iTrace-Chrome: Enabling Eye Tracking in the Browser for Software Engineering Research

Elijah Rohrs
University of Nebraska–Lincoln
Lincoln, Nebraska, USA
erohrs2@huskers.unl.edu

Zachary Kozak
University of Nebraska–Lincoln
Lincoln, Nebraska, USA
zkozak2@unl.edu

Kang-il Park
University of Nebraska–Lincoln
Lincoln, Nebraska, USA
kangil.park@huskers.unl.edu

Joshua A.C. Behler
Kent State University
Kent, Ohio, USA
jbehler1@kent.edu

Michael J. Decker
Bowling Green State University
Bowling Green, Ohio, USA
mdecke@bgsu.edu

Jonathan I. Maletic
Kent State University
Kent, Ohio, USA
jmaletic@kent.edu

Bonita Sharif
University of Nebraska–Lincoln
Lincoln, Nebraska, USA
bsharif@unl.edu

***Abstract*—The use of eye tracking in software engineering studies has increased steadily in recent years. Most eye-tracking empirical studies conducted on source code are typically shown within an integrated development environment (IDE) such as Eclipse, IntelliJ, or PyCharm. iTrace is community eye-tracking infrastructure that enables researchers to conduct realistic eye-tracking studies within an IDE. It provides a suite of tools to gather and process eye-tracking data on software artifacts that can later be used towards some functional goal. In a realistic setting, software developers work with a variety of artifacts besides source code. Source code and many of these artifacts are accessible via a web browser (e.g., an issue-tracker, online IDEs). Eye tracking within the web browser is inherently limited, even with state-of-the-art eye-tracking vendor software. This paper presents iTrace-Chrome, a highly scalable Google Chrome extension for the iTrace infrastructure supporting fine-grained eye tracking in a web browser. The architecture of iTrace-Chrome is presented along with two usage scenarios and guidelines on how software engineering researchers can use and adapt the plugin in their own research to study source code and related artifacts.**

***Index Terms*—eye tracking, software engineering, program comprehension, Stack Overflow, GitHub, Bugzilla, empirical studies**

I. Introduction

Software engineering researchers use eye-tracking equipment to study developer comprehension in source code with respect to certain tasks [1]–[5]. Researchers also use eye-tracking data in larger research areas, such as neural code summarization [6] and attention-based traceability link recovery [7]. Typically, eye-tracking studies are limited to a single fixed-size artifact (i.e., a single stimuli of one screen/page) and thus, subjects are unable to scroll within an artifact or switch between them and reliably maintain the collection of accurate eye tracking data. iTrace [8] is a community eye-tracking infrastructure that eliminates this limitation in a scalable fashion, unlike existing ad hoc solutions [9], [10].

iTrace [8] consists of a core engine, that implements the interface to eye-tracking hardware, and a set of plugins to Integrated Development Environments (IDEs). The plugins work with and utilize the iTrace-Core (eye-tracking engine interface) to allow developers to scroll and switch between different software artifacts (e.g., source files, text documents, etc.). However, software development activities are not solely limited to working in an IDE. It is common for a software developer to move between the IDE, Stack Overflow, bug reports, issue trackers, and other software artifacts during a typical coding session. For example, a developer might search for solutions to coding problems via a search engine (e.g., Google Search) or community forum (e.g., Stack Overflow), or they make use of other web-based artifacts such as an issue-tracker (e.g., GitHub Issue List). Many of these artifacts are available via web applications accessible through a web browser.

Existing approaches for eye tracking in browsers [9]–[11] suffer from several key limitations that hinder their use in realistic software engineering studies. First, many tools rely on static or pre-captured representations of web pages, assuming that content does not change during interaction. This makes them unsuitable for modern web applications where layouts dynamically change (e.g., pop-ups, collapsing sections, or asynchronous content loading). Second, several tools require manual specification of Areas of Interest (AOIs)—often through CSS selectors or post hoc annotation—which is both labor-intensive and brittle across different page structures. Third, prior systems are typically restricted to the browser alone, lacking integration with development environments and thus failing to capture the multi-artifact workflows that characterize real-world software development. Finally, some solutions are tightly coupled to specific hardware or architectures, limiting extensibility and broader adoption.

iTrace-Chrome[1] is designed to directly address these limitations. Unlike approaches that rely on static page representations, iTrace-Chrome operates on the live Document Object Model (DOM), enabling robust tracking on dynamic, evolving web pages. Rather than requiring manual AOI specification, it uses element descriptors to automatically detect AOIs in

[1]https://github.com/iTrace-Dev/iTrace-Chrome

real time, improving both accuracy and usability. In contrast to browser-only tools, iTrace-Chrome is integrated within the broader iTrace infrastructure, allowing simultaneous tracking across IDEs, text editors, and web browsers, thereby supporting realistic developer workflows. Furthermore, its modular architecture enables easy extension to new web pages through content scripts without modifying the underlying tracking engine. Together, these capabilities enable more flexible, scalable, and ecologically valid eye-tracking studies than previously possible.

II. Related Work

In this section, we present related work done on eye tracking in the browser. Dunagan and Sunshine detailed current frameworks for conducting software engineering studies in the browser, including experiment creation tools like jsPsych [12]. Eye tracking for such tools is limited to webcam-based data and cannot be extended outside a psychology-based study framework; it cannot be used to collect data on GitHub or Stack Overflow, for example. Other tools for analyzing web engagement focus on visualization rather than defining Areas of Interest (AOI) [13], [14], or they require manually marking AOIs post hoc on an image of the website [15]. Next, we present three tools closely related to iTrace-Chrome.

The work by Röhrl et al. is closest to our approach. They introduce WebGazeTrack [10], a Google Chrome extension able to track Document Object Model (DOM) changes and utilize a Tobii Pro Fusion eye tracker without the need of additional software, allowing the entirety of an eye-tracking study to run within the browser, including calibration. For user interaction studies, this tool still requires manually defining user events and AOIs using CSS selectors. As popular software moves to the browser, including web-based IDEs, this work shows the importance of enabling browser-based eye tracking for software engineering research. However, WebGazeTrack is not open source and tightly couples the eye-tracking logic with the semantic logic in the browser which limits reproducibility and adoption. We cannot test the tool (at time of writing) as it is not available online. Adding support for new trackers involves significant changes to the architecture. It may require custom-built applications depending on the usage scenario. It is also specific to the Tobii Pro Fusion hardware.

eyeScrollR [11], introduced by Larigaldie et al., is a related approach that maps eye-tracking data to scrollable web content using an R-based framework. While effective for analyzing gaze data on web pages, it assumes that the page remains static in structure and length during user interaction. This assumption limits its applicability to modern web environments, where elements frequently change—such as dynamic content loading, pop-ups, or layout adjustments. Because eyeScrollR relies on a pre-captured full-page image to map gaze data to Areas of Interest (AOIs), it cannot accurately accommodate such changes. In contrast, iTrace-Chrome overcomes this limitation by operating directly on the Document Object Model (DOM) and collecting data in real time. By using element descriptors to define AOIs, it can precisely determine whether a user is viewing an AOI even on highly dynamic web pages.

PosEyeDOM [9] is a Chrome extension capable of gathering eye tracking data and performing analysis on AI generated code suggestions within the web based VS Code editor. PosEyeDOM as a tool is created specifically for a specific eye tracker by SR Research and uses their WebLink software to record web browser data on AI code suggestions, while iTrace Chrome is built to support any webpage and is not specific tracker dependent nor is it dependent on vendor software.

iTrace-Chrome addresses gaps of accessibility and support by being open source, not dependent on commercial vendor software, and built for an existing community infrastructure for conducting eye-tracking studies that supports many common input sources such as code, text, Stack Overflow posts, GitHub issues and easily customizable to include other artifacts [8]. Unlike WebGazeTrack, iTrace-Chrome is designed to be modular separating the tracking engine from the different plugins it supports. Extending iTrace is easier because developers only need to modify what they need to extend such as support for a new plugin or support for a new eye tracker with the rest remaining intact. A comparative summary is provided in Table I to highlight the differences of iTrace-Chrome with existing tools.

Extending the iTrace [8] framework with the Google Chrome plugin has already proven useful for conducting software engineering studies [16], [17] described in more detail in Section IV.

III. iTrace-Chrome Architecture

iTrace[2] currently supports Eclipse, Visual Studio, JetBrains' IDEs, Atom, Sublime, Notepad++, and now Google Chrome. Additionally, it supports all Tobii Pro eye trackers, GazePoint, and SmartEye eye trackers. Besides this core functionality, it supports post-processing via iTrace-Toolkit [18] and iTrace-Visualize [19] with support for screen recording via iTrace-ScreenRecord[3]. More details are provided on our downloads page at https://www.i-trace.org.

iTrace-Chrome is implemented as a plugin[4] for the Google Chrome web browser (most recent version as of this writing) using Manifest V3. To function properly, it must be connected to a running iTrace-Core instance. There are three major components of iTrace-Chrome: The user interface, the content scripts, and the service worker. See Figure 1 for an overview of the architecture. A video demonstration[5] of iTrace-Chrome is available on our YouTube channel. Besides being available on our website for immediate download and use, iTrace-Chrome is currently in the process of being released on the Chrome Web Store.

iTrace-Core connects to an eye tracker and communicates with the plugin (iTrace-Chrome) by sending it the raw gaze coordinates of a developer as it receives them. The eye tracker

[2]https://www.i-trace.org/

[3]https://github.com/iTrace-Dev/iTrace-ScreenRecording

[4]https://github.com/iTrace-Dev/iTrace-Chrome

[5]iTrace-Chrome Video Demo: https://youtu.be/X_ksp9QOwYk

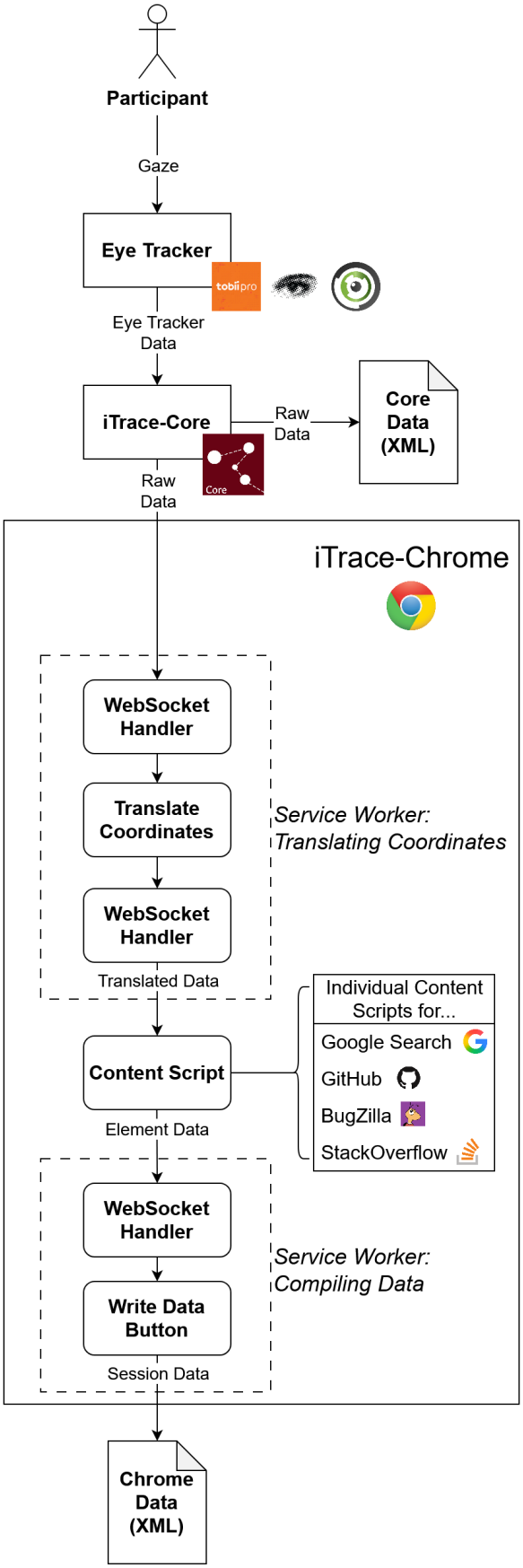

Fig. 1: An architecture overview of iTrace-Chrome

calibration is done via the iTrace-Core interface. It also writes the raw data gaze points, validity codes, and calibration data into an XML file (referred to as Core Data (XML) in Figure 1). iTrace-Chrome receives these coordinates via a web socket through a service worker and, based on the page being viewed, processes the data using a content script. The content scripts produce HTML element data that the raw gaze coordinates map to. The purpose of each of these areas, how they function, and the components that they contain are described below.

Fig. 2: iTrace-Chrome Browser Interface

A. User Interface

iTrace-Chrome's user interface consists of a single pop-up element accessible via Google Chrome's extension manager. This pop-up serves three primary purposes: to allow the user to initiate the extension's connection to iTrace-Core, to show the current status of the extension's connection to iTrace-Core, and to provide a button for downloading session data after eye-tracking data collection is complete. Figure 2 showcases iTrace-Chrome's popup UI.

B. The Service Worker

In Google Chrome extensions, service workers function as the primary means of handling events. iTrace-Chrome's service worker is responsible for handling the gaze data sent to it by iTrace-Core. The service worker translates the (x,y) pixel screen coordinates from the gaze data to (x,y) local coordinates within the browser window, and then communicates with the applicable content script to receive HTML element data relevant to the script's specified areas of interest. Areas of interest in the context of iTrace-Chrome are specified elements on a web-page, such as the title of a post, a hyperlink button, or a section relevant to ratings. After receiving the element data, the service worker then stores it until the session is complete. The data can then be downloaded as a single XML file (referred to as Chrome Data (XML) in Figure 1).

C. The Content Scripts

The content scripts are a collection of files responsible for identifying which element the user is currently looking at on a given web-page based on an input of (x, y) screen coordinates and whether that element is an area of interest. These content scripts are page-specific, as web pages may have very different layouts and use different structures for information that is significant. Currently, iTrace-Chrome supports the following web pages: 1) Google Search results pages, 2) Stack Overflow Questions pages, 3) Stack Overflow Search pages, 4) GitHub Issue List pages, 5) GitHub Profile pages, 6) GitHub PR pages, 7) GitHub PR List pages, and 8) Bugzilla Bug pages.

TABLE I: Comparison of iTrace-Chrome with Related Eye-Tracking Tools

Criteria	**iTrace-Chrome**	**WebGazeTrack** [10]	**eyeScrollR** [11]	**PosEyeDOM** [9]
Stimulus Type	Dynamic, live web pages (DOM-based)	Dynamic web pages but monolith solution	Scrollable but assumes static page	*://*.github.dev/* Sites only
Dynamic Content Handling	Fully supports dynamic changes	Tracks DOM changes	Assumes no structural changes	Limited to specific contexts and integrated with SR-Research WebLink
AOI Definition	Automatic via DOM descriptors	Manual (CSS selectors)	Image-based AOIs	Predefined for AI features
Real-Time Processing	Yes	Yes	No (post hoc)	Yes
Integration Scope	IDE + browser + text editors	Browser-only	Browser-only	Browser-only
Extensibility	Modular via content scripts	Limited / tightly coupled	Limited	Narrow (AI-specific),Tied to vendor software and tracker.
Hardware Support	Multiple tracker support (Tobii, SmartEye, GazePoint)	Limited (e.g., Tobii Fusion)	Depends on input data	Tied to SR Research EyeLink Portable Duo eye tracker
Scalability	Incremental via page scripts	General but less customizable	Limited by assumptions	Narrow domain scalability
Use Case Focus	Realistic developer workflows	General web tracking	Scrollable page analysis	AI code suggestion studies

Figure 3 shows an example of areas of interest on a Stack Overflow page. These areas of interest are automatically mapped to the translated gaze coordinates by the content scripts for each page. In this example, we see the automatically detected areas of interest for Question Title, Question Vote, Question Code, Question Text, and Question Image. The generated XML file includes unique tags for each of these areas, which can later be analyzed by researchers. One point to note is that these bounding boxes are calculated in real-time and do not need to be determined a priori.

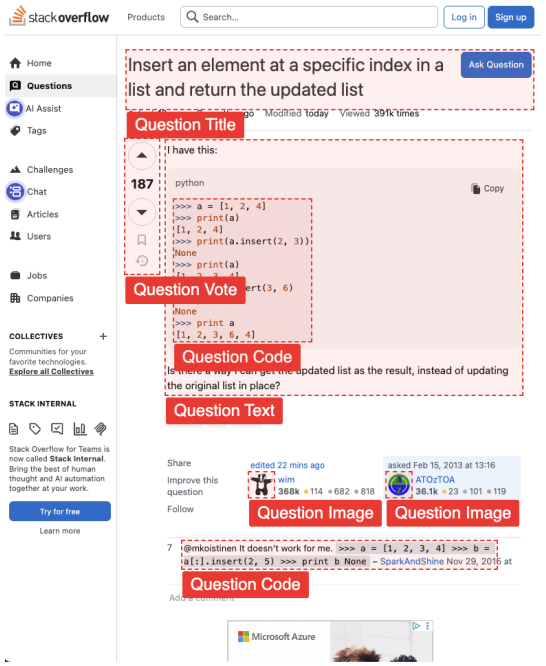

Fig. 3: Areas of interest for Stack Overflow *Title, Vote, Code, Text, Image* as identified by Stack Overflow Content Script

D. Extending iTrace-Chrome

iTrace-Chrome's data processing flow is summarized in Figure 1. iTrace-Chrome is extensible, allowing the support for additional websites beyond the ones listed previously. Extending iTrace-Chrome to support a new web-page involves creating a content script. Content scripts follow a set pattern that must have their areas of interest specifically tailored to a page. Areas of interest are defined by first identifying the HTML element structure of a target page feature and adding that specific structure as an area of interest in the content script. The element's structure includes the element's id, class, and/or tag name. After the content script contains all the desired areas of interest, it is added to the extension's manifest file's content script section and the `webSocketHandler` function inside the service worker. This function is responsible for determining which content scripts are communicated with during eye-tracking. Completing these steps results in a functional content script, allowing the addition of support for new web-pages in iTrace-Chrome.

E. Processing Eye-Tracking Data for Analysis

The two XML files (one from iTrace-Core and the other from iTrace-Chrome) are joined on the Event ID's where the eye-tracking specific data is combined with semantic data from web pages. After this step, fixation event detection algorithms are applied to the data to identify fixations and saccades. iTrace-Toolkit [18] has the I-DT and I-VT fixation detection algorithms [20] implemented for IDE-based plugins. Due to the ad-hoc nature of web-pages, this detection is left up to the researchers to implement after data collection from iTrace-Chrome is complete. The two XML output files contain all the necessary information to generate other higher-order eye-tracking metrics [21] such as linearity metrics, regressions, saccade length, and various derived measures [22], [23]. These are typically defined by researchers based on specific research questions of the study.

IV. Usage Scenarios

To demonstrate the use of iTrace-Chrome in research studies, we describe two representative studies that use it to investigate developer behavior on software repository platforms.

A. Studying Developer Decisions on GitHub

Wiese et al. conducted a pilot study that looks at what pieces of information developers use to prioritize what issues and pull requests to work on, what information is used to

assess a potential pull request, and what characteristics of a profile have a higher likelihood of getting their pull request accepted [16]. Using iTrace-Chrome to collect eye-tracking data, their results show that developers spend the most time on the title of a pull request to rank its importance; comments and code elements gather the most attention with up to 40% of time spent when deciding to accept or approve pull requests, and the contribution heat map and language of repositories contributed to were the most important factor in judging a GitHub profile. A replication package for this study, along with all post-processing scripts for Chrome data, is available at https://doi.org/10.5281/zenodo.6402728.

B. Studying GitHub Conversations on Pull Requests for Sentiment Analysis

In addition to the general comment and code elements as AOIs, iTrace-Chrome can also investigate time spent on smaller sub-elements within an element. Park et al. conducted a study using iTrace-Chrome that examines conversations on GitHub pull requests to understand the effect of using emojis [17]. They compared the results of each conversation run through several sentiment analysis tools against human participants and what elements of the pull request page developers give their most attention. Their results show that while participants only rate 5% of pull requests as having negative sentiment, sentiment analysis tools rate more pull requests as negative, with up to 71% of pull requests rated as negative by Stanford CoreNLP, a tool that neither takes emoji into account nor is trained in the domain of software engineering. In addition, eye-tracking data shows that far more attention is given to emojis than plain text, suggesting that emojis influence the perceived sentiment in developer communication. We refer the reader to the replication package for this study at https://doi.org/10.5281/zenodo.4602631.

V. Discussion and Impact

Eye tracking plays a key role in studying how developers interact with code and its related artifacts. iTrace[6] is currently the only eye-tracking infrastructure that provides a scalable Chrome extension designed to incorporate additional websites. iTrace-Chrome advances the current state of the art of collecting eye tracking data on the web. It has been assessed in realistic studies and has shown clear tangible results when studying developers interacting with web-based tools such as GitHub profiles and many other GitHub views.

iTrace-Chrome is not limited to fixed-size stimuli; instead, it overcomes this limitation by operating directly on the website's underlying structure (Document Object Model). This does not mean that studies must rely on scrolling; rather, it introduces greater flexibility as a methodological choice, allowing researchers to explore a broader range of options and experimental designs. iTrace-Chrome collects and records data in real time and, by using element descriptors to define Areas of Interest (AOIs), can precisely determine when a user is viewing an AOI on a dynamic webpage.

[6]https://www.i-trace.org

If a study is conducted on a live site such as an active GitHub repository or StackOverflow, iTrace-Chrome will record what the site displays dynamically, since iTrace does not use static caches of webpages—unlike other vendor-specific (iMotions or Tobii Pro Lab) and research eye tracking tools [11]. While in Chrome, users can open and switch between multiple tabs; iTrace-Chrome will record data on each tab seamlessly. During a session, a user can open a new tab and navigate to a page that has an associated content script and the page will track the user's gaze.

An additional advantage of integrating iTrace-Chrome into the iTrace infrastructure is that it enables researchers and practitioners to study both code and web-based artifacts within a single unified environment. iTrace supports simultaneous connections with tools such as Eclipse, Visual Studio, JetBrains IDEs, Atom, Sublime, Notepad++, and now, Chrome. This allows researchers to observe scenarios where a developer, for example, works in Visual Studio while concurrently reading documentation in Notepad and browsing web-based resources, like GitHub, in Chrome. Throughout this process, iTrace seamlessly captures events, timing, and plugin-level data across all platforms. Additionally, the iTrace-ScreenRecord plugin allows for recording the entire desktop screen during the process.

Since each website is structured differently, it is impossible to track every website element natively. Hence, each website/application requires the development of custom scripts. Creating such scripts is relatively simple, as described previously in Section III-D. Each supported webpage presents an opportunity to develop a tailored content script—while this introduces some setup effort, it enables more precise customization and scalability as the system evolves.

Finally, it is worth noting that even though the iTrace-Chrome content scripts we create are for software engineering related artifact sites, this method is extensible to support other general-purpose artifact types as well.

VI. Conclusions and Future Work

iTrace-Chrome's main differentiator is its DOM-driven, real-time, multi-platform integration, enabling ecologically valid studies of developer behavior across both code and web artifacts—something that traditional and existing browser-based tools struggle to support simultaneously. The architecture is described along with how researchers can extend it to fit other web applications. Feasibility of use was shown in two studies that have used the plugin towards a functional goal. As part of future work, we plan on extending this functionality to other popular browsers such as Firefox and incorporating other sites such as Reddit, where developers might spend time discussing issues. Additionally, while iTrace-Chrome supports extension through custom content scripts, we aim to broaden the web pages that iTrace-Chrome supports by default. Of particular interest are LLM services like CoPilot and Gemini, as developers are using LLMs more in their workflow.

REFERENCES

[1] U. Obaidellah, M. Al Haek, and P. C.-H. Cheng, "A survey on the usage of eye-tracking in computer programming," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, p. 5, 2018.

[2] N. Mansoor, C. S. Peterson, M. D. Dodd, and B. Sharif, "Assessing the effect of programming language and task type on eye movements of computer science students," *ACM Trans. Comput. Educ.*, vol. 24, no. 1, pp. 2:1–2:38, 2024. [Online]. Available: https://doi.org/10.1145/3632530

[3] J. A. S. da Costa, R. Gheyi, F. Castor, P. R. F. de Oliveira, M. Ribeiro, and B. Fonseca, "Seeing confusion through a new lens: on the impact of atoms of confusion on novices' code comprehension," *Empir. Softw. Eng.*, vol. 28, no. 4, p. 81, 2023. [Online]. Available: https://doi.org/10.1007/s10664-023-10311-0

[4] P. Roberto, R. Gheyi, J. A. S. da Costa, and M. Ribeiro, "Assessing python style guides: An eye-tracking study with novice developers," in *Proceedings of the 38th Brazilian Symposium on Software Engineering, SBES 2024, Curitiba, Brazil, September 30 - October 4, 2024*, I. Steinmacher, S. Marczak, T. E. Colanzi, D. Viana, L. Teixeira, S. R. Vergilio, M. Barcellos, I. Machado, and J. Marques, Eds. SBC, 2024, pp. 136–146. [Online]. Available: https://doi.org/10.5753/sbes.2024.3325

[5] S. Yabesi, M. Amini, J. Ristic, and Z. Sharafi, "Exploring the effects of urgency and reputation in code review: An eye-tracking study," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, ser. ICPC '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 202–213. [Online]. Available: https://doi.org/10.1145/3643916.3644425

[6] A. Bansal, B. Sharif, and C. McMillan, "Towards modeling human attention from eye movements for neural source code summarization," *Proc. ACM Hum. Comput. Interact.*, vol. 7, no. ETRA, pp. 1–19, 2023. [Online]. Available: https://doi.org/10.1145/3591136

[7] B. Sharif, J. Meinken, T. Shaffer, and H. H. Kagdi, "Eye movements in software traceability link recovery," *Empir. Softw. Eng.*, vol. 22, no. 3, pp. 1063–1102, 2017. [Online]. Available: https://doi.org/10.1007/s10664-016-9486-9

[8] D. T. Guarnera, C. A. Bryant, A. Mishra, J. I. Maletic, and B. Sharif, "iTrace: eye tracking infrastructure for development environments," in *Proceedings of the 2018 ACM Symposium on Eye Tracking Research & Applications*. Warsaw Poland: ACM, Jun. 2018, pp. 1–3. [Online]. Available: https://dl.acm.org/doi/10.1145/3204493.3208343

[9] T. Alakmeh, S. D'Angelo, and T. Fritz, "An eye for ai: Eye-tracking the micro-interruptions of genai code suggestions," in *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering, ICSE 2026, Rio de Janeiro, Brasil, April 12-18, 2026*. IEEE Computer Society, 2026. [Online]. Available: https://doi.org/10.1145/3744916.3773132

[10] S. Röhrl, F. Hauser, T. Ezer, L. Grabinger, and P. D. J. Mottok, "WebGazeTrack: A Web-based Eye Tracking Tool," in *Proceedings of the 6th European Conference on Software Engineering Education*, ser. ECSEE '25. New York, NY, USA: Association for Computing Machinery, Jun. 2025, pp. 125–134. [Online]. Available: https://dl.acm.org/doi/10.1145/3723010.3723024

[11] N. Larigaldie, A. Dreneva, and J. Orquin, "eyeScrollR: A software method for reproducible mapping of eye-tracking data from scrollable web pages," *Behavior Research Methods*, vol. 56, no. 4, pp. 3380–3395, Jun. 2024. [Online]. Available: https://doi.org/10.3758/s13428-024-02343-1

[12] L. Dunagan and J. Sunshine, "Conducting Browser-Based User Studies in Software Engineering," Jun. 2022, publisher: Plateau Workshop. [Online]. Available: https://kilthub.cmu.edu/articles/conference_contribution/Conducting_Browser-Based_User_Studies_in_Software_Engineering/19630005/1

[13] S. Zelinskyi and Y. Boyko, "Integrating session recording and eye-tracking: development and evaluation of a Chrome extension for user behavior analysis," *Radioelectronic and Computer Systems*, vol. 2024, no. 3, pp. 38–54, Aug. 2024. [Online]. Available: http://nti.khai.edu/ojs/index.php/reks/article/view/reks.2024.3.03

[14] R. Menges, S. Kramer, S. Hill, M. Nisslmueller, C. Kumar, and S. Staab, "A Visualization Tool for Eye Tracking Data Analysis in the Web," in *ACM Symposium on Eye Tracking Research and Applications*. Stuttgart Germany: ACM, Jun. 2020, pp. 1–5. [Online]. Available: https://dl.acm.org/doi/10.1145/3379156.3391831

[15] A. K. Chinnaswamy, B. A. Sabarish, P. Karhik, P. Karthikeyan, J. P. Maheshvar, and S. Adithya, "Web-Based Task-Driven Eye Tracking: A No-Integrations Approach," *Procedia Computer Science*, vol. 260, pp. 655–664, Jan. 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050925009883

[16] I. S. Wiese, J. Boyer, E. Rasgorshek, G. Pinto, M. Gerosa, I. Steinmacher, and B. Sharif, "How developers make decisions when choosing issues and reviewing code: An eye tracking github study," in *Proceedings of the 2025 Symposium on Eye Tracking Research and Applications*, ser. ETRA '25. New York, NY, USA: Association for Computing Machinery, 2025. [Online]. Available: https://doi.org/10.1145/3715669.3723108

[17] K.-i. Park and B. Sharif, "Assessing perceived sentiment in pull requests with emoji: Evidence from tools and developer eye movements," in *2021 IEEE/ACM Sixth International Workshop on Emotion Awareness in Software Engineering (SEmotion)*, 2021, pp. 1–6.

[18] J. Behler, P. Weston, D. T. Guarnera, B. Sharif, and J. I. Maletic, "itrace-toolkit: A pipeline for analyzing eye-tracking data of software engineering studies," in *45th IEEE/ACM International Conference on Software Engineering: ICSE 2023 Companion Proceedings, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 46–50. [Online]. Available: https://doi.org/10.1109/ICSE-Companion58688.2023.00022

[19] J. Behler, G. Chiudioni, A. Ely, J. Pangonis, B. Sharif, and J. I. Maletic, "itrace-visualize: Visualizing eye-tracking data for software engineering studies," in *IEEE Working Conference on Software Visualization, VISSOFT 2023, Bogotá, Colombia, October 1-2, 2023*. IEEE, 2023, pp. 100–104. [Online]. Available: https://doi.org/10.1109/VISSOFT60811.2023.00021

[20] R. Andersson, L. Larsson, K. Holmqvist, M. Stridh, and M. Nyström, "One algorithm to rule them all? An evaluation and discussion of ten eye movement event-detection algorithms," *Behavior Research Methods*, vol. 49, no. 2, pp. 616–637, 2017. [Online]. Available: https://doi.org/10.3758/s13428-016-0738-9

[21] Z. Sharafi, B. Sharif, Y. Guéhéneuc, A. Begel, R. Bednarik, and M. E. Crosby, "A practical guide on conducting eye tracking studies in software engineering," *Empir. Softw. Eng.*, vol. 25, no. 5, pp. 3128–3174, 2020. [Online]. Available: https://doi.org/10.1007/s10664-020-09829-4

[22] T. Busjahn, R. Bednarik, A. Begel, M. E. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm, "Eye movements in code reading: relaxing the linear order," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015*, A. D. Lucia, C. Bird, and R. Oliveto, Eds. IEEE Computer Society, 2015, pp. 255–265. [Online]. Available: https://doi.org/10.1109/ICPC.2015.36

[23] Z. Sharafi, T. Shaffer, B. Sharif, and Y. Guéhéneuc, "Eye-tracking metrics in software engineering," in *2015 Asia-Pacific Software Engineering Conference, APSEC 2015, New Delhi, India, December 1-4, 2015*, J. Sun, Y. R. Reddy, A. Bahulkar, and A. Pasala, Eds. IEEE Computer Society, 2015, pp. 96–103. [Online]. Available: https://doi.org/10.1109/APSEC.2015.53