Name – Puspak Chakraborty, Roll No. – DA24S002

Low Level Design: Doodle Classifier System (UI, FastAPI Backend, MLflow Inference, Monitoring)

1. Overview

Implementation-level design for the Doodle Classifier system's operational (non-pipeline) components: the UI, FastAPI backend, MLflow-hosted model inference, and the monitoring stack (Prometheus, Grafana, Windows Exporter).

2. Component Breakdown

2.1 User Interface (UI)

Responsibilities:

- Provide a drawing canvas for users to create doodles.
- Encode drawn images as base64 and send them to the backend for classification or labeling.
- Display classification results and allow users to save images with correct labels if needed.

Implementation Details:

- **Technology:** Likely HTML/JavaScript (see Docker Compose, UI runs in its own container).
- **Port:** 9080 (exposed via Docker Compose).
- **Endpoints Called:**
  - POST /invocations/ (for prediction)
  - POST /save-image/ (for saving labeled images)
- **Data Handling:**
  - Images are flattened (28x28 grayscale, 784 values), base64-encoded, and sent as JSON.

2.2 FastAPI Backend

Responsibilities:

- Expose REST API endpoints for image classification and saving.
- Preprocess and validate incoming image data.
- Forward prediction requests to the MLflow model server.
- Save labeled images to persistent storage.
- Expose Prometheus metrics for monitoring.
- Log all actions and errors.

Implementation Details:

- **Technology:** Python 3.10, FastAPI, Uvicorn, Prometheus client, logging.
- **Port:** 8000 (Docker Compose).
- **Environment Variables:**
  - IMG_DATA_LOCATION (default: ./uploaded_img_data/category)
  - MLFLOW_HOSTED_MODEL_URL (default: http://localhost:5002/invocations/)
- **Endpoints:**
  - POST /save-image/

        - **Input:** JSON with image (base64 string), category (string)
        - **Logic:**
            - Decode base64 to numpy array.
            - Validate length (should be 784 for 28x28).
            - Reshape and save as PNG in category folder.
            - Increment Prometheus counters and histograms for requests/latency.
            - Log all actions and errors.
            - Error handling: returns 400 for invalid input, 500 for server errors.
    - POST /invocations/
        - **Input:** JSON with inputs (base64-encoded image)
        - **Logic:**
            - Forward request to MLflow model server /invocations/.
            - Return model response as JSON.
            - Prometheus metrics for request count and latency.
            - Error handling: logs and returns 500 on failure.
    - GET /metrics
        - **Output:** Prometheus metrics exposition format.
        - **Logic:** Returns all tracked metrics (requests, latency, errors).
- **Metrics Tracked:**
    - model_prediction_requests_total (Counter)
    - model_prediction_latency_seconds (Histogram)
    - save_image_requests_total (Counter)
    - save_image_latency_seconds (Histogram)
    - api_errors_total (Counter, labeled by endpoint)
- **Logging:**
    - Logs to app.log with timestamps and levels.
    - Errors are logged with stack traces.

2.3 MLflow Model Server

Responsibilities:

- Host the latest trained doodle classifier model.
- Expose a REST API for prediction.

Implementation Details:

- **Technology:** MLflow models serve
- **Port:** 5002 (default, see env in backend).
- **Endpoints:**
    - POST /invocations/
        - Accepts JSON
        - Returns predictions as JSON.
- **Input Format:** JSON with key inputs (base64-encoded or flattened image array).

- **Deployment:** Can be run locally or as a service/container. The backend connects via the URL in MLFLOW_HOSTED_MODEL_URL.

2.4 Monitoring Stack

Responsibilities:

- Collect and visualize system and application metrics.
- Provide dashboards for operational monitoring.

Implementation Details:

- **Prometheus:**
  - **Image:** prom/prometheus
  - **Port:** 9090
  - **Config:** prometheus.yml mounts as /etc/prometheus/prometheus.yml
  - **Scrapes:**
    - FastAPI backend /metrics endpoint.
    - Windows Exporter (for host metrics, typically on port 9182).
- **Grafana:**
  - **Image:** grafana/grafana
  - **Port:** 3000
  - **Data Source:** Prometheus
  - **Dashboards:** Visualize API metrics (requests, latency, errors) and system metrics (CPU, RAM, disk, network).
  - **Security:** Admin password set via env.
- **Windows Exporter:**
  - **Image:** Running in localhost
  - **Port:** 9182
  - **Metrics Collected:** CPU, memory, disk IO, network IO.

2.5 Containerization and Orchestration

- **Docker Compose** (docker-compose.yml):
  - Defines services: fastapi-backend, ui, prometheus, grafana.
  - Exposes ports for each service.
  - Sets environment variables for configuration.
  - Mounts volumes for persistent data (uploaded images, Grafana storage).
  - Ensures inter-service communication via Docker networking.
  - MLflow model server is running on host.

3. Data and Control Flow

3.1 Image Classification Flow

1. User draws doodle in UI.
2. UI encodes image as base64, sends to FastAPI /invocations/.
3. FastAPI forwards request to MLflow model server.
4. MLflow model returns prediction; FastAPI returns result to UI.
5. UI displays classification result.

3.2 Image Saving Flow

1. User submits a labeled doodle via UI.
2. UI sends image and category to FastAPI /save-image/.
3. FastAPI decodes, validates, saves PNG in category folder.
4. FastAPI returns confirmation to UI.

3.3 Metrics Flow

1. FastAPI increments counters/histograms for each API call and error.
2. Prometheus scrapes /metrics from FastAPI and metrics from Windows Exporter.
3. Grafana visualizes metrics for operational monitoring.

4. Detailed API & Module Design

4.1 FastAPI Backend (app.py)

- **Classes:**
  - ImageData(BaseModel): For image saving (image, category).
  - ImgDataForPrediction(BaseModel): For prediction (inputs).
- **Functions:**
  - save_image(data: ImageData): Handles image save requests.
  - infer_using_mlflow_hosted_model(data: ImgDataForPrediction): Handles prediction requests.
  - metrics(): Exposes Prometheus metrics.
- **Error Handling:**
  - All endpoints use try/except, log errors, and increment error counters.
  - Returns appropriate HTTP status codes and error messages.
- **Logging:**
  - All actions and errors are logged.

4.2 Prometheus Metrics Integration

- **Metrics Exposed:**
  - Request counts and latencies for both prediction and image saving.
  - Error counts by endpoints.
- **Integration:**
  - Metrics are updated directly in endpoint logic.
  - /metrics endpoint exposes all metrics in Prometheus format.

4.3 MLflow Model Server

- **Input/Output:**
  - Accepts POST requests at /invocations/ with JSON payload.
  - Returns predictions as JSON.
- **Integration:**
  - FastAPI backend sends requests using Python requests library.

5. Deployment and Operations

- **Startup:** All services (except MLflow server and windows exporter) are started via docker-compose up.
- **Monitoring:** Prometheus and Grafana provide real-time monitoring and alerting.

Low Level Design of Pipeline used for retraining

1. Overview

This document details the implementation of the retraining pipeline for the Doodle Classifier system. The pipeline is orchestrated as a sequence of Python scripts, each responsible for a specific stage, with orchestration, monitoring, and error handling. The pipeline ensures that new user data is ingested, processed, merged, versioned, used for retraining, and the resulting model is registered and deployed in a robust, traceable, and reproducible manner.

2. Pipeline Orchestration

2.1 Orchestrator Script: pipeline_orchestrator.py

- **Purpose:** Controls the execution flow of all pipeline stages, manages run state, and records progress in the database.
- **Key Steps:**
    - Initializes the database and creates a new pipeline run.
    - Sequentially executes each stage as a subprocess.
    - Updates stage status (waiting, running, complete, failed, not_triggered) in the pipeline_runs table using db_util.py.
    - If a stage fails, marks current and subsequent stages as failed and halts execution.
    - If insufficient new data is detected in Stage 1, marks subsequent stages as not_triggered and exits.
    - Logs timing and outputs for each stage.
- **Inputs:** None (invoked as a script).
- **Outputs:** Pipeline run and stage statuses in the database, logs for each stage.

3. Database Utilities

3.1 Database Config: db_config.py

- **Purpose:** Stores database credentials (user, password, host, dbname) for PostgreSQL connection.

3.2 Database Utility: db_util.py

- **Functions:**
    - get_db_connection(): Returns a PostgreSQL connection.
    - initialize_database(): Creates the pipeline_runs table if it does not exist.
    - create_new_run(): Inserts a new row in pipeline_runs with all stages set to waiting, returns the run_id.
    - update_stage_status(run_id, stage_number, status, start_time=None, end_time=None): Updates the status and timestamps for a stage; if failed, marks all subsequent stages as failed.
    - set_stages_not_triggered(run_id, start_stage, end_stage): Sets stages as not_triggered if pipeline is halted early.
- **Schema:**
    - pipeline_runs table tracks run ID, start time, per-stage status, and timestamps for start/end of each stage.

- **Database Design**
    - **Table:** pipeline_runs
    - **Schema:**
        - run_id (SERIAL PRIMARY KEY)
        - start_time (TIMESTAMP, default CURRENT_TIMESTAMP)
        - For each stage (1-7):
            - stage_X_status (VARCHAR, e.g., 'waiting', 'running', 'complete', 'failed', 'not_triggered')
            - stage_X_start (TIMESTAMP)
            - stage_X_end (TIMESTAMP)
        - **Schema Management:**
        - Created and managed via db_util.py (initialize_database()).

4. Pipeline Stages

4.1 Stage 1: Scan for New Data (scan_file_count.py)

- **Purpose:** Counts the total number of new images available for retraining.
- **Logic:**
    - Iterates over all category folders in img_data_location.
    - Counts .png files in each category.
    - Logs and prints the total count (used by orchestrator to decide if pipeline should proceed).
- **Error Handling:** Logs exceptions and prints "failure" on error.

4.2 Stage 2: Merge New Data (move_preexisting_new_data_to_old_data.py)

- **Purpose:** Merges new .npy files into the main dataset, ensuring no data is lost.
- **Logic:**
    - Lists .npy files in both old_npy_file_path and new_npy_file_path.
    - For each new file:
        - If not present in old data, copies it over.
        - If present, loads both arrays, stacks them, and saves the merged array.
        - Deletes the new file after merging.
    - Logs actions and prints "success" on completion.
- **Error Handling:** Logs exceptions and prints "failure" on error.

4.3 Stage 3: Convert Images to NPY (convert_image_to_npy_delete_png.py)

- **Purpose:** Converts all new PNG images to .npy arrays for efficient loading during training.
- **Logic:**
    - For each category in img_data_location:
        - Loads all .png images, converts to grayscale, resizes to 28x28 if needed.
        - Flattens each image to a 784-element vector.
        - Aggregates all images in the category into a single numpy array.
        - Saves the array as <category>.npy in new_npy_file_path.

        - Deletes the original .png files after conversion.
    - Logs actions and prints "success" on completion.
- **Error Handling:** Logs exceptions and prints "failure" on error.

4.4 Stage 4: Data Versioning (version_data.py)

- **Purpose:** Ensures all data changes are tracked and reproducible using DVC and Git.
- **Logic:**
    - Runs dvc add <folder>, git add <folder>.dvc, and git commit -m "Update tracked data in <folder>".
    - Each command is executed via subprocess.run with error handling.
    - Logs command outputs and errors.
- **Error Handling:** Logs all exceptions and prints "failure" on error.

4.5 Stage 5: Retrain Model (retrain_model.py)

- **Purpose:** Retrains the classifier using the updated dataset.
- **Logic:**
    - Loads class names from categories.txt.
    - Loads base and new .npy files, samples data from each class, combines old and new data.
    - Splits data into training and validation sets.
    - Loads the previous best ResNet50 model, replaces the final layer for the correct number of classes.
    - Freezes all layers except the final layer.
    - Trains for a fixed number of epochs, logging training and validation metrics to MLflow.
    - Saves the best model checkpoint based on validation accuracy.
- **Error Handling:** Logs exceptions and prints "failure" on error.

4.6 Stage 6: Register Latest Model (register_latest_model.py)

- **Purpose:** Registers the newly trained model with MLflow, tags it for deployment.
- **Logic:**
    - Loads class names.
    - Defines a CustomModel class wrapping the trained PyTorch model, with preprocessing and prediction logic.
    - Logs the model to MLflow using mlflow.pyfunc.log_model.
    - Registers the model and assigns the deployment tag using the MLflow client.
    - Logs actions and prints run/model info.
- **Error Handling:** Logs exceptions and prints "failure" on error.

4.7 Stage 7: Data Hygiene (clean_empty_images_folders.py)

- **Purpose:** Removes empty category folders from the image data directory.
- **Logic:**
    - Iterates through all subfolders in img_data_location.
    - Deletes any folder that is empty.

        - Logs actions and prints "success" on completion.
    - **Error Handling:** Logs exceptions and prints "failure" on error.

5. Logging and Monitoring

- Each stage logs to a dedicated log file (e.g., stage1.log, stage5.log) for debugging and auditability.
- The orchestrator and database utilities ensure all pipeline runs and stage transitions are tracked with timestamps and statuses.
- Failures are logged and reflected in the database, enabling UI/dashboard monitoring.

6. Configuration Management

- All scripts import configuration variables (paths, model names, MLflow URIs, etc.) from a shared config.py.
- Database credentials are managed in db_config.py.
- File and folder paths are parameterized for easy adaptation.

7. Error Handling and Robustness

- Each stage uses try/except blocks to catch and log exceptions.
- The orchestrator halts the pipeline on failure and updates all subsequent stages as failed in the database.
- If insufficient new data is detected in Stage 1, the pipeline marks subsequent stages as not_triggered and exits gracefully.

8. Data and Model Versioning

- Data is versioned using DVC and Git, ensuring every change to training data is tracked and reproducible.
- Models are versioned and registered in MLflow, with deployment tags for serving.

Low Level Design Document: Pipeline Viewer (Dashboard)

1. Overview

The Pipeline Viewer is a web-based dashboard for visualizing and tracking the status of retraining pipeline runs in the Doodle Classifier system. It consists of:

- A **backend REST API** (FastAPI) that fetches pipeline run data from a PostgreSQL database.
- A **frontend HTML dashboard** that displays pipeline run statuses in a tabular format.
- **Database utilities** for managing and querying pipeline run metadata.

2. Backend API Design

Framework: FastAPI

File: fetch_pipeline_runs.py

Port: 8001 (default)

Endpoints

- On /pipeline_runs:
    - Logs the request.
    - Connects to PostgreSQL via get_db_connection() from db_util.py.
    - Fetches all rows from pipeline_runs, ordered by start_time DESC.

- Converts each row to a dictionary with field names.
- Returns as a list of PipelineRun objects (JSON).

Error Handling:

- Returns HTTP 500 if database errors occur.

Logging:

- Logs exceptions to dashboard_backend.log.
- Logs all requests, errors, and returned row counts.

3. Frontend Design

File: index.html

Role: Presents a dashboard table of pipeline runs and their stage statuses.

UI Elements

- Table headers: Run ID, Start Time, Stage 1–7 (status for each).
- Each row: One pipeline run, showing status per stage.
- JavaScript fetches /pipeline_runs from backend and populates the table dynamically.

User Flow

- User opens dashboard in browser.
- Frontend fetches data from backend API.
- Table updates to show latest pipeline runs and statuses every 5 seconds.