Log-it: Supporting Programming with Interactive, Contextual, Structured, and Visual Logs

Peiling Jiang
peiling@ucsd.edu
University of California San Diego
La Jolla, California, USA

Fuling Sun
fulingsun@ucsd.edu
University of California San Diego
La Jolla, California, USA

Haijun Xia
haijunxia@ucsd.edu
University of California San Diego
La Jolla, California, USA

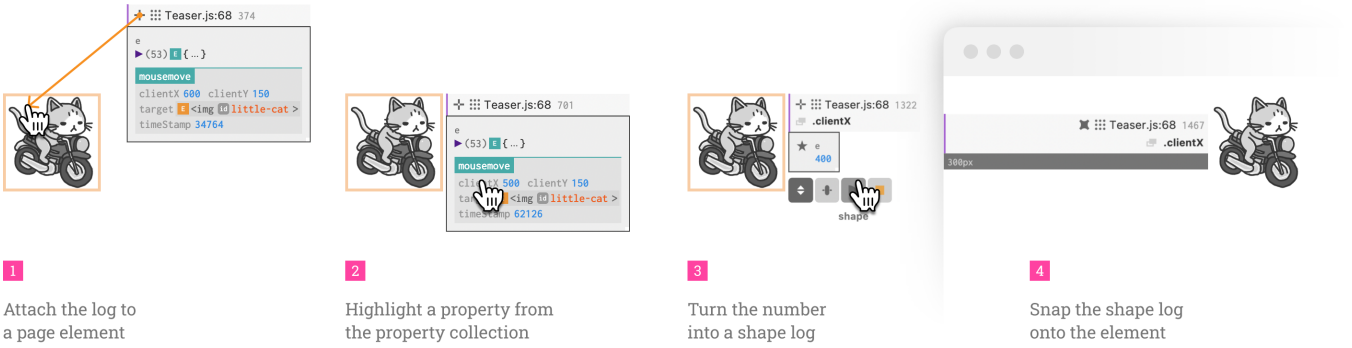

Figure 1: An example workflow of interacting with a *Stream* using Log-it.

ABSTRACT

Logging is a widely used technique for inspecting and understanding programs. However, the presentation of logs still often takes its ancient form of a linear stream of text that resides in a terminal, console, or log file. Despite its simplicity, interpreting log output is often challenging due to the large number of textual logs that lack structure and context. We conducted content analysis and expert interviews to understand the practices and challenges inherent in logging. These activities demonstrated that the current representation of logs does not provide the rich structures programmers need to interpret them or the program's behavior. We present Log-it, a logging interface that enables programmers to interactively structure and visualize logs in situ. A user study with novices and experts showed that Log-it's syntax and interface have a minimal learning curve, and the interactive representations and organizations of logs help programmers easily locate, synthesize, and understand logs.

CCS CONCEPTS

• **Human-centered computing** → **Information visualization**; • **Information systems** → *Users and interactive retrieval*; • **Software and its engineering** → *Software testing and debugging*.

CHI '23, April 23–28, 2023, Hamburg, Germany

ACM ISBN 978-1-4503-9421-5/23/04.
https://doi.org/10.1145/3544548.3581403

KEYWORDS

Programming support, Program comprehension, Visualization

ACM Reference Format:
Peiling Jiang, Fuling Sun, and Haijun Xia. 2023. Log-it: Supporting Programming with Interactive, Contextual, Structured, and Visual Logs. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23), April 23–28, 2023, Hamburg, Germany.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3544548.3581403

1 INTRODUCTION

By inserting statements that output variables, data structures, and customized messages to a terminal or console window, print and log statements are the most ubiquitous and dominant mechanism that programmers use to inspect a program's status and execution [11]. With the help of statements such as `toString()`, objects of various types can be cast into text strings so that programmers can flexibly manipulate and log them to suit their diverse needs. Because log statements require little setup, they can also be easily placed anywhere in one's source code and have a minimal effect on the program logic and code execution [40].

Despite its simplicity and versatility, interpreting logged output displayed in a console or terminal is often very challenging due to the large amount of heterogeneous log statements a program has and the lack of meaningful structure and representations for the logged messages. For example, when using log statements within nested for-loops, an abundance of log messages are printed, making it difficult to track any particular variables or messages. Correlating output logs with the statements at different locations in a program is also very challenging, as logged output is organized linearly and solely based on the order of program execution and does not reflect the different scopes the originating log statements are in.

To enhance the interpretability of logged data, programmers often utilize various text formatting techniques, such as using spacing and special characters. However, when logs are used for examining programs that produce rich visuals or interactivity, these techniques may not be effective due to the inherent limitations of textual representations and the linear organization of logs. For example, when a visualization designer prints out the attributes of each data point in a scatter plot, it is almost impossible for them to map the one-dimensional log stream in a console to the visual elements distributed in the two-dimensional visualization. The situation becomes even more complicated when data points start animating and relocating as the underlying data changes.

This research thus seeks to augment existing textual logging mechanisms to address their many challenges and consequently support the use of logs in inspecting, understanding, and debugging programs. A formative study with seven senior web programmers was conducted to achieve this goal. Findings showed that the purposes of logging varied from surfacing execution status to debugging unexpected code behaviors, supported by diverse compositions of log statements and messages. However, the current simple and monotone organization of logs was incompatible with the complex needs of programmers, leading to challenges including identifying targeted outputs among cluttered logs, aggregating information distributed in different log messages, and interpreting them towards the next steps of programming — i.e., locating, synthesizing, and understanding logs.

To address these problems, we propose Log-it, a logging interface that outputs log messages with interactive, contextual, structured, and visual representations. Log-it organizes individual outputs of the same log statement as a *Stream*, which serves as the foundation for imbuing interaction, visual, and context into streams of log outputs. By turning *Streams* of logs into interactive objects, for example, programmers can flexibly inspect the data structures in log outputs, such as navigating through different levels of the data structures and creating collections of properties of interest to better inspect them simultaneously. Streams of logs can also be imbued with rich contextual information. For example, when programming, the programmers accumulate rich contextual information on how the source code is structured syntactically or visually. However, this information is lost in the present linear organization of the log outputs. Log-it brings the contextual information to the log outputs, resulting in, for example, indented log outputs that match the indentation of the log statements, enabling the programmers to establish the context of the log outputs quickly. With the augmented *Stream* organization, Log-it assists programmers in better locating, synthesizing, and understanding logs.

We evaluated Log-it with six novice and six expert JavaScript programmers in a user study. Participants were asked to complete programming tasks using Log-it. Qualitative analysis of the results revealed that Log-it addresses the challenges associated with the current logging mechanisms by organizing log outputs into *Streams*, allowing programmers to easily locate, synthesize, and understand log messages. Moreover, Log-it enhances the existing logging experience with interactive exploration, visualization, structured formatting, and contextual information. Based on feedback from the participants, we also identified limitations and suggested potential avenues for future research.

This research thus makes the following contributions:

(1) The identification of the challenges and needs inherent in the use of logs in JavaScript web programming via a formative study of interviews with professional programmers.
(2) Log-it, a novel web-based logging interface that introduces the idea of interactive, contextual, structured, and visual logs. The Typescript package of Log-it (provided as supplementary material) is open-sourced and available for public installation through Node Package Manager (NPM)[1].
(3) A user study evaluation of Log-it that demonstrated advantages and limits of Log-it and provided insights into the future development of logging interfaces in the context of rich visual and interactive environments.

2 RELATED WORK

The present research resides at the intersection of empirical research that seeks to understand programmers' logging behaviors, and systems that have been developed to support logging, program visualization, and debugging.

2.1 Understanding Logging Behaviors

Substantial effort has been invested in understanding the methods that programmers use to instrument and manage logs for the software runtime analysis [9, 37], explaining fails and crashes [10, 15], and investigating user behaviors [48, 67].

Gholamian and Ward, for example, summarized the existing challenges inherent in authoring log statements in what, where, whether, and how to log, as well as log analysis and mining [15]. Yuan et al. found that it was common for programmers to modify their log statements, which revealed opportunities for compilers and support tools to automate these manual efforts of log authoring and modification [69]. Fu et al. answered the important question of *"Where do developers log?"* with an investigation into large-scale industry-level software [14]. They divided logging statements into five categories by their location in code (e.g., assertion-check, return-value-check, exception, logic-branch, and observing-point) and identified common factors to be considered for logging decisions.

While prior research has comprehensively examined the various characteristics of log statements and messages found in source code and log databases, less has investigated the challenges programmers have in synthesizing and understanding the logs that are created during programming. For example, logs that record systems' status and behaviors for *post hoc* diagnosis and analysis are often formatted according to established guidelines to facilitate restoring the system context and data mining [15, 37, 48]. In contrast, the log statements and messages that we focus on are created *ad hoc* to understand the code on the fly. As a result, the goals and behaviors of logging, log formats, and the challenges that are embedded within the logging process are fundamentally distinct.

However, few prior works investigated the behavior and underlying challenges in understanding ad hoc logs. To understand the challenges in interpreting ad hoc log output, we interviewed professional programmers. Our interviews uncovered challenges they face in interpreting log output, resulting from the absence of meaningful organization, structure, and representation of log

[1]Log-it package is published at https://www.npmjs.com/package/log-here-now.

messages, as well as the lack of connection between log messages with other elements in the development environment, such as the source code, the execution output, and debugging tools provided by integrated development environments (IDEs).

2.2 Supporting Log Generation and Interpretation

The writing of log statements and interpretation of logged output have long been a focus within the research community. Opportunities to enhance logging, including automatic logging, tracing, log filtering, categorization, and visualization, have been identified and supported based on the logging behavior research [14].

Authoring high-quality log-printing statements is an important way for programmers to diagnose failures and inspect the underlying status of a program [27]. Prior work has developed comprehensive methods to assist programmers in finding the optimized location to insert logging statements [38, 68, 70, 73], set appropriate log levels [28, 36], and choose the critical variables to output using logging statements [39, 70, 73]. Projects such as Errlog, for example, suggested logging points that could be used to cover potential failures based on 250 real-world failure reports [68]. LogEnhancer identified important variables and added them to log statements [70]. LCAnalyzer helped programmers find poor logging practices using a set of proposed categories of logging patterns that should be avoided, e.g., a duplicated calculation that has happened before or after the log statement [8].

The output of log statements is usually a linear stream of text-based messages. Efforts that help programmers interpret it included filtering, categorizing, and visualizing logs. Log++, for example, used the dual-level control of logs to separate logs for debugging and production status tracking, with no cost on execution from the inactive logs [40]. Emoji-nized Log Browser attached emojis to logs based on customized rules to help read text-based logs for server communications [55]. Tudumi produced 3D directed graph visualizations to help system administrators monitor and audit logs as a defense against security breaches [56]. Beschastnikh et al. presented a method to infer system models out of logs, assisted by customized log-parsing rules [4]. On the other hand, ChronoViz and similar systems visualized data and logs to aid the exploration and synthesis of multimodal recordings [13, 26, 42, 43].

The above log-assist research efforts mainly focused on *post hoc* logs that occurred during system deployment, e.g., recording server status and calls, leading to domain-specific topics that prioritized log mining, printing statement automation, performance diagnosis, log maintenance, and log parsing [15]. In the meantime, support for *ad hoc* logging while programming has largely been ignored. Such logging requires the real-time exploration, modification, and interpretation of log messages rather than the mining, analysis, and modeling of a collected log database [4].

In the present research, front-end JavaScript web development was used as an example programming scenario where programmers need to tackle various challenges, including asynchronous processes, elements and visuals in the interface, and user interactions. We investigated this untapped logging context and designed Log-it that effectively addresses programmers' pressing needs while exploring and interpreting ad hoc log output on the fly.

2.3 Understanding Programs through Visualization

Visualizations have been widely adopted to support understanding and debugging programs in the professional and educational contexts [23, 24, 52]. Lerner [34] and Guo [19, 20] visualized variable values and underlying program states next to the code. Victor demonstrated direct manipulation and visualization of code for creative programming [60]. Omnicode used a matrix of scatter plots to visualize the entire history of all variable values as a novice learning tool [25]. LaToza and Myers visualized the call graphs through a connected graph that supports reasoning about the causality, ordering, and repetition of the control flow [33]. Examplore showed common API use patterns across hundreds of examples with a unified synthetic code skeleton [18]. Skyline [66] and ExampleNet [65] visualized, profiled, and assisted in developing deep neural networks. Relo progressively built visualizations that mirrored a programmer's mental model as they explored unfamiliar large codebases [50]. OverCode enabled the visualization and exploration of thousands of programming solutions by clustering similar solutions through algorithms and help from the instructors [17]. Theseus colored unexecuted code blocks and showed call counts of functions to address programmers' common misconceptions about their code [38]. Bifröst and WiFröst, on the other hand, helped users debug IoT projects with embedded code and circuits, with a time-synchronized interface visualizing recorded states of different components in a networked system [42, 43].

As to web development, the event-driven and asynchronous nature of JavaScript introduces other challenges in recognizing the code behavior, and tools were developed to assist in tracing execution upon interactivity [1, 6, 21]. Isopleth, for example, supported the sense-making of production websites by exposing hidden functions and event relations, displaying functionally related code blocks, and enabling code manipulation directly through its interface [21]. Alimadadi et al. preserved the details of event-based interactions at different semantic levels of granularity [1]. Other systems addressed difficulties synthesizing different components of a web project, where control flow, content definitions, and styling are separately specified by JavaScript, HTML, and CSS [7, 22]. Telescope, for example, bridged the visual and functional connections between JavaScript code and HTML markups [22]. Scry presented a method to traverse back to HTML definitions and related JavaScript code through captured visual changes on the page [7].

A key reason for the plethora of visualization systems for programming is that programming states and executions are typically invisible to the programmers. As a result, programmers often have to simulate the program behaviors and maintain a large number of program variables in their working memory, incurring excessive cognitive load [44]. By representing the various programming elements with suitable visualizations, programmers can offload the numerous program states, variables, and behaviors to external representations, significantly reducing their cognitive strains. The present research shares this spirit. However, prior programming visualization systems only focused on elements that are directly related to the program logic (e.g., variables, logic, and APIs), and less explored the visualizations of the ephemeral and assistive, yet ubiquitous, log statements.

2.4 Supporting Program Debugging

Debugging is the most time-consuming activity during software development and has been thoroughly investigated by the research community [12, 16, 31, 41, 47, 62, 72]. When debugging, programmers need to trace program execution, reproduce failures, hypothesize underlying program behavior, locate and diagnose the corresponding code, and observe changes before and after the fixes, in an interleaved and trial-and-error manner [12, 16]. Von Mayrhauser and Vans summarized the actions, processes, and information needs for debugging, where programmers made connections between different levels of code abstraction based on domain expertise and knowledge of the code [62]. McCauley et al. surveyed the causes and types of bugs, and the knowledge that aids debugging [41].

To address programmers' complex needs in such multistage processes, prior research has proposed multiple methods and tools that can assist in the debugging task [31, 72], including the traditional log-based and breakpoint-based debugging [41, 53], as well as novel methods such as omniscient debugging [49], automatic debugging [2, 3], interrogative debugging [29, 30], relative debugging [71], program slicing [58, 64], and program visualization (as discussed in Section 2.3). As an example, omniscient debuggers addressed these limitations by recording the complete execution process without interruptions and allowed programmers to navigate through the obtained trace afterward [35].

Despite the plethora of advanced debugging systems, fundamental debugging tools such as logging and breakpoints remain ubiquitous for their versatility and lightweight nature. While systems such as program visualization and omniscient debuggers may suffer from scalability challenges and often require dedicated environments that create an extra learning overhead for beginners and may conflict with the existing workflows [41, 49, 52], logging requires no special setup or runtime environment and can be placed anywhere in the code to inspect program state and behaviors. As a result, not only can logging be used a standalone debugging tool, it is frequently used in tandem with more advanced debugging systems. For example, Replay.io, an omniscient debugger, allows programmers to go back to certain points through the log output and video recordings and insert new logs as needed [51].

The present work focuses on improving logging, a fundamental program inspection and debugging instrument. The proposed interface and interaction techniques were carefully designed to ensure that they do not jeopardize the simplicity and flexibility nature of logging. In doing so, the versatility and ubiquity of the logging mechanism lend the proposed improvements to a variety of debugging needs and workflows.

3 FORMATIVE STUDY

To understand the challenges programmers encounter with existing logging mechanisms, an interview study was conducted with professional programmers.

3.1 Interview Preparation: Content Analysis

Prior work has comprehensively investigated the usage of log statements by analyzing the statements extracted from source code [14, 15]. However, we found the literature did not provide sufficient first-hand data to gain a concrete understanding of how log statements are used in context. Therefore, we opted for content analysis to situate findings from the literature and to inform our interview. We collected 64,225 JavaScript and TypeScript log statements on GitHub and randomly drew 400 statements for detailed inspection. When analyzing the log statements, we focused on examining their composition and context, which were used to infer their goals. We developed a codebook based on literature and an overview of statements in the dataset, which guided our analysis of the collected log statements [14, 15, 32]. In this process, we iteratively refined the code book to accommodate statements that could not be categorized. We identified 4 goals (i.e., reachability check, value inspection, visual divider, and catching error), 4 content compositions (i.e., labeled values, pair or group of items, function execution, and object or partial object), and 4 contexts (e.g., group of log statements, comment, associated with elements, inside logging wrapper). Since our findings are largely consistent with prior works, we report detailed analysis steps and results in supplemental material along with the full log dataset [14, 15, 32, 69].

3.2 Expert Interviews

With the understanding gained from the use patterns that arose from the content analysis, seven experienced programmers were interviewed to further investigate the challenges that occur during programming (i.e., P1–P7, 6 male and 1 female, age 24–35, with 5–8 years of programming experience and 2–8 years of professional web development experience). Before the interview session, interviewees were required to prepare one or more of their ongoing or recent JavaScript projects to share with the interviewers. All interviews were conducted remotely via Zoom videoconferencing and lasted around one hour. Each interviewee received 40 USD as compensation for their time.

During each interview, interviewees first described their experience in JavaScript and introduced the projects they had prepared. They were then asked to share their source code using screen sharing and walk through the `console.log()` statements in their code with the interviewers. For each `console.log()` statement, interviewees were asked to describe the goal of that statement, how they wrote and modified it, how they interpreted the output of the statement, and any challenges they encountered during this process. The content analysis that was conducted enabled the interviewers to quickly recognize recurring patterns with logged statements, which sped up the interview process. Interviewees were also asked to retrospectively discuss the statements they had originally added but later deleted. After reviewing the `console.log()` statements, interviewees were asked to reflect on their general practices and frustrations with existing logging mechanisms.

All interviews were recorded and then transcribed. The first and second authors conducted a thematic analysis and summarized themes regarding why and how programmers utilize log statements and the challenges they encounter [5].

3.3 Findings and Discussion

Besides the goals identified in the content analysis, interviewees also used log statements and their output for communication purposes. P4 and P6 mentioned that they would leave logs in their

code as reminders to develop new features, which acted as placeholders until the corresponding function was implemented. The diverse goals of the logging mechanism highlighted its flexibility and versatility. As P5 noted, *"I have complete control over what I want an output when using* `console.log()`*."* Similarly, P7 noted that *"what's great about the log is that it just is super ad hoc, and then you can just jam, whatever stuff you want into it."* Despite these strengths, they all faced some issues, which were summarized into four **C**hallenges with current logging mechanisms:

C1: Logged Output Lacking a Meaningful Organization. While programming, it is common to have logged output flood the console panel of the browser and make it hard to find specific logs. For example, P5 said, *"you would end up with a lot of lines [of logs], and you have no idea where it's happening and what is happening."* P2 reported that *"I will use a sequence of symbols like* `console.log('======')` *to create an outstanding block of a group of logs that I want to focus on."* The lack of a meaningful organization in logs creates significant challenges when programmers try to understand logs. For example, interviewees often put a log statement inside for-loops to inspect variable changes or recursive functions to trace execution call stacks. While interviewees need to inspect all the log message output by the same statement to understand a control flow, these log messages are often distributed far away from each other in the console panel, requiring interviewees to frequently comment or un-comment other log statements in the program, scroll back and forth in the console panel to locate all related messages, or adjust the log output format to make messages of interest more distinctive.

C2: Data Structures Lacking Informative Visualizations and Interaction. While traditional logging mechanisms flattened all data structures as text, the logging mechanism used in modern web development still preserves some data structures. For example, arrays and Javascript Objects (JSON) output to the console preserve their data structures and allow programmers to inspect elements and values stored within them. While this is helpful, for nested complex objects and arrays, interviewees needed to drill down to retrieve what they needed. P1–P3 and P7 all shared their struggles exploring nested objects. P1 said that *"there's a little bit of syntax highlighting here [in the console], but everything's very kind of clumped together and you have to expand [them], ... not as kind of visually parsable."* If interviewees would like to log the specific properties deep in the data structure, they would need to switch back to the source code and write new log statements that can access the property. Values in the data structures are also often hard to interpret. Interviewees suggested their trouble understanding the meaning of numbers, especially when they are about positions and dimensions. P5 suggested that sometimes he needed to use screenshot tools (e.g., those displaying the size of the captured screen area) to measure the size of a UI element to have an *"confirmation that, what I have in the DOM, is actually aligned with the data that I [want to] visually represent."*

C3: Loss of Context due to Frequent View Switching. Interviewees reported that they often used the console panel alongside another set of views to fully understand a program's behavior, especially when debugging visual and interactive content (e.g., the Element panel of Chrome DevTools, web pages, and source code in an editor). Switching among them caused interviewees to forget the context of the logs in one view or another, resulting in an excessive amount of time needed to connect information from different views. P4 said, *"I don't remember where this* `console.log()` *is coming from. What I have to do is to copy this log and do a global search to find the* `console.log()` *[statement in the codebase]. [Then] I realize, okay, it's here and there might be something wrong here."*

C4: Trade-offs when Crafting and Interpreting Logs. When asked how they deal with the inconvenience of log statements, interviewees often said that they rarely spent time improving log statements, as logging was rarely the ultimate goal of their project. P1 said that *"usually I'm too lazy to do that because it feels like a lot of work for something that's not actually going to end up in my code, which maybe is like a fault on my part."* Likewise, P7 acknowledged some improvements when looking at the logs in their project but said that *"I guess I could code that in, but I am lazy."* In general, due to the ephemeral nature of logs, interviewees lacked the motivation to improve them, which led to poorly formatted logs and eventual difficulty when interpreting them, thus leading to the aforementioned challenges.

3.4 Design Goals

To tackle the challenges associated with using logs while maintaining their ease of use and flexibility, our system aims to provide support for logging by incorporating new organizations, visualizations, interactions, and contexts, which have been designated as the following four **D**esign **G**oals:

DG1: Offering Meaningful and Informative Log Structures. The unstructured log output hinders programmers from locating and interpreting relevant information. Rather than presenting logs in a linear format, we advocate for using familiar and informative structures that programmers are already accustomed to, such as scope structures in the source code. Additionally, customized structures tailored to specific tasks, such as comparison tasks, can be utilized to provide relevant context and aid in log comprehension.

DG2: Supporting Interactive and Visual Logging. Textual representations used in log messages have inherent limitations that can make them difficult to comprehend. As a result, programmers may need to continually refine log statements in the source code to achieve the desired log output. Log messages should be augmented with visual representations and interactions to facilitate visual understanding and interactive exploration.

DG3: Embedding Logs within Contexts. The lack of connections among the various views that programmers need to visit when they write code and understand log messages often leads to the loss of context. The log output should be embedded with the contexts to facilitate log interpretation, e.g., providing information on the originating log statements and corresponding element and interaction output within the log output.

DG4: Minimum Effort for the Desired Enhancements. As logging is not the end goal of programming, programmers are often reluctant to invest substantial effort in refining log statements. Therefore, the new logging mechanism should require minimum effort to set up and use while simultaneously providing the benefits of structured, interactive, visual, and contextual logs. This allows programmers to efficiently gain insights from logs without diverting significant resources from their primary programming objectives.

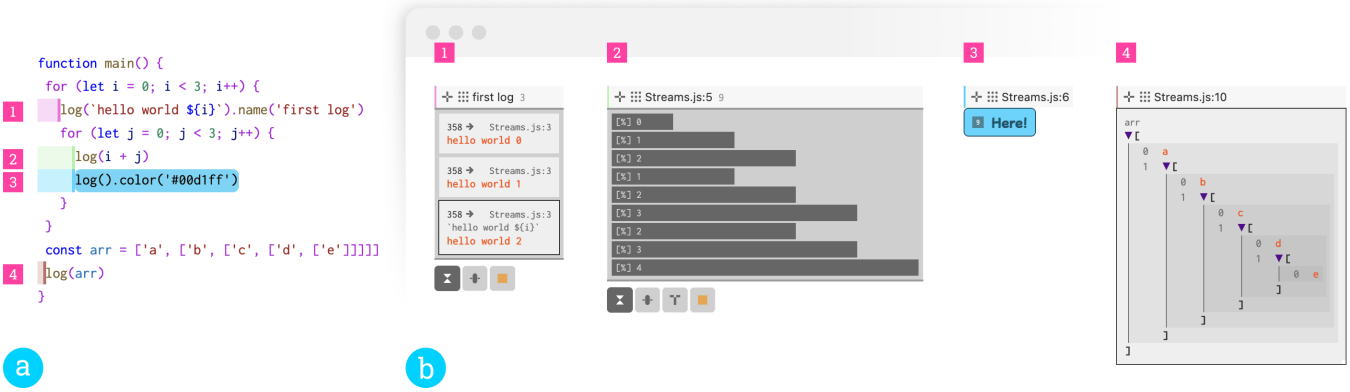

Figure 2: Screenshots of an example code snippet in Visual Studio Code displayed with the Log-it extension (a), and the corresponding log *Streams* produced (b). VS Code extension highlights the color coding of the statements which matches that of the *Streams* in the browser window.

4 LOG-IT

We propose to augment logs with interactivity, context, structure, and visualization to help programmers easily locate, synthesize, and understand the log output. To this end, we have designed, implemented, and open-sourced **Log-it**, which consists of a typed JavaScript package and an optional Visual Studio Code (VS Code) extension. To use Log-it in place of `console.log()` statements, a programmer would code

`log('hello world')`

after the package was imported into their project rather than using the more traditional `console.log('hello world')` (Figure 2.a). Further, extending the original `console.log()` function, Log-it offers function chaining to customize the *Streams*, such as `.name()` to set the name of the *Stream* (Figure 2.1) or `.color()` to change the background color (Figure 2.3).

4.1 Turning Log Outputs Into Log Streams

A log-printing statement outputs one or more log messages. Instead of seeing a series of log outputs originating from the same log statement as individual output, we propose structuring these logs into a *Stream* as the unit for organizing logs. The direct mapping from log statements to the *Stream* of log outputs enables programmers to tailor how each *Stream* would be displayed and situated properly to help them gain insights into the code from the logs.

A *Stream* consists of *three* parts that enable interactions or convey information about itself, its hosted logs, and the underlying `log()` statement — The *Stream Header* can be dragged to relocate the *Stream*, and its crosshairs can be dragged to attach the *Streams* to page elements (Section 4.3.5). The *Stream Header* also displays the name of the *Stream* and the count of logged items hosted by it, as shown in Figure 2.b. The *Stream Body* hosts the log outputs produced by the `log()` statement. It defaults to a chronological thread of logged items and offers a rich variety of visual representations to be introduced in the following sections. The *Stream Menu* offers a set of controls for the *Stream*, including *Pause/Resume* (to control the execution of each `log()` statement individually on the fly), *Delete* (only shown after the *Stream* is paused, to remove the *Stream* and disable the statement for the session), and *Unfold/Collapse* (to minimize the *Stream* to show only the latest output or unfold it to view all the logs). The menu also offers dynamic controls (e.g., for changing the output representation), when available, for different data types and structures of the logged items.

Each *Stream* has three attributes that contribute to its advanced interpretability: *Interactivity* (what ad hoc interactions the *Stream* offers for dynamic exploration), *Representation* (how the *Stream* is structured and visualized), and *Context* (how the *Stream* is situated with page elements and links to the code).

4.2 Interactions for Ad Hoc Object Exploration

Programmers frequently log program variables to inspect their values. However, the traditional console view does not provide adequate interactions for programmers to flexibly inspect logged variables of complex data structures. To address this issue, the *Stream* preserves the data structure and provides rich interactions to help programmers inspect and compare values within and across the hosted logged messages.

4.2.1 Property Highlighting. Programmers often need to log a subcomponent of a complex data structure, such as a property within a nested JSON, to inspect its value. However, to specify the correct index of the desired property in a typical `console.log()`, the programmers need to log the entire JSON object, navigate to the desired property, record the index path manually, and reformulate the log statement in the source code to output the value of a property. If they want to inspect other values, they have to repeat this process again and again.

We introduce *Property Highlighting* that enables programmers to dynamically modify the object display in the log messages on the fly so that they can focus on only part of an object's output (DG2). Programmers can bring a particular part of the object of interest "to the center stage" and hide the other parts (Figure 3.a). They can highlight a property value by double-clicking on its key (Figure 3.1) and move back to any level above it by clicking on it in the indexing path (Figure 3.2). After finding the targeted indexing,

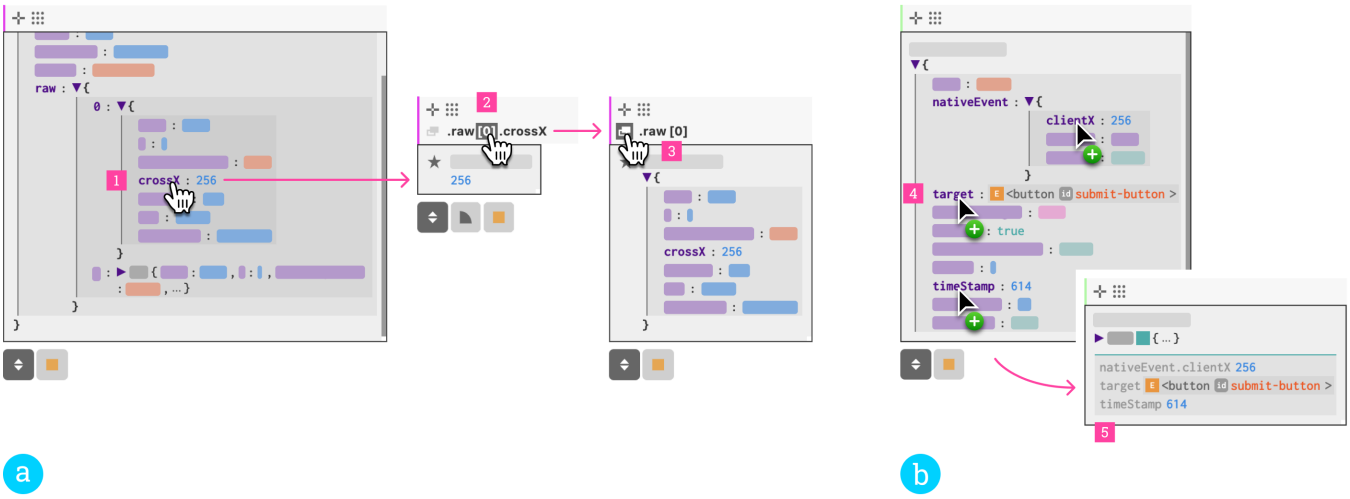

Figure 3: *Property Highlighting* of a single property (a), and multiple properties (b). During the single-property highlighting, the programmer double-clicks the property of interest (1), after which the indexing path is shown in the *Stream Header*. The programmer can move back to the levels above by clicking on the path (2), or keep double-clicking the property of the sub-object, if possible. They can click the *Copy* button in the Header to copy the indexing path (3). To highlight a collection of objects, they sequentially right-click on the property they would like to highlight (4), which will be appended to the collection at the end of the original object (5). They may then collapse the original object and track the small collection of highlighted indexes.

programmers can copy the whole indexing path (Figure 3.3) and paste it back into the code.

A programmer, however, may want to inspect multiple properties at the same time. They can select the properties they want to inspect one by one by right-clicking on the keys of the properties of interest to add the properties to the collection of the object for highlighting (Figure 3.4). Right-clicking again on a selected property removes it from the collection. Selected properties are aggregated in a collection card that allows the programmers to inspect all of them simultaneously (Figure 3.5). Because the properties may be at different depths of the nested JSON, the index paths of them are therefore maintained individually.

4.2.2 Synchronized Interaction of Log Stream. Building upon the stream structure, interactions with one log message are synchronized with all other log messages generated by the same log statement hosted in a *Stream*. For example, when a programmer highlights the properties in one log message, the same properties in other log messages are also highlighted, enabling the programmer to inspect the values changing across different log messages. Other interactions with the log messages are also synchronized including the expansion of an object property and the scrolling of the viewport (each logged item is bounded with a maximum width and height to avoid page overflow), as shown in Figure 4.a. The synchronized interaction eliminates the need to manually manipulate each individual log message one by one, allowing the programmers to manipulate any message in a *Stream* as a template to broadcast to the entire *Stream* of logs.

4.2.3 In-Place Sliding. A linear view is a typical way to display logs from a stream. However, it is unsuitable for tracking changes among a large number of complex objects, as comparing differences when there is displacement and distance is usually hard [45]. To address this, logs in a *Stream* can be organized to view in-place — instead of showing a list of logs, only one logged item is shown at a time, and the indexed log is controlled by a slider with timestamps (Figure 4.b). To further facilitate the inspection of complex data structures, changed values within the data structure are highlighted with a fading background (Figure 4.4), allowing the programmer to spot the changes more easily.

4.3 Context of the Source Code and Web Page

Programmers need to interpret logs with contexts, i.e., where they come from, what elements or interactions they are related to, and what they imply. However, such contexts are often missing due to frequent view-switching and uninformative logging statements (C3). For example, when logging with the existing `console.log()` mechanisms, it is common to see a list of unlabeled numbers with no information about their source variable names or the originating log statements' position in the source code. Similarly, when developing a sequence of page elements, e.g., for a scatter plot visualization with each data point logging information about its attributes, it is challenging to map the linear log stream to the scattered visual elements. To address these challenges, Log *Stream* are augmented with relevant contextual information to assist the interpretation of logs. They can also be juxtaposed with the targeted visual elements to enable the in-situ understanding of log messages.

4.3.1 Showing Original Code. Based on the results from the formative study, manually adding labels to a log statement is a common practice (e.g., `console.log('obj.size[0]', obj.size[0])`), yet it creates unnecessary complexity, especially when a number of program elements (e.g, variables, functions) are composed in one

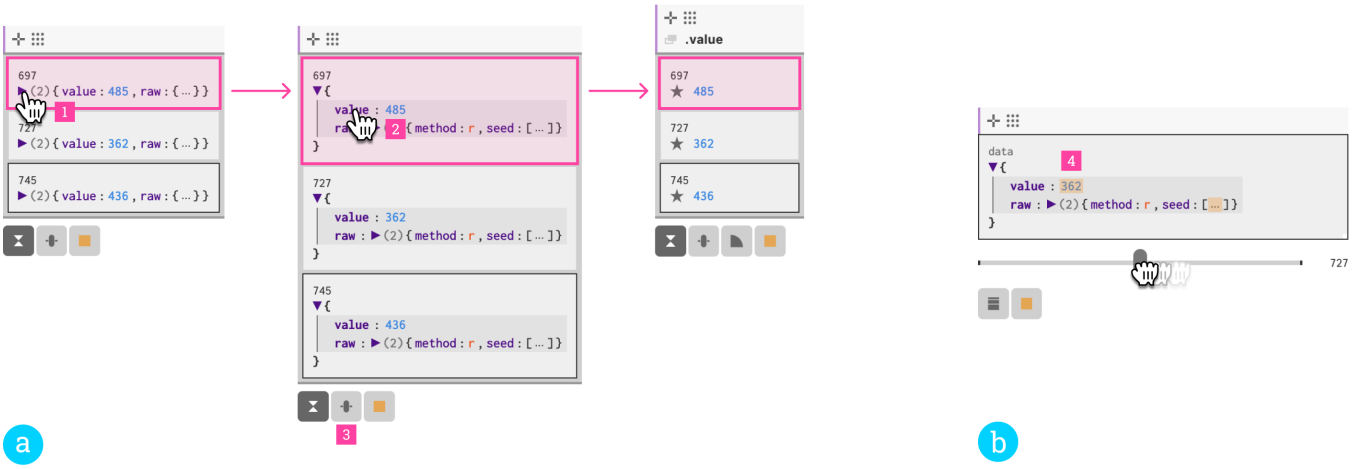

Figure 4: Synchronized interactions (a) and *In-Place Sliding* (b). While the programmer interacts with only the first log message (as highlighted in pink rectangles), i.e., expands a property (1) and highlights a property (2), these two operations are then synchronized across all output in the same *Stream*. The programmer can also use the *Stream Menu* (3) to re-organize logs in the *Stream* to enable *In-Place Sliding*, with which the programmer can view the changes of previous logs in place. The changes are highlighted with a fading background (4).

Figure 5: *Streams* label the log messages with the source code text within the log statement.

log statement. *Stream* automatically labels log output with the original code (i.e., the source code text within the log statement) by leveraging the Abstract Syntax Tree (AST) generated for the entire source code. This is achievable as the AST provides the beginning offset and length of the textual source code with the log statement.

4.3.2 Unified Stream. The output *Streams* of multiple `log()` statements, such as those at different scopes of a nested for-loop, can be combined into a single, aggregated view to allow programmers to easily compare and relate logs from different parts of the code. To merge *Streams*, a programmer can set the same *Stream* identifier for multiple statements by chaining `.id()` in the code or dragging one *Stream* onto another.

To track the entire program's execution as a whole, all *Streams* can be merged into one *Unified Stream*, i.e., a thread of all logs in one place in a linear fashion similar to the traditional console view (Figure 6.b). Further, we augment the scope of the statements in the source code to mitigate tracking and synthesizing issues that traditional consoles have.

4.3.3 Showing Code Scopes. Modern IDEs, e.g., VS Code, render plain code text with augmented structures depicted by line breaks, text indentations, colors, in-line icons, etc. to improve its readability [54]. However, the log listed in the console fails to preserve any structures except for being organized as a chronological list — a linear stream of left-aligned text messages. To create visual structures for the log output, programmers insert visual dividers using log statements, such as `console.log('======')`, while other kinds of structural information are harder to restore manually, e.g., the "depth" of a statement in nested for-loops. As `log()` statements are within the scopes in the source code, *Stream* output thus can be augmented with *Scope Indentation* to reflect their scopes in the source code, as shown in Figure 6.

With *Scope Indentation*, the scope information of `log()` statements is extracted using the AST. The scopes are then represented in the *Unified Stream* of log output with corresponding levels of indentation (Figure 6.c). This enables the programmers to leverage their mental model of source code, established during programming, to inspect the log outputs and more efficiently locate the desired *Streams*. Log *Streams* can be further organized into different columns based on threads of execution and top-level declarations (e.g., functions declarations) (Figure 6.d). As such, logs from different files, functions, and third-party packages, can be separated. Programmers can switch between indentation modes by clicking the arrow on the left side of the *Unified Stream* (Figure 6.1).

4.3.4 Leveraging Color Coding. Log outputs and their corresponding log statements can be further connected visually by leveraging color coding. By default, a hex color code is assigned to each statement based on its position in the code (i.e., the file name and line number) and can be displayed by the IDE extension, as shown in Figure 6.a. The same color is also assigned to log output, as shown at the top-left corner of the *Streams* and the left side of logs in the *Unified Stream*. The color can also be customized by chaining `.color()` after the log statement, which takes a standard CSS color string, such as `#00d1ff` or `red`. Color coding for both the code and log *Streams* updates synchronously.

4.3.5 Attaching Logs to Visual Elements. In addition to contextual connections with the source code, log outputs can be attached to

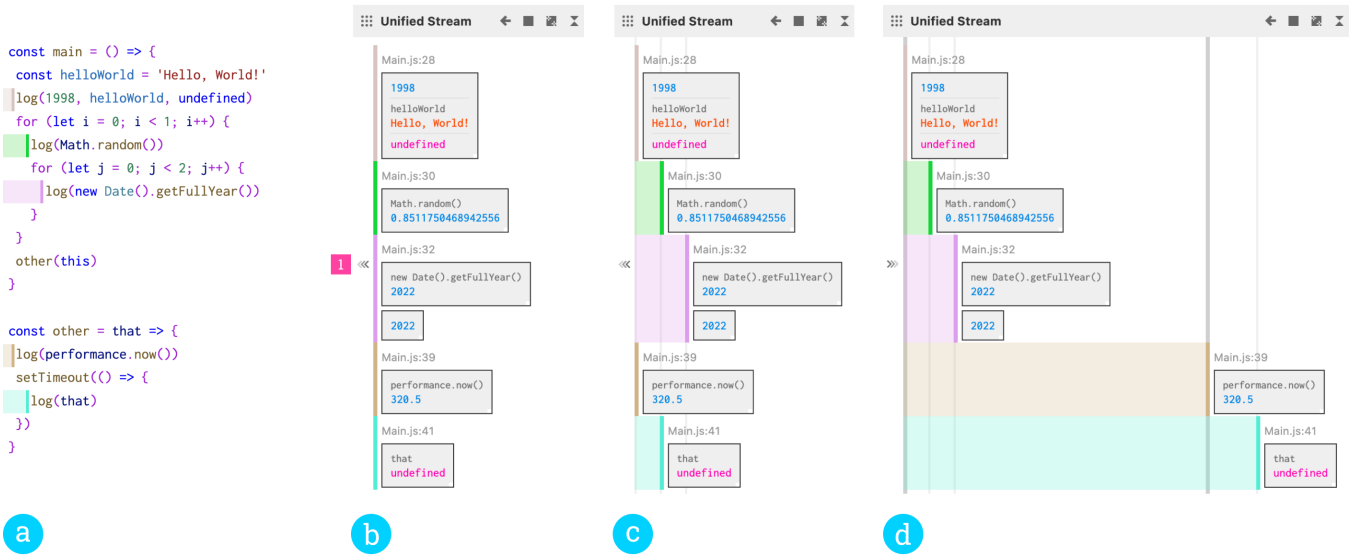

Figure 6: An example code snippet color-coded by the Log-it VS Code extension (a), its log output displayed in the *Unified Stream* view (b), and the different levels of *Scope Indentation* (c, d). Different indentation levels can be switched by the button on the side of the *Unified Stream* (1).

a specific page element. This enables the programmers to inspect the log output side by side with the related elements. When the *Stream* contains error messages (chaining `.error()`), *Element Alert* highlights the attached element using a blinking red overlay.

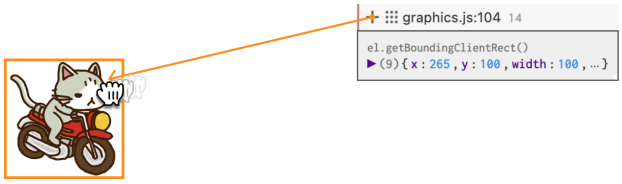

Figure 7: Attaching a log *Stream* to a page element[2].

The attached log *Stream* follows the anchored element during page scrolling or when the element is relocated. The attaching can be accomplished through the chaining function `.element()` that takes an `HTMLElement` object or the `id` of the element as input, or through the interface by directly dragging from the crosshair in the *Stream Header* to the element (Figure 7). Each *Stream* of logs is located near the element with the smallest overlap with other page elements or log streams. The position is automatically updated when the page elements change. Adding log *Streams* to the page can easily lead to visual clutter. To mitigate this problem, a programmer can activate the *Area Filter* to display only the log *Streams* that are attached to the page elements within the filtered area (Figure 8).

4.4 Dynamic Visual Representations

Textual logging is limited in its ability to describe information about rich visuals and interactions and often lacks the necessary structure for programmers to either divide or synthesize output from multiple statements for interpretation. *Streams* provide alternative ways to display log messages as visual elements and embed scope structures for improved organization and interpretability.

Spatial information is more intuitive and easier to understand when represented visually [57, 59]. For example, while it's unclear how big a 100-pixel-squared rectangle is on the current web page by the number itself, it becomes concrete when such a rectangle is displayed directly on the screen in situ. In the formative study, JavaScript expert interviewees have also suggested there can be difficulties in learning a reversed coordinate system, and in tracking and decoding the influx of changing numbers to describe visuals and interactions meaningfully. *Stream* enables alternative representations of interactive plots and graphics.

A *Stream* of number logs can be transformed into a horizontal bar plot using the *shape* button in the *Stream* menu (Figure 2.2). The widths of the bars are the values in pixels (or other CSS length units like `rem`, if specified by the programmer), or the percentage of the width of the bar of the maximum value in this *Stream*, if the programmer toggles on "displaying proportionally" by clicking on the bars. To visualize and place spatially-meaningful values (such as `clientX` of a page element) in situ for interpretation, programmers can *snap* the plotted logs onto the top, right, bottom, or left side of an element. This snapping can be performed by dragging from the crosshair in the *Stream Header* to the respective side of the element ad hoc, or by chaining `.snapTo([side], [element])` (where *side* can be `top`, `right`, `bottom`, or `left`, and *element* can be the `HTMLElement` or its `id`) to the `log()` statement.

Visualizing attributes (such as location and size) of event and page element objects helps interpret them (C2). *Stream* offers in-situ visualizations on the page for such objects. Based on the properties the object contains, the *Stream* menu offers the control to visualize

[2]The "cat riding a motorcycle" cartoon in Figures 1, 7, 8, and 9 was created with the assistance of DALL·E 2 by OpenAI.

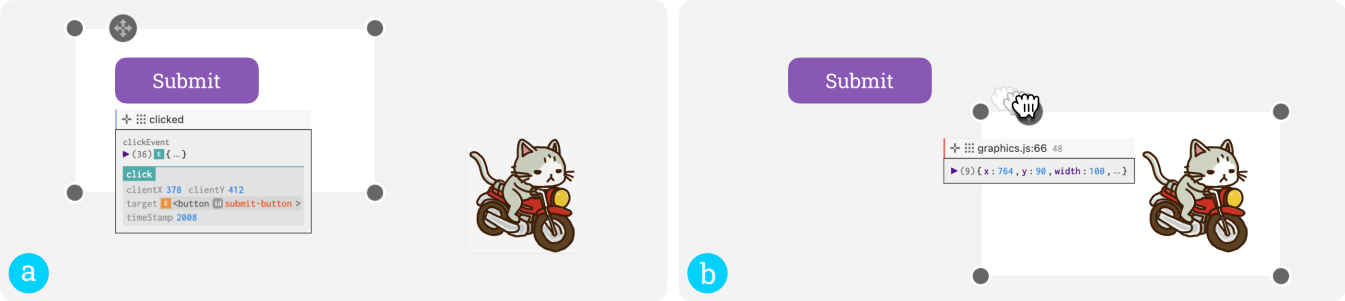

Figure 8: When *Area Filter* is enabled, only the *Streams* attached to the elements overlapping with it will show.

Figure 9: Visualizing the movement of an element by showing the history of its client-bounding rectangles over time in a heat-map style. With *In-Place Sliding*, the programmer can select and view a specific log in the history, and the highlighted visualization and its measures are updated synchronously.

it as a point (only containing pairs of location properties, like `x` and `y`, or `offsetX` and `offsetY`, etc.) or a rectangle (containing location properties and `width` and `height`). By clicking on the point dot or the measures button on the client-bounding rectangle, the programmer can toggle the display of the measures. Similar to the unfolding and collapsing of the *Stream*, the programmer can also switch between showing only the last shape or showing the entire history in a heat-map style (Figure 9).

4.5 The Log Stream Orchestra

We propose structuring logs into *Streams* as an interactive unit to organize logs from the same `log()` statement (or a group of statements with the same identifier) with tailored representations and rich context. We have presented a series of features of *Stream* to augment interactivity, context, structure, and the visual representations of the log messages, addressing existing log interpretation challenges. While these features contribute to each category of *Stream* attributes (i.e., interactivity, representation, and context) respectively, many of them work together seamlessly, as different interactivity features can be applied to different structures and representations, resulting in the flexible *Streams* for comprehensive real-world program understanding and debugging tasks.

5 IMPLEMENTATION

Log-it is built with React, and each *Stream* is managed by a component sourced from its underlying typed stream object. The stream object maintains important attributes of the *Stream* organization such as unique identifiers for the *Stream* and the attached element, original statement arguments, timestamps, collections of the highlighted property index paths and property expansions (for *Synchronization*), and the real-time ad hoc interactions (e.g., the *Stream* being dragged). A new stream object is constructed when its corresponding log statement is called for the first time. Each logged item is stored in its corresponding stream object, and a global directory stores pointers to all of them, including the scope information in the source code, allowing quick chronological sorting and structuring of logs when multiple *Streams* are merged, e.g., *Unified Stream*. A global listener monitors the position and presence of page elements being attached to, so the attaching *Streams* could properly update their positions when needed.

Log-it is open-sourced and available as a typed JavaScript package that can be installed into web development projects. Some features require a separate VS Code extension that reads and sends the source code and the AST of it to the front end. These features include *Showing Original Code*, *Scope Indentation*, and the IDE-side of in-line *Color Coding* display, as they require information from

the source code, including the original log statement in the source code, its position and scope in the code, and others. The connection to the extension is automatic without manual configuration. Other features work unaffectedly without the extension.

6 EVALUATION

To evaluate the degree to which Log-it addresses the problems programmers have with current logging mechanisms and achieves the design goals informed by the formative study, we conducted a user study with both novice and expert programmers. Qualitative analysis of study results verified the effectiveness and usefulness of Log-it for programmers at different skill levels with different types of tasks. Feedback provided insights into the concept of Log-it, existing features, and the future development of logging interfaces in visual and interactive contexts.

6.1 Participants and Apparatus

Six novice programmers (N1–N6, 3 female, 3 male, age 21–30) and six expert programmers (E1–E6, 4 male, 1 non-binary, 1 preferred not to report, age 21–35) were recruited for the study. Novice programmers had around one year of experience in JavaScript programming and had only used it for personal or course projects. Expert participants had programming experiences spanning 5–12 years and had worked with JavaScript for 2–8 years professionally.

The user study took place within an assignment-style React project where each task was left as a blank code block with guided comments above for participants to complete. Participants were asked to use VS Code and the browser of their choice, and to share the screen when doing the tasks. All studies were performed remotely using Zoom videoconferencing. Each study, including the follow-up interview, lasted approximately one hour, and each participant was compensated 40 USD for their time.

6.2 Procedure

6.2.1 Step 1: Introduction and Preparation (5 minutes). At the beginning of the study, the experimenters introduced the goal of Log-it, and then asked participants to download a compressed project folder for the study. A short text instruction guided the participants to set up the study environment, including installing the necessary packages to run the React project, the NPM package and VS Code extension for Log-it, and an assistive VS Code extension highlighting our inline instructions. After installation, participants started the React project with their preferred browser. Code changes within the study environment could refresh the web page automatically. We asked participants to place the editor and the browser side by side for easy viewing and screen-sharing.

6.2.2 Step 2: Programming Tasks with Log-it (40 minutes). Once the setup was complete, participants read through the code and completed the eighteen tasks in order. For each task, participants explored one or more features of Log-it. During the course of task completion, the help from experimenters included one-line documentation of each related feature of Log-it, the explanation of features when the participants were confused and asked, and clarification of JavaScript concepts for novice programmers. Participants were told to edit code and explore Log-it freely.

6.2.3 Step 3: Questionnaire and Interview (15 minutes). After finishing the tasks, participants were asked to rate the utility of Log-it on a 7-point Likert scale (which was later compressed to a 3-point scale during the report). The experimenters then conducted a semi-structured interview based on the results and observed use patterns to learn participants' views on Log-it.

6.2.4 Thematic Analysis. Studies were recorded and transcribed. Two authors coded transcripts and observed behaviors and generated themes by both the top-down and bottom-up approaches to understand how users used and assessed Log-it, as well as its strengths and limitations [5].

6.3 Results

All participants were able to complete the tasks and use all Log-it features. Overall, they found the *Stream* organization and its functionalities provided by Log-it helpful and easy to use. One of the key findings was that many of the features related to structuring, visualizing, contextualizing, and enabling interactive exploration of logs helped participants interpret the logs in ways that went beyond our initial design goals. For example, visualization techniques were originally intended to help understand logs containing spatial information, but participants also used them to locate specific logs. We present a detailed qualitative analysis of the transcription and behavioral data in themes corresponding to the four design goals, participant expertise, and limitations.

6.3.1 Context and Structure Helped Locate Logs. Participants found the various structures employed by Log-it to organize the log messages enabling them to locate the targeted logs more easily and quickly (all 12 participants agreed in the Likert-scale survey). We observed all participants using the *Scope Indentation* feature to indent and split a stream of logs based on scopes in the source code to locate the log messages and corresponding log statements. For example, when inspecting log messages printed by a nested for-loop, N6 and E2 used *Scope Indentation* to indent the log messages from inner and outer loops. Participants further split the log stream into columns based on the function structure to position logs printed from different functions. For example, in a stream of repeated log messages printed by a for-loop, E4 spotted a message of a different format. With *Scope Indentation*, the log message was separated into a different column rather than indented, indicating that this message is from another function (with an asynchronous callback) rather than the inner loops.

Participants found the *"different levels [i.e., scope, function, file] of indentation can be easily switched"* (N6) to organize log messages into different views. N3 and E2 suggested this feature could be more useful and powerful in understanding and debugging complex programs in production. E3 suggested that splitting the log messages based on the threads they are from could greatly help debug *"multi-threaded or pseudo-multi-threaded"* programs, which are *"usually the hardest things in any language."*

All participants attached log streams to the corresponding element on the web page whenever possible. When asked, they reported that attaching the logs to relevant elements helped them easily locate the target *Streams*. As N4 explained, *"tracking everything is impossible"* in a typical log console; extracting and attaching

the logs helped them *"de-cluttering"* the console and *"keeping track of"* the most relevant *Stream*. We also noticed that participants were more used to making the attachment through the interface with drag and snap, instead of through the code, i.e., chaining a `.element()`. As suggested by N5, attaching through the interface had more flexibility and allowed them to adjust the attachment ad hoc based on the changing inspection needs. On the other hand, N5 and E5 reported that making attachments permanent could be beneficial so they wouldn't get lost between page refreshes. Log-it can be easily extended to append the code automatically (e.g., `.element()`) to make the attachment permanent.

We observed different attitudes and use cases of *Color Coding*. Most participants favored it and utilized it to match the code and the interface (N1–N4, all experts except E3), visually locate the target log (especially the reachability logs) among others (E1, E4), and distinguish two similar logs from each other (E1, E2). However, N4 was confused when the generated colors for two different logs were very close, suggesting improvements in the color-generating algorithms. E3, on the other hand, suggested that they did not benefit from the feature much and requested stronger visual implications.

6.3.2 Interaction and Visualization Helped Synthesize Logs. Intuitive interactions and tailored visualizations were agreed upon to help synthesize logs, i.e., track the changes of a log statement over time and compare two or more log messages (all 12 participants agreed). For example, N3 found *Property Highlighting* helpful, which highlights a value in a complex object with a double click, because it enabled them to focus on a specific value in an object, or identify outliers of a value among many objects. We noticed that participants could *"seamlessly"* (E1) use *Property Highlighting*, *Synchronization*, and visualizations to inspect how values changed. For example, to inspect how a particular field of an object changes, N3 highlighted it with *Property Highlighting*. The highlighting was automatically synced across all log messages in the *Stream*. They then turned the numbers into a plot which allowed them to identify the increasing trend instantly.

The various visualization techniques also helped participants track changes in and compare visual attributes, e.g., sizes and positions of page elements. To make a page element draggable, participants needed to develop a function to update its `x` and `y` positions corresponding to the mouse movement. When composing the function, E6 couldn't decide which one of the two calculations was correct by just looking at the code and the numbers in the log messages. However, with Log-it, the updated offsets of the element were visualized, which enabled E6 to visually inspect the movements. Similarly, E2 did not know whether `innerWidth` or `outerWidth` of the window should be used for a follow-up collision detection task. They logged out both boundaries and the x-position of the page element and turned them into graphics to compare visually.

The *In-Place Sliding* also enabled participants to track value changes and identify anomalies by inspecting the values of a variable in place. E6 noticed an unexpected log in the *Stream* of a sequence of log output when the `mousemove` events were logged, *"There is no 'target' [in this log compared to others]... Oh, that's how. I moved my mouse away from the [page element]."* Besides, *In-Place Sliding* was also frequently used accompanied by other features as a log-locating method. For example, N6 toggled on visualizations for event object logs before using the slider, so the changes among logs were easier to capture. After finding a targeted object log, they then moved on to use other features like *Property Highlighting* to inspect details.

6.3.3 Locating and Synthesis Facilitated Understanding Logs. With features that support locating and synthesizing log output, Log-it facilitated understanding logs (10 participants agreed, 1 was neutral, and 1 disagreed). We observed participants iteratively adding chaining functions and trying different log structures and visualizations with the interface. They commented that Log-it enabled them to *"progressively"* (E1) explore and understand the log. N4 appreciated the improved readability of logs with Log-it, whereas in the traditional console, *"repeated logs easily fill up"* the console panel and they *"have to constantly resize the window to view complex objects."* On the other hand, E3 suggested that Log-it helped him understand both *"high-level things"* like program structure and control flow with features such as *Scope Indentation*, and low-level *"errors and attributes related to a particular component"* with *Element Alert* and various visualization techniques.

In general, participants suggested that the understanding of log was facilitated by many features of Log-it that helped them stay focused on the most relevant information (N4, N5, E1–E3), find the needed contexts for interpretation (N2, N3, N6, E1, E2, E4, E5) (as E4 suggested, *"through implicit and explicit connections to the elements and the source code"*), as well as view and explore logged information with intuitive and clear representations with interactions and visualizations (all participants).

6.3.4 Install, then Enjoy. Simple and easy to use is a major design goal of Log-it (DG4). To evaluate it, we provided no formal tutorials and minimal documentation of Log-it. The consensus was that it was easy to set up, learn, and use (all participants agreed). Using without the prefix `console` and chaining the typed function calls using dot operators greatly increased usability, as suggested by E2. Eleven participants indicated that they were willing to use Log-it for daily programming given the current presentation of Log-it.

Most features, such as *In-Place Sliding* and turning logs into plots and graphics, were self-explanatory and easy to find thanks to the auto-completion for chaining functions and the *Stream* menus in the interface. Expert participants explored features without being prompted by the corresponding tasks. E2 and E5 directly clicked into the package type definition file to inspect the full list of chaining functions and were able to discover almost all the organizing and structuring techniques with the interface even before they started doing their first task.

Participants encountered some discoverability and understandability issues with Log-it, which could be addressed by experimenting with the system or with little guidance from the interviewer. N5 and E4, for example, used `.name()` instead of `.element()` when trying to attach the log statement to an element. Hinted by the interviewer, they re-read the in-line documentation and corrected the function call. N6 was originally confused by the pausing and deleting operations of *Streams*, and instantly understood them after placing a log inside a click-event listener, pausing the *Stream*

through the interface, and triggering the event again. These confusing situations in using Log-it suggested future improvements in the graphical user interface and the code API design.

6.3.5 Novices Versus Experts. The study with both novice and expert programmers demonstrated both groups found Log-it useful. However, their use patterns, preferences, and expectations varied. Expert programmers tended to explore features on their own without being prompted by specific tasks, as indicated in Section 6.3.4, while novices mostly followed the instructions. Experts also performed more flexible combinations of multiple features for tailored log inspection. E1 and E3, for example, set shape logs into different colors to visually distinguish them. On the other hand, while novices found *Element Alert* useful (all except N4), experts expressed concerns about its effectiveness for coding pure logic (E3) and situations when elements were off-screen (E6). E5 suggested simplifying API, e.g., snapping through `.element("#button", "x")` instead of the current `.element("#button").snapTo("x")`.

Specific programming experiences and needs also affected how participants found a feature useful. For example, N1 and E1, who were working on creative coding projects, found visualizations their favorite feature, while E3, who worked on a complex database with many asynchronous processes, found features assisting in debugging parallel threads, e.g., *Scope Indentation*, the most helpful.

6.3.6 Limitations and Summary. While we believe a twelve-people group of both novice and expert programmers provided us sufficient data to evaluate Log-it and inform future work, the main limitation of the study comes with the short period of time the participants were able to utilize to learn and use our system. As participants were so familiar with the traditional logging mechanism, carried habits may affect how they could benefit from Log-it. For example, most participants (N3–N6, E1–E3) commented out deprecated log statements from the previous tasks or a previous stage of a task once or more, when they could leverage organizing and filtering tools, such as *Area Filter*, or directly pause or delete the *Stream* from the interface. Some features, such as *Scope Indentation*, were designed to address the needs of programming complex parallel systems, which can be challenging to include as a specific task within the constraints of a short session. On the other hand, multiple or longer sessions with more complex tasks and resulting learning effects might allow us to see the use of Log-it in practice with more diverse strategies and workflows.

In summary, our user study evaluation showed that Log-it, with interactive, contextual, structured, and visual logging, effectively helped novice and expert programmers during the whole log interpretation process, i.e., locating, synthesizing, and understanding logs. The *Stream* organization and many of its features were also easy to learn and use. Many usage patterns were identified, e.g., a feature may help more than one aspect of log interpretation, and multiple features were frequently used together in practice. Concerns and suggestions were recorded, e.g., caching changes made by the graphical interface as well as extracting and customizing a subset of features for domain-specific programming tasks. The feedback we received will help us improve Log-it in the future, and inform the development of logging interfaces for visual and interactive environments more broadly.

7 DISCUSSION AND FUTURE WORK

While Log-it was appreciated by novices and experts during the user study, limitations, insights into interactive, contextual, structural, and visual logging, and opportunities for further improvement and extensions were also identified, which we discuss below.

7.1 Limitations

Our main limitation is inherent in the logging mechanism — the ubiquitous log and printing mechanism is flexible and easy to use, but also produces an overwhelmingly large amount of output [49]. As user study participant E1 mentioned, *"there's just too many of them, and most of them that record past states a long time ago are not informative anymore."* Our study suggested that Log-it offers effective ways to organize, filter, and present logs, while we may further apply intelligent filtering and dropping of past logs to preserve a minimal set of logs that can best describe the latest program status to fit the programmer's dynamic needs.

In addition, as a comprehensive alternative to traditional logging, Log-it offers a set of features for varied needs to locate, synthesize, and understand logs, which creates an image of a complex system. For example, while front-end visual and interaction programmers may find visualizing graphics handy, others who primarily focus on algorithms and program logic may never use it. Lighter variants with partial features have been proposed by the participants during the user study, such as a version that specializes in log visualization that helps graphics and interaction developers. In the next step, we will explore ways to allow people to customize a subset of features for domain-specific tasks during installation or importing.

7.2 Beyond Web Programming

We focused on JavaScript web programming to instantiate our idea of structured and visual logs as this domain offers a wide range of functionality from client-server communication to user interfaces and interaction, thus varying the goals and types of logs that programmers author and interpret.

By extending the well-adopted logging mechanism, Log-it can be broadly applied to many other domains and programming tasks. For example, contextual logging can be effective for many other interaction- or visual-related scenarios, like gaming and virtual reality (VR). When developing a VR experience, programmers often need to take the headset on and off to switch between the VR environment and the console for logs, and map the linear stream of them to interactions and objects in a 3D world. Like Log-it, organizing the logs in streams and placing them in situ and in-depth with the 3D objects help embed necessary contexts to the logs for interpretation. Interpreting spatial information (e.g., where is the bounding box and how large it is) is also difficult in 3D environments, where the contextual visualization techniques that Log-it offers can help.

Structural and visual logging help even when people are programming for tasks without visual output. For example, *Unified Stream* and *In-Place Sliding* can enable programmers to declutter logs from different sources, trace parallel value changes, inspect program execution, and so on in typical log panels in most programming IDEs. Providing related contexts from code, such as matched color coding and showing the original code, continue to be effective

in these cases in reducing programmers' time and efforts spent on locating and synthesizing log output. Visualizations, such as turning numbers into an inline bar plot, can still help programmers trace output values and compare their changes.

7.3 Log Messages as Data Points

Log-it demonstrates multiple techniques for making textual log messages interactive, structured, contextual, and visualized. Instead of treating logs as plain text, Log-it treats log messages as "data points" that can be visualized and organized. This lends log messages to being enhanced with the rich literature on real-time and interactive data visualization and analysis [13, 46, 61, 63]. For example, by combining Log-it and DataInk [63], which demonstrated methods for creating and manipulating glyph-based data visualizations, log messages can be represented with user-defined marks that encode the timestamps and variable values of the messages. Leveraging customized visualization techniques applied to data points allows programmers to further tailor the logging interface to suit their needs. Similarly, by combining Log-it with unit visualization systems like SandDance [46], programmers can observe how log messages from different threads are produced at different rates with compelling animations. We believe this can open up opportunities to build "plug-ins" for Log-it that benefit from many of existing visualization tool kits and concepts.

7.4 Log-it and other Programming Support Tools

Formative study interviewees acknowledged the use of alternative tools to logging to understand program execution or to debug programs. The debuggers and profilers, for example, help programmers inspect and trace the variable values, control flow, and program performance. However, the formative study detailed the tedious setup process and dedicated environments they usually require that create an extra learning overhead and may conflict with other workflows in production. P5, for example, mentioned that using a profiler can be *"very tedious"* because it requires the programmer to carefully control the recording periods and configurations. On the other hand, program visualization tools structure and display data in a template fashion and may not offer the flexibility to reorganize, filter, or style the output to fit the programmer's preferences and needs. In contrast, using Log-it requires minimal setup and enables programmers to build their customized program inspection "dashboard" by dragging and attaching *Streams*, changing colors, filtering out deprecated ones, and so on, to fit diverse programming and debugging preferences and scenarios.

When designing Log-it, we strove to ensure that the simplicity and flexibility of logging were not compromised while adding new features and functionality. As shown in the user study, Log-it is lightweight and easy to use, can be easily integrated into existing workflows, and can work seamlessly with other programming support tools, just as the traditional `console.log()`. Log-it is already publicly available, and we plan to continue to improve and deploy Log-it in the wild to explore how Log-it can be used together with other programming tools.

8 CONCLUSION

Logging is a widely used technique for inspecting and understanding programs. However, the large amount of heterogeneous textual messages makes it hard for programmers to locate, synthesize, and understand log outputs. Our formative study showed that the many challenges of using the existing logging mechanism are rooted in the lack of structure and context for linear textual messages. We proposed augmenting logs with interaction, context, structure, and visualization. We demonstrated Log-it, a novel logging mechanism that structures log outputs as *Streams* with interactivity, tailored representations, and contextual information. Log-it enables programmers to locate log outputs from the cluttered space without back-and-forth scrolling, synthesize multiple log messages from relevant log statements, and understand log messages to better reason program behaviors. Results from the user study evaluation with novices and expert programmers showed that Log-it addressed the many challenges of the current logging mechanism and enabled new ways of understanding logs and programs.

ACKNOWLEDGMENTS

Matthew Beaudouin-Lafon contributed to the system design and did the voice-over of the video figure. William Song and Xiaoshuo Yao served as coders in the content analysis. We thank anonymous reviewers for their constructive and insightful reviews.

REFERENCES

[1] Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. 2014. Understanding JavaScript Event-Based Interactions. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 367–377. https://doi.org/10.1145/2568225.2568268

[2] Andrea Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. IEEE, IEEE, 162–168. https://doi.org/10.1109/CEC.2008.4630793

[3] Mikhail Auguston, Clinton Jeffery, and Scott Underwood. 2002. A framework for automatic debugging. In *Proceedings 17th IEEE International Conference on Automated Software Engineering,*. IEEE, IEEE, 217–222. https://doi.org/10.1109/ASE.2002.1115015

[4] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. 2011. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 267–277. https://doi.org/10.1145/2025113.2025151

[5] Virginia Braun and Victoria Clarke. 2012. *Thematic analysis.* American Psychological Association.

[6] Brian Burg, Richard Bailey, Amy J. Ko, and Michael D. Ernst. 2013. Interactive Record/Replay for Web Application Debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology* (St. Andrews, Scotland, United Kingdom) *(UIST '13)*. Association for Computing Machinery, New York, NY, USA, 473–484. https://doi.org/10.1145/2501988.2502050

[7] Brian Burg, Amy J. Ko, and Michael D. Ernst. 2015. Explaining Visual Changes in Web Interfaces. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology* (Charlotte, NC, USA) *(UIST '15)*. Association for Computing Machinery, New York, NY, USA, 259–268. https://doi.org/10.1145/2807442.2807473

[8] Boyuan Chen and Zhen Ming Jiang. 2017. Characterizing and Detecting Anti-Patterns in the Logging Code. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 71–81. https://doi.org/10.1109/ICSE.2017.15

[9] Boyuan Chen and Zhen Ming (Jack) Jiang. 2021. A Survey of Software Log Instrumentation. *ACM Comput. Surv.* 54, 4, Article 90 (may 2021), 34 pages. https://doi.org/10.1145/3448976

[10] Marcello Cinque, Domenico Cotroneo, and Antonio Pecchia. 2012. Event logs for the analysis of software failures: A rule-based approach. *IEEE Transactions on Software Engineering* 39, 6 (2012), 806–821.

[11] Brian Clark. 2017. Debugging in Node.js. https://www.clarkio.com/2017/04/25/debugging-in-nodejs/#debugging-progression

[12] Simon P. Davies. 1993. Models and Theories of Programming Strategy. *Int. J. Man-Mach. Stud.* 39, 2 (aug 1993), 237–267. https://doi.org/10.1006/imms.1993.1061

[13] Adam Fouse, Nadir Weibel, Edwin Hutchins, and James D. Hollan. 2011. ChronoViz: A System for Supporting Navigation of Time-Coded Data. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems* (Vancouver, BC, Canada) *(CHI EA '11)*. Association for Computing Machinery, New York, NY, USA, 299–304. https://doi.org/10.1145/1979742.1979706

[14] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where Do Developers Log? An Empirical Study on Logging Practices in Industry. In *Companion Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE Companion 2014)*. Association for Computing Machinery, New York, NY, USA, 24–33. https://doi.org/10.1145/2591062.2591175

[15] Sina Gholamian and Paul A. S. Ward. 2021. A Comprehensive Survey of Logging in Software: From Logging Statements Automation to Log Mining and Analysis. *CoRR* abs/2110.12489 (2021). arXiv:2110.12489 https://arxiv.org/abs/2110.12489

[16] David J. Gilmore. 1991. Models of debugging. *Acta Psychologica* 78, 1 (1991), 151–172. https://doi.org/10.1016/0001-6918(91)90009-O

[17] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Trans. Comput.-Hum. Interact.* 22, 2, Article 7 (mar 2015), 35 pages. https://doi.org/10.1145/2699751

[18] Elena L. Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. 2018. Visualizing API Usage Examples at Scale. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3174154

[19] Philip Guo. 2021. Ten Million Users and Ten Years Later: Python Tutor's Design Guidelines for Building Scalable and Sustainable Research Software in Academia. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '21)*. Association for Computing Machinery, New York, NY, USA, 1235–1251. https://doi.org/10.1145/3472749.3474819

[20] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 579–584. https://doi.org/10.1145/2445196.2445368

[21] Joshua Hibschman, Darren Gergle, Eleanor O'Rourke, and Haoqi Zhang. 2019. Isopleth: Supporting Sensemaking of Professional Web Applications to Create Readily Available Learning Experiences. *ACM Trans. Comput.-Hum. Interact.* 26, 3, Article 16 (apr 2019), 42 pages. https://doi.org/10.1145/3310274

[22] Joshua Hibschman and Haoqi Zhang. 2016. Telescope: Fine-Tuned Discovery of Interactive Web UI Feature Implementation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (Tokyo, Japan) *(UIST '16)*. Association for Computing Machinery, New York, NY, USA, 233–245. https://doi.org/10.1145/2984511.2984570

[23] Jeisson Hidalgo-Céspedes, Gabriela Marín-Raventós, and Vladimir Lara-Villagrán. 2016. Learning principles in program visualizations: a systematic literature review. In *2016 IEEE frontiers in education conference (FIE)*. IEEE, IEEE, 1–9. https://doi.org/10.1109/FIE.2016.7757692

[24] CHRISTOPHER D. HUNDHAUSEN, SARAH A. DOUGLAS, and JOHN T. STASKO. 2002. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages & Computing* 13, 3 (2002), 259–290. https://doi.org/10.1006/jvlc.2002.0237

[25] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, New York, NY, USA, 737–745. https://doi.org/10.1145/3126594.3126632

[26] Jun Kato, Sean McDirmid, and Xiang Cao. 2012. DejaVu: Integrated Support for Developing Interactive Camera-Based Programs. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology* (Cambridge, Massachusetts, USA) *(UIST '12)*. Association for Computing Machinery, New York, NY, USA, 189–196. https://doi.org/10.1145/2380116.2380142

[27] B.W. Kernighan and R. Pike. 1999. *The Practice of Programming*. Addison-Wesley. https://books.google.com/books?id=to6M9_dbjosC

[28] Taeyoung Kim, Suntae Kim, Sooyong Park, and YoungBeom Park. 2020. Automatic recommendation to appropriate log levels. *Software: Practice and Experience* 50, 3 (2020), 189–209. https://doi.org/10.1002/spe.2771 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2771

[29] Amy J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vienna, Austria) *(CHI '04)*. Association for Computing Machinery, New York, NY, USA, 151–158. https://doi.org/10.1145/985692.985712

[30] Amy J. Ko and Brad A. Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 301–310. https://doi.org/10.1145/1368088.1368130

[31] Amy J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering* 32, 12 (2006), 971–987. https://doi.org/10.1109/TSE.2006.116

[32] Thomas D. LaToza and Brad A. Myers. 2010. Developers Ask Reachability Questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 185–194. https://doi.org/10.1145/1806799.1806829

[33] Thomas D. LaToza and Brad A. Myers. 2011. Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 117–124. https://doi.org/10.1109/VLHCC.2011.6070388

[34] Sorin Lerner. 2020. Projection Boxes: On-the-Fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3313831.3376494

[35] Bil Lewis. 2003. Debugging Backwards in Time. *CoRR* cs.SE/0310016 (2003). http://arxiv.org/abs/cs/0310016

[36] Heng Li, Weiyi Shang, and Ahmed E Hassan. 2017. Which log level should developers choose for a new logging statement? *Empirical Software Engineering* 22, 4 (2017), 1684–1716.

[37] Tao Li, Yexi Jiang, Chunqiu Zeng, Bin Xia, Zheng Liu, Wubai Zhou, Xiaolong Zhu, Wentao Wang, Liang Zhang, Jun Wu, Li Xue, and Dewei Bao. 2017. FLAP: An End-to-End Event Log Analysis Platform for System Management. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Halifax, NS, Canada) *(KDD '17)*. Association for Computing Machinery, New York, NY, USA, 1547–1556. https://doi.org/10.1145/3097983.3098022

[38] Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing Misconceptions about Code with Always-on Programming Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) *(CHI '14)*. Association for Computing Machinery, New York, NY, USA, 2481–2490. https://doi.org/10.1145/2556288.2557409

[39] Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2021. Which Variables Should I Log? *IEEE Transactions on Software Engineering* 47, 9 (2021), 2012–2031. https://doi.org/10.1109/TSE.2019.2941943

[40] Mark Marron. 2018. Log++ Logging for a Cloud-Native World. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages* (Boston, MA, USA) *(DLS 2018)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/3276945.3276952

[41] Renée McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: a review of the literature from an educational perspective. *Computer Science Education* 18, 2 (2008), 67–92. https://doi.org/10.1080/08993400802114581 arXiv:https://doi.org/10.1080/08993400802114581

[42] Will McGrath, Daniel Drew, Jeremy Warner, Majeed Kazemitabaar, Mitchell Karchemsky, David Mellis, and Björn Hartmann. 2017. BifröSt: Visualizing and Checking Behavior of Embedded Systems across Hardware and Software. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, New York, NY, USA, 299–310. https://doi.org/10.1145/3126594.3126658

[43] William McGrath, Jeremy Warner, Mitchell Karchemsky, Andrew Head, Daniel Drew, and Bjoern Hartmann. 2018. WiFröSt: Bridging the Information Gap for Debugging of Networked Embedded Systems. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18)*. Association for Computing Machinery, New York, NY, USA, 447–455. https://doi.org/10.1145/3242587.3242668

[44] George A. Miller. 1956. The magical number seven plus or minus two: some limits on our capacity for processing information. *Psychological review* 63 2 (1956), 81–97.

[45] Robert S Moyer and Richard H Bayer. 1976. Mental comparison and the symbolic distance effect. *Cognitive Psychology* 8, 2 (1976), 228–246.

[46] Deokgun Park, Steven M. Drucker, Roland Fernandez, and Niklas Elmqvist. 2018. Atom: A Grammar for Unit Visualizations. *IEEE Transactions on Visualization and Computer Graphics* 24, 12 (2018), 3032–3043. https://doi.org/10.1109/TVCG.2017.2785807

[47] Strategic Planning. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology* (2002), 1.

[48] Nicolas Poggi, Vinod Muthusamy, David Carrera, and Rania Khalaf. 2013. Business process mining from e-commerce web logs. In *Business process management*. Springer, 65–80.

[49] Guillaume Pothier and Éric Tanter. 2009. Back to the Future: Omniscient Debugging. *IEEE Softw.* 26, 6 (nov 2009), 78–85. https://doi.org/10.1109/MS.2009.169

[50] Vineet Sinha, David Karger, and Rob Miller. 2005. Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases. In

Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology EXchange (San Diego, California) *(eclipse '05)*. Association for Computing Machinery, New York, NY, USA, 21–25. https://doi.org/10.1145/1117696.1117701

[51] Replay Solutions. 2022. Replay.io. https://www.replay.io/

[52] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Trans. Comput. Educ.* 13, 4, Article 15 (nov 2013), 64 pages. https://doi.org/10.1145/2490822

[53] Richard Stallman, Roland Pesch, Stan Shebs, et al. 1988. Debugging with GDB. *Free Software Foundation* 675 (1988).

[54] Matúš Sulír, Michaela Bačíková, Sergej Chodarev, and Jaroslav Porubän. 2018. Visual augmentation of source code editors: A systematic mapping study. *Journal of Visual Languages & Computing* 49 (2018), 46–59. https://doi.org/10.1016/j.jvlc.2018.10.001

[55] Tetsuji Takada and Takaaki Abe. 2018. Emoji-Nized Log Browser: Visualization of Server-Logs by Emoji for System Administrators. In *Proceedings of the 2018 International Conference on Advanced Visual Interfaces* (Castiglione della Pescaia, Grosseto, Italy) *(AVI '18)*. Association for Computing Machinery, New York, NY, USA, Article 86, 3 pages. https://doi.org/10.1145/3206505.3206578

[56] T. Takada and H. Koike. 2002. Tudumi: information visualization system for monitoring and auditing computer logs. In *Proceedings Sixth International Conference on Information Visualisation*. IEEE, 570–576. https://doi.org/10.1109/IV.2002.1028831

[57] Holly A Taylor and Barbara Tversky. 1996. Perspective in spatial descriptions. *Journal of memory and language* 35, 3 (1996), 371–391.

[58] Frank Tip. 1994. *A Survey of Program Slicing Techniques*. Technical Report. NLD.

[59] Barbara Tversky. 2001. Spatial schemas in depictions. In *Spatial schemas and abstract thought*, Vol. 79. 111.

[60] Bret Victor. 2012. Learnable Programming. *worrydream.com* (2012). http://worrydream.com/LearnableProgramming/

[61] Bret Victor. 2013. Drawing dynamic visualizations. *worrydream.com* (2013). http://worrydream.com/DrawingDynamicVisualizations/

[62] Anneliese von Mayrhauser and A. Marie Vans. 1997. Program Understanding Behavior during Debugging of Large Scale Software. In *Papers Presented at the Seventh Workshop on Empirical Studies of Programmers* (Alexandria, Virginia, USA) *(ESP '97)*. Association for Computing Machinery, New York, NY, USA, 157–179. https://doi.org/10.1145/266399.266414

[63] Haijun Xia, Nathalie Henry Riche, Fanny Chevalier, Bruno De Araujo, and Daniel Wigdor. 2018. DataInk: Direct and Creative Data-Oriented Drawing. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3173574.3173797

[64] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A Brief Survey of Program Slicing. *SIGSOFT Softw. Eng. Notes* 30, 2 (mar 2005), 1–36. https://doi.org/10.1145/1050849.1050865

[65] Litao Yan, Elena L. Glassman, and Tianyi Zhang. 2021. Visualizing Examples of Deep Neural Networks at Scale. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 313, 14 pages. https://doi.org/10.1145/3411764.3445654

[66] Geoffrey X. Yu, Tovi Grossman, and Gennady Pekhimenko. 2020. Skyline: Interactive In-Editor Computational Performance Profiling for Deep Neural Network Training. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 126–139. https://doi.org/10.1145/3379337.3415890

[67] Xiuming Yu, Meijing Li, Incheon Paik, and Keun Ho Ryu. 2012. Prediction of web user behavior by discovering temporal relational rules from web log data. In *International Conference on Database and Expert Systems Applications*. Springer, 31–38.

[68] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 293–306. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/yuan

[69] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *2012 34th International Conference on Software Engineering (ICSE)*. 102–112. https://doi.org/10.1109/ICSE.2012.6227202

[70] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2011. Improving Software Diagnosability via Log Enhancement. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1950365.1950369

[71] Andreas Zeller. 2002. Isolating Cause-Effect Chains from Computer Programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering* (Charleston, South Carolina, USA) *(SIGSOFT '02/FSE-10)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/587051.587053

[72] Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging*. Elsevier.

[73] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2015. Learning to Log: Helping Developers Make Informed Logging Decisions. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 415–425. https://doi.org/10.1109/ICSE.2015.60