Low-Level Design (LLD) Document:

Crack Detection

1. Objective

This Low-Level Design (LLD) document describes the internal components, interfaces, data contracts, and logic flows of the Crack Detection System MLOps pipeline. It is intended for developers, MLOps engineers, and testers to understand the precise structure and behavior of the application modules and how they interact within a containerized local deployment.

---

2. Component-Level Design

2.1 Frontend: Streamlit UI (`frontend/app/streamlit_app.py`)

Purpose: Provides a user interface for image upload, crack prediction visualization, and feedback submission.

Detailed Flow:

1. A user navigates to the Streamlit UI hosted on port 8500.
2. They upload an image (PNG or JPEG) via a file input form.
3. The image is encoded and sent via an HTTP POST request to the `/predict` endpoint of the FastAPI backend.
4. The backend responds with a predicted mask array.
5. The frontend decodes the response and overlays the segmentation result on the original image.
6. If the user marks the prediction as unsatisfactory, the image is sent again via POST to `/save-for-retrain`.

Static Content:

- Hosted under `frontend/app/static/`

Environment Variables:

- `BACKEND_URL` — Used to send requests to FastAPI backend.

Ports:

- 8500 (Streamlit server)

2.2 Backend: FastAPI Inference Server (`backend/app/main.py`)

Purpose: Handles inference requests, saves user feedback, and exposes Prometheus metrics.

Detailed Flow:

- Upon startup, the backend loads the latest PyTorch model using `load_model()`.
- Receives images via `/predict`, converts to tensors, applies preprocessing, and runs inference with Attention U-Net.
- The output mask is post-processed, converted to a binary 2D array, and returned in JSON.
- Feedback images received at `/save-for-retrain` are stored directly to the local directory for later retraining.
- Metrics about request duration, status codes, and latency are exposed at `/metrics` for Prometheus scraping.

Endpoints:

Method	Endpoint	Description	Request Body Type
GET	`/ping`	Health check endpoint	None
POST	`/predict`	Accepts image, returns prediction mask	multipart/form-data
POST	`/save-for-retrain`	Saves user feedback image	multipart/form-data
GET	`/metrics`	Prometheus metrics endpoint	None

Model Architecture:

- Attention U-Net, defined in `src/model.py`.

Monitoring:

- Instrumented with `prometheus_fastapi_instrumentator`.

Ports:

- 8000

2.3 Retraining Pipeline (`model_retrain/run_retrain_pipeline.py`)

Purpose: Retrains the model using user feedback images collected from the UI.

Detailed Flow:

1. Script checks if sufficient feedback images exist in `model_retrain_data/images/`.
2. If yes, training is triggered using a PyTorch pipeline defined in `src/train.py`.
3. Combines both original training data and feedback images.
4. Logs metrics and artifacts to MLflow.

5. Pushes updated dataset and models to DVC for version control.
6. The trained model is saved back to the `models/` folder.

Dependencies:

- PyTorch
- DVC
- MLflow

Output Artifacts:

- `models/attention_unet_<timestamp>.pth`
- `mlruns/` experiment logs

2.4 MLflow

Purpose: Provides a lightweight experiment tracking tool.

Usage Flow:

- Training or retraining runs log hyperparameters, performance metrics, model artifacts, and training duration.
- UI or CLI interface can be used to compare experiments.
- Storage is handled via shared volume `mlruns/`.

Status: Not containerized but accessible via volume.

2.5 DVC

Purpose: Ensures reproducibility by tracking datasets and models.

Usage Flow:

- Data and models are added via `dvc add`.
- Version control is maintained via Git.
- Remote storage is local (`.dvc_storage/`) but can be extended to cloud.
- Sync using `dvc push` and `dvc pull`.

2.6 Monitoring Stack

Prometheus

- Collects metrics from backend at `/metrics` every 5 seconds.

- Metrics include HTTP request count, duration, and error rate.
- Runs as a container using official Prometheus image.

Grafana

- Displays backend and system-level metrics.
- Dashboards preloaded from `monitoring/grafana/dashboards/`.
- Visualizations include:
    - FastAPI request performance
    - Host system metrics via Windows Exporter (CPU, memory, network, disk IO)

2.7 Docker Compose

Purpose: Orchestrates services including frontend, backend, Prometheus, and Grafana.

Configuration: Defined in `docker-compose.yml`

Service	Build Path	Port	Dependencies
frontend	./frontend	8500	backend
backend	./backend	8000	none
prometheus	prom/prometheus	9090	backend
grafana	grafana/grafana	3000	prometheus

Shared Volumes:

- `./models` to `/app/models`
- `./model_retrain` to `/app/model_retrain`
- `./mlruns` to `/app/mlruns`

---

3. Data Specifications

3.1 `/predict` Endpoint

- **Input**: JPEG or PNG image
- **Output**:

```
{
  "status": "success",
  "prediction": [[0, 1, ...]],
  "shape": [128, 128]
}
```

3.2 `/save-for-retrain` Endpoint

- **Input**: JPEG or PNG image

- **Output**:

```
{ "status": "saved" }
```

---

4. Testing (Located in `tests/`)

File	Tests
`test_api.py`	All endpoints respond correctly
`test_train.py`	Model training produces valid model
`test_save.py`	Feedback images are correctly saved
`test_metrics.py`	Prometheus metrics exposed and updated

Test execution:

```
pytest tests/
```

---

5. Error Handling & Logging

- Backend uses Python's `logging` to log all requests and errors.
- Any prediction or training failure is returned with HTTP 500 and detailed error message.
- Frontend captures and logs user-level failures in browser console.
- Missing models or corrupt input images are gracefully rejected.

---

6. Security & Input Validation

- MIME type and image shape are validated before processing.
- `/predict` explicitly rejects non-image files.
- Uploaded image size and prediction array dimensions are checked for consistency.
- Potential enhancements: OAuth for API, CSRF for frontend forms.

---

7. Future Extensions

- Auto-trigger retraining after N feedback images.
- Containerize MLflow server (port 5000).