QLUE: A Computer Vision Tool for Uniform Qualitative Evaluation of Web Pages

Waleed Hashmi  
NYUAD  
Abu Dhabi, UAE  
waleedhashmi@nyu.edu

Moumena Chaqfeh  
NYUAD  
Abu Dhabi, UAE  
moumena@nyu.edu

Lakshmi Subramanian  
NYUAD  
New York, USA  
lakshmi@cs.nyu.edu

Yasir Zaki  
NYUAD  
Abu Dhabi, UAE  
yasir.zaki@nyu.edu

ABSTRACT
The increasing complexity of the web has attracted a number of solutions to offer optimized versions of web pages that are lighter to process and faster to load. These solutions have been quantitatively evaluated to show significant speed-ups in load times and/or considerable savings in bandwidth/memory consumption. However, while these solutions often produce optimized versions from existing pages, they rarely evaluate the impact of their optimizations on the original content and functionality. Additionally, due to the lack of a unified metric to evaluate the similarity of the pages generated by these solutions in comparison to the original pages, it is not yet possible to fairly compare the results obtained from different user studies campaigns, unless recruiting the exact same users, which is extremely challenging. In this paper, we demonstrate the lack of qualitative evaluation metrics, and propose QLUE (QuaLitative Uniform Evaluation), a tool that automates the qualitative evaluation of web pages generated by web complexity solutions with respect to their original versions using computer vision. QLUE evaluates the content and the functionality of these pages separately using two metrics: QLUE’s Structural Similarity, to assess the former, and QLUE’s Functional Similarity to assess the latter—a task that is proven to be challenging for humans given the complex functional dependencies in modern pages. Our results show that QLUE computes comparable content and functional scores to those provided by humans. Specifically, 90% of 100 selected pages were given a similarity score between 90% and 100% by the human evaluators, while QLUE shows similar scores for more than 75% of the same pages. QLUE’s time complexity results show that it is capable of generating the scores for the same set of pages in a matter of few minutes.

CCS CONCEPTS
• Information systems → World Wide Web; • Computing methodologies → Computer vision.

KEYWORDS
QLUE, Uniform Qualitative Evaluation, Web Pages, Structural Similarity, Functional Similarity.
existing tools, in an automated manner with minor user intervention, supporting both desktop and mobile versions of web pages.

QLUE is evaluated by comparing the similarity scores it computes for 100 popular web pages to the scores given by 30 human evaluators who participated in a user study. These pages were taken from Alexa top million sites [12] and simplified using a state-of-the-art web complexity solution, namely SlimWeb [18]. Results show that QLUE exhibits similar trends in the overall similarity scores compared to the user study for both QSS and QFS. Specifically, QLUE achieves a QSS score of $\geq 90\%$ for more than 70% of the pages in contrast to 90% of the pages in case of human evaluation. While similar results are shown in terms of the QFS, a slightly wider gap can be seen between the scores generated by QLUE and the scores given by the human participants. These results can be explained by the systematic rules of QLUE in penalizing the score for every missing component in a given web page (regardless of its size or relevance). These rules make QLUE less forgiving in comparison to humans. In fact, QLUE’s scores provide a lower bound for a potential user study. The paper’s main contributions are:

- **QLUE**: A novel uniform unbiased computer-vision approach to evaluate the quality of web pages generated by web complexity solutions, using two scoring metrics: QSS and QFS.
- An evaluation of the QSS/QFS scores of 100 pages in comparison to scores given by 30 human participants of the user study.
- A time complexity evaluation of QLUE’s in terms of structural similarity processing and QFS computation.

## 2 MOTIVATION

Over the past decade, several solutions are proposed to tackle the web complexity problem from both the industry and academia standpoints. These solutions typically block or modify the pages’ contents to offer faster/smaller-sized pages, pre-process the pages by offloading complex processing tasks from the browser to a proxy, or restructure the page load process to avoid bottlenecks at the client [45].

More specifically, blocking solutions are often deployed in a from a of a browser [1, 5, 41] or in-browser extension [13, 14, 16, 23, 32, 42]. For example, Brave [5] was recently released as a mobile browser with built-in ad-blocking features, while Pervical [13] extends Google Chrome and Brave to block ads using deep learning. In the case of Opera Mini [41] browser, a proxy server renders web pages before sending them to the users. In contrast, JSCleaner [19] offers a proxy-based solution to block non-critical JavaScript in mobile pages to reduce the processing burdens imposed on low-end mobile devices. To offer simplified versions of existing pages, Facebook-Lite eliminates the secondary features of Facebook [25], whereas SpeedReader [26] converts pages that are suitable for the reader-mode into simpler reader-friendly pages. In a recent work [38], the authors proposed to eliminate less-useful functions from web page to improve the memory consumption on low-end devices. To save the users’ data plans, Flywheel [15] compresses responses between the servers and the browsers, while BrowseLite [33] applies different image compression techniques. To accelerate mobile pages without blocking any of their original content, both Shandian [46] and Prophecy [40] pre-process web pages on a server and then send modified versions to the users’ browsers, while Polaris [39] modifies the sequence in which the different page components are loaded.

As such, web complexity solutions often generate modified versions from existing pages, where the original content and/or functionality might be sacrificed due to optimizations in resources and processes. Thus, a structural and a functional assessment of the generated pages is crucial to ensure the similarity of these pages to their original versions. However, this assessment is either ignored [26, 39, 40, 43, 46] or evaluated with small-scaled user studies [19, 33] due to the challenges of conducting large scale user studies and the lack of alternative evaluation scenarios. For example, in SpeedReader [26], the authors noted that they did not attempt any user evaluation on the quality of their generated pages, and left a deployment plan with subjective presentation evaluation for future work. While SpeedReader cannot handle around 78% of web pages [2] (since it never fetches or executes JavaScript), we believe that a comprehensive large-scale analysis on the impact of SpeedReader on the pages content and functionality can help in covering a larger portion of pages. Similarly, and despite significant speedups, the impact of pre-processing in Shandian [46] and Prophecy [40] on the page structure and functionality have not been evaluated. Table 1 provides a comparison of the metrics used to evaluate web complexity solutions, with a note on the evaluation of the aforementioned impact. The table highlights the lack of such an evaluation in many of the web complexity solutions apart from a few, where small-scaled user studies are conducted. This motivates the need for a uniform and rapid evaluation for the content and functionality of the pages.

## 3 RELATED WORK

With the rapid development of web complexity solutions, existing tools assist developers in evaluating their pages in terms of timing and/or saving gains. For instance, Lighthouse [8] tool runs in Chrome DevTools to generate an evaluation report for a given page and how to improve its timing performance. Similarly, BrowserTime [34] collects timing metrics and records a video of the browser screen to calculate visual metrics such as Speed Index. A common framework

---

**Table 1: Evaluation metrics used in web complexity state-of-the-art solutions**

<table>
<thead>
<tr>
<th>Solution</th>
<th>Evaluation metrics</th>
<th>The impact on the page content/functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpeedReader</td>
<td>Data Size, Memory Consumption, Page Load Time</td>
<td>Not evaluated</td>
</tr>
<tr>
<td>WProf [45]</td>
<td>Page Load Time</td>
<td>Not evaluated</td>
</tr>
<tr>
<td>Shandian [46]</td>
<td>Page Load Time, Page Size</td>
<td>Not evaluated</td>
</tr>
<tr>
<td>Polaris [39]</td>
<td>Page Load Time</td>
<td>Not evaluated</td>
</tr>
<tr>
<td>Vroom [43]</td>
<td>Page Load Time</td>
<td>Not evaluated</td>
</tr>
<tr>
<td>Prophecy [40]</td>
<td>Page Load Time, Above the fold Time, SpeedIndex, ReadyIndex</td>
<td>Not evaluated</td>
</tr>
<tr>
<td>Flywheel [15]</td>
<td>Page Load Time, Time to first byte, Time to first paint, Page Size</td>
<td>Not required</td>
</tr>
<tr>
<td>BrowseLite [33]</td>
<td>Bandwidth Saving, Page Size, Speed Index</td>
<td>User Study, Visual Completeness</td>
</tr>
<tr>
<td>JSCleaner [19]</td>
<td>Page Load Time</td>
<td>User Study</td>
</tr>
<tr>
<td>Web Medic [38]</td>
<td>Memory consumption</td>
<td>User Study</td>
</tr>
</tbody>
</table>

With the rapid development of web complexity solutions, existing tools assist developers in evaluating their pages in terms of timing and/or saving gains. For instance, Lighthouse [8] tool runs in Chrome DevTools to generate an evaluation report for a given page and how to improve its timing performance. Similarly, BrowserTime [34] collects timing metrics and records a video of the browser screen to calculate visual metrics such as Speed Index. A common framework
to assist web developers in creating light-weight pages is Google’s Accelerated Mobile Pages (AMP) [27]. To compare the performance of an AMP page to the corresponding non-AMP page, a manual evaluation is expected [28]. Instead of creating new accelerated pages, developers commonly seek to reuse features in existing pages. In Ply [35, 36], a CSS inspection tool is proposed to assist developers in replicating visual features of existing complex pages. To automatically identify irrelevant properties of elements in these pages, Ply disables a given property, captures a screenshot of the result page and compares it to a reference screenshot with all properties enabled. It uses a simple pixel-level screenshot comparison to compute the visual regression between screenshots and remove the property with no visual impact. Unlike QLUE, Ply doesn’t consider interactive features driven by JavaScript.

Other developer tools [20, 24, 30, 37, 48] focus on debugging and tracking code snippets in web pages, or reproducing interactive content from existing pages. For instance, Chrome DevTools are extended in Unravel [30] to support reverse engineering of complex pages, by enabling developers to track and visualize both: page changes and JavaScript function calls. On the other hand, Fusion [48] allows to borrow functionalities from existing pages, by extracting components from these pages and turning them into self-contained widgets that can be embedded into other pages. In [20], a simple markup language is combined with reactive components to reduce the effort needed to produce interactive documents. In comparison to QLUE, these tools assume the intention of reusing content on different pages, while QLUE aims to assist developers who intended to provide alternative pages that accommodate the needs of users relying on handheld mobile devices to access the web.

4 DESIGN CONSIDERATIONS

QLUE aims to evaluate the content and the functionality of web pages generated by different web complexity solutions—referred to as the Modified pages, in comparison to their original counterpart pages—referred to as the Reference pages. QLUE introduces two metrics to evaluate a given modified page: content completeness, which we refer to as QLUE Structural Similarity QSS, and the degree to which the interactivity and the functional features are retained, which we refer to as QLUE Functional Similarity QFS.

Identify page’s individual components: QLUE aims to break a given page into individual components to achieve accurate similarity score. Using existing filters (such has OpenCV’s threshold filter), an image with several objects is not taken as one single component, but split into a number of separate components according to the embedded objects. To identify individual components and their locations within the page, QLUE aims to consider the entire images, since the modified page either maintains an entire given image or not (there won’t be a case where only a subset of the image is retained within the modified page). To this end, QLUE uses a custom threshold filter (Sec. 5.1.1). Figure 2 shows an example comparing the outcome of OpenCV’s adaptive threshold filter to QLUE’s custom filter.

Handle dependent events in assessing the QFS: A dependent event can be defined as an event that cannot be triggered unless a set of one or more prior events are triggered. For example, a sub-menu item can be clicked only when the icon of the main menu is first clicked, and then the corresponding menu item is clicked afterwards. The chances of dependent events increases as the interactivity of a given web page is increased. While some of these events might not be easily identified by a human evaluator—since they are hidden under a number of consecutive dependant events, QLUE utilizes a computer

Achieve accurate QSS scores: A direct comparison to assess the QSS of a modified web page with respect to a reference page might not yield an accurate score. This is because the modified page may lack a number of elements affecting the position of other elements within the page (given the dynamic nature of HTML that reorders the components based on their relative positions). Figure 1 shows two examples of how openCV’s Structural Similarity Index Measure (SSIM) fails in providing an accurate similarity score. In the first example, due to a missing banner, the modified version of the page has shifted the subsequent elements following the missing banner. As a result, SSIM reported a 61% similarity between the two pages (which is not accurate given that both pages look almost identical apart from the missing banner). On the other hand, the second example shows two different screenshots of different pages (where the modified version is not created from the same reference page), however the SSIM method reported a similarity score of 69% even though the elements within the pages are completely different (due to the large number of background pixels that match between the two pages). To handle this inaccuracy, QLUE breaks the pages into several components while eliminating the background pixels (Sec. 5.1.2). The score is then computed by matching these components individually (Sec. 5.1.3). It attempts to match each component in the reference page with a distinct component in the modified page.

Figure 1: OpenCV’s SSIM failure examples computing the score

(a) 61% SSIM similarity score, while intuitively a higher score is expected given that the pages look very similar
(b) 69% SSIM similarity score, while intuitively a score of 0% is expected given that the pages are different

Emulate human perception of the content similarity: QLUE emulates human perception when assessing the similarity of a given web page, by searching for missing components and penalizing the score based on the importance of these components. The importance of a given component is approximated by measuring the area it occupies in the page to reflect the human’s behavior in penalizing large missing components while overlooking small missing ones.

Uniformly evaluate the impact of web complexity solutions on the page content and functionality: To provide either an alternative when a user study is challenging, or a rapid evaluation prior to an assessment with real users, QLUE systematically emulates the human behavior in evaluating the similarity of the modified pages in comparison to their corresponding reference pages using computer vision. Since the impact of web complexity solutions can be seen either in the content/structure or the functionality of the modified page, the two aforementioned metrics are introduced to assess each aspect of this impact in 0-100% similarity score.

Identify page’s individual components: QLUE aims to break a given page into individual components to achieve accurate similarity score. Using existing filters (such has OpenCV’s threshold filter), an image with several objects is not taken as one single component, but split into a number of separate components according to the embedded objects. To identify individual components and their locations within the page, QLUE aims to consider the entire images, since the modified page either maintains an entire given image or not (there won’t be a case where only a subset of the image is retained within the modified page). To this end, QLUE uses a custom threshold filter (Sec.5.1.1). Figure 2 shows an example comparing the outcome of OpenCV’s adaptive threshold filter to QLUE’s custom filter.

Handle dependent events in assessing the QFS: A dependent event can be defined as an event that cannot be triggered unless a set of one or more prior events are triggered. For example, a sub-menu item can be clicked only when the icon of the main menu is first clicked, and then the corresponding menu item is clicked afterwards. The chances of dependent events increases as the interactivity of a given web page is increased. While some of these events might not be easily identified by a human evaluator—since they are hidden under a number of consecutive dependant events, QLUE utilizes a computer
bot that interacts with all the functional elements of the page to assess the missing functionality. The bot effectively considers a full set of dependent events in a given page, by automatically extracting a complete list of event-listeners and then triggering each of them.

5 THE DESIGN OF QLUE

5.1 QSS Assessment

The QSS assessment aims to compute a score that describes the level of content similarity in the modified page in comparison to a reference page. To compute the score, the QSS goes over various processes: i) screenshots generation and processing, ii) components extraction, iii) components matching, and finally iv) computing the score. Figure 3 shows a high-level architecture of QSS computation.

5.1.1 Screenshots Generation and Processing

To join the small islands together, QLUE uses OpenCV’s dilation function to dilate these pixel islands. Pixel Dilation is a morphological operation that traverses through the binary array and replaces all the 0s with 1s if one of the neighboring pixels is 1. Figure 4 shows an example of the dilation process over a textual paragraph, where the bounds of the neighborhood can be changed by modifying the kernel size. For example, for a kernel size of 2, 2, the 2x2 squares around the pixel in question will be checked. For QLUE, it is crucial to have a kernel size that captures islands in a way that avoids considering

Figure 2: An Example showing the impact of two different threshold filters on the same original screenshot.

Figure 3: QSS Evaluation

5.1.2 Components Extraction

To compute the score, the QSS goes over various processes: i) screenshots generation and processing, ii) components extraction, iii) components matching, and finally iv) computing the score. Figure 3 shows a high-level architecture of QSS computation. A step-by-step walk-through example of the QLUE’s structural similarity evaluation is provided in Appendix C, where Wikipedia mobile page is selected to highlight the impact of the individual processes.

The score can then be computed by matching these components individually (see Sec. 5.1.3). To identify the individual components, QLUE searches within the 2D array for groups of connected pixels (adjacent 1s in the array without 0s in between), we refer to these groups as islands. These islands can represent large elements such as images, or small elements such as alphabets in a sentence. For many of the small close-by islands, QLUE merges them into a larger island for two main reasons: i) to minimize the number of separate islands in the reference page, so that the time to search for them in the modified page is minimized, and b) to correlate repeated islands to their corresponding components. More specifically, islands that often repeat in a page (such as alphabets) are required to form a unique combination (such as a sentence or a paragraph island), to avoid false matches due to the presence of repetitive smaller islands, and ensure that the missing components are correctly penalized.

Figure 4: Dilation on a textual paragraph. The text in green is displayed for illustrative purposes highlighting the dilated pixels.
the entire screenshot as a single island or considering every alphabet as an island (see Figure 5 on the impact of the kernel size). The initial kernel size values are set by multiplying a constant with the ratio of the foreground and the background pixels in both the x and y directions. This constant depends on the type of the web page (mobile or desktop) and the display screen resolution. The initial value, $K$, of the kernel is defined as:

$$K = \left\{ K_x, K_y \right\} = \left\{ \begin{array}{ll} K_x = \frac{C}{\text{numRows} \times 1} \cdot \frac{\delta_i}{\gamma} \\ K_y = \frac{C}{\text{numCols} \times 1} \cdot \frac{\delta_i}{\gamma} \end{array} \right. \quad (1)$$

where $\delta$ is the number of pixels with value 0 at the $i^{th}$ row or column (depending on whether we are computing $k_x$ or $k_y$), and $\gamma$ is the number of pixels with value 1 at the $i^{th}$ row or column (depending on whether we are computing $k_x$ or $k_y$), and $C$ is a constant that inflates the probability of zero pixel into a larger dilation factor. $C$ is set to 15, which means that when the probability of the zero pixel per screenshot is low (when the page is dense in components), the kernel size would be around 1 to 2 pixels. On the other-hand, when the latter probability is high (when the page is sparse in components), the kernel size would be set to a higher value not exceeding $C$.

After the initial value of $K$ is computed for a given page using the above formula, QLUE dilates the pixel of the screenshot and computes the number of islands (disconnected components) using openCV’s connected components method (see Figure 3 green process, where the pixels of each connected component are labeled with a unique number from 1 to N). Based on an analysis of 100 modern web pages, we discovered that the average number of components per page varies in area between 4000 and 12000 pixels. To verify the initial value of our kernel $K$, we compare the number of components computed earlier with the theoretical upper and lower bounds, by dividing the screenshot area by both bounds of the average component area. That is $[(x \times y)/(12000)]$ and $[(x \times y)/4000]$, where $x$ and $y$ are the width and height of the 2D array. If the number of components is outside this range, the kernel size will be automatically tweaked depending on whether the number of components is lower or higher than the range. In case it is lower it suggests that the kernel size is larger than required and should be reduced, whereas it should be increased if the number of components exceeds the upper range. This kernel size is automatically increase/decreased by 1 pixel in each iteration and the dilation process is performed again until the number of components is within the range.

![Figure 5: The impact of different kernel size on pixels dilation](image)

Next, each of the components is grouped into a single island. Once the number of components is computed and within the range, the bounding box of each component is computed by finding the minimum and maximum $x$ and $y$ coordinates across all pixels of that component, i.e., $x_{\text{min}}, y_{\text{min}}, x_{\text{max}}, y_{\text{min}}, x_{\text{max}}, y_{\text{max}}$. QLUE stores the coordinates of each component, and uses them to extract the components images by cropping the original 3D array at the same coordinates of the bounding boxes. Figure 6 shows how the bounding boxes overlay over both the reference and the modified screenshots. Then, each of the pixels islands is identified, and each bounding-box is determined, where QLUE crops the components’ images of the reference page’s screenshot. All of the above steps are first applied on the reference page and then on the modified page. The outcome is a set of images for each page representing individual components in these pages and serving as the basis for components matching between the reference and the modified page.

![Figure 6: QLUE’s components matching example, where green boxes highlight the fully matched components, and the red boxes highlight the missing components of the modified page](image)

5.1.3 Components Matching. At this phase, QLUE has extracted all the individual components from both the reference and the modified pages in the form of images cropped from the 3D arrays of the reference and modified screenshots. These components are matched to identify which components are missing in the modified page. To achieve this goal, QLUE utilizes an image search algorithm called the image integrals [21] to match each component in the reference page with a component in the modified page.

For every successfully matched component, QLUE assigns a full similarity score of 100%. Figure 6 shows an example of the matched components, highlighted by the green bounded-boxes. For the components that are not found in the modified page, QLUE uses OpenCV’s similarity index and the mean squared error for each of the unmatched components from the reference page against each similarly-sized unmatched components from the modified page. Then, for a given unmatched component in the reference page, the component from the modified page with the highest score is considered a partial match with the score obtained from OpenCV’s similarity index which ranges between 0 - 100%. For a given reference page’s component with no match in the corresponding modified page (verified when QLUE runs out of components without exact or partial matches), a score of 0% is assigned (see the components with red bounded-boxes in Figure 6 for an example of unmatched components). The output of components matching is a list of matched components, each with an individual score between 0 - 100%.

Finally, QLUE iterates over all identified components in the reference page and searches for possible matches in the modified page. There will be three different categories of components: a) components that are fully matched with a score of 100%, b) partially matched components, and c) missing components with a 0% score.
5.1.4 Computing the Final Score. The individual scores computed in the precious step for each component in the reference page are combined into a unique value to represent the overall page QSS score. Since humans tend to forgive or overlook smaller components and penalize larger components, QLUE averages the scores weighing each component’s score by its relative area (scores are multiplied by the area of each component and divided by the overall area of all components) to emulate human’s perception of the pages similarity, where components contribute to the score in a proportion relative to their area. The QSS has a value between 0-100%, is computed as:

\[
\text{QSS Score} = \frac{\sum_{i=0}^{C} a_i \times s_i}{\sum_{i=0}^{C} a_i}
\]

where \( C \) is the number of components, \( s_i \) is the score of the component \( i \), and \( a_i \) is the area of component \( i \). A score of 100% affirms that the modified page is identical to the reference page.

5.2 QFS Assessment

This assessment refers to evaluating the modified page in terms of retaining the functional elements found in the corresponding reference page along with their proper interactivity features. For example, a drop-down interactive menu found in a reference page should exist and function properly in the corresponding modified page. To compute the QFS score, the user interactivity events are emulated through an interaction bot, by extracting all of the event listeners from both the reference and the modified page, triggering each event in each page, and capturing a screenshot whenever an event is triggered. Then, the QFS score is computed based on the retained functionality in the modified page with respect to the reference page.

5.2.1 Emulating User Interactivity. To emulate the user interactivity, QLUE leverages the browser’s built-in functionality to identify all the event listeners in the reference page, and map each of them to their corresponding page element (such as mapping a click event to the corresponding menu element). These events have various forms including but not limited to: mouseup, mousedown, mouseover, mouseout, keyup, keydown, drag, dragend.

Page elements are identified using their XPaths to facilitate navigating through a given web page and accessing all elements and their attributes to construct an event-dependency graph, such that a group of dependent events can be triggered in a given order. For example, an event that closes a menu cannot be triggered unless the open event corresponding to that menu is triggered first. In the HTML document of a given web page, elements are structured using \(<\text{div}>\) tags, where each tag has a unique “id”, with a potential reference to a “class” of attributes from the accompanying Cascading Styling Sheets (CSS). QLUE constructs the XPath of an element using the nearest parent with an “id” or “class” as the root. Position-based indexing is then used to identify descendants. For elements with no ancestor with an “id” or “class” attribute, the “body” element is used as the root, while position-based indexing is initiated from the body.

When the events of a given reference page are identified and mapped to the corresponding page elements, they are traversed in a depth-first order. An automated browser environment is used to trigger each event associated with a given element. Triggering an event often leads to changes in the page appearance (see a sample snapshot of a hover event over a drop-down menu in Figure 8). To assess if the original functionalities of the reference page are retained in the modified page, a screenshot is captured and stored whenever an event is triggered in the reference page (see examples of such events in Figure 7). To increase the efficiency of the QFS assessment, any screenshot that renders no visible change—in comparison to
the master screenshot of the page captured without any event being triggered—is not stored. This is confirmed by a pixel-by-pixel comparison, such that an event that does not change the appearance of the page is dropped from the list of events. For a given triggered event, QLUE saves only the area of the page that shows a visual change in comparison to the master screenshot. The final set of event listeners identified in the reference page along with their screenshots showing their impact are used to assess the functionality in the modified page, by searching for each of them in the modified page. If an event is found, it is triggered and a screenshot is captured. The output of the user interactivity emulation is a set of screenshots for each page representing the impact of each event on the page.

5.2.2 Computing the Score. QLUE iterates over all events screenshots captured for the reference page, and compares each of them to its counterpart screenshot captured for the modified page. QLUE has two different approaches when comparing these screenshots: a) using the same considered QSS approach (Sec. 5.1, or b) using openCV’s SSIM method for a quick similarity score computation. The choice between the two is left for the user, since this strikes a balance between computation accuracy and performance. Specifically, the former option is more accurate while the latter is tentatively faster but less accurate (Sec. 4 and the examples shown in Figure 1). However, in the case of an event screenshot, the possibility of having a missing element is rare, given that the functionality of the element associated with that event is either fully retained or completely missing. An event that exists in the reference page but not found in the corresponding modified page is given a score of zero. On the other hand, for a matched event, a score between 0 and 1 is computed to assess the similarity of the screenshot captured when that event is triggered in the modified page with respect to the corresponding screenshot captured when the same event is triggered in the reference page (using image subtraction). The average of the scores given to each of the events is computed to represent the overall QFS score of the modified page. The functions of a page are considered equally important and no weights are assigned to different events types.

6 IMPLEMENTATION

QSS Implementation: QLUE provides a flexible approach with three modes of operation depending on the source of the pages:
- **Proxy mode**: here, the users configure QLUE with two different proxies, one serving the reference pages, while the other serves the modified one. The use of proxies is a common approach for many of the web acceleration solutions, where the developer cache cloned versions of these pages to guarantee reproducible results.
- **URL mode**: in this mode, QLUE can be used to evaluate live web pages against each other without the need for caching. An example of this mode can be evaluating live production web pages against each other. It is configurable (and is set to 20 iterations by default). This number is introduced, given that certain web pages can virtually display endless content when scrolling. Then these screenshots are passed to the: pixel dilation, components extraction, components matching, and computing the final score. These are all implemented in Python using openCV [17], Scikit-Image [44], and numPy [29]. Users can run several parallel instances to speedup the evaluations.
- **Screenshots mode**: this mode is used to directly provide the pages screenshots (i.e., reference and modified). In contrast to the above two approaches, where QLUE generates the screenshots internally. For the screenshot generation, QLUE uses Selenium Chrome WebDriver [31], where depending on the need of the users, their solution, and their selected mode, QLUE can be configured to either emulate a desktop or a mobile phone chrome browser when generating the screenshots (valid for the first two modes of operation). It also runs as a headless browser, and automatically scrolls through the entire page in an iterative manner while waiting in each iteration for the content to be fully displayed. The maximum number of iterations is configurable (and is set to 20 iterations by default). This number is introduced, given that certain web pages can virtually display endless content when scrolling. Then these screenshots are passed to the: pixel dilation, components extraction, components matching, and computing the final score. These are all implemented in Python using openCV [17], Scikit-Image [44], and numPy [29]. Users can run several parallel instances to speedup the evaluations.

QFS Implementation: The QFS implementation has two main modules: an interactivity bot, and a score generation module. The interactivity bot emulates user actions on the web page by first extracting all the event-listeners from the DOM structure of the page. Given that all of the pages interactivity/functionality are triggered using an event-listener. The interactivity bot is implemented in Java, relying on Selenium. The bot can operate in two modes: **Proxy mode**, and **URL mode**. The score generation module is implemented in Python. The initial comparison to see if the events rendered any change, image subtraction, and image matching are performed using openCV [17] and Scikit-Image [44]. QLUE’s source-code can be found under https://github.com/comnetsAD/QLUE.

7 EVALUATIONS

7.1 User Study

To evaluate the effectiveness of QLUE on how well it emulates the human perception when comparing web pages against each other, we conducted a user study with 30 participants to compare 100 modified web pages created using SlimWeb [18] with respect to their original pages. We split the pages among participants, and asked each to evaluate 20 unique pages (each page evaluated by 6 participants). Participants were recruited from an international University who browse the web on a daily basis, without any assumptions on their web development experience. They were trained to manually evaluate the quality of the pages with respect to the original pages in terms of content completeness and retained functionality. We met with each participant online to explain the task and the evaluation tool in a 5-10-minute Zoom session. We also allow participants to ask questions about how to use the evaluation tool, but we did not share any evaluation expectations or details on the percentage of penalization they should consider for missing content or functionality.

7.1.1 Evaluation Tool and Metrics. We designed an evaluation tool that automatically picks a URL from the list of URLs that a participant is supposed to assess, and displays two pages side-by-side (the reference and the modified) in two instances of Chrome browser. The first browser connects to a proxy serving the reference pages, and the second browser connects to another proxy serving modified pages. We used proxy servers to serve the same cloned versions of pages for all users (to avoid the page regular updates over time). The evaluation tool randomly selects a web page and asks the user to compare the two versions and fill a form with the following considerations:
- **Human perceived content similarity score**: where a participant rates hers/his perceived content similarity of the modified page in comparison to the reference page using a slider with a 0-to-10 scale. A score of 0 is interpreted as the two pages being completely
different, whereas a score of 10 means that the two pages are identical. A score between 0 and 10 refers to a partial similarity.

- Human perceived functional similarity score: measures the participant’s perceived similarity of the modified page compared to the reference page from the functional completeness perspective. Participants are asked to rate their perceived functionality score in a 0-to-10 scale by manually assessing the modified page in terms of the presence as well as the operation of all functional elements found in the reference page, such as: interactive menus, navigational elements, search bars, and image scrollers, etc.

7.1.2 Evaluation Results. Figure 9 shows the histograms and the cumulative distribution functions (CDFs) of the structural similarity (Figure 9a) and the functional (Figure 9b) of the modified pages in comparison to the corresponding reference pages for both the user study and QLUE. The results shown in Figure 9a compares the automated structural similarity of QLUE (the light blue curve) to the manual human perceived structural similarity (the dark blue curve). The figure represents the scores given by the participant as percentages. The user study results show that for 90% of the pages, the participants gave a score $\geq 90\%$, whereas for the rest of the 10% of the pages, almost all (with the exception of two outliers) have a score $\geq 80\%$. In comparison, QLUE results show more conservative scores, where 75% of the pages have a score of $\geq 90\%$, while the rest (apart from 3 outliers) scores between 75%-90%. This highlights that QLUE is less forgiving than the human evaluators, evident by the smooth and gradual increase of the scores. This can be explained by the fact that QLUE systematic rules in penalizing the score for every missing component no matter how small it is, or how important is the component to the page main content. In summary, QLUE score can be considered as the lower bound of the page structural evaluation. Similar observations to the above can be viewed in QLUE’s functional comparison shown in Figure 9b. QLUE scores follow a similar trend to the scores given by human evaluators with slightly lower values. This can be explained by the fact that human evaluators tend to overlook minute differences, and that they are more forgiving in their assessment when major elements in the two pages are matching. In addition, human evaluators tend to miss evaluating certain functional elements, especially when they are triggered after triggering a series of previous dependant events.

7.2 Time Complexity

To evaluate QLUE’s time complexity, we compared the different timing metrics on per-process basis, using the same 100 pages considered in the user study. These metrics represent the timings of time-consuming processes in computing the QSS score: threshold filter, components extraction, components matching, and QSS total time. Figure 10a shows the CDFs of the QLUE’s timing metrics measured in seconds, whereas Figure 10b shows them as a function of the number of a page’s components. Figure 10a shows that the threshold filter doesn’t impact the overall time given that it is completed in a matter of milliseconds (hence the straight blue line), in comparison to the maximum total time of around 220 seconds in the worse case scenario. The Figure also shows that components matching (highlighted in orange) is the most time-consuming process in computing the QSS—taking around a minute at the median. In contrast, the components extraction process (highlighted in green) is relatively quick, with a maximum time of around 49 seconds. To better understand the relationship between the page complexity in terms of the number of components it contains and the timings metrics, we plot these metrics as a function of the number of components in each page, shown in Figure 10b. Results reveal that the relationship between QLUE’s overall time and the number of components in the page follows a linear trend. Additionally, results show that the total time required to perform the threshold filter is almost constant and does not depend on the number of components present in a page.

8 CONCLUSION

In this paper, we presented QLUE, a tool that aims to provide a unified approach for performing qualitative evaluations of web pages using computer vision. A user study of 30 participants has shown that QLUE computes comparable similarity scores to those provided by humans, and effectively assesses the retainment of web pages functionality. We envision that QLUE to used in two scenarios: a) uniformly comparing the quality of different web complexity solutions, b) as a built-in module within some of the aforementioned solutions. We reached out to the WebMedic [38] authors for an independent expert assessment (Appendix B).
We believe that it is crucial to consider all missing elements equally. We would like to thank Usama Naseer and the rest of the WebMedic authors for their valuable feedback and suggestions serving as independent expert evaluators for QLUE usability.

A HOW QLUE COMPARES TO HUMAN EVALUATORS

In Figure 11, we illustrate three examples on how the QSS scores computed by QLUE are comparable to the scores given by the human evaluators. In Figure 11a, we show that QLUE computes approximately the same score given by the human evaluators. The 2% reduction in QLUE’s score reflects the two missing menu elements (highlighted in red at the upper right corner of the reference page). QLUE identifies exactly the same missing images in the reference page as the human evaluators. Similarly, Figure 11b shows that QLUE identifies the same missing elements as the human evaluators, however, QLUE penalize the similarity score more accurately, since the size of the missing images spans a large area in the reference page. The reason why human evaluators gave a higher score than QLUE is that they attributed the missing images to advertisements (reported as non-important missing elements in the evaluation tool).

We believe that it is crucial to consider all missing elements equally regardless of their perceived category (e.g., advertisements) when measuring the qualitative score of the pages modified by web complexity solutions. This is because QLUE aims at providing a unified qualitative scoring metric to compare different web complexity solutions in a uniform and unbiased manner. Finally, Figure 11c shows the comparison of the BBC.com web page. Here, QLUE reports a similarity score that is 10% lower than the score given by the human evaluators due to the fact that it correctly identifies many missing navigation elements at the end of the page—that were overlooked by the human evaluators. This example highlights how easy it is for human evaluators to overlook many elements because it is very hard to recognize them when they have similar structural appearance.

B AN INDEPENDENT EXPERT ASSESSMENT ON QLUE’S USAGE

Given the specificity of QLUE, and to independently assess its usage potential, we sought the opinion of experts in the field, mainly authors of existing state-of-the-art web complexity solutions. Here, we present WebMedic’s authors [38] point of view on QLUE’s usability, not only as a final scoring mechanism of the overall page quality, but also as part of WebMedic’s internal algorithm—to be used for measuring the appearance metric of their page utility, instead of relying on the simple pHash algorithm. We shared the conceptual and technical details of QLUE with the authors offline and then conducted an interview over zoom with one of them. Below is a record of the interview transcript.

**Question:** Do you see the value of using QLUE as a module within the WebMedic framework?

**WebMedic author:** “Given that QLUE can accurately identify the user-centric importance of web page components, and by extension, the JavaScript that interacts with the given components, QLUE can be integrated into WebMedic as system module to automatically identify the key JavaScript functions without the need of conducting an independent expert assessment.

**Question:** How would the functional comparison in QLUE improve WebMedic accuracy in avoiding page breakage?

**WebMedic author:** “There are two cases for page breakage: a) a JavaScript function accessing some non-existent state, e.g., foo defines array, bar accesses the array, and if we cut foo without cutting bar then the array accessed in bar no longer exists, or b) an event listener attached to an element (e.g., button) gets removed,
thereby breaking the button. We currently instrument JavaScript to track every event listener added to the web page and can thereby track if an event listener is missing. In this case QLUE can help, since WebMedic only checks if the event listener exists or not. My only concern is that it might be very time consuming to run for too many pages, due to the fact that WebMedic cuts at functional level, and given that a web page might have a high number of functions (i.e., 1000 functions or above). If we make a different version of the web page for every missing function, we’ll have a large number of page variants. It is definitely useful but maybe beyond the initial operation of WebMedic, given that we have to test on permutations of all the JavaScript functions. However, we are on the evaluation phase of WebMedic after creating the candidate pages, QLUE’s functional comparison will be very helpful, given that QLUE takes it a step further from just checking if the event listeners exist or not.”

C QSS WALK-THROUGH EXAMPLE

Figure 12 shows a step-by-step walk-through example on how QLUE works using the Wikipedia page as an example.

D QLUE’S LIMITATIONS

Here, we show two three corner cases that represent the current limitations of QLUE. The first case occurs when a web page displays different images upon re-load. For instance, a web page that utilizes an image slider component may display a different image every time the page is loaded (either in the next visit or when the page is refreshed). In this case, there is a high chance that the screenshot captured by QLUE for the modified page would have a different image in the slider component than the image shown on the corresponding reference page. While this difference should not penalize the QSS score, the current implementation of QLUE does not recognize such cases, hence, the final score will be unnecessarily impacted. An extended version of QLUE can overcome this limitation by predicting components with changeable content, such as image sliders and advertising containers, where QLUE can take multiple screenshots of the page in order to collect all possible images.

The second corner case that QLUE does not automatically handle is a web page that uses a floating banner which always appears as the user scrolls. This poses a challenge in the screenshot generation process because the banner would appear multiple times in the full screenshot captured for the page, unless the user fixes the banner location to appear only at the top of the page. This can manually be handled by checking for such case and modifying the CSS styling of these banners before proceeding with the screenshot generation. In our future work, we plan to extend QLUE screenshot generation to automatically detect such floating elements and modifying their CSS styling accordingly.

QLUE is capable of evaluating search bars functionality in web pages, by filling the search bar with a search query and triggering the search event. Given that the page would return a valid visual response to the search query that QLUE can compare between the two versions of the page (i.e., the modified and the reference page). However, general web forms, although similar in spirit to search bars, are not handled by the current implementation of QLUE’s functional comparison. The reason behind this is the fact that most of the responses triggered by submitting a form do not necessarily reveal whether the filled data were properly sent to the server or not (apart from a simple thank you message that is usually displayed as a default response).