EAGL: Evolution-Aware Generation of Robust Locators

HILAL TAHA, SnT, University of Luxembourg, Luxembourg
JEONGJU SOHN, Kyungpook National University, Republic of Korea
MIKE PAPADAKIS, SnT, University of Luxembourg, Luxembourg

Modern web applications evolve continuously, often causing test breakages when test scripts fail to adapt to changes in the Document Object Model (DOM). Such breakages frequently arise from web locators, which identify the DOM elements that tests interact with. Since locators rely on specific elements and attributes to navigate the DOM, they are inherently vulnerable to changes. To address this, we propose EAGL, a technique for generating robust locators by leveraging the stability of DOM positions—resolving to either element or attribute nodes—predicted from historical DOM changes. EAGL synthesises locators by dynamically selecting stable and unique positions adequate to compose a locator for an element. We constructed a dataset of DOM position changes across 181 real-world websites and trained a stability prediction model that identifies persistent positions. Experimental results show that EAGL achieves strong stability prediction performance, with high classification metrics (precision 0.87, recall 0.85, F1 0.80, ROC-AUC 0.71) and robust ranking capability (MAP 0.83), reflecting its ability to prioritise stable DOM positions. Leveraging these models, EAGL generates robust and precise locators, outperforming Selenium and ROBULA+ while maintaining high efficiency (average locator generation time under 0.25 s). Cross-project evaluations confirm that EAGL generalises well across different websites, demonstrating its practical usefulness.

CCS Concepts: • **Software and its engineering** → **Maintaining software**; **Software evolution**.

ACM Reference Format:
Hilal Taha, Jeongju Sohn, and Mike Papadakis. 2026. EAGL: Evolution-Aware Generation of Robust Locators. 1, 1 (September 2026), 37 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The fast evolution of the web environment results in modern web applications undergoing numerous changes over time, with the frequency of these changes typically happening daily [6]. While advances in automated web application testing have somewhat eased the burden of manual maintenance on developers of these changes [21, 27], the challenge of maintaining test suites to keep pace with the rapid and continuous evolution of web applications remains [9, 15, 16, 22]. Web application tests can break for various reasons; a broken locator for a web element in the Document Object Model (DOM) is a frequently observed reason [13, 14].

Locators allow DOM-based test scripts to automatically identify the elements to interact with, removing the need for human interruption during testing. To navigate DOMs, locators often depend on attributes or explicit paths (i.e., XPath) of elements. As these attributes and paths change when the web application evolves, the locators based on them are intrinsically volatile [26]. Indeed, previous studies have shown that broken web element locators are a significant contributor to test breakages in web GUI tests, accounting for as much as 73% of such failures [13, 14].

Authors' addresses: Hilal Taha, hilal.taha@uni.lu, SnT, University of Luxembourg, Luxembourg; Jeongju Sohn, jeongju.sohn@knu.ac.kr, Kyungpook National University, Daegu, Republic of Korea; Mike Papadakis, michail.papadakis@uni.lu, SnT, University of Luxembourg, Luxembourg.

XXXX-XXXX/2026/9-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

In recent years, there has been a growing body of research on addressing broken locators throughout the evolution (i.e., changes) of web applications [9, 12, 14, 15, 17, 20, 23–26, 33, 35]. This research can be roughly divided into two: 1) those focused on the generation of robust locators to prevent breakages [23–26, 33, 35] and 2) those repairing broken locators [9, 12, 14, 15, 17, 20].

Existing studies on robust locator generation aim to reduce breakages by leveraging attributes or patterns that are statistically less likely to change across web page versions, often guided by their past success in identifying target elements [7, 24, 26, 32]. Leotta et al. proposed SIDEREAL, a learning-based approach that constructs robust XPath locators by adaptively weighting combinations of attribute values–primarily from the target element and its ancestors–based on their historical success (i.e., frequency) in multiple versions of a target web application [23].

In contrast, locator repair approaches attempt to recover broken locators by identifying elements that resemble the original target, using heuristics such as attribute similarity, DOM structure, or tree-edit distance [9, 12, 14, 20]. Despite differences in method, both classes of techniques–generation and repair–share the underlying assumption that certain attributes or structural patterns are more robust and can serve as reliable cues over time.

In this research, we propose EAGL (**E**volution-**A**ware **G**eneration of robust **L**ocators), a fundamentally different approach to robust locator generation. Instead of relying on heuristics or historically successful patterns, EAGL focuses on identifying *DOM positions* (i.e., locations within a DOM tree) that persist structurally across versions. This position-centric view shifts the focus from selecting robust attributes or elements to identifying stable structural anchors that are inherently less susceptible to change. XPath locators refer to structural positions in the DOM tree. Given the brittle nature of XPath locators, which often break due to structural shifts, EAGL uses these persistent positions as anchors for constructing robust locators.

An alternative strategy could involve identifying semantically consistent nodes–attributes or elements that may change or move within a DOM tree but retain the same role across versions. However, such semantic tracking remains a costly and unresolved challenge [23, 24, 26, 33, 35, 46]. EAGL avoids this by focusing purely on structural persistence, offering a scalable and principled basis for robust locator construction.

Studies on traditional software evolution in the context of maintenance and debugging show that certain structural or contextual properties are strong indicators of component instability, i.e., a key factor in test fragility [19, 29, 42, 45, 47]. Building on this insight, EAGL predicts the stability of DOM positions using lightweight structural features derived from the node each position resolves to, such as whether it is an element or attribute, its sibling structure, and tree depth. Specifically, EAGL trains a Random Forest regressor on six features—one for node type, three for neighbouring context, and two for location in the tree—extracted by analysing version histories of 181 actively maintained websites. The model predicts the likelihood that a DOM position will remain unchanged across versions, allowing EAGL to prioritise stable positions when constructing robust XPath locators.

For a locator to be effective, it must be both *robust* to web evolution and sufficiently *precise* to identify the intended element. A locator that is robust but overly general—returning multiple elements instead of uniquely identifying the target—is not useful in practice. To ensure that a locator is not only robust but also precise, EAGL incorporates a notion of uniqueness, which quantifies how rare the node at a given DOM position is within its local context, serving as a proxy for the precision of the resulting locator by estimating how much that position helps narrow the search space.

We compared EAGL with Selenium, the state-of-the-practice in locator generation, which is lightweight yet fairly effective, and ROBULA+, a widely studied technique for robust locator generation. Our empirical evaluation shows that EAGL can generate robust and effective locators

that can withstand changes more frequently than Selenium and ROBULA+ while being fairly precise. The technical contributions of this paper can be summarised as follows.

- We propose EAGL, a novel technique for generating robust locators by analysing the evolution of web applications. EAGL identifies stable DOM positions—specific locations in the DOM tree that persist across versions—using six structural DOM features, and uses them as anchors to construct robust locators.
- We build a publicly available dataset that captures the historical structural changes of 181 websites, along with six stability prediction features computed for each DOM position, that is, a tree location where an element or attribute node appears.[1]
- We present in-depth experimental analyses demonstrating the effectiveness of EAGL over existing locator generation methods, including both the one widely used in practice and the one extensively studied.

The remainder of the paper is organised as follows. Section 2 explains how locators are used in web testing, and Section 3 describes the methodology of EAGL. Section 4 introduces our research questions and the setup for the experiments, and Section 5 analyses the results of these experiments; Section 6 further discusses additional points of the experiments. Section 7 examines potential threats to validity, and Section 8 presents the related work. Lastly, Section 9 concludes our research.

2 LOCATORS IN WEB APPLICATION TESTING

Web tests often involve interacting with elements to simulate the practical usage of web applications or systems. Locators, the means of identifying elements for interaction, thereby play a pivotal role in the seamless execution of tests, enabling tests to validate button states, input validity, link destinations, and more: if a locator fails to pinpoint the intended element for a web test, the test fails [36, 38]. Locators can be broadly categorised into stand-alone strategies, which rely on a single attribute or element property, and composite strategies, such as XPath expressions or CSS selectors [5, 27, 39]. For simplicity, we refer to both element nodes and the attribute nodes as *nodes* hereafter. Reliable web testing requires constructing locators that are resilient to DOM evolution—that is, robust against structural changes in the web application's DOM over time.

2.1 XPath expressions

XPath (XML Path) is a query language used to select nodes within an XML document. In web development, it is commonly employed to locate elements in an HTML document—represented as a Document Object Model (DOM)—which has a tree-like structure similar to XML documents [41]. XPath enables element selection with various criteria, such as tag names (element types), attributes, and hierarchical relationships between HTML components, offering testers flexibility in locating elements. Based on their starting point, XPaths can be categorised into two: *absolute* and *relative*.

(1) Absolute XPath: a complete path from the root element (<html>) of the DOM to the location of a target element, specifying every node along the way: e.g., `/html/body/div[2]//ul/li[3]/a`
(2) Relative XPath: a path that starts from a selected node in the DOM and navigates to the location of a target element using attributes or relationships, instead of a full hierarchical path: e.g., `//a[@class='login']` or `//div[@id="form1"]/ul[2]/li[4]`

Given the number of nodes and the structural complexity of a DOM, there can be an immense number of possible relative XPaths; any node—element or attribute—may serve as a starting point to construct a path to a target element. Consequently, some studies approach XPath selection as a graph exploration problem [23]. An XPath query may return zero or more elements. In web

[1] https://zenodo.org/records/10974174

application testing, the goal is typically to construct an XPath query/locator that consistently returns the intended element for interaction. Although an XPath traverses multiple nodes—each potentially affected by DOM changes—, its fragility depends not merely on path length, but more critically on the stability of the location within a DOM tree to which those nodes are bound. The key challenge is thereby selecting structurally persistent components from which to build reliable XPath locators [26].

2.2 Locator breakage

We explain the locator breakage problem caused by changes (i.e., evolution) in web applications through an example. Figure 1 shows two consecutive versions of a webpage, where a change occurs in a form element: its `class` attribute is modified from *form-1* to *form-2* in *Version 2*. In *Version 1*, two XPath locators correctly identify the same text input element named "*username*": *Locator1* (`//*form[@class="form-1"]/input[1]`) uses the `class` attribute, and *Locator2* (`//*form[@id="myForm"]/input[1]`) uses the `id` attribute. However, in *Version 2*, *Locator1* breaks due to the changed `class` value, while *Locator2* continues to function as the `id` remains unchanged. This behavioural difference between *Locator1* and *Locator2* highlights how some attributes are more volatile than others across DOM changes and are, therefore, not well-suited as reliable anchors for navigating to a target element. Consequently, to generate robust locators, it is important to select structural locations within the DOM tree that are likely to remain stable over time, i.e., reliably resolve to the intended node. In this paper, we explore how EAGL predicts the stability of such anchors—DOM positions—based on the structural properties of the nodes currently residing at those positions.

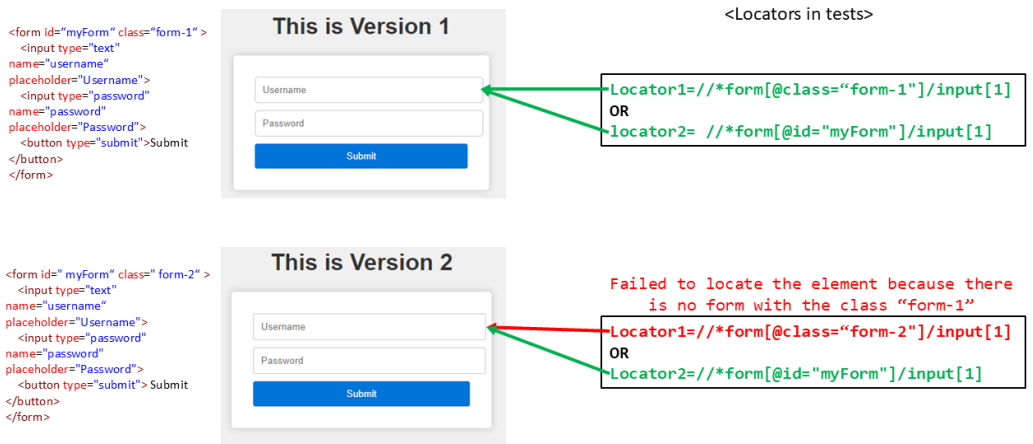

Fig. 1. Locators breaking in tests for web applications

3 RESEARCH METHODOLOGY OF EAGL

In this section, we outline the architecture of EAGL and describe in detail the steps to generating a robust locator. Figure 2 presents an overview of EAGL in which it goes through two main phases: 1) *stability prediction* and 2) *locator generation*.

3.1 Terminology

EAGL aims to construct robust XPath locators by identifying stable anchors—specific locations in the DOM tree that persist across versions and can reliably guide navigation to a target element. We refer to these anchors as **DOM positions**, emphasising that our focus is on *structural location stability* rather than the identity of any specific node.

Although a DOM position may resolve to an element or attribute node in any given version, we *avoid* calling them "stable nodes". A node may remain semantically unchanged across versions, but its *location* within the DOM tree can shift due to structural changes—making it unreliable as an anchor. By targeting **DOM positions** that are structurally stable over time, EAGL ensures that the constructed XPath locators are anchored at locations more resilient to DOM evolution (i.e., more stable). We argue that although an XPath is composed of a sequence of node steps, its robustness depends on the stability of the DOM positions these nodes occupy across versions. As a result, EAGL first identifies such stable DOM positions for the generation of robust locators.

3.2 Stability Prediction

We define the stability of a DOM position as its likelihood of remaining unchanged in the future. While prior work, such as SIDEREAL [23][2], focuses on adaptively selecting attribute combinations based on their historical success in previously generated locators, this adaptation is limited to attribute-level statistics. In contrast, EAGL learns general structural stability from past DOM changes, enabling a broader approach to stability estimation using structural features extracted from the nodes located at each DOM position. As shown in Figure 2, EAGL leverages website version histories to extract these features and identify DOM changes at each DOM position.

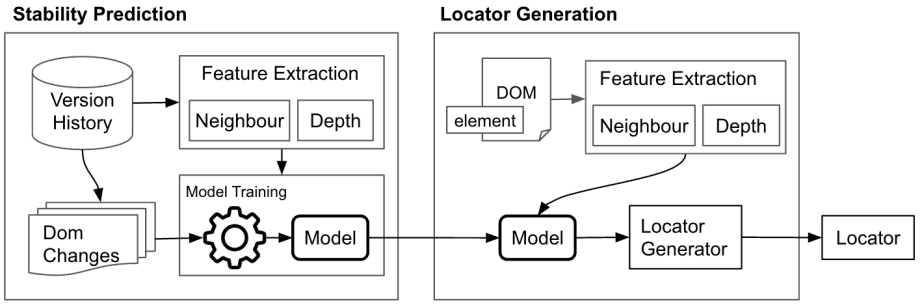

Fig. 2. Overview of EAGL. EAGL involves two main phases: stability prediction and locator generation. "element" in the locator generation module denotes a target element to locate.

[2]We were unable to find a functional implementation of SIDEREAL supplied by the authors. SIDEREAL also requires pre-existing tests to calculate adaptive weights (the calculation utilises only the nodes that have tests previously written for them), which are often absent or challenging to retrieve for many of the studied websites in this study.

3.2.1 Stability Features. EAGL leverages three categories of location stability features: two *location-based*, one *node type-based*, and three *neighbour-based*, for a total of six. Figure 3 illustrates how these features are computed. We define these structural features based on *the node*—either attribute or element— currently *at each DOM position*.

Neighbour-based. There are three types of neighbour nodes in the DOM: parent, child, and sibling nodes.

- **Parent nodes**: a parent node is a node that includes the given node as one of its subnodes or properties. For element nodes, the parent is another element node that includes the given node as one of its sub nodes in the child node list (i.e., `childNodes`). For attribute nodes, the parent is the element to which the attribute belongs.
- **Child nodes**: a child node is a node that directly descends from the given node, forming the next level of the hierarchical structure. For element nodes, child nodes are the direct descendants in the `childNodes` list, which can include other element nodes, constituting the next level of the hierarchy. Attribute nodes do not have child nodes, as they are leaf nodes in the DOM.
- **Sibling nodes**: sibling nodes are nodes at the same hierarchical level as the given node, sharing the same parent. For element nodes, siblings are other nodes in the same `childNodes` list. For attribute nodes, siblings are other attributes of the same element grouped within the element's attribute collection.

From these neighbouring relationships, we define three features for each node at each DOM position:

- **the number of siblings** counts the number of sibling nodes. In Figure 3, the rightmost `div` node in the tree, outlined in green, has three siblings (`input`, `div`, and `textarea` nodes), and thereby the value is three. The `cols` node in the tree, outlined with orange, has one sibling node (`rows` attributes), and thus, the value is one.
- **the number of children** applies only to element nodes to capture their hierarchical role. In Figure 3, `div` node has `textarea` and `a` nodes as children, and thus, the feature value is two. For attribute nodes, this value will simply be zero; for `cols` node, highlighted in orange, the value is zero, as it is an attribute node.
- **the position of the node** indicates its relative position or index among its siblings, which ranges from 1 to the number of siblings + 1. It is computed as $\frac{\text{the position of a node}}{\text{the number of siblings+1}}$. In Figure 3, the `div` node, highlighted in green, appears fourth among its siblings, and thus: $4/(3+1) = 1$. The `cols` node, the attribute of this `div` node, is positioned second, ahead of its only sibling—a `rows` attribute node—: $2/(1+1) = 1$.

Location-based. In addition to the above three neighbour-based features, we define two features based on the location of a node within a DOM tree: *Length* and *Subtree Height*.

- **Length** reflects the distance to a node from the root node in the DOM. It is calculated by counting the number of parent nodes along the path from the target node to the root node `<HTML>`. The counting is then normalised by dividing it by the longest XPath length in the DOM (i.e., the maximum number of parent nodes any node in the DOM can have, plus one). In Figure 3, the `div` node (outlined in green) has four parent nodes in its path to the root (i.e., `<form>`, `<section>`, `<body>`, and `<HTML>`), located at level four in the DOM hierarchy. In this DOM, the longest XPath length is seven, corresponding to the length of the absolute XPath of the cols node (outlined in orange): `/HTML/body/section/form/div/textarea/@cols`.

Thus, the *length* value for the `div` node is calculated as $\frac{4+1}{7} = \frac{5}{7}$. For the `cols` node, which is at the longest depth in the DOM, the *Length* value is 1 ($= \frac{7}{7}$).

- **Subtree Height**, simply as *Height*, represents the impact of changes on a given node. It is calculated as the height of the subtree rooted at the current node plus one, which corresponds to the length of the longest XPath originating from the node. The value is then normalised by dividing it by the length of the longest XPath in the entire DOM. In Figure 3, the subtree rooted at the *div* node (outlined in green) has a height of 2. Adding one for the root node, the feature value is calculated as $\frac{2+1}{7}$. For the *cols* node (outlined in orange), with no children being an attribute, the height is 0, and the value is $\frac{0+1}{7}$.

Node type. The node type feature is a binary value, where 0 represents an element node and 1 represents an attribute node. This feature differentiates the cases where element nodes appear and the cases where attribute nodes appear at the DOM positions, when composing an XPath expression, as they may exhibit different trends or patterns of stability. In Figure 3, the *div* node (outlined in green) has zero as the feature value, while the *cols* node (outlined in orange) has 1.

Together, these six features—capturing the type of node, its structural location and potential influence, and its surrounding context—provide a lightweight yet expressive representation of the structural role of a DOM position, enabling the model to infer its potential stability across versions.

DOM changes are tracked by labelling each DOM position as changed or unchanged between consecutive versions (Section 4.3.2). These labels, along with six structural features, compose the dataset used to train our stability prediction model (Figure 2).

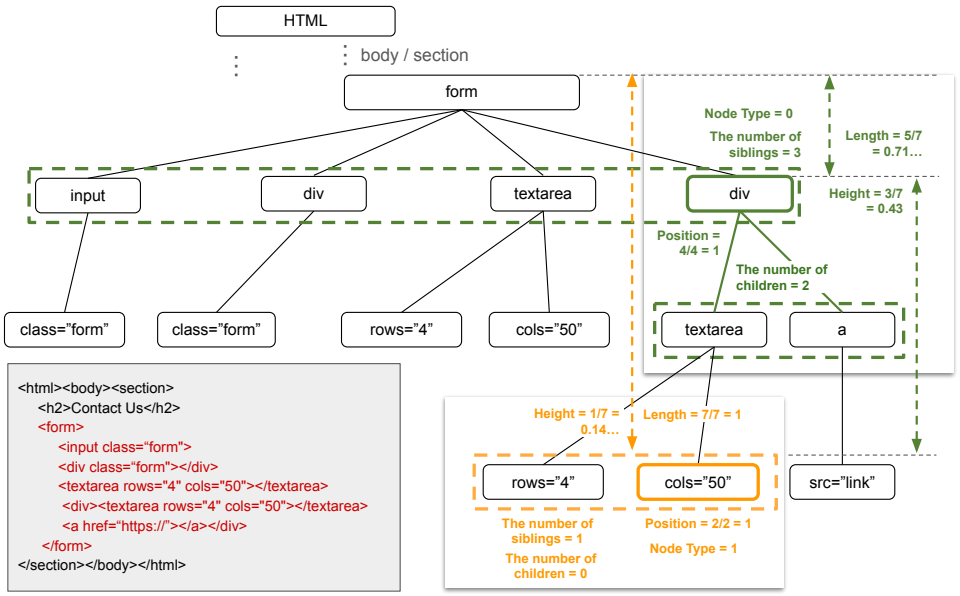

Fig. 3. Feature Computation. A rectangle in the top-right corner illustrates the computation of six prediction features for a location where an element *div* resides, while a rectangle in the bottom-right corner illustrates the computation of six prediction features for a location where an attribute node *cols* resides.

3.2.2 Prediction Model Training. By leveraging the six previously defined features, we train a model to predict the likelihood of a DOM position being changed in future. Previous studies on

general software systems have shown that frequently changed code elements are more prone to bugs [19, 29, 42, 45, 47]. Since this tendency is primarily driven by developer activity, a similar pattern may emerge in the evolution of web applications and locator breakages: *locators anchored at DOM positions with a higher likelihood of change are more likely to break in the future.* Consequently, our approach, EAGL, uses the trained model to forecast changes of DOM positions, as shown in Figure 2 (Model Training Module), computing change likelihood values (C) of individual positions. We refer to the complement of these likelihood values ($1 - C$) as *stability values*, with higher values indicating more stable DOM positions. Section 4.4 describes the model training process in detail. These stability values of DOM positions serve as key input to the next phase, i.e., *locator generation*.

3.3 Locator Generation

3.3.1 DOM Position Adequacy. After training a stability prediction model, EAGL constructs an effective locator, specifically an XPath locator, for a target element; hereafter, we call an XPath locator simply a locator. For locators to be effective across the continuous web evolution, they should be both *robust* and *precise*. To identify adequate DOM positions to anchor on for constructing robust locators, we define a new selection criterion called *DOM Position Adequacy*, or simply, *Position Adequacy*. This adequacy criterion extends beyond stability predictions from the trained model by additionally considering the precision of the resulting XPath locator when anchored at a specific position in the DOM tree. Equation (3) formalises *Position Adequacy* by combining the stability of a DOM position (S)—how consistently a DOM position has been preserved over time— with the uniqueness (U), which approximates how precise a locator can be when it relies on the node occupying that position.

DOM Position Stability (S). Equation (1) defines the computation of DOM position stability— referred to simply as Position Stability— for an arbitrary DOM position occupied by a node (*nd*) in an input *DOM*. EAGL estimates this stability using the pre-trained model M described in Section 3.2. Although the model M takes as input the structural features of node *nd*, its prediction represents the likelihood that the DOM position of *nd* remains stable over time, with values ranging from 0 to 1. Hence in this equation, the node serves as a concrete proxy—i.e., a reference point— for the underlying position within the DOM tree.

The function *ChangeLikelihood* involves computing features from the current DOM (*DOM*) and then estimating the likelihood of change using model M. Since our goal is to estimate the stability of the DOM position corresponding to a node *nd*, we define the stability as the complement (i.e., one minus) of the predicted change likelihood. For instance, if the model predicts the likelihood of the DOM position of the `form` node being changed as ≈ 0.335 (in Phase 1 of Figure 4), then its estimated stability will be: $S(DOM, M, \texttt{form}) = (1 - 0.335) \approx 0.665$.

$$S(DOM, M, nd) = (1 - ChangeLikelihood(DOM, M, nd)) \tag{1}$$

DOM Position Uniqueness (U). We approximate the precision of a resulting locator using the uniqueness of the nodes currently residing at the DOM positions it anchors to. The underlying rationale is that when a locator depends on positions resolving to highly unique nodes, it is more likely to identify the intended element unambiguously. Hence, although we refer to this uniqueness metric as *DOM position uniqueness*, it is in fact derived from the properties of the node currently occupying each position.

During XPath navigation, each step in the path incrementally narrows the search space by matching nodes. If the node residing at a DOM position is not unique within its context (i.e., subtree), a locator anchored at this position may match multiple nodes during evaluation, increasing ambiguity. In contrast, nodes with high uniqueness help the locator discriminate more effectively,

improving precision. Equation (2) formally defines the uniqueness of the node nd residing at a given DOM position, denoted as $U(DOM, nd, subtree_{nd_{ref}})$. The uniqueness is computed relative to a reference node nd_{ref}–that is, within the subtree rooted at $subtree_{nd_{ref}}$; here, nd_{ref} denotes the node at the last selected DOM position. The function $countSameNodes(DOM, nd, subtree_{nd_{ref}})$ counts how many nodes within this subtree are similar to nd at the position: if nd is an element node, the function counts element nodes with the same tag name; if nd is an attribute node, it considers attribute nodes with the same value. This count is then normalised by the total number of nodes in the same subtree (computed by $countNodesWithin(DOM, subtree_{nd_{ref}})$). To ensure that higher uniqueness corresponds to higher scores, we subtract the normalised count from 1.

$$U(DOM, nd, subtree_{nd_{ref}}) = 1 - \frac{countSameNodes(DOM, nd, subtree_{nd_{ref}})}{countNodesWithin(DOM, subtree_{nd_{ref}})} \tag{2}$$

Equation (3) combines the stability $S(DOM, M, nd)$ and uniqueness $U(DOM, nd, subtree_{nd_{ref}})$ of the node nd—used as a concrete representative of a DOM position— to compute the final position adequacy, denoted as $PositionAdequacy(DOM, M, nd, e_{target}, subtree_{nd_{ref}})$. Alongside these two factors, the calculation involves a distance factor $d_{dist(e_{target}, nd)}$, where e_{target} denotes the element that the constructed locator aims to identify. This distance ($dist$) reflects the number of hops between nd and e_{target} in the DOM tree, and is introduced to prevent DOM positions near the root from disproportionately receiving high stability values, as such positions tend to change less frequently than those near the leaves. An extreme case is the root itself, which, being occupied by the <HTML> node, would always appear the most adequate due to its constant presence and uniqueness. To mitigate this bias, the final adequacy score includes an exponential decay term, $d^{dist(e_{target}, nd)}$, where d is a constant empirically set to 0.95. The resulting adequacy values range between 0 and 1, favouring positions that are both stable and unique while being reasonably close to the target. These scores guide the selection of DOM positions that can act as reliable and precise anchors—structurally stable and likely to resolve to unique nodes—for constructing robust XPath locators.

$$\begin{aligned} PositionAdequacy(DOM, M, nd, e_{target}, subtree_{nd_{ref}}) &= S(DOM, M, nd) * d^{dist(e_{target}, nd)} \\ &\quad * U(DOM, nd, subtree_{nd_{ref}}) \\ dist(e_{target}, nd) &= length(e_{target}) - length(nd) \end{aligned} \tag{3}$$

Figure 4 illustrates how position adequacy values are computed and used to generate locators. The locator generation of EAGL consists of three phases, each detailed in the following sections.

3.3.2 Robust locator generation. The locator generation process in EAGL consists of three main steps. First, EAGL traverses the DOM path from the target element to its nearest unique ancestor and treats each DOM position along this path as a candidate anchor (***Candidate Anchor Selection***). The traversal and selection algorithm operates over the nodes currently residing at each DOM position, since nodes serve as concrete reference points and are more manipulable than positions themselves, as briefly noted in Section 3.3.1. Nevertheless, the selection is *conceptually over positions*, each treated as a potential anchor point for the final locator. It then predicts the stability of each candidate position—via the node currently residing there—and dynamically computes its uniqueness based on the nodes previously selected in the evolving XPath (***Final Anchor Selection***). These two values are combined to compute position adequacy, which determines whether the position

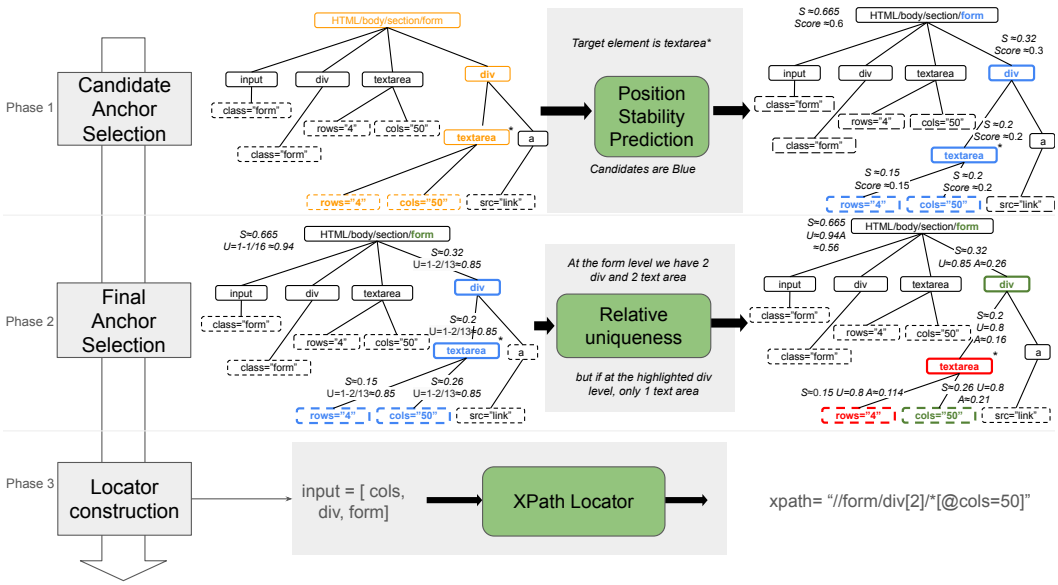

Fig. 4. An illustration of EAGL's three-phase locator generation, dashed borders are attribute nodes. In Phase 1, candidate nodes are selected – `form`, `div`, `textarea`, `rows`, and `cols` – highlighted in blue. In Phase 2, EAGL computes node adequacy (A) based on stability (S) and uniqueness (U) values (shown next to each node) and selects the final nodes: `form`, `div`, and `cols` (highlighted in green). In Phase 3, these nodes are used to construct the final XPath locator: `//form/div[2]/*[@cols=50]`.

should be included in the final locator. Lastly, EAGL constructs the XPath locator using the selected positions, resolved through their corresponding nodes (***Locator Construction***).

Algorithm 1 presents the overview of EAGL. Below, we will describe each phase along with the pseudocode of the main function of each phase.

Phase 1 - Candidate Anchor Selection. Phase 1 of EAGL selects candidate anchor positions for constructing an XPath locator for the target element. To define the search scope, EAGL invokes $findLocatorStartingNode$ (Line 2 in Algorithm 1, detailed in Algorithm 2), which traverses the DOM path from the root to the target element (e_{target}). It then examines the nodes along this path—each corresponding to a DOM position—in reverse order, from target to root. For each node at the candidate anchoring DOM position, it first checks whether the node has only one instance, i.e., unique, within the DOM (Lines 6-7). Only nodes that appear exactly once in the DOM are considered, thereby avoiding ambiguity in the locator. If the node appears only once within the context (i.e., the DOM), EAGL computes its predicted stability and combines this value with a distance-based weight (using an empirically set decay factor $d = 0.95$) to derive a final score ($score$, Lines 8–10). This score reflects the preference for positions that are both stable and closer to the target as the starting point. The node residing at the position with the highest score is selected as the starting point for constructing the candidate list. Using $getAllNodesOnPath$, EAGL then collects all element and attribute nodes along the DOM path from the selected starting point to the target element, in top-down order (from $node_{start}$ to e_{target}). These nodes correspond to the DOM positions that form the candidate set ($XPathList_{cand}$) from which the final locator will be composed. The top portion of Figure 4 illustrates this process for the target element `textarea`. EAGL first examines the path from `HTML` to `textarea` (highlighted in orange) and filters out non-unique nodes

Algorithm 1: EAGL. Robust Locator Generation

input : a DOM *dom*, a stability prediction model *M*, an element to locate e_{target}
output : a locator to the target element *target_locator*

1 $e_{current}, XPathList_{cand} \leftarrow e_{target}, []$
Phase 1. *Candidate Anchor Selection ($XPathList_{cand}$).*
/* Select a starting node of candidate search */
2 $node_{start} \leftarrow findLocatorStartingNode(dom, M, e_{target})$
/* Collect all element nodes on the DOM path from $node_{start}$ to e_{target}, and all attribute nodes associated with those elements in the top-down order Each node corresponds to a candidate anchoring position*/
3 $XPathList_{cand} \leftarrow getAllNodesOnPath(dom, node_{start}, e_{target})$
Phase 2. *Final Anchor Selection ($XPathList_{final}$)*
/* Select the nodes at the final anchors among the candidates */
4 $XPathList_{final} \leftarrow selectNodesOnXPath(dom, M, XPathList_{cand}, e_{target})$
Phase 3. *Locator Construction*
/* Construct an XPath locator from $XPathList_{final}$ and return it */
5 $locator_{target} \leftarrow constructLocator(dom, XPathList_{final})$
6 **return** $locator_{target}$

within the DOM (e.g., there are two `textarea` elements and multiple `cols` attributes with the same value). Among the unique candidates highlighted in red—`HTML`, `body`, `section`, `form`, and `div`—, the `form` node receives the highest score (e.g., $score = 0.6$ based on stability (S) and distance ($dist$)) and is thereby selected as the starting point. All nodes from `form` to `textarea`, including their associated attributes, are then collected as candidates (highlighted in blue)—each representing a candidate anchor position.

Phase 2 - Final Anchor Selection. In Phase 2, EAGL selects a subset of candidate anchors from the previously collected candidates ($XPathList_{cand}$) to be included in the final XPath locator (Line 4 in Algorithm 1). The selection is handled by the *selectNodesOnXPath* function (Algorithm 3), which evaluates candidates iteratively in a top-down order. For each candidate, it computes stability and dynamic uniqueness (Lines 10–27), and selects the one with the highest adequacy score as the next anchor—represented as a *node* in the algorithm ($node_{selected}$), but conceptually corresponding to a *DOM position*. This score is calculated as the product of the predicted stability (Lines 2–4), a distance-based decay factor (Line 14), and the uniqueness relative to previously selected anchors (Line 13). After each selection, EAGL narrows the scope to the subtree rooted at the selected anchor (Lines 11–12) and updates uniqueness accordingly (Line 13). This process continues until the target element is reached (Lines 23–27).

The second part of Figure 4 illustrates this process. In the example, the nodes at candidate positions are `form`, `div`, `textarea`, `rows(rows="4")`, and `cols(cols="50")` (in blue). Among them, `form` is selected first with the highest adequacy score (0.56). Next, `div` is selected (0.26). After selecting `div`, uniqueness values are updated within its subtree, resulting in a slightly decreased adequacy for `textarea` (from 0.17 to 0.16) and `cols` (from 0.22 to 0.21). In the final iteration, `cols(cols="50")` is selected. Since this attribute belongs to the target element (`textarea`), the algorithm terminates and returns the selected anchors: `form`, `div`, and `cols(cols="50")`.

Phase 3 - Locator Construction. Algorithm 4 presents the pseudo-code for the final phase of EAGL: constructing an XPath locator using the selected anchors, denoted as *constructLocator*. This function receives $XPathList_{final}$, the list of selected anchors, represented via nodes in the

Algorithm 2: findLocatorStartingNode

input : a DOM dom, a prediction model M, an element to locate e_{target}
output : a starting node of candidate search $node_{start}$

$node_{start} \leftarrow e_{target}$
$score_{highest} \leftarrow 0$
$dfactor \leftarrow 0.95$ // an empirically obtained distance factor value
/* Collect all element and attribute nodes along the DOM path from $dom.root$ to e_{target}, inclusive, in the top-down order */
$nodes_{visiting} \leftarrow getAllNodesOnPath(dom, dom.root, e_{target})$
/* Visit nodes in $nodes_{visiting}$ in reverse order, i.e., from e_{target} to $dom.root$, and select the node with the highest $score$ */
for $node_{curr} \in reverse(nodes_{visiting})$ **do**
    $cnt_{sameNode} \leftarrow countSameNodes(dom, node_{curr}, subtree_{dom.root})$
    if $cnt_{sameNode} == 1$ **then**
        $nodeStability \leftarrow computeStability(M, node_{curr})$
        $dist \leftarrow distance(node_{curr}, e_{target})$
        $score \leftarrow nodeStability * dfactor^{dist}$
        if $score_{highest} < score$ **then**
            $score_{highest} \leftarrow score$
            $node_{start} \leftarrow node_{curr}$
        end
    end
end
return $node_{start}$

algorithm, from Phase 2, ordered from the top-most to the target element. Based on this order, the algorithm generates a string representing the final XPath locator.

Lines 6–24 describe how the algorithm constructs the locator by iterating through each node in $XPathList_{final}$, each corresponding to a selected anchor position. It handles element and attribute nodes differently (Lines 15–23) and determines the appropriate path separator ("/" vs. "//") based on whether the current node is a direct child of the previously added node (representing a selected anchor position) in the locator path (Lines 7–14). For element nodes that are direct children, the algorithm appends their sibling index (Lines 20–22) to disambiguate among similar siblings and improve locator precision.

Figure 4 illustrates this process with an example. Given the selected list `form`, `div`, `cols`, which includes both element and attribute nodes representing selected anchor positions, the algorithm produces the XPath locator `//form/div[2]//*[@cols=50]`. The first anchor, `form`, is not the DOM root, so the locator begins as a relative path (`//`). The next anchor, `div`, is a direct child of `form`, so it is appended with its sibling index, yielding `/div[2]`. The final anchor, `cols`, is an attribute of an element that is a direct child of `div`, resulting in the relative step `/*[@cols=50]`.

4 EXPERIMENTAL SETUP

In this section, we present our research questions and the details of the experimental setup of our empirical study.

Algorithm 3: selectNodesOnXPath

input : a DOM dom, a stability prediction model M, an ordered list of candidate nodes $XPathList_{cand}$, an element to locate e_{target}
output : the final list of selected nodes to construct an XPath locator $XPathList_{final}$

/* Compute the stability values of candidate nodes
$nodeStabilityDict$: a dictionary with key = a node, $value$ = a stability of the node */
$nodeStabilityDict \leftarrow dict()$
for $node \in XPathList_{cand}$ **do**
  $nodeStabilityDict[node] \leftarrow computeStability(M, node)$
end
$adequacy_{highest}, XPathList_{final}, node_{selected} \leftarrow 0, [], null$
$e_{start} \leftarrow XPathList_{cand}[0]$ // get the first candidate node, i.e., the starting node
$node_{root} \leftarrow e_{start}$ // set the initial root node of the traversing subtree as e_{start}
$dfactor \leftarrow 0.95$ // an empirically obtained distance factor value
$node_{check} \leftarrow null$ // a node to check the termination condition
/* Select final nodes in steps, picking the best and skipping ahead each time */
do
  $subtree \leftarrow extractSubtree(dom, node_{root})$ // extract a subtree rooted at $node_{root}$ node
  for $node_{curr} \in (XPathList_{cand} \cap NodesOf(subtree))$ **do**
    /* Dynamically compute the uniqueness of a node using the current subtree */
    $uniqueness \leftarrow computeUniqueness(dom, node_{curr}, subtree)$
    $dist \leftarrow distance(e_{target}, node_{curr})$
    $adequacy \leftarrow nodeStabilityDict[node_{curr}] * dfactor^{dist} * uniqueness$
    if $adequacy > adequacy_{highest}$ **then**
      $adequacy_{highest} \leftarrow adequacy$
      $node_{selected} \leftarrow node_{curr}$
    end
  end
  $XPathList_{final}.append(node_{selected})$
  /* Update the root node of the next subtree to be searched to the last selected node */
  $node_{root} \leftarrow node_{selected}$
  /* Set $node_{check}$ to check the termination condition */
  $node_{check} \leftarrow node_{selected}$
  if $node_{selected}$ *is an attribute* **then**
    $node_{check} \leftarrow node_{selected}.parent$ // if $node_{selected}$ is an attribute, check the element to which it belongs
  end
while $node_{selected} \neq e_{target}$;
return $XPathList_{final}$

4.1 Research Questions

RQ1 Prediction Effectiveness: How effective is the stability prediction model of EAGL in predicting stable positions?

To answer RQ1, we train a model to predict the likelihood of changes in DOM positions based on past versions of different websites, and evaluate its performance on recent versions. Sections 4.3 and 4.4 provide details on the dataset and training process. Although the task is formulated as binary classification (change vs. no change), the model outputs continuous likelihood scores, which are used in downstream steps. Accordingly, we evaluate the model using both classification

Algorithm 4: constructLocator

input : a DOM *dom*, the final list of selected nodes to construct an XPath locator $XPathList_{final}$
output : an XPath locator $locator_{target}$ in a string form

$locator_{target} \leftarrow$ "//" // initially set the locator as a relative XPath
if $XPathList_{final}[0] = dom.root$ **then**
  $locator_{target} \leftarrow$ "/" // if start from $dom.root$, reset to "/" to make it an absolute XPath
end
$node_{prev} \leftarrow null$ // to keep track of a previously selected node
for $node_{curr} \in XPathList_{final}$ **do**
  if $node_{prev} \neq null$ **then**
    /* Check if $node_{curr}$ (or its parent, if attribute) is a child of $node_{prev}$ */
    if $(node.parent = node_{prev})$ *or*
    $((node\ is\ an\ attribute)\ and\ ((node.parent).parent = node_{prev}))$ **then**
      $locator_{target} \leftarrow locator_{target} +$ "/"
    end
    else
      $locator_{target} \leftarrow locator_{target} +$ "//"
    end
  end
  /* Append *nodeName* property: attribute → *name*, element → *tagName* (+ index if direct child) */
  if $node_{curr}$ *is an attribute* **then**
    $locator_{target} \leftarrow locator_{target} +$ "*[" $+ node_{curr}.name +$ "]"
  end
  else
    $locator_{target} \leftarrow locator_{target} + node_{curr}.tagName$
    if $node.parent = node_{prev}$ **then**
      $locator_{target} \leftarrow locator_{target} +$ "[" $+ node.position +$ "]"
    end
  end
  $node_{prev} \leftarrow node_{curr}$
end
return $locator_{target}$

metrics (Precision, Recall, F1, AUC-ROC) and a ranking metric (MAP), capturing its ability to both distinguish stable from changing DOM positions and prioritise stable positions for anchor selection.

The effectiveness of the stability prediction model may diminish over time due to the continuous evolution of the web, as its training data becomes outdated. This need for frequent updates to maintain the model's performance likely degrades the usability of EAGL. Hence, we further explore the model's sensitivity to web evolution, thereby asking:

RQ2 Sensitivity: How sensitive is the stability model to the continuous evolution of the web?

To assess this, we evaluate the performance of the stability prediction model on progressively newer data. We first split the dataset of past website versions into a 60:40 ratio, using the earlier 60% for training.[3] The remaining 40% is then divided into four equal folds (each representing 10% of the dataset). We then analyse how the model's effectiveness changes across these folds, assessing its capability to maintain the initial performance as it is tested on increasingly recent data.

[3]For RQ1, the training and test data ratio is 80:20.

After examining the effectiveness and sensitivity of the stability prediction model of EAGL, we now investigate whether the predicted stability can be effectively leveraged by EAGL to generate robust locators. Hence, we ask:

RQ3 Locator Effectiveness: How effective is EAGL in generating robust locators?

To answer this question, we evaluate the effectiveness of generated locators based on two key aspects: *precision* and *robustness*.

Locator Precision. A precise locator should correctly identify the intended element and *only* that element. This requirement concerns the uniqueness of the element identified by the locator: even if a locator correctly identifies an element, it cannot be considered *precise* if it matches multiple elements.

Locator Robustness. Sometimes, a locator being overly precise can make it highly susceptible to changes. For instance, an absolute XPath locator may break if any DOM position along its path changes. To remain effective over time, a locator must be robust—capable of withstanding changes in the DOM while consistently locating the intended element without breaking.

Section 4.5.2 describes the metrics used to evaluate these aspects.

To evaluate whether EAGL can generate precise and robust locators, we select the most recent N DOM versions per website, which are held out from the stability model training. We set N to 20% of the total versions, using the remaining 80% (the oldest DOMs) to train the stability model, following the same configuration as in RQ1. For each website, we use the oldest DOM among these held-out N versions to generate locators for 20 randomly selected elements from that DOM.[4] We then assess the robustness and the precision of these locators across the subsequent versions to evaluate their effectiveness.

The locator length is likely correlated with robustness, as longer locators tend to include more DOM positions, which increases the chance of breaking due to structural changes. Hence, while addressing this RQ, we further investigate the length of the generated locators, complementing our precision and robustness evaluation. In addition, we analyse how different locator generation approaches (single-locator vs. similarity-based) affect the observed precision–robustness trade-offs and practical effectiveness.

EAGL requires a sufficient history of past changes on the target DOM (e.g., the main webpage) to train its DOM position stability prediction model. However, in some cases—such as the early stage of website development or in the absence of a system for maintaining past versions—, such data may be unavailable. This leads to the following question:

RQ4 Generalisability: How well can EAGL generate effective locators when using a stability model trained on a different website within the same functional category?

To answer this question, we evaluate how well stability knowledge learned from a subset of websites generalises to other websites within the same functional category. For each category, we group all websites and perform a 3-fold cross-project evaluation: in each round, models are trained on two folds (websites) and tested on the remaining fold, ensuring that training and testing are always performed on different websites. This setup enables us to assess generalisability while reducing variance compared to training on a single website. We refer to this setting as cross-project training (within-category).

We compare the stability prediction effectiveness of cross-project models with models trained individually on each website to assess generalisability. Additionally, we evaluate the effectiveness

[4]We exclude elements that contain web links like "a", "link", and "src" from our targets since in the Wayback API, the timestamps are automatically added to the past links, making them change for every version.

of locators generated using the cross-project predictions, analysing the potential trade-off between prediction generalisability and downstream locator performance.

In practice, websites often undergo numerous changes due to fast development cycles [40]. In order to remain practical, EAGL should be able to generate locators within a reasonable time frame. Hence, we ask:

RQ5 Efficiency: How efficient is EAGL at generating robust locators?

To address this question, we measure the average execution time of EAGL, dividing it into two components: *model prediction time* and *locator generation time*. Model prediction time refers to the total time taken to predict the stability of individual positions within a target DOM, while locator generation time reflects the process of formulating an XPath locator with the predicted values. We further measure *the model training time* to evaluate the initial computational cost. All times are reported in seconds (s).

Since the efficiency of EAGL may depend on the structure of the input DOM, we further analyse the potential impact of DOM complexity on execution time. In particular, because the locator generation step involves identifying stable positions (i.e., locations) likely to yield precise locators, more complex DOM structures may increase processing time. To confirm this, we evaluate DOM complexity using two metrics: *the maximum depth* and *the total number of nodes in the DOM tree*, and inspect the correlation between this structural complexity and the performance of EAGL.

4.2 Baselines

4.2.1 Selenium. Selenium is a state-of-the-practice tool for locator-based web testing for its simplicity and effectiveness [1, 14, 20, 26, 27]. It supports standard locator strategies and seamless integration into the testing workflow. The standard locator strategies covered by Selenium are as follows [5]:

- **ID (By.id strategy)**: locates elements using their id attributes.
- **Name (By.name strategy)**: locates elements using their name attribute.
- **Class Name (By.className strategy)**: locates elements using their class attributes .
- **XPath (By.xpath)**: locates elements using full or relative XPath expressions.
- **Tag Name (By.tagName strategy)**: locates elements by their tag name.
- **Link Text and Partial Link Text (By.linkText and By.partialLinkText strategies)**: locates elements based on their visible text.
- **CSS Selectors (By.cssSelector strategy)**: locates elements using CSS selectors.

Selenium IDE automatically generates locators for elements using the strategies outlined above and ranks them based on internal heuristics to determine the most suitable locator. It also supports record-and-playback functionality, allowing users to record and rerun tests for validation. While useful, the recording session must be restarted manually when transitioning from one webpage or application to another. Since our experiments involve testing the main web pages of multiple websites, this limitation introduces unnecessary overhead by disrupting the workflow. To address this, we re-implemented the functionality of Selenium, more specifically, its standard locator strategies, enabling a more consistent process across multiple websites.

As mentioned earlier, Selenium uses internal heuristics to rank and select the most suitable locator among the candidate locators. However, the details of this ranking method are not publicly available. Hence, instead of relying on these heuristics, we generate and test all possible locators for a given element and select the best-performing for the experiment. We used the best-performing one for the following experiment of locator generation. This approach not only bypasses the limitations

[5] https://www.selenium.dev/documentation/webdriver/elements/locators/

of the availability of the internal ranking of Selenium but also allows us to investigate whether EAGL can outperform Selenium in any given scenario. In addition, we decided to focus on XPath and exclude CSS selectors in our implementation due to the similarity between these two locator strategies and the broad support of XPath in various automation tools and browsers [2, 4].

4.2.2 ROBULA+. ROBULA+ is a robust XPath generation method designed to tolerate structural changes in the DOM [26]. It builds upon the original ROBULA [24] by introducing additional transformation strategies and heuristics aimed at improving locator resilience. Starting from a generic XPath (`//*`), the method incrementally refines it through a fixed sequence of transformations—such as adding tag names (element types), attributes, and minimal position constraints—until it uniquely identifies the target node. ROBULA+ also prioritises empirically stable attributes (e.g., `id`, `name`) and avoids fragile ones (e.g., `style`, `onclick`) to reduce susceptibility to DOM changes.

While the authors provide a Firefox add-on and a Java implementation [24], these are not readily integrable into our large-scale, script-driven evaluation pipeline. To ensure compatibility with our Python-based infrastructure, we adopt and extend a third-party Python reimplementation [6]. We reviewed its logic against the original specification and, where needed, made adjustments, along with additional adaptations for our experimental setting (e.g., DOM snapshot handling).

4.2.3 Similo & VON Similo. Similarity-based web element localisation techniques, such as Similo [34] and its successor VON Similo [33], represent the state of the art in robust element relocalisation across evolving web pages. These approaches aim to map a previously identified element to its corresponding element in a new DOM version by comparing structural, attribute-based, and spatial features. VON Similo further incorporates visual information to account for visually overlapping nodes. Given a new page version, Similo and VON Similo compute similarity scores for candidate elements and return a ranked set of potential matches. While these methods are effective for robust element re-identification, they defer the final target selection to downstream usage or evaluation criteria, rather than committing to a single deterministic locator at generation time. In our study, we use the implementations and ground truth datasets released by the authors[7] to compare EAGL against these approaches.

4.3 Data Collection

4.3.1 Dom version history collection. To build a representative dataset of real-world web evolution, we used the Wayback API [18] to collect historical snapshots of websites. The Wayback Machine (Wayback, for short) is a non-profit digital archive that preserves past versions of public web pages. Using its API, we retrieved the DOM structures of the main webpages for 181 websites randomly selected from the top 1,000 most visited domains, ranked by Alexa.[8] Since some archived snapshots were spaced only seconds apart—offering little meaningful difference—we enforced a *minimum interval of one day* between collected versions. For each website, we retrieved up to 1,000 recent versions, subject to this time constraint.

Table 1 summarises the statistics of the ten largest websites in the dataset, ranked by total number of DOM elements; for simplicity, we assigned subject IDs (S1 to S10) to these top ten projects. The fourth column shows the number of versions successfully collected. As shown in this table, some websites yielded fewer than 1,000 versions due to parsing failures in some archived DOMs. During collection, we observed that some DOM versions retrieved from the Wayback API were corrupted,

[6] https://github.com/ZeusFSX/robula-plus
[7] https://figshare.com/s/e63c3679e925730397ac?file=37900863
[8] https://www.htmlstrip.com/alexa-top-1000-most-visited-websites

altered from the original content. To filter out such cases, we applied a constraint that the parsed DOM must contain at least 20 elements within the <html> tag.

4.3.2 Ground truth. We construct two types of ground truth: (1) DOM position changes, used to train and evaluate the stability prediction model of EAGL, (2) locator breakage, used for the overall evaluation of robust locator generation.

DOM position changes for stability prediction. A DOM position resolves to either an element node or an attribute node. In this section, we distinguish between these two cases and describe how we define and collect ground-truth stability for DOM positions in each case.

Positions Resolving to Element Nodes. We define the structural persistence of a DOM position based on its presence across versions, using exact XPath matching. If an absolute XPath expression resolving to a given DOM position appears in two subsequent versions, we label the position as stable—unchanged; otherwise, it is unstable.

Positions Resolving to Attribute Nodes. For attribute nodes, we restrict ground-truth labelling to `id` attributes only. While any attribute can reside at a DOM position, many—such as `class`, `style`, or `data-*`—are often shared, reused, or tied to semantic or visual roles. These attributes are prone to frequent, non-structural changes. For instance, a developer may update the general style of buttons by replacing their class attributes with a new CSS class; while the DOM structure remains intact, the attribute values change across elements, showing their instability to be structural anchors. In contrast, `id` attributes are defined to be unique within the DOM and often serve as structural anchors in XPath locators (e.g., `//*[@id="foo"]`). Their consistent structural role across versions makes them a reliable signal of persistence. By labelling only positions that resolve to `id` attributes, we reduce noise from semantically unstable attributes and preserve EAGL's position-centric modelling focus.

Locator breakage for robustness evaluation. To evaluate the effectiveness of generated locators, we require ground-truth information about whether a locator remains unbroken—that is, whether it continues to locate the correct element— as the DOM evolves. Ideally, this would involve mapping elements across DOM versions, which is known to be a costly and still unresolved challenge, as noted in prior work [23, 24, 26, 33, 46]. To address this, we adopt a scalable approximation. We consider a locator unbroken if it continues to return exactly one element in a given DOM. If it returns no element, we treat it as broken. If it returns multiple elements, we consider it imprecise – partial breakage. While this heuristic does not guarantee correctness, it aligns with known failures of brittle XPath locators and provides a practical, conservative proxy for large-scale evaluation. Section 6.4.1 further examines the reliability of this approximated locator breakage ground truth through manual validation on a representative subset of the studied website.

4.4 Model Training

4.4.1 Learning Algorithm. We use Random Forest (RF), an ensemble learning algorithm of decision trees, to train models that predict the stability of DOM positions—that is, their likelihood of remaining unchanged across future versions. Since EAGL relies on ranking DOM positions by their relative stability, we use the regressor variant of RF, which outputs continuous values more suitable for this purpose than classification probabilities. To validate our choice, we compared the RF regressor with two alternatives: the RF classifier and the XGBoost regressor. The RF regressor consistently showed lower prediction error (mean squared error of 0.112) than the RF classifier (0.170) and slightly outperformed the XGBoost regressor (0.118). Given its efficiency and effectiveness, we adopt the RF regressor as the stability prediction model in EAGL.

Table 1. The largest ten studied websites: the statistics of the top ten largest websites and the aggregation of the entire studied subjects. # Node$_{chgd}$ refers to the average number of changed nodes per version, # Node$_{unchgd}$ counts those unchanged.

Site	SID	period	# versions	# Position$_{unchgd}$	# Position$_{chgd}$
aliyun	S1	2019.08.08 - 2022.08.18	996	6245	673
hulu	S2	2019.11.28 - 2022.08.22	995	2884	2322
digikala	S3	2018.12.16 - 2022.02.19	829	4515	204
allegro	S4	2019.11.09 - 2022.08.21	841	3140	1328
softonic	S5	2019.10.22 - 2022.08.19	1000	3416	306
behance	S6	2019.11.26 - 2022.08.20	962	2478	1245
dkn	S7	2019.05.16 - 2022.09.22	995	2794	474
slickdeals	S8	2019.12.11 - 2022.09.23	995	2104	1159
uol	S9	2019.11.11 - 2022.08.04	971	2596	734
theepochtimes	S10	2019.12.30 - 2022.09.23	999	2977	256
Others (171)		—	871	1078	221
Total (181)		—	924	1205	258

4.4.2 Training Configuration. To train the model, we use DOM positions that resolve to element nodes and `id` attribute nodes only, as described in Section 4.3.2. We exclude other attribute types during labelling to avoid noise from semantically unstable attributes. While the training labels for attribute nodes are limited to `id`s, the model is applied to predict the stability of DOM positions resolving to all attribute types. The rationale and implications of this generalisation are discussed in Section 6.3

To construct the training and test datasets, we first sort all web pages(versions) chronologically within each project. We then take the earliest 80% for training and the remaining 20% for testing (except for RQ2, which is about sensitivity; see Section 4.1), ensuring the model is trained on earlier versions to predict the stability of nodes in future versions; later locator evaluation is on this set of the latest 20%. To mitigate potential bias and improve experimental robustness, we repeat the training process ten times, yielding ten different models for each web project. We then select the model whose R-squared score on the test set is closest to the average test R-squared across all runs to ensure selecting a representative model.

4.4.3 Feature computation. We use six features, three neighbour-based and two location-based, and one to differentiate between attributes and elements (Section 3.2.1). For all the features, we used a min-max scaler to normalise all the feature values to be between zero and one per unique DOM, meaning that we normalised each DOM in a project separately.

4.5 Evaluation Metrics

This section introduces the metrics used to evaluate the EAGL's performance in the prediction of DOM position stability and the effectiveness of its generated locators.

4.5.1 Stability Prediction. We use four standard classification metrics—Precision, Recall, F1-score, and ROC-AUC—to evaluate the *classification performance* of the stability prediction model, with changed and unchanged positions labelled as 0 and 1, respectively. To assess *ranking performance* of the model, we use Mean Average Precision (MAP), which captures how highly stable nodes (i.e., nodes less likely to change) are ranked based on their predicted change likelihoods. The stability prediction metrics assess the predictive performance of the models in forecasting structural changes. The reported values indicate how well the models distinguish stable nodes from likely-change

nodes. While these metrics do not directly measure locator quality or usability, they support the prioritisation of structurally stable nodes used in subsequent locator generation.

Equation (4) and Equation (5) define Average Precision (*AP*) and *MAP*, respectively. *AP* reflects the average precision at the rankings of all unchanged positions, which are resolved to nodes, within a DOM. For each unchanged position to node n in DOM D, precision at rank $r(n)$ is denoted as $Prec(n, D)$, and is computed as the number of unchanged nodes ranked above or at n, divided by $r(n)$, where $r(n)$ is the ranking of node n when sorting all nodes in descending order of their predicted stability. *MAP* is then computed as the average of *AP* scores across all DOMs in a given subject. Specifically, S denotes a subject in the dataset, D_S refers to the set of DOMs (one per version) from the main webpage of subject S, and *isUnchgd(n)* indicates whether the position resolved to node n is unchanged (1) or not (0).

During evaluation, we follow a three-step process. First, we exclude DOM versions that contain no changes, as such cases provide no meaningful data for change prediction or *MAP* computation– *AP* is undefined when there are no positive labels (denominator being zero). Second, we compute all evaluation metrics at the level of individual DOM versions, as the model predicts changes per DOM snapshot. Finally, we report the average of these per-DOM metric values for each project.

$$AP(D) = \frac{\sum_{n \in D} \text{Prec}(n, D) \cdot \text{isUnchgd}(n)}{\text{the total number of changed DOM positions}} \tag{4}$$

$$MAP(S) = \frac{1}{|D_s|} \sum_{D \in D_s} \text{AP}(D) \tag{5}$$

4.5.2 Locator Effectiveness. After evaluating the prediction effectiveness of EAGL's stability model, we assess the effectiveness of its locator generation module. As introduced in Section 4.3.2, we use a practical proxy to detect locator breakage, which distinguishes between complete breakage (*no elements returned*), partial breakage (*multiple elements returned*), and correct resolution (*exactly one element returned*). Based on this, we define proxy precision (*ProxyPrec*) as the inverse of the number of elements returned by a locator (*numLocated*) (Equation (6)), operationalising locator precision in terms of element uniqueness: locators that return fewer elements are treated as more precise. Manual validation across returned-element cases confirms that locator correctness degrades as the number of returned elements increases, motivating the use of *ProxyPrec* as a graded surrogate for locator precision rather than a binary correctness measure.

To evaluate **locator precision**, we first compute the average *ProxyPrec* per locator across versions for each website. We then take the average across all locators to avoid bias from different survival durations. Once a locator experiences complete breakage (i.e., returns no elements), we stop tracking its *ProxyPrec*, and values beyond the breakage point are not included in the average. Hence, although *ProxyPrec* is defined for the case of complete breakage (zero), such values are excluded from the actual computation during evaluation. We note that *ProxyPrec* may overestimate locators that return multiple elements none of which correspond to the true target. Accordingly, reported precision results should be interpreted as surrogate measures of locator specificity rather than exact correctness, with empirical correctness support limited to single-element resolutions.

$$ProxyPrec(locator) = \begin{cases} \frac{1}{numLocated(locator)} & \text{if } numLocated(locator) \geq 1 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

$$DoS(locator) = \text{the number of versions until a } locator \text{ breaks} \tag{7}$$

$$RDoS(locator_A, locator_B) = DoS(locator_A) - DoS(locator_B) \tag{8}$$

4.6 Implementation and Environment

For our experiments, the CPU had a clock speed of 2.6 GHz and one core. The operating system (OS) used was Windows with the Linux terminal WSL [3] installed. The system's RAM was 32GB, which affected the speed and efficiency of the system's performance. The primary programming language used for this project was Python, a high-level, general-purpose language popularly used in scientific computing, data analysis, and machine learning. We made the implementation of EAGL publicly available [9]. The repository contains all the necessary code and documentation to use EAGL. The repository also includes the implementation of the ROBULA+ algorithm for generating locators and Selenium. Moreover, since our experiments included collecting data from the Wayback Machine, we provide an easily executable code version without the websites. The data of the processed DOMs (i.e. extracted features with the ground truth) and the trained models for the websites are publicly available.[10]

5 RESULTS

5.1 RQ1. Prediction Effectiveness

Overall (T), the stability prediction model demonstrates acceptable performance, with precision (0.872) and recall (0.845) being relatively high, while F1 (0.802) and ROC-AUC (0.714) indicate moderate classification performance. From the ranking perspective, the model achieves a MAP of 0.832, suggesting a moderately strong in prioritising stable DOM positions over those non-stable. Taken together, these results indicate that the model performs reliably across both classification and ranking tasks.

Among the top 10 web applications (S1–S10), we observe variations across classification and ranking metrics. S6 and S8 exhibit the lowest MAP values (0.587 and 0.557, respectively), which may be related to their high ratios of changed positions (Table 1 such a high change ratio could reduce the discriminative power of the current feature set, making it more challenging for the model to separate stable and unstable nodes. In contrast, S7 obtains a high MAP of 0.902 despite low recall (0.215), F1 (0.248), and ROC-AUC (0.448), indicating that while the model captures fewer stable elements overall due to the classification threshold, it can still effectively rank stable positions above unstable ones. S9 shows a somewhat similar pattern, with relatively lower recall (0.457), F1 (0.524), and ROC-AUC (0.668) but a higher MAP (0.758), suggesting that ranking performance remains robust even when classification metrics are moderate. The remaining sites (S1, S2, S3–S5, and S10) perform well across both classification and ranking metrics, with moderate to high precision, recall, and MAP values, reflecting balanced predictive capability. For the remaining 171 projects (O), the average performance is similar to that of the total 181 projects (T). This further confirms the acceptable performance of the stability prediction model in both classification and ranking tasks.

Answer to RQ1: The stability prediction model of EAGL, on average, achieves precision (0.872), Recall (0.845), F1 (0.802), ROC-AUC (0.714), and MAP (0.832), indicating moderate performance in both classification and ranking. Among the top 10 projects, S6 and S8 show relatively lower MAP values (0.587 and 0.557, respectively), which is likely due to high change ratios, whereas S7 and S9 show weaker classification performance but maintain strong ranking performance (MAP = 0.902 and 0.758, respectively). The remaining projects in the top 10 (S1, S2, S3–S5, and S10) perform consistently well across in both classification and ranking.

[9] https://github.com/hilalosoft/EAGL
[10] https://doi.org/10.5281/zenodo.7649059

Table 2. Stability Prediction Effectiveness. nr_{ver} denotes the number of unique versions of DOM in test data per project out of 200. All values are reported on average for each project.

		Classification				Ranking
Site	**nr_{ver}**	**Precision**	**Recall**	**F1**	**ROC-AUC**	**MAP**
S1	199/200	0.891	0.939	0.893	0.653	0.883
S2	199/200	0.725	0.822	0.653	0.729	0.706
S3	166/200	0.957	0.964	0.959	0.796	0.915
S4	168/200	0.911	0.711	0.782	0.694	0.835
S5	200/200	0.953	0.887	0.901	0.631	0.952
S6	193/200	0.604	0.679	0.430	0.703	0.587
S7	199/200	0.968	0.215	0.248	0.448	0.902
S8	199/200	0.729	0.918	0.736	0.817	0.557
S9	194/200	0.820	0.457	0.524	0.668	0.758
S10	200/200	0.887	0.972	0.914	0.872	0.762
O(171)	181/200	0.874	0.850	0.808	0.715	0.834
T(181)	182/200	0.872	0.845	0.802	0.714	0.832

5.2 RQ2.Sensitivity

Table 3 reports the model's performance when tested on progressively newer DOM versions, split into four chronological folds. Each metric is computed as the average across DOM versions per fold, and results are reported separately for the top-10 projects (S1–S10), the aggregated 171 remaining projects (O), and overall (T). Figure 5 visualises these sensitivity analysis results using box-plots from fold 1 to fold 4, where each data point represents the metric value for a single DOM version within the corresponding fold.

Overall, across 181 projects (T), EAGL maintains mostly consistent performance as the test data moves further away from the training data. MAP remains stable across all folds (0.792-0.834), confirming that the model retains its ability to prioritise stable DOM positions, similar to what we observed in RQ1. Meanwhile, recall shows a mild decrease (0.922 to 0.834), and precision remains stable (0.858–0.870). Together, these results indicate that the stability prediction model of EAGL retains its core ability to prioritise stable anchors despite increasing temporal distance.

Among the top 10 projects, S2 (Hulu) shows a clear upward trend, with precision improving from 0.447 to 0.771 and MAP increasing from 0.430 to 0.743 across all folds; F1 also improves, likely by the gain in precision. A closer inspection of Hulu's DOM versions suggests that previously frequent updates became more stable in later versions, which may have contributed to this improvement. In contrast, S7 exhibits a sharp drop in recall (0.986 to 0.299, and to the lowest, 0.131 at fold 3), while precision and MAP remain high. This indicates that although the threshold-based classification performance drops, the model continues to rank stable positions effectively above unstable ones.

For the other top projects, performance fluctuates slightly but shows no clear pattern of degradation across newer versions. This highlights that EAGL can generalise well as DOM versions evolve, with MAP consistently reflecting the model's capacity to prioritise stable anchors even under temporal drift.

Table 3. Sensitivity analysis. Per-website evaluation results of the top 10 largest sites across four chronological folds (*fold 1* to *fold 4*). Each value represents the fold-wise average.

	Classification																Ranking			
	Precision				Recall				F1				ROC-AUC				MAP			
Site	fold1	fold2	fold3	fold3	fold1	fold2	fold3	fold4	fold1	fold2	fold3	fold4	fold1	fold2	fold3	fold4	fold1	fold2	fold3	fold4
S1	0.974	0.917	0.890	0.891	0.991	0.964	0.966	0.911	0.980	0.926	0.900	0.887	0.605	0.732	0.708	0.598	0.953	0.864	0.871	0.895
S2	0.447	0.476	0.680	0.771	0.988	0.873	0.825	0.818	0.475	0.453	0.619	0.687	0.889	0.681	0.638	0.820	0.430	0.468	0.668	0.743
S3	0.959	0.965	0.970	0.944	0.976	0.970	0.966	0.961	0.966	0.967	0.968	0.951	0.862	0.815	0.848	0.743	0.899	0.926	0.930	0.900
S4	0.942	0.912	0.889	0.933	0.799	0.737	0.733	0.690	0.859	0.809	0.780	0.783	0.746	0.657	0.692	0.695	0.861	0.867	0.822	0.849
S5	0.959	0.933	0.996	0.909	0.923	0.869	0.905	0.868	0.925	0.866	0.948	0.854	0.685	0.639	0.669	0.593	0.942	0.944	0.996	0.907
S6	0.657	0.614	0.606	0.603	0.907	0.770	0.636	0.723	0.629	0.483	0.390	0.470	0.883	0.781	0.706	0.700	0.554	0.578	0.585	0.589
S7	0.959	0.929	0.970	0.965	0.986	0.714	0.131	0.299	0.971	0.691	0.168	0.327	0.981	0.760	0.407	0.489	0.791	0.790	0.918	0.887
S8	0.693	0.557	0.789	0.669	0.879	0.893	0.903	0.932	0.681	0.591	0.788	0.684	0.802	0.605	0.811	0.823	0.557	0.483	0.608	0.506
S9	0.846	0.877	0.855	0.785	0.503	0.450	0.452	0.462	0.574	0.552	0.540	0.508	0.699	0.702	0.675	0.661	0.742	0.809	0.790	0.725
S10	0.966	0.965	0.906	0.868	0.979	0.978	0.975	0.969	0.972	0.972	0.930	0.899	0.944	0.941	0.888	0.856	0.831	0.827	0.777	0.748
O(171)	0.859	0.874	0.878	0.872	0.924	0.895	0.863	0.838	0.838	0.836	0.818	0.799	0.825	0.773	0.724	0.707	0.794	0.822	0.837	0.835
T(181)	0.858	0.871	0.876	0.870	0.922	0.891	0.857	0.834	0.836	0.830	0.811	0.794	0.824	0.771	0.723	0.706	0.792	0.818	0.834	0.831

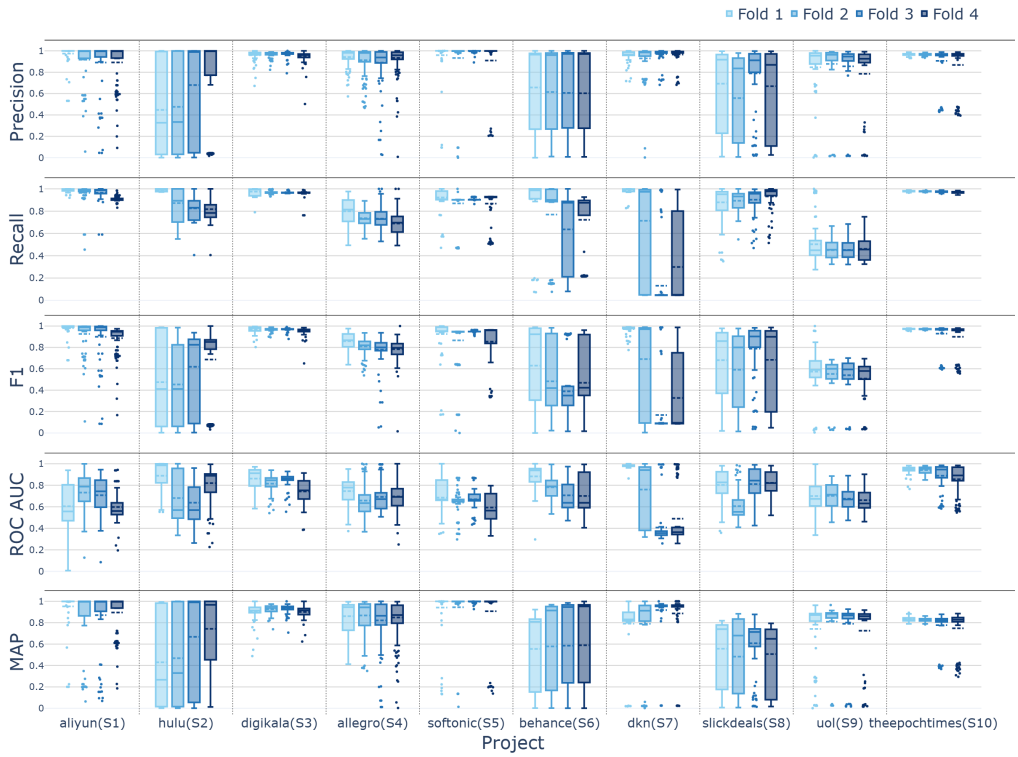

Fig. 5. Sensitivity analysis. Per-website evaluation results of the top 10 across four chronological folds (*fold 1* to *fold 4*). The dotted line indicates the mean.

Answer to RQ2: The stability prediction model of EAGL maintains mostly consistent performance as the test data moves further from the training data, with MAP remaining stable across folds (between 0.792 and 0.834). While recall decreases slightly (0.922 to 0.834), precision remains stable (0.858–0.870). Combined, these indicate that the model's core ability to prioritise stable DOM positions remains robust to temporal drift.

5.3 RQ3. Locator Effectiveness

Table 4 compares the effectiveness of EAGL with Selenium and ROBULA+ in terms of both precision and robustness. The second column of this table shows the number of DOM versions used for evaluation. For each project, locators are generated on the oldest version among the most recent 200 versions and then tested across the subsequent 199 versions (or fewer, if some versions are missing). The number of evaluated versions differs across projects due to the delay in rendering and dramatic structural changes (e.g., shift in the domain). Overall, around 55% of the past DOM versions were selected for the evaluation—6 to 194 past versions were dropped across projects.[11]

For precision, we report the average Proxy Precision per project for the top 10 (S1–S10), which is obtained by first averaging the values per locator (20 locators per project, tracked until complete breakage), and then averaging across these locators. For O (171 smaller projects) and T (all 181 projects), we further compute the average of these per-project Proxy Precision values. Among the top 10 projects (S1–S10), EAGL achieves the highest precision in 8 out of 10 cases, with values ranging from 0.894 to 1.000, while Selenium and ROBULA+ perform comparatively lower. This trend persists in the remaining 171 projects (O), where the average Proxy Precision of EAGL is 0.976, compared to 0.893 for Selenium and 0.850 for ROBULA+. Across all 181 projects (T), EAGL consistently maintains the best performance, with an overall Proxy Precision of 0.976 versus 0.892 for Selenium and 0.841 for ROBULA+.

The third and fourth columns in Table 4 compare EAGL with Selenium and ROBULA+ in terms of survival against DOM changes (evolution), showing how frequently EAGL's locators last longer (W), shorter (L), or the same (T) as those of the baselines, with the average relative duration of survival (RDoS) given in parentheses. For instance, a cell with '9 (32)' under W indicates that EAGL has nine locators that survive longer than the baseline, with an average survival advantage of 32 DOM versions. Within the top 10 projects (S1–S10), the project-level wins between EAGL and Selenium are evenly split (5 wins vs. 5 losses), but EAGL achieves better performance at the locator level, with 40 locators outlasting Selenium compared to 31 in the opposite direction. The RDoS values range from 2 to 108 for EAGL and 9 to 105 for Selenium, reflecting that both achieve longer survival in different scenarios rather than one consistently dominating, though ties happen most frequently. When wins occur, the wins often cluster under the same tool within a project rather than alternating (e.g., 9 (32) vs. 1 (105) in S4). For the remaining 171 projects (O), EAGL outperforms Selenium in 560 locators versus 443, with a longer average RDoS (44 vs. 38). At the project level, EAGL wins in 86 projects compared to 64 for Selenium. The trend is maintained when considering all 181 projects (T), where EAGL surpasses Selenium in 600 vs. 482 locators, with an average RDoS of 43 vs. 39, and wins in 86 projects versus 64 for Selenium. In comparison with ROBULA+, EAGL performs better. Among the top 10 projects, EAGL wins in 5 projects compared to 2 for ROBULA+, with 19 vs. 6 locators and similar RDoS ranges. Across the remaining 171 projects (O), EAGL wins in 88 projects compared to 53 for ROBULA+, with 398 vs. 263 locators and an average RDoS of 46 vs. 40. Across all 181 projects (T), EAGL shows a clear lead, with 426 vs. 278 locators and an average RDoS of 46 vs. 41, winning in 92 projects compared to 55 for ROBULA+. From these results, we posit that EAGL can outperform both Selenium and ROBULA+ by generating locators that are more robust across evolving DOM versions.

After inspecting relative performance, we further examine the overall robustness of all generated locators by comparing their average DoS values (fifth column of Table 4). For the top 10 projects (S1–S10), DoS is reported as the average across the 20 locators per project, while for O (171 projects)

[11] Most past DOM versions of S3 were excluded due to a rendering timeout (>2 mins). The site is in Persian, involving RTL layout and font shaping, which may contribute to observed delays. S9 underwent a major domain shift from e-commerce to news, leading to the exclusion of earlier DOMs due to structural changes

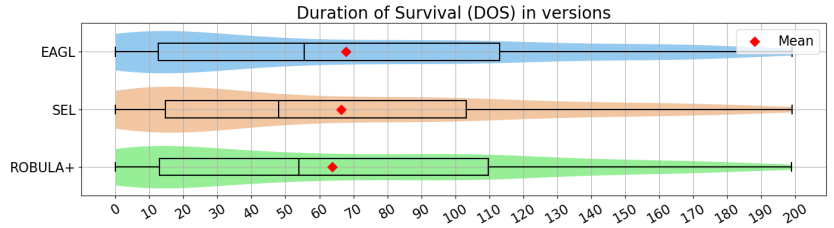

Fig. 6. Duration of survival comparison between the locator generation techniques. The x-axis is the number of versions that the locators

and T (all 181 projects), we report the average DoS across projects. Overall, the average DoS across all projects (T), which measures the general robustness of the generated locators, is highest for EAGL at 69.1, compared to 64.4 for Selenium and 67.4 for ROBULA+. Figure 6 illustrates the distribution of average DoS values across the 181 projects. While the differences are not dramatic, EAGL achieves better performance with higher median, mean, and 75th percentile values. Combined with the results of relative performance, we posit that EAGL is capable of generating locators that are relatively more robust than those produced by the baselines.

The length of a locator often relates to its fragility, as longer locators have more potential breakage points. Overall, the locators generated by ROBULA+ are the shortest, with an average length of 3.365, followed by EAGL with 4.649, while Selenium produces the longest locators with an average length of 7.971. Considering that EAGL achieves higher Proxy Precision and robustness, we posit that although EAGL does not generate the shortest (and potentially easiest-to-maintain) locators, its locators remain both maintainable and effective.

Table 4. Locator effectiveness across websites. nr_{ver} is the number of tested versions. *RDoS* (in parentheses) and *DoS* show the average robustness across all locators per website, while 'W,' 'L,' and 'T' indicate aggregated win/loss/tie counts. For the 'O' and 'T' group—the group of websites outside the top 10 and the total set of 181 studied websites—, averages across websites are reported for *ProxyPrec*, *RDoS*, *DoS*, and locator length; in these two, W/L/T are shown as aggregated counts in the form 'A/B', where 'A' counts DOM versions and 'B' counts individual locators.

		ProxyPrec			Robustness									Locator Length		
					EAGL vs Sel			EAGL vs R+			DoS					
Site	nr_{ver}	EAGL	SEL	R+	W (RDoS)	L (RDoS)	T	W (RDoS)	L (RDoS)	T	EAGL	SEL	R+	EAGL	SEL	R+
S1	194/199	0.989	**0.990**	0.740	0 (0)	**2** (80)	18	1 (192)	**3** (82)	16	87	**95**	81	5.950	8.685	**3.000**
S2	107/199	**0.999**	**0.999**	0.950	**1** (108)	0 (0)	19	1 (66)	1 (2)	18	**91**	86	86	4.850	7.698	**3.444**
S3	5/199	**1.000**	0.902	0.575	**2** (2)	0 (0)	18	**4** (2)	0 (0)	16	**3**	**3**	2	6.350	8.537	**3.000**
S4	159/199	0.967	**0.995**	0.915	**9** (32)	1 (105)	10	5 (26)	5 (69)	10	30	21	**45**	**3.700**	15.500	5.556
S5	165/199	**0.894**	0.849	0.764	**17** (65)	1 (68)	2	**5** (4)	1 (68)	14	**73**	21	68	**7.000**	10.094	7.667
S6	191/199	**0.960**	0.769	0.492	0 (0)	**2** (105)	18	2 (35)	**3** (191)	15	58	68	**124**	7.050	13.963	**3.000**
S7	158/199	**0.976**	0.750	0.476	0 (0)	**1** (156)	19	**4** (157)	0 (0)	16	127	**135**	54	6.350	9.222	**3.444**
S8	109/199	**1.000**	0.845	0.490	0 (0)	**17** (9)	3	**1** (99)	0 (0)	19	17	**25**	23	3.650	6.086	**2.300**
S9	15/199	**0.963**	0.837	0.773	0 (0)	**9** (11)	11	0 (0)	0 (0)	20	9	**15**	8	4.950	12.700	**3.400**
S10	91/199	**1.000**	0.862	0.782	**11** (10)	6 (74)	3	**5** (73)	2 (51)	13	59	**80**	31	6.450	7.204	**4.333**
O(171)	109/199	**0.976**	0.893	0.850	**81/560(44)**	59/443(38)	2447	**88/398(46)**	53/263(40)	2789	**70**	65	68	4.591	7.854	**3.333**
T(181)	110/199	**0.976**	0.892	0.841	**86/600(43)**	64/482(39)	2568	**92/426(46)**	55/278(41)	2946	**69**	64	67	4.649	7.971	**3.365**

We further evaluated EAGL on the VON Similo replication package[12], Which provides manually curated ground-truth element pairs across two versions of each website. The replication package contains evaluation results for both Similo and VON Similo on the ground-truth. The ground-truth

[12]https://figshare.com/s/e63c3679e925730397ac?file=37900863

set obtained from the repository contains 606 element pairs; the original Similo paper reports results on 598 pairs. In our study, EAGL could be executed on 526 element pairs. The remaining cases were excluded due to archival rendering failures or invalid ground-truth locators that returned no match. These exclusions account for the difference in the total number of evaluated pairs reported in Table 5. Since our interest in this comparison lies in relative non-located rates under the ground-truth definition used by the Similo benchmark, we reuse the results for Similo and ROBULA+ as reported by their authors and evaluate EAGL under the same success criterion [33].

Table 5 reports robustness results on the Similo ground-truth dataset. On the 526 executable pairs, EAGL successfully locates 385 targets, corresponding to a non-located rate of 27%, which is lower than the reported non-located rate of ROBULA+ (35%). Similo reports a lower non-located rate (12%), reflecting its evaluation criterion that considers a target located if it appears among a ranked list of candidate matches. These results highlight a methodological difference between similarity-based multi-candidate approaches and single-locator generation approaches, rather than a direct equivalence in success criteria.

Table 5. Robustness on the Similo ground-truth dataset. Reported results for Similo and ROBULA+ are taken from the original study (598 pairs). Results for EAGL are computed on 526 executable pairs. **Located** counts successful element locations and **Non-located** counts the failure, The percentage of **Non-located** is computed over the total number of pairs for each method.

Method	Total Pairs	Located	Non-located (%)
ROBULA+ (reported)	598	387	211 (35%)
Similo (reported)	598	526	72 (12%)
EAGL	526	385	141 (27%)

Similarity-based approaches such as Similo and VON Similo return ranked candidate elements to tolerate structural differences across versions, whereas single-locator approaches generate one executable locator per target element. As a result, similarity-based methods defer target element selection to downstream evaluation or usage, while single-locator approaches expose failures directly during automated execution. In the Similo benchmark, VON Similo further treats visually overlapping elements as correct matches during evaluation, which differs from the stricter success criterion used by single-locator approaches. This difference reflects distinct design and evaluation choices rather than equivalent notions of correctness. Finally, the Similo benchmark evaluates element matching across versions separated by multiple years. While this setting captures long-term evolution, it differs from scenarios involving frequent incremental changes, which are the primary focus of EAGL. In the case of VON Similo, the results are better because VON Similo considers all visually overlapping nodes as correct. EAGL removes this ambiguity by using traditional single locators that are specific and withstand evolution on day-to-day changes. Another aspect is that the evaluation of Similo and VON Similo uses two versions of each website that are 2-4 years apart which does not reflect frequent changes websites undergo.

Table 6 further shows the precision-recall trade-off among the evaluated approaches at the best performing threshold according to the original work for Similo of 0.28 and the best performing threshold for VON Similo of 0.40 [33]. At 0.28 threshold, Similo and VON Similo emphasise recall (0.8727 and 0.9888, respectively), reflecting their design goal of retrieving the correct element within a ranked list of candidate matches, which results in lower precision (0.8153 and 0.8244, respectively). While at the threshold 0.40 Recall drops (0.6483 and 0.9286, respectively) for both to increase precision (0.9062, 0.9651, respectively). In contrast, EAGL generates a single deterministic locator per target and achieves higher precision (0.9821), with competitive overall effectiveness

of F1 being 0.8397. This behaviour reflects a different operating point and is desirable in practice. For instance, when web pages contain many visually or structurally similar elements (e.g., menus, lists, or news portals), returning multiple plausible matches may increase the risk of flaky test outcomes, whereas a single precise locator simplifies diagnosis and targeted repair. We note that the evaluation of VON Similo considers visually overlapping elements as correct matches, even when they do not correspond to the oracle-specified target. We report the complete results in the online repository, including all locators generated by EAGL.

Table 6. Precision score comparison between Similo and EAGL

Site	Threshold	Precision	Recall	F1
Similo	0.28	0.8153	0.8727	0.8430
	0.40	0.9062	0.6483	0.7559
VON Similo	0.28	0.8244	0.9888	0.8991
	0.40	0.9651	0.9286	0.9465
EAGL	NA	0.9821	0.7333	0.8397

Answer to RQ3: EAGL demonstrates higher overall effectiveness in generating locators compared to Selenium and ROBULA+. It achieves consistently higher precision (Proxy Precision of 0.976 vs. 0.892 and 0.841) and better robustness, with longer average survival durations (DoS of 69.1 vs. 64.4 and 67.4) and more locators outlasting the baselines. While ROBULA+ produces the shortest locators, EAGL balances locator length (4.649 elements) with better precision, and even better robustness, indicating its locators remain both maintainable and effective. Compared with similarity-based methods, EAGL is weaker in long-term robustness but compensates with higher precision locators. On the Similo replication dataset, EAGL further surpasses ROBULA+ while narrowing the robustness gap to similarity-based approaches, offering higher precision (0.98, 0.96, 0.81) than VON Similo and Similo but lower recall (0.73, 0.92, 0.87), and a balanced F1 score (0.84, 0.95, 0.84) across the three methods EAGL, VON Similo at threshold 0.40 and Similo at threshold 0.28 respectively.

5.4 RQ4. Generalisability

Table 7 reports the manual categorisation results of 181 websites based on each website's core functionality—what it is primarily used for—as the main criterion. Websites with ambiguous purposes or those that appear only once are grouped under C17 (Other). For the generalisability study, we focus on categories with at least ten websites as our target, since cross-project evaluation requires sufficient diversity within the same category. This results in five categories: C1 to C5, which include Entertainment, News, Retail, Social Media, and Web Services (highlighted in **bold**).

Table 8 reports the performance of cross-project stability prediction models, where stability knowledge learned from websites in the same functional category is evaluated on unseen websites within that category. Compared to the within-project setting in RQ1 (T(181): precision 0.872, recall 0.845, F1 0.802, ROC-AUC 0.714, MAP 0.832), cross-project models show an overall decrease in both classification and ranking metrics. Notably, the mean MAP drops from 0.832 to 0.485, while the median is 0.774; this indicates that a few very low-performing categories significantly reduce the mean, while at least half of the categories achieve MAP values above 0.774. In contrast, recall shows the opposite trend: its median decreases from 0.745 to 0.485, but the mean remains at 0.665, implying

Table 7. Categories of study subjects and the Number of Websites in each Category. We mark the five largest categories in bold.

Category	Count	Category	Count	Category	Count
C1: News	**25**	C7: Advertising	4	C13: Finance	6
C2: Retail	**12**	C8: Information	5	C14: Logistics	5
C3: Social Media/Blogging	**14**	C9: E-commerce	7	C15: Business/Banking	6
C4: Web services	**11**	C10: Search	5	C16: Employment	6
C5: Entertainment	**13**	C11: Technology/Security	9	C17: Other	37
C6: Software as a Service	7	C12: Media	9		

that a few high-performing categories raise the average, although most categories have recall values below 0.665. Among the five largest categories (C1 to C5) that we studied, C1 (the largest) retains the most stable performance across both classification and ranking metrics. Although all metrics decline compared to the within-project setting, the decrease is not uniformly severe, suggesting that stability knowledge learned within the same functional category can still generalise to other websites in the category to some extent.

Table 8. Generalisability. Evaluation of cross-project stability prediction models, with mean and median values reported for Precision, Recall, F1, ROC-AUC, and MAP.

	Classification								Ranking	
	Precision		Recall		F1		ROC-AUC		MAP	
Category	mean	median	mean	median	mean	median	mean	median	mean	median
C1	0.71	0.68	0.694	0.507	0.815	0.71	0.68	0.694	0.507	0.815
C2	0.611	0.614	0.612	0.501	0.726	0.611	0.614	0.612	0.501	0.726
C3	0.638	0.642	0.64	0.434	0.791	0.638	0.642	0.64	0.434	0.791
C4	0.831	0.82	0.826	0.481	0.908	0.831	0.82	0.826	0.481	0.908
C5	0.551	0.6	0.553	0.503	0.632	0.551	0.6	0.553	0.503	0.632
T(5)	0.668	0.671	0.665	0.485	0.774	0.668	0.671	0.665	0.485	0.774

In the end, what matters most for EAGL is the effectiveness of the generated locators. Table 9 reports the effectiveness of locators generated by EAGL under a cross-project setting. Overall, the trends observed in the within-project evaluation (Table 4) remain consistent. For $ProxyPrec$, EAGL continues to outperform both Selenium (SEL) and ROBULA+ (R+), achieving 0.968 compared to 0.864 and 0.844, respectively. A similar pattern holds for robustness metrics: across all categories (C1–C5), EAGL achieves the highest win counts (W)—for Selenium and ROBULA+, loss counts (L)—along with greater RDoS values and average DoS, reaffirming its resilience to DOM changes observed in the within-project results. Locator length also follows the same ordering, with ROBULA+ producing the shortest locators, followed by EAGL and then Selenium. These consistent trends suggest that EAGL can generalise effectively across categories without significant loss of locator quality.

Table 9. Locator Generalisability across website categories (C1–C5). "Catg" stands for Category, the results of top five largest website categories (C1 to C5) are reported. For metric definitions and notation (e.g., nr_{ver}, RDoS, W/L/T, A/B), refer to the caption of Table 4.

		ProxyPrec			Robustness									Locator Length		
					EAGL vs Sel			EAGL vs R+			DoS					
Catg	nr_{ver}	EAGL	SEL	R+	W (RDoS)	L (RDoS)	T	W (RDoS)	L (RDoS)	T	EAGL	SEL	R+	EAGL	SEL	R+
C1	181/199	**0.976**	0.877	0.796	**123 (79)**	101 (64)	196	**78 (95)**	37 (54)	305	**119**	110	107	4.586	7.279	**4.014**
C2	188/199	**0.955**	0.922	0.892	**56 (80)**	37 (36)	127	**18 (51)**	17 (36)	184	**99**	85	96	4.234	8.706	**3.001**
C3	176/199	**0.949**	0.918	0.867	**35 (70)**	19 (71)	166	**22 (73)**	13 (79)	185	108	102	**111**	3.918	7.762	**3.510**
C4	128/199	**0.985**	0.858	0.784	**42 (35)**	21 (44)	157	16 (49)	**22 (75)**	182	93	94	**98**	4.691	7.767	**2.988**
C5	185/199	**0.975**	0.747	0.879	**60 (58)**	35 (21)	265	**69 (89)**	19 (11)	272	**105**	85	91	4.931	7.841	**4.204**
T(5)	171/199	**0.968**	0.864	0.844	**32/316(64)**	21/213(47)	911	**41/203(71)**	14/108(51)	1,128	**104**	95	100	4.472	7.871	**3.543**

Answer to RQ4: The cross-project evaluation shows that while the classification and ranking performance of the stability prediction model declines compared to the within-project setting the median MAP (0.774) and other metrics indicate that stability knowledge can still generalise to unseen websites within the same functional category. Among the top five categories, C1 maintains relatively stable performance, suggesting that functional similarity supports partial transfer of stability patterns. Moreover, EAGL continues to outperform Selenium and ROBULA+ in locator effectiveness as observed in the within-project setting. These results confirm that EAGL generalises well across categories without significant loss of locator effectiveness.

5.5 RQ5.Efficiency

Table 10 compares the efficiency of EAGL with Selenium and ROBULA+. Unlike the baselines, EAGL incurs an initial model training cost, averaging 1165.791 seconds (approximately 19 minutes), as shown in the second column. However, results from the sensitivity and generalisability analyses in RQ3 and RQ4 show that reusing older models does not significantly impact stability prediction or locator generation, making this largely a one-time cost. Furthermore, model training can be performed in the background while EAGL generates locators using an existing model.

For locator generation time, Selenium is the fastest, returning a locator almost instantaneously. EAGL follows, with an average time of 0.221 seconds, including both the prediction and generation time. ROBULA+ is much slower, averaging over 1.8 minutes (110.108 seconds), likely due to the exhaustive comparison of locator combinations during the generation process. Considering the superior locator robustness demonstrated in RQ3 and RQ4, we argue that EAGL's average locator generation time–under 0.3 seconds–offers acceptable trade-off between effectiveness and efficiency. The generated locators remain robust over time, making them more sustainable and cost-effective in long-term testing workflows.

Answer to RQ5: EAGL incurs an initial one-time training cost but generates locators in under 0.3 seconds on average, offering a practical balance between efficiency and robustness. Compared to baselines, EAGL provides significantly more robust locators with an acceptable overhead for real-world testing scenarios.

Table 10. Efficiency. Initial Cost is reported for EAGL only. $Dom_{\#\text{ positions}}$ and Dom_{height} refer to the average number of DOM positions (i.e., the total number of element and attribute nodes in the DOM) and the maximum height of a position in the DOM, respectively. For EAGL, the locator generation time is further separated into the total stability prediction time for all DOM positions (**Pred**) and the total time taken to generate a locator with the stability values (**Gen**). All times are averaged across each website version and reported in seconds.

	Initial Cost(seconds)	Locator Generation(seconds)					DOM complexity	
Site	Model Training (EAGL)	EAGL (Pred, Gen, Total)			Selenium	Robula+	$Dom_{\#\ positions}$	Dom_{height}
S1	9560.634	0.161	0.719	0.880	**0.000**	119.783	34589.0	29
S2	4698.586	0.142	0.132	0.274	**0.000**	40.588	26006.0	31
S3	3956.808	0.155	0.925	1.080	**0.000**	290.29	23542.0	24
S4	6309.767	0.314	0.390	0.704	**0.000**	55.781	22341.0	68
S5	2830.052	0.237	0.305	0.542	**0.000**	96.021	18588.0	22
S6	4025.117	0.099	0.341	0.440	**0.000**	167.048	18539.0	31
S7	3615.939	0.077	0.239	0.316	**0.000**	174.394	16322.0	27
S8	6125.395	0.112	0.242	0.354	**0.000**	154.62	16301.0	20
S9	4575.244	0.121	0.344	0.465	**0.000**	88.224	16648.0	30
S10	3973.988	0.092	0.184	0.276	**0.000**	65.177	16130.0	16
O(171)	889.838	0.088	0.103	0.191	**0.000**	103.153	5071.678	23
T(181)	1165.791	0.097	0.124	0.221	**0.000**	110.108	6232.822	25

6 DISCUSSION

6.1 End-to-End Test Cases

End-to-end (E2E) tests validate the complete workflow of a web application by simulating real user interactions and ensuring that all components function correctly. Such tests often involve 'reserved areas' of websites(i.e. restricted sections requiring authentication), such as login pages, dashboards, or administrative panels. Although EAGL is expected to work consistently for both public and reserved areas due to their structural similarities, this study focuses on publicly accessible data. Future work will extend our evaluation to include reserved areas.

6.2 Dynamic content

Dynamic elements are inherently volatile, making them less suitable as indicators of long-term stability. Our model, therefore, focuses on stable elements consistently present across versions, such as static layout components, navigation bars, or main content areas. Nonetheless, while dynamic content was excluded from our training set for this reason, its partial omission may affect dataset representativeness. Future work will explore methods to capture dynamic content more systematically or assess the implications of its exclusion, potentially incorporating tools that simulate dynamic behaviour to improve dataset diversity.

6.3 From ID-Anchored Supervision to Broad DOM Stability Prediction

Although the stability prediction model of EAGL is trained only on DOM positions resolving to element nodes and `id` attribute nodes (Sections 4.3.2 and 4.4.2), it is applied to all DOM positions, including those resolving to other attribute types. The model relies exclusively on structural features—such as tree depth, sibling count, and positional index—without using semantic cues like tag names or attribute types. This structural emphasis enables the model to capture generalisable patterns of persistence that are not tied to specific element or attribute types. In particular, DOM positions resolving to non-`id` attributes may exhibit structural traits similar to those of `id` attributes,

such as appearing early among a small number of sibling attributes or residing at shallow tree depths. The model can extend its structural understanding to these cases, despite their absence during training.

The *node position* feature used for stability prediction is correlated with `id` attributes (which often appear first), introducing some inductive bias. However, this bias is acceptable and even desirable to some extent, as it helps identify structurally stable anchors while avoiding volatile attributes (e.g., `class` or `style`). Thus, although trained on a subset, the model can generalise to broader cases by leveraging structural indicators of persistence, consistent with EAGL's position-centric design.

6.4 Proxy Ground Truth Reliability

In Section 4.3.2, we introduced two types of ground truth. To build the second ground truth (i.e., locator breakage), a crucial challenge lies in matching the element located in one version of an application to the corresponding element in the next version. While manual validation is ideal and provides the highest possible accuracy, it limits the ability to evaluate the locators over short spans of changes. To address this, we introduced a scalable approximation for locator matching, in which we state that if an element is located by a crafted locator, then it is likely the same element in both versions.

While this approximation does not hold in all cases, we hypothesise that it is valid in the majority of cases. To confirm this, we sampled five websites from our dataset, one from each category, and inspected the locators generated by ROBULA+, Selenium, and EAGL across all versions of the sampled websites. The results show that, out of the 7,523 validated cases, the hypothesis did not hold in only 0.37% of the cases. Specifically, we observed 28 cases in which a locator matched exactly one element in each version, yet the matched elements corresponded to different elements across versions. Section 6.4.1 discusses the detailed inspection.

To evaluate 7,523 cases, we employed an automation-assisted inspection process. We developed a lightweight tool that displays two web pages side by side to support visual inspection. The validation procedure consists of the following steps:

- **Step 1:** The tool displays a list of available versions of a website to the inspector.
- **Step 2:** The inspector selects a version to be inspected. The tool then automatically displays the selected version together with its immediately preceding version, along with the locators generated by ROBULA+, Selenium, and EAGL for the corresponding elements.
- **Step 3:** The tool highlights the first locator under inspection and places visual rectangles around the elements located by the locator in both displayed versions.
- **Step 4:** The inspector visually examines the highlighted elements in both versions and, using dedicated interface controls, determines whether the element located in the inspected version corresponds to the element in the previous version.
- **Step 5:** Once the decision is recorded, the tool proceeds to the next locator and repeats the process until all locators are inspected.
- **Step 6:** After all locators for the selected version have been inspected, the inspector returns to the list of versions in **Step 1** to select the next version.

The recorded information includes the timestamp of the version under inspection, the locator under evaluation, the locator generation algorithm used, the number of elements matched in **version A** and **version B**, and the final decision of the inspector (correct or incorrect match). A **correct match** is recorded when *locator X*, which targets *element Y* in **version A**, also matches *element Y* in **version B**. Conversely, an **incorrect match** is recorded when *element Y* is absent from the set of elements matched by *locator X* in **version B**.

Based on our inspection, we observed cases in which a locator matched a single element in **version A** and multiple elements in **version B**. According to our criteria, if one of the returned elements corresponds to the correct target, the locator is considered successful. We did not observe any case in which the correct element was absent among the matched elements in such multiple-match cases. This behaviour arises when a locator depends on nodes that are unique in **version A** but, as the website evolves, match additional elements in **version B**. As a result, unintended elements may be matched, leading to false positives; however, the correct element is still included in the matched set. The manual evaluation data for each website is provided in our online GitHub repository [13].

6.4.1 Reliability of Locator Breakage Ground Truth. The locator breakage ground truth used in this study relies on a scalable approximation that distinguishes between complete breakage (no elements returned), partial breakage (multiple elements returned), and correct resolution (exactly one element returned). To assess the reliability of this approximation, we performed targeted manual inspection focusing on locator resolutions. Specifically, we manually examined the resolution of the generated locators between different versions of the DOM, investigating whether the located element remained consistent over time.

The inspection revealed that mismatches in these cases—where a locator returned a single element but referred to a different underlying element across versions—were rare. Table 11 shows the manual inspection results. Out of 7314 cases, 13, 12, and 3 cases have been identified as the cases where the single element located refers to different elements between versions, respectively for ROBULA+, Selenium, and EAGL. This observation provides empirical support for treating single-element resolutions as unbroken under the adopted proxy.

We also validate cases where a locator returns multiple elements. To analyse these cases, if the locator returns the target element among the returned set of elements, then the locator is correct but lacks preciseness. This occurs when a locator at the time of generation returns only the target element, but returns multiple after evolution. Table 11 shows the results of the multi-element inspection. Out of 60 cases, 0, 24, and 4 cases have been identified where the target element was not included in the set of elements returned by the locator, respectively for ROBULA+, Selenium, and EAGL. These results indicate that the majority of cases where a locator starts returning multiple elements, it loses reliability.

Additionally, our inspection included the cases where locators were breaking, and we observe that out of the 7523 cases, 51, 38 and 32 cases the locators were completely broken, and based on this random inspection, EAGL exhibits a similar robustness advantage over ROBULA+ and Selenium.

Given the scale of the studied systems and the known difficulty of precise cross-version element matching, this conservative treatment allows large-scale evaluation while minimising the risk of overstating locator correctness. Consequently, reported results should be interpreted as reflecting robustness and specificity under the adopted proxy, rather than exact semantic correctness in all cases.

7 THREATS TO VALIDITY

In this research paper, we discuss the validity of several factors, including internal validity, external validity, and construct validity.

Internal validity: An important internal threat stems from the data collection process. Ideally, one would examine each snapshot of the website; however, due to the rapid pace of changes—often

[13] https://github.com/hilalosoft/EAGL

Table 11. Assisted inspection results of locator resolutions across tools for single-element and multi-element mismatching

Category	Tool	Total Cases	Broken (1.6%) (*numLocated*=0)	Single-Element (*numLocated*=1)(97.6%)			Multi-Element (*numLocated*>1)(0.8%)		
				Inspected	Mismatch	%	Inspected	Not Included	%
C1	R+	143	18	125	3	2.10	0	0	0.00
	Sel	469	7	453	0	0.00	9	0	0.00
	EAGL	469	11	458	1	0.21	0	0	0.00
C2	R+	438	7	431	6	1.37	0	0	0.00
	Sel	544	5	539	6	1.10	0	0	0.00
	EAGL	544	2	538	0	0.00	4	4	0.74
C3	R+	119	2	115	0	0.00	0	0	0.00
	Sel	119	1	116	0	0.00	0	0	0.00
	EAGL	119	0	117	0	0.00	0	0	0.00
C4	R+	785	0	785	0	0.00	0	0	0.00
	Sel	785	0	785	0	0.00	0	0	0.00
	EAGL	785	0	785	0	0.00	0	0	0.00
C5	R+	526	19	507	4	0.76	0	0	0.00
	Sel	839	25	767	6	0.72	47	24	2.86
	EAGL	839	19	820	2	0.24	0	0	0.00
Totals	R+	2011	51	1963	13	0.65	0	0	0.00
	Sel	2756	38	2660	12	0.44	56	24	0.87
	EAGL	2756	32	2717	3	0.11	4	4	0.15
Overall Total		7523	121	7314	28	0.37	60	28	0.37

occurring within minutes—capturing meaningful alterations within that time-frame proved impractical. To address this concern, we implemented a one-day interval between versions, which may introduce a degree of bias. To mitigate this potential bias, we randomly selected a version from each day without any manual inspection.

Additionally, as noted in Section 4.2, both Selenium IDE and ROBULA+ (i.e., our baselines) were adapted to integrate with our evaluation infrastructure, as their original implementations were either not scriptable or incompatible with our infrastructure. We closely followed the documented functionalities and implementations to minimise the discrepancies–which can affect the comparability– introduced by these re-implementations.

External validity: To ensure the generalisability of our findings, we constructed a large-scale dataset from 181 popular websites across diverse categories, ensuring broad coverage of real-world DOM structures and changes. Nonetheless, this selection may still introduce bias, as these websites tend to follow better engineering practices. Consequently, the results may differ for less popular or domain-specific websites with more irregular structures.

Construct validity: We assess locator correctness by checking whether generated locators can resolve to unique elements in updated DOMs. While this synthetic approach ensures scalability, it may miss semantic mismatches, for instance, resolving to structurally similar–at the similar positions– but semantically different elements. Another limitation stems from relying on static snapshots from the Wayback Machine for the past DOM collection, excluding dynamic contents. This could affect the generalisability of our findings, even though our main interest lies on structurally stable positions (e.g., layout or navigation nodes), which aligns with the static nature of the collected data. We plan to incorporate dynamic elements in future work through a collection process that captures both static and dynamic content of websites.

8 RELATED WORK

There are several works addressing problems in E2E tests in web applications. According to Ricca et al [37], these problems can be categorised into three categories: 1) Fragility of tests, 2) Strong coupling and low cohesion, and 3) Incompleteness. Our work addresses test fragility, a topic also explored in several other studies, which we will go over in this section.

Some works address the repair of GUI test scripts [13, 30]. In Sitar [13], they map text scripts using an event-flow graph (EFG) used for the repair. The automatic repair of test scripts using EFG is done with human input. It is acknowledged that the knowledge a tester brings is essential to repair tests, and it cannot be validated without it. Chen et al. [11] proposed a "search-based test suite repair" technique that combines test case generation and repair techniques to automatically fix failing test cases in a DOM-based web test suite. The approach uses a genetic algorithm to search for repair actions to fix the test cases. Moreover, research has been done to target locator repairs, specifically in web tests. Web visual repair [43] uses the visual representation of elements to avoid the mis-selection issue based on locator info. The approach visually records the steps of the tests and the interacted elements to repair a test. It matches the old element representation with the new one and adjusts the interaction accordingly. This approach has two weaknesses: 1- It can easily miss non-visually represented elements, 2- It increases the time of the test suite execution by 3.7 times because it uses computer vision. COLOR [20] addresses the issue in a simple but effective approach by saving different attributes from the previous version of the DOM and, upon breakage, replacing the broken locator with a nonchanged attribute. The main challenge with locator repair techniques is that the repaired locator might not correspond to the same element from the previous DOM. In EAGL, we do not work on the repair but on the robustness of tests, more specifically, the locator's robustness.

Another field of research addressing fragility is the robust generation of web tests [8, 31, 44]. In a recent work on this topic, Chang et al. [10] implemented reinforcement learning to generate test cases for web applications based on state abstraction. EAGL focuses explicitly on the generation of locators, which are used in the process of writing or generating tests. Furthermore, in web applications, the prominent factor of test breakages is locator breakages. To address this, using visual clues, Yandrapally et al. [46] search for anchors that can uniquely identify the target element. However, they often overlook the importance of the stability of these anchors, prioritising uniqueness over stability. While a unique node can serve as an anchor, not all anchors are stable. EAGL effectively combines both the uniqueness and stability of the positions that can serve as anchors to create reliable locators. ROBULA [24] uses heuristic approaches to improve the locator as it gets closer to the element, starting from the root. At the same time, ROBULA+ [26] adds prioritisation and blacklisting of attributes to the previous algorithm to improve its performance. The drawback of these techniques is that developers may be prone to bias. In our study, we observed that some projects use sets of attributes that differ significantly from others. Our approach adapts to different websites by utilising trained models that are specifically tailored for each site. Similo [33] ranks the elements compared to the target element based on a score calculated through the attributes, and the correct element is then selected based on the match with the highest score. Lastly, De Luca et al. [28] did an investigation on the topic of robust locator generators in web applications. They inject *hooks* (i.e. nodes) that are strategically placed to help identify elements. EAGL uses machine-learning capabilities to discover relationships between elements on a web page and provides the most reliable nodes to locate our target element. A multi-locator solution was also proposed [25], focusing on generating multiple locators for a target element. It increases robustness by expecting a portion of the locators to survive the evolution and, as such, replace a

broken locator. However, this solution cannot be used in the testing tools, and EAGL is a single locator generator.

The evolution of web applications has been studied from the perspective of logical changes. Like the analysis of the semantic structure of the web page [40], the authors identify elements that are relevant to the semantics of the website and produce locators that depend on these semantics to be less likely to change. This means that websites with frequent changes in semantic structure are more prone to breakages. So, some approaches were proposed by learning from the logical side of the website, Choudhary et al. introduced *Water*, an automated web test repair approach that can suggest repairs for broken tests caused by structural changes by broken locators [12], using the property information from an old DOM and searching them in a new DOM to obtain new locators for the found elements. Nguyen et al. [35] also focus on the semantic structure of the neighbouring nodes to construct an XPath to the target element. The latest contribution that uses the project's history is SIDEREAL [23]. It leverages an adaptive approach of assigning weights to nodes, and it treats the problem as a graph exploration problem while only targeting nodes that are part of a locator. EAGL aims at learning from the evolution of the DOM and estimating the likelihood of change of nodes for all nodes in the DOM. Unlike the logical changes of web applications, there is not much work on the structural evolution. This is what we address in our work EAGL.

9 CONCLUSION AND FUTURE WORK

Web application tests often suffer from fragility due to unstable locators, leading to frequent breakages and increased maintenance costs. In this study, we introduced EAGL, an approach that predicts the stability of DOM positions—resolving to either element or attribute nodes—and uses these stability signals to construct robust locators. By analysing historical versions from 181 real-world websites, EAGL identifies structurally persistent positions and effectively exploits them to build precise and resilient locators, improving the reliability and maintainability of web testing. Our contributions are threefold: (1) a large-scale dataset capturing DOM position changes across 181 websites, (2) a stability prediction model that leverages structural features of the DOM to estimate position stability, and (3) a locator generation tool that uses these stability predictions to generate locators robust to web evolution. Through extensive evaluation on 181 websites, we demonstrated that EAGL outperforms Selenium and ROBULA+ in overall locator effectiveness—both precision and robustness—while also generalising well across different websites within the same categories and maintaining high efficiency. Future work will explore extending stability prediction for automated test repair, generation, and bug detection by tracking changes in highly stable positions.

ACKNOWLEDGMENTS

This work is supported by the Luxembourg National Research Funds (FNR) through the CORE project C23/IS/18182513/MiCE and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2021-NR060080).

REFERENCES

[1] [n. d.]. *Companies using Selenium.* https://enlyft.com/tech/products/selenium

[2] 2020. *XPath vs CSS Selector: The Difference and How to Choose.* https://www.testim.io/blog/xpath-vs-css-selector-difference-choose/

[3] 2023. *Windows Subsystem for Linux.* https://learn.microsoft.com/en-us/windows/wsl/about

[4] 2023. *XPath vs CSS Selectors: A Detailed Guide.* https://www.lambdatest.com/blog/xpath-vs-css-selectors/

[5] 2024. *Robocorp Automation Studio User Guide.* https://robocorp.com/docs-robot-framework

[6] Arora A and Sinha M. 2012. Web Application Testing: A Review on Techniques, Tools and State of Art. *International Journal of Scientific & Engineering Research* 3 (2012). Issue 2. http://www.ijser.org

[7] Tobias Anton. 2005. XPath-Wrapper Induction by generating tree traversal patterns. In *LWA*. https://api.semanticscholar.org/CorpusID:9336145

[8] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. 2008. Finding bugs in dynamic web applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08)*. Association for Computing Machinery, New York, NY, USA, 261–272. https://doi.org/10.1145/1390630.1390662

[9] Sacha Brisset, Romain Rouvoy, Lionel Seinturier, and Renaud Pawlak. 2022. Erratum: Leveraging Flexible Tree Matching to Repair Broken Locators in Web Automation Scripts. *Inf. Softw. Technol.* 144, C (apr 2022), 16 pages. https://doi.org/10.1016/j.infsof.2021.106754

[10] Xiaoning Chang, Zheheng Liang, Yifei Zhang, Lei Cui, Zhenyue Long, Guoquan Wu, Yu Gao, Wei Chen, Jun Wei, and Tao Huang. 2023. A Reinforcement Learning Approach to Generating Test Cases for Web Applications.

[11] Wei Chen, Hanyang Cao, and Xavier Blanc. 2021. An Improving Approach for DOM-Based Web Test Suite Repair. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 12706 LNCS (2021), 372–387. https://doi.org/10.1007/978-3-030-74296-6_29

[12] Shauvik Roy Choudhary, Dan Zhao, Husayn Versee, and Alessandro Orso. 2011. WATER: Web Application TEst Repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering* (Toronto, Ontario, Canada) *(ETSE '11)*. Association for Computing Machinery, New York, NY, USA, 24–29. https://doi.org/10.1145/2002931.2002935

[13] Zebao Gao, Zhenyu Chen, Yunxiao Zou, and Atif M. Memon. 2016. SITAR: GUI Test Script Repair. *IEEE Transactions on Software Engineering* 42 (2 2016), 170–186. Issue 2. https://doi.org/10.1109/TSE.2015.2454510

[14] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. 2016. WATERFALL: An Incremental Approach for Repairing Record-Replay Tests of Web Applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 751–762. https://doi.org/10.1145/2950290.2950294

[15] Javaria Imtiaz, Muhammad Zohaib Iqbal, and Muhammad Uzair khan. 2021. An automated model-based approach to repair test suites of evolving web applications. *Journal of Systems and Software* 171 (2021), 110841. https://doi.org/10.1016/j.jss.2020.110841

[16] Javaria Imtiaz, Salman Sherin, Muhammad Uzair Khan, and Muhammad Zohaib Iqbal. 2019. A systematic literature review of test breakage prevention and repair techniques. *Information and Software Technology* 113 (2019), 1–19. https://doi.org/10.1016/j.infsof.2019.05.001

[17] Javaria Imtiaz, Salman Sherin, Muhammad Uzair Khan, and Muhammad Zohaib Iqbal. 2019. A systematic literature review of test breakage prevention and repair techniques. *Information and Software Technology* 113 (9 2019), 1–19. https://doi.org/10.1016/j.infsof.2019.05.001

[18] Internet Archive. 2025. Wayback Machine. https://web.archive.org/. Accessed: 2025-06-26.

[19] Y. Kamei and E. Shihab. 2016. Defect Prediction: Accomplishments and Future Challenges. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 5. 33–45.

[20] Hiroyuki Kirinuki, Haruto Tanno, and Katsuyuki Natsukawa. 2019. Recommending correct locator for broken test scripts using various clues in web application. *Computer Software* 36 (2019), 3–17. Issue 4. https://doi.org/10.11309/jssst.36.4_3

[21] Shakti Kundu. 2012. Web Testing: Tool, Challenges and Methods.

[22] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2016. Chapter Five - Approaches and Tools for Automated End-to-End Web Testing. Advances in Computers, Vol. 101. Elsevier, 193–237. https://doi.org/10.1016/bs.adcom.2015.11.007

[23] Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2021. Sidereal: Statistical adaptive generation of robust locators for web testing. *Software Testing, Verification and Reliability* 31 (5 2021), e1767. Issue 3. https://doi.org/10.1002/STVR.1767

[24] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2014. Reducing Web Test Cases Aging by Means of Robust XPath Locators. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. 449–454. https://doi.org/10.1109/ISSREW.2014.17

[25] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2015. Using multi-locators to increase the robustness of web test cases. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 -*

Proceedings. https://doi.org/10.1109/ICST.2015.7102611

[26] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2016. Robula+: An Algorithm for Generating Robust XPath Locators for Web Testing. *J. Softw. Evol. Process* 28, 3 (mar 2016), 177–204. https://doi.org/10.1002/smr.1771

[27] Yuan Fang Li, Paramjit K. Das, and David L. Dowe. 2014. Two decades of Web application testing—A survey of recent advances. *Information Systems* 43 (7 2014), 20–54. https://doi.org/10.1016/J.IS.2014.02.001

[28] Marco De Luca, Anna Rita Fasolino, and Porfirio Tramontana. 2024. Investigating the robustness of locators in template-based Web application testing using a GUI change classification model. *Journal of Systems and Software* 210 (4 2024), 111932. https://doi.org/10.1016/j.jss.2023.111932

[29] S. McIntosh and Y. Kamei. 2018. Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction. *IEEE Transactions on Software Engineering* 44, 5 (May 2018), 412–428. https://doi.org/10.1109/TSE.2017.2693980

[30] Atif M. Memon. 2008. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Transactions on Software Engineering and Methodology* 18 (11 2008). Issue 2. https://doi.org/10.1145/1416563.1416564

[31] Ali Mesbah, Arie Deursen, and Danny Roest. 2012. Invariant-Based Automatic Testing of Modern Web Applications. *Software Engineering, IEEE Transactions on* 38 (01 2012), 1 – 1. https://doi.org/10.1109/TSE.2011.28

[32] Jussi Myllymaki and Jared Jackson. 2002. Robust Web Data Extraction with XML Path Expressions. https://api.semanticscholar.org/CorpusID:10997495

[33] Michel Nass, Emil Alégroth, Robert Feldt, and Riccardo Coppola. 2023. Robust web element identification for evolving applications by considering visual overlaps. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 258–268. https://doi.org/10.1109/ICST57152.2023.00032

[34] Michel Nass, Emil Alégroth, Robert Feldt, Maurizio Leotta, and Filippo Ricca. 2023. Similarity-Based Web Element Localization for Robust Test Automation. *ACM Trans. Softw. Eng. Methodol.* 32 (4 2023). Issue 3. https://doi.org/10.1145/3571855

[35] Vu Nguyen, Thanh To, and Gia Han Diep. 2021. Generating and selecting resilient and maintainable locators for Web automated testing. *Software Testing Verification and Reliability*. https://doi.org/10.1002/stvr.1760

[36] Heidar Pirzadeh, Sara Shanian, and Farzin Davari. 2014. A novel framework for creating user interface level tests resistant to refactoring of web applications. *Proceedings - 2014 9th International Conference on the Quality of Information and Communications Technology, QUATIC 2014* (12 2014), 268–273. https://doi.org/10.1109/QUATIC.2014.43

[37] Filippo Ricca, Maurizio Leotta, and Andrea Stocco. 2018. *Three Open Problems in the Context of E2E Web Testing and a Vision: NEONATE*. https://doi.org/10.1016/bs.adcom.2018.10.005

[38] Renaud Rwemalika, Sarra Habchi, Mike Papadakis, Yves Le Traon, and Marie-Claude Brasseur. 2022. Smells in system user interactive tests. *Empirical Software Engineering* 28 (2022), 20. Issue 1. https://doi.org/10.1007/s10664-022-10251-1

[39] Renaud Rwemalika, Marinos Kintis, Mike Papadakis, Yves Le Traon, and Pierre Lorrach. 2019. On the Evolution of Keyword-Driven Test Suites. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 335–345. https://doi.org/10.1109/ICST.2019.00040

[40] Fei Shao, Rui Xu, Wasif Haque, Jingwei Xu, Ying Zhang, Wei Yang, Yanfang Ye, and Xusheng Xiao. 2021. WebEvo: taming web application evolution via detecting semantic structure changes. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 16–28. https://doi.org/10.1145/3460319.3464800

[41] John E. Simpson. 2002. *XPath and XPointer: Locating Content in XML Documents*. O'Reilly & Associates, Inc., USA.

[42] Jeongju Sohn and Shin Yoo. 2017. FLUCCS: Using Code and Change Metrics to Improve Fault Localisation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2017)*. 273–283.

[43] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. Visual web test repair. *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 503–514. https://doi.org/10.1145/3236024.3236063

[44] Suresh Thummalapenta, K. Vasanta Lakshmi, Saurabh Sinha, Nishant Sinha, and Satish Chandra. 2013. Guided test generation for web applications. In *2013 35th International Conference on Software Engineering (ICSE)*. 162–171. https://doi.org/10.1109/ICSE.2013.6606562

[45] M. Wen, J. Chen, Y. Tian, R. Wu, D. Hao, S. Han, and S. C. Cheung. 2019. Historical Spectrum based Fault Localization. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2948158

[46] Rahulkrishna Yandrapally, Suresh Thummalapenta, Saurabh Sinha, and Satish Chandra. 2014. Robust Test Automation Using Contextual Clues. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 304–314. https://doi.org/10.1145/2610384.2610390

[47] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen. 2008. Mining Software Repositories to Study Co-Evolution of Production & Test Code. In *2008 1st International Conference on Software Testing, Verification, and Validation*. 220–229. https://doi.org/10.1109/ICST.2008.47