

Automated UI Component Detection and Visual Layout Quality Analysis from Webpage Screenshots Using Deep Learning

¹Aditya Dwa, ^{2*}Aryan Silawal, ³Ayush Prajapati, ⁴Nikhil Dwa

^{1,2,3}*Department of Electronics and Computer Engineering, Pulchowk Campus, Institute of Engineering, Tribhuvan University, Lalitpur, Nepal*

⁴*Department of Electronics and Computer Engineering, Western Regional Campus, Institute of Engineering, Tribhuvan University, Pokhara, Nepal*

Email: ¹adeetyadwa011@gmail.com, ³ayushpjpt10@gmail.com, ⁴nikhildwa95@gmail.com

*Corresponding Author: ²silawalaryan16@gmail.com

Abstract

This paper addresses the lack of automated, image-based tools for evaluating the visual layout quality of web user interfaces as existing approaches rely on source code and Document Object Model analysis rather than reasoning about the rendered interface as perceived by the end user. The study employs a YOLOv11s object detection model combined with a three criteria visual layout analysis pipeline covering clutter assessment, geometric alignment consistency and WCAG-based color contrast evaluation to detect and classify UI components and produce quantitative quality scores directly from webpage screenshots. The proposed method is evaluated on 46,131 processed samples from the WebUI dataset using standard object detection metrics including precision, recall, F1 score, mAP@0.5 and mAP@0.5:0.95. The results demonstrate an overall test set mAP@0.5 of 0.6236 and a precision of 0.7204 across 14 UI component categories, with individual class performance ranging from 0.9239 for footer to 0.2810 for button. The findings indicate that automated visual UI quality assessment is a tractable objective and that structurally consistent components are reliably detected, while underperforming classes such as button and navigation are primarily limited by class definition noise. This work contributes to the field of UI evaluation by providing a scalable, visual assessment framework that addresses actionable quality issues, serving as a practical tool for frontend developers, UI/UX designers and automated testing pipelines.

Keywords: *UI component detection, visual layout analysis, object detection, YOLOv11, webpage screenshot analysis, web accessibility*

I. INTRODUCTION

User Interface (UI) forms a visual bridge between an application and its users, allowing the users to properly interact with the application. The visual quality of a user interface plays a paramount role in determining the success of an application. Studies have shown that users form a lasting impression of an application within seconds of first interaction, thus making effective UI design not merely an aesthetic concern but also a determinant of user retention and engagement [1]. Despite carrying

a lot of unwavering significance, evaluation of UI quality has remained largely subjective. It has been dependent upon assessment by expert reviewers, manual reviews from end users and developer intuition. These processes are inconsistent, time consuming and non-quantifiable.

There exist some automated evaluation tools which address some UI quality concerns. They primarily focus on performance and accessibility of webpages. They analyze the underlying source code, metadata and Document Object Model (DOM) rather than the visual layout of the webpage that the users perceive. Visual layout analysis seems largely non-existent and even their evaluation processes are hindered when the aforementioned source code and DOM information are unavailable. Recent advances in computer vision and deep learning have demonstrated the viability of convolutional neural network-based approaches for detecting UI anomalies and display issues directly from rendered screenshots [2]. Though commendable, these efforts have focused primarily on anomaly identification rather than proper multi-criteria quantitative assessment of visual layout quality. Therefore, a unified system that combines UI component detection and structured scoring framework, operating from visual screenshots rather than underlying source code and DOM analysis represents an unaddressed gap in the current literature.

This paper addresses this gap by presenting an automated image-based system for UI component detection and visual layout quality analysis. A YOLOv11s object detection model is trained on a large-scale dataset of rendered webpage screenshots to detect and classify 14 categories of UI components. The detected components are then subjected to a three-criteria visual layout analysis pipeline that quantifies clutter, alignment consistency and color contrast in accordance with established accessibility standards. The system is accessible through a web-based interface with a visual screenshot as the only requirement. With no requirements of underlying source code, the system can perform visual layout analysis of any web pages with just their screenshots.

The major contributions of this paper are as follows:

- A novel end-to-end visual UI quality assessment pipeline that operates purely from rendered webpage screenshots, eliminating the source code and DOM dependency that constrains existing automated evaluation tools.
- A systematic evaluation of the YOLOv11s architecture for 14-class UI component detection on a large-scale web screenshot dataset, achieving an overall test set mAP@0.5 of 0.6236 and providing a quantitative baseline for this task.
- A three-criteria layout scoring framework combining clutter assessment, geometric alignment consistency and WCAG-based color contrast evaluation, producing interpretable quality scores for any rendered webpage.

The remainder of this paper is organized as follows. Section II reviews related work in automated UI evaluation and deep learning-based interface analysis. Section III

describes the methodology including dataset preparation, model architecture, training procedure, and layout analysis framework. Section IV presents experimental results and discussion. Section V concludes the paper and outlines directions for future work.

II. LITERATURE REVIEW

Evaluation of web user interfaces has been approached from two broad directions in the literature. One being DOM-based evaluation tools that analyze the underlying code structure of a webpage and the other being deep learning-based visual analysis systems that reason directly from rendered screenshots. This section reviews representative work in each category and identifies the gap that the proposed system addresses.

A. Traditional UI Evaluation Tools

Automated UI evaluation schemes have relied on inspecting underlying source code and Document Object Model rather than the rendered visual layout that end users see. Google Lighthouse is the most widely adopted tool in this category. It provides automated audits covering performance, accessibility, search engine optimization and best practices by analyzing page metadata, resource loading behavior and DOM structure [3]. Its evaluation scheme does include basic contrast checks which align with the WCAG standard but it is completely code driven. It does not reason about various UI factors like component arrangement, spacing consistency that users notice. Furthermore, tools in this category require direct access to the page source or DOM, limiting their applicability in contexts where only the rendered visual output of a webpage is available. This fundamental constraint motivates the need for an image-based evaluation approach that operates independently of source code access.

B. Deep Learning-Based Visual UI Analysis

The application of deep learning to visual UI understanding has gained increasing research attention as an alternative to code-based evaluation. Convolutional neural network-based approaches have been explored for UI anomaly detection and component recognition directly from rendered screenshots.

Owl Eyes applies CNN-based models to detect UI display issues in mobile application screenshots, identifying visual anti-patterns and rendering anomalies through learned visual features [4]. Nighthawk formulates display issue detection as an object detection task. It adopts the Faster-RCNN model to not only identify screenshots with UI display issues but accurately point out the location of the visual issues within the screenshot, which better helps developers and testers debug their GUI code [2]. The UI issues are categorized into five categories including component occlusion, text overlap, missing image, null value and blurred screen in this research model. Nighthawk demonstrated that rendering problems can be precisely located within a screenshot using standard object detection architectures. These works demonstrate the

viability of visual and deep learning-based approaches to UI evaluation but they focus on detecting anomalies and display error rather than comprehensive multi-criteria UI evaluation.

C. Research Gap

The reviewed literature reveals a consistent and well-defined gap. DOM based tools such as Google Lighthouse are unable to reason visually about the rendered interface and require source code access that is not always available. Deep learning-based systems have demonstrated the feasibility of visual UI analysis but have remained focused on anomaly detection. No existing system combines automated UI component detection with a quantitative scoring framework covering spatial arrangement, geometric alignment and accessibility-based color contrast, operating purely from screenshot input without any source code dependency. The proposed system directly addresses this gap.

III. METHODOLOGY

The proposed system follows a sequential pipeline progressing from dataset preparation through deep learning-based component detection to quantitative visual layout analysis. Fig. 1 illustrates the overall system architecture.

A. Dataset

The WebUI dataset [5] was adopted as the primary data source for this work. The dataset comprises 46,871 rendered webpage screenshots captured at a resolution of 1920×1080 pixels, each accompanied by automatically extracted structural metadata derived from the browser's accessibility layer and rendering engine. While the dataset is referred to as WebUI-70K, the actual number of samples available in the repository at the time of this study was 46,871. Each sample in the dataset corresponds to a single rendered web page and is organized as a self-contained folder containing several associated files. Each sample folder contains four relevant files: the webpage screenshot, an accessibility tree file encoding the semantic role of every interactive DOM element, a bounding box file recording the pixel-space coordinates of each element and a viewport file indicating which elements are visible within the rendered browser window.

Semantic labels were derived from the accessibility tree role values and mapped to 14 UI component categories as listed in Table I. These categories encompass structural elements including header, footer, navigation, and sidebar; interactive elements including button, input, and link; and content elements including heading, text, image, list, table, dialog, and container.

B. Data Preprocessing

Raw dataset samples were processed through a three-phase pipeline to produce clean and aligned image-label pairs in YOLO annotation format.

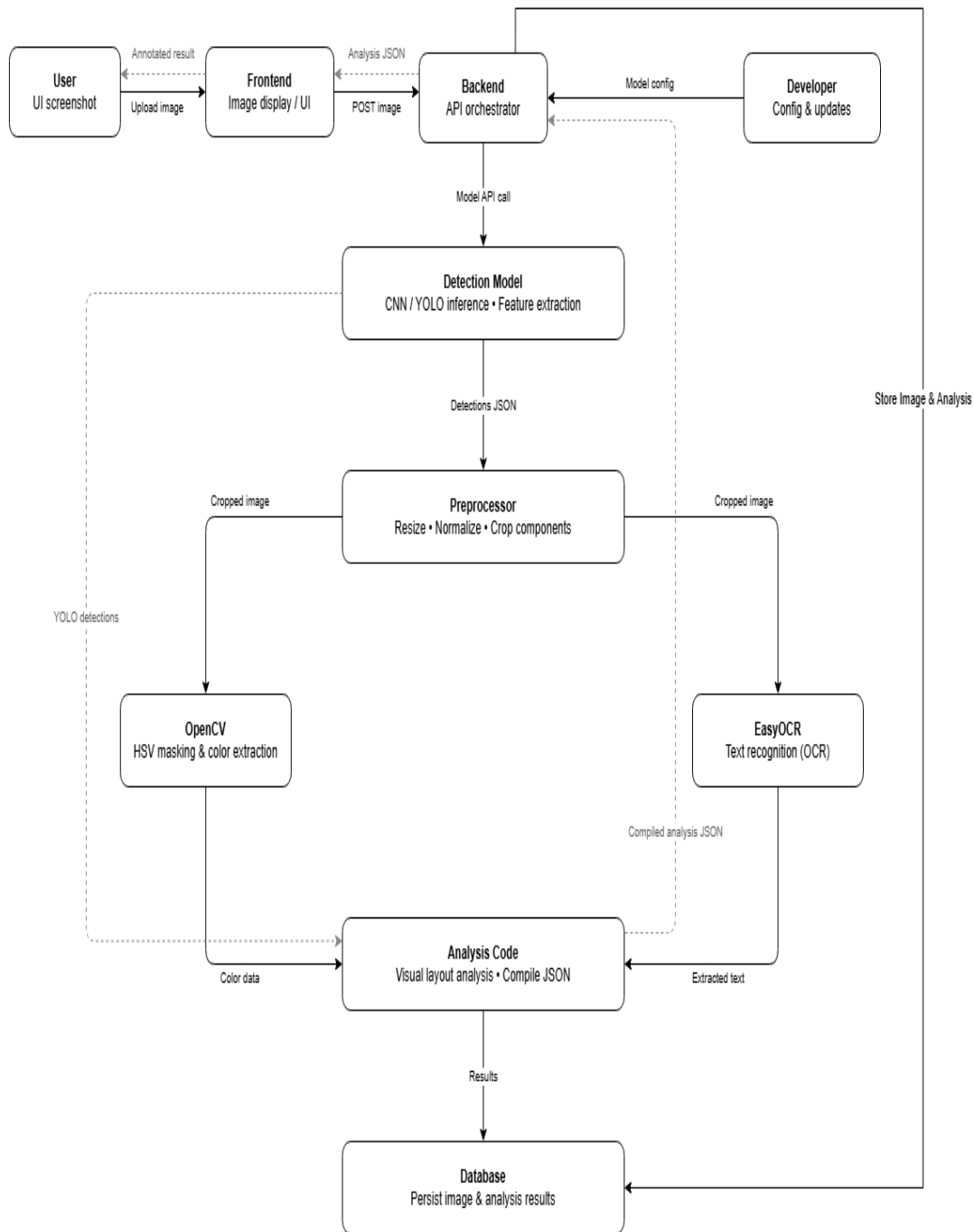


Fig. 1: Overview of the proposed UI evaluation pipeline.

Table 1: UI Component Classes And Numeric Identifiers

ID	Class
0	Heading
1	Link

2	Image
3	Text
4	List
5	Header
6	Footer
7	Table
8	Input
9	Button
10	Navigation
11	Sidebar
12	Dialog
13	Container

1. **Directory Cleanup:** Each sample folder was validated against predefined naming conventions i.e., a set of allowed prefixes and suffixes to remove irrelevant and auxiliary files. Every nested sub-folder within a sample folder was deleted to maintain the flat structure of sample folders. Each surviving sample folder was expected to contain exactly a predefined number of files, corresponding to the complete set of annotation and image files required for one training sample. Any folder whose file count deviated from this expectation was considered incomplete and unsuitable for training. Such folders were permanently removed from the dataset and their names were recorded in an audit log file for traceability.
2. **Resource Initialization:** All image files were extracted from their individual sample folders and consolidated into a single centralized image dataset directory. Each image was renamed to the folder name to reflect the identity of the sample it belonged to, ensuring consistent and unambiguous naming across the dataset. The original files were removed from their source locations after copying thus, completing the relocation. Annotation JSON files distributed in compressed format were decompressed and written to disk in their raw JSON form, with the compressed originals removed upon successful extraction. This step was a prerequisite for the subsequent annotation parsing phase.
3. **JSON to Label Conversion:** Annotation files were processed to generate YOLO-format label files. For each sample, elements confirmed as visible in the viewport file were retained. Similarly, in the accessibility tree, nodes that carried no meaningful semantic role and elements explicitly marked with a null role were discarded. The pixel coordinates (x, y, width, height) of each node were retrieved from the accompanying bounding box annotation file. The bounding box data was matched to each element using the shared unique identifier present across all annotation files. Bounding box coordinates were clipped to the 1920×1080

viewport boundary following the principles of the Liang-Barsky clipping algorithm and elements whose clipped dimensions fell below ten pixels in either dimension was removed [6]. The retained coordinates were normalized to the range $[0, 1]$ as required by the YOLO annotation format. Samples yielding no valid annotations after filtering were removed alongside their corresponding images maintaining complete image-label alignment. The processed annotation data for each valid sample was written to a plain text label file following the YOLO format, where each line encoded one annotated UI element as a sequence of five values: the numeric class identifier followed by the normalized center coordinates and normalized dimensions of its bounding box. Beginning from the 46,871 available samples, the preprocessing pipeline removed 740 samples due to incomplete annotations or missing files, yielding a final valid dataset of 46,131.

- C. **Model Architecture:** The YOLOv11s object detection model was selected as the detection backbone for this work. YOLOv11 follows the established three-component architecture comprising a backbone for multi-scale feature extraction, a neck for feature pyramid aggregation and a detection head for bounding box regression and classification [7]. The small variant was adopted to balance detection accuracy with computational efficiency across the range of hardware used during the course of this work.

The model input resolution was set to 1024×1024 pixels rather than the default 640×640 . This decision was motivated by the need to preserve finer spatial detail for small UI elements particularly text-based components such as links, headings and input fields that occupy relatively small pixel regions on the original 1920×1080 canvas. The detection head was configured for 14 output classes listed in Table I. The model was initialized from the official pretrained YOLOv11s weights which were trained on the COCO dataset for general object detection. This enabled the transfer of previously learned visual feature representations to the UI component detection task.

- D. **Training:** Training was conducted on the full dataset of 46,131 samples, divided into training, validation and test splits following an 80/10/10 ratio using an NVIDIA RTX 4090 GPU with 24 GB of VRAM. This yielded approximately 36,905 samples for training, 4,613 for validation, and 4,613 for testing. A batch size of 8 was adopted with 6 parallel data loading workers. The learning rate was retained at the YOLOv11s default configuration. Training was executed for a maximum of 750 epochs with early stopping applied at a patience value of 50 epochs.
- E. **Visual Layout Analysis:** Following UI component detection, the bounding box coordinates and color attributes of each detected element were subjected to a three-criteria visual layout analysis pipeline. The analysis operates independently of the webpage source code, using only the detection outputs and pixel-level image data

as inputs. For each detected UI component i , let $B_i = (x_i, y_i, w_i, h_i)$ denote its bounding box. Let N denote the total number of detected components and $W \times H$ the viewport dimensions.

1. Clutter Score: The clutter score is designed to assess how efficiently multiple UI components were distributed within the viewport. It is formulated as a weighted combination of four sub-criteria: component density D , component area ratio A_{ratio} , spacing variance S , and overlap penalty P_n :

$$\text{Clutter Score} = 100 \times (w_1 D + w_2 A_{ratio} + w_3 S + w_4 P_n)$$

where $w_1 = w_2 = w_3 = w_4 = 0.25$, representing equal importance assigned to each sub-criterion.

Component density is computed as:

$$D = \frac{N}{N_{max}}$$

where $N_{max} = 80$ is an empirically defined maximum component threshold. It was set based on the dataset distribution where the mean number of components per sample is 61.7, yielding a density score of approximately 0.77 for an average page. This calibration ensures that typical pages register as moderately dense rather than clean or maximally cluttered, preserving meaningful score discrimination across the dataset. The component area ratio measures the proportion of the viewport area occupied by detected components:

$$A_{ratio} = \frac{\sum_{i=1}^N (w_i \times h_i)}{W \times H}$$

Spacing variance assesses whitespace consistency across the layout by computing the normalized variance of minimum inter-component distances. For each component, the minimum distance to its nearest neighbor is computed as $d_i = \min_{j \neq i} \text{dist}(B_i, B_j)$. The normalized variance is then:

$$S = \frac{\sigma_d^2}{\sigma_{max}^2}, \quad \text{where } \sigma_d^2 = \frac{1}{N} \sum_{i=1}^N (d_i - \bar{d})^2$$

where $\sigma_{max}^2 = \left(\frac{\sqrt{W^2 + H^2}}{4} \right)^2$ is the maximum variance normalizer defined as the squared quarter-diagonal of the viewport.

The overlap penalty applies a normalized IoU (Intersection over Union)-based penalty to component pairs exhibiting intersection or near-intersection. For each component pair (i, j) where $i \neq j$, a pairwise penalty is defined as:

$$p_{ij} = \begin{cases} 1.0 & \text{if } IoU_{ij} > 0.1. \\ 0.5 & \text{if } \text{dist}(B_i, B_j) < \epsilon \\ 0 & \text{otherwise} \end{cases}$$

The normalized overlap penalty is then computed by aggregating over all pairs and normalizing by the number of detected components N :

$$P_n = \frac{1}{N} \sum_{i=1}^N \sum_{j=i+1}^N p_{ij}, \quad P_n \in [0, 1]$$

Based on the computed clutter score, each webpage is classified into one of three categories: clean, moderate, or crowded.

2. **Alignment Consistency:** Alignment consistency measures the degree to which detected components adhere to a coherent geometric grid structure. The score is formulated as the average of three sub-criteria, each measuring alignment along a different spatial reference. Three spatial references are evaluated: left edge position $L_i = x_i$, horizontal center position $C_i = x_i + w_i/2$, and baseline position $B_i = y_i + h_i$. For each reference, the variance of the positional attribute across all components is computed and normalized to produce an alignment score in the range $[0, 1]$.

$$A_L = 1 - \frac{\sigma_L^2}{\sigma_{L,max}^2}, A_C = 1 - \frac{\sigma_C^2}{\sigma_{C,max}^2}, A_B = 1 - \frac{\sigma_B^2}{\sigma_{B,max}^2}$$

where σ_L^2 , σ_C^2 , σ_B^2 are the variances of left edge, center and baseline positions respectively and

$$\sigma_{L,max}^2 = (W)^2, \sigma_{C,max}^2 = (W)^2, \sigma_{B,max}^2 = \left(\frac{H}{2}\right)^2$$

The final alignment score is the arithmetic mean of the three sub-criteria scores.

$$\text{Alignment Score} = \frac{A_L + A_C + A_B}{3}$$

Based on the computed alignment score, each webpage is classified into one of three categories: highly consistent, acceptable, or poor alignment.

3. **Color Contrast:** Color contrast is evaluated per detected component by computing the WCAG relative luminance contrast ratio between the dominant foreground and background colors. Dominant colors are extracted using K-means clustering with $K = 2$ applied to the pixel values of each cropped component region. The text region localization is assisted by EasyOCR to refine foreground pixel separation.

RGB channel values are first normalized as $C_s = C/255$. Gamma correction is then applied to each normalized channel to account for the nonlinear manner in which the human visual system perceives luminance differences.

$$C = \begin{cases} \frac{C_s}{12.92} & \text{if } C_s \leq 0.04045 \\ \left(\frac{C_s + 0.055}{1.055} \right)^{2.4} & \text{if } C_s > 0.04045 \end{cases}$$

Relative luminance is then computed as:

$$L = 0.2126R + 0.7152G + 0.0722B$$

The contrast ratio between the lighter luminance L_1 and darker luminance L_2 is:

$$\text{Contrast Ratio} = \frac{L_1 + 0.05}{L_2 + 0.05}$$

Components are evaluated against the WCAG AA compliance threshold of 4.5:1 [8]. This determines whether each component met the minimum accessibility standard for color contrast and readability. The system reports the average contrast ratio, per-component compliance status and actionable suggestions for failing components.

IV. RESULTS AND DISCUSSION

The trained YOLOv11s model was evaluated on the held-out test set of 4,613 samples following the completion of the 715-epoch training run. Performance was measured using precision, recall, F1 score, mAP@0.5 and mAP@0.5:0.95 computed at the component level and aggregated across all 14 classes. The visual layout analysis pipeline was subsequently demonstrated on representative webpage samples to illustrate the system's end-to-end output.

A. Training Convergence

Fig. 2 shows the training and validation loss curves across all three loss components: box loss, class loss and distribution focal loss. All three graphs demonstrate consistent and stable convergence over the 715 training epochs. All three loss values decreased steadily and continued to decline and started to plateau at the end of the training indicating curve fitting. As a major observation, the validation loss remained consistently below the training loss throughout the duration of training across all three loss components. This indicates that the model generalized well to unseen data and did not exhibit significant overfitting. The consistently lower validation loss is attributable to the heavy augmentation applied exclusively during training including mosaic augmentation and random geometric transforms. This artificially increased the difficulty of the training task and inflate training loss relative to the validation set which did not go under any augmentation. The training, validation and test splits were generated from a single randomized partition with no sample overlap. This confirms that the pattern does not indicate data leakage.

The overall mAP@0.5 progression shown in Fig. 3 confirms the model's continuous improvement throughout training. Starting from an mAP@0.5 of approximately 0.22 in epoch one, the metric rose sharply in the initial training phase before transitioning into a slower but consistent upward trajectory. The best validation mAP@0.5 of 0.6089 is seen in epoch 679. This checkpoint was used for all subsequent test set evaluation, with the test set mAP@0.5 of 0.6236 reported in Table II reflecting a distinct held-out split. The presence of a plateau near the end of the training reinforces the observation that the maximum yield has been reached. Although the results represent a lower bound for a decent image recognition model, the datasets used,

ambiguity in the adjacent classes and the proposed architecture could only reach performance up to this point.

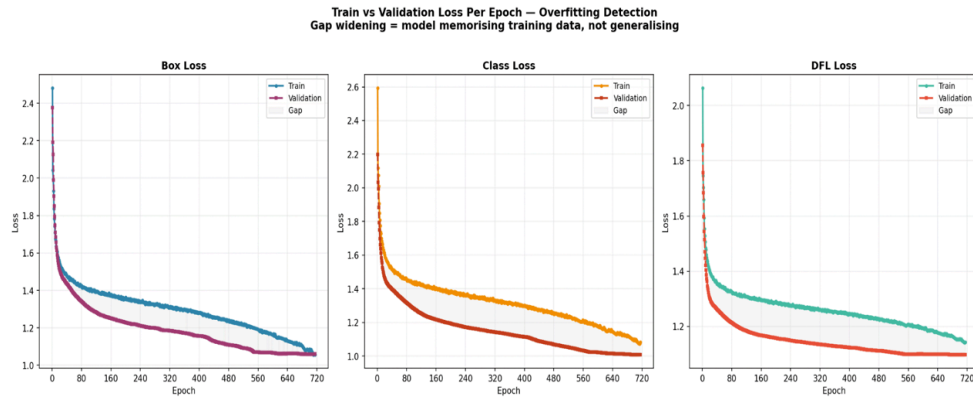


Fig. 2: Train vs. validation loss per epoch for box loss, class loss and DFL loss across 715 training epochs

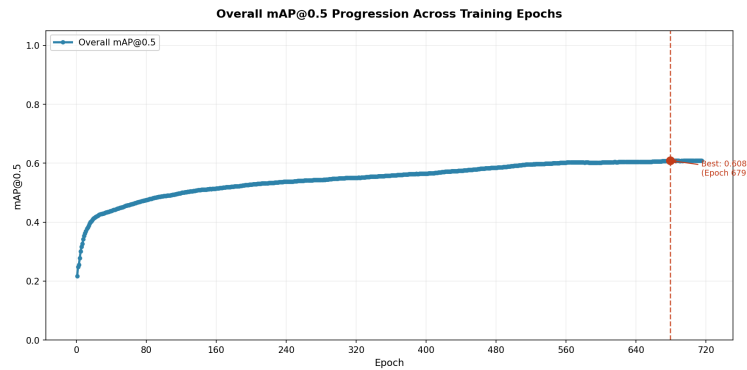


Fig. 3: Overall validation mAP@0.5 progression across 715 training epochs. Best value of 0.6089 recorded at epoch 679.

B. Overall Detection Performance

Table II presents the overall detection performance of the proposed system on the test set. The model achieved an overall test set mAP@0.5 of 0.6236 and a precision of 0.7204, demonstrating decent detection capability across a diverse set of 14 UI component categories on unseen webpage screenshots.

The precision of 0.7204 indicates that approximately 72% of all component detections correspond to genuine UI elements, reflecting a conservative detection behavior that prioritizes avoiding false positives. Furthermore, the recall of 0.5458 indicates that the model detects approximately 54% of all ground truth components. This precision-recall asymmetry is largely a contributing factor to the underperformance of specific classes discussed in Section IV-C. The gap between test set mAP@0.5 (0.6236) and mAP@0.5:0.95 (0.4859) reflects the inherent difficulty of achieving tight bounding box localization for UI elements whose boundaries are frequently ambiguous in the annotations derived from accessibility tree.

C. Per-Class Detection Performance

Table III presents the per-class detection metrics on the test set. Fig. 4 provides a visual comparison of per-class mAP@0.5, clearly distinguishing classes that exceed the 0.5 performance threshold from those that fall below it.

Ten of the 14 classes achieved an mAP@0.5 above 0.5, with footer recording the highest score of 0.9239 followed by header at 0.8128 and heading at 0.7749. The strong performance of these structurally consistent classes reflects their distinctive visual presentation and predictable spatial positions within webpage layouts. Input fields also demonstrated strong precision at 0.8242, reflecting the relatively standardized visual appearance of form elements across web interfaces.

Table 2: Overall Detection Performance On Test Set

Metric	Value
Precision	0.7204
Recall	0.5458
mAP@0.5	0.6236
mAP@0.5:0.95	0.4859

Table 3: Per-Class Detection Performance On Test Set

Class	P	R	F1	mAP@0.5
Heading	0.7660	0.7155	0.7399	0.7749
Link	0.7400	0.5526	0.6327	0.6508
Image	0.7586	0.5798	0.6573	0.6561
Text	0.6705	0.5370	0.5963	0.5859
List	0.7546	0.5992	0.6680	0.7214
Header	0.6966	0.7759	0.7341	0.8128
Footer	0.8820	0.8942	0.8880	0.9239
Table	0.6155	0.5746	0.5944	0.5911
Input	0.8242	0.6172	0.7059	0.7070
Button	0.5955	0.2156	0.3166	0.2810
Navigation	0.6794	0.2759	0.3924	0.4384
Sidebar	0.7821	0.5898	0.6725	0.6913
Dialog	0.6227	0.3488	0.4471	0.4183
Container	0.6984	0.3647	0.4792	0.4778

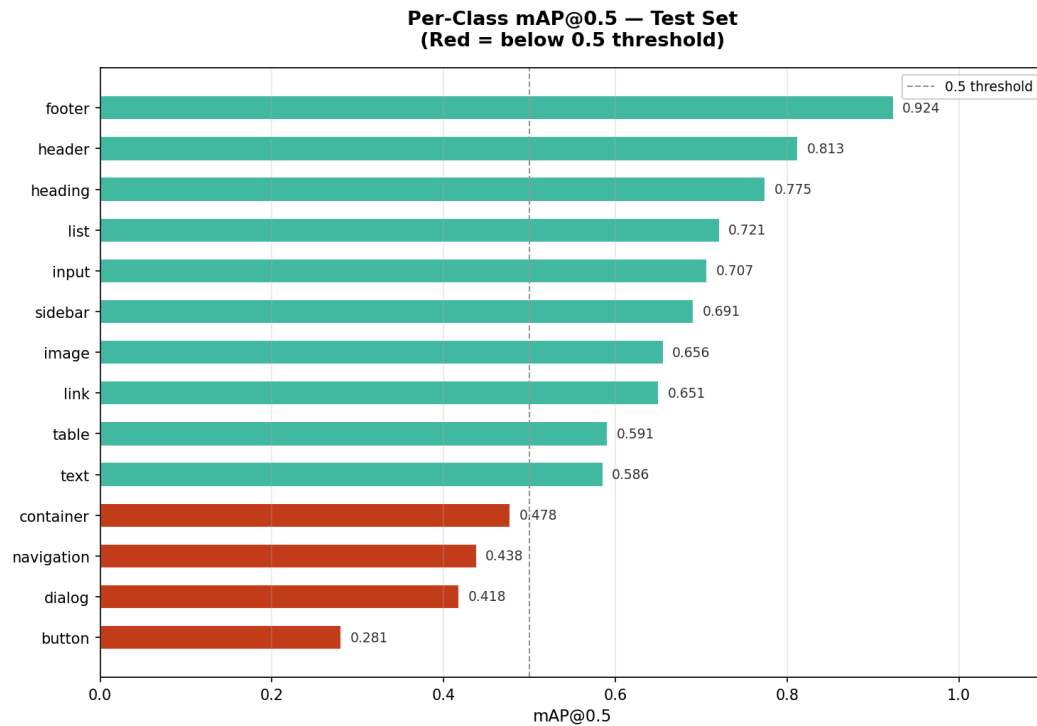


Fig. 4: Per-class mAP@0.5 on the test set. Classes shown in red fall below the 0.5 threshold.

Four classes fell below the 0.5 mAP@0.5 threshold: button, navigation, container and dialog. The button class recorded the lowest performance across all metrics with a recall of only 0.2156 which is caused by the class definition rather than insufficient training. The button category aggregates traditional clickable buttons alongside checkboxes, radio buttons, sliders and progress bars which creates extreme intra-class visual variance. This prevents the model from converging onto a unified feature representation. Navigation similarly suffered from a recall of 0.2759 which was attributable to the highly varied visual forms that navigation components take across different websites. Additionally, low recall for navigation class is attributable to extremely low presence of navigation components throughout the training sets. Dialog class also suffers from the shortcomings faced by navigation class thus, leading to its underwhelming performance. The container underperformance is a direct consequence of mapping the generic div role to this class, which introduces spatially overlapping annotations with nearly every other class and creates contradictory supervision signals during training. These findings indicate that the underperformance of these four classes is fundamentally a class definition issue rather than an architectural limitation.

D. Confusion Matrix Analysis

The normalized confusion matrix presented in Fig. 5 provides further insight into inter-class classification behavior. The diagonal entries confirm the per-class detection

trends, with footer achieving the highest diagonal value of 0.89 and button and navigation achieving the lowest diagonal value of 0.19 each. The background row reveals that missed detections dominate over inter-class confusion for the underperforming classes. For example, 68% of button instances and 54% of navigation instances are classified as background rather than as any other UI component class. This confirms that the primary challenge is detection rather than discrimination for these categories. The container class exhibits the most distributed off-diagonal pattern with misclassification spread across nearly all other classes. This directly confirms the label noise introduced by the generic div role mapping. These findings collectively indicate that architectural improvements alone are insufficient to resolve the underperformance of button, navigation, and container.

E. Visual Layout Analysis

The visual layout analysis pipeline was applied to two representative webpage samples to demonstrate the system's end-to-end output. Table IV summarizes the scoring results for both samples.

Fig. 6 presents a content-rich webpage with 24 detected components, receiving a clutter score of 58.4, an alignment score of 74.1, and a color contrast score of 6.01. The average contrast ratio of 6.01 exceeds the WCAG AA threshold of 4.5:1, with 4 of 6 text-bearing components compliant. Two link components were identified as failing the contrast requirement. Also, the system generates specific per-component remediation suggestions.

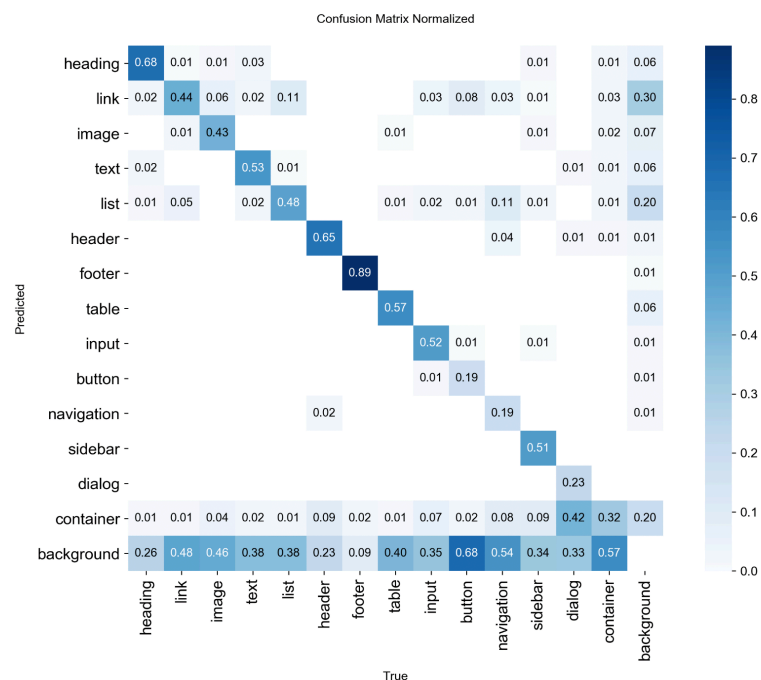


Fig. 5: Normalized confusion matrix on the test set. Rows represent predicted classes and columns represent true classes. Background column represents missed detections.

Table 4: Visual Layout Analysis Scores For Representative Samples

Sample	Clutter	Alignment	Contrast
Well-structured webpage	58.4	74.1	6.01
Newspaper archive webpage	3.3	97.6	0.0

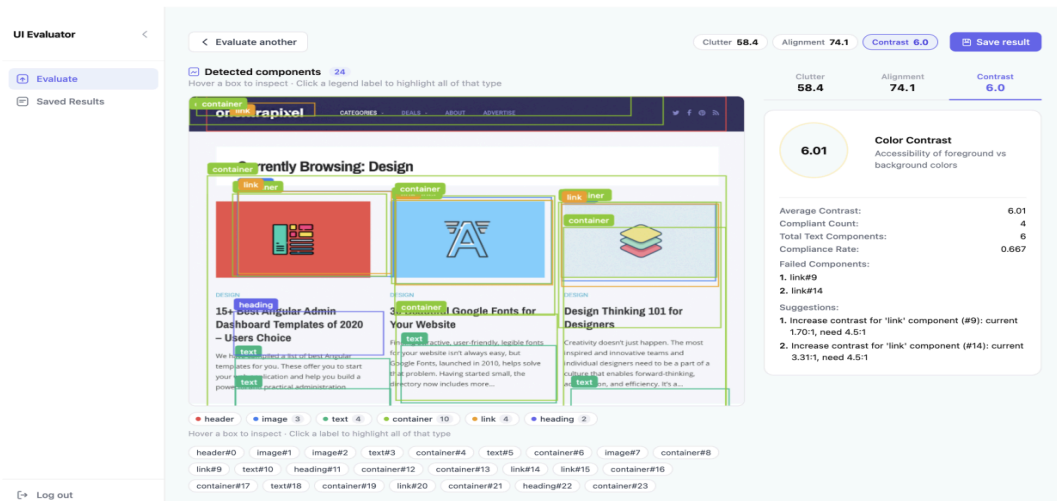
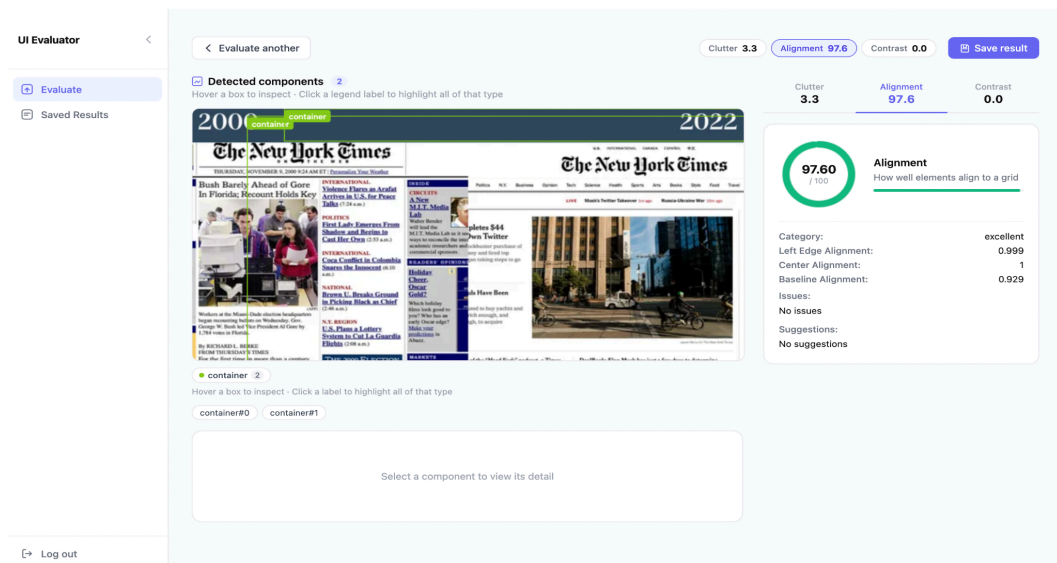
**Fig. 6:** Layout analysis output for a well-structured webpage. Clutter: 58.4, Alignment: 74.1, Contrast: 6.01.**Fig. 7:** Layout analysis output for a newspaper archive webpage. Clutter: 3.3, Alignment: 97.6, Contrast: 0.0. Despite excellent geometric alignment, all text components fail the WCAG contrast threshold.

Fig. 7 presents a newspaper archive screenshot with only 2 detected components. It records a clutter score of 3.3 and an alignment score of 97.6 reflecting the highly

consistent column grid structure of the layout. However, the contrast score of 0.0 reveals a complete failure against the WCAG AA threshold, with zero text components meeting the minimum 4.5:1 requirement. This result illustrates a critical capability of the proposed system i.e., a webpage can exhibit near perfect geometric alignment while simultaneously failing on accessibility readability standards. This is a distinction that DOM-based tools and single-score quality metrics are unable to surface. The three-criteria scoring framework directly addresses this limitation by providing independent and interpretable scores across each quality dimension.

The two samples together demonstrate the complementary nature of the three scoring criteria and confirm that no single metric is sufficient to characterize visual UI quality. The proposed system successfully identifies actionable quality issues across both the geometric and accessibility dimensions of interface design while operating purely from screenshot input without any requirement for source code or DOM access. This addresses directly the gap identified in Section II.

F. Comparison with Existing Approaches

A direct quantitative comparison against prior systems was determined to be infeasible for reasons specific to each related work. This limitation is addressed explicitly rather than being omitted.

Google Lighthouse operates on a fundamentally different input modality. It requires live page source, network access, DOM inspection and produces outputs such as performance scores, SEO grades and code-level accessibility flags. This approach shares no common evaluation protocol with per-component bounding box detection metrics such as mAP@0.5. Although WCAG AA contrast checking is performed by Lighthouse, it is implemented by reading declared CSS property values from the computed style tree rather than from rendered pixel data. As a result, contrast failures arising from background images, semi-transparent overlays or CSS inheritance chains that alter the visually perceived contrast at render time cannot be identified. The proposed system instead computes contrast from actual rendered pixel values via K-means color extraction, more faithfully reflecting what is perceived by the end user. Nighthawk which is the most directly comparable visual deep learning system, addresses a categorically different problem i.e., rendering bug detection in Android applications. The model is trained and evaluated on synthetically generated buggy screenshots derived from the Rico dataset. In contrast, the proposed system targets semantic component classification across 14 UI element categories on real webpage screenshots from the WebUI dataset and multi-criteria layout quality scores are produced rather than bug-presence flags. The output formats, evaluation metrics, training datasets and task definitions share no common ground between the two systems. Applying Nighthawk's evaluation protocol which is binary buggy/not-buggy classification with AP/AR computed over five bug region categories, to the 14-class component detection task addressed in this paper would not meaningfully evaluate

either system against its intended objective. This is interpreted not as a limitation of either system but as a consequence of the two systems addressing genuinely different problem.

Table 5 summarizes a qualitative comparison across capability dimensions relevant to the problem statement in Section I.

Table 5: Qualitative Comparison Of Ui Evaluation Approaches Across Key Capability Dimensions.

Capability	Google Lighthouse	Nighthawk	Proposed system
Screenshot-only input (no DOM/source code required)	No	Yes	No
Semantic UI component classification	No	No	Yes (14 classes)
Spatial localization of detected elements	No	Yes (5 bug types)	Yes (14 classes)
Multi-criteria layout quality scoring	No	No	Yes
Pixel-level WCAG accessibility evaluation	No	No	Yes

Nighthawk demonstrated that object detection can localize visual issues without source code access which motivated the visual approach adopted here but the proposed system focuses on classifying different UI components present in a webpage screenshot and scoring its layout quality, rather than detecting what is broken. Lighthouse remains the most mature accessibility pipeline among the three but is entirely source-code-driven and is unable to reason about rendered pixel-level contrast.

V. CONCLUSION, SUGGESTIONS AND RECOMMENDATIONS

A. Conclusion

This paper presented an automated image-based system for UI component detection and visual layout quality analysis operating purely from webpage screenshots. The proposed YOLOv11s-based detection model achieved an overall test set mAP@0.5 of 0.6236 across 14 UI component categories. The model showed strong performance on structurally consistent classes and identified underperformance in button, navigation and container classes attributable to class definition noise rather than architectural limitations. The three-criteria layout analysis framework demonstrated the ability to

address actionable quality issues across clutter, alignment and accessibility dimensions simultaneously. This capability is not provided by existing DOM-based evaluation tools. The results confirm that automated visual UI quality assessment is a tractable and practically valuable objective. Thus, it provides a scalable alternative to subjective manual evaluation without requiring source code or DOM access.

B. Suggestions and Recommendations

Based on the findings of this work, the following directions are recommended for future development. Refined class definitions for button and navigation, separating visually dissimilar component subtypes, are expected to yield significant recall improvements. A post-processing filtering strategy for the container class would reduce label noise and improve detection precision. Extended training beyond 715 epochs and evaluation of larger YOLOv11 variants are recommended given the absence of convergence at the current training limit. Finally, extending the system to handle dynamic UI states and expanding the layout analysis framework with additional criteria such as visual hierarchy and typography consistency would further enhance the practical utility of the proposed system.

ACKNOWLEDGMENT

We would like to express sincere gratitude to the Department of Electronics and Computer Engineering, Pulchowk Campus, Institute of Engineering, Tribhuvan University, for providing the computational resources and academic support that made this research possible.

AUTHOR CONTRIBUTIONS

Aditya Dwa: Conceptualization, data acquisition, analysis framework development, writing (review and editing).

Aryan Silawal: Conceptualization, model architecture selection and training, system implementation, writing (original draft).

Ayush Prajapati: Visualization, system implementation, writing (review and editing).

Nikhil Dwa: Data preprocessing, data transformation, writing (review and editing).

DATA AVAILABILITY

The WebUI dataset used in this study is publicly available on Hugging Face at <https://huggingface.co/datasets/nicholasKluge/webui-70k>.

CONFLICT OF INTEREST

We declare no conflict of interest.

REFERENCES

- [1] G. Lindgaard, G. Fernandes, C. Dudek, and J. Brown, "Attention web designers: You have 50 milliseconds to make a good first impression," *Behaviour and Information Technology*, vol. 25, no. 2, pp. 115–126, 2006.
- [2] Z. Liu, C. Chen, J. Wang, X. Che, B. Huang, Z. Xu, and Q. Wang, "Nighthawk: Fully automated localizing UI display issues via visual understanding," *IEEE Transactions on Software Engineering*, 2021.
- [3] Google, "Lighthouse: Automated auditing, performance metrics and best practices," <https://developers.google.com/web/tools/lighthouse>, 2024, accessed: 2024.
- [4] Z. Liu, C. Zhao, J. Wang, and Z. Xu, "Owl eyes: Spotting UI display issues via visual understanding," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.
- [5] J. Wu, S. Wang, S. Shen, Y. Wan, J. Nichols, and T. J.-J. Li, "WebUI: A dataset for enhancing visual UI understanding with web semantics," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, 2023.
- [6] Y. D. Liang and B. A. Barsky, "A new concept and method for line clipping," *ACM Transactions on Graphics*, vol. 3, no. 1, pp. 1–22, 1984.
- [7] G. Jocher and J. Qiu, "Ultralytics YOLO11," <https://github.com/ultralytics/ultralytics>, 2024, version 11.0.0.
- [8] World Wide Web Consortium, "Web content accessibility guidelines (WCAG) 2.1," <https://www.w3.org/TR/WCAG21/>, 2018, accessed: 2024.