Echo Engine – Handover Documentation

Task 1 – MLflow Setup and Evaluation for Model Experiment Tracking (Sprint 1)

Objective

Establish a robust experiment tracking system within the Echo Engine using MLflow and compare it with other platforms such as DVC, Comet ML, and Weights & Biases.

Work Completed

- Set up a Python virtual environment inside the Project Echo directory to isolate the ML experimentation environment.
- Installed and configured MLflow, Comet ML, and Weights & Biases.
- Created and executed a demo script using MLflow that logged model parameters (e.g., max_depth, n_estimators), performance metrics (e.g., accuracy, f1_score), and model artifacts.
- Successfully launched the MLflow UI locally via mlflow ui to visualize experiment runs and validate core tracking functionalities.
- Pushed the MLflow experiment tracking setup to GitHub under the branch engine/feature/mlflow-tracking-setup.
- Conducted comparative analysis between MLflow and DVC for core tracking, registry, and CI/CD applicability.
- Generated exact code snippets to test:
    - Parameter and metric logging
    - Model registry support
    - Deployment hooks
    - Artifact handling and retrieval
- Created a comprehensive evaluation report documenting the performance and capabilities of both frameworks.

Future Improvements

- Integrate MLflow Tracking Server into the Docker Compose environment.
- Connect with remote artifact storage (e.g., S3) to enable collaborative experiments.
- Extend logging to include confusion matrix plots, training loss curves, and input examples.
- Introduce version tags in Git tied to MLflow runs.
- Automate CI/CD tests via GitHub Actions.

Task 2 – Simulate Data Locally in Echo Engine Simulator (Sprint 2)

Objective

Enable local simulation of audio playback inside the Echo Engine container to eliminate reliance on cloud storage and reduce costs.

Work Completed

- Identified limitations of sourcing test audio from cloud, including cost inefficiencies and slow response times.
- Designed a new data simulation pipeline using locally stored audio samples (.wav/.mp3).
- Created simulator/local_audio/ directory for storing test assets.
- Developed simulate_audio.py to scan and play audio files using ffplay, improving playback reliability inside the Docker environment.
- Built a lightweight Dockerfile (python:3.9-slim base) that installs ffmpeg, copies simulation files, and runs the audio simulator.
- Tested the build locally with various sample files; validated proper execution of simulation in container.
- Added optional volume mount command to allow dynamic file swapping without rebuilding:
- docker run -v "$(pwd)/simulator/local_audio:/app/local_audio" local-audio-simulator
- Pushed simulation setup to GitHub under engine/feature/audio-simulator-setup.
- Documented end-to-end setup instructions, PowerShell one-liners, and troubleshooting tips.

Future Improvements

- Add interactive playback controls and randomized or filtered audio playback.
- Implement annotation support and playback logging for auditability.
- Develop a REST API interface to trigger simulation remotely.
- Integrate into the Docker Compose stack with volume sharing.
- Connect playback to ML inference testing and optionally build a GUI panel.