# Deployment Submission

## Project Title : Waste Classification System

## Team Members


1. **Khin Yadanar Hlaing** — khinyadanarhlaing.kt@gmail.com
2. **Myo Myat Htun** — myomyatskywalker@gmail.com

## 1. Overview

**Objective:** The objective of this deployment phase was to transition the **"Waste Classification System"** from a local development environment to a production-ready, interactive web application. The goal is to allow users to upload images of waste items and receive real-time classification regarding their recyclability and material type.

**Deployment Steps:** The deployment process involved the following key steps:

1. **Codebase Finalization:** Consolidating the model loading, preprocessing, and inference logic into a single modular script (`app.py`).
2. **Environment Setup:** Defining dependencies (TensorFlow, NumPy, Pillow, Streamlit) in a `requirements.txt` file to ensure the cloud environment matches the local development setup.
3. **Hosting:** Deploying the application on **Streamlit Cloud**, which pulls code directly from the GitHub repository and provisions the necessary server resources.

## 2. Model Serialization

**Serialization Process:** The trained Deep Learning models were serialized to ensure they could be stored on disk and reloaded without retraining. We utilized the **HDF5 (`.h5`)** format, which is the standard mechanism for Keras/TensorFlow models.

**Format Details:**

- **Models Saved:**
    1. `binary_v2.h5`: Used for the binary classification task (Recyclable vs. Non-Recyclable).
    2. `waste_type_model.h5`: Used for multi-class material detection (Glass, Metal, Organic, Paper, Plastic).
- **Efficiency Considerations:** HDF5 allows for the efficient storage of the complete model architecture, weights, and optimizer state in a single file.
- **Git LFS:** Due to the large size of these model files (approx. 180MB), **Git Large File Storage (LFS)** was used to version control and upload them to the GitHub repository, preventing repository size limit errors.

## 3. Model Serving

**Serving Mechanism:** The models are served using the **Streamlit** framework. To ensure high performance and low latency, we utilized Streamlit's caching mechanism:

- **Caching Strategy:** The `@st.cache_resource` decorator was applied to the model loading functions (`load_binary_model` and `load_type_model`). This ensures that the heavy `.h5` files are loaded into memory only once when the app starts, rather than reloading them every time a user uploads a new image.

**Deployment Platform:** We selected **Streamlit Cloud** as the hosting provider.

- **Reasoning:** As a Platform-as-a-Service (PaaS), Streamlit Cloud manages the underlying infrastructure, automatically handles HTTPS security certificates, and offers seamless Continuous Deployment (CD) integration with GitHub.

## 4. Application Interface & Integration

**Integration Strategy:** Instead of a headless API, the models are integrated directly into a Graphical User Interface (GUI) to maximize accessibility for end-users.

**Input & Output Specifications:**

- **Input Handling:**
  - **Widget:** `st.file_uploader` allows users to upload images.
  - **Validation:** Input is strictly limited to `['jpg', 'png', 'jpeg']` formats to prevent errors.
  - **Preprocessing:** A dedicated `process_image()` function resizes user inputs to **224x224 pixels** and normalizes pixel values (scaling by `1/255.0`) to match the format required by the training data.
- **Response Format:**
  - **Visual Feedback:** The app displays the uploaded image alongside the classification results.
  - **Outputs:**
    1. **Status:** Displays "Recycle" (Green/Success) or "Non-Recycle" (Red/Error) with a confidence percentage.
    2. **Material:** Displays the predicted material type (e.g., Plastic, Glass) with a confidence percentage.
  - **Detailed Metrics:** An expandable section shows the raw probability distribution for all classes.

## 5. Security Considerations

To ensure the stability and security of the deployment, the following measures were implemented:

1. **Transport Security:** Streamlit Cloud automatically enforces **HTTPS/TLS encryption**, ensuring that image data uploaded by users is encrypted during transit.

2. **Input Sanitization:** The file uploader widget restricts file types to images only, preventing the execution of malicious scripts or non-image files.
3. **Environment Isolation:** The application runs in a virtualized container environment, isolating dependencies from the host system.
4. **Information Hiding:** TensorFlow logs and warnings were suppressed using `os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'` to prevent internal stack traces or system paths from being exposed to the user in the event of a warning.

## 6. Monitoring and Logging

**Performance Monitoring:** We implemented specific logic to track the health of the application:

- **Startup Checks:** The app includes a logic check at startup (`if binary_model is None...`) to verify that model files are present. If a file is missing, the app halts execution (`st.stop()`) and displays a clear warning instead of crashing unexpectedly.
- **Exception Handling:** `try-except` blocks wrap the model loading process. If a model fails to load (e.g., file corruption), the error is caught and logged to the UI via `st.error`, alerting the administrator immediately.
- **User Feedback:** The `st.spinner('Loading AI Models...')` widget provides visual feedback during the heavy lifting of model loading, ensuring the user knows the system is active and not frozen.