, not hard-coded in the code.

- **Input validation**
    - Validate province names, numeric ranges (e.g., temperature, rainfall), and data types to prevent malformed or malicious requests.

- **Rate limiting and abuse prevention**
    - Add basic rate limiting to protect the service from being overloaded with requests.

- **Logging without sensitive data**
    - Log only aggregated information (timestamp, province, input features, predicted class, model version) and avoid logging any personally identifiable information if the system is extended.

If the system is deployed inside a hospital or government network, it should also be restricted by network rules (VPN or internal-only access) and integrated with existing authentication systems.

6. Monitoring and Logging

To ensure that the deployed model continues to perform well and remains trustworthy, I planned a basic monitoring and logging approach:

1. **Prediction logging**
    - For each prediction, log:
        - timestamp
        - province
        - input features (weather, search index if used)
        - predicted risk category and probabilities
        - model name and version (e.g., `xgboost_model_v1`)
    - These logs can be saved to a CSV file or database.
2. **Performance monitoring**
    - When real, up-to-date dengue case counts become available, they can be joined with logged predictions to compute:
        - accuracy
        - F1 score, especially for the “High” risk class
        - recall of high-risk weeks (important for public health use)
    - The evaluation plots in Step 8 (confusion matrices, ROC curves, precision–recall curves, temporal and provincial performance) provide a baseline for what “good” performance looks like.
3. **Drift and stability checks**
    - Monitor distributions of key input features (temperature, rainfall, humidity, search index) and predicted probabilities over time.
    - Sudden large shifts could indicate climate anomalies, changes in search behavior, or that the model is no longer well calibrated and might need retraining.
4. **Alerting**
   In a more advanced deployment, monitoring could be connected to an alerting system (for example, sending notifications when model error exceeds a threshold or if high-risk weeks are consistently misclassified).

For the capstone scope, I generated detailed offline evaluation reports and figures and designed the logging strategy so it can be implemented later when integrated with real health data and infrastructure.