Structural Defect Detection AI Project

Artificial Intelligence for Structural Engineering / Chemical Engineering

Team member:
Thanh Nam Vu
Xuan Tuan Minh Nguyen
Contributor:
Aidid Yassin
Leon Nhor
Matthew Hadkins

Contents

1 Introduction

1.1 Background and Motivation

With growing emphasis on proactive infrastructure maintenance, the demand for AI-powered inspection systems has risen significantly. Our project explores the intersection of chemical and structural engineering through the development of a deep learning model capable of identifying visual structural defects, specifically corrosion and cracks, from high-resolution images. We chose this project due to its strong relevance to real-world engineering challenges and its ability to provide hands-on experience in computer vision and deep learning. As drone technology and high-resolution image capture become increasingly accessible, AI-based inspection tools offer a scalable and non-invasive solution to structural monitoring. These tools can support earlier detection of surface degradation, improve safety, and reduce the need for labour-intensive manual inspections.

Moreover, the use of image-based data aligns well with our prior exposure to engineering contexts involving visual analysis, and complements our shared interest in machine learning applications. This project enables us to build practical competencies in image annotation, data preprocessing, and neural network training, while also contributing to the advancement of safer and more efficient inspection practices.

The final system is intended for use by asset maintenance teams, structural engineers, facility managers, and civil inspectors, providing them with a reliable tool for early-stage fault detection. By automating the identification of critical defects such as rust or cracks, the solution has the potential to optimize maintenance planning, reduce operational costs, and extend the service life of infrastructure systems.

1.2 Project Objectives

The objective of this project is to develop and evaluate a deep learning model capable of detecting and classifying structural defects in high-resolution photographic data. Our approach is guided by two central research questions:

- Can a deep learning model reliably identify structural defects in image data?
- Can it accurately distinguish between different types of flaws, such as cracks and corrosion?

To address these questions, we will carry out a complete AI development pipeline involving:

- Collecting, annotating, and preprocessing high-resolution tower images using tools that support both polygonal segmentation and bounding box annotation formats.
- Training and optimizing several convolutional neural network models, specifically YOLOv8m for object detection and YOLOv8m-seg for segmentation, tailored to identify and differentiate between cracks and rust on structural surfaces.
- Evaluating both the training process and final model performance using a combination of metrics: training and validation set loss curves to monitor convergence, learning stability, and how well they generalize to unseen data; the performance of the models using standard metrics such as confidence scores, Precision, Recall, and mean Average Precision (mAP) to assess both detection precision and segmentation quality.
- Exploring the real-world applicability of AI-powered defect detection systems in engineering inspection workflows, particularly for use by civil engineers, asset managers, and infrastructure maintenance teams.

This project not only aims to produce a technically sound AI solution but also reinforces our core capabilities in image annotation, preprocessing, model validation, and the practical application of AI in structural engineering contexts.

1.3 Summary of Outcomes

The final outcome of this project is a working AI model capable of detecting structural flaws in high-resolution images and classifying them with associated confidence scores. Specifically, the system takes as input a photographic image or video, such as that of a solar panel or telecommunications tower, and outputs annotated regions indicating structural defects. Each detected defect is labeled with a predicted class (e.g., crack or corrosion) and a corresponding confidence score reflecting the model's certainty.

Throughout the project, we developed a complete AI pipeline that encompassed dataset annotation, model training, evaluation, and performance benchmarking. This process involved the use of both bounding box and segmentation annotations, as well as the training and comparison of multiple YOLO-based deep learning models (YOLOv5m, YOLOv5m-seg, YOLOv8m, and YOLOv8m-seg).

In addition to producing a functional detection and segmentation system, the project has significantly strengthened our technical capabilities in areas such as image preprocessing, annotation workflows, model training, and metric-based validation. More importantly, it has demonstrated the potential of AI to support engineering workflows, particularly in domains where traditional visual inspections may be time-consuming, hazardous, or cost-prohibitive.

We believe that the models and methodology developed in this project can be adapted for a broad range of real-world infrastructure monitoring tasks, ultimately helping make inspections safer, faster, and more reliable.

2 Dataset

2.1 Data Source

The original dataset consisted of 572 high-resolution drone images of telecommunications towers. These images, captured under varying lighting conditions and from multiple angles, depict complex metal structures including lattice frameworks, cross-bracing, mounting platforms, and structural joints. Such variability introduces realistic challenges, making the dataset highly suitable for developing AI models aimed at structural inspection.

Out of the full dataset, 534 images were manually annotated. These labeled samples provided the foundation for training and evaluating our models. Many images displayed visible signs of structural defects, particularly *corrosion and cracks*, often concentrated at high-stress locations such as base joints and welded connections. These naturally occurring flaws made the dataset especially valuable for training deep learning models focused on real-world defect detection.

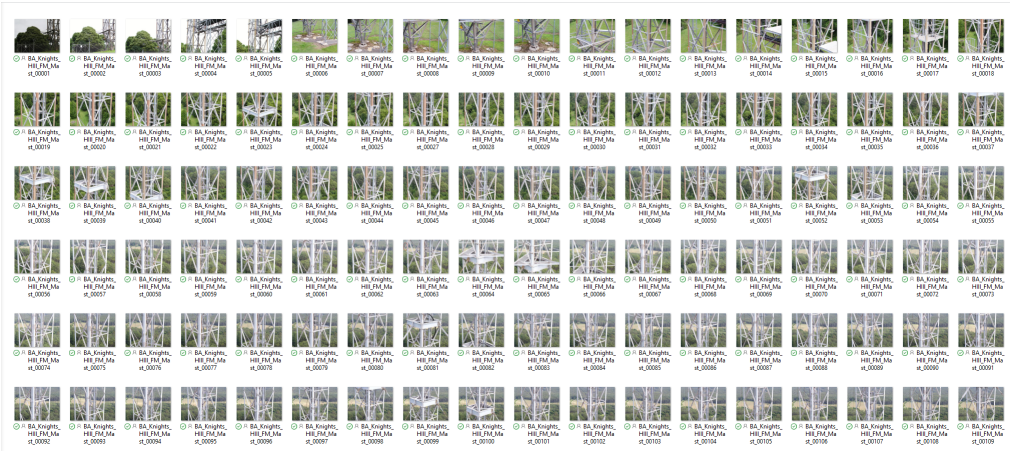


Figure 1: Original Dataset

2.2 Data Processing

Roboflow served as the central platform for our data processing workflow, offering integrated tools for annotation, dataset organization, and preliminary dataset analysis. It played a critical role in facilitating both collaborative labelling and systematic preparation of data for model training.

Our data processing steps began with an initial review of a small subset of images to evaluate image quality, identify typical defect patterns, and establish consistent annotation guidelines for both bounding boxes and polygon masks. This early assessment helped ensure that annotations across team members remained consistent and aligned with the project objectives.

Annotations were carried out using Roboflow, which enabled consistent labeling across team members and provided version control for the dataset. Each team member was responsible for annotating approximately 120 images, ensuring balanced contributions and efficient workflow management.

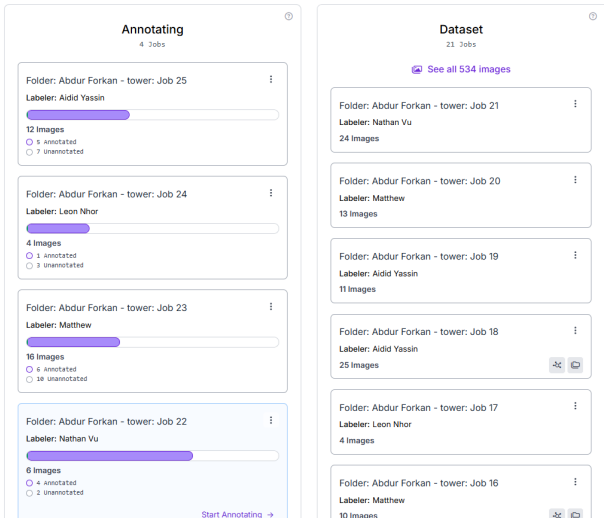


Figure 2: Assigning annotating task for all members

Using Roboflow's interface, team members then carried out collaborative annotation, assigning both bounding box and polygon labels depending on the task. Bounding boxes were used to annotate defect regions such as rust and cracks, while polygons were drawn to segment the full tower structure. This dual approach enabled the dataset to serve both the object detection and segmentation pipelines.

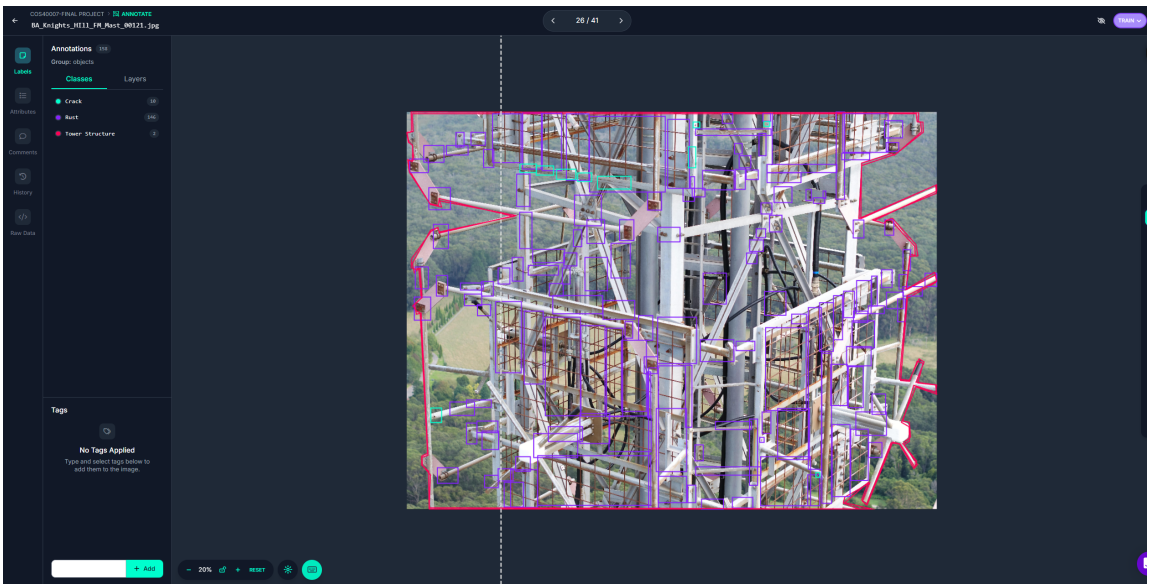


Figure 3: Annotation example on one image

To understand the dataset's composition, we used Roboflow's built-in analytics to examine annotation density, image resolution, and class distribution. As shown in Figure 1, the dataset is highly imbalanced:

- The Rust class accounts for approximately 97% of all labeled defects, totaling 36,213 annotations.
- The Crack class is significantly underrepresented with only 402 annotations.
- The Tower Structure segmentation class includes 537 annotated instances.

Images had a median resolution of 5280×3956 pixels, with an average resolution of 20.89 megapixels and a mean of 69.6 annotations per image.

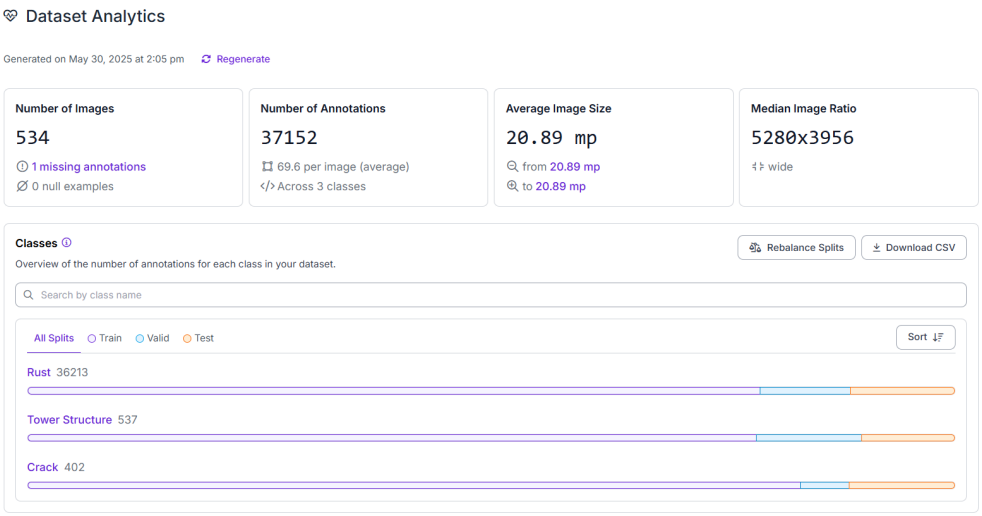


Figure 4: Exploratory Data Analysis

This class imbalance poses a potential risk for performance bias, particularly favoring the Rust class during training. As such, addressing this imbalance through augmentation and weighted loss functions will be critical to ensuring fair model performance across all defect types.

Roboflow was also used to streamline the dataset export process. It handled train/validation/test splits, enabled Yolov8-format exports, and allowed us to confirm class balance before training. These utilities significantly accelerated our preprocessing pipeline and ensured that the data was compatible with YOLOv8's training requirements.

Finally, to support both object detection and segmentation tasks, each of the annotated images included two types of labels:

- Bounding box annotations for identifying and classifying defect regions, such as cracks and rust.
- Polygonal annotations for segmenting the entire tower structure within each image.

Since YOLOv8 requires different data formats for object detection and segmentation tasks, we wrote a custom `Python` script to programmatically separate the labeled dataset into two subsets:

- A dataset formatted specifically for object detection using bounding box labels.
- A separate dataset prepared for segmentation using polygon annotations.

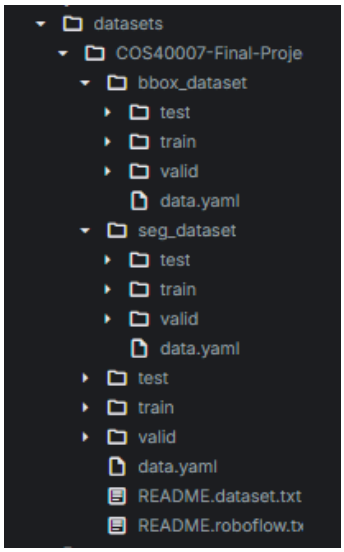


Figure 5: Dataset structure

This separation ensured compatibility with the respective input requirements of YOLOv8m and YOLOv8m-seg, allowing us to independently train and evaluate each model type using task-specific data.

3 AI Model Development

3.1 Image Processing

Given the extremely high resolution of our source images (median size approximately 5280×3956), a direct resize to model input dimensions such as 640×640 or even 1280×1280 would lead to substantial information loss. In particular, this would make it difficult for the model to detect small-scale defects like rust, which may occupy only a few pixels in the downscaled version. To mitigate this, we employed a tiling strategy for the object detection dataset.

Tiling(mainly for Defects Detection task)

Each original image was tiled into four sub-images by dividing it into 2 rows × 2 columns, resulting in smaller tiles of approximately 1320×989 pixels. This approach was carefully selected to enhance the visibility of fine-grained features such as small cracks or rust spots, which could otherwise be lost during downscaling.

Moreover, tiling allowed us to effectively zoom in on local regions of the tower structure while still preserving enough surrounding context for the model to understand the spatial relationship between components. This balance between resolution and context is essential for effective object detection, as overly zoomed-in tiles might remove critical structural cues, while overly scaled-down images would blur or erase subtle defect features.

In addition to improving feature visibility, tiling significantly increased the dataset size, which further enhanced the model's ability to generalize. By exposing the detection model to varied sections of each tower image, we encouraged it to learn more diverse patterns and textures. This is particularly important when working with relatively limited data, as it acts as a form of implicit data augmentation.

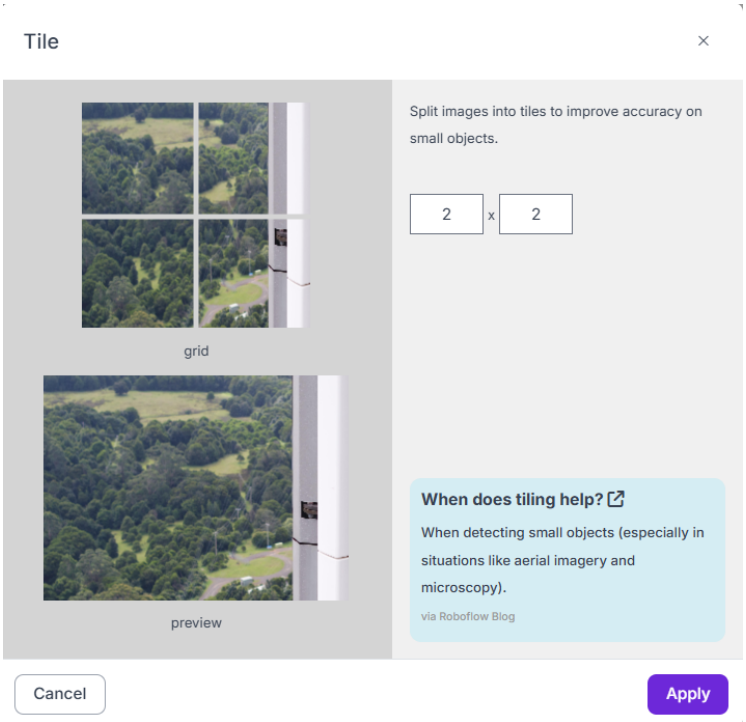


Figure 6: Tiling

Resizing

Following tiling, we resized the detection images to 1280×1280 before training the YOLOv8m model. This resolution was chosen as a compromise between detail preservation and computational feasibility. On one hand, it retained enough high-frequency information to ensure that small objects like rust and cracks remained detectable. On the other hand, it remained within the GPU memory constraints of our A100 environment in Google Colab (40GB), avoiding runtime bottlenecks or memory overflows during training.

For the segmentation model (YOLOv8m-seg), the preprocessing pipeline slightly differed. Since segmentation models operate on a per-pixel basis, they are significantly more memory-intensive than detection models. To accommodate this, we resized the segmentation images to 640×640. Although this resolution is lower, it is still sufficient to preserve the global shape and edges of the tower structure, which is generally large and occupies a significant portion of each image. The decision to downscale segmentation inputs also enabled us to train more efficiently, reducing GPU load and accelerating training iterations without meaningfully compromising segmentation quality. In fact, due to the tower's strong visual contrast against the natural green backgrounds, its features remained distinguishable even at this reduced resolution.

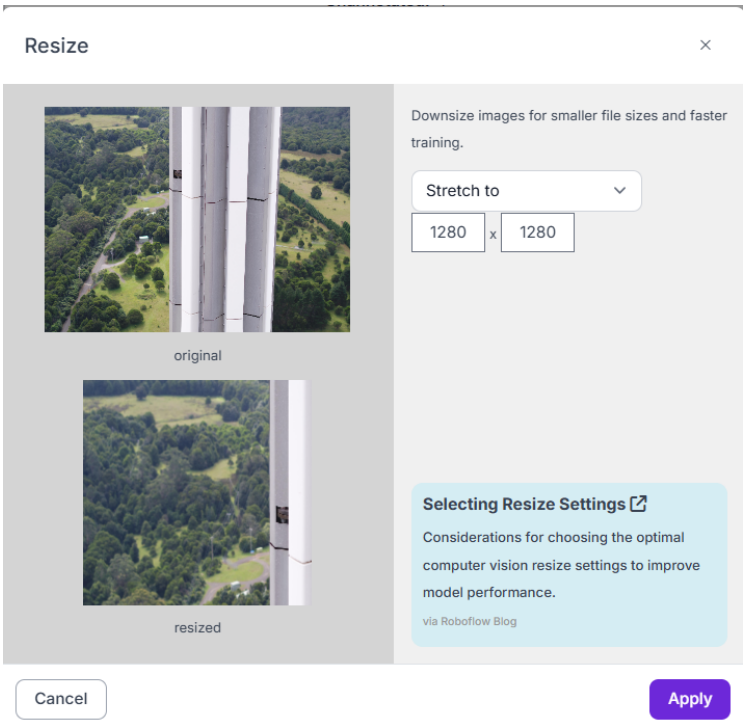


Figure 7: Tiling

Auto-Orientation

In addition to spatial transformations, we applied auto-orientation correction to all images. This preprocessing step is critical in drone-captured or handheld imagery, where orientation can vary dramatically between shots.

Auto-orientation is essential for maintaining consistency in the dataset, which in turn allows the model to generalize better to test-time scenarios. Moreover, in field deployment, particularly with drones, images may be captured at arbitrary angles. By training on consistently aligned images, we ensure that the model will be more resilient when processing real-world, aerial, or dynamically oriented footage.

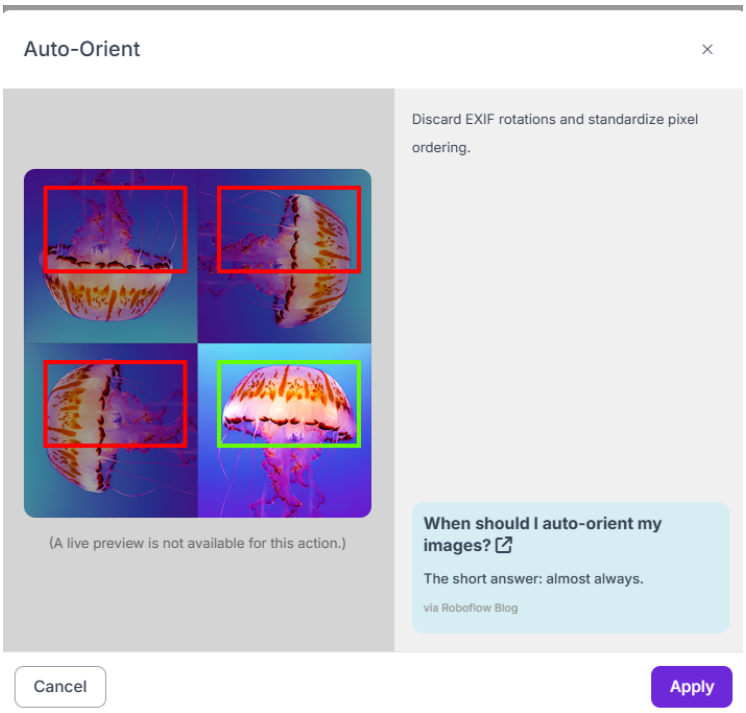


Figure 8: Tiling

Image Augmentation

To improve the robustness and generalization of both the object detection and segmentation models, we applied data augmentation techniques using Roboflow, exclusively to the training set. These augmentations simulate the environmental and visual variability expected in real-world infrastructure inspections, where images may be captured under inconsistent lighting, from various angles, or using drone-mounted cameras.

Data augmentation was especially important in our case due to the relatively small size of the original dataset (572 images). By applying controlled transformations, we were able to generate more diverse training samples, which helps mitigate overfitting and improves model performance on unseen data as much as possible.

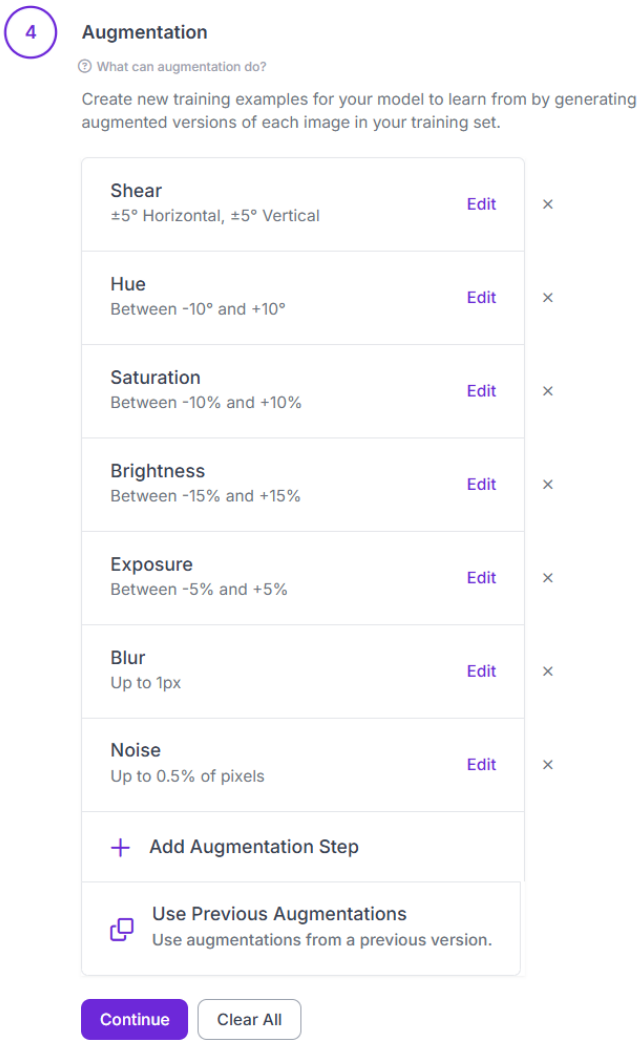


Figure 9: Data Augmentation

One of the key augmentations was shear transformation (±5°), which mimics geometric distortion that can occur when towers are photographed at an angle, which is common during aerial or drone-based inspections. This allows the model to remain accurate under slanted perspectives.

We also applied hue, saturation, brightness, and exposure adjustments to replicate lighting variations caused by time of day, weather, or reflections from metal surfaces. These color-based augmentations ensure that the model learns to detect rust and cracks consistently, regardless of lighting conditions.

To account for potential image quality issues, blurring (up to 1px) was introduced to simulate motion blur from drone movement or wind, while Gaussian noise (up to 0.5%) was added to mimic sensor noise in challenging conditions. These augmentations help the model remain effective even when images are not perfectly clear.

Each original image was used to generate two additional augmented samples, effectively tripling the dataset size. This expanded dataset exposed the model to a wider range of scenarios, enabling it to learn more robust visual features such as edges, textures, and structural outlines, which is crucial for detecting fine defects like rust and cracks across varied field conditions.

3.2 Train/Test Split

The final processed dataset consisted of 3,044 tiled and augmented images, which was divided into training, validation, and test sets following an 88/6/6 split, closely approximating a standard 90/5/5 ratio.

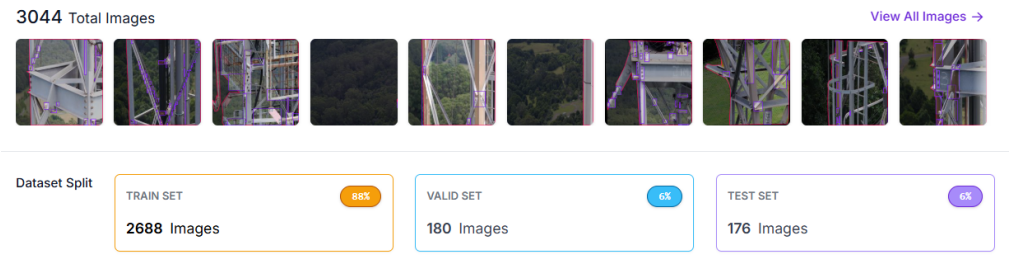


Figure 10: Roboflow Dataset with 3044 Rust tower images 88/6/6

A high proportion of training data, nearly 90%, was deliberately allocated to maximize the model's exposure to diverse rust features. Given the limited dataset size, this allocation helped the model learn to recognize subtle and varied patterns of corrosion more effectively, which is critical for defect detection tasks where visual cues can be minimal or inconsistent.
The remaining 6% of the dataset was assigned to both the validation and test sets. This relatively small allocation reduced the overall model development time, aligning with the project's goal of producing a functional prototype within a constrained research timeframe. As a trade-off, the reliability and statistical validity of performance evaluation were somewhat limited due to the smaller sample sizes in the validation and testing phases.
Ultimately, the 88/6/6 split was chosen to prioritize efficient learning of defect features, particularly rust, over strict evaluation fidelity, reflecting the exploratory and time-sensitive nature of this project.

3.3 Training Model

We adopted an iterative development approach to train, test, and refine four structural defect detection models: YOLOv5m, YOLOv5m-seg, YOLOv8m, and YOLOv8m-seg. Each iteration followed a cycle of design, implementation, verification, and evaluation, enabling continuous refinement of both the models and dataset configurations.
Across nine iterative cycles, we experimented with different dataset versions, each featuring adjusted preprocessing or augmentation strategies. For example, we compared the effects of 2×2 tiling versus 5×4 tiling, and tested changes in augmentation parameters such as brightness variation range. In each cycle, we evaluated the performance impacts of these modifications, retaining changes that improved generalization or dropping those that led to overfitting or reduced accuracy.

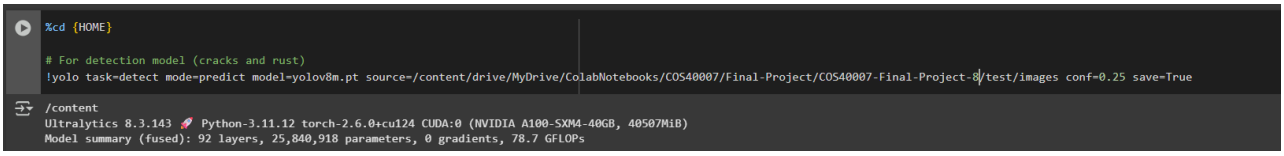


Figure 11: Yolov8 bounding box training code

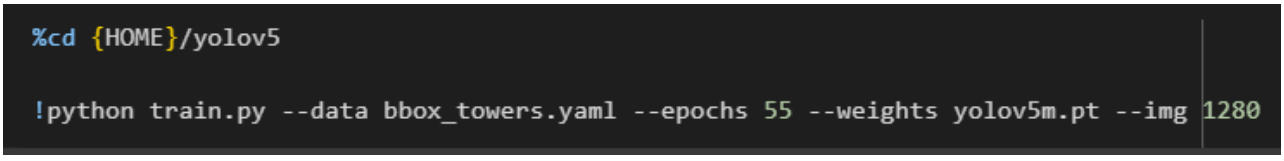


Figure 12: Yolov5 bounding box training code

Figure 13: Yolov8 segmentation training code

```
[ ]  %cd {HOME}/yolov5

     !python segment/train.py --data seg_towers.yaml --epochs 55 --weights segment/yolov5m-seg.pt
```

Figure 14: Yolov5 segmentation training code

Within each training loop, we applied manual hyperparameter tuning, particularly focusing on the number of epochs, batch size, and input image resolution. For each model, training was initially run for up to 100 epochs, but we observed that all four models began to overfit after approximately 50 epochs. Therefore, the most generalizable model weights were consistently obtained between epochs 40–50. We used both validation loss and mAP50 as the primary indicators of model performance during these evaluations.

In the case of the bounding box models (YOLOv5m and YOLOv8m), we modified the default image resolution from 640×640 to 1280×1280. This adjustment was necessary because rust occupies a very small portion of the high-resolution tower images. Increasing the input resolution allowed the models to retain finer visual detail, improving the detection of small, rust-affected areas. Conversely, for the segmentation models (YOLOv5m-seg and YOLOv8m-seg), we maintained the default resolution of 640×640, as the tower structure spans a significant portion of each image. Thus, downscaling did not negatively affect performance or context preservation.

Iterative Development: YOLOv8 Bounding Box Model Variants

One of the core challenges in training bounding box models was the difficulty in detecting small defects such as rust patches. These features often occupy only a tiny fraction of the total image area, especially given the high-resolution (5K) images used in this project. To address this, we adopted a tiling approach that splits each image into smaller tiles (e.g., 5×4 or 2×2 grid), allowing the model to focus on fine-grained detail without exceeding GPU memory limits.

We conducted three key training experiments using YOLOv8m to evaluate the effects of different tiling strategies, input resolutions, and pretrained weight initialization. A summary of the experimental configurations and their mAP50 results is shown below:

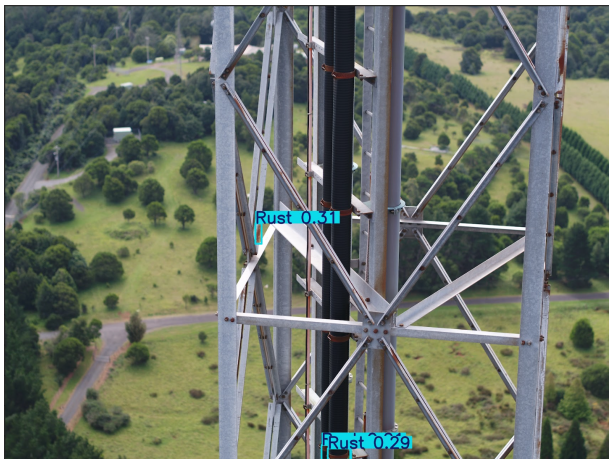
Iter.	Tiling	Res.	Pretrained	Folder	mAP50
1	5×4	1024p	No	trainfortileimages	0.07896
2	2×2	1280p	No	trainondefaultmodelwith1280img	0.06734
3	2×2	1280p	From Iteration 1	trainforlargeimagesonpretrainmodel	0.05397

Table 1: YOLOv8 bounding box training results with different tiling and resolutions

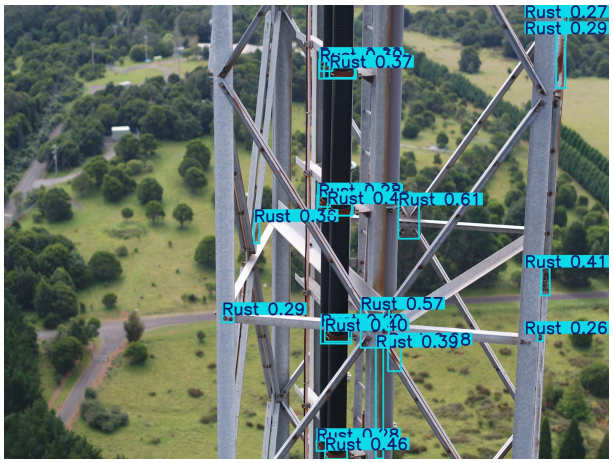
While iteration 1 achieved the highest mAP50 during training, it failed to generalize effectively to non-tiled, full-resolution test images. This was likely due to the model overfitting on zoomed-in tiles, leading to reduced performance on global context detection. In contrast, iteration 2, trained directly on 2×2 tiled images at 1280p resolution, demonstrated more consistent performance on unseen full images. Iteration 3, which fine-tuned the model from iteration 1 using larger images, underperformed due to the inherited overfitting bias toward zoomed-in tiles.

This iterative evaluation of tiling strategies and resolution settings helped inform our final model training approach, reinforcing the importance of maintaining consistency in training data structure and ensuring models are exposed to the types of images expected during inference.

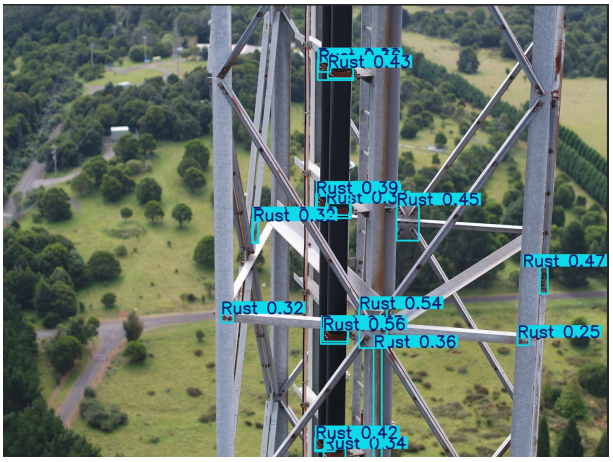
This comparison can be further illustrated by three predicted outputs from the three YOLOv8 models applied to the same original test image. Each model's prediction reflects how different training strategies affect the model's ability to detect small defects like rust.



(a) Prediction from Iteration 1



(b) Prediction from Iteration 2



(c) Prediction from Iteration 3

Figure 15: Predicted outputs from three YOLOv8 models on the same test image. Each result highlights how different tiling and training strategies affect detection accuracy and generalization.

Based on both the visual inspection of predictions and the model's ability to generalize to unseen, non-tiled test images, we selected the weights from *Solution 2* as the final YOLOv8 bounding box model. Although Solution 1 achieved the highest mAP50 during training, it failed to perform reliably on full-scale images. In contrast, Solution 2 provided a more balanced trade-off between training performance and real-world applicability.

3.4 Evaluation of Best AI Model

We adopted *mean Average Precision (mAP@0.5)* as the primary evaluation metric for model comparison, instead of Intersection over Union (IoU). This choice is particularly suited for our task of detecting small, obliquely oriented rust bars, where precise alignment is challenging and minor bounding box offsets can disproportionately reduce IoU.

Unlike IoU, which measures overlap on a per-instance basis, mAP provides a dataset-wide evaluation by averaging precision and recall across all classes and confidence thresholds. It also accounts for false positives and false negatives, making it more robust in scenarios with class imbalance and varying object sizes.

As mAP is the standard benchmark metric in object detection (e.g., COCO, PASCAL VOC), it ensures consistency with best practices and facilitates meaningful comparison across YOLO model variants. Therefore, all evaluation results in this section are reported using mAP@0.5, along with supporting precision and recall metrics.

mAP averages performance across all classes and accounts for false positives and false negatives. In our case, where rust might be rare or appear in varying sizes and angles, this makes mAP a more reliable indicator of real-world performance.

Bounding Box Detection Comparison

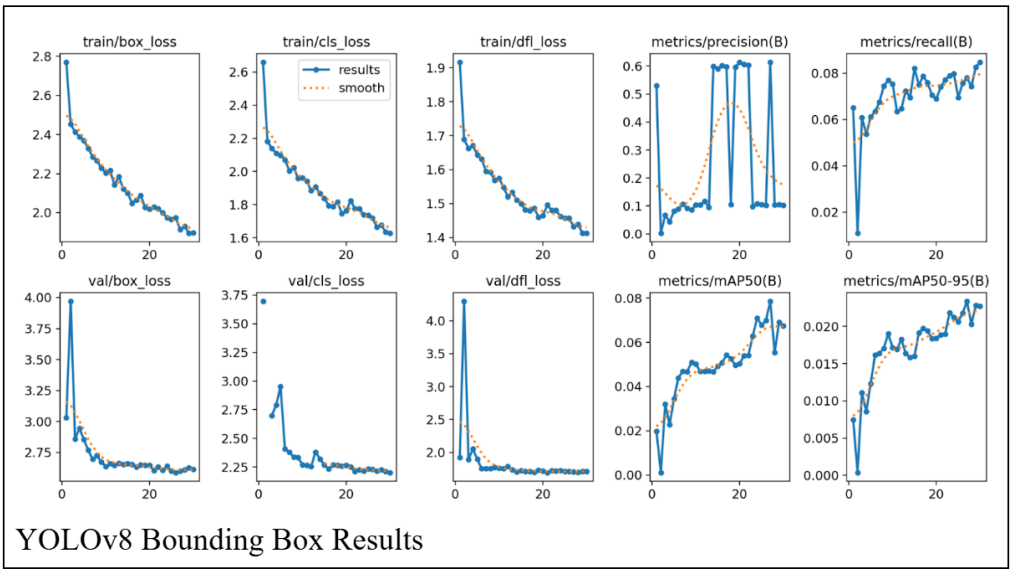


Figure 16: YOLOv8 Bounding box result

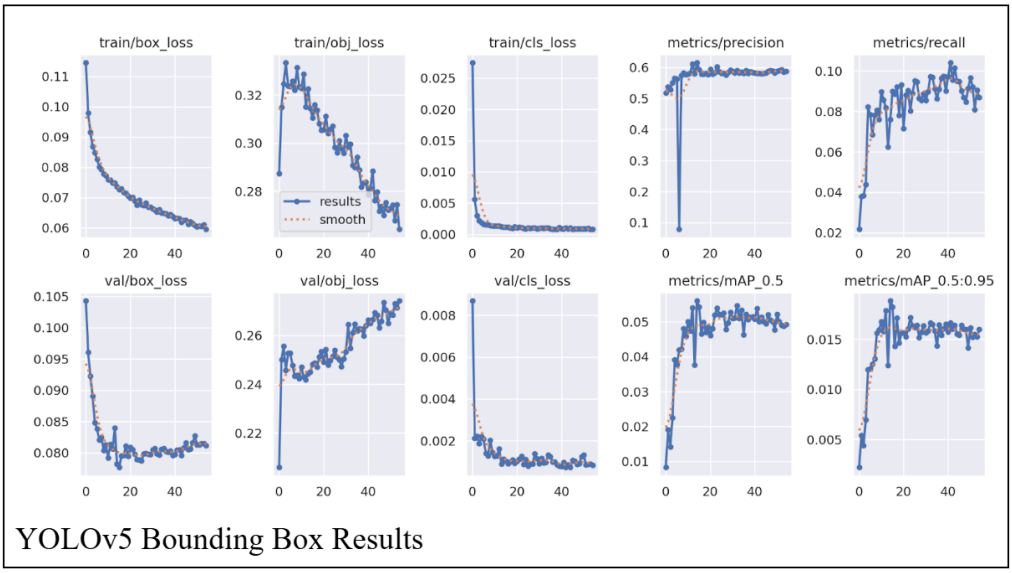


Figure 17: YOLOv5 Bounding box result

Model	Precision	Recall	mAP50
YOLOv8m	0.686	0.179	0.065
YOLOv5m	0.615	0.152	0.052

Table 2: Bounding box performance comparison between YOLOv8m and YOLOv5m

In the bounding box task for detecting rust and cracks, YOLOv8m achieved higher precision, recall, and mAP50 than YOLOv5m. Although the absolute performance values are relatively low due to the small and imbalanced dataset, YOLOv8m demonstrated better sensitivity to small objects. This improvement is likely attributed to architectural enhancements such as Distributed Focal Loss in YOLOv8.

Segmentation Model Comparison

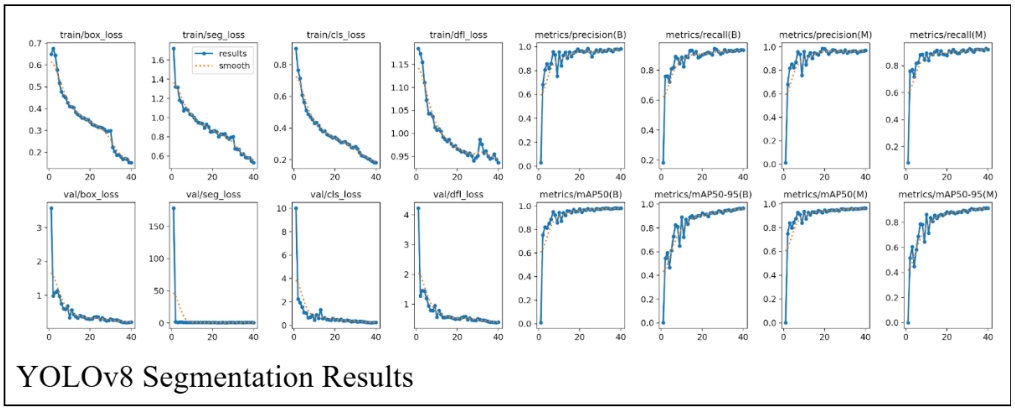


Figure 18: YOLOv8 Segmentation result

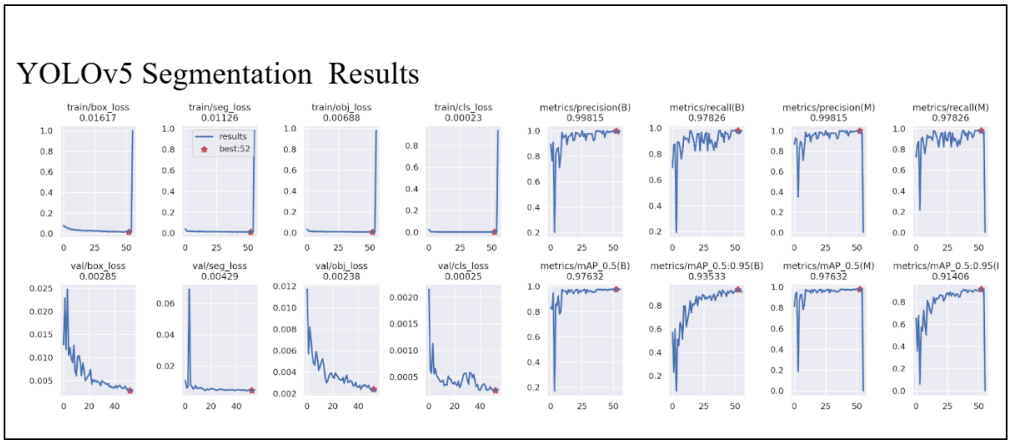


Figure 19: YOLOv5 Segmentation result

Model	Precision	Recall	mAP50
YOLOv8m-seg	0.966	0.947	0.989
YOLOv5m-seg	0.998	0.978	0.976

Table 3: Segmentation performance comparison between YOLOv8m-seg and YOLOv5m-seg

For segmentation of the full tower structure, both models performed extremely well, with mAP50 values approaching 1.00. YOLOv5m-seg achieved slightly higher precision and recall, while YOLOv8m-seg produced a marginally better mAP50. This shows that both models are highly capable of segmenting large, well-defined structures. However, the differences are minor, and the choice between models may come down to training time or architectural preference.

Based on the evaluation metrics, we selected *YOLOv8m* as the final model for both the bounding box and segmentation tasks. For bounding box detection, YOLOv8m demonstrated superior performance in precision, recall, and mAP50 compared to YOLOv5m, particularly in detecting small, hard-to-spot defects like rust. Although YOLOv5m-seg slightly outperformed YOLOv8m-seg in precision and recall, YOLOv8m-seg achieved a marginally higher mAP50, indicating more accurate segmentation overlap with ground truth. Considering these trade-offs, and YOLOv8's architectural advantages such as improved loss functions and future maintainability, YOLOv8 was chosen as the unified framework for both tasks in our final deployment pipeline.

4 AI Demonstrator

Our AI demonstration is a website application that is built using Streamlit to showcase the practical implementation of our structural defect detection models. Our application provides a comprehensive interface for users to interact with our trained YOLOv8, both bounding box and segmentation models, and analyze structural defects across different input formats.

4.1 Development Architecture

The AI demonstrator website follows the practice of modular architecture, which consists of several key components:

Prediction Classes

- ImageModelPredictor: Handles image analysis with batch processing.
- VideoModelPredictor: Analyze defects inside the given video files with the inputted frame-by-frame configuration.
- LiveModelPredictor: Manages real-time webcam feeds for live detection.

Model Integration

This application uses the trained YOLOv8 models for the specific scenario of structural defect detection. The models are loaded through two classes that implement the IModelLoader interface, which are BoundingBoxModelLoader for bounding box model and SegmentationModelLoader for segmentation model. These two model loaders ensure the initialization process of the models runs smoothly and maintain consistent prediction formatting across all input types.

User Interface Design

The application features a clean, professional interface with a main navigation page and three sub-pages that correlate to each specialized prediction. The provided sidebar navigation and easy file-based routing from Streamlit allows seamless switching between different analysis modes while maintaining session state and prediction history throughout the application.

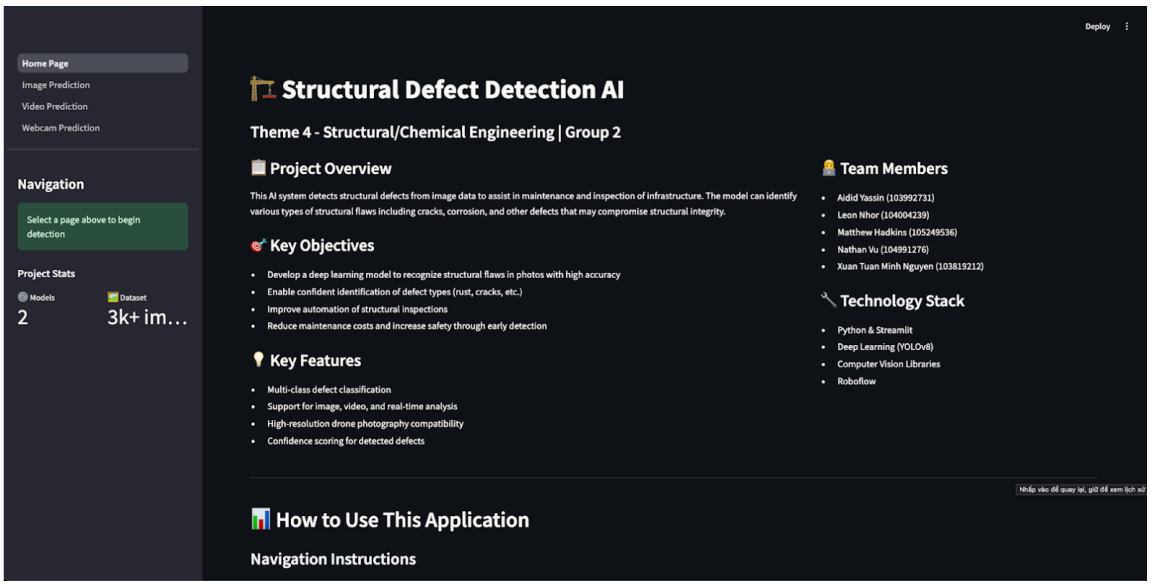


Figure 20: Main User Interface Design using Streamlit

4.2 Input Requirements and Capabilities

Our application accepts three different types of input, each are implemented for different specific inspection cases:

Image-based Prediction Page

Users can upload images in JPG, PNG, or JPEG format for detailed structural analysis. Our system supports high-resolution images, making it compatible with drone photography and detailed close-up inspections. Users can apply various filters to focus on defect types (Rust, Cracks, Tower Structures) and adjust confidence thresholds to control detection sensitivity.

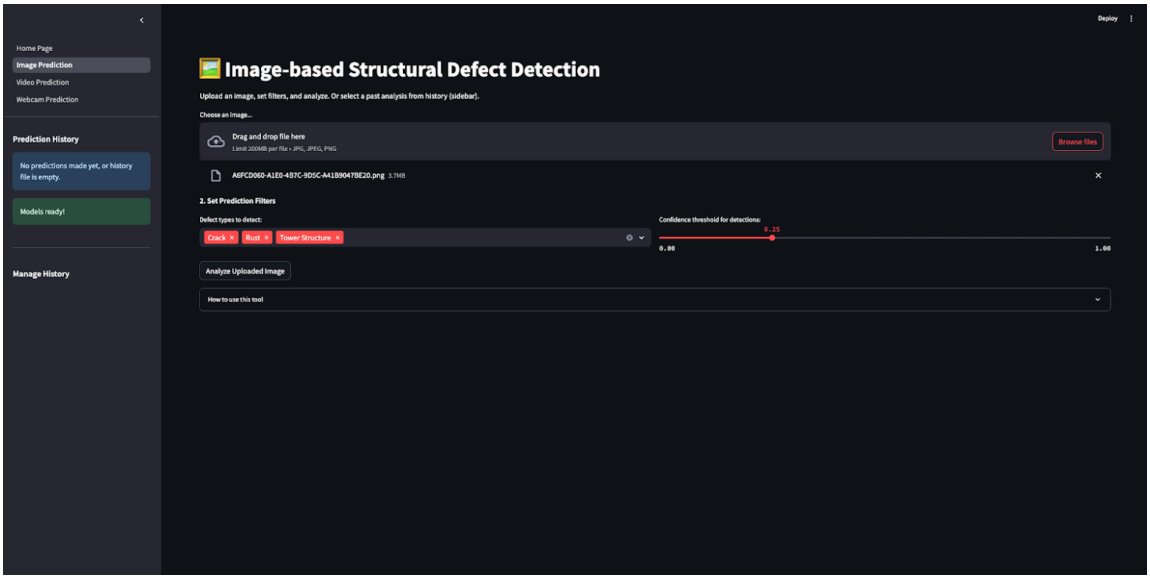


Figure 21: Image-based Prediction Page with Various Filters

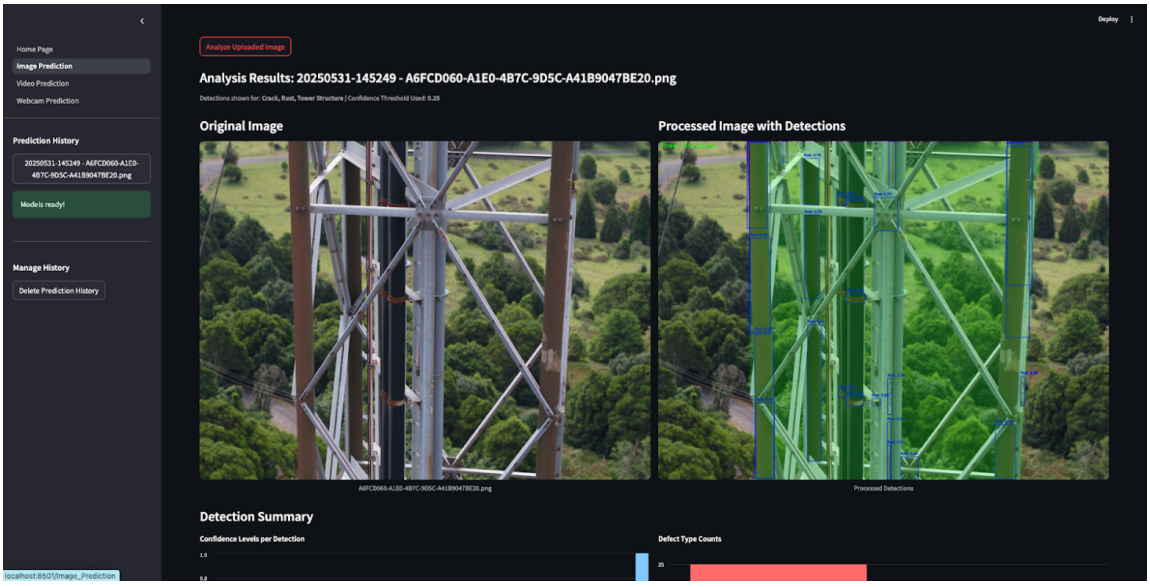


Figure 22: Image-based Prediction Page after Analyzing the Image

Figure 23: Detection Summary of the Predicted Image

Video-based Prediction Page

The video-based prediction page processes uploaded video files with the required frame-by-frame configuration, creating a comprehensive structural conditions analysis. This page is particularly valuable for analyzing video footage from drones or handheld cameras during structural inspection sessions. Under the hood, the system will save both original and processed videos under the data/ directory, which helps in persisting the audit log of the analysis.

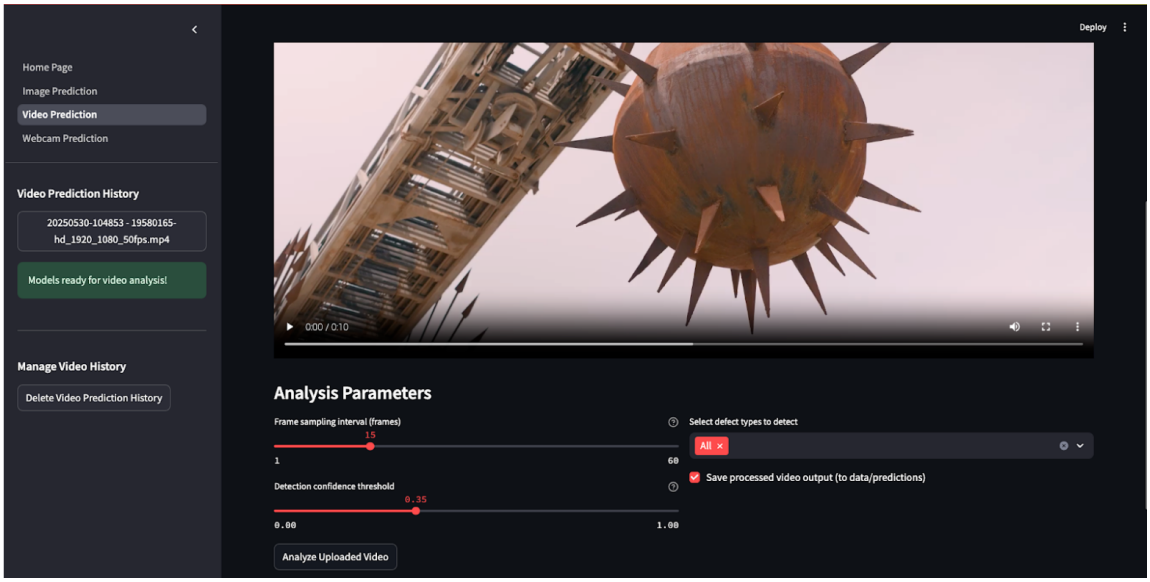


Figure 24: Video-based Prediction Page with Various Filters

Figure 25: Defected Summary of the Predicted Video

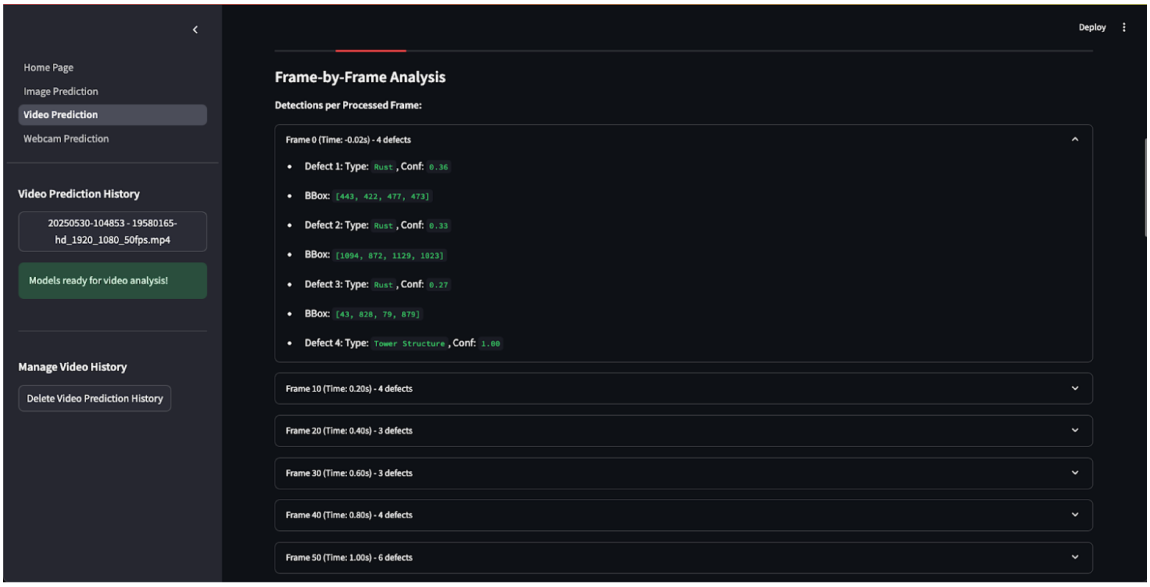


Figure 26: Frame-by-Frame Analysis of the Predicted Video

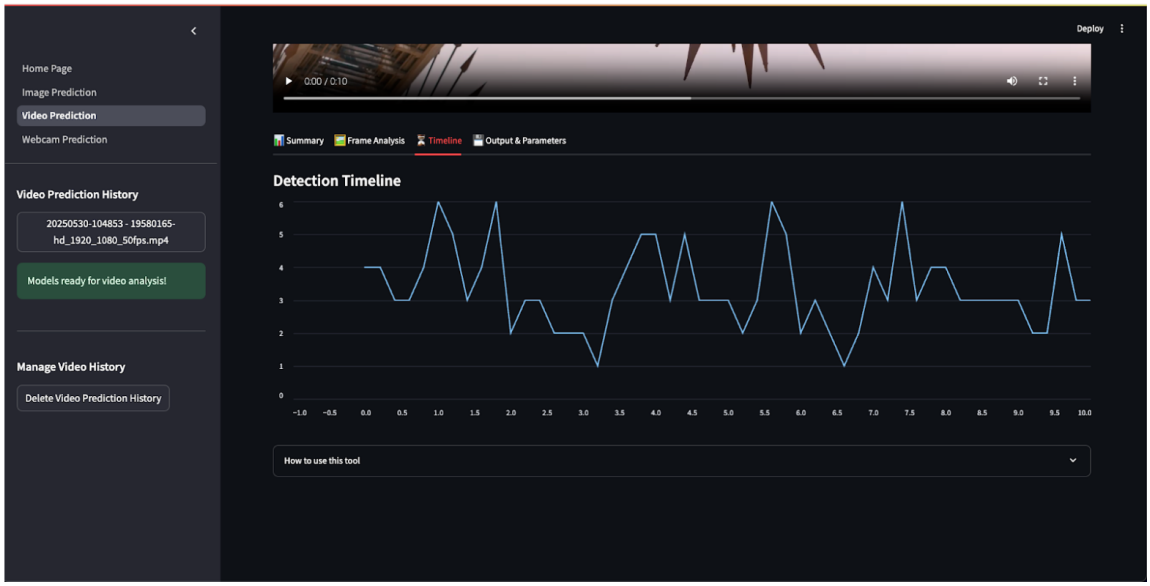


Figure 27: Detection Timeline of the Predicted Video

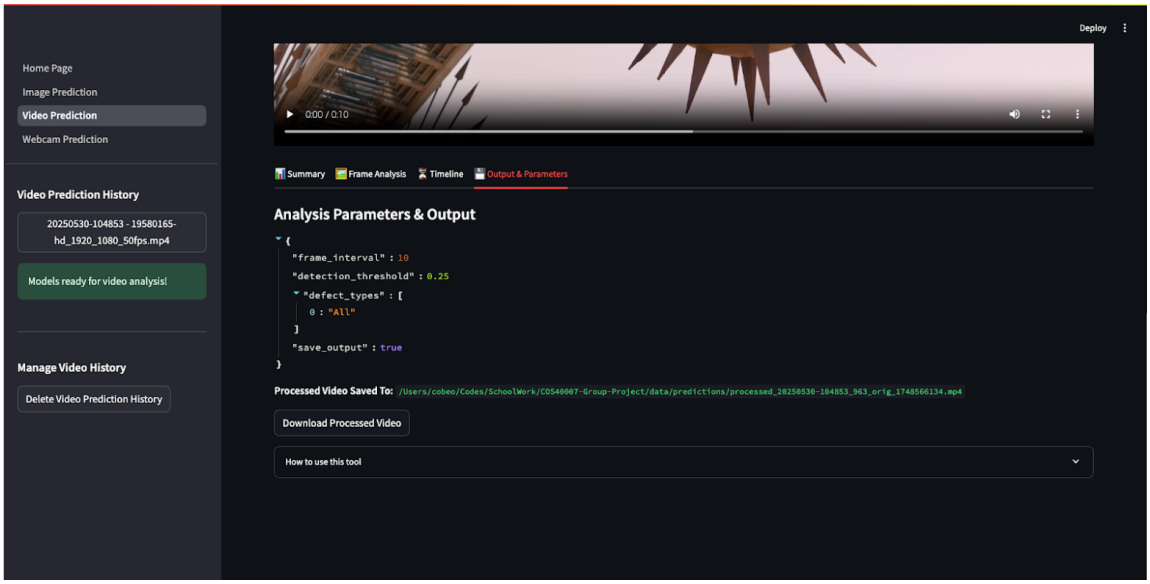


Figure 28: Analysis Parameters and Download Processed Video

Real-time Webcam Prediction Page

The Real-time Webcam Prediction page capable of analyzing structural defection in real-time using device cameras. This feature utilizes smartphones or tablets as portable inspection tools, allowing engineers to get immediate analysis during on-site assessments. The system automatically captures and saves frames containing detected defects for later review.

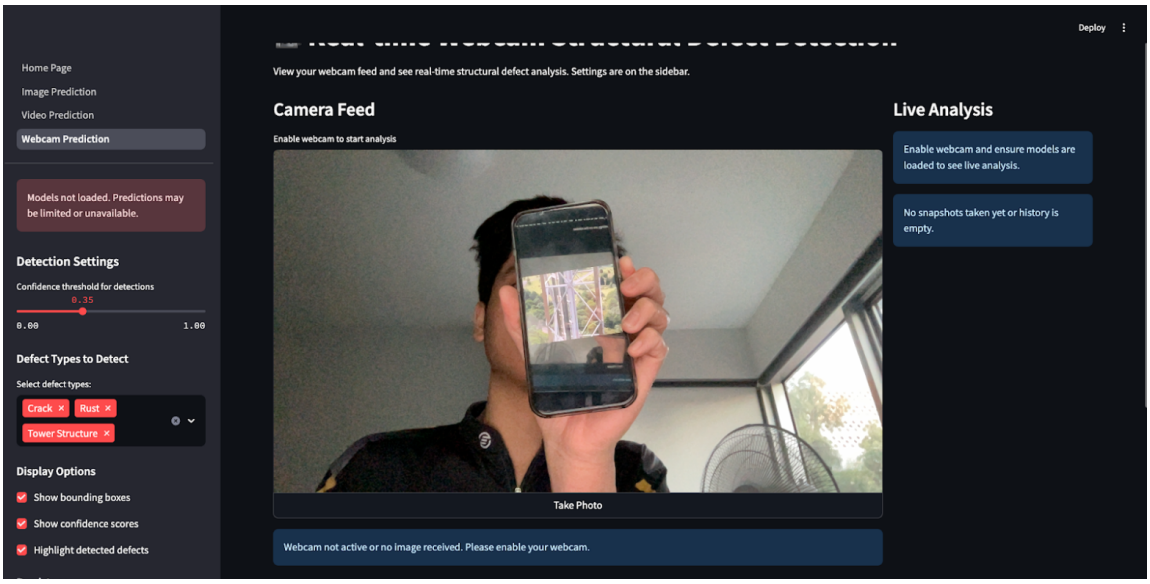


Figure 29: Real-time Webcam Prediction Page with Various Filters

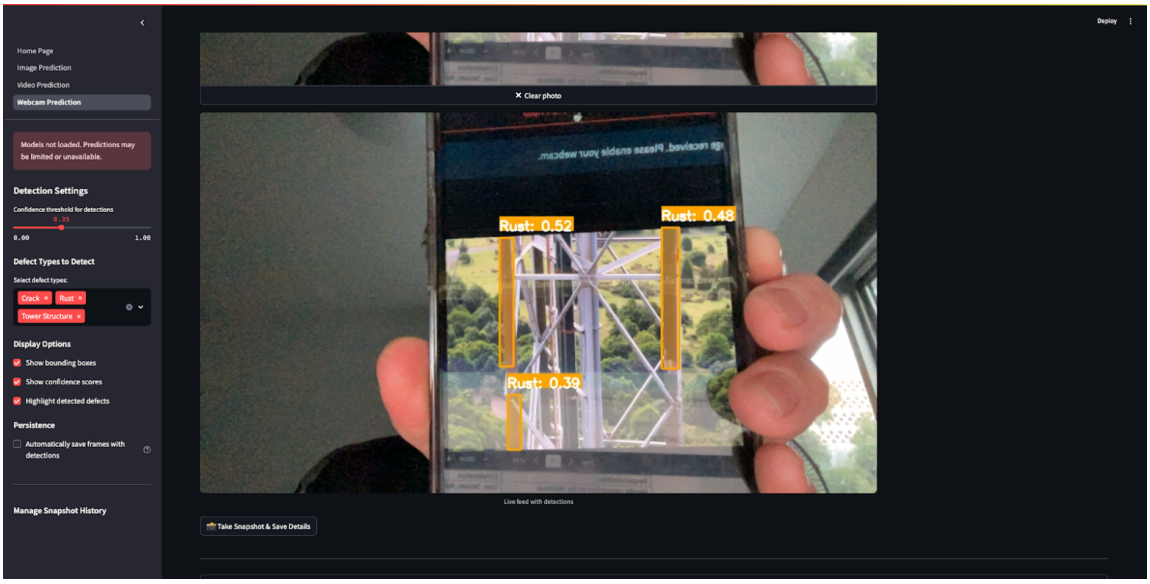


Figure 30: Real-time Webcam Prediction Page after Analyzing the Snapshot

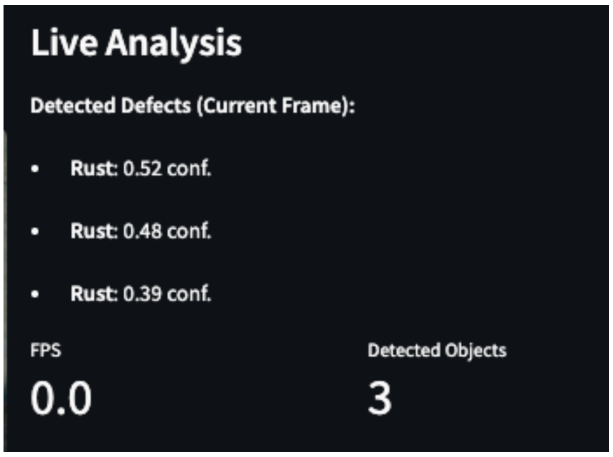


Figure 31: Live Analysis of the Snapshot

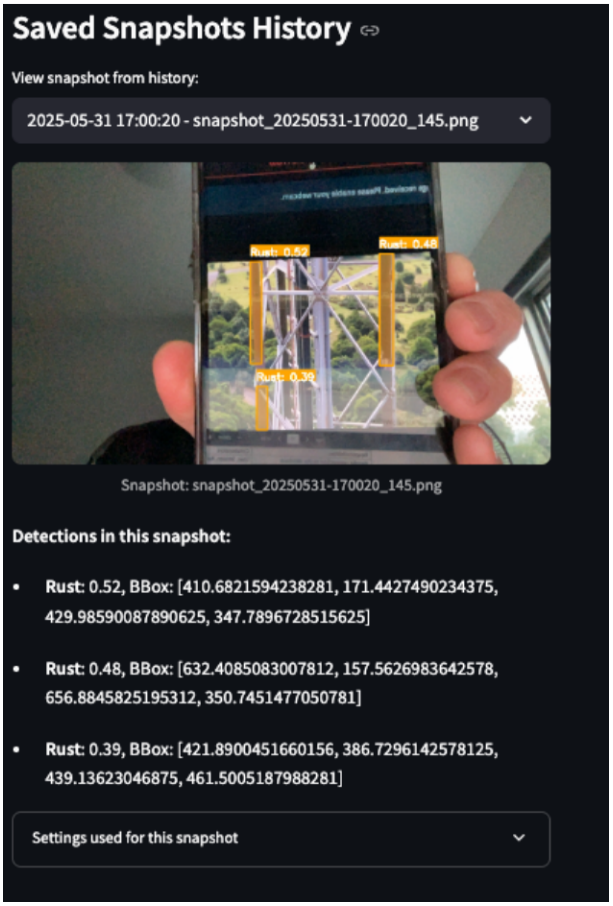


Figure 32: Saved Snapshot with Rust Analysis

4.3 Output and Results

The Structural Defect Detection application provides comprehensive analysis results across all input modalities:

Visual Detection Results

- Annotated images/videos with bounding boxes highlighting detected defects and tower structure.

- Color-coded classifications for different defect types and tower structure.
- Confidence scores for each detection.
- Real-time processing feedback and status indicators.

Quantitative Analysis

- Detection summaries for various defect types.
- Confidence level distribution analysis and metrics.
- Scrollable list of detailed detection summaries.

Data Management and History

- Access to saved images, videos, and snapshots with analysis.
- Persistent prediction histories across all modules.
- Data stored in JSON format with corresponding media for long-term monitoring.

Export and Documentation

- All processed outputs are saved in the data/ folder.
- Metadata logged in JSON files for future reference.

5 Conclusion

This structural defect detection project delivers a comprehensive AI-powered system capable of identifying rust, cracks, and tower structures across multiple input types and prediction modalities. The final system integrates two YOLOv8 models, one trained for bounding box-based detection of rust and cracks, and the other for polygonal segmentation of the full tower structure. These models were trained on a carefully processed dataset of high-resolution drone imagery, and deployed through a robust Streamlit-based web application supporting image analysis, video processing, and real-time webcam detection.
The system demonstrates the practical feasibility of using deep learning for infrastructure inspection, enabling interactive, multi-format predictions with high-resolution support and customizable confidence filtering. Through its modular architecture and real-time feedback capabilities, the application represents a fully functional prototype suitable for future adaptation and scaling.

5.1 Challenges and Solutions

Data Quality and Consistency

One of the most significant challenges during data processing was ensuring consistent annotation practices across the team. Initially, each member followed slightly different labeling conventions, which introduced inconsistencies in the dataset, particularly in complex cases such as horizontally positioned rust bars. This issue was addressed by establishing a standardized labeling procedure and encouraging the use of smaller, more precise bounding boxes instead of broader, less informative annotations.

Data Processing and Model Performance

A major technical challenge was optimizing image resolution and tiling strategy to balance detail preservation with computational efficiency. The original dataset consisted of 5K-resolution images, and early attempts to resize them to 640×640 pixels resulted in the loss of small rust features. Through iterative testing, we discovered that heavy tiling (5×4) on high-resolution images exceeded GPU memory limits and halted training entirely. The breakthrough came with a 2×2 tiling strategy, resizing each tile to 1280×1280 pixels, which preserved both fine-grained defect features and contextual structure, leading to improved mAP performance. Attempts to create hybrid models using varied tiling strategies resulted in training instability and reduced accuracy, reinforcing the importance of consistent data preprocessing pipelines.

Infrastructure and Format Compatibility

Hardware limitations also posed a barrier to experimentation and scalability. Free-tier GPUs on Google Colab and Kaggle were insufficient for training on our optimized dataset. To resolve this, we upgraded to Google Colab Pro with an A100 40GB GPU, which significantly reduced training time and enabled stable model development. Additionally, Roboflow's label format incompatibility between detection and segmentation models necessitated the development of a custom Python script to convert the unified annotation dataset into separate, model-compatible formats.

User Interface Development Challenges

The user interface component presented both strategic and technical hurdles. Initially, the project aimed to use React and FastAPI for frontend and backend development. However, due to time constraints and the complexity of managing multiple prediction modalities, we transitioned to Streamlit for faster prototyping and deployment. This shift enabled rapid development but introduced new UI challenges, including managing session state across image, video, and webcam modules, inconsistent filter visibility, and prediction history persistence.

We resolved these issues through improved rendering logic, refactored state handling, and a JSON-based storage system to maintain consistency across sessions. The webcam module posed the greatest challenge, requiring seamless real-time detection while preserving past snapshots and allowing intuitive user interaction. Despite these obstacles, Streamlit's flexibility enabled us to deliver a fully functional interface within the project timeline.

5.2 Key Learnings

This project has provided hands-on insight into the end-to-end machine learning development cycle. Our experimentation with tiling and image resolution emphasized how subtle preprocessing decisions can significantly impact model performance. We learned that preserving visual detail while retaining structural context demands a careful balance, often influenced by GPU constraints and architectural considerations.

We also gained a deeper understanding of the importance of annotation consistency and team coordination in dataset preparation. Iterative model training, combined with metric-driven validation, reinforced the need for structured experimentation to reach optimal results.

From a deployment perspective, we learned valuable lessons in user interface design, error handling, and state management in real-time systems. The integration of persistent data management with real-time prediction capabilities revealed the complexity of building production-ready AI tools.

Ultimately, this project highlighted that a successful AI implementation requires more than high-performing models. It also depends on thoughtful UI design, robust data pipelines, and system architecture decisions that can scale with future applications.

6 Appendix A

Source code/notebooks of the project

- YoloV5M Source Code: Click Here.
- YoloV8M Source Code: Click Here.
- User Interface Source Code: Click Here.

Intermediate data files and final model file

- Labeled dataset with annotations, features, and splited for bounding box and segmentation: Click Here.
- YoloV5M Bounding Box and Segmentation Models: Click Here.
- Roboflow: Click Here.
- YoloV8M Bounding Box and Segmentation Models: Click Here.
- YoloV8M Training with Three Iterations: Click Here.
- Final Model Used for Demonstration: Click Here.

A 1-page document/short video that describes how to run AI demonstrator with some sample inputs.

- Installation and Usage Document: Click Here.
- Demonstration Video: Click Here.

References

[1] Jonathan Hui. IoU — A Better Detection Evaluation Metric. https://towardsdatascience.com/iou-a-better-detection-evaluation-metric-45a511185be1/, 2019.

[2] Paul Guerrie. How to train yolov5 instance segmentation on a custom dataset. *Roboflow Blog*, Sep 2024.

[3] wkentaro. Labelmeai/toolkit: Tools to streamline dataset creation with labelme., Jan 2025.

[4] Silong Peng Yue Guo Junchi Yan Wen Qian, Xue Yang. Learning modulated loss for rotated object detection. *ariv*, unknown.

[5] A. Data. Everything you need to know about bounding boxes - aya data, 2025. Accessed: 2025-06-01.

[6] Nicolai Nielsen. How to auto label your custom dataset with roboflow in 2 minutes, 2024.

[7] Roboflow. Tackling the small object problem in object detection, 2020.

[8] Roboflow. How to train yolov5 instance segmentation with custom data, 2022.

[9] Roboflow team. Yolov8: How to train for object detection on a custom dataset, 2023.

[10] Ultralytics. *YOLOv8 Documentation*, 2023.

[11] J. Torres. How to evaluate yolov8 model: A comprehensive guide, 2024.

[12] Roboflow. How to train yolov5 instance segmentation on a custom dataset, 2022. Accessed: 2025-06-01.

[13] robguinness. Is there a rule-of-thumb for how to divide a dataset into training and validation sets?, 2012.