DOM Instrumentation to Display Provenance Data

Jayen Gulab

***Abstract*—With ever-evolving privacy laws, end users of software systems are increasingly expressing concerns about the usage of their data. Real-world incidents, such as the 2019 data breach affecting up to 112,000 Air New Zealand Airpoints customers, have contributed to a growing awareness of data privacy issues. In response to these concerns, New Zealand regulators re-evaluated the privacy act in 2020, imposing financial liability on data providers to mitigate potential data breaches. This project aimed to contribute to an infrastructure enabling users to monitor the usage of their data when interacting with web applications. This has been accomplished by providing end users with increased transparency regarding the handling of their data during the browsing of web applications. Modern web applications have complex layered architectures, often involving server-side applications with existing solutions. Developing and retrofitting systems to support this transparency is both intricate and costly; hence, automation is the desirable approach. This research explored the implementation of a solution to expose existing provenance data from the server-side domain to the client-side domain through manipulation of the client-side Document Object Model (DOM). To achieve this objective, a range of prototypes to instrument the client-side DOM and expose existing provenance data were developed. The prototypes were browser plugins, pure JavaScript instrumentations, and framework plugins. Throughout the project, each decision was carefully evaluated before initiating implementation. Performance was assessed by measuring performance overheads to help determine whether the performance cost is worth the functionality.**

Keywords - Automation, Client-Side DOM Instrumentation, Data Transparency, Development, End Users, JavaScript, Performance Analysis, Privacy, Provenance Data, Web Applications

I. Introduction

A. Background

The General Data Protection Regulation (GDPR) represents a set of EU-based data protection regulations [1]. One of the rules within this regulatory framework stipulated an individual's entitlement to request the deletion of their data. As per the GDPR's definition, personal data encompassed "any information relating to an identified or identifiable natural person" [2]. It is imperative for companies to maintain compliance with data protection regulations and handle data with utmost care. A notable instance of mishandling data occurred in April 2023 when Meta Platforms Ireland Limited was fined 1.2 billion euros by the Irish Data Protection Agency, marking the largest GDPR fine issued to date [3]. The penalty was a consequence of Meta's failure to comply with GDPR standards concerning data transfers. To mitigate such occurrences, numerous companies deliberately obscure the origin and destination of data. This obfuscation can result in a lack of transparency, making it challenging to fully understand how your data is being utilised.

B. Motivation

The primary objective of this project was to cater to end-users who want to gain insights into what information server-side applications utilise when data is inputted into a web application or when retrieving it from a server application. By shedding light on obscured data, this project aimed to notify users when their data is being utilised for purposes that differ from the original intent. As an example, one scenario could be where you send an image to a server, and that image is covertly utilised for training an AI model. Users would become aware of this through the provenance data that is now made visible to them. Examples of provenance data include information about the APIs being utilised when processing your data, and their storage location. Contemporary web applications are layered and have existing solutions for server-side applications. Given this ongoing development it became crucial to establish solutions that could be integrated into existing applications without the need for extensive rewriting. The scope of this project did not encompass rendering data directly to the presentation layer which is often seen in real-world application scenarios by using CSS. Rather, creating a simple rendering solution using a popup which can be dynamically displayed on web applications was the approach taken. This was pivotal in determining the credibility and integrity of incoming and outgoing data based on its source [4]. The main assumption made during this project was that the server would generate this provenance data, and it could be extracted by analysing header information. The proposed approach involved DOM manipulation and instrumentation. A similar concept of provenance is employed by companies like Facebook and Google for their advertisements. These companies use data to display personalised ads and include a "Why am I seeing this ad?" link where the user could access provenance information by clicking the provided link [5] as seen in Figure 1.

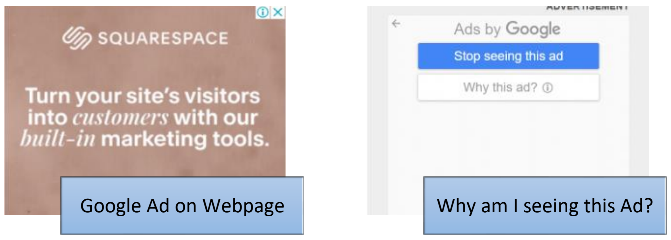


Figure 1. Example of Provenance Data

Ad embedded on webpage. Trying to close the ad will give the option to see why you are seeing this add.

C. Approaches

In pursuit of solving this objective, three distinct approaches were explored. These included the development of browser plugins, the utilisation of pure JavaScript instrumentations, and the incorporation of Framework plugins. Each of these approaches presented its unique set of challenges and intricacies in design, with some being impractical due to their specific limitations.

D. Environmental Concerns

Concerning its environmental footprint, this project did not need hardware components, resulting in negligible resource consumption during the development phase. The sole potential environmental consideration revolved around the additional power consumption associated with running the software instrumentation. To address this issue, a deliberate effort was made to ensure that all implementations imposed minimal overhead. The Chrome Performance Profiler was employed as my solution for this task. While this tool does not offer direct power profiling capabilities, we utilised the CPU profiler as a suitable alternative, effectively serving as a proxy for power profiling.

II. Pre-Development Research and Related Work

A. DOM Manipulation

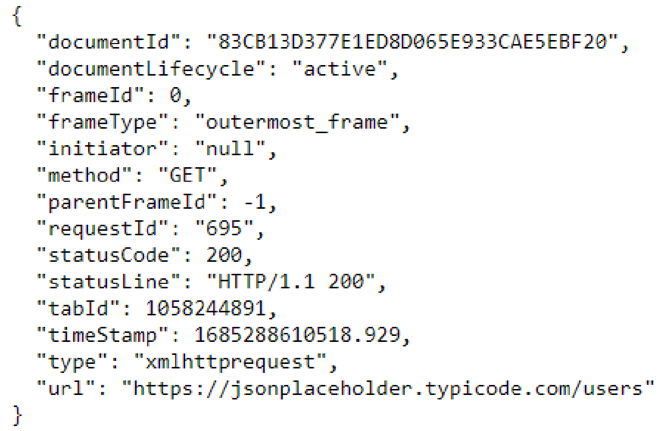
The Document Object Model (DOM) is a programming interface standardised by the W3C that treats HTML and XML documents as a tree structure [6, 7]. Each node in the structure represents a different part of the document on a webpage. This allows the use of JavaScript to create dynamic web applications [8]. This project goal is to manipulate DOM elements to expose provenance data back to the end user. One way this can be done is through JavaScript. Using JavaScript, users can get different elements on a webpage, select distinct types of nodes, and create and append new nodes [9].

B. Data Provenance

Provenance is the documentation of the origin, history, and transformations of data, providing a clear lineage and accountability for its usage [10]. In recent years data provenance has become more important as it exposes information flows of systems and allows users to make choices based on this information. This increased transparency of data flows allows us to hold misleading or incorrect data accountable. There are several applications in which provenance information is helpful. These include *Data Quality* where the quality of information can be estimated through its lineage, *Audit Trail* to trace the trail of data and detect errors, *Replication Recipes* which allows for the repetition of data, *Attribution* which can establish copyright or ownership of data, and *Informational* which can provide context from the lineage of metadata [11]. Exposing provenance data will allow end users to apply these applications on data from a webpage. These applications are beneficial for people in professions such as science, medicine, computational, and geospatial fields [10]. All of these fields require information to be accurate due to potential dangers which can occur from incorrect or misleading data. Being able to infer the quality of data, where it is from, having the ability to replicate it, establish copyright, and analyse its metadata are all valuable tools that can be used to increase the validity of a piece of data.

C. HTTP Headers

The Hypertext transfer protocol (HTTP) allows information to pass between a client and a server. An example of this is an end user's browser and server-side applications [12]. When this information is exchanged request and response headers are generated [13]. These headers contain additional information that can be used for things such as security. Examples of these requests are GET and POST Requests. Figure 2 shows what a HTTP header capture may look like. Some of the headers we are interested in are “method,” “type,” and “URL.” These headers are important because they can provide data provenance. In the example header image provided in this section, the “URL” header shows the origin of where the data came from. These URLs can be directly from the server-side backend application or a URL on the internet. It is worth highlighting that these headers can be customised to suit specific requirements. For instance, in POST responses, the location header is employed to appoint the redirection destination for webpage navigation [14]. Customised headers can be controlled and created by server-side applications.



```
{
  "documentId": "83CB13D377E1ED8D065E933CAE5EBF20",
  "documentLifecycle": "active",
  "frameId": 0,
  "frameType": "outermost_frame",
  "initiator": "null",
  "method": "GET",
  "parentFrameId": -1,
  "requestId": "695",
  "statusCode": 200,
  "statusLine": "HTTP/1.1 200",
  "tabId": 1058244891,
  "timeStamp": 1685288610518.929,
  "type": "xmlhttprequest",
  "url": "https://jsonplaceholder.typicode.com/users"
}
```

Figure 2. Example of HTTP Header Response

D. JavaScript Instrumentation

Software instrumentation is a technique that is used in software profiling, performance analysis, optimisation, testing, and error detection [15]. Instrumentation involves adding extra code to an application for monitoring some program behaviour. It can be performed statically at compile time or dynamically at runtime. There are different use cases when it comes to Instrumentation. In one study Instrumentation was used to apply policy-based code to web pages to avoid cyber security threats such as XSS attacks [16]. This study holds significance for my project as it aimed to enhance web application functionality by introducing an additional layer of security. Instead of relying on traditional security measures, this study employed instrumentation to reveal provenance data. Instrumentation serves another valuable purpose which is enabling the measurement of test coverage. Test coverage quantifies the portion of application code that undergoes testing and verifies if the test cases comprehensively cover all aspects of the code. An illustrative instance of this is JaCoCo, a freely available code coverage tool tailored for Java [17]. JaCoCo employs bytecode instrumentation while a Java agent is in operation to assess code coverage.

E. Plugin Architecture and Design

Developing browser and framework plugins are fairly simple with a low barrier to entry. All you need is a JavaScript file with your code and a manifest.json file for chromium-based browsers or a package.json file for framework-based plugins. This file has all of the metadata about your extension such as what scripts you are using and browser actions [18]. An important concept when it comes to good plugin design is to have good security policies that do not compromise an end user's experience [19]. Some common vulnerabilities in browsers include Cross-Site Scripting where an extension uses ***eval*** or ***document.write*** without sanitising inputs. Replacing Native APIs where a malicious web page can confuse a browser extension by replacing native DOM APIs with its own methods. JavaScript Capability Leaks where if an extension leaks one of its own objects the attacker can often access other JavaScript objects, which could include powerful extension APIs [19]. Mixed Content where an active attacker can control content loaded through HTTP. The most severe form of this attack happens when a browser extension loads a script through HTTP and runs it. A natural approach to mitigating these vulnerabilities is to limit extension privileges. Browser plugins can be divided into five groups "critical" where the plugin can run arbitrary code, "high" where the plugin can access site-specific information like cookies and passwords, "medium" where the plugin can access private user data like their history, "low" where the plugin can annoy the user, and "none" where the extension has no privileges [19]. Ideally, plugins will fall into the "low" and "none" category. This was achieved by ensuring that the plugins did not require more permissions than necessary.

F. Study of Existing Solutions

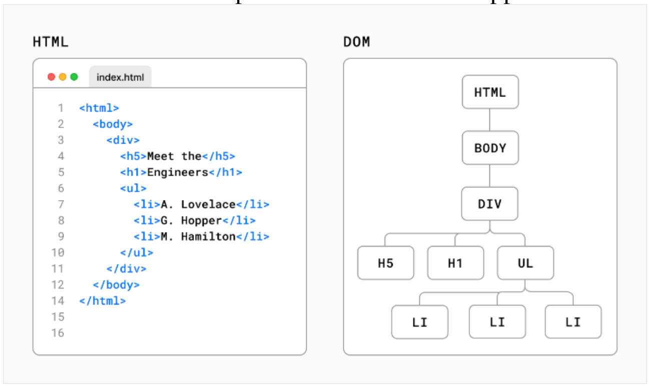
Through research, it was discovered that no one had attempted to address this problem in the same manner as we did. However, many studies have been conducted that have approached parts of the project goals. These existing solutions and studies came in helpful while developing different prototypes.

One of the studies found was conducted at the Blekinge Institute of Technology in 2022 where they compared DOM manipulation performance when using vanilla JavaScript and front-end JavaScript Frameworks [20]. This study was relevant to the project because two prototypes we were attempting to develop included a pure JavaScript implementation and a framework plugin implementation. One of the project's prerequisites was to minimise overhead during DOM instrumentation. The performance evaluation conducted in this study played a pivotal role in determining which approach received a greater allocation of resources. The frameworks used in this study were Angular, React, and Vue.js. This experiment was conducted by creating Vanilla JavaScript and the selected frameworks test applications. These applications were used as a base for comparing application size and for comparison tests of DOM performance-related metrics using Google Chrome and Firefox. The results of this study found that there was a distinct difference between the JavaScript and framework implementations. It found that vanilla JavaScript had the best performance and the smallest application size. The conclusions gained from this study stated that both methodologies are viable when manipulating the DOM, but a pure JavaScript implementation will yield better performance. This was reflected back in my implementation where the pure JavaScript implementation returned a better performance compared to the other approaches. This study also found that the DOM manipulation tests ran faster on Google Chrome compared to Firefox.

Another study discussed ways of visualising provenance data. This study proposes multiple provenance visualisation techniques where end-users can evaluate and understand the provenance data. Proposed visualisation methods use the W3C PROV-O specification for provenance data [21]. This data is displayed through a user interface where you can filter data to get information on a specific piece of data. While the initial plan did not include displaying this provenance data to the extent demonstrated in this project, it provided insights on potential methods to present it. One of these ideas was to display the raw data on a separate interface where the data was not formatted heavily to meet the project specifications. This was evident in the final product, where the provenance data was displayed within the webpage as a popup.

III. Design Requirements

To achieve the goal of exposing end-user data, software solutions were developed to instrument web applications DOM exposing their provenance data. The DOM is an interface standardised by the W3C that treats HTML documents as a tree structure [6, 7, 22]. Each node in this structure represents a different part of the web document. Figure 3 illustrates how this process looks and the difference between HTML code and the DOM. The idea was to manipulate elements of the DOM on web applications through instrumentation to create new node/s or child nodes to expose existing provenance data back to the end user. This instrumentation would happen when an HTTP Request is sent through a JavaScript framework. The scope of this project was to develop provenance support for the middle application layer only. Primarily this would be used for one-



```
<html>
  <body>
    <div>
      <h5>Meet the</h5>
      <h1>Engineers</h1>
      <ul>
        <li>A. Lovelace</li>
        <li>G. Hopper</li>
        <li>M. Hamilton</li>
      </ul>
    </div>
  </body>
</html>
```

Figure 3. HTML code and how the DOM represents this as a tree-like structure of nodes.

page applications which do not reload the entire page but use Ajax to fetch information. Ajax (Asynchronous JavaScript and XML) allows for the creation of asynchronous web applications [23]. An advantage of this is that data can be loaded onto a webpage without needing to reload the entire page. Instrumentation is a technique that is used in software profiling, performance analysis, optimisation, testing, and error detection [15, 24]. It involves adding extra code to an application for monitoring some program behaviour. It is used to transparently add functionality without touching the actual application or at least minimising modifications to reduce overhead. DOM manipulation was represented as a red line where clicking it displays the provenance data specific to that element.

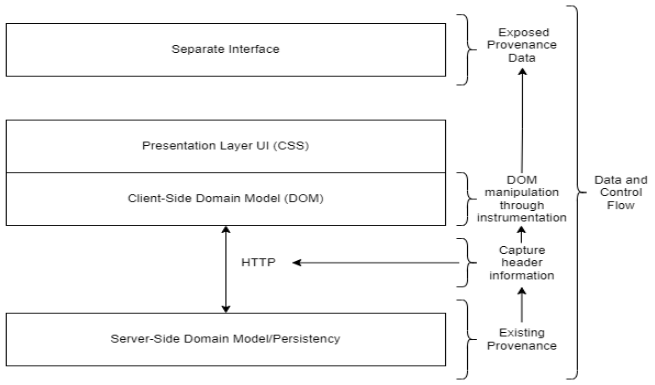

Separate Interface
Presentation Layer UI (CSS)
Client-Side Domain Model (DOM)
HTTP
Server-Side Domain Model/Persistency
Exposed Provenance Data
DOM manipulation through instrumentation
Capture header information
Existing Provenance
Data and Control Flow

Figure 4. Proposed data and control flow of my implementation

HTML will be altered because of DOM Instrumentation when header information is captured. HTTP makes a request to the server. Then the server sends back response headers. Provenance Data is persistent and sent by a server-side application. Exposed Provenance Data is received and displayed back to the user as a separate interface.

Figure 4 shows the data and control flow that is executed during this process. Existing Provenance is exposed by capturing HTTP headers which will cause the DOM to be instrumented and exposed provenance data to be displayed back to the end user. This diagram assumes that the server-side application facilitates provenance by incorporating a header within the response headers that directs to provenance data encoded in JSON. The development of this layer is not within the current scope but is actively in progress through the SFTI-funded Veracity Project [25, 26]. As part of my testing efforts for my final solution, a mock-up of this functionality was created.

Three types of solutions were developed for this problem, each of which contained multiple approaches attempted to solve the problem. A browser plugin solution, a JavaScript instrumentation solution activated in browser code, and a framework plugin solution. Iterations of these software solutions were developed to achieve the goal of DOM instrumentation. While developing these solutions the Rational Unified Process (RUP) was followed [27]. RUP has four stages. The first stage is the inception phase. In this phase, an attempt was made to gather feedback on the proposed solutions to assess their viability. The subsequent stage is the elaboration phase, during which the requirements of the solutions are examined. This stage resulted in a more profound understanding of the systems that were being developed. The third step is the construction phase, where the software components are built. During this phase, a return to the elaboration phase was made whenever the development of a new software component was required. The Transition phase marked the conclusion of the project, involving performance testing and the deployment of the software solutions to GitHub for future development. Following this process meant that my project met the ISO9126 usability standards as it provided specification and an evaluation model for the quality of my software [28]. This standard can be divided into four parts. The *Quality model*, *External Metrics*, *Internal Metrics*, *Quality in use metrics*. Within the *Quality model* it specifies six characteristics to meet this standard. Functionality, Reliability, Usability, Efficiency, Maintainability and Portability. To ensure the software met these characteristics the criteria we evaluated the prototypes against are runtime overhead, difficulty to install, development costs, and how effectively it meets the project requirements. While awaiting the development of the Server-Side Provenance software, we opted for alternative solutions by utilising various HTML and header metadata in its place.

IV. Implementation

A. Browser Plugin

The development of a browser plugin was initially undertaken as the first solution. The initial strategy involved selecting the specific web browser to target for this plugin, recognising that each browser comes with its unique requirements and operates on distinct frameworks (e.g., Chromium). To make an informed decision, research was conducted to determine which browser engine would be the most suitable choice.

Once this was settled, the next step was to delve into the intricacies of creating browser plugins tailored to that specific platform. To validate the feasibility of this endeavour, a basic mock plugin for testing purposes was used. Subsequently, the primary objective shifted towards the development of a plugin capable of instrumenting the Document Object Model (DOM) and exposing some type of data not necessarily the provenance data for the moment. Achieving this required extensive research into the techniques for injecting JavaScript code into a browser through a plugin.

This process involved the iterative process of refining and rigorously testing prototype browser plugins that would seamlessly integrate the functionalities of the previously developed plugins. This resulted in having many different iterations of browser plugins where some approaches were more viable than others.

1) *Planning and Justification*

The Chrome browser was the opted target, primarily because it has the highest popularity among its competitors, such as Firefox and Safari [29]. A notable advantage of choosing Chrome was its foundation on the Chromium web browser, developed by Google. This decision also offered compatibility with other browsers utilising Chromium, including Microsoft Edge, Opera, and the Brave browsers [30].

This multi-browser support represented a significant advantage of this approach.

Nonetheless, a drawback of this choice was the lack of compatibility with other widely used browsers like Firefox. The incompatibility issue stemmed from the fact that each browser employs its unique rendering methods and JavaScript engines. For instance, Safari and Chrome utilise the Web kit browser engine, while Firefox relies on the Quantum engine [31]. These differences result in varying interpretations of data by different browsers. Consequently, the development of plugins requires conforming to distinct standards for each browser, complicating cross-browser compatibility.

Having settled on Chrome as the target, research into Chrome plugin architecture began. This enabled familiarity with Chrome plugin development.

2) Chrome Plugin Architecture

Chrome Plugin Architecture, also known as Chrome Extension Architecture, is a framework that enables developers to enhance the functionality of the Google Chrome web browser. These extensions or plugins are small software programs that can modify and extend the browser's capabilities, add features, customise the user interface, or interact with web applications in unique ways. Here is a list of the aspects that were of primary interest for this project [32]:

Manifest File:

At the core of a Chrome extension is the manifest file (manifest.json). This file acts as metadata, defining the extension's structure, permissions, and various settings. It specifies what the extension can and cannot do, what web pages it can access, and more.

Background Pages:

Extensions often include background scripts or pages that run in the background, even when the user is not actively interacting with the extension. These scripts can manage tasks like monitoring changes in web pages, handling user settings, or managing communication between distinct parts of the extension.

Content Scripts:

Content scripts are JavaScript files that can be injected into web pages. They allow extensions to interact with and modify the content and behaviour of web applications. This is useful for tasks like injecting custom CSS, altering page content, or adding interactive elements.

Permissions:

To ensure security and privacy, extensions need to declare the permissions they require in the manifest file. These permissions specify what the extension can access, such as tabs, cookies, and storage.

3) Plugin Implementations

The development journey commenced with the creation of a plugin designed to insert a button into webpages whenever an "img" tag was detected [33]. This button's purpose was to redirect users to the original URL of the image upon clicking. The mechanism behind this plugin involved assembling all "img" tags into an array, which was subsequently traversed in a loop. Within this loop, a "div" element was appended to the page, serving as a parent to the "img" tag. Inside this "div"

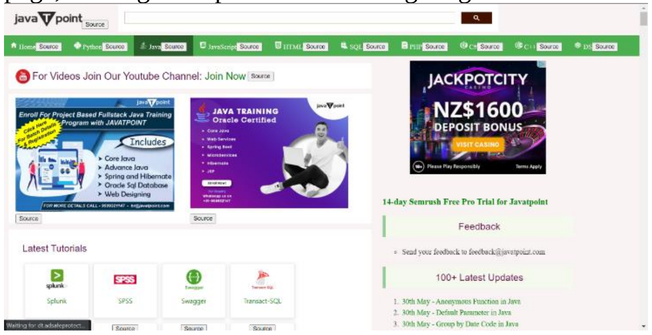


Figure 5. Webpage with browser plugin activated

Source button attached to images by wrapping the image element in a div element.

element, a button was dynamically added to facilitate image URL navigation. You can visually see this in Figure 5 which showcases the webpage after the plugin activation. You can see that a button has been integrated with each of the images on the webpage. The plugin was utilised as an educational tool to gain experience in developing Chrome extensions and injecting HTML elements into webpages using JavaScript.

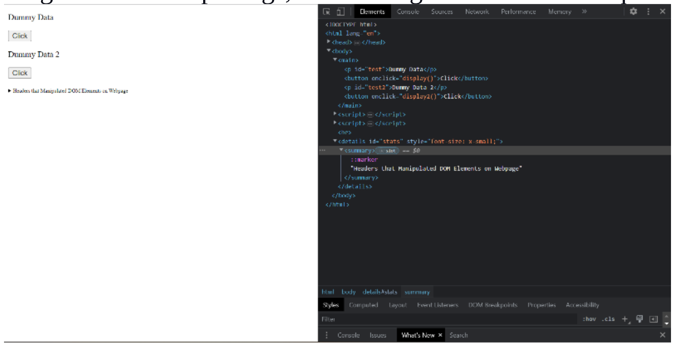
Moving forward, the next endeavour was to explore how to capture header requests and responses transpiring within web pages. This project necessitated the capability to capture HTTP Headers. To fulfil this requirement, an Intercept Headers Test Plugin was designed, which harnessed the WebRequest API to intercept and capture all headers arriving and departing from a webpage [34]. These captured headers were then presented in a separate Chrome devTools tab.

Lastly, before the final browser implementation, a Display Headers Test Plugin was developed, which was responsible for rendering the captured headers in a list formatted within an HTML structure [35]. This process functioned by capturing each incoming response header from a webpage and associating it with a unique Chrome session key. This key was automatically cleared upon closing the browser. Subsequently, upon each browser refresh, these headers were appended to a list and presented as a popup.

4) Primary Plugin Prototype

Through the testing of various plugins, a prototype plugin was successfully developed. This plugin had two primary functions: firstly, it can display data via HTTP headers, and secondly, it can instrument a web applications DOM by highlighting elements that have been altered by Ajax [36].

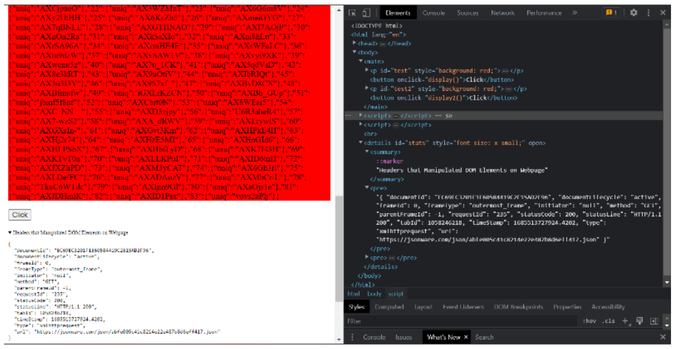
To evaluate this implementation, a mock webpage was designed that can be launched in a web browser. Tests were also conducted on specific internet web pages. The core functionality of this plugin was centered around identifying DOM elements that have been altered using Ajax, specifically altered using the XMLHttpRequest API. It accomplishes this by scanning the webpage, detecting instances of XMLHttpRequest usage within "script" tags, and storing the associated request.



```
Dummy Data
Click
Dummy Data 2
Click
▸ Headers that Manipulated DOM Elements on Webpage
```

```
<!DOCTYPE html>
<html lang="en">
▸<head>…</head>
▾<body>
  ▾<main>
     <p id="test">Dummy Data</p>
     <button onclick="display()">Click</button>
     <p id="test2">Dummy Data 2</p>
     <button onclick="display2()">Click</button>
   </main>
  ▸<script>…</script>
  ▸<script>…</script>
   <hr>
  ▾<details id="stats" style="font-size: x-small;">
    ▾<summary> == $0
       ::marker
       "Headers that Manipulated DOM Elements on Webpage"
     </summary>
   </details>
 </body>
</html>
```

Figure 6. Webpage and DOM before instrumentation



```
<!DOCTYPE html>
<html lang="en">
▸<head>…</head>
▾<body>
  ▾<main>
    ▸<p id="test" style="background: red;">…</p>
     <button onclick="display()">Click</button>
    ▸<p id="test2" style="background: red;">…</p>
     <button onclick="display2()">Click</button>
   </main>
  ▸<script>…</script> == $0
  ▸<script>…</script>
   <br>
  ▾<details id="stats" style="font size: x small;" open>
    ▾<summary>
       ::marker
       "Headers that Manipulated DOM Elements on Webpage"
     </summary>
    ▾<pre>
       "{ "documentId": "FC69FC3201F1F60584419C2F15AD2F96", "documentLifecycle": "active",
       "frameId": 0, "frameType": "outermost_frame", "initiator": "null", "method": "GET",
       "parentFrameId": -1, "requestId": "235", "statusCode": 200, "statusLine": "HTTP/1.1
       200", "tabId": 1058246218, "timeStamp": 1685513727924.4202, "type":
       "xmlhttprequest", "url":
       "https://jsonware.com/json/ab1e005c41c0214e22ed07b0d6e11417.json" }"
     </pre>
    ▸<pre>…</pre>
   </details>
 </body>
</html>
```

Figure 7. Webpage and DOM after instrumentation

Simultaneously, the plugin records all incoming responses to the webpage. When a response is received, it compares the response URL to the request URLs collected earlier. If there is a match, it indicates that a DOM element has been manipulated through Ajax and highlights that element in red. Furthermore, the plugin can list all headers that have affected DOM elements at the bottom of the page using HTML manipulation with JavaScript.

For a visual representation of this process, refer to Figures 6 and 7, which provide before-and-after images of a test webpage developed along with corresponding HTML [37]. Additionally, a short video demonstration of the main prototype's capabilities has been created for those interested in a showcase: Demo Video.

5) Challenges and Limitations

Throughout the development of these plugins, several challenges were encountered due to the inherent limitations of plugin development. One notable limitation was the absence of a direct method to associate DOM Elements with HTTP Headers. To address this limitation in the plugin's implementation, we resorted to performing string searches to determine if the URL arriving at the webpage matched the URL sent in the HTTP Request. However, this approach had its drawbacks, primarily the potential for false positives. This is due to the exclusion of any query parameters included in the URL during the string search. The query parameters are appended when the request is sent, rather than being part of the XMLHttpRequest. Consequently, there is a possibility of associating a header with a webpage arrival that did not manipulate the DOM. In future iterations, alternative approaches to address this issue were explored.

Another challenge encountered was the plugin's impact on browser performance. Some web pages generate numerous requests, resulting in a slowdown when the plugin compares each request URL with all the URLs used by the XMLHttpRequest API. When observed, it had around 50% slowdown on normal web application processes. To optimise the plugin's performance, we modified its behaviour. Instead of storing all incoming URLs in an array, We send each URL as a message to my main.js script, where it is compared to the request URLs received in each message. This approach eliminates the need to compare two arrays, which had been slowing down the browser.

Plugin security is also another challenge frequently encountered. This was due to Chrome's built-in security which helps mitigate exposing unneeded data to plugins [38]. Unfortunately, this led to limitations on what tools we could use. One of these was the cross-origin limitation. Plugins often need to interact with external websites or APIs, but Chrome's Same-Origin Policy and CORS (Cross-Origin Resource Sharing) restrictions can hinder these interactions. This made it extremely hard to evaluate whether the implementation worked on public web applications. Secondly, Chrome enforces CSP rules (Content Security Policy) that restrict the use of inline scripts and any external resource in web pages [39]. This can affect the ability of plugins to inject scripts into web pages or manipulate page content, making it challenging to implement certain features.

While these security measures are essential for safeguarding user privacy and maintaining the integrity of the Chrome browser, they do add complexity and constraints to plugin development. These limitations underscore the motivation for exploring a different approach to solve the project goals. As a result of this, the development of this software was continued using a different approach.

B. JavaScript Instrumentation

Having determined that the browser plugin approach was not the most suitable solution for this project, we subsequently shifted to exploring a pure JavaScript approach to fulfil the project's objectives.

This process involved incorporating a script into the browser's HTML code within a <script> tag to instrument DOM elements [40]. Initially, research was conducted to gain insights into JavaScript-based instrumentation techniques. Following this research phase, a mock HTML webpage was created specifically designed to assess webpage instrumentation. The goal here was to assess its functionality and effectiveness. Subsequently, a script was created to instrument the Document Object Model (DOM) for the purpose of exposing metadata. This script was evaluated on the mock website to ensure it performed as expected. To make this instrumentation available to the public, users needed to include the script in their projects by inserting it within a <script> tag when publishing their web applications. This approach aimed to achieve the desired instrumentation goals while ensuring compatibility and usability for end-users.

By the end of the JavaScript Instrumentation development cycle, three program iterations had been produced, each one extending upon the last.

1) Iteration One: MutationObserver

a) Research and Planning

Initiation began by exploring methods to identify DOM changes exclusively using pure JavaScript, without relying on any frameworks. During this investigation, a JavaScript API known as the MutationObserver was discovered [41]. The MutationObserver provides a way to asynchronously observe and react to changes in the DOM. It allows for the monitoring of mutations signified by changes in the structure of an HTML document. Mutations can include things like adding or removing elements, modifying attributes, or changing the content of elements. The key components of the MutationObserver API include the MutationObserver object which you can initialise to observe specific elements in the DOM. These mutation types are:

childList:

Observes changes in the child elements of the target node.

attributes:

Observes changes to attributes of the target node.

characterData:

Observes changes to the text content of the target node.

subtree:

Extends the observation to the entire subtree of the target node.

You can also designate a specific target node for observation. In the context of this project, the target node encompassed the entire webpage document. Whenever there is a DOM mutation, it produces a mutation record, enabling the extraction and analysis of both the previous and the updated DOM values. This facilitates straightforward comparisons to identify the specific changes made to the DOM element. The Mutation Observer is commonly used for implementing features like auto-saving forms, lazy-loading content, and updating UI components in response to changes in the DOM. This tool is powerful for constructing dynamic web applications, precisely what we needed for this project.

b) Architecture and Implementation

The architecture of this iteration involved two main components inject.js and detect.js [42]. The inject.js script facilitates DOM manipulation while detect.js responds to these manipulations. This architecture allows for dynamic and conditional manipulation of web applications.

During implementation, the inject.js script was designed to search for <script> elements in the HTML document that contain the text "new XMLHttpRequest()". This is a similar approach used in the browser plugin implementation. If found, it modifies those <script> elements by adding code to set properties of specific elements identified by their IDs. These properties include the URL and Headers received by the HTTP response initiated by the Ajax request. Specifically, it stores these properties in a document variable. It assigns the response headers to "document.getElementById().provenance". This occurs when the webpage loads which is important for detect.js.

The detect.js script actively tracks DOM manipulations and responds to them accordingly. When these changes take place, it retrieves and presents properties defined in inject.js to the browser console. These properties include the request URL and response headers. Additionally, it visually emphasises the altered DOM element by applying a randomly selected colour from a colours array. The design pattern applied in this context is the observer pattern, which falls under the category of behavioural design patterns [43]. The observer pattern proves valuable when seeking to monitor the state of an object and receive notifications whenever it undergoes any changes. Within this pattern, the entity observing the state of another object is referred to as the "Observer," while the object being observed is called the "Subject." This design pattern was implemented through the utilisation of the MutationObserver.

c) Challenges and Limitations

While the code may work for some simple cases, it has several limitations and issues. The code relies on parsing and manipulating the JavaScript code embedded within <script> elements. This approach is fragile because it assumes that the Ajax calls are constructed using a specific pattern ("new XMLHttpRequest()") and that they are directly embedded within <script> elements. Modern web applications often use more complex mechanisms for making Ajax requests, such as using libraries like jQuery for which this code does not account. This code's effectiveness also depends on the browser's ability to oversee dynamic script replacement. Some browsers may not execute dynamically inserted scripts reliably during runtime, leading to inconsistent results and potential errors [44]. This code also uses a linear search to iterate through all <script> elements in the document to find the ones containing the target text. This can be inefficient when there are many scripts on the page, potentially impacting performance.

d) Evaluation

To evaluate this implementation, a simulated index.html page equipped with buttons that triggered Ajax requests to different public endpoints was constructed [45]. Through this testing process, several of the limitations we had previously outlined were found. While this iteration partially fulfilled certain project goals, it became evident that it needed further enhancement due to the uncovered limitations. Specifically, there was a recognition of the need to dynamically identify Ajax calls during runtime rather than in a pre-runtime context, and to instrument other types of Ajax requests, including those using jQuery. This iteration served as the foundational stepping stone for all subsequent iterations.

2) Iteration Two: Aspect-Orientated Approach

a) Research and Planning

In the subsequent iteration, the direction of using Aspect-Oriented Programming (AOP) to instrument DOM elements during runtime was suggested by the supervisor. Aspect-Oriented Programming is a programming paradigm that aims to modularise cross-cutting concerns in software applications [46, 47]. Cross-cutting concerns are aspects of a program that affect multiple modules or components, such as logging, security, error handling, and performance monitoring. AOP provides a way to separate these concerns from the main logic of a program, making the codebase more maintainable and less cluttered with repetitive code. You can wrap functions with "advice" functions that execute before, after, or around the target function. One of the project's requirements is to ensure that the program is lightweight and can be easily integrated into webpages, and this programming paradigm addresses that requirement effectively as it will automate the instrumentation process.

b) Architecture and Implementation

The architecture of this iteration consists of several JavaScript files that work together to implement a system for adding aspects to functions in an API [48]. It uses JavaScript's Proxy object to intercept function calls and execute additional code (aspects) before, during, and after the original function execution. Aspects are functions that can execute code before, during, or after the original method execution. This architecture encourages code reuse and allows for the dynamic adaptability of the system. This approach is especially beneficial in large and complex software systems where cross-cutting concerns can quickly become unwieldy if not effectively managed.

Implementing this approach involved four JavaScript files. The addAspects.js file defines a function called addAspects that accepts a variable number of aspects as arguments. Inside the function, it creates a get function that is used as a handler for a JavaScript Proxy [46]. The get function intercepts property accesses on an object (in this case, it is used as a Proxy handler). If the accessed property is not a function, it returns the original value using ***Reflect.get***. If the accessed property is a function, it wraps it in an async function that executes aspects before, during, and after the original function call. The aspects are executed using a run function for each pointcut ('before', 'during', 'after').

The api.js file creates an API object by calling the addAspects function and passing in an aspect. The API object has the addScripts method.

The logger.js file defines a logger object with the method addScripts. It defines an aspect as an object with a before method. In the before method, it parses and processes JavaScript code looking for specific patterns (e.g., XMLHttpRequest). The aspect does not modify the original function's behaviour.

The detect.js is the same as the last iteration as it still captures manipulations that occur in the DOM during runtime.

c) Challenges and Limitations

The introduction of aspects can make it more challenging to understand the flow of a program, as behaviour can be scattered across several aspects making the codebase harder to maintain and debug. This is known as weaving which occurs during runtime [49]. We also found it had quite a steep learning curve as it introduces a unique way of thinking about code organisation and behaviour. It is commonly associated languages such as Java and C#, but support in JavaScript is limited [50].

d) Evaluation

To evaluate this code, a simulated index.html page was constructed equipped with buttons that triggered Ajax requests to different public endpoints the same as the previous iteration [45]. During the testing process, it was realised that this approach would result in the generation of numerous false positives and could be challenging to comprehend. While this iteration fulfilled the project goal of identifying Ajax calls dynamically during runtime rather than pre-runtime, more research was conducted as a simpler approach was desired. This iteration served as another foundational stepping stone as in the final JavaScript Instrumentation iteration, the concept of proxying was used.

3) Iteration Three: Instrumentation through Proxying

a) Research and Planning

This was the final iteration of the JavaScript Instrumentation approach. From researching the technique of proxying another technique called functional hooking was found. Functional hooking is a technique used to intercept and modify the behaviour of functions or methods at runtime to extend its functionality [51, 52]. It is commonly used for debugging, profiling, monitoring, and extending the functionality of existing code without modifying the original code. This was precisely the technique that was sought after, as it would facilitate the straightforward instrumentation of JavaScript code while it was executing in the browser. It is remarkably similar in design to Aspect-Oriented Programming except a lot easier to understand and implement. Expanding on this proxying is a technique employed in Aspect-Oriented Programming (AOP). In this approach, however, ***XMLHttpRequest.prototype.open()*** is directly proxied. While this method is lower level in nature compared to AOP, it offers several advantages. Notably, it provides developers with a greater degree of control, enhances efficiency, and empowers them to wield a higher level of control over the process [53]. Functional hooking is a powerful technique that provides flexibility for extending and modifying the behaviour of functions or methods in a dynamic and non-invasive way. The plan moving forward was to implement this technique to proxy Ajax methods when they are called. Then detect when a DOM element is manipulated by Ajax using the MutationObserver API.

b) Architecture and Implementation

This iteration consists of three files: proxy.js, observe.js, and init.js. The observe.js file utilises the MutationObserver API, similar to previous iterations, to identify changes in the DOM. The init.js file is responsible for initializing global variables. Lastly, the proxy.js file serves as a proxy for the "open()" method of the XMLHttpRequest API. This architectural design enables the monitoring and instrumentation of JavaScript-triggered Ajax requests made by web applications, with the collected information stored in a document variable for subsequent analysis.

The code starts by creating proxies for the "***XMLHttpRequest.prototype.open()***" method and the fetch function [54]. When "***XMLHttpRequest.prototype.open()***" is called, it creates event listeners for onload and load events. These listeners capture response data, response headers, and the request URL. It then stores the information in the DOM element with the target's ID. Similarly, when a fetch request is made, the code intercepts the response headers, stores them, and updates the DOM element with the target's ID. When a mutation is detected, it checks if the provenance property of the mutated target (DOM element) has data stored in it. The provenance property is what the previous information was stored in. If this has data written to it, it calls the "provenanceString" function to log information about the mutation, including the old and new values, the DOM element's ID, the request URL, and response headers. The "provenanceString" function then formats and logs information about DOM mutations, including the old and new values of the DOM element, the ID of the element, the request URL, and response headers.

c) Challenges and Limitations

In general, implementing this iteration was smooth and straightforward, with few challenges and limitations encountered. However, one notable challenge arose when attempting to extend this functionality to server-sent requests which is a push protocol. It is worth noting that this was introduced as an experimental feature, as it was not even under consideration in the previous iterations discussed.

Server-Sent Events (SSE) in JavaScript, also known as Server-Side Events, provide a one-way, real-time communication channel from the server to the client, typically a web browser [55]. SSE allows for the immediate delivery of updates or events from the server to the client, eliminating the need for continuous client polling. Implementing this approach involved creating a PHP server which would then send data to the client as an "EventSource" every second [56]. A specific issue encountered while working with SSEs was the absence of response headers in SSE requests. To address this limitation, we experimented with making a secondary fetch request each time an SSE event occurred. Unfortunately, this approach had a significant impact on performance since it required two requests for each SSE event. Additionally, it did not accurately reflect the original SSE call's results. Following a discussion of this challenge with the supervisor, the decision was made to forgo the instrumentation of Server-Sent Events (SSEs) for this specific project. This choice was driven by the recognition that the performance overhead introduced by the additional fetch request outweighed the benefits of monitoring SSEs.

d) Evaluation

My assessment of this iteration mirrors my experience with previous iterations, where an index.html file was constructed to execute various HTTP requests via Ajax. Overall, this process was found to be remarkably comprehensible and easy to integrate into web applications. Moreover, it was observed that this methodology aligned well with the project's objectives. It effectively enabled the instrumentation of DOM elements in real-time using the functional hooking technique and the exposure of provenance data facilitated by HTTP Requests through the MutationObserver API.

Notably, this was achieved solely with pure JavaScript, without reliance on any JavaScript frameworks. This independence from external dependencies means that it can seamlessly integrate into a wide array of modern web applications, ensuring its versatility and adaptability. As a result, I implemented a similar approach when creating a Framework plugin, which enabled the utilisation of the same code with varying syntax depending on the JavaScript framework.

C. Framework Plugin

Framework plugins are valuable tools in modern web development because they allow developers to leverage existing solutions and reduce the amount of code they need to write [57]. This can speed up development and improve code maintainability. Fortunately, prior experiences working on the pure JavaScript program have supplied a codebase that can be readily adapted into a JavaScript Framework. The next step in this process was selecting the specific framework intended for use before embarking on the implementation phase. After making this decision, we proceeded to seamlessly incorporate the previous JavaScript code into the chosen framework. Once the integration was finalised, the subsequent task revolved around deciding the method to be employed for publishing this plugin.

1) Planning and Justification

The planning process commenced with an assessment of the ideal framework for the plugin's development. Among the notable JavaScript frameworks under consideration were ReactJS, Angular.js, Vue.js, and Next.js. Of these, ReactJS and Angular.js emerged as strong contenders due to prior experiences with them. Each framework had its own set of advantages and drawbacks within the development community. The decision leaned towards ReactJS for several compelling reasons [58]. Firstly, React's utilisation of a virtual DOM allowed for efficient updates and rendering of UI changes, targeting only the modified portions of the DOM, thus enhancing performance—a critical factor for this lightweight and highly integrable implementation. Secondly, React has widespread popularity, ranking among the most frequently used JavaScript frameworks, ensuring extensive documentation and a broader user base, making the plugin more accessible. Additionally, React exhibited excellent compatibility with the jQuery JavaScript library [59]. This synergy was attributed to React's component-based architecture, which worked well with jQuery's approach to managing UI elements, particularly in cases involving HTTP calls where jQuery employed Ajax.

After finalising the framework selection, the next phase of planning focused on the publication strategy. It was decided that NPM would serve as the optimal platform for releasing the plugin. NPM's immense user base, composed of JavaScript and Node.js developers, would offer unparalleled accessibility to a thriving community well-versed in the ecosystem [60].

2) ReactJS Architectural Elements

Numerous architectural elements in React piqued our interest, with one of the most significant ones being components. In React, components serve as the foundational building blocks of an application. They are reusable, self-contained entities that encompass both the user interface (UI) and the behaviour of specific parts of the application. Components in React can take two forms: functional or class-based, with class-based components offering additional features like state management. Given these characteristics, it was concluded that the most effective approach for incorporating this functionality into the project was to transform my JavaScript instrumentation code into a React Component. End-users could then easily install and utilise it through the Node Package Manager (NPM).

ReactJS components can also maintain an internal state that represents data specific to that component. This state can be modified within the component, and whenever it changes, React automatically initiates a re-render of the component and its child elements. This mechanism was invaluable, particularly in the displaying of provenance information to the end-users.

3) Implementation

Implementing this plugin was straightforward due to the existing codebase that we could seamlessly migrate. We only had to adjust the code to align with ReactJS programming conventions [61].

The code begins by importing the useEffect hook from the React library. The useEffect hook is a critical tool for managing side effects in functional components. Within this custom React hook, the useEffect hook is employed to establish and dismantle a MutationObserver, which observes changes within the Document Object Model (DOM). The observer's configuration encompasses monitoring character data alterations, subtree mutations, and attribute modifications in the DOM. When the component utilising this hook is unmounted, the observer is disconnected to ensure resource cleanup. Otherwise, the code functions identically to the JavaScript instrumentation code previously developed, retaining its ability to instrument DOM elements in response to Ajax calls.

After this integration, this component was then published to NPM so that it could be used by any end-users who are developing React Projects.

4) Evaluation

The evaluation of this solution entailed the development of a React web application using Node.js, similar to previous implementations. This web application incorporated Ajax functionality for testing purposes. Subsequently, a server application was created to assess the effectiveness of the instrumentation when Ajax calls were initiated.
The evaluation yielded results similar to those obtained through my earlier pure JavaScript instrumentation approach. Notably, several challenges encountered previously did not reoccur, thanks to my prior experience in developing diverse solutions to address this issue. Furthermore, the solution's accessibility was enhanced as it was made readily available through NPM, simplifying its deployment and integration.

D. Final Implementation

The approach found to be the most suitable for meeting the requirements, and the one chosen for further development, was the pure JavaScript Instrumentation approach. This decision was primarily influenced by its versatility and straightforward usability. Here is a demo video of how this implementation works Demo Video.

1) Usage Instructions

To integrate this into any JavaScript web application, follow these steps:

1. Add the instrument.js file to your HTML page with the following script tag:
   <script src="instrument.js"></script>
2. Include the popup.css stylesheet in your HTML page with the following link tag:
   <link rel="stylesheet" href="popup.css">
3. In the instrument.js file, you can customize the "***provHeaderName***" variable to match the name of the header you are using to store the URL pointing to your provenance generation endpoint.

This straightforward setup allows for the seamless incorporation of this functionality into web applications while offering the flexibility to adapt it to specific requirements.

2) Architecture and Implementation

This implementation consists of two core components: a JavaScript file named "instrument.js" responsible for program logic, and a CSS file called "popup.css" dedicated to styling the provenance popup [62]. The code is modular and organised into contrasting functions to streamline instrumentation.

DOM Mutation Observation:

The program initiates a MutationObserver, which tracks changes in the Document Object Model (DOM). When DOM mutations occur, the observer's asynchronous callback function is invoked. It inspects mutations for the presence of provenance data within manipulated DOM elements. If provenance data is detected, the code schedules a delayed call to displayProvenance() to present this information.

Proxying AJAX Calls:

The code extends the functionality of the XMLHttpRequest object's open method to capture details about Ajax requests. Similarly, it augments the behaviour of jQuery Ajax requests using the addEventListener method. Information such as the HTTP method, URL, response data, and headers is captured. The findProvenanceHeader() function is invoked to search for a specified provenance header within the response headers.

Header Search and Data Retrieval:

The findProvenanceHeader() function searches the response headers for the specified provenance header. If the header is found, it triggers getProv() to fetch and store the associated provenance data. The getProv() function performs a fetch operation to obtain provenance data from a designated server, using the provided URL. The acquired data is then stored within the relevant DOM element.

Displaying Provenance Information:

The displayProvenance() function is invoked when provenance data is identified in a DOM mutation. This function extracts pertinent information and creates a toggleable pop-up element for presenting the provenance data to end users. Furthermore, createTable() is called to generate an HTML table from the JSON-based provenance data, which is then inserted into the pop-up.

Table Creation:

The createTable() function is responsible for producing an HTML table from a JSON object. It is designed to manage nested JSON objects and construct the table accordingly.

Popup Creation:

The createPopup() function generates a pop-up element for DOM elements that have been updated via Ajax. These popups allow users to toggle the visibility of provenance information. If a popup for a specific element already exists, it is updated; otherwise, a new one is created. The content within the pop-up includes comprehensive details about the Ajax request and the associated provenance data.

Styling:

The styling and formatting of the popup elements are defined in the "popup.css" file.

In summary, this implementation serves as a robust instrumentation system for tracking and presenting provenance data linked to Ajax requests in web applications. It effectively combines DOM mutation observation with the interception of XMLHttpRequest and jQuery calls to achieve these goals. The result is a user-friendly display of provenance information within popups and informative console logs for debugging purposes.

3) Justification

The selection of the JavaScript Instrumentation approach was primarily driven by the code's readability and ease of understanding. A significant advantage of this choice is the enhancement of collaboration. Code that is straightforward to read and grasp promotes smoother collaboration and contributes to a more open-source environment. It allows for multiple developers to work on the same codebase efficiently, as they can swiftly comprehend the logic and purpose of the code. This ease of comprehension also ensures that future developers can easily onboard, which is a notable advantage.

Furthermore, this approach offers flexibility, which was a pivotal reason for favouring it over the Framework plugin approach. The code's flexibility enables seamless adaptation to various JavaScript frameworks, exemplified by the development of the React plugin. Adapting the code to match React's syntax and coding conventions was a straightforward task. Consequently, should the need arise to employ the code in a different JavaScript framework in the future, minimal adjustments would be required, thanks to its lightweight nature. Its lightweight quality also translates into effortless usage, as demonstrated in the Usage Instructions section. Deploying this code on a web application entails only three simple steps, primarily involving referencing it in HTML documents.

The code also boasts low coupling, meaning that its modules or components have minimal external dependencies. This reduces the risk of unintended side effects when making modifications and broadens its applicability in various modern web applications.

In terms of design, a deliberate choice we made during the final implementation was to consolidate all the code into a single file. This decision was motivated by the desire to simplify usage within web applications, where the inclusion of just one JavaScript file enhances user-friendliness. Additionally, this consolidation made it more convenient to monitor the program's performance within a web application, as it facilitated the need to only track one JavaScript file rather than multiple files, allowing me to easily gauge Overhead.

4) Challenges and Limitations

During development, notable challenges we encountered, particularly when attempting to display the provenance data within the popup. The difficulty in this aspect primarily revolved around dynamically formatting the provenance JSON into a table. This posed a challenge because a solution needed to be devised so that it could manage the complexities of potentially nested JSON structures. Fortunately, this challenge was successfully resolved by researching and implementing algorithms for JSON formatting [63].

Overall, this implementation exhibits few limitations that would hinder the achievement of project objectives. Many of the limitations are a result of my conscious effort to maintain alignment with the project's defined scope, ensuring that the development did not become overly complex. For instance, choosing only to instrument Ajax calls, as specified in the project outline, rather than extending instrumentation to other types of HTTP calls.

One limitation that warrants mention, not explicitly outlined in the project guidelines, is that an HTML ID must be attached to the element for it to be instrumented. For example, an HTML element like ***<p>Example Text</p>*** will not be instrumented, whereas ***<p id="example">Example Text</p>*** will be. This limitation was accepted based on the convention of assigning important HTML elements an ID for identification, tracking, and styling purposes. To ensure transparency with end users, this has been clearly documented in the project’s README under the limitations section [62].

V. Evaluation of Final Solution

A. Methodology

My evaluation methodology spanned all facets of the project, encompassing both the development and post-development phases. Throughout the development phase, the Rational Unified Process (RUP) was employed, a structured approach that comprises four distinct stages [27].

The inception phase marked the project's initiation, during which we conducted extensive research and meticulous planning. This phase laid the groundwork, defining project goals and objectives. Subsequently, the elaboration phase followed, where plans were justified, refined project requirements, and actively sought feedback. This iterative process allowed for the validation and adjustment of project direction. As development progressed into the construction phase, solutions were diligently implemented while adhering to the refined project requirements. Whenever required, a return to the elaboration phase was made to ensure alignment with project goals. The final transition phase involved performance testing and the transition of my codebase to GitHub. This last step ensured a solid foundation for future development and collaboration.

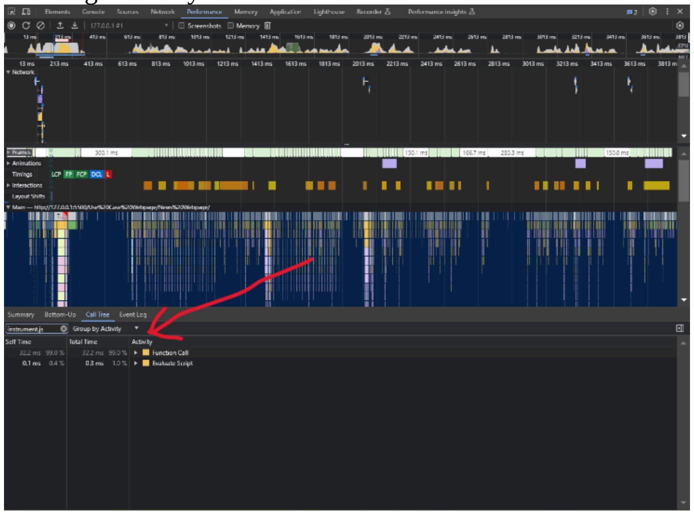
Following the conclusion of the development phase, an evaluation of the finalised solution was undertaken. This evaluation sought to determine the extent to which the project goals had been met. Based on this assessment, a well-informed judgment regarding the overall success of the project was made.

For transparency, it is important to note again that the server-side provenance software, which is being developed by a separate entity, had not reached a state of completion at the time of this project. For testing purposes, initiative was taken to create our own provenance endpoint and generate relevant provenance data to demonstrate what the final implementation would resemble.

B. Performance Testing

A mock webpage was established on localhost to assess the functionality and performance of the instrumentation [64]. Performance testing was conducted using the native Chrome performance tools to assess the additional load introduced by the solution.

These performance assessments were saved as JSON files, allowing for easy visualisation of the event timeline within



Self Time		Total Time		Activity
32.2 ms	99.0 %	32.2 ms	99.0 %	Function Call
0.1 ms	0.4 %	0.3 ms	1.0 %	Evaluate Script

Figure 8. Performance metrics within Chrome DevTools

the Chrome DevTools as seen in Figure 8 [65]. This panel allows developers to record and analyse the runtime performance of web applications. It provides insights into CPU usage, memory allocation, rendering performance, and network activity. The simplicity of this process was facilitated by the fact that my implementation resided within a single file.

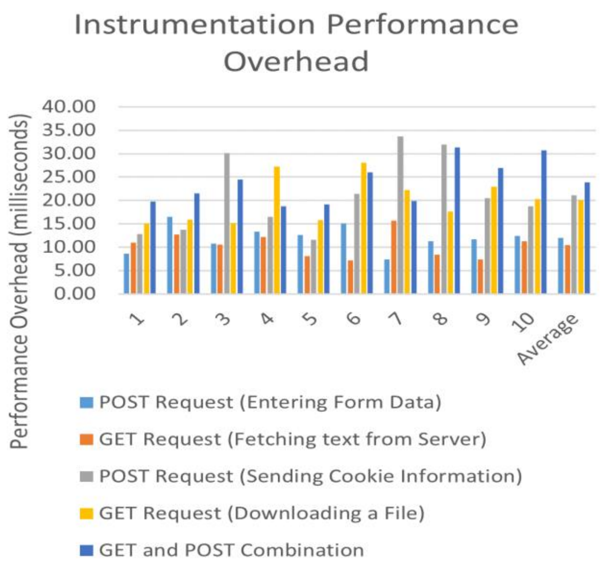


Figure 9. Performance Metrics

To quantify the extra overhead a straightforward approach was adopted where the performance metrics of functions that were invoked during a browser session contained within "instrumet.js" were examined. Figure 9 illustrates the outcomes derived from this performance testing. A series of ten tests were conducted for each use case on the mock webpage. Average execution time for all scripts was then calculated. The results of this testing revealed the following average overhead generated by my instrumentation:

- Inputting data into a form and performing a POST request: 11.97ms
- Fetching text from a server and performing a GET request: 10.46ms
- Sending information via a POST request: 21.09ms
- Downloading a file through a GET request: 20.03ms
- Performing both a GET request and a POST request: 23.85ms

When consolidating these averages, my implementation introduced an average overhead of 17.48ms on my mock webpage. It is important to note that this value may vary when testing on Chrome using different machines, as each device has varying amounts of available RAM for the browser's use.

This result is highly favourable since Google recommends an average response time of under two hundred milliseconds to create the perception of an instant response [66]. The instrumentation we have implemented typically introduces a mere twenty milliseconds of overhead on average. This amounts to just 10% of the average response time recommended by Google, indicating that my software falls comfortably within the range of desirable response times. The percentage contribution when evaluating the mock webpage, typically fell within the range of 0.2% to 0.5%. On average, this value is less than one percent, indicating that it has a minimal impact on the overall program performance.

C. Evaluation

In summary, confidence is held that the project goals initially set out in this report have been successfully achieved. These goals revolved around exposing end-user data by creating software solutions tailored to instrumenting the Document Object Model (DOM) of web applications. This instrumentation was specifically designed to reveal provenance data when changes in the DOM occurred due to Ajax requests.

This has been achieved by implementing DOM instrumentation, which proved to be effective in consistently capturing and displaying provenance data whenever the DOM was manipulated. This capability ensured that changes were monitored and recorded accurately, guaranteeing the reliable capture of provenance data. Moreover, user-friendliness was prioritised in the design of these solutions, making them intuitive and convenient for both developers and end-users to work with. Furthermore, a focus was placed on optimising the software's performance to ensure it maintained a lightweight profile and met acceptable performance benchmarks.

Throughout this project, the Rational Unified Process (RUP) methodology was followed, which played a pivotal role in helping attain the project objectives.

VI. Conclusion and Future Work

Throughout this project, the focus was on the development of software solutions aimed at instrumenting the DOM of web applications and presenting provenance data to end users. This endeavor aimed to enhance the transparency and traceability of data within web applications, particularly for one-page applications using Ajax. Through the manipulation of the DOM via instrumentation, a method was devised to exhibit provenance data to users as they interacted with specific elements on a webpage. Three different approaches to address this challenge were pursued: a browser plugin solution, a JavaScript instrumentation solution within browser code, and a framework plugin solution. Throughout the development process, Rational Unified Process (RUP) was adhered to, which helped ensure the quality and usability of the software [27]. To evaluate the quality of my software, criteria such as runtime overhead, installation complexity, development costs, and alignment with project requirements were considered. Through adherence to these criteria, it was guaranteed that the software satisfied the ISO9126 characteristics of Functionality, Reliability, Usability, Efficiency, Maintainability, and Portability as the final implementation met all these broad aspects [28].

While this project has achieved its primary objectives, there remain opportunities for future work and enhancements. Expanding the compatibility of the browser plugin and JavaScript instrumentation solutions to work seamlessly across various web browsers would make the software more accessible to a wider user base. Instrumenting HTTP calls other than Ajax to cover more use cases. Once the Server-Side Provenance software is developed by veracity [25], explore how effectively it works with this solution, making changes when needed.
By addressing these future work areas, the usability, reliability, and effectiveness of our DOM instrumentation solutions can be enhanced, providing users with a valuable tool for understanding data provenance in web applications.

VII. References

[1] "General Data Protection Regulation (GDPR) Compliance Guidelines," GDPR.EU [Online]. Available: https://gdpr.eu. [Accessed 4 April 2023].

[2] "Art. 4 GDPR Definitions," Intersoft Consulting, [Online]. Available: https://gdpr info.eu/art-4-gdpr/. [Accessed 31 May 2023].

[3] "1.2 billion euro fine for Facebook as a result of EDPB binding decision," EDPB, 22 May 2023. [Online]. Available: https://edpb.europa.eu/news/news/2023/12-billion-euro-fine-facebook-result-edpb-binding-decision_en. [Accessed 31 May 2023].

[4] "Importance of Data Provenance in Scientific Research," Washington.edu, [Onlin Available: https://faculty.washington.edu/hazeline/ProvEco/generic2.html. [Accessed 28 March 2023].

[5] "How personalized ads work," Google, [Online]. Available: https://support.google.com/My-Ad-Center-Help/answer/12155656?hl=en. [Accessed 28 March 2023].

[6] J. Robie, "What is the Document Object Model?," Texcel Research, [Online]. Available: https://www.w3.org/TR/WD-DOM/introduction.html. [Accessed 2023 03 20].

[7] W3C, "W3C," [Online]. Available: https://www.w3.org. [Accessed 11 October 2023].

[8] "JavaScript HTML DOM," w3schools, [Online]. Available: https://www.w3schools.com/js/js_htmldom.asp. [Accessed 29 May 2023].

[9] "DOM manipulation in JavaScript," Scaler Topics, [Online]. Available: https://www.scaler.com/topics/javascript-dom-manipulation/. [Accessed 29 May 2023].

[10] T. D. Huynh, M. Ebden, J. Fischer, S. Roberts and L. Moreau, "Provenance Network Analytics," 15 February 2018. [Online]. Available: https://link.springer.com/content/pdf/10.1007/s10618-017-0549-3.pdf. [Accessed May 2023].

[11] B. Plale, D. Gannon and L. Y. Simmhan, "A Survey of Data Provennace in e-science," 3 September 2005. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/1084805.1084812. [Accessed 29 May 2023].

[12] "HTTP Vs HTTPS | What is HTTPS? | What does HTTPS mean?," cWatch, 9 November 2022. [Online]. Available: https://cwatch.comodo.com/blog/website-security/what-is-https-and-why-switching-to-https/. [Accessed 29 May 2023].

[13] W. J. Buchanan, S. Helme and A. Woodward, "Analysis of the adoption of securi headers," 27 September 2017. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/pdfdirect/10.1049/iet-ifs.2016.062 [Accessed 29 May 2023].

[14] MDN, "Location," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Location. [Accessed 11 October 2023].

[15] T. Kempf, K. Karuri and L. Gao, "Software Instrumentation," 2008. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/9780470050118.ecse386. [Accessed 29 May 2023].

[16] K. Harukal, Y. Dachuan, C. Ajay, I. Hiroshi and S. Igor, "JavaScript Instrumentation in Practice," 2008. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-89330-1_23. [Accessed 29 May 2023].

[17] Codilime, "Top 7 code coverage tools for Java," 10 March 2023. [Online]. Available: https://codilime.com/blog/code-coverage-tools-for-java/. [Accessed 11 October 2023].

[18] "manifest.json," MDN, [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json. [Accessed 30 May 202

[19] A. Barth, A. P. Felt and P. Saxena, "Protecting Browsers from Extension Vulnerabilities," [Online]. Available: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/38394.pdf. [Accessed 30 May 2023].

[20] P. M, "JavaScript DOM Manipulation Performance : Comparing Vanilla JavaScri and Leading JavaScript Front-end Frameworks," 2020. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1436661/FULLTEXT01.pdf. [Accessed 29 May 2023].

[21] I. M. Yazici and M. S. Aktas, "A novel visualization approach for data provenanc 24 June 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/pdfdirect/10.1002/cpe.6523. [Accessed 29 Ma 2023].

[22] "From JavaScript to React," Next.js, [Online]. Available: https://nextjs.org/learn/foundations/from-javascript-to-react. [Accessed 31 May 2023].

[23] "What is AJAX," w3schools, [Online]. Available: https://www.w3schools.com/whatis/whatis_ajax.asp. [Accessed 28 March 2023].

[24] K. Torsten, K. Kingshuk and G. Lei, "Software Instrumentation," Wiley Online Library, 15 February 2008. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470050118.ecse386. [Accessed 20 March 2023].

[25] Veracity Lab, "Veracity Lab," [Online]. Available: https://veracity.wgtn.ac.nz. [Accessed 11 October 2023].

[26] M. G. M. L.-R. J. Dietrich, "On Retrofitting Provenance for Transparent and Fair Software - Drivers and Challenges," in *2023 IEEE/ACM International Workshop Equitable Data & Technology (FairWare)*, Melbourne, Australia, 2023.

[27] K. Joydip, "Introduction to Rational Unified Process (RUP)," Developer.com, 7 December 2022. [Online]. Available: https://www.developer.com/project-management/rational-unified-process-rup/. [Accessed 6 April 2023].

[28] "ISO/IEC 9126 in Software Engineering," Geeksforgeeks, 8 December 2022. [Online]. Available: https://www.geeksforgeeks.org/iso-iec-9126-in-software-engineering/. [Accessed 6 April 2023].

[29] "Most Popular Web Browsers in 2023," Oberlo, [Online]. Available: https://www.oberlo.com/statistics/browser-market-share. [Accessed 23 May 2023

[30] T. Alvaro, "15 Best Chromium Browsers 2023," Alvaro Trigo, [Online]. Availabl https://alvarotrigo.com/blog/best-chromium-browsers/. [Accessed 23 May 2023].

[31] D. Nield, "Which Browser Engine Powers Your Web Browsing and Why Does It Matter?," Gizmodo, 4 August 2022. [Online]. Available: https://www.gizmodo.com.au/2022/08/which-browser-engine-powers-your-web-browsingand-why-does-it-matter/. [Accessed 31 May 2023].

[32] Chrome for developers, "Architecture overview," 18 September 2012. [Online]. Available: https://developer.chrome.com/docs/extensions/mv3/architecture-overview/. [Accessed 5 10 2023].

[33] J. Gulab, "Browser Plugin," GitLab, [Online]. Available: https://gitlab.ecs.vuw.ac.nz/course-work/project489/2023/gulabjaye/dom-instrumentation-to-display-provenance-data/-/tree/main/Browser%20Plugin/Browser%20Plugin%20Injection%20Test. [Acces 23 May 2023].

[34] J. Gulab, "Intercept Headers Test," [Online]. Available: https://gitlab.ecs.vuw.ac.nz/course-work/project489/2023/gulabjaye/dom-instrumentation-to-display-provenance-data/-/tree/main/Browser%20Plugin/Intercept%20Headers%20Test. [Accessed 5 Octob 2023].

[35] J. Gulab, "Display Headers Test," [Online]. Available: https://gitlab.ecs.vuw.ac.nz/course-work/project489/2023/gulabjaye/dom-instrumentation-to-display-provenance-data/-/tree/main/Browser%20Plugin/Display%20Headers%20Test. [Accessed 5 Octobe 2023].

[36] J. Gulab, "DOM Manipulation Detector," [Online]. Available: https://gitlab.ecs.vuw.ac.nz/course-work/project489/2023/gulabjaye/dom-instrumentation-to-display-provenance-data/-/tree/main/Browser%20Plugin/DOM%20Manipulation%20Detector. [Accessed 5 October 2023].

[37] J. Gulab, "Test Website," [Online]. Available: https://gitlab.ecs.vuw.ac.nz/course-work/project489/2023/gulabjaye/dom-instrumentation-to-display-provenance-dat /tree/main/Browser%20Plugin/Test%20Website. [Accessed 5 October 2023].

[38] Chrome, "Stay secure," [Online]. Available: https://developer.chrome.com/docs/extensions/mv3/security/. [Accessed 5 Octobe 2023].

[39] MDN, "Content Security Policy (CSP)," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP. [Accessed 11 Octobe 2023].

[40] I. Okechukwu, "A light intro to instrumentation on the web frontend," Medium, 2 January 2022. [Online]. Available: https://isocroft.medium.com/a-light-intro-to-instrumentation-on-the-web-frontend-27a12e1e965a. [Accessed 21 March 2023].

[41] MDN, "MutationObserver," [Online]. Available: https://developer.mozilla.org/en US/docs/Web/API/MutationObserver. [Accessed 5 October 2023].

[42] J. Gulab, "Detect changes in DOM Elements," [Online]. Available: https://gitlab.ecs.vuw.ac.nz/course-work/project489/2023/gulabjaye/dom-instrumentation-to-display-provenance-data/-/tree/main/JavaScript%20Instrumentation/Detect%20changes%20in%20DOM%2 lements. [Accessed 5 October 2023].

[43] DigitalOcean, "Observer Design Pattern in Java," 4 August 2022. [Online]. Available: https://www.digitalocean.com/community/tutorials/observer-design-pattern-in-java. [Accessed 11 October 2023].

[44] "How is it possible that a <script> tag was injected, but not executed?," [Online]. Available: https://security.stackexchange.com/questions/240353/how-is-it-possib that-a-script-tag-was-injected-but-not-executed. [Accessed 11 October 2023].

[45] J. Gulab, "Test Website," [Online]. Available: https://gitlab.ecs.vuw.ac.nz/course-work/project489/2023/gulabjaye/dom-instrumentation-to-display-provenance-dat /blob/main/JavaScript%20Instrumentation/Detect%20changes%20in%20DOM% Elements/index.html. [Accessed 5 October 2023].

[46] C. Nicholas, "Aspect-Oriented Programming in JavaScript," 29 November 2021. [Online]. Available: https://www.ctnicholas.dev/notes/aspect-oriented-programming-in-javascript. [Accessed 5 July 2023].

[47] K. Gregor, L. John, M. Anurag, M. Chris, L. Cristina, L. Jean Marc and I. John, "Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds) ECOOP'97 — Object-Oriented Programming," in *ECOOP 1997. Lecture Notes in Computer Science, vol 1241*, Berlin, Heidelberg, 1997.

[48] J. Gulab, "Aspect-Orientated Approach Example," [Online]. Available: https://gitlab.ecs.vuw.ac.nz/course-work/project489/2023/gulabjaye/dom-instrumentation-to-display-provenance-data/-/tree/main/JavaScript%20Instrumentation/Aspect-Orientated%20Approach%20Example?ref_type=heads. [Accessed 6 October 202

[49] Spring, "Aspect Oriented Programming with Spring," [Online]. Available: https://docs.spring.io/spring-framework/docs/3.0.x/spring-framework-reference/html/aop.html. [Accessed 11 October 2023].

[50] H. P. Singh, "Aspect Oriented Programming," 17 December 2022. [Online]. Available: https://medium.com/hprog99/aspect-oriented-programming-b9a06ca256db. [Accessed 6 October 2023].

[51] StackOverflow, "https://stackoverflow.com/questions/10273309/need-to-hook-in a-javascript-function-call-any-way-to-do-this," [Online]. Available: https://stackoverflow.com/questions/10273309/need-to-hook-into-a-javascript-function-call-any-way-to-do-this. [Accessed 6 October 2023].

[52] MDN, "Using XMLHttpRequest," [Online]. Available: developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest. [Accessed 6 October 2023].

[53] L. Eggleston, "A Guide to Low Level Programming for Beginners," [Online]. Available: https://www.coursereport.com/blog/a-guide-to-low-level-programming for-beginners. [Accessed 11 October 2023].

[54] J. Gulab, "Instrumentation through Proxying," [Online]. Available: https://gitlab.ecs.vuw.ac.nz/course-work/project489/2023/gulabjaye/dom-instrumentation-to-display-provenance-data/-/tree/main/JavaScript%20Instrumentation/Instrumentation%20through%20Proxyi ?ref_type=heads. [Accessed 6 October 2023].

[55] MDN, "Using server-sent events," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events. [Accessed 6 October 2023].

[56] J. Gulab, "Instrumentation Proxying (SSE)," [Online]. Available: https://gitlab.ecs.vuw.ac.nz/course-work/project489/2023/gulabjaye/dom-instrumentation-to-display-provenance-data/-/tree/main/JavaScript%20Instrumentation/Instrumentation%20Proxying%20(SSE ef_type=heads. [Accessed 6 October 2023].

[57] J. Olawanle, "What is a Framework? Software Frameworks Definition," 6 September 2022. [Online]. Available: https://www.freecodecamp.org/news/what-a-framework-software-frameworks-definition/. [Accessed 7 October 2023].

[58] N. Raval, "React vs Angular: Which JS Framework to Pick for Front-end Development?," 3 July 2023. [Online]. Available: https://radixweb.com/blog/reac vs-angular. [Accessed 7 October 2023].

[59] M. Mandal, "How to use jQuery in your React App | by Manish Mandal | How To React | Medium," 26 April 2020. [Online]. Available: https://medium.com/how-to react/how-to-use-jquery-in-your-react-app-b425727a32fd. [Accessed 6 October 2023].

[60] NPM, [Online]. Available: https://www.npmjs.com. [Accessed 7 October 2023].

[61] J. Gulab, "Framework Plugin," [Online]. Available: https://gitlab.ecs.vuw.ac.nz/course-work/project489/2023/gulabjaye/dom-instrumentation-to-display-provenance-data/-/tree/main/Framework%20Plugin?ref_type=heads. [Accessed 7 October 2023].

[62] J. Gulab, "Instrumentation Code," [Online]. Available: https://gitlab.ecs.vuw.ac.nz/course-work/project489/2023/gulabjaye/dom-instrumentation-to-display-provenance-data/-/tree/main/Instrumentation%20Code?ref_type=heads. [Accessed 10 October 202

[63] Geeksforgeeks, "How to convert JSON data to a html table using JavaScript/jQue ?," [Online]. Available: https://www.geeksforgeeks.org/how-to-convert-json-data to-a-html-table-using-javascript-jquery/. [Accessed 9 October 2023].

[64] J. Gulab, "Use Case Webpage," [Online]. Available: https://gitlab.ecs.vuw.ac.nz/course-work/project489/2023/gulabjaye/dom-instrumentation-to-display-provenance-data/-/tree/main/Use%20Case%20Webpage/News%20Webpage?ref_type=heads. [Accessed 9 October 2023].

[65] J. Gulab, "Performance Testing," [Online]. Available: https://gitlab.ecs.vuw.ac.nz/course-work/project489/2023/gulabjaye/dom-instrumentation-to-display-provenance-data/-/tree/main/Use%20Case%20Webpage/Performance%20Testing?ref_type=heads. [Accessed 9 October 2023].

[66] Sematext, "What Is Response Time & How to Reduce It - Sematext," [Online]. Available: https://sematext.com/glossary/response-time/. [Accessed 9 October 2023].

[67] "Data Provenance," National Library of Medicine, [Online]. Available: https://www.nnlm.gov/guides/data-glossary/data-provenance#:~:text=Definition,to%20where%20it%20is%20presently.. [Accesse 20 March 2023].