XBIDetective: Leveraging Vision Language Models for Identifying Cross-Browser Visual Inconsistencies

Balreet Grewal[1], James Graham[2], Jeff Muizelaar[2], Jan Odvárko[3], Suhaib Mujahid[2], Marco Castelluccio[2], Cor-Paul Bezemer[1]

[1]University of Alberta [2]Mozilla Corporation [3]Independent Researcher

[1]{balreet, bezemer}@ualberta.ca [2]{jgraham, jmuizelaar, smujahid, mcastelluccio}@mozilla.com [3]odvarko@gmail.com

Abstract

Browser rendering bugs can be challenging to detect for browser developers, as they may be triggered by very specific conditions that are exhibited on only a very small subset of websites. Cross-browser inconsistencies (XBIs), variations in how a website is interpreted and displayed on different browsers, can be helpful guides to detect such rendering bugs. Although visual and Document Object Model (DOM)-based analysis techniques exist for detecting XBIs, they often struggle with dynamic and interactive elements. In this study, we discuss our industry experience with using vision language models (VLMs) to identify XBIs. We present the `XBIDetective` tool which automatically captures screenshots of a website in Mozilla Firefox and Google Chrome, and analyzes them with a VLM for XBIs. We evaluate `XBIDetective`'s performance with an off-the-shelf and a fine-tuned VLM on 1,052 websites. We show that `XBIDetective` can identify cross-browser discrepancies with 79% accuracy and detect dynamic elements and advertisements with 84% and 85% accuracy, respectively, when using the fine-tuned VLM. We discuss important lessons learned, and we present several potential practical use cases for `XBIDetective`, including automated regression testing, large-scale monitoring of websites, and rapid triaging of XBI bug reports.

CCS Concepts

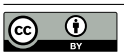
• **Computing methodologies** → **Matching**; *Intelligent agents*; Information extraction; • **Information systems** → *Browsers*.

Keywords

AI4SE, Cross-Browser Compatibility, Web Browsers

ACM Reference Format:

Balreet Grewal, Marco Castelluccio, Suhaib Mujahid, Jeff Muizelaar, James Graham, Jan Odvárko, and Cor-Paul Bezemer. 2026. `XBIDetective`: Leveraging Vision Language Models for Identifying Cross-Browser Visual Inconsistencies. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-SEIP '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3786583.3786892



ICSE-SEIP '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2426-8/2026/04
https://doi.org/10.1145/3786583.3786892

1 Introduction

Cross-browser inconsistencies (XBIs) occur when the same website renders differently across browsers due to how the browser may interpret or display a website's source code depending on the architecture of the browser [26]. Even with web standards intended to unify behaviour, differences in implementation details, feature support, or CSS and JavaScript handling remain, and these can act as signals of deeper rendering bugs in browsers. XBIs can range from subtle layout shifts to complete unavailability of a website [26] making them valuable indicators for detecting rendering bugs. For browser developers, ensuring broad website compatibility is crucial, but the process is time-consuming, especially when rendering bugs are triggered only under certain conditions. Catching these bugs before a browser update can significantly reduce post-release issues, and automated approaches offer a solution to achieve this.

Most automated approaches for detecting XBIs involve computer vision techniques [5, 6, 29] or DOM (Document Object Model) analysis [4, 44]. However, computer vision approaches face challenges with variable element detection [30], and DOM model analysis may not capture all elements of a website such as HMTL5 `<canvas>` elements [20]. Most XBI detection techniques are relatively dated, likely due to their limitations in handling variable or interactive elements, such as dynamic elements or advertisements, which have become increasingly common in modern websites. With recent advances in vision language models (VLMs), it may now be possible to revisit XBI detection in ways that overcome prior limitations, particularly with variable and dynamic elements, and make the techniques more practical for use on real-world websites.

This paper proposes `XBIDetective`, a tool for leveraging VLMs to detect XBIs in websites rendered on different browsers. Specifically, to detect XBIs, `XBIDetective` takes two screenshots of the same website in Mozilla Firefox [1] and Google Chrome [2], and then prompts a VLM to identify XBIs.

We evaluated `XBIDetective` on 1,052 bug reports of potential cross-browser inconsistencies, comparing its results with the ground truth from the reports. Using both an off-the-shelf thinking VLM and a fine-tuned VLM, `XBIDetective` achieved precision scores of 77% and 79%, respectively, for identifying XBIs. Both versions of `XBIDetective` also identify dynamic elements and advertisements with high accuracy. In a larger-scale analysis of 1,695 websites, `XBIDetective` correctly ignored changing advertisements but struggled with dynamic elements that changed on each reload and with pop-up elements.

[1]https://www.firefox.com
[2]https://www.google.com/chrome/

This study demonstrates that VLMs can meaningfully analyze visual differences in website renderings by leveraging their ability to process both visual and textual information. Thus, browser developers can leverage `XBIDetective` to aide in identifying XBIs, which in turn can help to identify potential breakages before a browser update is made available to users. In summary, our main contributions are as follows:

- A demonstration of how VLMs can capture XBIs by comparing website renderings.
- The `XBIDetective` tool to capture and identify XBIs in websites loaded in different browsers available at [9].
- A discussion of the lessons learned of running `XBIDetective` in a large-scale analysis of 1,695 websites.

The rest of the paper further describes the study with Section 2 discussing related work. Section 3 covers a motivational study. Sections 4 and 5 cover the methodology for `XBIDetective` and the experimental setup of the study. Sections 6 and 7 present the experimental results and lessons learned. Section 8 discusses threats to validity of the study. Finally, Section 9 concludes the paper.

2 Related Work

2.1 Detecting cross-browser inconsistencies

Sabaren et al. [30] conducted a literature review on cross-browser inconsistency tools and found that most research focuses on techniques such as DOM model analysis, visual analysis, navigation model analysis, record/replay, static analysis, attribute comparison, and heuristic comparison. The authors highlight the challenges with these approaches; for instance, computer vision techniques struggle with detecting variable elements, like image carousels, and face difficulties in capturing accurate screenshots. DOM model analysis is challenged by interactive elements, different DOM models of the same webpage, and security measures that complicate DOM extraction. Navigation model analysis faces challenges with trigger state changes, unreachable states, and interactive elements.

Many of the current tools consist of combining visual analysis with the other techniques for desktop [4–6, 29] and mobile browsers [31, 43, 44]. For example, Watanabe et al. [44], building on their previous work [43], proposed a classification model that combines features of DOM-based analysis and computer vision techniques. Their approach, applied to mobile browsers, reports higher F1-scores for identifying external and internal layout failures.

Further, research has explored identifying the causes of XBIs [18, 22, 24, 25, 46]. Notably, Xu et al. [46] propose X-Diag, an automated technique that aims to find the root cause of XBIs by checking if the inconsistency is caused by incompatible DOM APIs, CSS properties, or HTML elements. X-Diag achieves a precision of 89% in identifying a root cause for an inconsistency between browsers, with a median runtime of 7.95 seconds. Currently, $\mathcal{X}$fix, a tool proposed by Mahajan et al. [22], is one of the only automated techniques that generates repairs for XBIs. $\mathcal{X}$fix resolves a median of 93% of XBIs reported by X-PERT [28], an XBI detection approach.

Some techniques have also explored detecting rendering bugs in browsers primarily using fuzzing [32, 33, 48], but it is not vastly explored. Recently, Zhou et al. [48] proposed Janus, a practical fuzzer that relies on Visual Delta Consistency, a test oracle. The intuition in the test oracle is that changes to an HTML file should be rendered either the same or differently by all browsers. Janus detects 31 non-crash rendering bugs, with 8 being fixed by developers.

To the best of our knowledge, our study is the first to explore the use of visual language models as a tool for identifying content and structural XBIs. The objective of this study is to evaluate whether a tool with current state-of-the-art VLMs can replace traditional XBI detection techniques and serve as a viable tool for finding XBIs linked to underlying rendering bugs. We also evaluate `XBIDetective` on real-world websites to assess its usability and effectiveness in practical scenarios.

2.2 Visual analysis of software

The rapid advancement of VLMs has led to their growing use for game bug detection [35, 36, 38]. For example, Lu et al. [19] utilize GPT-4o to rank keyframes of a gameplay video based on how closely it matches a textual bug description. Their approach provides a method for reducing manual effort of quality assurance teams by providing an automated bug retrieval pipeline.

Similarly, VLMs have been applied to bug detection and testing of web applications [7, 11, 13, 17, 20]. In particular, Wang et al. [42] propose VETL, an end-to-end vision language model (VLM)-driven web testing technique that consists of two components: a text input generator and a target element selector. VETL effectively explores web state/action spaces and detects functional bugs, exposing issues in top-ranking commercial websites.

Further, visual analysis techniques are used in various areas of research such as regression testing [39, 41], web page testing [1, 21, 23, 34], and game testing [12, 27, 37, 40].

In game testing, Paduraru et al. [27] suggest computer vision techniques for testing games. The authors point out that some current methods for aspects of game testing included using Tesseract OCR from OpenCV[3] for textual recognition, scene segmentation or template matching for output image recognition, and OpenPose [3] for animation testing. The authors further observe that automated agents for checking visual results can be effective, given that human testers are susceptible to errors. For web testing, Mahajan et al. [23] present a computer vision–based technique that detects and localizes presentation failures in web pages by identifying difference pixels to locate faulty HTML elements. To deal with dynamic text or images, the technique allows developers to specify those regions. Overall, the technique was able to identify 100% of presentation failures and locate the faulty element in 93% of cases.

For mobile applications, research has investigated the use of computer vision techniques to identify display issues in the UI of applications [2, 15, 16, 45, 47] and to support mobile UI testing [8, 14]. For example, Liu et al, [16] propose Nighthawk, a fully automated approach to detect GUIs with display issues, and locate the region of the issue in a GUI.

In line with these applications, we investigate the use of a VLM for identifying visual XBIs. Our approach leverages screenshots of the same website rendered in different browsers, using the VLM's image understanding capabilities to identify potential inconsistencies. While techniques like VETL apply VLMs to support web application developers in testing the functionality of a site's GUI,

[3] https://opencv.org

our focus is instead on assisting browser developers by detecting XBIs that may point to underlying rendering bugs. Hence, comparing our work with prior research experimentally is difficult, since the expected output of the approaches is different.

3 Motivational Study

We begin by investigating how well a VLM model can can effectively identify continuously changing elements such as dynamic elements and advertisements on a website page. As stated by Sabaren et al. [30], most computer vision techniques for XBI identification struggle with variable elements such as image carousels; we aim to assess if limitations of traditional image detection methods can be overcome by leveraging a VLM.

We used screenshots captured of a list of 1,052 websites in Mozilla Firefox and Google Chrome. As described in Section 4 we took five screenshots of each website and overlaid them onto each other. We then prompted Gemini 2.0 Flash (VLM_{base}) and Gemini 2.0 Flash Thinking ($VLM_{thinking}$) to identify advertisements and dynamic elements as shown in Listing 1 and Listing 2, respectively.

We also fine-tuned VLM_{base} on a sample of 88 bug reports to create two separate models: one for detecting advertisements and one for detecting dynamic elements to assess whether the model can perform similar to the thinking model.

Listing 1: Prompt template used to instruct VLM to identify advertisements.

```
Two renderings of the same website are provided, displayed in Chrome (
    image_1) and Firefox (image_2). Please analyze each rendering and
    focus on identifying any advertisements (excluding pop-ups) that
    might be present in either rendering. Answer the following
    question after doing your analysis:

1. Are there advertisement(s) in either rendering (not including pop-ups)
    ? (Answer Yes or No) If there is, where is it?

Here is an example using the following two renderings:
<Examples 1-5>

Now, it is your turn to idenitfy advertisements in the Chrome rendering (
    image_1) and the Firefox rendering (image_2) as described. Please
    number your answer as:
1.
```

We evaluated the VLMs' performance using precision, recall, and accuracy based on the model's "Yes" or "No" responses when identifying the presence of dynamic elements or advertisements, compared to the ground truth labelled by the first author.

Findings: **VLM_{base} achieves an accuracy of 86% at identifying advertisements in the screenshots of websites.** As shown in Table 1, $VLM_{thinking}$ and $VLM_{fine\text{-}tuned}$ achieve a slightly lower accuracy of 85%. Overall, the thinking and non-thinking, and fine-tuned VLMs perform similarly, though recall decreases for $VLM_{thinking}$, and $VLM_{fine\text{-}tuned}$ has a notably lower precision. We also observe that $VLM_{thinking}$ often hallucinates, incorrectly identifying advertisements in multiple sections of a website. Additionally, we find that $VLM_{thinking}$ struggles to recognize advertisement placeholders such as grey boxes labelled as "ads" that indicate an ad slot but may not contain a loaded ad. The lower precision of $VLM_{fine\text{-}tuned}$ reflects its higher number of false positives (97), suggesting that it frequently hallucinates the presence of advertisements Overall, all three VLM versions perform similarly at advertisement detection.

Listing 2: Prompt template used to instruct VLM to identify dynamic elements.

```
Two renderings of the same website are provided, displayed in Chrome (
    image_1) and Firefox (image_2). Please analyze each rendering and
    focus on identifying dynamic elements (sliders, carousels, progress
    bars, videos, dynamic graphs or charts, personalized
    recommendations, location-based recommendations, and real-time
    content) present in either rendering, excluding pop-ups. Answer
    the following question after doing your analysis:

1. Are there any type of dynamic element(s) (only look for sliders,
    carousels, progress bars, videos, dynamic graphs or charts,
    personalized recommendations, location-based recommendations,
    and real-time content) in either rendering? (Answer Yes or No) If
    there is, where is it? (do not include pop-ups)

Here is an example using the following two renderings:
<Examples 1-5>

Now, it is your turn to idenitfy dynamic elements in the Chrome
    rendering (image_1) and the Firefox rendering (image_2) as
    described. Please number your answer as:
1.
```

$VLM_{thinking}$ achieves an accuracy and recall value of 90% and 93% respectively at identifying dynamic elements in web renderings. The precision achieved by the model is 89%. As seen in Table 1, VLM_{base} and $VLM_{fine\text{-}tuned}$ achieve a lower accuracy of 83% and 84%, respectively, indicating that $VLM_{thinking}$ performs much better at identifying dynamic elements. The results from $VLM_{thinking}$ contain 28 false negatives from which 12 (43%) are caused by the model misidentifying real-time based content such as a list of trending news stories. $VLM_{thinking}$ also does not correctly identify 4 (14%) content carousels, 4 changing background images in websites, and 4 video players. While $VLM_{fine\text{-}tuned}$ does not match the performance of $VLM_{thinking}$ on this task, it outperforms VLM_{base}, indicating that fine-tuning improves detection accuracy. Analyzing the false positives made by $VLM_{fine\text{-}tuned}$, we find that the model misses 15 instances (37%) of video players and 5 instances (12%) of carousels. However, compared to $VLM_{thinking}$, $VLM_{fine\text{-}tuned}$ performs better at detecting live content such as news articles, missing only 3 instances.

We find that, for the most part, $VLM_{thinking}$ performs better than VLM_{base} and $VLM_{fine\text{-}tuned}$ at identifying advertisements and dynamic elements. Overall the results show that VLMs can reliably detect advertisements and dynamic elements without misclassifying them as cross-browser inconsistencies. This is encouraging, as prior work [30] has shown that computer vision techniques often struggle to correctly recognize these variable elements, leading to false positives.

Task	VLM version	Accuracy	Precision	Recall
Advertisement detection	VLM_{base}	86%	70%	90%
	$VLM_{thinking}$	85%	72%	76%
	$VLM_{fine\text{-}tuned}$	85%	67%	98%
Dynamic element detection	VLM_{base}	83%	80%	92%
	$VLM_{thinking}$	90%	89%	93%
	$VLM_{fine\text{-}tuned}$	84%	83%	89%

Table 1: Experimental results for the base, thinking and fine-tuned versions of the VLM at detecting advertisements and dynamic elements

While $VLM_{thinking}$ outperforms $VLM_{fine\text{-}tuned}$, there are still scenarios where the fine-tuned model may be preferable. Fine-tuning can be more cost-effective depending on the number and frequency of prompts to the VLM. Moreover, fine-tuning on a larger and more diverse dataset could mitigate the lower precision values as seen in Table 1 by exposing the model to a broader range of examples and cases. However, assembling sufficiently large and diverse datasets is costly and time-consuming, and poor fine-tuning practices may still bias the model toward the training data.

Takeaway: *We find that VLMs can reliably detect advertisements and dynamic elements, addressing the limitations of earlier computer vision techniques for XBI detection. While thinking models achieve the highest performance, fine-tuned models offer comparable accuracy at lower cost, making them a practical choice for large-scale or continuous XBI monitoring.*

4 Detecting XBIs with `XBIDetective`

`XBIDetective` consists of two stages, as seen in Figure 1: capturing screenshots from a list of websites, and prompting the VLM for XBI identification. We explain both portions below.

4.1 Capturing screenshots

To capture full-page screenshots of the websites, we use Selenium[4], a web browser testing tool. We capture website screenshots in two browsers running in headless mode. Five screenshots of each site are taken and overlaid to differentiate dynamic elements (e.g., image carousels) from static elements (e.g., backgrounds). For example, as shown in Figure 2, we take screenshots at one-second intervals to capture changes in the main carousel of a website displaying video suggestions. The overlay on the right merges instances where the carousel is in transition, allowing the VLM, when prompted, to recognize the element as not static by observing the change.

4.2 Identifying XBIs

A VLM that supports multiple images can be prompted to identify XBIs by providing two screenshots of the same website (cropped to the same size) rendered in different web browsers (or the same for regression testing). There are three stages to identifying XBIs.

4.2.1 Stage 1: Advertisement detection: In the first stage, advertisements are identified in the screenshots. Identifying advertisements ensures that we can avoid marking them as XBIs when identifying them later on. The prompt that we used is seen in Listing 1.

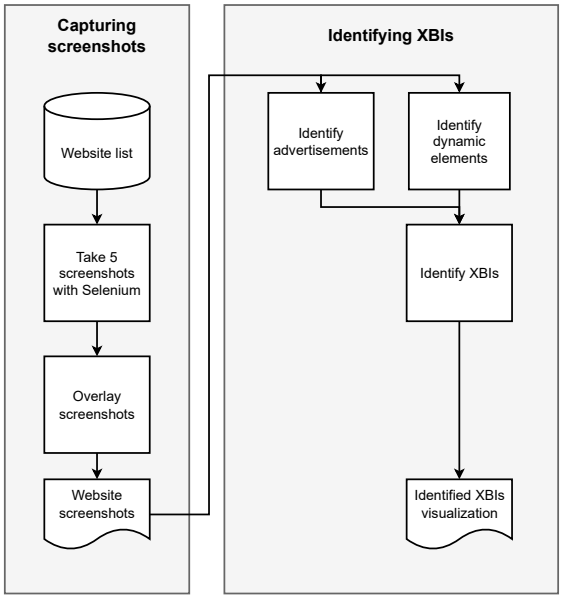


Figure 1: Overview of `XBIDetective`.

4.2.2 Stage 2: Dynamic element detection: In the second stage, the following dynamic elements are identified in the screenshots: sliders, carousels, progress bars, videos, dynamic graphs or charts, personalized recommendations, location-based recommendations, and real-time content. These elements are excluded from XBI detection because, while they may change during website rendering, such changes do not reflect inconsistencies between two websites. The VLM prompt we used is shown in Listing 2.

4.2.3 Stage 3: XBI detection: In the final stage, XBIs are identified while ignoring the advertisements and dynamic elements detected in the previous stages. Listing 3 shows the prompt we used, which includes examples to help the model understand the task. During this stage, an impact score is also assigned to each identified XBI, categorizing it into one of four severity levels. These impact scores

[4]https://www.selenium.dev

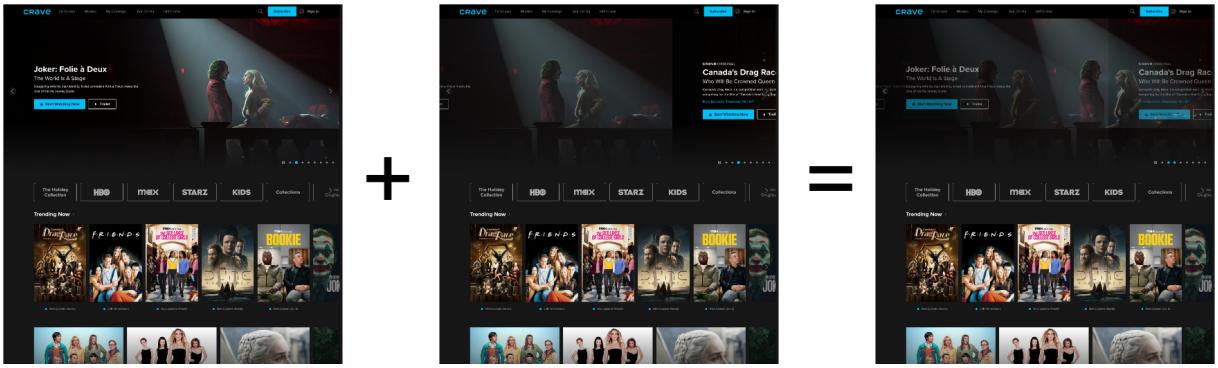


Figure 2: Example of overlay process with two screenshots taken of a website (https://www.crave.ca/en) with a dynamically changing carousel. Note that only 2 of the 5 screenshots used for the overlay are shown for brevity.

Listing 3: Prompt template used to instruct VLM to identify XBIs between browsers.

```
Two renderings of the same website are provided, displayed in Chrome (
    image_1) and Firefox (image_2).

In terms of advertisements, this is what was found: <prompt 1 output>

In terms of dynamic elements, this is what was found: <prompt 2 ouput>

Please analyze each rendering and focus on comparing each rendering in
    terms of layout, font style and size, colour consistency, alignment,
    and the presence or absence of elements (such as buttons, text
    fields, advertisements, pop-ups and images). Ensure to analyze
    beyond a pop-up (if applicable). Ignore the advertisements and
    dynamic elements identified as mentioned. Answer the following
    questions after doing your comparison:

1. Is there any functionally or perceptually meaningful difference
    between the renderings (ignoring the advertisements for question 1
     and ignoring the dynamic elements found in question 2 unless
    they cause a meaningful difference between renderings)?
    INCLUDING the presence or absence of pop-ups? (Answer Yes or
    No.)
If your answer was yes to the question 1, then answer these questions:
2. What specific element(s) are affected?
3. How do the renderings differ?
4. What would be the impact be? The impact can be evaluated by
    selecting the relevant label. The labels and their definitions are:
(blocked-unsupported) ... (significant-visual) ... (minor-visual) ...

Here is an example using the following two renderings:
<Examples 1-5>

Now, it is your turn to compare the Chrome rendering (image_1) and the
    Firefox rendering (image_2) as described. Please ensure to number
    your answers as:
<1.-4.>

Remember only answer questions 2-4 if your answer to 1 was yes and
    think about the impact while analyzing the renderings.
```

provide Mozilla developers with guidance on which XBIs to prioritize for bug analysis. The impact scores, originally used internally at Mozilla and adapted for the task of XBI detection, are as follows:

- **minor-visual**: the site has an XBI, but it does not affect the content or functionality of the site, and users are unlikely to notice. Some examples include different focus outlines on elements, small discrepancies in text rendering such as font that does not comprise readability, slight misalignments, or different background colours.
- **significant-visual**: the entire site does not load, the site loads but is effectively unusable, the site has visible layout problems, some parts of the page content (text, images, videos, or pop-ups) are missing or hard to access, or some features of the site are missing or broken. Some examples include, an entirely blank page, missing copy/paste buttons in a text editor, missing a pop-up on the website, or a layout that renders the website as unreadable.
- **blocked-unsupported**: there is a message indicating the browser is not supported. This is considered an XBI because a rendering bug might be preventing the website from displaying, even though it should be accessible to users. Additionally, the browser may be unsupported due to site requirements or rendering bugs that could potentially be addressed by the browser developer.
- **no-XBI**: no XBIs are observed.

Once the XBIs are identified, we generate an HTML-based visualization of the VLM's results for developers.

5 Experimental setup

Below we describe the experimental setup, where we collected and verified web compatibility bug reports to evaluate the effectiveness of XBIDetective at identifying XBIs. The overview of the experimental setup can be seen in Figure 3.

5.1 Collecting web compatibility bugs

To conduct an evaluation of XBIs, we collected a list of websites and their corresponding bug reports of web compatibility issues from Bugzilla[5], an issue tracker for Firefox. The collected bugs

[5]https://bugzilla.mozilla.org/home

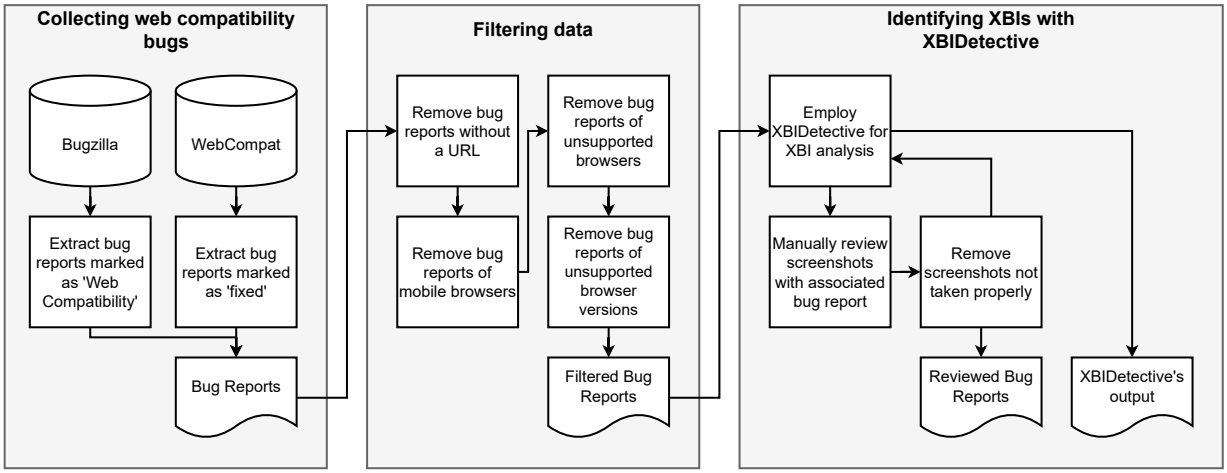


Figure 3: Overview of experimental setup.

are those marked as web compatibility issues in Bugzilla and that have undergone triage. Additionally, we gathered websites and bug reports from WebCompat[6], a website dedicated to reporting web compatibility bugs. The web compatibility issues are collected from the 'fixed' milestone in the WebCompat GitHub repository[7]. The web compatibility tags used by both websites refer to reported XBIs for Mozilla Firefox (in both Bugzilla and WebCompat) as well as for other browsers (in WebCompat). By selecting triaged Bugzilla bugs and WebCompat bugs marked as fixed, we ensured that all collected reports corresponded to verified XBIs.

We extracted the following fields from each bug report:

- **BugID**: the unique ID of the bug.
- **URL**: the URL of the affected website.
- **Browser/version**: the browser and version where the issue occurs (some reports may lack version details).
- **Summary**: a description of the bug.
- **Impact score**: If available for some Bugzilla reports, the severity of the bug as determined by the WebCompat team, used for the ground truth.

Although WebCompat allows reporters to submit screenshots, these are not included in the dataset. In total, we collected 4,725 web compatibility bug reports for analysis.

5.2 Filtering data

We first removed bug reports that do not provide a website URL. To ensure compatibility with Selenium, bug reports collected from WebCompat were filtered to exclude mobile browsers, and all browsers except Mozilla Firefox. To maintain relevance, we also excluded reports referencing Firefox versions older than 100, given that the latest version is 143. After filtering, 1,052 bug reports remain.

5.3 Identifying XBIs with `XBIDetective`

For this experiment, we used Selenium with Mozilla Firefox and Google Chrome. For bug reports for older Firefox versions, we reverted Selenium to that version to capture the screenshot to increase the chances of reproducing the bug.

To verify the quality of the screenshots from Selenium, the first author manually reviewed the screenshots with respect to the following criteria in order to prepare a ground truth:

- Proper website loading and full-page capture.
- Any XBIs found that were not originally mentioned in the bug report are appended to the report.
- The presence of advertisements.
- The presence of the following dynamic elements: sliders, carousels, progress bars, videos, dynamic graphs or charts, personalized recommendations, location-based recommendations, and real-time content.
- An impact score is assigned to each bug report if one is not already specified. The impact scoring process was calibrated in consultation with Mozilla developers.

After manual analysis, there were screenshots from 243 websites that contained an XBI. 538 screenshots were the same across browsers, likely because the underlying XBIs had been resolved, and 271 screenshots were deemed unusable, e.g., due to being blocked by bot detectors.

We evaluated three VLMs: Gemini 2.0 Flash, Gemini 2.0 Flash Thinking, and a fine-tuned variant of Gemini 2.0 Flash. This comparison allows us to assess the performance of a base model, a thinking model, and a fine-tuned base model in identifying XBIs. Thinking models may provide better results due to their extended reasoning capabilities, whereas base models are often more cost-effective. Finally, we analyzed whether a fine-tuned base model can achieve performance comparable to that of a thinking model.

To fine-tune Gemini 2.0 Flash, we used a statistically representative sample of 88 randomly selected bug reports (with 90% confidence and a 10% margin of error) from the reviewed reports described above. Supervised fine-tuning was performed using the prompt template shown in Figure 2. We omitted the bug reports used for fine-tuning during the rest of the experiments for the fine-tuned model.

[6]https://webcompat.com
[7]https://github.com/webcompat/web-bugs

We evaluated the three VLM's performance using precision, recall, and accuracy based on the model's predicted impact scores with the impact score assigned by the first author. The metrics are defined as follows:

$$\text{Precision} = \frac{1}{N}\sum_{i=1}^{N}\frac{\text{TP}_i}{\text{TP}_i + \text{FP}_i}$$

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

$$\text{Recall} = \frac{1}{N}\sum_{i=1}^{N}\frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i}$$

Where N is the 4 impact labels. The components of the confusion matrix are defined as follows:

- **True Positive (TP):** The model assigned the correct impact score denoting the presence of an XBI to a bug report.
- **True Negative (TN):** The model correctly identified the bug report as not containing an XBI.
- **False Positive (FP):** The model identifies an XBI and assigns an impact score when there is no XBI.
- **False Negative (FN):** The model identifies that a bug report has no XBI when it does contain an XBI.

The first author manually compared Gemini 2.0 Flash Thinking 's textual output describing the identified XBIs (true positives) to the ground truth to determine the number of correctly detected XBIs.

To evaluate XBIDetective on a broader dataset of websites, we collected a secondary dataset of 1,695 websites consisting of the top 1,000 websites with the highest number of reported bugs and 695 bug reports from WebCompat. For this dataset, we did not manually filter screenshots or establish a ground truth. We then used the fine-tuned version of XBIDetective to analyze the screenshots and detect XBIs.

For the remainder of the paper XBIDetective with the use of Gemini 2.0 Flash as the VLM will be referred to as $\texttt{XBIDetective}_{\text{base}}$, XBIDetective with Gemini 2.0 Flash Thinking as $\texttt{XBIDetective}_{\text{thinking}}$, and XBIDetective with the use of the fine-tuned version of Gemini 2.0 Flash as the VLM will be referred to as $\texttt{XBIDetective}_{\text{fine-tuned}}$.

6 Experimental Results

$\texttt{XBIDetective}_{\text{fine-tuned}}$ achieves an accuracy and precision of 79%, and 72%, respectively at labelling the impact scores. The recall achieved by the fine-tuned XBIDetective is 59%. Whereas, $\texttt{XBIDetective}_{\text{thinking}}$ achieves an accuracy and precision of 77%, and 69%, respectively at labelling the impact score of XBIs. The recall achieved by $\texttt{XBIDetective}_{\text{thinking}}$ is 48%. In comparison, as shown in Table 2, $\texttt{XBIDetective}_{\text{base}}$ performs considerably worse, achieving a precision of 42% and producing many more incorrect classifications, as shown in Figure 4. As seen from the confusion matrix in Figure 5, $\texttt{XBIDetective}_{\text{thinking}}$ incorrectly classifies 121 web renderings as "no-XBI" despite the presence of XBIs. Since false negatives are the most frequent type of error by the model, we examine them in more detail to understand which discrepancies the model is likely to miss. The false negatives primarily involve missed XBIs in layout differences (34 instances), the presence of pop-ups

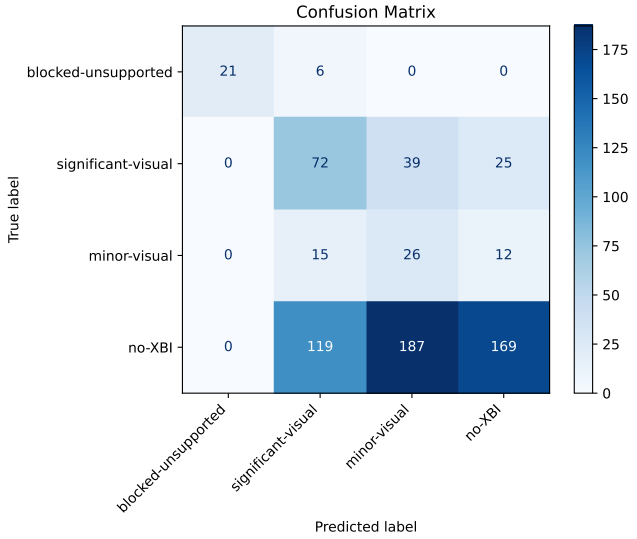


Figure 4: Confusion matrix of $\texttt{XBIDetective}_{\text{base}}$'s labelling of the impact score.

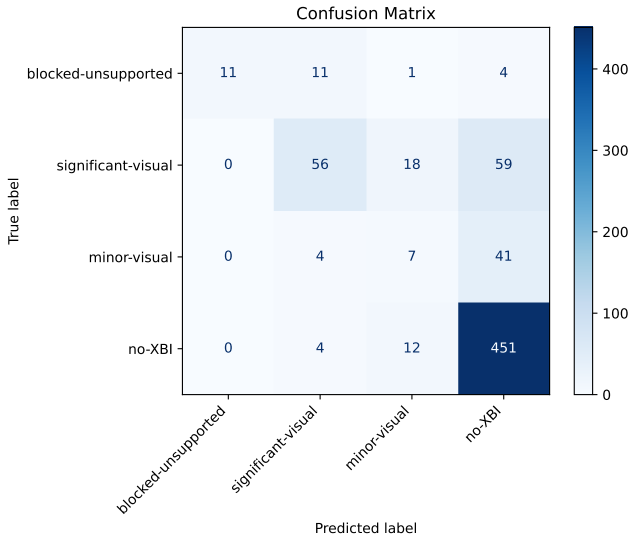


Figure 5: Confusion matrix of $\texttt{XBIDetective}_{\text{thinking}}$'s labelling of the impact score.

(24 instances), the dynamic elements themselves (15 instances), an image not rendering (13 instances), and the site failing to load (7 instances).

As seen in Figure 6 $\texttt{XBIDetective}_{\text{fine-tuned}}$ incorrectly identifies 106 screenshots as "no-XBI". These false negatives, similar to $\texttt{XBIDetective}_{\text{thinking}}$, involve missing XBIs such as the presence of pop-ups (31 instances), font discrepancies (13 instances), and images not rendering (8 instances). Notably, 81 of the false negatives identified by the fine-tuned XBIDetective were also identified by

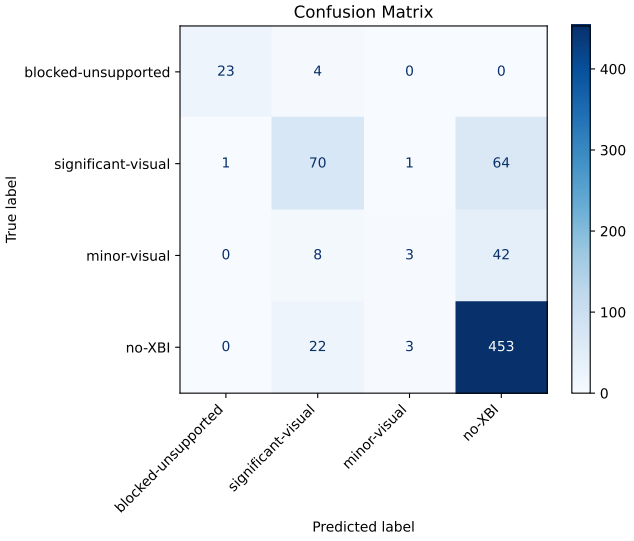


Figure 6: Confusion matrix of $\texttt{XBIDetective}_{\text{fine-tuned}}$'s labelling of the impact score.

XBIDetective version	**Accuracy**	**Precision**	**Recall**
$\texttt{XBIDetective}_{\text{base}}$	42%	57%	54%
$\texttt{XBIDetective}_{\text{thinking}}$	77%	69%	48%
$\texttt{XBIDetective}_{\text{fine-tuned}}$	79%	72%	59%
Without advertisement detection			
$\texttt{XBIDetective}_{\text{thinking}}$	68%	59%	42%
$\texttt{XBIDetective}_{\text{fine-tuned}}$	76%	64%	58%
Without dynamic element detection			
$\texttt{XBIDetective}_{\text{thinking}}$	73%	63%	45%
$\texttt{XBIDetective}_{\text{fine-tuned}}$	76%	65%	58%

Table 2: Experimental results on the performance of XBIDetective versions in assessing XBI impact score

the base version of XBIDetective, indicating potential ambiguity in those renderings.

Overall, $\texttt{XBIDetective}_{\text{thinking}}$ and $\texttt{XBIDetective}_{\text{fine-tuned}}$ perform well, indicating that they can reliably detect XBIs. Developers are mostly interested in the significant-visual and blocked-unsupported categories, as the impact of such XBIs is likely the largest. Hence if we combine the minor-visual and no-XBIs categories, we focus on the discrepancies that browser developers are most likely to prioritize. Under this perspective, the model's performance appears even stronger for real-world applications, achieving an accuracy of 85% for both versions of XBIDetective, since minor visual issues, such as a search bar with a smaller width than the page width, may not be prioritized for fixes. Nevertheless, it is important to consider the non-combined metrics, as they show how many minor issues are correctly flagged by XBIDetective, providing a more complete picture of its detection behaviour.

$\texttt{XBIDetective}_{\text{thinking}}$'s textual output identifying the XBI, correctly matched 92% of XBIs labelled in the ground truth. In these instances, $\texttt{XBIDetective}_{\text{thinking}}$ correctly located the XBI on each page. Among the incorrect classifications (9 instances), $\texttt{XBIDetective}_{\text{thinking}}$ occasionally hallucinates issues, including misidentifying spacing inconsistencies (5 instances), and incorrectly identifying an advertisement as a website element (1 instance). Further, XBIDetective also misclassified a dynamic element change as an XBI (1 instance). For the final 2 instances, the hallucinations made by XBIDetective are of a change in colour between the screenshots of the websites, and the presence of a sidebar in the website.

Explicitly directing the VLM to identify dynamic elements and ads increases the accuracy of XBIDetective in detecting XBIs. Without prompting $\texttt{XBIDetective}_{\text{thinking}}$ and $\texttt{XBIDetective}_{\text{fine-tuned}}$ to identify advertisements, their accuracy in identifying XBIs drops to 68%, and 76% respectively. Similarly, without prompting the XBIDetective to identify dynamic elements, the accuracy in detecting XBIs drops from 77% to 73% for $\texttt{XBIDetective}_{\text{thinking}}$ and from 79% to 76% for $\texttt{XBIDetective}_{\text{fine-tuned}}$ The drop in performance in identifying XBIs by both versions of XBIDetective suggests that prompting the model to identify advertisements and dynamic elements, then ignore them is an essential step within the pipeline. Without ad detection, both versions of XBIDetective generate significantly more false positives, incorrectly classifying no-XBI sections as significant-visual (56 instances vs. 5 for $\texttt{XBIDetective}_{\text{thinking}}$) and minor-visual (39 instances vs. 11 for $\texttt{XBIDetective}_{\text{thinking}}$). $\texttt{XBIDetective}_{\text{fine-tuned}}$'s performance does not drop as drastically as $\texttt{XBIDetective}_{\text{thinking}}$, but it is still significant enough to make the results from the pipeline less informative. Further, because dynamic elements can appear differently across browsers when a page loads, failing to identify them may lead the model to misinterpret them as XBIs, even though these variations are considered to be expected variations in a website.

> ***Takeaway:*** *Overall,* $\texttt{XBIDetective}_{\textit{fine-tuned}}$ *demonstrates better performance than* $\texttt{XBIDetective}_{\textit{thinking}}$ *in identifying XBIs.* $\texttt{XBIDetective}_{\textit{thinking}}$ *uses a thinking model that involves multiple reasoning steps for the VLM, making it more computationally expensive and slower to run. The cheaper, faster* $\texttt{XBIDetective}_{\textit{base}}$ *underperforms in identifying XBIs. However, after fine-tuning* $\texttt{XBIDetective}_{\textit{base}}$ *into* $\texttt{XBIDetective}_{\textit{fine-tuned}}$*, we can still use a non-thinking model which is optimized for the three specific stages to identify XBIs, allowing it to operate more efficiently in the long run.*

7 Lessons Learned

The ground truth dataset described in the previous section was carefully curated. For instance, we manually removed broken or incomplete screenshots. While this level of curation was necessary to evaluate XBIDetective, it would not be feasible in a realistic large-scale run on many websites. To better understand how XBIDetective performs under such conditions, we conducted a larger-scale run on 1,695 websites. From this run, XBIDetective identified 78 XBIs (55 significant-visual, 10 minor-visual, and 13

blocked-unsupported). Note that these are realistic results, as the normal assumption is that most websites do not contain XBIs. In this section, we discuss the lessons learned from that experiment.

7.1 Lesson 1: Capturing comparable screenshots across browsers is hard

We find that capturing consistent and comparable screenshots across different browsers presents a major challenge. The most important obstacles for capturing comparable screenshots were:

Obstacle 1: Rendering quirks in headless mode. When using Selenium with Google Chrome, the browser does not support full-page screenshots unless it is run in headless mode. However, headless mode is not ideal as it can introduce its own rendering quirks [10]. Despite this limitation, we use headless mode as a necessary compromise to ensure consistent sizes across screenshots.

Obstacle 2: Inherent stylistic differences between browsers. Browsers behave differently and may render websites with slight variations, e.g., because of differences in how scrollbars are handled.

Obstacle 3: Blocked websites. Some websites blocked `XBIDetective`, most likely due to anti-bot systems (such as Cloudflare). Especially Google Chrome is more susceptible to bot detection when driven by Selenium, and was frequently blocked by anti-bot systems.

The first two obstacles are difficult to overcome automatically. We found two effective ways to ignore blocked websites:

- **Preprocessing:** During the screenshot taking process, if a site displays keywords that suggest blocked access, we do not take the screenshot. Keywords include phrases such as "403 Forbidden" or "you have been blocked".
- **Post-inference filtering:** After the fine-tuned `XBIDetective` provides its output for XBI detection, we pass the images and the `XBIDetective` 's output to a secondary VLM. This model analyzes the text for any mention of a page not loading, and reviews the images to ensure they do not contain a message indicating they are blocked. Prompting the VLM allows for cases where blocked messages are in a different language or the original VLM output mentions being blocked to be filtered out, removing false positives from the report that is analyzed by developers.

These filtering steps reduce the number of false positives that browser developers need to inspect in large-scale runs. Importantly, filtering websites has little effect on the overall usefulness of the results, since large-scale analyses are still likely to uncover multiple instances of XBIs that point to the same underlying problem.

7.2 Lesson 2: `XBIDetective` can be used for regression testing as well

In the experiments we discuss in this paper, we focused on XBIs. Another interesting application of `XBIDetective` is using two versions of the same browser to do regression testing. Using `XBIDetective` in this setting actually removes a lot of the challenges we came across compared to the cross-browser setting. When using `XBIDetective` for regression testing for Firefox, there are fewer false positives and negatives. The screenshots are largely consistent in shape and size when captured from two different versions of the same browser. Additionally, the regression test encounters fewer blocking issues, allowing an extra 180 screenshot sets (998 for the regression test vs 818 for the cross browser test) to be compared.

7.3 Lesson 3: Unaddressed pop-ups, such as cookie consent dialogs, can lead to false positives and should be explicitly handled

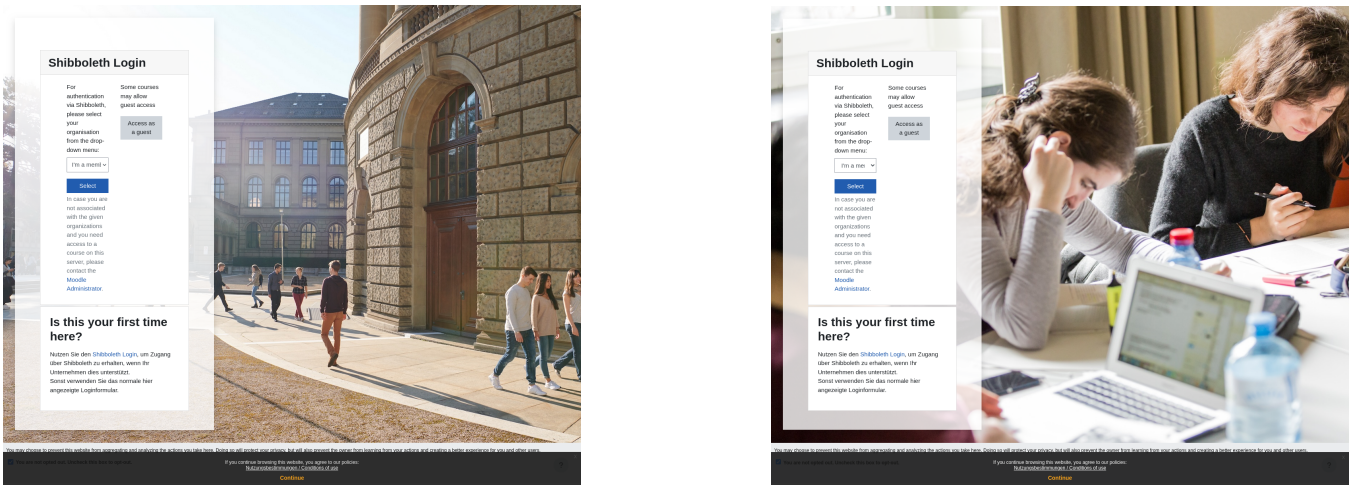
Pop-ups (or more precisely, modal dialogs), can vary in position, content, or behaviour across browser sessions. These pop-ups, such as cookie consent banners, subscription prompts, or advertisement overlays, can differ depending on timing, or whether the pop-up successfully loads. In some cases, a pop-up may appear in one rendering but not in another, leading `XBIDetective` to falsely interpret this variation as an XBI.

To address false positives caused by pop-ups, we use Selenium to close the most common types of pop-ups. Selenium is provided with filters specifying the pop-up types it can attempt to close. However, the filters used to close pop-ups are limited in scope and cannot account for all variations, particularly less common or dynamically introduced pop-ups. As a result, some pop-ups persist and introduce visual differences that do not represent true XBIs. We found that the most effective strategy is to isolate results from `XBIDetective` that contain information about pop-ups and place them into a separate table for review by developers. Thus, noise is reduced in the primary results and developers are able to focus on issues that are more likely to be genuine XBIs. We also preliminarily explored the use of web agents to automate the task of closing pop-ups. This approach appears promising, as the agents can identify and dismiss pop-ups without the need for predefined filters.

7.4 Lesson 4: Some false positives are very difficult to prevent

These false positives are seen in scenarios involving dynamic elements such as changing pictures, backgrounds, or placeholder text. Because `XBIDetective` processes a single static instance of the page, it often lacks sufficient context to recognize that these elements are dynamic. Although dynamic elements can create discrepancies between website screenshots, they are not considered XBIs, as such variations are expected on a website. For example, as seen in Figure 7, a login page with a different background image across visits might be flagged as an XBI, even though the variation is intentional and not a true inconsistency. The same issue occurs with placeholder text or images that change on each page reload. While one potential solution is to reload the site multiple times to capture variability, this approach is computationally expensive and time-consuming.

In contrast, we noticed no instances where an advertisement is marked as a XBI. Ads, despite also being dynamic, often include indicators (such as labels or structural cues) that help `XBIDetective` correctly identify them as non-critical changes, while dynamic elements may not have an indication of such.



(a) Firefox screenshot

(b) Chrome screenshot

Figure 7: Example of two screenshots taken of https://moodle-app2.let.ethz.ch with a dynamically changing background in Firefox and Chrome

8 Threats to Validity

Construct validity: Our approach intentionally ignores advertisements or dynamic elements to reduce false positives in XBI identification. While this reduces noise from expected content variability, it may also omit genuine XBIs in these elements. As a result, our approach currently misses XBIs that occur in dynamic elements or advertisements. This choice was intentional to reduce false positives in the generated reports.

Internal validity: A threat to the validity of the study stems from the use of Selenium to take the screenshots of the websites. Because an automation tool such as Selenium is used, some websites block the tool from taking screenshots, thus decreasing the number of websites analyzed. Future studies should consider looking into other web testing frameworks to take the screenshots, or settings for Selenium that would decrease the number of blocked screenshots.

Changing VLM versions are also a potential threat to the validity of this study. New models are released at a rapid pace, making it challenging to keep evaluations up to date. As such, newer models may perform either better or worse than the ones used with `XBIDetective` in this study. We designed `XBIDetective` to be easily adaptable to different model versions, but future work should evaluate its performance with newer VLM releases.

A final threat to the internal validity of this study lies in the creation of the ground truth for evaluating `XBIDetective`, which was manually verified by the first author. This process may introduce bias in labelling the impact of each bug report, a task that is already inherently ambiguous. To mitigate this risk, we verified a sample of the training data for fine-tuning with Mozilla developers.

External validity: A potential threat to the validity of this study is that we only use bugs reported on Bugzilla or WebCompat. This website is managed by Mozilla employees and volunteers, thus, the web compatibility bugs may be biased toward issues related to Mozilla's Firefox browser. Consequently, the dataset may not fully reflect the diversity of web compatibility bugs across all browsers.

Further, Selenium only supports a few browsers for taking screenshots, thus we only analyze Google Chrome and Mozilla Firefox. Though these browsers make up the majority of popular web browser engines, the results of this study may not generalize to compatibility bugs found exclusively on other bug reporting platforms or browsers.

9 Conclusion

We introduce `XBIDetective`, a tool to leverage vision language models to detect cross-browser inconsistencies by comparing website screenshots. We evaluate the effectiveness of both off-the-shelf (base and thinking) and fine-tuned VLMs with `XBIDetective` in identifying XBIs across browser renderings. We find that both the base and fine-tuned versions of `XBIDetective` perform well at this task, achieving an accuracy of 77% and 79%, respectively.

A cornerstone of our approach is to explicitly direct the VLM to identify advertisements and dynamic elements before analyzing for XBIs. To evaluate the foundation for this cornerstone, we show that VLMS are effective in identifying these elements, though the fine-tuned version shows a drop in accuracy for dynamic element detection. We also show that a fine-tuned base VLM can perform better than a thinking VLM at a fraction of the cost.

In a large-scale evaluation of 1,695 websites with no ground truth established, we found that reducing false positives, from ambiguous cases involving dynamic elements and pop-ups is essential for generating usable reports for developers. Furthermore, ensuring accurate XBI identification requires minimizing differences in how screenshots are captured across browsers.

Overall, this work demonstrates that `XBIDetective`, especially when fine-tuned, is a approach for detecting of cross-browser inconsistencies in web development, helping to highlight discrepancies in website renderings that may indicate underlying bugs.

References

[1] I. Althomali, G. M. Kapfhammer, and P. McMinn, "Automatic visual verification of layout failures in responsively designed web pages," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 183–193.

[2] L. Ardito, A. Bottino, R. Coppola, F. Lamberti, F. Manigrasso, L. Morra, and M. Torchiano, "Feature matching-based approaches to improve the robustness of android visual gui testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–32, 2021.

[3] Z. Cao, G. Hidalgo, T. Simon, S.-E. Wei, and Y. Sheikh, "Openpose: Realtime multi-person 2d pose estimation using part affinity fields," *IEEE transactions on pattern analysis and machine intelligence*, vol. 43, no. 1, pp. 172–186, 2019.

[4] S. R. Choudhary, M. R. Prasad, and A. Orso, "Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 171–180.

[5] S. R. Choudhary, H. Versee, and A. Orso, "Webdiff: Automated identification of cross-browser issues in web applications," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[6] V. Dallmeier, M. Burger, T. Orth, and A. Zeller, "Webmate: Generating test cases for web 2.0," in *Software Quality. Increasing Value in Software and Systems Development: 5th International Conference, SWQD 2013, Vienna, Austria, January 15-17, 2013. Proceedings 5*. Springer, 2013, pp. 55–69.

[7] B. F. Demissie, Y. N. Tun, L. K. Shar, and M. Ceccato, "Vlm-fuzz: Vision language model assisted recursive depth-first search exploration for effective ui testing of android apps," 2025. [Online]. Available: https://arxiv.org/abs/2504.11675

[8] S. Feng, M. Xie, and C. Chen, "Efficiency matters: Speeding up automated testing with gui rendering inference," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 906–918.

[9] B. Grewal, J. Graham, J. Muizelaar, J. H. Odvarko, S. Mujahid, M. Castelluccio, and C.-P. Bezemer, "XBIDetective: Leveraging vision language models for identifying cross-browser visual inconsistencies - artifact," 2025. [Online]. Available: https://doi.org/10.6084/m9.figshare.30223654

[10] J. G. V. Jr. (2025, Jan.) Understanding headless vs. headed modes in playwright: A guide for qa automation engineers / sdets. Accessed: 2025-09-24. [Online]. Available: https://medium.com/@JohnnyV_5G/understanding-headless-vs-headed-modes-in-playwright-a-guide-for-qa-automation-engineers-sdets-153452e65cbf

[11] B. Ju, J. Yang, T. Yu, T. Abdullayev, Y. Wu, D. Wang, and Y. Zhao, "A study of using multimodal llms for non-crash functional bug detection in android apps," in *2024 31st Asia-Pacific Software Engineering Conference (APSEC)*, 2024, pp. 61–70.

[12] X. Liang, J. Qi, Y. Gao, C. Peng, and P. Yang, "Ag3: Automated game gui text glitch detection based on computer vision," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1879–1890. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/3611643.3613867

[13] R. Liu, X. Teoh, Y. Lin, G. Chen, R. Ren, D. Poshyvanyk, and J. S. Dong, "Guipilot: A consistency-based mobile gui testing approach for detecting application-specific bugs," *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3728909

[14] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, "Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[15] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Owl eyes: spotting ui display issues via visual understanding," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 398–409. [Online]. Available: https://doi.org/10.1145/3324884.3416547

[16] ——, "Nighthawk: Fully automated localizing ui display issues via visual understanding," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 403–418, 2023.

[17] Z. Liu, C. Li, C. Chen, J. Wang, M. Chen, B. Wu, Y. Wang, J. Hu, and Q. Wang, "Seeing is believing: Vision-driven non-crash functional bug detection for mobile apps," 2024. [Online]. Available: https://arxiv.org/abs/2407.03037

[18] Z. Long, G. Wu, Y. Zhang, W. Chen, and J. Wei, "Poster: Repair cross browser layout issues by combining learning and search-based technique," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 470–473.

[19] W. Lu, A. Senchenko, A. Hindle, and C.-P. Bezemer, "Automated bug frame retrieval from gameplay videos using vision-language models," 2025. [Online]. Available: https://arxiv.org/abs/2508.04895

[20] F. Macklon and C.-P. Bezemer, "Exploring the capabilities of vision-language models to detect visual bugs in html5 <canvas> applications," 2025. [Online]. Available: https://arxiv.org/abs/2501.09236

[21] F. Macklon, M. R. Taesiri, M. Viggiato, S. Antoszko, N. Romanova, D. Paas, and C.-P. Bezemer, "Automatically detecting visual bugs in html5 canvas games," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/3551349.3556913

[22] S. Mahajan, A. Alameer, P. McMinn, and W. G. J. Halfond, "Xfix: an automated tool for the repair of layout cross browser issues," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 368–371. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/3092703.3098223

[23] S. Mahajan and W. G. J. Halfond, "Detection and localization of html presentation failures using computer vision-based techniques," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–10.

[24] ——, "Websee: A tool for debugging html presentation failures," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–8.

[25] S. Mahajan, B. Li, and W. G. Halfond, "Root cause analysis for html presentation failures using search-based techniques," in *Proceedings of the 7th International Workshop on Search-Based Software Testing*, 2014, pp. 15–18.

[26] MDN contributors. (2025, Apr.) Introduction to cross-browser testing. Accessed: 2025-08-11. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Testing/Introduction

[27] C. Paduraru, M. Paduraru, and A. Stefanescu, "Automated game testing using computer vision methods," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, 2021, pp. 65–72.

[28] S. Roy Choudhary, M. R. Prasad, and A. Orso, "X-pert: a web application testing tool for cross-browser inconsistency detection," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 417–420.

[29] T. Saar, M. Dumas, M. Kaljuve, and N. Semenenko, "Browserbite: cross-browser testing via image processing," *Software: Practice and Experience*, vol. 46, no. 11, pp. 1459–1477, 2016.

[30] L. N. Sabaren, M. A. Mascheroni, C. L. Greiner, and E. Irrazábal, "A systematic literature review in cross-browser testing," *Journal of Computer Science & Technology*, vol. 18, 2018.

[31] N. Semenenko, M. Dumas, and T. Saar, "Browserbite: Accurate cross-browser testing via machine learning over image features," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 528–531.

[32] S. Song, J. Hur, S. Kim, P. Rogers, and B. Lee, "R2z2: Detecting rendering regressions in web browsers through differential fuzz testing," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1818–1829.

[33] S. Song and B. Lee, "Metamong: Detecting render-update bugs in web browsers through fuzzing," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1075–1087.

[34] A. Stocco, R. Yandrapally, and A. Mesbah, "Visual web test repair," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 503–514. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/3236024.3236063

[35] M. R. Taesiri and C.-P. Bezemer, "Videogamebunny: Towards vision assistants for video games," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2025.

[36] M. R. Taesiri, T. Feng, A. Nguyen, and C.-P. Bezemer, "Glitchbench: Can large multimodal models detect video game glitches?" in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.

[37] M. R. Taesiri, A. Ghildyal, S. Zadtootaghaj, N. Barman, and C.-P. Bezemer, "Videogameqa-bench: Evaluating vision-language models for video game quality assurance," https://arxiv.org/abs/2505.15952, 2025.

[38] M. R. Taesiri, F. Macklon, Y. Wang, H. Shen, and C.-P. Bezemer, "Large language models are pretty good zero-shot video game bug detectors," 2022. [Online]. Available: https://arxiv.org/abs/2210.02506

[39] H. Tanno, Y. Adachi, Y. Yoshimura, K. Natsukawa, and H. Iwasaki, "Region-based detection of essential differences in image-based visual regression testing," *Journal of Information Processing*, vol. 28, pp. 268–278, 2020.

[40] J. Tuovenen, M. Oussalah, and P. Kostakos, "Mauto: Automatic mobile game testing tool using image-matching based approach," *The Computer Games Journal*, vol. 8, no. 3, pp. 215–239, 2019.

[41] T. A. Walsh, G. M. Kapfhammer, and P. McMinn, "Redecheck: an automatic layout failure checking tool for responsively designed web pages," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 360–363. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/3092703.3098221

[42] S. Wang, S. Wang, Y. Fan, X. Li, and Y. Liu, "Leveraging large vision-language model for better automatic web gui testing," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2024, pp. 125–137.

[43] W. M. Watanabe, G. L. Amêndola, and F. C. Paes, "Layout cross-platform and cross-browser incompatibilities detection using classification of dom elements," *ACM Trans. Web*, vol. 13, no. 2, Mar. 2019. [Online]. Available: https://doi.org/10.1145/3316808

[44] W. M. Watanabe, D. A. dos Santos, and C. de Oliveira, "Layout cross-browser failure classification for mobile responsive design web applications: Combining classification models using feature selection," *ACM Transactions on the Web*, vol. 17, no. 4, pp. 1–34, 2023.

[45] M. Xie, S. Feng, Z. Xing, J. Chen, and C. Chen, "Uied: a hybrid tool for gui element detection," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1655–1659.

[46] S. Xu, C. Zhou, Z. Gu, G. Wu, W. Chen, and J. Wei, "X-diag: Automated debugging cross-browser issues in web applications," in *2018 IEEE International Conference on Web Services (ICWS)*. IEEE, 2018, pp. 66–73.

[47] B. Yang, Z. Xing, X. Xia, C. Chen, D. Ye, and S. Li, "Uis-hunter: Detecting ui design smells in android apps," in *2021 IEEE/ACM 43rd international conference on software engineering: companion proceedings (ICSE-companion)*. IEEE, 2021, pp. 89–92.

[48] C. Zhou, Q. Zhang, B. Qian, and Y. Jiang, "Janus: Detecting rendering bugs in web browsers via visual delta consistency," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 153–164.