Design Document

Design Principle

The application is designed with modularity, scalability, and loose coupling in mind. The major principles followed include:

Separation of Concerns: The frontend user interface, backend inference engine, and ML model logic are separated into distinct components. Each is responsible for a specific task, improving maintainability and testability.

Loose Coupling: The frontend and backend are completely decoupled and communicate only via well-defined **REST API** calls. This allows independent development, testing, and deployment of each component.

Containerization: All components(frontend, backend, prometheus, grafana, db) are containerized using **Docker**, ensuring consistent environments across development, testing, and production setups.

Configuration over Hardcoding: Parameters such as host URLs, ports, and file paths are configurable via environment variables, allowing easy deployment in various environments.

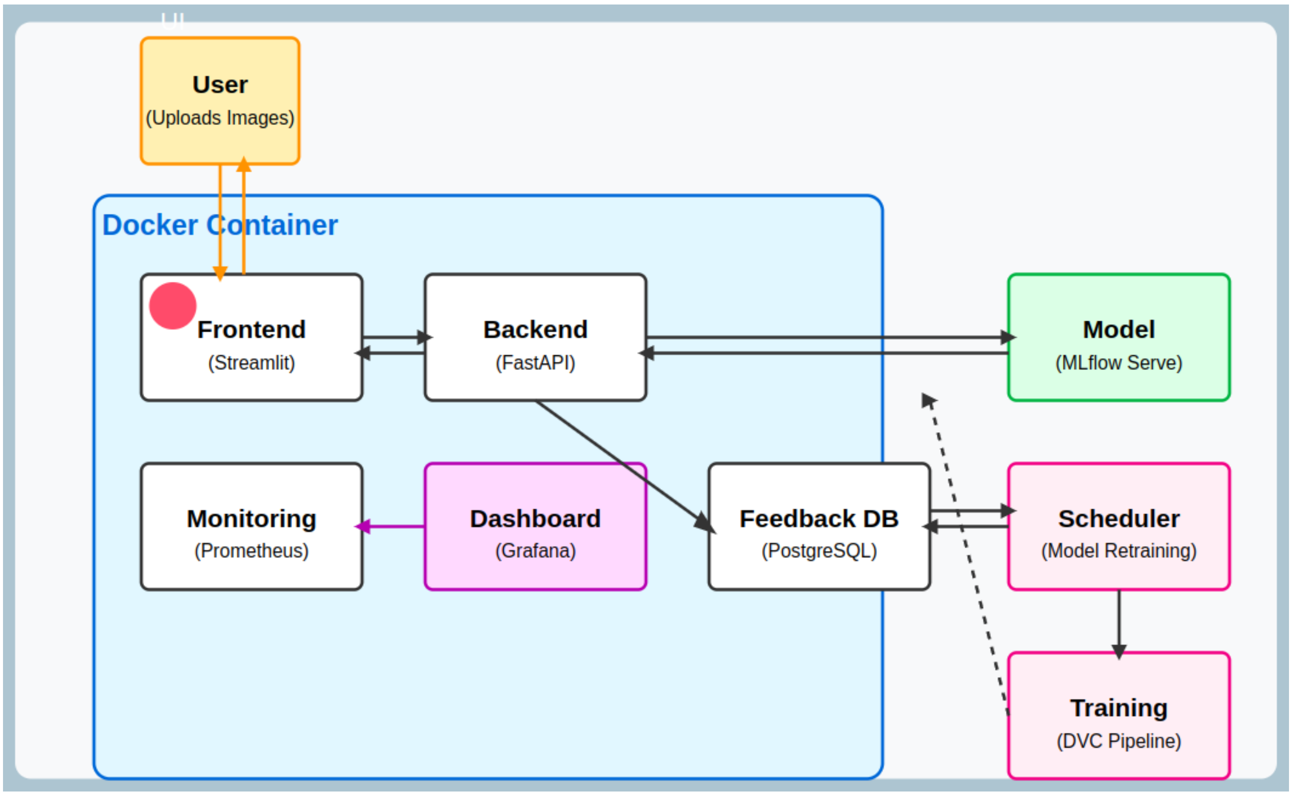
Extensibility: The design allows easy extension of functionality—such as swapping models, enhancing the UI, or adding authentication—without modifying the core logic.

Programming Paradigm

The application predominantly follows the Functional Programming paradigm. Most of the core logic—such as image preprocessing, model inference, and REST API handling—is implemented using pure functions with clear input-output behavior and minimal side effects. This approach promotes readability, testability, and modularity.

However, Object-Oriented Programming (OOP) is also employed in specific parts of the application. For example, the deep learning model and Dataset loading is encapsulated within a class structure to manage loading, inference, and configuration.

High-Level Design Diagram



Components:
- Docker Container

1. Frontend (Streamlit)
    - User interface for image uploads
    - Displays prediction results

2. Backend (FastAPI)
    - Processes requests from the frontend
    - Communicates with MLflow for predictions

3. Monitoring (Prometheus)
    - Tracks system performance and health

4. PostgreSQL Database
    - Stores flagged data (incorrect predictions)
    - Serves as a trigger point for retraining

5. Grafana
    - Creates dashboard using prometheus metrics

- Outside Docker Container

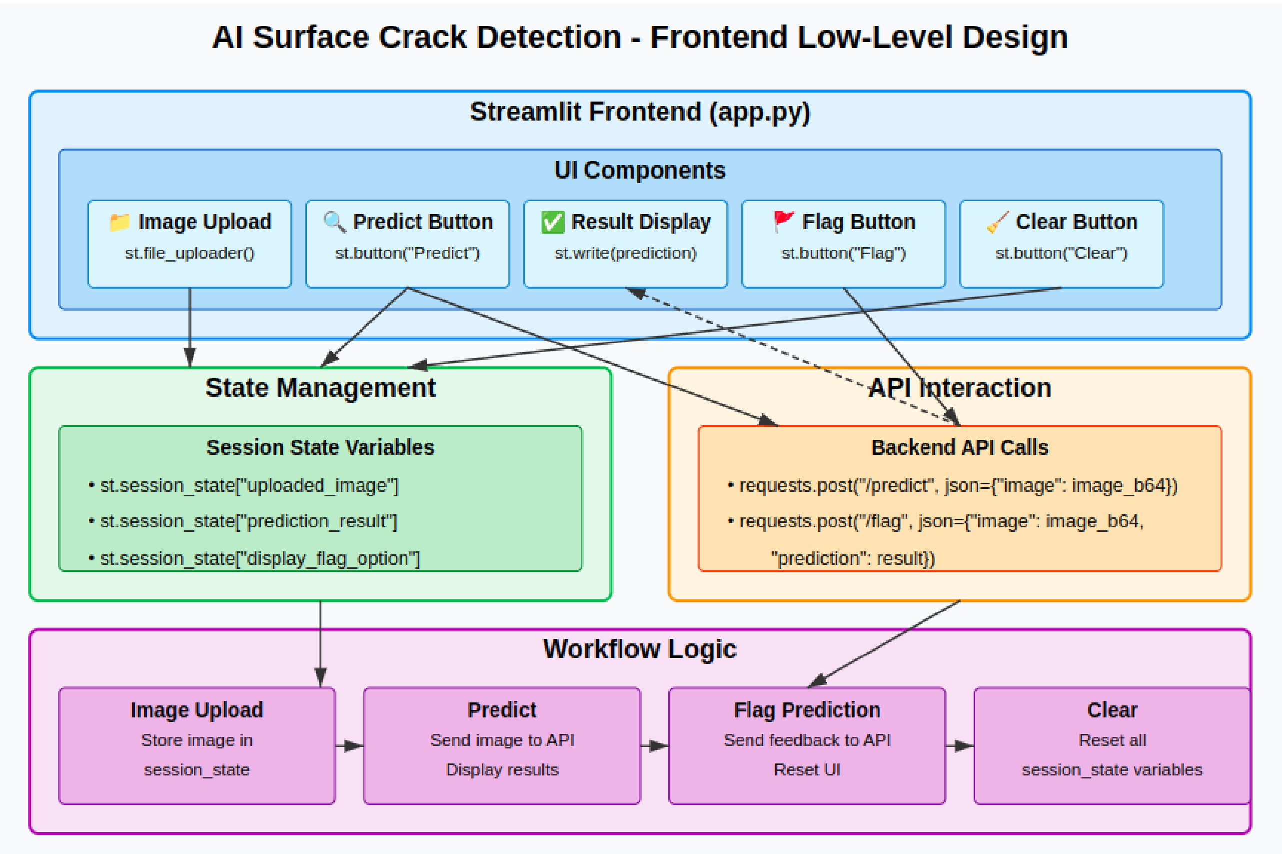
1. MLflow Model Serving
    - Hosts the deployed crack detection model
    - Provides prediction endpoints

2. Scheduler
    - Monitors database for changes
    - Initiates model retraining when needed

3. DVC Training Pipeline
    - Rebuilds models using new flagged data
    - Updates the MLflow served model

Low-Level Design Diagram

Frontend(Streamlit UI)



Dependencies :

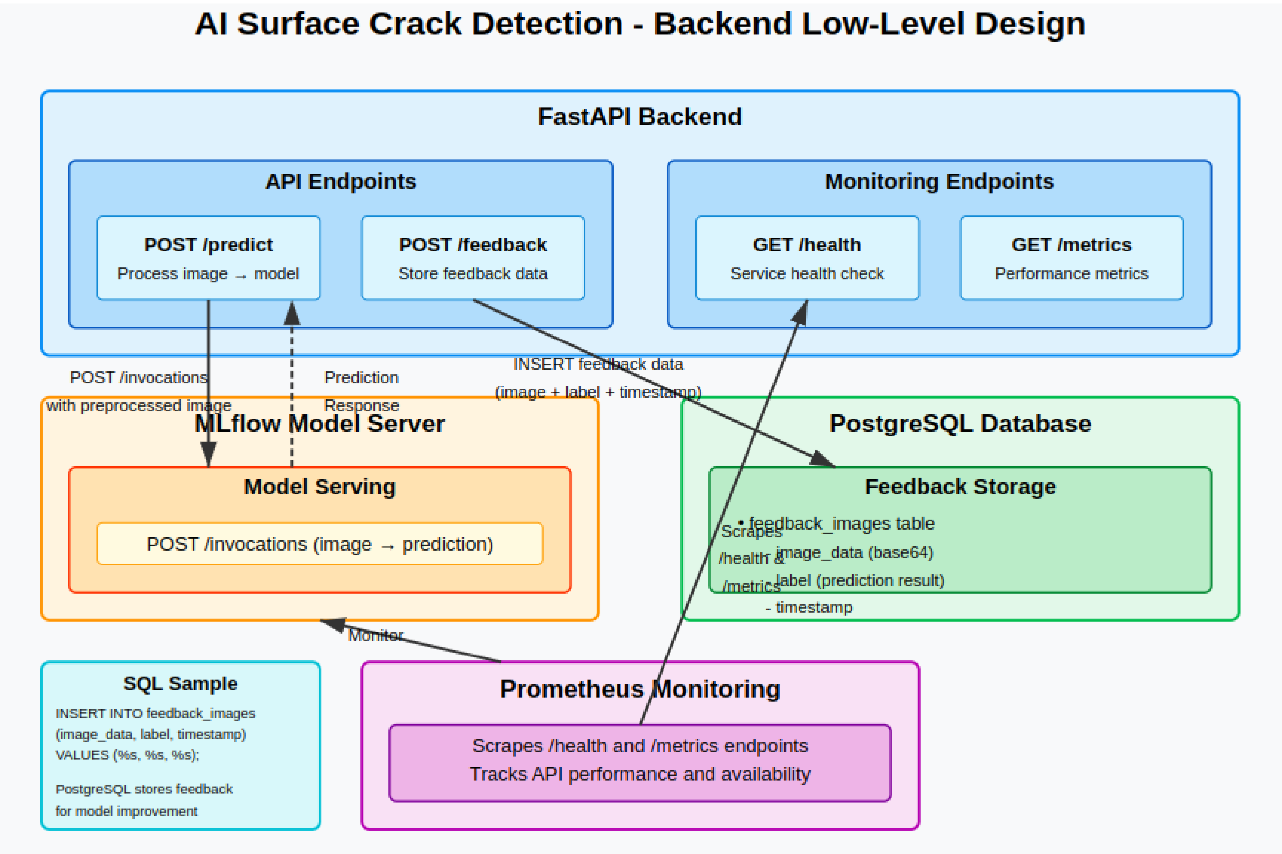
streamlit

requests (to call FastAPI)
base64 (for handling image data)

Backend(Streamlit UI)

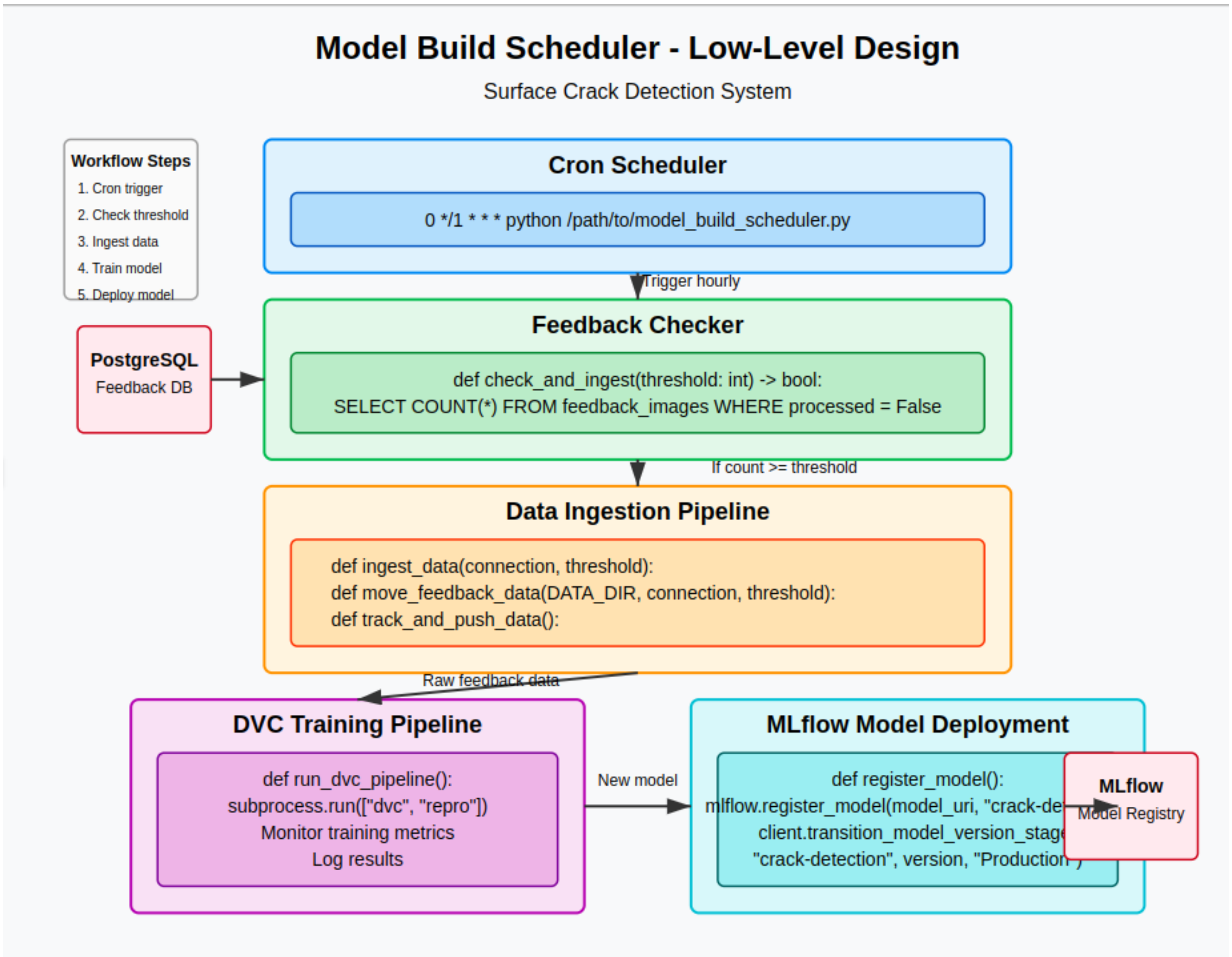
Design Considerations

- Loose coupling: Only communicates with model and DB via FAST API and SQL
- Functional code structure: logic split into route handlers and helper functions
- Handles image encoding/decoding and JSON parsing internally



Dependencies :
FastAPI – for API server
requests – for making calls to model server
psycopg2 / sqlalchemy – for PostgreSQL interaction

Model Retraining Scheduler

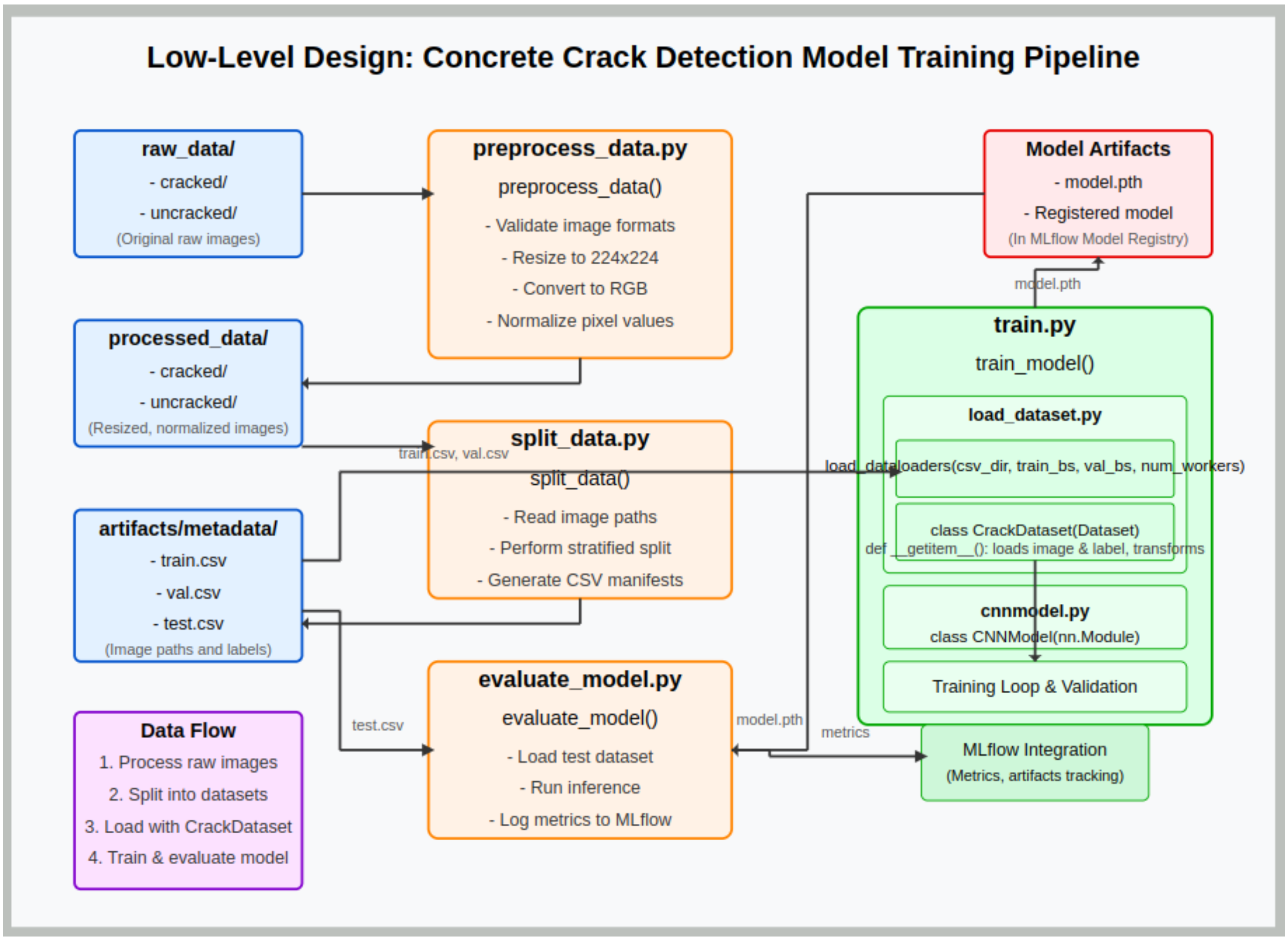


Periodically retrains the crack detection model using:

- Existing training dataset
- Flagged feedback data (from users) Then, logs the new version to MLflow and updates the deployed model

Dependencies:
psycopg2 / sqlalchemy – for PostgreSQL interaction
CronTab – Periodic trigger
DVC – For reproducible training pipelines
MLflow – Model tracking & registration

Model Training Pipeline



raw_data/ --> [preprocess_data] --> processed_data/
processed_data/ --> [split_data] --> train.csv, val.csv, test.csv
train.csv, val.csv --> [train_model] --> model.pth --> MLflow + Registered
test.csv + model.pt --> [evaluate_model] --> test metrics → MLflow