Web-Based Machine Learning Application for Automated Railroad Defect Detection

Hwapyeong Song*, Andrew d'Arms*, Aayush Damai*, Van Trung Le*, Shiyu Liu†, Dylan Lester†,
Husnu S. Narman*, Ammar Alzarrad‡, Pingping Zhu†
* Department of Computer Science
† Department of Biomedical and Electrical Engineering
‡ Department of Civil Engineering
Marshall University, Huntington, WV, USA
{song24, darms1, damai2, le57, liu137, lester299, narman, alzarrad, zhup}@marshall.edu

***Abstract*—Railroad track defects pose significant safety and operational risks, necessitating timely and reliable inspection solutions. This study introduces a browser-accessible visual analytics system that leverages machine learning to automatically identify railroad infrastructure defects from image and video data. The proposed Railroad Visual Detection Application integrates multiple pre-trained object detection models within a unified web framework, allowing non-technical users to submit visual inputs and obtain defect identification results without requiring programming expertise. The system supports recognition of four representative defect categories: joint bar cracks, missing bolts, track gauge deviations, and insufficient ballast. For each detected anomaly, the platform reports the defect class, corresponding timestamp, spatial location, and associated confidence score. In addition to system design and implementation, the study examines computational performance under realistic deployment conditions by comparing CPU- and GPU-based inference across single- and multi-stream processing scenarios. Experimental results demonstrate that GPU acceleration significantly reduces per-frame inference latency for single-stream workloads, while CPU execution scales more effectively under increased parallelism due to multi-core utilization. These results underscore the complementary advantages of heterogeneous computing resources and suggest that hybrid execution strategies can further enhance operational efficiency. Overall, the proposed framework offers a scalable and user-friendly solution for automated visual inspection in real-world railroad maintenance workflows.**

***Index Terms*—Railroad Defect Detection, Machine Learning, Railroad Inspection Web Application, Object Detection, GPU Acceleration**

I. Introduction

Railroad transportation is a critical component of modern infrastructure, supporting the movement of passengers and essential goods across long distances. Defects in train tracks are a major cause of derailments, which can result in severe safety hazards, loss of life, and substantial economic disruption. To reduce these risks, track defects must be identified and repaired in a timely manner before they evolve into catastrophic failures. Traditionally, railroad defect detection has relied on manual inspection, where trained personnel visually examine railway infrastructure on site. Although widely adopted, this approach raises significant concerns related to inspector safety, inspection consistency, and detection accuracy, particularly when defects are subtle, inspections are conducted at scale, or environmental conditions are unfavorable.

979-8-3315-4568-0/26/$31.00 ©2026 IEEE

Recent advances in Artificial Intelligence (AI) have enabled more reliable and scalable approaches for railroad defect detection. In particular, object detection, which is a fundamental task in computer vision, allows AI models to identify and localize objects of interest by learning from annotated datasets. Object detection has been successfully applied in domains such as autonomous driving, video surveillance, and healthcare [1]. Within the railway domain, prior studies have demonstrated that object detection models can achieve high detection accuracy for a range of defect types, including track gauge deviations [2], missing track bolts [3], and structural anomalies, as well as distinguishing between visually similar conditions such as cracks and gap [4]. Despite these advances, widespread adoption remains limited due to the lack of standardized interfaces and deployment frameworks. Object detection models are often developed in isolation and require specialized technical expertise to configure and deploy, making their integration into a unified and accessible web application challenging.

To address this challenge, this research presents a web based application architecture that standardizes diverse object detection models within a single framework, enabling multiple railroad defect detection tasks to be accessed through one interface. The proposed application bridges the gap between advanced machine learning research and practical field deployment by providing a code free and user friendly environment for video based railroad defect detection. Users can upload image or video data, select defect categories of interest, and receive automated detection results without requiring programming knowledge. For each detected defect, the application reports the defect type, timestamp of occurrence, model confidence score, and spatial coordinates within the image or video frame. By lowering technical barriers and simplifying access to AI based inspection tools, the proposed application facilitates broader adoption by non technical users and supports more efficient and scalable railroad maintenance operations, ultimately contributing to improved railway safety.

A. Research Objectives and Contributions

The primary *objective* of this research is to develop an accessible web based railroad defect detection application that leverages Machine Learning (ML) to enhance railway safety. The application is designed with a strong emphasis

on usability, enabling non technical users to upload image or video data and automatically detect railroad defects while receiving structured outputs that include defect type, timestamp, spatial location, and model confidence score.

Beyond application development, this work makes the following key *contributions*:

- Unified web based detection application: A modular web application that integrates multiple heterogeneous machine learning models for railroad defect detection within a standardized and extensible backend architecture.
- Code free access for non technical users: A user friendly application interface that eliminates the need for programming expertise, lowering the barrier to adopting AI driven inspection tools in railroad maintenance workflows.
- Application level performance evaluation: A comprehensive performance analysis of CPU and GPU based execution under single and multi stream configurations, quantifying inference latency and normalized per stream execution time to assess scalability and deployment efficiency in practical settings.

A secondary objective is to provide actionable insights into hardware selection and parallelization strategies for real time video based railroad inspection. By jointly addressing usability, computational efficiency, and deployment considerations, this study aims to deliver a scalable and practical application suitable for real world railroad safety inspection scenarios.

B. Evolution of Railway Defect Detection Technologies

Since the introduction of railroads, maintaining infrastructure integrity and ensuring reliable operation have been persistent challenges. As inspection technologies have evolved, railroad defect detection has transitioned from labor intensive manual inspections to advanced Machine Vision Based Inspection Systems (MVIS) [5]. Traditional inspection approaches rely on technicians visually examining tracks and rolling stock, often at speeds below 18 mph. These methods are time consuming, prone to human error, and inherently constrained by the limitations of human perception. To overcome these challenges, modern MVIS incorporate advanced sensing technologies and artificial intelligence based analysis. High resolution Charge Coupled Device (CCD) cameras and Light Detection and Ranging (LiDAR) sensors are commonly employed to capture detailed visual and spatial information. Deep learning models such as Convolutional Neural Networks (CNNs) and You Only Look Once (YOLO) enable high speed, real time defect detection with significantly improved accuracy and consistency. Although these inspection applications have demonstrated strong performance, they are often expensive, complex to deploy, and dependent on heterogeneous sensor data. Commonly used sensors include Fiber Bragg Grating (FBG) sensors for vibration analysis, inertial measurement units (IMUs) for track geometry assessment, and ultrasonic sensors for internal fracture detection. Integrating these diverse data sources increases complexity and limits accessibility, motivating the development of streamlined vision based inspection applications.

C. Integration of Machine Learning in Web Based Applications

The increasing adoption of AI and ML technologies has accelerated their integration into web based applications across numerous domains. One of the most common implementations is the use of AI powered chatbots to enhance user interaction and accessibility. A comprehensive review [6] demonstrates the feasibility and effectiveness of integrating ML models into web applications using platforms such as Node.js and Python, while also examining their interaction with web application security mechanisms, including web application firewalls.

The review highlights that Python based frameworks such as Django and Flask provide robust support for ML integration due to their extensive ecosystem of AI and data processing libraries. In contrast, Node.js offers a more limited set of ML focused tools, which can constrain the scope and flexibility of AI driven web applications. These findings motivate the adoption of backend architectures that leverage Python based ML frameworks while maintaining scalable, responsive, and user friendly web application interfaces.

D. Paper Organization

The remainder of this paper is organized as follows. SectionII presents an overview of the proposed application architecture. Section III describes the implementation details, including datasets, model training procedures, backend workflow, and front end integration. Section IV presents the performance evaluation and experimental results. Section V discusses key findings and limitations of the proposed application. Finally, Section VI presents the conclusion and suggests directions for future research.

II. Overall Architecture

The proposed application integrates multiple machine learning models to enable automated detection of railroad defects from image and video data. The architecture follows a modular design that separates data acquisition, preprocessing, and inference into distinct components. This modularity enables flexible evaluation of models, simplifies integration of additional defect detection capabilities, and supports scalable deployment within a web based application environment.

A. Data Acquisition and Preprocessing Module

Image data for railroad defect detection were collected from multiple publicly available datasets, primarily sourced from Kaggle and Roboflow such as [7]–[10]. These datasets cover a diverse range of defect types relevant to railroad maintenance applications. To improve dataset quality and reduce the risk of model overfitting, duplicate and near duplicate samples were removed prior to training. In addition to publicly available data, supplemental images were acquired through field testing and extracted from recorded video sequences or captured directly as photographs. This combination of curated public data and field collected samples improves coverage of real world operating conditions.

All images were subjected to data augmentation operations to enhance model generalization. Augmentation

techniques include variations in color saturation, exposure adjustment, horizontal flipping, and the introduction of noise and grain artifacts. These transformations increase dataset diversity and improve model robustness to variations in lighting, viewpoint, and environmental conditions commonly encountered in railroad inspection scenarios.

For track gauge deviation detection, a dedicated preprocessing pipeline was implemented to extract four required data components from Intel RealSense bag files. These include color images stored in jpg format, depth images stored in grayscale png format, camera intrinsic parameters stored as text files, and serialized timestamp mappings. The pipeline operates in non real time playback mode and aligns depth frames with corresponding color frames. Color and depth images are provided as input to machine learning models, including YOLO and SAMURAI, for rail detection and tracking. Camera intrinsic parameters are used within a three dimensional reconstruction workflow to estimate the distance between detected rails. Extracted timestamps enable accurate annotation of detected gauge deviations and support later correlation with spatial location data such as GPS coordinates.

B. Web Application Framework

The application is implemented using the Laravel framework, which was selected for its robust support for database integration and clear separation between frontend and backend components. Although Laravel is PHP based, it supports seamless interoperability with Python based machine learning workflows, enabling execution of PyTorch models from within the application. The framework provides strong scalability and maintainability features, supported by comprehensive documentation and an active developer community. These characteristics make Laravel suitable for hosting an extensible machine learning enabled railroad inspection application.

C. Machine Learning Models for Railroad Defect Detection

The current web application supports detection of four primary railroad defect types: joint bar cracks, missing bolts, track gauge deviation, and insufficient ballast. Each defect category is handled by a dedicated machine learning model selected based on prior validation studies. YOLO [11] is employed for the majority of detection tasks due to its efficiency and accessibility through the Ultralytics implementation. A defining characteristic of the YOLO framework is its ability to perform object detection in a single forward pass of the input image. This design enables fast inference and makes YOLO well suited for near real time inspection applications. While multiple variants of YOLO have been developed, the core architecture consistently provides a favorable balance between detection accuracy and computational efficiency, supporting deployment in both CPU and GPU based execution environments.

III. Implementation

The Railroad Visual Detection Application (RVDA) is implemented as a web based railroad defect detection application that leverages machine learning algorithms for automated inspection. The application follows a client server architecture designed to support scalability and modular integration. The client component provides a browser accessible user interface, while the server component manages data processing and machine learning inference. Users access the application through a standard web browser, and the current deployment operates on an Apache server within a XAMPP environment. The application backend is developed using the PHP based Laravel framework, while MySQL serves as the relational database for managing input data, detection tasks, and results. Frontend content is rendered through Laravel Blade components, with HTML5, CSS, and JavaScript used to construct an interactive and responsive user interface. Backend processing logic is implemented in PHP, and machine learning inference is executed through Python scripts integrated into the application workflow. The XAMPP stack provides a stable local development environment for application testing and deployment.

A. Datasets

1) Joint Bar Cracks: The joint bar crack dataset used in this research was obtained from prior published work [4]. Both single class and combined datasets were employed. The single class dataset was introduced in [10] and consists of approximately one thousand images containing various crack and gap patterns. These images include a range of crack sizes and severities. A limitation of the single class formulation is that grouping multiple defect types into one class can reduce model discrimination capability and negatively impact generalization performance. To address this limitation, a combined dataset was constructed by integrating the single class dataset with additional publicly available datasets [7]–[9], [12]. The resulting dataset contained approximately two thousand images. Data augmentation techniques were then applied to expand the dataset to five thousand images. Augmentation operations include horizontal flipping, shear transformations up to ten degrees, and rotational transformations up to fifteen degrees. The final dataset was partitioned into training, validation, and testing subsets using a seventy twenty ten split.

2) Missing Bolts: The missing bolt dataset was obtained from previously published work [3]. The dataset was constructed by extracting and relabeling relevant images from several existing railroad defect datasets. Specifically, datasets containing three hundred eighty four images [12], one thousand seventy five images [7], three hundred eighty images [9], and seven hundred seventeen images [8] were examined. Images relevant to rail joint bars with missing bolts were manually relabeled and consolidated into a single dataset containing two hundred sixty four images. The dataset was divided into one hundred fifty two training images, thirty nine testing images, and seventy three validation images. True negative samples were not included in the current dataset, reflecting a focus on defect localization rather than binary defect presence classification.

To improve robustness and generalization, standard data augmentation techniques were applied to the missing bolts dataset. These augmentations introduce variations in orientation, geometry, color characteristics, and noise, allowing

the model to better handle differences in camera viewpoint, lighting conditions, and environmental disturbances encountered during real world railroad inspections.

3) Track Gauge Deviations: The track gauge deviation dataset used in this study was obtained from previously published work [2]. The original dataset consisted of one thousand seven hundred seventy one annotated images labeled with two classes, namely rail lines and ballast. Preprocessing included data augmentation techniques such as controlled exposure variation, Gaussian blur with a maximum kernel size of two point five pixels, and salt and pepper noise affecting up to zero point two two percent of pixels. After augmentation, the dataset contained four thousand nine hundred three images. These were split into training, validation, and testing subsets by following standard ratios.

4) Insufficient Ballast: The ballast sufficiency dataset consists of one hundred forty six labeled frames sampled from seven thousand nine hundred seventy seven train mounted RGB D video frames. A total of three hundred seventy four ballast regions were annotated, including two hundred four sufficient ballast regions and one hundred seventy insufficient ballast regions. A further one hundred forty one images from the same dataset were employed for sleeper labeling. These annotations support supervised learning for ballast condition assessment. Additional field testing data extracted from RealSense bag files (collected from field) are reserved for future dataset expansion and validation.

B. Model Training and Validation

The prepared datasets were used to train deep learning models for railroad defect detection. Object detection networks were initialized using pretrained weights and subsequently fine tuned on task specific training datasets. Hyperparameters including learning rate, batch size, and number of training epochs were selected based on validation performance. Model evaluation was performed using standard detection metrics such as mean average precision and per class recall on the held out test datasets. When applicable, cross validation was employed to improve generalization performance. Model selection for each defect type was guided by performance comparisons and prior validation results reported in earlier studies [2]–[4], [13], [14].

C. Backend Workflow

The backend workflow coordinates all server side processes required for automated defect detection. It manages file handling, preprocessing, model invocation, inference execution, and result storage while ensuring efficient resource utilization and reliable execution.

1) Server Environment Setup: To support stable and efficient execution, the server environment was configured with a structured directory hierarchy under the public folder. Dedicated directories were created for uploaded inputs, detection outputs, and stored model files. Required machine learning libraries, including PyTorch and TensorFlow, were installed, and software version compatibility was carefully verified to ensure consistent model behavior.

2) Model Selection for Defect Types: The backend dynamically invokes specific pretrained models based on the defect type selected by the user. Model selection was guided by prior evaluations of detection accuracy and suitability for integration into the application. Joint bar cracks are detected using YOLOv9e [4]. Missing bolts are identified using a YOLOv8x object detection network [3]. Track gauge deviation detection employs YOLOv5x segmentation for initial rail inference followed by SAMURAI based tracking for improved temporal stability [2]. Insufficient ballast detection is performed using gradient boosting paired with YOLOv8 [15].

3) Inference Workflow: During operation, pretrained machine learning models are loaded and maintained on the server to support inference requests. When a user initiates a detection task through the web interface, a PHP based controller triggers the execution of Python scripts responsible for defect detection. These scripts load the appropriate model and perform inference on the submitted image or video data. Model selection is performed dynamically based on the user specified defect category, and model weights are preloaded to reduce initialization overhead. Upon completion of inference, detection results are stored in the database. Each recorded result includes the defect type, associated video frame timestamp, spatial coordinates, and model confidence score. The stored results are then transmitted to the frontend application, enabling visualization, filtering, and user interaction.

D. Database

The software's database supports the railway defect detection workflow and stores information regarding input sources, detection tasks, results, and defect types. The database model comprises four primary entities: sources, jobs, detections, and issues. This section describes each entity and its attributes, as well as the relationships among them.

- **Sources:** This entity contains information about the user-provided data used for detection. It records metadata for each file, including a unique identifier, file name, type, path, size, creation date, and the timestamp of upload to the application.
- **Jobs:** This entity stores information about detection tasks. Each entry includes a unique job identifier, the date and time of the task, the job name, and associated GPS data.
- **Detections:** This entity records the outputs of detection tasks. Attributes include a unique detection identifier, the path to the result image, the timeframe of detection, GPS information, and the confidence level of the model. Foreign keys link this entity to sources, jobs, and issues to maintain relational integrity.
- **Issues:** This entity represents the defect types detectable by the application. Each record contains a unique identifier, the issue name, and information about the ML algorithm employed for detection.

The relationships among these entities establish clear traceability between detection results, their originating data sources, associated detection tasks, and corresponding defect types. This structured linkage enables efficient

storage, reliable retrieval, and systematic analysis of all detection activities within the application.

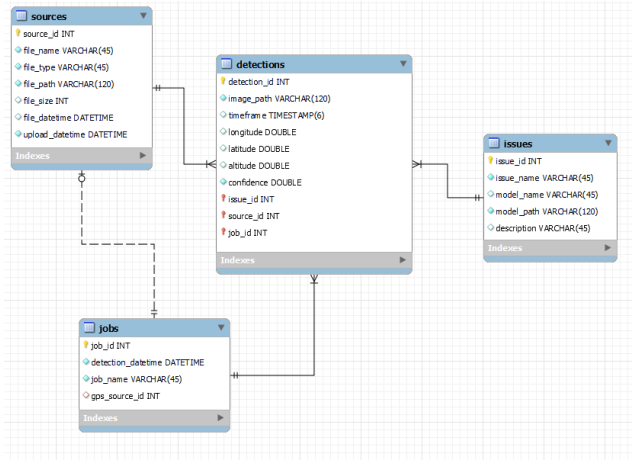


Fig. 1. Database ERD of the Railroad Visual Detection application

Figure 1 illustrates the relationships among the primary entities in the application. The Entity Relationship Diagram (ERD) presents the database schema and defines the dependencies across four core entities. Each source entity may generate multiple detection results, whereas each detection result is associated with exactly one source, enforced through the `source_id` attribute. Likewise, a single issue type may be linked to multiple detection results, but each detection result references only one issue type via `issue_id`. Detection results are also grouped under detection jobs, where each job may produce multiple results and each result belongs to a single job, established through the `job_id` attribute.

A source may participate in multiple detection jobs, while a detection job cannot exist independently of its associated source. Consequently, the jobs entity references the sources entity for identification, forming a weak relationship between these two entities. This relational design ensures clear traceability and consistent linkage across all detection records within the application database.

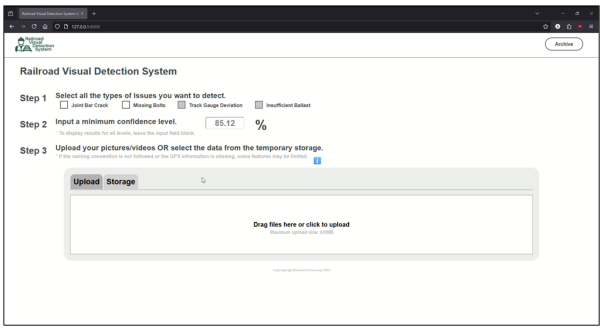
E. Front end Integration for Visualization of Defects

The application provides an interactive interface for railroad defect detection using machine learning algorithms and is accessible through a standard web browser in a local deployment environment. The front end is designed to be intuitive and easy to use, allowing users to perform detection tasks with minimal prior technical knowledge. A responsive design approach is adopted to ensure consistent functionality and usability across different devices and screen sizes.

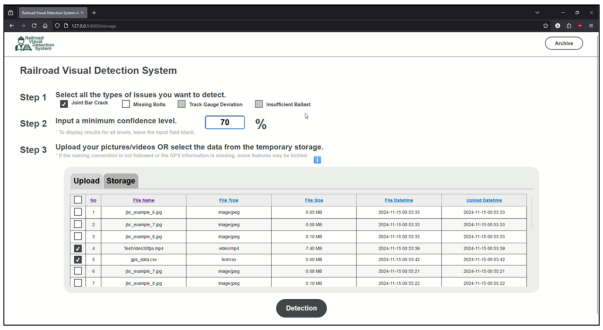
1) Detection: The detection module enables users to identify railroad defects in uploaded image and video data. Users initiate a detection task by selecting the desired defect type, specifying a confidence threshold, and uploading or selecting input files. Once configured, the detection process is started through a single action, simplifying the overall workflow.

To support efficient user interaction and targeted analysis, the application allows selection among multiple defect categories and provides confidence based filtering of detection results. Only detections exceeding the user specified confidence threshold are displayed, which helps reduce false positives and improves the reliability of reported results.

The module supports simultaneous upload of multiple images, videos, and CSV files through a drag and drop interface. Previously uploaded files are retained in a repository for future reuse, reducing redundant data uploads. Detection results are presented in a sortable tabular format and can be exported in ZIP, CSV, or PDF formats for further analysis and reporting. In addition, image based detection results can be visually inspected through enlarged previews, allowing users to quickly assess detected defects with greater clarity.



(a) Upload



(b) Storage

Fig. 2. Detection module of the web-based railway defect detection application

Figure 2 presents the implemented web-based detection application. (a) shows the main detection interface, where users select the defect type, specify a confidence threshold, and upload data for analysis. (b) displays the storage tab, which provides access to previously uploaded datasets for reuse. Once the desired detection defect type are selected, the corresponding pre-trained ML model is applied. The analyzed outputs are then displayed on the results page. Users can inspect and interpret the results directly.

Figure 3 shows the detection results. This page provides detailed information for each analyzed output, including the result image, an identifier for distinguishing images, the file type of the input data, the time frame, the detected issue type, GPS information, and the model confidence level. For video inputs, the time frame corresponds to the frame in which the issue was detected, while a default value of 00:00:00 is displayed for images. Detection results are organized in a sortable table, allowing each

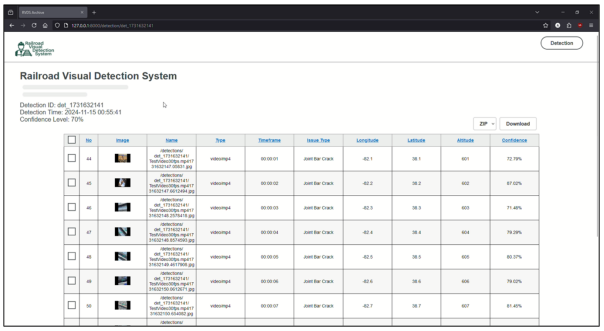


Fig. 3. Detection results of the application

column to be sorted in ascending or descending order by clicking the column header.

The application also provides additional features to facilitate the interpretation and utilization of detection results. Enlarged views of result images are available, allowing users to examine detected issues in greater detail. In addition, the platform supports exporting selected results in multiple formats, including ZIP, CSV, and PDF, enabling efficient storage and further analysis of the data.

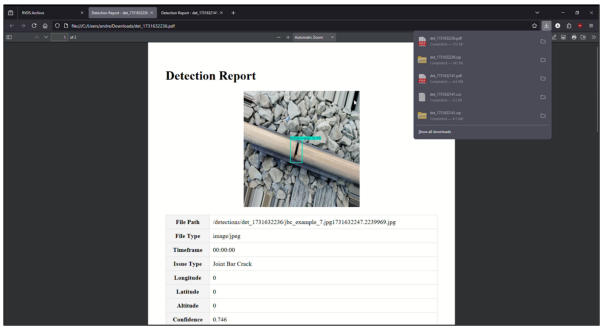
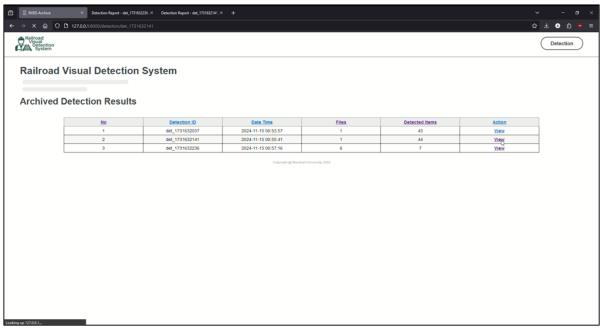


Fig. 4. PDF report summarizing the detection results

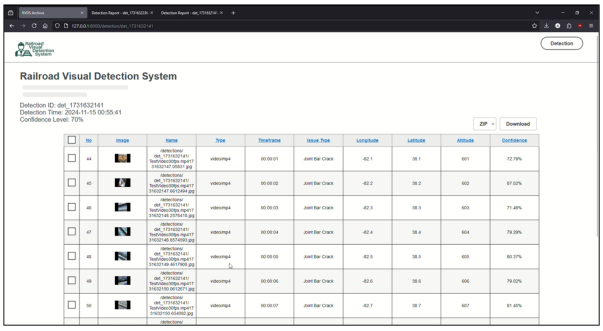
Figure 4 shows the results when a pdf file is selected among the download options. This file is in report format and contains all the data from the results table.

2) Archive: The Archive module provides access to previously performed detection tasks. It provides the same key functionalities as the Detection Results page. The previous outputs are presented in a structured table. The table can be organized by unique detection IDs or by the date and time of each task. Detailed information is provided for each item, allowing users to examine the analysis results comprehensively. The archived data also can be exported in ZIP, CSV, or PDF formats. This supports further analysis and record-keeping. The module also facilitates navigation to the detection workflow, enabling new detection tasks to be carried out when required.

Figure 5(a) shows the list of previously performed detection tasks. Detailed information for each task can be accessed from this list. Figure 5(b) displays the details page of a selected archived task. It presents the detection results and providing the same functionalities as the Detection Results page, including image enlargement, sorting, and file export.



(a) List of previously performed detection tasks



(b) Detailed results for individual detection tasks

Fig. 5. Archive module of the web-based railway defect detection application

IV. Performance Evaluation and Results

A. Experimental Setup

The primary objective of this study is to evaluate the computational performance and deployment efficiency of a web-based railroad defect detection application, with a particular emphasis on parallel inference work across diverse hardware resources. The application targets four representative railroad defect types: joint bar cracks, missing bolts, track gauge deviation, and insufficient ballasts.

Model selection for each defect category was guided by approaches that have already been validated in prior studies. For joint bar cracks and missing bolts, object detection models such as those in the YOLO family have been widely used because they can localize small and irregular defects while still supporting real-time inference. Track gauge deviation is inherently a geometric measurement problem, and earlier work has typically addressed it using regression-based or measurement-oriented techniques to estimate rail spacing. In the case of insufficient ballast, the problem is more closely related to visual texture and spatial distribution, which has commonly been handled using CNN-based models capable of capturing contextual visual patterns.

Following these established practices, representative models are adopted for each defect type without modification. This allows the study to focus on evaluating system-level behavior, particularly how these models perform under different execution conditions, rather than on changes to the model architectures themselves.

All experiments were conducted using a fixed input source consisting of a single segment extracted from a test video recorded during the field testings. This controlled setup ensures consistency across all experimental

conditions, particularly when evaluating the impact of parallelization. The joint bar crack detection task was selected as the primary benchmark for performance analysis, as it relies on an object detection pipeline and involves identifying small, low-contrast defects, which increases sensitivity to both inference latency and throughput. In addition, object detection involves several processing steps, such as feature extraction, localization, and post-processing, which makes it closer to the kind of workload encountered in real deployment compared to simpler tasks. A lightweight YOLOv11n model was employed for this purpose.

The hardware platform consists of an Intel Core i7-14700K CPU with 20 cores, divided between performance cores and efficiency cores, and an NVIDIA RTX 3090 GPU. The system has 64 GB of RAM and runs Windows 11. YOLOv11n was implemented in Python using CUDA-enabled libraries for GPU acceleration. Streaming mode was enabled for parallel execution experiments, and verbose logging was configurable to evaluate its effect on runtime performance. This combined hardware and software configuration provides a controlled environment to examine the trade-offs between single-device acceleration, multi-core CPU parallelism, and GPU-based inference across multiple streams.

B. Evaluation Methodology

Unlike conventional evaluation approaches that primarily emphasize detection accuracy, this study adopts an application oriented evaluation methodology that reflects practical deployment constraints in web based environments. The evaluation focuses on runtime efficiency, scalability under parallel workloads, and effective utilization of hardware resources, which are critical factors for real world deployment of machine learning driven inspection applications.

Several performance metrics are used to capture different aspects of application behavior. Inference time, measured in milliseconds per frame, represents the average processing time required for model execution under a given configuration. Total execution time measures the end to end duration required to process an entire video segment. To enable fair comparison across different levels of parallelism, an effective inference time is also reported by normalizing the total execution time by the number of concurrently processed streams. In addition, hardware utilization metrics, including CPU and GPU usage, are monitored to identify performance bottlenecks and assess resource efficiency.

Scalability is evaluated across multiple execution configurations, including single stream and multi stream scenarios with varying degrees of concurrency, specifically three, five, and ten parallel streams. Each configuration is tested using both CPU only execution and GPU accelerated inference to characterize performance trade offs across hardware platforms.

In addition to model execution parameters, application level factors such as streaming mode and logging verbosity are systematically varied to evaluate their impact on runtime performance. Although often overlooked, these parameters can introduce measurable overhead and significantly influence overall application behavior in practical deployment settings.

C. Web Application Performance

This section evaluates the runtime performance of the proposed web-based application, with a particular focus on parallel inference behavior under different hardware configurations. The analysis focuses on the joint bar crack detection task, which is used as the primary benchmark since it relies on an object detection pipeline and is sensitive to both latency and throughput.

TABLE I
SINGLE-STREAM INFERENCE PERFORMANCE

Execution Mode	Total Time (s)	Inference Time (ms/frame)	CPU (%)	GPU (%)
CPU-dominant	55	33	~ 15	$\sim$ **30**
GPU	**10-12**	**4.8**	~ 15	~ 36

Table I shows the single-stream inference performances. Under a single-stream configuration, GPU-based inference achieved an average inference time of approximately 4.8 ms per frame, while CPU-based inference required approximately 33 ms per frame. The total execution time was approximately 10–12 seconds for the GPU and 55 seconds for the CPU. Notably, even during CPU-based execution, GPU utilization remained at approximately 30%, suggesting that certain operations were implicitly offloaded to the GPU.

TABLE II
MULTI-STREAM PARALLEL INFERENCE PERFORMANCE

Execution Mode	Streams	Total Time (s)	Normalized Time (s/stream)	Inference Time (ms/frame)
CPU	3	87	29.0	33*
	5	113	22.6	103
	10	212	21.2	106
GPU	3	20	6.7	4.8*
	5	30	6.0	4.8*
	10	90	9.0	21.8

* It estimated from single-stream measurements due to test limitations.

To evaluate scalability, multiple parallel configurations were tested. Table II summarizes the multi-stream parallel inference performance for a 1-minute video. CPU-based execution exhibited higher single-stream execution times, but it showed consistent reductions in execution time under parallel configurations. In the 10-stream configuration, the total execution time was approximately 212 seconds. When normalized by the number of streams to illustrate per-stream efficiency, the effective execution time was approximately 21.2 seconds, representing approximately a 60% reduction relative to single-stream execution time of 55 seconds. Similar reductions were observed in the 3- and 5-stream configurations, with decreasing incremental improvements as the number of streams increased. CPU latency, defined as the time to process a single frame, was measured directly for single-stream execution and amounted to 33 ms per frame. Inference latency for GPU

3- and 5-stream configurations was derived from single-stream measurements.

GPU-based execution achieved lower per-frame inference latency. In the 10-stream configuration, the total execution time was approximately 1.5 minutes and the inference latency per frame was 21.8 ms. Normalized per-stream latency was approximately 2.2 ms. Across increasing levels of parallelism, total execution time decreased for both CPU and GPU. The rate of improvement diminished beyond five concurrent streams. Minimal differences were observed between the 5- and 10-stream GPU configurations.

A comparison between CPU and GPU runs indicates that GPU processing results in lower per-frame inference latency, while CPU-based operations achieve greater reductions in normalized per-stream time under higher concurrency.

Additional experiments were conducted to evaluate the impact of application-level configurations. Disabling verbose logging resulted in an execution time reduction of approximately 2 seconds in short-duration workloads. Enabling streaming mode reduced execution time compared to non-streaming execution. When identical video inputs were reused across multiple parallel streams, disk utilization remained minimal.

V. Discussion

The application demonstrates the practical potential of integrating ML techniques with a web-based visual detection application. The application enables automated detection of railroad defects by analyzing image data using pre-trained models, providing faster and more consistent inspection compared to traditional manual methods. By integrating these models into a web application, users can upload railroad images and receive detection results, supporting more efficient and scalable inspection workflows.

The experimental results indicate a clear distinction between GPU-based and CPU-based execution in terms of latency and scalability. GPU inference consistently achieves lower per-frame latency under single-stream conditions, which aligns with expectations given its highly parallel architecture and optimized tensor computation capabilities. However, as the number of concurrent streams increases, GPU utilization rapidly approaches saturation. This result suggests that performance is primarily limited by resource contention among concurrent inference streams. In addition, scheduling overhead and the absence of explicit batching mechanisms may also contribute to the observed performance characteristics.

In contrast, CPU-based execution exhibits more gradual performance degradation as the level of parallelism increases. The multi-core CPU architecture allows concurrent execution across multiple streams, and the operating system scheduler distributes workloads across available cores. Although individual inference latency on the CPU is higher compared to the GPU, the application benefits from improved aggregate throughput when multiple streams are processed in parallel. This results in more stable scalability within the tested range, particularly when the number of concurrent tasks aligns with the number of available cores.

The results also indicate that the interaction between CPU and GPU resources contributes to overall application behavior. Even during CPU-based execution, a noticeable level of GPU utilization was observed. This suggests that some portions of the workload may still utilize GPU resources. This behavior implies that the application is not strictly bound to a single processing unit, and that resource usage may overlap depending on the underlying framework and execution pipeline.

Overall, these findings highlight a trade-off between latency and scalability. GPU-based execution is more suitable for latency-sensitive workloads, while CPU-based execution provides better scalability under increased parallel workloads. These findings suggest that a hybrid execution strategy leveraging both CPU and GPU resources could further enhance overall application performance in multi-stream scenarios.

Several limitations should be considered. First, the performance of the detection model is highly dependent on the quality and diversity of the training dataset. If the dataset does not sufficiently represent different lighting conditions, rail types, or defect variations, the model's generalization ability may be limited. Second, performance may be affected by hardware constraints and resource contention. When processing large image files or multiple concurrent requests, limitations in CPU, GPU, and memory resources may impact response time and overall application efficiency.

Future work can focus on improving the robustness of the ML model by expanding the dataset and applying more advanced deep learning architectures. In addition, optimizing resource utilization and workload distribution between CPU and GPU will be important for improving performance in multi-stream scenarios. Further research into efficient parallel processing strategies within a local environment would also enhance application scalability.

VI. Conclusion

This study presented a web-based railroad defect detection application designed for practical deployment under real-world constraints. The application integrates multiple machine learning models selected for the target defect types, focusing primarily on evaluating runtime performance.

Experimental results show that GPU-based inference achieves substantially lower per-frame latency under single-stream conditions. In contrast, CPU-based execution demonstrates more consistent scalability across multiple concurrent streams due to multi-core parallelism and operating system-level scheduling. These results suggest that GPU and CPU resources have complementary characteristics. GPU execution is suitable for latency-sensitive tasks, while CPU execution can better accommodate increased parallel workloads.

Additional experiments indicate that system-level factors, including streaming configurations and logging settings, can influence execution time and resource utilization. Based on these findings, a hybrid strategy leveraging both CPU and GPU resources is proposed as a potential approach to optimize overall performance, although further experimental validation is needed.

Future work may focus on further improving the robustness and efficiency of the application in practical deployment scenarios, as well as evaluating strategies to maximize the utilization of available hardware resources under local execution constraints.

ACKNOWLEDGMENT

This research was supported by the Engineer Research and Development Center (ERDC): #W912HZ249C006.

REFERENCES

[1] Z. Zou, K. Chen, Z. Shi, Y. Guo, and J. Ye, "Object detection in 20 years: A survey," *Proceedings of the IEEE*, vol. 111, no. 3, pp. 257–276, 2023.

[2] v. Trung Le, H. Song, H. S. Narman, P. Zhu, A. Alzarrad, A. Cisko, and J. Beasley, "Automated measurement of horizontal gauge deviation in railroads using depth sensor camera and machine learning," *IEEE Access*, vol. 13, 2025.

[3] A. Damai, H. Song, H. S. Narman, A. Lambert, and A. Alzarrad, "Enhancing railway safety: A machine learning approach for automated detection of missing track bolts," in *Computing in Civil Engineering 2025*. American Society of Civil Engineers, 2025, pp. 224–234. [Online]. Available: https://ascelibrary.org/doi/abs/10.1061/9780784486436.024

[4] A. d'Arms, H. Song, H. S. Narman, N. C. Yurtcu, P. Zhu, and A. Alzarrad, "Automated railway crack detection using machine learning: Analysis of deep learning approaches," in *IEEE Annual Information Technology, Electronics and Mobile Communication Conference*, 2024.

[5] A. Kumar and S. Harsha, "A systematic literature review of defect detection in railways using machine vision-based inspection methods," *International Journal of Transportation Science and Technology*, vol. 18, pp. 207–226, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2046043024000716

[6] K. Mani and A. K. Shenoy, "Machine learning models in web applications: A comprehensive review," *ICT Express*, vol. 11, pp. 1110–1119, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405959525001304#sec3

[7] T. Solution, "Railwaytrackcrackdetection object detection dataset," https://universe.roboflow.com/technofly-solution/railwaytrackcrackdetection, 2022.

[8] S. T. Project, "Damage detection in tracks object detection dataset and pretrained model," https://universe.roboflow.com/system-thinking-project-4rfww/damage-detection-intracks, 2024.

[9] C. Ranganath, "Railway track fault detection classification dataset," https://universe.roboflow.com/chinmay-ranganath-ohlji/railway-track-faultdetection, 2023.

[10] T. Group, "Railway crack detection dataset," https://universe.roboflow.com/thesis-group/railway-crack-detection, 2024.

[11] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788.

[12] S. I. Eunus, S. Hossain, A. E. M. Ridwan, A. Adnan, M. S. Islam, D. Z. Karim, G. R. Alam, and J. Uddin, "Ecarrnet: An efficient lstm-based ensembled deep neural network architecture for railway fault detection," *AI*, vol. 5, no. 2, pp. 482–503, 2024. [Online]. Available: https://www.mdpi.com/2673-2688/5/2/24

[13] M. Liu, V. T. Le, H. Song, A. Chandramouli, H. S. Narman, and A. Alzarrad, "Comparing object detection, instance segmentation, and semantic segmentation for automated vegetation detection in railroad systems," in *IEEE Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)*, 2025, pp. 0184–0190.

[14] A. Chandramouli, H. Song, M. Liu, A. Damai, H. S. Narman, and A. Alzarrad, "Deep learning approaches for railroad infrastructure monitoring: Comparing yolo and vision transformers for defect detection," in *IEEE Annual Ubiquitous Computing, Electronics and Mobile Communication Conference*, 2025, pp. 0205–0211.

[15] S. Liu, D. Lester, H. Narman, A. Alzarrad, and P. Zhu, "Depth-enhanced yolo-sam2 detection for reliable ballast insufficiency identification," 2026. [Online]. Available: https://arxiv.org/abs/2602.18961