EdgeX AI Challenge

"Operation FractureScope" – Industrial Anomaly Detection in Harsh Environments

SUBMISSION BY:

Mayur Pratim Das
MTech (CSE) | IIT Ropar
Mayurpd.official@gmail.com
+91-9957351454

Table of Contents:

TASK 1 | Design a Multi-Modal AI Pipeline

SUBTASK 1.1: Creating the Dataset

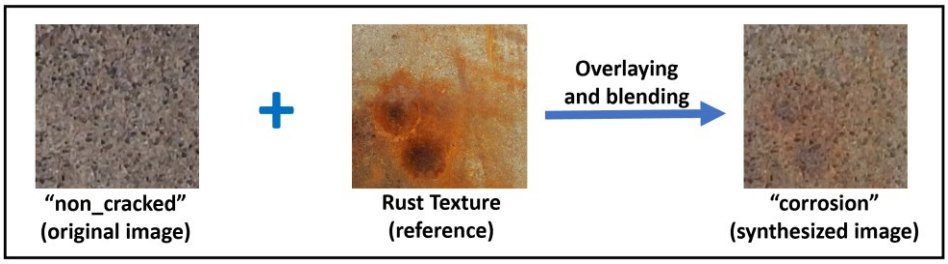
Challenge: The preliminary challenge was to create a dataset that has at least three labelled classes of concrete defects. The SDNET2018 dataset has images of three different surfaces, Deck, Pavement and Wall and only two classes: ***cracked*** and ***non_cracked*** which is meant for only binary classification. Aim was to synthesize a multimodal dataset consisting of three types of defects for our pipeline to perform multi-class classification.

My Solution: I decided upon creating a dataset with concrete images of three types of defects (or classes):

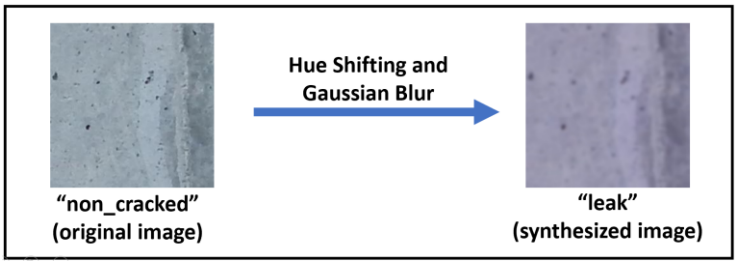
- **cracked** (just copied from SDNET2018) ➔1000 images
- **corrosion** (synthesized from non_cracked images in SDNET2018 using OpenCV) ➔1000 images
- **leak** (synthesized from non_cracked images in SDNET2018 using OpenCV) ➔1000 images

Simulating Defects:

The corrosion images were synthesized by taking 1000 non_cracked images from SDNET2018 and a sample texture image of concrete rust was overlayed on the clean images and blending techniques were used to try induce realism.



The leak images were synthesized by applying simple CV techniques like colour space changing from BGR to HSV then shifting the hue towards blue or green to mimic water and moisture leakage. Gaussian blur also applied at the end to mimic the blurry appearance of water flow of seepage.

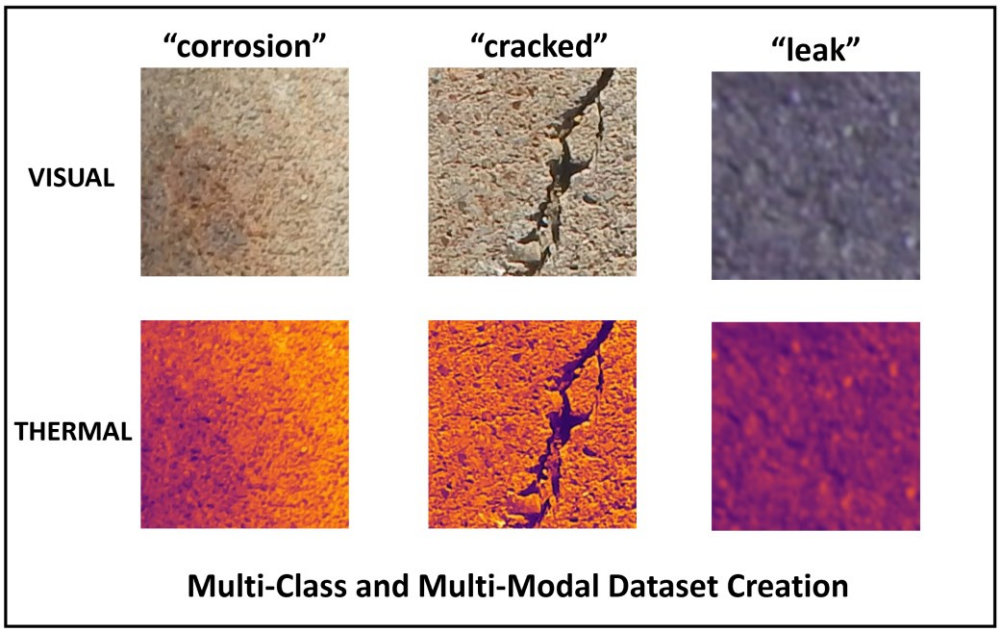


Inducing Multi-Modality (Visual + Thermal):

The next challenge was to induce multi-modality to our dataset. I decided upon adding thermal imaging as the second modality alongwith visible RGB images as that makes sense in industrial inspection. To synthesize thermal images, I took each image from the cracked, corrosion and leak class and applied the following steps:

- convert to grayasale (using cv2.COLOR_BGR2GRAY)
- apply color map "inferno" (using cv2.COLORMAP_INFERNO)
- store each original images in two corresponding folders ***visual*** and ***thermal*** with same file names.

This thermal synthesis process enabled the model to learn robust multimodal feature representations, even without relying on actual thermal hardware data. The results are shown below:



Now that our dataset was created, we can move on to the next steps, which is preprocessing the data.

SUBTASK 1.2: Data Preprocessing

To prepare our multimodal dataset for training and evaluation, we designed a robust preprocessing pipeline tailored for defect detection tasks using both visual (RGB) and synthetic thermal images.

1 **Folder Structure Standardization:** Each sample consisted of two aligned images:

- An RGB visual image.
- A corresponding synthetic thermal image.

Organized under class-specific directories (e.g., normal, corrosion, leak) to facilitate dataset loading using custom PyTorch Dataset classes. The final directory structure of the dataset as as shown:

```
synth # synthesized dataset
├── corrosion # class: 1
│   ├── thermal
│   └── visual
├── cracked # class: 0
│   ├── thermal
│   └── visual
└── leak # class: 2
    ├── thermal
    └── visual
```

2 **Synchronized Image Transformations:** To ensure that RGB and thermal inputs remained spatially aligned, we implemented paired transformations, applied identically to both modalities:

- ***Image Resizing:*** All input images were resized to a fixed resolution of **224×224 pixels** to match the input requirement of the CNN model. This was applied uniformly to both RGB and thermal images.
- ***Data Augmentation (for Training):***
  - **Horizontal Flipping:** simulate different camera orientations and variations in real-world.
  - **Color Jitter (in RGB images only):** brightness and contrast variations were applied to RGB images to simulate lighting changes, while thermal images remained unchanged to retain physical consistency
  - **Normalization:** Pixel values of images were normalized to the range **[0, 1]** by dividing by 255.0. This standard normalization improves convergence and training stability

3 **Custom PyTorch Dataset class and Loader (To facilitate Early Fusion: RGB + Thermal):**
We implemented a custom `SynthDataset` class that:

- Loads both RGB and thermal image paths.
- Applies synchronized transforms.
- Returns a **6-channel tensor** by concatenating RGB (3-channel) and thermal (3-channel) images.
- Maps each sample to its corresponding class label.

SUBTASK 1.3: Architecture Design and Model Selection

1 Motivation: In industrial inspection settings, combining multiple modalities (e.g., RGB and thermal) helps capture complementary information:

- **RGB images** reveal texture, colour, and visual patterns of surface defects like corrosion.
- **Thermal images** capture temperature gradients, useful for detecting leaks or hidden defects invisible in the RGB spectrum.

We chose an **Early Fusion CNN architecture** to leverage this **multi-modal synergy** by fusing both modalities at the input level, enabling the network to learn unified representations of defects from the beginning.

2. Architecture Design:

EarlyFusionCNN – Overview

- **Input:** A 6-channel tensor created by **stacking RGB (3 channels)** and **thermal (3 channels)** images.

  Justification: Early Fusion of modalities simpler and faster than mid/late fusion techniques which will play a crucial role while deployment in edge devices. Also, it encourages model to learn low-level joint features (e.g., aligned RGB+thermal edges or patterns).

- **Backbone:** Pretrained **ResNet-18** used as the feature extractor.

  Justification:

  - ResNets are benchmark CNNs for Classification.
  - Lightweight yet effective for edge deployment.
  - Residual connections allow deeper gradient flow, reducing vanishing gradients and improving convergence.
  - Easily extensible for quantization and pruning (**which is a crucial step for edge deployment**).

- **Modification:** The first convolution layer of ResNet was modified to accept **6-channel inputs** instead of 3 channels by default.

  Justification: A necessary step to ensure our model architecture matches with input dimensions.

- **Classifier Head:** Global Average Pooling followed by a fully connected layer (3-class output for background, corrosion, and leak).

  Justification: Reduces overfitting by limiting the number of parameters and preserves semantic information across the entire image without needing large FC layers.

SUBTASK 1.4: Model Training and Performance Evaluation

1. Training the Model

Training Parameters:

PARAMETER	VALUE
Optimizer	Adam
Learning Rate	1e-4
Batch Size	32
Epochs	30 (with Early Stopping)
Loss Function	Categorical Cross-entropy

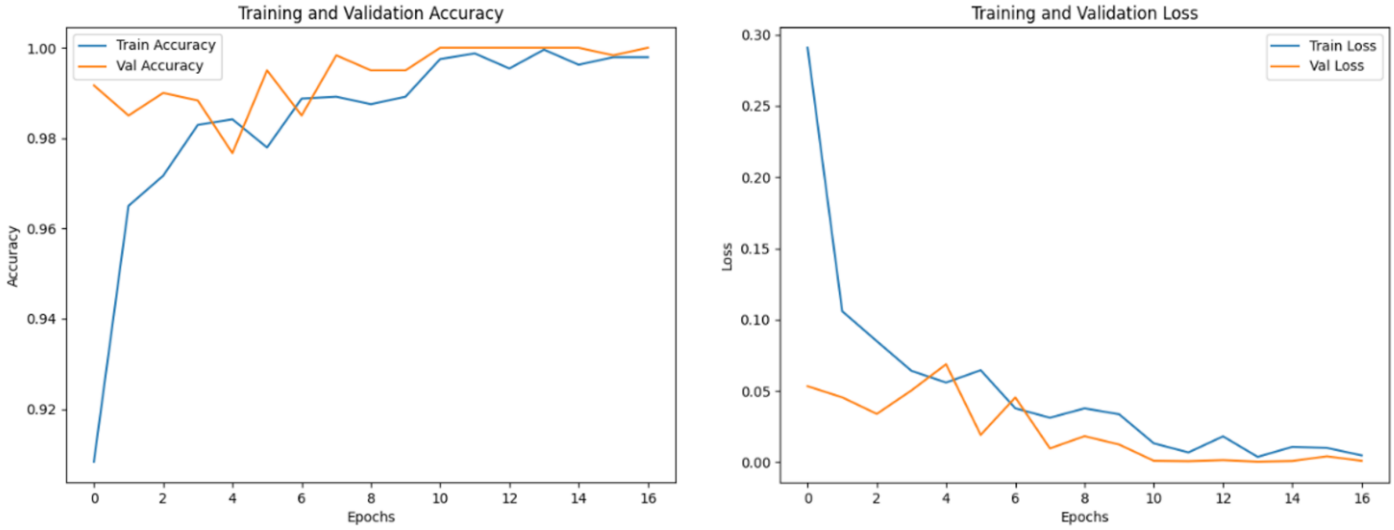
Training done with 3 class dataset each with 1000 samples, each sample is a 6-channel input (3 visual + 3 thermal). Dataset was split into Training and Validation sets in 80:20 ratio by Dataloader.

Training was done for 30 epochs by default but **Early Stopping was implemented based on Validation loss with patience = 3, to avoid overfitting.**

In each epoch, validation loss and validation accuracy was monitored and two best models were saved:

Best_validation_accuracy model and *Best_validation_loss model.*

The Training Results are plotted and shown below:



Max Validation Accuracy was 1.0 when Early Stopping kicked in at around epoch 16. Patience = 3 was set for stopping. The high validation accuracy is caused by fairly simple and distinguishable classes. As the dataset is mostly synthetic, some anomalies are expected. But the pipeline is robust and effective.

2. Model Evaluation

Test Dataset Creation: A new unseen dataset was synthesized in same manner as training dataset but unseen images from original SDNET2018 were used this time. Each class, cracked, corrosion and leak had 80 samples and each sample had dual modality, visual and thermal.

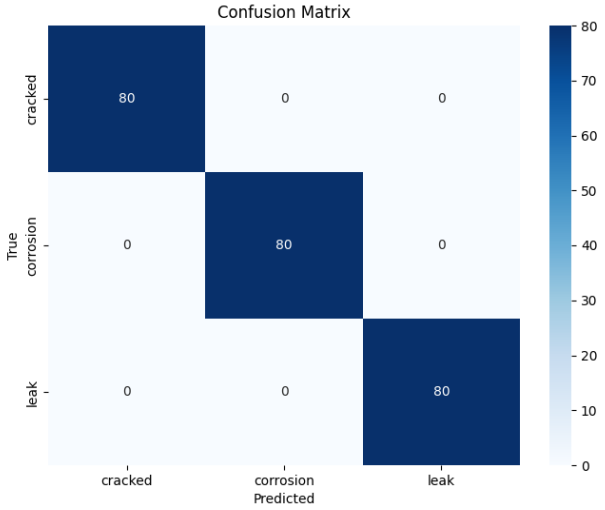
Evaluation: The *best_accuarcy_model* that was saved during training was loaded and inference was done on all 240 samples. Different Performance Metrices were used like Test Accuracy, Test Loss, Precision, Recall amd F1-score and stored in a CSV file. Also a Confusion Matrix was also plotted to reflect the model performance which was stored as a png image. All the metrices are shown below:

The contents of classification report are as follows:

```
=== Test Evaluation Summary ===
Total Test Samples: 240
Test Loss: 0.0023
Test Accuracy: 1.0000
Test Precision: 1.0000
Test Recall: 1.0000
Test F1 Score: 1.0000

Detailed Classification Report:
              precision    recall  f1-score   support
cracked             1.0       1.0       1.0      80.0
corrosion           1.0       1.0       1.0      80.0
leak                1.0       1.0       1.0      80.0
accuracy            1.0       1.0       1.0       1.0
macro avg           1.0       1.0       1.0     240.0
weighted avg        1.0       1.0       1.0     240.0
```

Also, the Confusion matrix is shown below:



Observation: The Test results are excellent. Infect, the too perfect scores maybe again attributed to the synthetic nature of the dataset. Real world performance is expected to be less perfect. This only for demo purpose, the pipeline can perform well and give realistic performance with real data.

CONCLUSION: The Task of creating a Multi-Modal AI Pipeline has been completed successfully. The final results are not indicative of real-world model performance as the training was done on simulated data. But the implementation and working is perfect and will be efficiently handle real world multi-modal data with minor modifications. I did not go with more complex architecture keeping our end goal of edge device deployment in mind.

P.S. ***Please find the detailed description of the Git Repo at the last section of the report.***

TASK 2 | Model Optimization & Deployment on Edge Devices

SUBTASK 2.1: Model Optimization

We have applied Pruning and Quantization optimization techniques on our saved model from Task 1.

Step 1: Apply Pruning to the model

To optimize the **EarlyFusionCNN** model for edge deployment, we applied **L1 unstructured pruning** to remove less important weights from the model's convolutional and linear layers:

Checkpoint loading: The pretrained model (**best_acc_model_20250505-060834.pth**) from Task 1 was loaded and set to evaluation mode.

Pruning step:

- For every Conv2d and Linear layer, **30% of the weights** with the smallest L1-norm were pruned (i.e., set to zero).
- The pruning masks were then **permanently removed**, making the sparsity structural rather than just a mask.

Exporting: The pruned model was exported to **ONNX format** (**`EarlyFusionCNN_pruned.onnx`**) with support for **dynamic batch sizes** and named inputs/outputs.

This pruning process resulted in:

- **Reduced model size and compute load**, beneficial for low-power inference.
- Maintained accuracy by selectively pruning unimportant weights only.

Details of this step can be found in **`prune_onnx_export.py`** script in **`/deploy`** directory.

SUBTASK 2.2: Applying Quantization and final model export to ONNX

To further optimize the **pruned EarlyFusionCNN model**, this script performs **static post-training quantization** using ONNX Runtime tools. Here's what the process involved:

1. **Model Setup**
   • **Input model**: The previously pruned ONNX model (**`EarlyFusionCNN_pruned.onnx`**).
   • **Output model**: The quantized ONNX model is saved as **`EarlyFusionCNN_pruned_quant.onnx`**.
   • **Calibration data**: A small folder of sample images (**`data/sample_images`**) is used to collect activation statistics needed for quantization.

2 **Custom CalibrationDataReader:** A class **`ImageCalibReader`** was created to feed calibration images to the quantizer. Each image is loaded and resized to 224×224, normalized to [0, 1] float range, fused into a 6-channel input by duplicating the RGB image (to simulate sensor fusion) and transposed to (6, 224, 224) and wrapped in a batch dimension.

3 **Static Quantization:** The **`quantize_static()`** function was used with **QUInt8** precision for both activations and weights and calibration based on real data samples (provided via **`ImageCalibReader`**). This converts the floating-point model into a **low-precision (int8) version**, significantly reducing: **Model size**, **Inference latency** and **Memory bandwidth** requirements — all critical for real-time edge deployment.

4 **Export to final ONNX:** The resulting quantized model (**`EarlyFusionCNN_pruned_quant.onnx`**) is now ready for deployment on resource-constrained devices like **Jetson Nano/Orin** or **Raspberry Pi**.

Details of this step can be found in **`quantize_onnx_export.py`** script in **`/deploy`** directory.

SUBTASK 2.3: Created Demo Inference Pipeline Using ONNXRuntime

To run real-time predictions on edge devices, we created a lightweight inference script using ONNX Runtime, which is optimized for high-speed execution on CPUs or lightweight GPUs like those found in Jetson or Raspberry Pi.

1. Loading the Quantized ONNX Model: We used `onnxruntime` to load the final optimized model.

- `InferenceSession` initializes the runtime session.
- We specified `CPUExecutionProvider` to simulate behavior on devices like Jetson Nano (which uses ARM CPUs and/or integrated GPUs).

2 Input Processing:

- Reads and resizes each image to 224x224.
- Normalizes pixel values to [0, 1].
- Simulates a 6-channel input by concatenating RGB twice (as a proxy for RGB + thermal).
- Reshapes data to model input format: (1, 6, 224, 224).

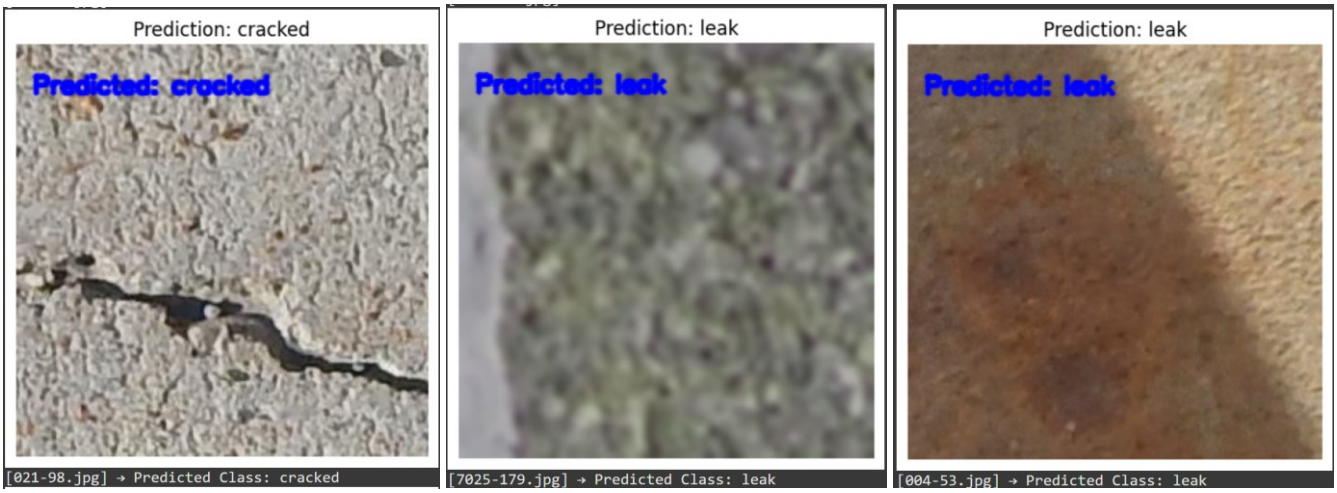
3 Inference:

- Runs the input through the ONNX model using `session.run`.
- Retrieves the predicted class using `np.argmax`.

4 Visualization:

- Draws the prediction as overlay text on the image using OpenCV.
- Converts the image to RGB format for display.
- Uses `matplotlib` to show the image and predicted class without GUI dependencies (ideal for Colab or headless environments).

5. Inference Results: I ran the script on some random samples in Google collab and got inferences as below:



The above demo results simulate our model deployed in edge devices with less compute power and resources. Thus, our model is successfully optimized and ready for deployment in edge devices.

Details of this step can be found in **`edge_inference_visualizer.py`** script in **`/deploy`** directory.

SUBTASK 2.4: Designing a Feedback Loop via 4G Connectivity

I have not implemented this part due to limited time but I am providing a schematic plan how to create the feedback loop.

1. System Overview

In a typical industrial inspection scenario, the robot (equipped with an edge device like Jetson Nano) performs real-time inference using the quantized ONNX model. Upon detecting an anomaly (crack, corrosion, or leak), a lightweight feedback loop is triggered to inform the remote human operator over a 4G network.

2. Core Components

Component	Description
Edge Device	Runs the inference pipeline on-site, detects anomalies locally.
4G Dongle/Module	Provides mobile internet access for the device to send feedback
MQTT/HTTP Client	Sends anomaly alerts and optional snapshots over the 4G network.
Remote Operator UI	A lightweight dashboard (web/mobile) to receive alerts, images, and logs.

3. Feedback Loop Design

Step-by-step flow:

1. **Local Detection**
    - The robot continuously inspects using the ONNX inference engine.
    - When a defect (e.g., leak or corrosion) is detected with high confidence, an alert is generated.
2. **Alert Packaging**
    - The edge device creates a payload:

```
{
  "timestamp": "2025-05-06T12:34:56Z",
  "location": "Pipe A-42",
  "predicted_class": "leak",
  "confidence": 0.92,
  "image_base64": "<optional compressed image>"
}
```

3 **Data Transmission**

a. Payload is transmitted over 4G using **MQTT** (for low latency) or **HTTP POST** to a cloud or operator endpoint.

b. If bandwidth is constrained, image data can be compressed or sent only for severe anomalies.

4 **Operator Notification**

a. On the remote side, a dashboard/mobile app receives alerts in real time.

b. Operator can view the prediction, timestamp, image, and optionally send commands back (e.g., “Stop”, “Inspect again”).

*When data is scarce or sensitive, we address the limitations by leveraging **synthetic data generation** and **federated learning**. Synthetic data, created using simulation tools or domain randomization, helps augment the dataset with diverse, labeled samples without requiring additional real-world data collection. This enhances model generalization while preserving privacy. For sensitive data scenarios, federated learning allows multiple devices or institutions to collaboratively train a shared model without sharing raw data—only model updates are exchanged. This ensures data privacy while still enabling robust, distributed learning.*

TASK 3 | Secure AI Logging System

I am giving a rough overview of how the system would look.

System Overview: Each AI inference event at the edge device logs the following metadata:

- **Timestamp**: Exact UTC time of inference.
- **Location**: Robot GPS or local coordinate system.
- **Image Hash**: SHA-256 hash of the input image (ensures data integrity).
- **Prediction**: AI model output (e.g., "crack", "corrosion", "leak").

This metadata is embedded into a **block** and appended to a local blockchain ledger. The structure ensures **immutability**, preventing retrospective tampering of results and enabling robust forensic analysis in compliance scenarios.

Block Structure (Pseudo-Code)

Each block in the chain has the following format:

```
class InspectionBlock:
    def __init__(self, index, timestamp, image_hash, prediction, previous_hash):
        self.index = index
        self.timestamp = timestamp
        self.image_hash = image_hash
        self.prediction = prediction
        self.previous_hash = previous_hash

    #function to calculate hash
    def calculate_hash(self):

        return sha256(data.encode()).hexdigest()
```

On every inference, a new block is created and linked with the previous block's hash.

To complete the system, the blockchain ledger is linked with a **3D digital twin dashboard** that visualizes asset health in real time.

This secure logging system ensures that all AI inspection data is transparently recorded, securely stored, and verifiable.

APPENDIX | Submission Format and Git Repo description

I have created a Git Repo named **OP_FractureScope** for submitting my assignment. The directory structure is shown below:

```
OP_FractureScope
├── data // contains all images and datasets
├── deploy // contains all quantization, pruning, onnx and edge inference scripts
├── outputs //contains all runs outputs
├── requirements.txt // list of dependencies
├── report // contains PDF report describing modeling, design, and conclusions
├── src // contains all model definition and training related scripts
└── README.md

data
├── sample_images // images used for edge inference using onnxruntime
├── sdnet // Original SDNET2018 images used to synthesize our dataset
│   ├── cracked
│   └── non_cracked
├── synth // Our Main Dataset
│   ├── corrosion // CLASS: 1
│   │   ├── thermal
│   │   └── visual
│   ├── cracked // CLASS: 0
│   │   ├── thermal
│   │   └── visual
│   ├── leak // CLASS: 2
│   │   ├── thermal
│   │   └── visual
│   └── test_dataset // Test Dataset synthesized for model evaluation
│       ├── corrosion
│       │   ├── thermal
│       │   └── visual
│       ├── cracked
│       │   ├── thermal
│       │   └── visual
│       └── leak
│           ├── thermal
│           └── visual
└── textures
    └── rust_texture.png // reference rust texture used for synthesis of "corrosion' images

src
├── data
│   ├── dataset.py // file with Custom PyTorch Dataset class SynthDataset and Loader
│   └── transforms.py // data augmentations script
├── models
│   └── early_fusion.py // our EarlyFusionCNN model definition
├── train
│   ├── evaluate_best_model.py // model evaluation script
│   └── train.py // main training loop
└── utils
    ├── generate_test_thermal.py // script to synthesize thermal images for TEST dataset
    ├── generate_thermal.py // script to synthesize thermal images
    ├── simulate_defects.py // script for synthesis of defects, "corrosion" and "leak"
    └── simulate_test_dataset.py // script to generate the TEST dataset
```

```
outputs
├── metrics // contains all output metrices and plots after training
│   ├── accuracy_plot_20250505-060834.png
│   ├── classification_report_20250505-060834.txt
│   ├── confusion_matrix_20250505-060834.png
│   └── loss_plot_20250505-060834.png
├── models // saved model checkpoints
│   ├── best_acc_model_20250505-060834.pth
│   └── early_fusion_best_loss20250505-060834.pth
├── tensorboard // tensorboard logs
│   └── 20250505-060834
└── test_results // model evaluation results on TEST dataset
    ├── classification_report.csv
    ├── confusion_matrix.png
    └── evaluation_summary.txt

deploy
├── edge_inference_demo.py // script of running edge inference on onnxruntime
├── edge_inference_visualizer.py // script of edge inference on onnxruntime (with visual output)
├── optim_models // contans all saved optimized models
│   ├── onnx
│   │   ├── EarlyFusionCNN_pruned.onnx // pruned model
│   │   └── EarlyFusionCNN_pruned_quant.onnx // pruned, qunatized and onnx exported final model
│   └── optimizedEarlyFusionCNN.pth
├── optimize_model.py //script to apply pruning and quantization only
├── prune_onnx_export.py // script to apply pruning and onnx export
└── quantize_onnx_export.py // final script to apply pruning, quantization and exoprt to onnx
```
