# High-Level Design Document (HLD): Crack Detection System

---

## 1. Objective

This High-Level Design (HLD) outlines the architecture, components, interactions, and system design decisions for a local, containerized MLOps application that performs crack detection in images using a deep learning Attention U-Net model. The system supports real-time inference, user feedback for retraining, version-controlled data and model tracking, and full observability with monitoring dashboards—all running locally via Docker Compose.

---

## 2. Architectural Overview

The system follows a **modular microservices architecture**. It comprises the following interconnected services:

| Component | Technology | Containerized | Port |
|---|---|---|---|
| Frontend | Streamlit | Yes | 8500 |
| Backend | FastAPI | Yes | 8000 |
| Model Training | PyTorch, DVC | Manual/Script | N/A |
| Monitoring | Prometheus, Grafana | Yes | 9090/3000 |
| Experiment Tracking | MLflow | Local Volume | N/A |
| System Metrics | Windows Exporter | Yes/Host | N/A |

All services run locally in isolation within containers, with inter-service communication handled via HTTP and shared Docker volumes.

---

## 3. Component Breakdown

### 3.1 Frontend (Streamlit)

- **Purpose**: Provides a UI for users to upload images, view predictions, and optionally flag unsatisfactory results.
- **Functionality**:
  - Accepts image files via form input.
  - Sends `POST` requests to the FastAPI backend for:
    - `/predict`: to fetch crack segmentation.
    - `/save-for-retrain`: to submit feedback images.
- **Ports & Networking**:
  - Exposed on port **8500**.

- Communicates internally with the backend via Docker bridge network.

---

### 3.2 Backend API (FastAPI)

- **Purpose**: Hosts the trained deep learning model and exposes RESTful API endpoints.
- **Endpoints**:
  - `GET /ping` — service health check.
  - `POST /predict` — inference endpoint.
  - `POST /save-for-retrain` — stores feedback images.
  - `GET /metrics` — exposes Prometheus-formatted metrics.
- **Model**: PyTorch Attention U-Net loaded from `models/` directory.
- **Observability**: Instrumented with `prometheus_fastapi_instrumentator`.
- **Ports & Networking**:
  - Exposed on port **8000**.
  - Mounted volumes:
    - `./models/` — for loading/saving models.
    - `./model_retrain/` — for feedback image storage.
    - `./logs/` and `./mlruns/` — for logging and MLflow output.

---

### 3.3 Retraining Pipeline

- **Execution**: Triggered manually via CLI:
- `python model_retrain/run_retrain_pipeline.py`
- **Functionality**:
  - Detects newly submitted feedback images in `model_retrain_data/images/`.
  - Uses `train.py` from `src/` to retrain model.
  - Logs metrics and models to MLflow.
  - Tracks datasets and models using DVC.
- **Versioning**:
  - DVC remote: `.dvc_storage/`
  - Artifacts tracked: training data, feedback data, model weights.
- **Outputs**:
  - Updated model written to `./models/`
  - New MLflow experiment recorded in `./mlruns/`

---

### 3.4 Monitoring Stack

- **Prometheus**:
  - Scrapes backend metrics from `/metrics` every 5 seconds.
  - Scrapes host metrics via Windows Exporter (disk, CPU, memory, network).
- **Grafana**:
  - Visualizes:
    - API usage (request rate, latency, error codes).

        - System metrics from Windows Exporter.
    - Preconfigured with dashboards located in `monitoring/grafana/dashboards/`.

---

### 3.5 Experiment Management

- **MLflow**:
    - Logs parameters, metrics, and model artifacts.
    - Uses `./mlruns/` volume for storage.
- **DVC**:
    - Tracks datasets, models, and pipeline stages.
    - Uses `.dvc_storage/` as the local DVC remote.

---

# 4. Data Flow

1. **User uploads image** through Streamlit UI.
2. UI sends image via `POST /predict` to FastAPI.
3. Backend returns segmentation mask generated by the PyTorch model.
4. If prediction is unsatisfactory:
    - User submits the image via `POST /save-for-retrain`.
    - Backend stores the image in `model_retrain/model_retrain_data/images/`.
5. Metrics for each request are exposed to Prometheus.
6. Prometheus scrapes and sends metrics to Grafana.
7. Retraining pipeline, when triggered, reads feedback images, retrains model, and updates `models/`.

---

# 5. Deployment and Orchestration

- All services are defined in `docker-compose.yml` and launched with:
- `docker-compose up --build`
- Volumes are mapped for:
    - Model weights
    - Logs
    - Feedback data
    - MLflow runs
- Network:
    - All services are on a shared bridge network `app-network`.

---

# 6. Testing

- Unit and integration tests located in `tests/`:
    - `test_api.py`: Tests backend REST endpoints.
    - `test_metrics.py`: Verifies exposed Prometheus metrics.
    - `test_train.py`: Validates model training logic.
    - `test_save.py`: Ensures feedback saving is functional.
- Executed manually or via CI trigger:
- `python tests/test_*.py`

---

# 7. Security and Robustness

- Input validation for uploaded images in backend.
- Exception handling and logging implemented in both frontend and backend.
- Retry logic (if needed) for Prometheus scrapes.
- Model retraining includes integrity checks for dataset completeness.

---

# 8. Key Design Choices and Rationale

| Decision Area | Choice | Rationale |
|---|---|---|
| Deployment | Docker Compose | Simplifies local orchestration |
| Frontend | Streamlit | Lightweight UI for fast prototyping |
| Backend | FastAPI | High-performance REST API, async support |
| Model Format | PyTorch `.pth` | Easy to load and retrain |
| Feedback Storage | Shared volume + file system | Keeps system local and flexible |
| Observability | Prometheus + Grafana | Industry standard, easy integration |
| Experiment Tracking | MLflow | Lightweight, open-source |
| Versioning | DVC | Git-like reproducibility and traceability |

---

# 9. Folder Structure

```
.
├── backend/                  # FastAPI inference API
├── frontend/                 # Streamlit UI
├── model_retrain/          # Retraining logic and feedback data
├── src/                    # Core ML logic and utilities
├── tests/                  # Test scripts for API and training
├── models/                 # DVC-tracked model artifacts
├── data/                   # Datasets (train/test)
├── .dvc_storage/           # Local remote for DVC
├── mlruns/                 # MLflow experiment runs
├── monitoring/             # Prometheus and Grafana config
└── docker-compose.yml      # Full stack orchestration
```

---

# 10. Next Steps and Recommendations

- Implement auto-trigger for retraining via feedback image count threshold.
- Improve Graphana Dashboards
- Containerize MLflow server and expose via port 5000