The Journal of Systems and Software 207 (2024) 111834

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

# Application Monitoring for bug reproduction in web-based applications☆

Di Wang *, Matthias Galster, Miguel Morales-Trujillo

*University of Canterbury, Christchurch, New Zealand*

ARTICLE INFO

*Keywords:*
Bug reproduction
Debugging
Application monitoring
Web applications
Single page applications

ABSTRACT

Web applications are often built as Single Page Applications (SPA), for example applications offered by Google, Facebook, Twitter or Netflix. Users interact with SPAs through a single HTML page that is dynamically rewritten with new data from the web server (instead of a web browser that loads entire new HTML pages). Just like with any type of software system, debugging is a common activity during the development and maintenance of SPAs. In order to fix bugs observed during runtime, developers often try to reproduce the bug first to better understand it. However, research has shown that reproducing bugs is not always possible. In this paper we (i) develop a technique for Application Monitoring (AM) to collect data to support bug reproduction; and (ii) apply the monitoring technique in a SPA test bed as well as a real-world SPA application to show its feasibility. As part of our research we developed an initial version of the AM technique and implemented it in a prototype. Our evaluation using this prototype showed that it not only improves the efficiency of the bug reproduction process but also reduces information gaps caused by incomplete bug reports submitted by users. Additionally, compared to the information provided by users, data provided by AM is more accurate and detailed and covers a wider range of data. Future work includes deploying the AM framework in more SPAs and investigating how AM can be integrated into software developer workflows.

## 1. Introduction

### 1.1. Problem and motivation

Modern web-based applications are often built as Single Page Applications (SPA) using frameworks such as Angular[1] and React[2] (Diniz-Junior et al., 2022). SPAs are web applications that interact with users by dynamically rewriting a web page with new data from the web server, instead of a web browser loading an entirely new web page. Unlike "traditional" web applications that perform most of the application logic on the web server, SPAs perform most of the user interface (UI) logic in a web browser, communicating with the web server primarily using web APIs. Examples of applications built as SPA include Google, Facebook, Twitter and Netflix (Garg, 2022).

Just like other types of software systems, SPAs have bugs that need to be fixed during development and maintenance. To fix bugs observed during runtime, it is useful to reproduce the bug to better understand it (Roehm et al., 2013). However, research has shown that reproducing bugs is not always possible (Zimmermann et al., 2010). One main reason for why bugs cannot be reproduced is the *lack of information needed* to reproduce a bug (Rahman et al., 2020). For example, bug reports submitted by users may miss critical information (e.g., when the bug occurred, what behavior it triggered, or what the state of the program was at the time the bug occurred) (Chaparro et al., 2017). Also, poor cooperation of clients and end users to fully understand a bug contributes to difficulties when reproducing bugs (Joorabchi et al., 2014). Furthermore, SPA are more complex compared to traditional static web pages, so collecting data required to understand user behavior that may have caused a problem becomes more challenging. Therefore, approaches for collecting relevant information for bug reproduction in SPA web applications and presenting them to developers in a meaningful way would support developers to reproduce (and eventually fix) bugs. For example, previous research has shown how screenshots help with bug reproduction (Sasaki et al., 2011). In this paper, we will go one step further to investigate how application monitoring can help with bug reproduction for web applications. For this purpose, we developed two research questions (RQs):

- **RQ1: How can application monitoring support bug reproduction?** The aim of this RQ is to understand the impact of application monitoring on bug reproduction. As part of answering it, we also develop a new technique for application monitoring geared towards bug reproduction in the context of SPA.

☆ Editor: Antonia Bertolino.

* Corresponding author.
*E-mail addresses:* di.wang@pg.canterbury.ac.nz (D. Wang), mgalster@ieee.org (M. Galster), miguel.morales@canterbury.ac.nz (M. Morales-Trujillo).

[1] https://angular.io/

[2] https://reactjs.org/

https://doi.org/10.1016/j.jss.2023.111834
Received 21 February 2023; Received in revised form 28 June 2023; Accepted 3 September 2023
Available online 9 September 2023

- **RQ2: How does application monitoring impact the performance of the software?** As part of RQ1 we develop a new technique for application monitoring. However, continuously monitoring applications may impact the performance of the monitored system, limiting the practical feasibility of application monitoring for SPA. Therefore, this RQ helps us understand how much impact and overhead application monitoring adds to the monitored software.

### 1.2. Paper contributions

In this paper, we investigated how application monitoring can support bug reproduction for SPAs and how it impacts the performance of the monitored software:

- We develop a test bed of a representative SPA for web application monitoring. This test bed can also be used as a benchmark system in other studies.
- We provide a web application monitoring technique. This technique monitors applications to collect data typically required to reproduce bugs, but often not provided by end users in their bug reports.
- We develop a prototype implementation of the technique that shows its feasibility and is available as a JavaScript library.
- We evaluate the effectiveness of our technique by applying it to a commercial software and reproducing previously non-reproducible bugs. Results showed that application monitoring improved efficiency for bug reproduction, required less customer involvement to understand bugs, and fewer people involved in reproducing them.
- We evaluate the performance impact of our application monitoring prototype by using the test bed system and comparing performance overheads in four different setups.

The paper follows the Empirical Standards for Engineering Research[3]: We (1) describe the proposed artifact in adequate detail (Section 5), (2) justify the need for, usefulness and relevance of the proposed artifact (Section 1, Section 2), (3) conceptually evaluate the proposed artifact; discusses its strengths, weaknesses and limitations (Section 8), (4) empirically evaluate the proposed artifact (Section 7). Section 2 discusses the background and related work. Section 3 gives a motivating example to illustrate the problem our study aims to address and Section 4 describes the test bed we developed for the study. Section 5 presents the overview of our proposed technique for application monitoring for bug reproduction. In Section 6 we discuss a prototype implementation of our technique. Section 7 describes the research methods we applied to evaluate the technique and discusses the outcomes of applying our web application monitoring technique in different settings to show its usefulness. Section 8 discusses limitations and threats to validity of our work, and finally Section 9 concludes this paper.

## 2. Background and related work

### 2.1. Key concepts

Information such as end user usage data and application execution logs can help developers understand the underlying problem of a bug and eventually help reproduce it (Goyal and Sardana, 2018; Rahman et al., 2020). There are various data sources for such information, like bug reports and application monitoring. **In this study we will look into how application monitoring data can support bug reproduction.** Below we define some key concepts related to our work:

- **Bug** is used as an umbrella concept for failure (a software service deviates from its correct behavior), error (a condition in a system that may lead to a failure), and fault (the underlying cause of an error in the source code) (Avizienis et al., 2004). This means, we do not differentiate bugs from software failures, errors and faults, even though failures could be considered as a consequence of a bug. Furthermore, in this work we focus on "functional" bugs, i.e., bugs that break the application workflow and prevent the user from completing their task. Other problems, such as security vulnerabilities and usability are not in the scope of this paper.
- **Bug reports** are the documents that provide information about the bugs users experienced. Bug reports can be in the form of emails from end users, tickets from support teams, failed tests report from testing teams, among others. In this work we focus on bug reports submitted by end users.
- **Bug reproduction** is the process to reproduce the error event reported by the user to make sure the bug is correctly reported and to better understand the circumstances of a bug. This helps locate the bug and eventually fix it.
- **Application monitoring data**, is data collected at runtime about user behavior and application context. There are two kinds of monitoring data: (1) end user usage data (i.e., data of what ens users did and that developers can use to "replay" user interactions); and (2) context data (i.e., additional data such as errors logged in the console or http requests submitted when bugs occurred). In Section 5.2 we will explain what data we consider for our application monitoring technique, including justifications for it.

### 2.2. Web application monitoring

Compared to desktop application monitoring, web application monitoring has its own characteristics. For example, a web application generally has its client running in a user's browser and exchanging data with external web servers. Therefore, an application monitoring system for web applications needs to collect data about user actions, execution logs, etc. from local applications (e.g., web browsers), but also about data exchange events with external web servers and services. While there are many studies around application monitoring (Ray and Banerjee, 2011; Dimitrov and Zhou, 2011; Mao et al., 2011; Yu et al., 2011; Mao, 2011; Zhang et al., 2012; Wong et al., 2012; Roychowdhury and Khurshid, 2012; Yousefi and Wassyng, 2013; Huang and Ai, 2015; Asadollah et al., 2016; DeMott et al., 2013), only a few focus on web applications.

In 2005, Sprenkle et al. conducted a study around web application monitoring, web session replaying and fault detection (Sprenkle et al., 2005). The paper proposed an approach to monitor HTML outputs (e.g., generated web pages) of a web application implemented in Java. It uses HTML outputs for session replaying and fault detection. Bug detection is done by an algorithm that evaluates the difference between output HTML and expected HTML. The approach proposed in the study is evaluated in an experiment conducted with a Java-based web application. One of the main limitations of this study is it assumes that the web application is delivered by static HTML pages, and data exchange between clients and server is done by regenerating HTML pages. However, modern SPAs only deliver static pages once and subsequent data exchange is done by API requests that updates the UI according to the response.

Choudhary and Orso developed an approach for web application monitoring (Choudhary and Orso, 2009) where a client-side agent collects data such as JavaScript errors and usage data from end users. A server-side agent stores collected data. Compared to other approaches for program monitoring in which monitoring is a passive action, this approach can send commands from the server to clients for executing some actions (e.g., to collect metrics from the client).

[3] https://github.com/acmsigsoft/EmpiricalStandards/blob/master/docs/EngineeringResearch.md

Another study by Kiciman and Wang discusses adaptive application monitoring (Kiciman and Wang, 2007). The proposed approach deploys differently instrumented (i.e., monitored) versions of applications over time and across users. This allows developers to get insights into end-to-end user interaction of web applications addressing developer needs dynamically. With the information collected specifically for each version, developers can understand and reproduce the software bug more effectively and efficiently. One drawback of the approach proposed in the study is that the SPA client-side code is cached until a user refreshes a web page. Therefore, the new code will not be immediately deployed, causing a delay from hours to days depending on how often the end-user refreshes the page.

### 2.3. Design considerations for web application monitoring

Several studies that explored performance monitoring of web applications defined particular aspects to be considered when designing a monitoring system for web applications. Al-Shammari and Husein (2020) argue that an application with an internal storage system (e.g., a file system or database) has longer response times compared to applications that do depend on an external storage system (e.g., an external database). This suggests that a reliable storage strategy (e.g., caching, asynchronous writing) will reduce the overhead for monitoring systems.

Magalhaes and Silva also looked into the impact of monitoring on the performance of web applications (Magalhães and Silva, 2013). Instead of monitoring every transaction in the system, they suggest selective monitoring to only log events that meet conditions defined by developers or users (e.g., only monitor sessions when user is logged in). Selective monitoring significantly reduces performance overheads. We experimented with selective monitoring to reduce the overhead.

In 2019, Filip and Cegan compared tools (Smartlook, HotJar, Inspectlet, Lucky Orange, Capturly) available for web session recording and replaying (Filip and Cegan, 2019). They summarized what functionalities and characteristics those tools have. The identified functionalities are:

- Session recording and replaying: record user interaction while using the software and replay actions when needed.
- Console logging: log run time console data.
- Click-maps: heat map based on click coordinates, target element, device resolution and web page content.
- Scroll-maps: heat map which shows how far a user scrolls down a web page.
- Business funnels: description of how users reached the end of a business transaction (e.g., page visit order when purchasing on an online store).
- A/B testing: ability for test-based comparison of two web site versions.

The identified characteristics which impact the practicality of tools include:

- Library size: size of monitoring library added to the web application (may impact on the load time).
- Transferred data: amount of data the monitoring system transferred during runtime.

Features related to session recording and console logging, and characteristics like library size and transferred data are also relevant to our goal of supporting bug reproduction. On the other hand, several functionalities mentioned above are more about understanding how users utilize the software and are therefore less of a concern for our work.

### 2.4. Commercial tools for application monitoring

In this section, we discuss three popular tools that are frequently mentioned on Stack Overflow and which are ranked high and recommended on G2.com (as of November 2021), a popular website used by developers to find software solutions: (1) Azure Application Insights,[4] an application monitoring system integrated on a cloud platform; (2) Data Dog[5]; and (3) Smart Look.[6] We briefly discuss the features of each tool and the gaps related to web application monitoring (i.e., collecting execution data, such as exception logs, background execution logs) and session replaying (i.e., collecting end user usage data and presenting it to developers for debugging). In Table 1 we compare the tools based on the functionalities provided and the performance impact of each. Three main functionalities are included:

- **Execution monitoring** tracks the software execution status and the data exchange between the client end and the server.
- **Session replay** records users interactions with the software and helps developers to understand users behaviors by reproducing user interactions.
- **On demand session replay** is an extra functionality build on session replay which allows users and developers to turn on the session replay function when needed to reduce the overhead and performance impact of continuously recording user interactions.

In Table 1 we also compare the types of data that each tool collects (this is based on the information needs of developers as discussed later in Section 5.2). As shown below, while web application monitoring is common in practice and supported by commercial tools, no solutions exist for the purpose of monitoring SPA to support bug reproduction.

#### 2.4.1. Azure application insights

Application Insights is a tool provided by Microsoft to help developers track the usage and runtime status of their application. The tool can be plugged into most web applications by either configuring the application via the Azure portal or by adding a piece of code to an application. Out of the box features include: (1) tracking users' behavior data (e.g., page visits); (2) tracking the status of HTTP requests (e.g., response time of requests); and (3) monitoring the server status (e.g. server RAM and CPU consumption). Further custom configurations allow collecting data, such as database query logs. While Application Insights collects data of user behavior, there is no session replaying feature. Furthermore, in order to reduce the data overhead, Application Insights samples data (e.g., page visit data is provided as the sum of how many times a page is visited rather than data of each page visit). This makes it difficult to use data for session replaying and bug reproduction. While data sampling can be "turned off", this would increase the data consumption which then leads to higher costs (Application Insights operates on a subscription model). On the other hand, if a web application is hosted on Azure's infrastructure, Application Insights provides good integration with other Azure services for analytics (e.g., Azure Cognitive Service, Azure Functions, etc.).

#### 2.4.2. Data dog

Data dog provides most features that Application Insights offers for web application monitoring. Furthermore, Data Dog provides functions for detailed user behavior tracking, such as click and scroll map (features that were also identified in the tool comparison by Filip and Cegan, 2019). There is also a session recording and replaying feature offered as a beta feature. However, the negative performance impact of Data Dog on the monitored web application reduces the quality

[4] https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview

[5] https://www.datadoghq.com/

[6] https://www.smartlook.com/

**Table 1**
Commercial tools for application monitoring and session replaying (Y: supported, N: not supported, N/A: not applicable as related feature is not supported).

| | Application insights | Data dog | Smart look |
|---|---|---|---|
| Execution monitoring (EM) | Y | Y | N |
| Performance impact of EM | Low | Low | N/A |
| Session Replay (SR) | N | Y | Y |
| Performance impact of SR | N/A | High | Low |
| On demand SR | N/A | Y | N |
| Data collected | | | |
| User interaction data | N | Y | N |
| Client–server communication data | Y | Y | Y |
| Execution event data | Y | Y | N |
| Other data | Y | Y | Y |

of the user experience. Also, session replaying focuses on reproducing what users did rather than what happens at the back-end (e.g., HTTP requests made when the user clicked a button). Back-end execution data is tracked, but stored in another part of the system which makes it difficult to map the session replaying timeline to the back-end execution data timeline to assist application diagnostics.

Similar to Application Insights, the cost for session recording in Data Dog is significant. To reduce costs, Data Dog offers on-demand session recording where developers can customize code to start and stop session recording (e.g., to start session recording when an HTTP request returns an error code). Session recording and replaying works with most standard web technologies, but it does not work with SPA frameworks, such as Angular.

#### 2.4.3. Smart look

Compared to Application Insights and Data Dog, Smart Look focuses on session recording and replaying only. It does not provide functions to record context data, such as HTTP requests. This means that the performance overhead caused by Smart Look is low. However, similar to Data Dog, Smart Look also does not work with SPAs. Furthermore, compared to Data Dog, session recording is activated all the time, which may result in high execution costs.

#### 2.4.4. Summary

We evaluated three popular commercial system monitoring and session replaying tools that are frequently mentioned on Stack Overflow. In Table 1 we provided a summary of the tools and a brief comparison of execution monitoring and session replaying. The types of data in this comparison (user interaction data, client–server communication data, execution event data, other) are based on the information needs of developers to reproduce bugs (see Section 5.2). Other monitoring tools also exist, for example, Raygun[7] monitors web applications, but does not provide the type of data we are interested in (e.g., these tools collect data about sessions, users, etc. but not detailed user interactions with the system).

In summary, although application monitoring has been explored in both the academic and commercial worlds, there are still gaps to be filled in web application monitoring to support bug reproduction. Most importantly, no studies explored system monitoring for SPAs (e.g., using frameworks like Angular, React,[8] etc.). In SPAs, static content such as HTML and JavaScript are only delivered once to a client. Therefore, system monitoring based on HTML pages generated by a web server is not applicable to SPAs. Furthermore, previously proposed approaches and studies (Sprenkle et al., 2005; Choudhary and Orso, 2009; Kiciman and Wang, 2007) only focus on replaying sessions of user interactions (e.g., clicks on a page, pages visited etc.), but do not record more detailed context data that can support bug reproduction (e.g., what HTTP requests are made to get the data from web server). Also, there is no study about how application monitoring can impact bug reproduction for SPA. Therefore, in this study we will explore how application monitoring can help satisfy the information needs of developers when trying to reproduce a bug.

## 3. Motivating example

We present a motivating practical example to illustrate our problem. The motivating example is based on real-world scenarios as they frequently occur in a medium-sized software company in the business process automation (Lazareva et al., 2022) domain.[9]

A typical starting point for debugging are bug reports. According to previous studies (Zimmermann et al., 2010; Chaparro et al., 2017), the quality of bug reports and the lack of detailed information impact the reproducibility of a bug. Fig. 1 shows an example of a bug report from a real project for a web application submitted by an end user via a bug tracker, in this case, JIRA.[10] The only meaningful information in this bug report is what the end user was trying to do and the problem they faced. However, there is no information that would allow a developer to reproduce the problem. Information such as which screen the user was on, what actions were performed by the user and what data the user entered are missing. This information is critical for developers to understand and reproduce the problem. After several rounds of communication between the support team in the software development organization and the end user, the bug report was updated with more information (see Fig. 2). Developers were then able to understand the bug better, and after some trial-and-error were able to reproduce the bug (and eventually fix it).

In the given context, communicating with the end user and obtaining additional information usually takes considerable time as the email communication between the support team and end users introduces delays. On the other hand, if the extra information collected by the support team was available already at the time that a bug was reported, then the time to fix this bug could be significantly reduced.

## 4. Test bed

We present the development of a test bed that is used to later in the development and evaluation of our application monitoring technique. Also, explaining the test bed design helps further illustrate the scope of the problem. Finally, the test bed can be used by others as a benchmark system for web SPAs monitoring. The test bed aims to represent common SPAs in general.

We developed a representative test bed system, a literature management system for systematic mapping studies. We chose this domain because most readers of software engineering research should be familiar with literature reviews, so there is no specific domain knowledge required. By "representative" we mean the following:

- The used technologies to implement the test bed resemble those commonly used to build SPAs (e.g., HTML, CSS, JavaScript, TypeScript, Angular framework, REST API, etc.).
- The types of user interactions supported in the test bed application are common to web applications: creating data entries, searching data, viewing details of data, and updating and deleting data.
- Additional characteristics of the test bed application resemble those common to a web application:
  - Provide access control for users via user profiles.

[7] https://raygun.com/documentation/product-guides/real-user-monitoring/for-web/sessions
[8] https://reactjs.org
[9] Details about the company are not included due to confidentiality reasons.
[10] https://www.atlassian.com/software/jira

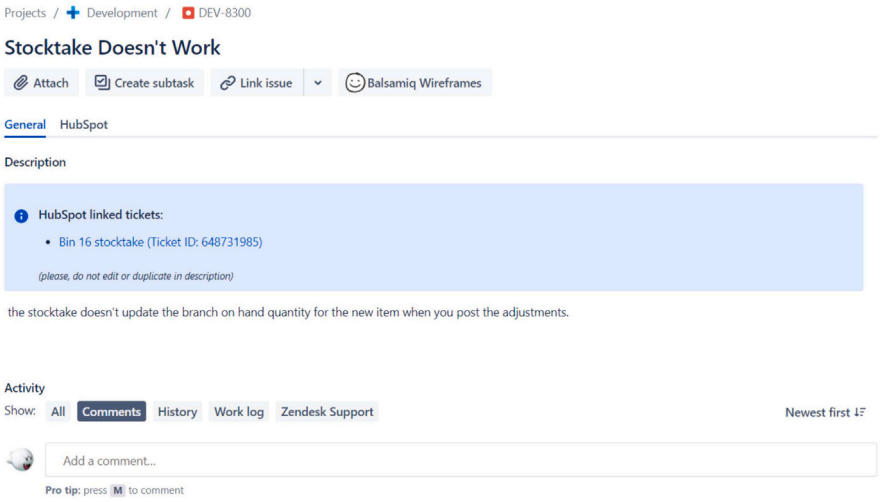


**Fig. 1.** Initial version of bug report.

- Keep data in a persistent storage service, such as a database.
- Allow users to perform common tasks (e.g., querying and updating data in a persistent data storage).
- Provide a usable graphical UI to allow users to interact with the application.

The test bed is developed with Angular, a frequently used front-end framework for SPAs (Uluca, 2018). Data exchange is done via a REST API developed with .Net core/C#. The endpoint is hosted on a Windows server with an Internet Information Services (IIS) server. The file storage for the web application is using the server's file system and the data storage is proved by MS SQL 2019 hosted on the same server as the IIS. The source code of test bed can be found on GitHub[11]

## 5. Application monitoring for web SPA

In this section we present an overview of our technique for web application monitoring.

### 5.1. Overview

The design of our technique is driven by the need to fit into the workflow of software developers. In general, when a software bug is reported by end users, it will be recorded in a software bug report. Then (as discussed above), to fix the bug, developers need to reproduce the bug first. However, the information provided by users and in the bug report are not always detailed enough. As shown in Fig. 3, in a typical attempt to reproduce bugs, developers and support teams need to rely on end users to collect the required information. However, as discussed in our motivating example, not only does this process take a lot of time, but also the required information may not always be available from end users (end users may not remember the information required, or their work environment has changed since the bug appeared, e.g., browser updates, etc.). Therefore, instead of asking end users for relevant information, we use monitoring to collect information from the software while the user is using it. In case a bug appears, we query the collected monitoring data to help with reproduction.

The overall idea for our approach is to collect data relevant to satisfy the information needs of developers when reproducing bugs. Our technique start with three categories of monitoring data to support bug reproduction as identified in previous studies on the difficulty of reproducing bugs (Sprenkle et al., 2005; Choudhary and Orso, 2009; Al-Shammari and Husein, 2020; Magalhães and Silva, 2013; Filip and Cegan, 2019): (1) user interaction data; (2) client–server communication data; and (3) execution event data. Furthermore, we include "other" types of data as a fourth category. We discuss these categories of data in more detail below.

The design of our technique has three three parts, (see Fig. 4): (1) a client end monitoring component focusing on data collection, (2) a presentation component to present collected data in a meaningful way to developers, and (3) a logic and data storage component that provide business logic and storage to save and query collected monitoring data.

### 5.2. Client end monitoring component

In order to find out the information gap developers face for bug reproduction and to satisfy the information needs of developers, we use the four data categories introduced above:

1. **User interaction data:** The aim of collecting user interaction data is to understand users' actions within the application, e.g., what web pages and parts of pages they view and what actions they perform. In previous work (Sprenkle et al., 2005) it is suggested to collect user interaction data by storing HTML code generated by a server, however, an SPA's HTML code does not change after the initial deployment to the client. Therefore, approaches which monitor HTML code from the server are not applicable to SPA. We collect user interaction data by monitoring the client-end DOM event and collecting relevant data.
2. **Client–server communication data:** This data is collected to understand the data exchange between clients (and a user's browser) and code on a web server. Since the most common way for modern web applications to exchange data between a client and a server is via REST APIs (Rodríguez et al., 2016), our approach aims to collect all REST API requests for a complete understanding of communication between a client and the web server.
3. **Execution event data:** Event data is used to understand the running status of an application. Event data helps developers and support teams understand an application's runtime status of when the user experienced a problem. Critical event execution logs and execution error details are considered as event execution data and are typically collected from console logs.
4. **Other data:** Other data are collected for developers to understand the application execution environment and status. This includes browser and operating system types and their versions.

[11] https://github.com/Alantod17/UcTracking/tree/master

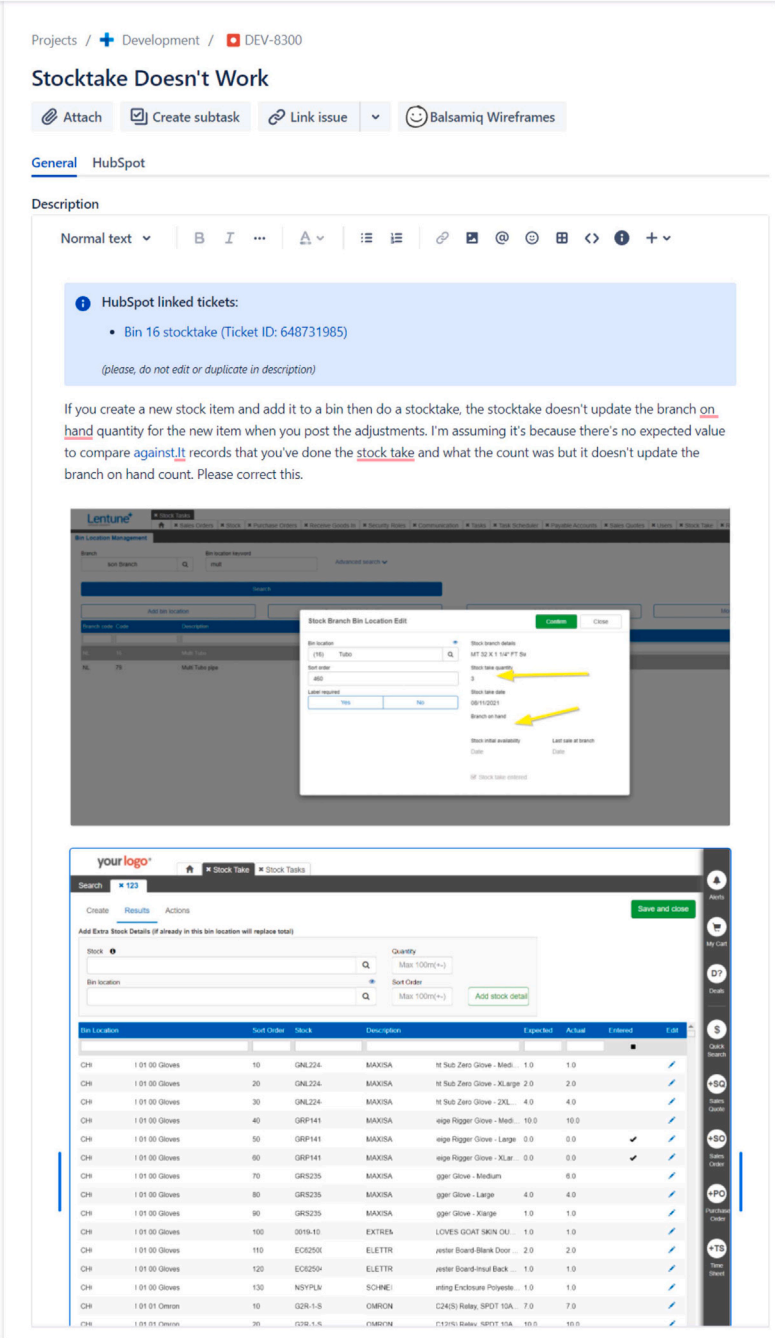


**Fig. 2.** Updated (and final) version of bug report.

### 5.3. Presentation component

This component provides a platform for developers and support teams to query and view the collected monitoring data. The presentation component has two sub-components: a data query component and data view component. The data query component allows developers to query the monitoring data by different criteria based on information available to developers in bug reports. For example, if a bug report includes information about the time of an event, developers can find the correct user session by querying the start and end time of an event (an example screenshot of the data query component where the user searched for events that happened on February 28, 2022 is shown in Fig. 5). After locating the correct session, developers can explore detailed data of the event and access interaction data, client–server communication data and execution event data to understand and reproduce a bug (an example screenshot of the data view component is shown in Fig. 6).

### 5.4. Logic and data storage component

The last part of our technique is the logic and data storage component. The data collected on the client is saved to the storage place. The presentation component uses the endpoints provided to query the monitoring data and present it to developers for bug reproduction. For example, the client monitoring component sends data to this component by using the API provided and the data will be processed and saved to the appropriate storage facility chosen.

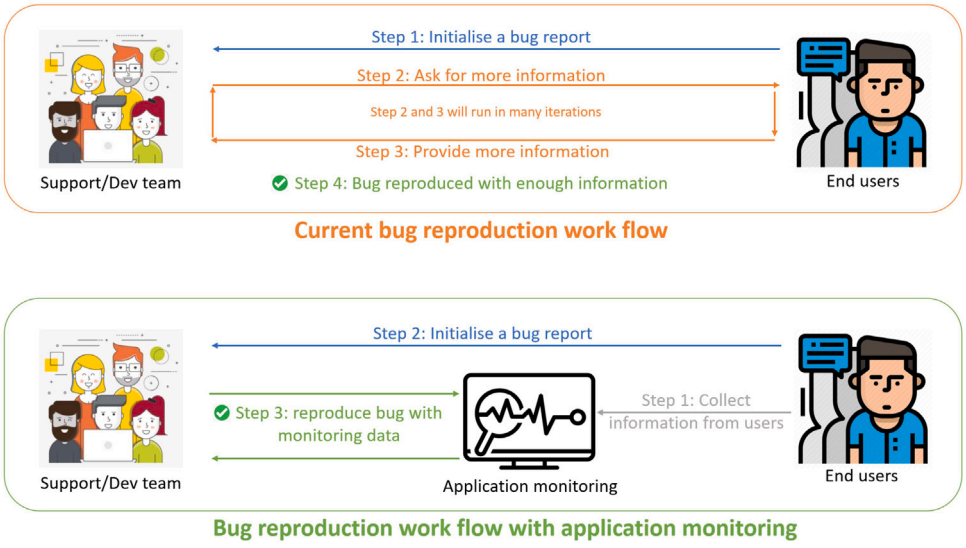


**Fig. 3.** Bug reproduction workflow.

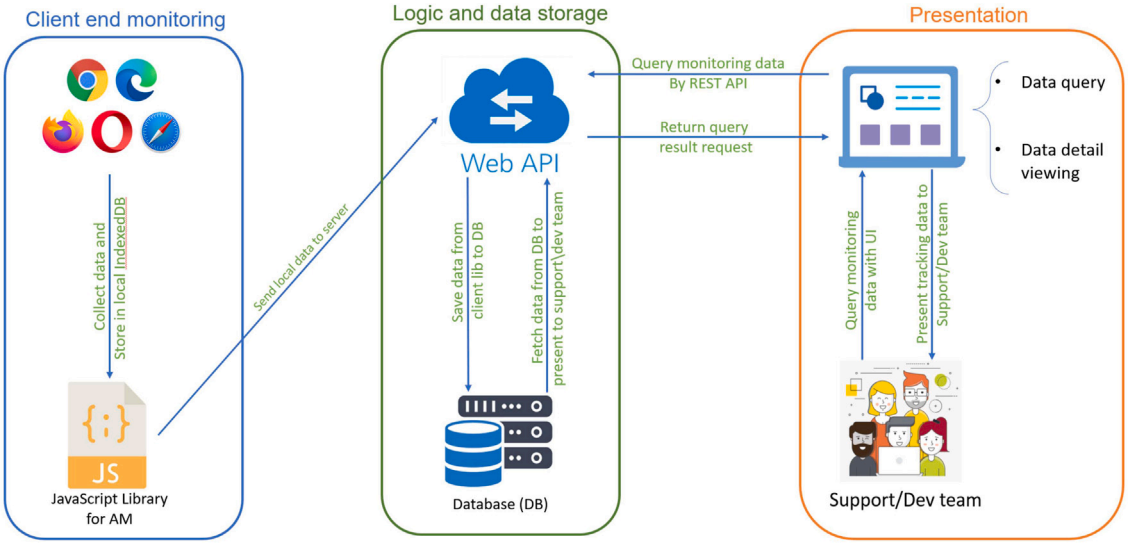


**Fig. 4.** Overview of proposed application monitoring technique.

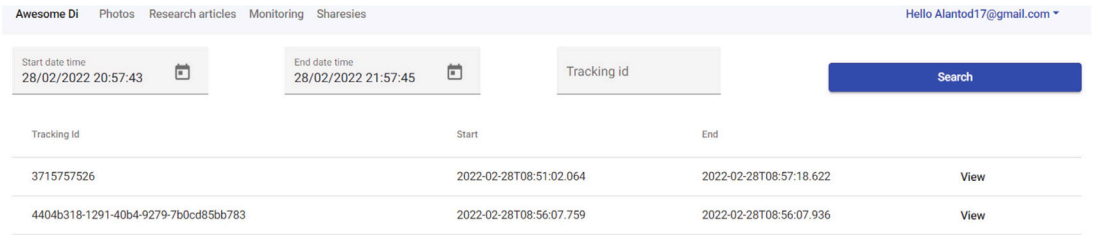


**Fig. 5.** "Data query view" in prototype.

## 6. Prototype implementation

The prototype implementation of our technique shows the feasibility of our work as well as how exactly we can collect the required monitoring data. The reference implementation is available as an online demonstrator.[12]

[12] https://github.com/Alantod17/UcTracking

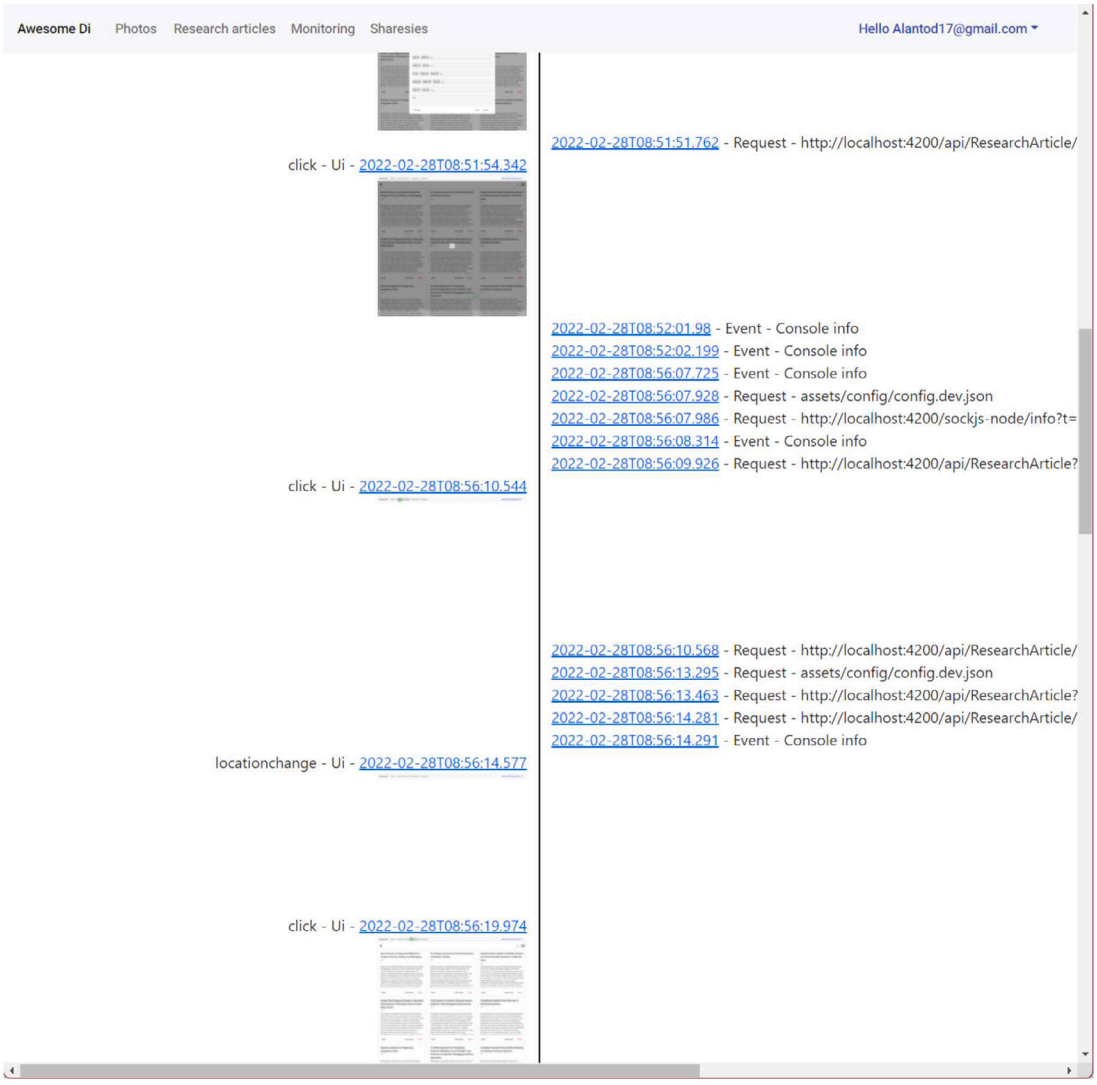


**Fig. 6.** "Detail view" in prototype.

### 6.1. Client end monitoring

We developed the client end monitoring component as a JavaScript library that developers can include in their client application. Our prototype implementation stores all data in IndexedDB,[13] which is available for most modern browsers. Below we explain how we collect the four types of data required by our technique:

- **Common data:** Some common data is collected for all four types (user interaction data, client–server communication data, execution event data, other) to help support/developer teams to understand the context of the data:
  - DateTime: Time the data was collected, used when we present event data in a time line.
  - Href: Screen Href when data is collected, identifies the screen a user was on when event happened.
  - TrackingId: Session tracking ID, used to identify the session of a page visit.
- **User interaction data:** This is collected by taking a screenshot at "event points" (e.g., when a is page loaded, a button clicked) and "route change events" (i.e., when the URL changes in the address bar). To obtain screenshots, we used HTML2Canvas,[14] a third-party library. At the same time, we also collect meta-data to track user interactions:
  - UiDataId: Unique ID for a UI interaction
  - Trigger: Event that triggered the action (e.g., route changed and button clicked); helps understand user's action
  - PageWidth and PageHeight: Current window's width/height; used to understand user's view size
  - MouseX and MouseY: Current mouse location, used to understand mouse location when event happened
- **Client–server communication data:** For each request we intercept XMLHttpRequest's open, send and load events and collect the following data:
  - RequestId: A unique id to keep in track of each request instance

[13] The Indexed Database API is a JavaScript API provided by web browsers for managing a NoSQL database of JSON objects. It is a standard maintained by the World Wide Web Consortium.

[14] https://html2canvas.hertzen.com/

  - EndPoint: API end point called; will be used to help of understanding which REST API endpoint was called
  - Method: REST request method; will be used to help of understanding the method used for the REST API endpoint that was called
  - StartTime: Date and time request was send; used together with EndTime to track the time spent on request
  - Parameter: Request parameter sent; used to understand request action
  - EndTime: Date and time the request ended; used together with StartTime to help with understand duration of request
  - Result: Result returned; used to track the data outcome of request
  - ResponseCode: Response code returned; used to track the outcome status of the request

  For storage efficiency our prototype only keeps the first 5,000 characters if a parameter's JSON string length is longer than 5,000 characters.
- **Execution event data:** Our implementation adds a log method before calling the original JavaScript's console methods to save all log data to the IndexedDB:
  - EventId: Unique ID for the event
  - Type: Console method called for the event (log, error, debug, warn and info); will be used to understand the type of information logged
  - EventDetail: Event details logged (e.g., the error thrown by the application or runtime information logged to the console)
- **Other data:** We collect runtime environment data from a browser's navigator property.[15] These data provide additional information to fully understand the context in which a bug occurred:
  - DataId: A unique ID for data item
  - DateTime: Time the event happened; used when we present data on a time line
  - TrackingId: Session tracking ID is used to identify the session of page visit
  - DataName: Name of the data property (browser type, browser version, operation system, operating system version)
  - DataValue: Actual value of the data collected

### 6.2. Presentation

Communication between the client end monitoring component and the presentation component happens via an open REST API endpoint. The data types are sent in different intervals and batch sizes per request (the interval below is refined with multiple test runs, and the final intervals set below for each data category were the optimum values considering the size of data and POST body size limits of REST API requests):

- User interaction data is sent every three minutes with two records in a request.
- Client–server communication data is sent every five minutes with 50 records in a requests.
- Execution event data is sent every five minutes with ten records in a requests.
- Other data is sent every five minutes with 10 records in a requests.

To present data, we built a web application with Angular as front-end which communicates with back-end by REST API. The UI is accessible only by an admin team. All end points are protected by Oauth 2 authentication with bear token. As shown in Fig. 5, the main UI allows support/development teams to query data based on the time and tracking ID provided by a user (e.g., in the bug report). Fig. 6 shows how the data for a particular user session is presented on a timeline, including a screenshot of the web page on the left side and detailed session data on the right side. By showing user interaction data, client–server communication data and event data, we provide a more complete picture of a problematic situation for the debugging process. Users can also click the link to each event to get a more detail about an event, see Fig. 7.

### 6.3. Logic and data storage

The last component of our technique is implemented as a REST API to allow the client side monitoring to save data from the local IndexedDB to the Microsoft SQL server database and the presentation component to query and present the data collected to the developers. The REST API and business logic is implemented with .Net 5 WebAPI, while the database storage uses Microsoft SQL server.

## 7. Evaluation

### 7.1. Research method

As suggested by the Empirical Standard for Engineering Research,[16] we evaluate our new technique using case study research "in which the researchers observe a real organization using the artifact". We conducted two case studies as proposed in Host et al. (2012) and are described below. The first study focuses on the *usefulness* of our technique and its potential to help reproduce bugs. The second study focuses on analyzing potential performance implications of continuous monitoring to assess the practicality of our technique.

#### 7.1.1. Case study 1: Support for bug reproduction

- **Objective:** The objective of this study is to answer RQ1: How can application monitoring support bug reproduction? The main aspects we investigate are the efficiency of the bug reproduction process and the effectiveness of data collected from application monitoring. We do this with *real* bug reports for a *real* system currently being developed by a commercial software company. As pointed out by Gopinath et al. (2014), artificial bugs used in studies may not necessarily represent a real-world bug and related challenges.
- **Case and units of analysis:** The case is a commercial software development project and the units of analysis are bug reports for that project and the information needs of developers that application monitoring will address. The commercial software we are working with is an ERP software (Mu, 2021) designed specifically for wholesalers. It provides extensive accurate reporting and provides features such as segmented discounting, automated supplier discounting, extensive reporting, overseas importing, banking integration, integrated payroll, and integrated CRM. The developing company has been working with large wholesalers for over 15 years. Customers are mostly from Australia and New Zealand. Five recent bug reports that were submitted in the last three months prior to the study were selected. These bug reports were originally marked as not reproducible by the developers. After spending significant effort obtaining the information missing in the bug reports, the bugs could be reproduced.

[15] https://developer.mozilla.org/en-US/docs/Web/API/Window/navigator

[16] https://github.com/acmsigsoft/EmpiricalStandards/blob/master/docs/EngineeringResearch.md

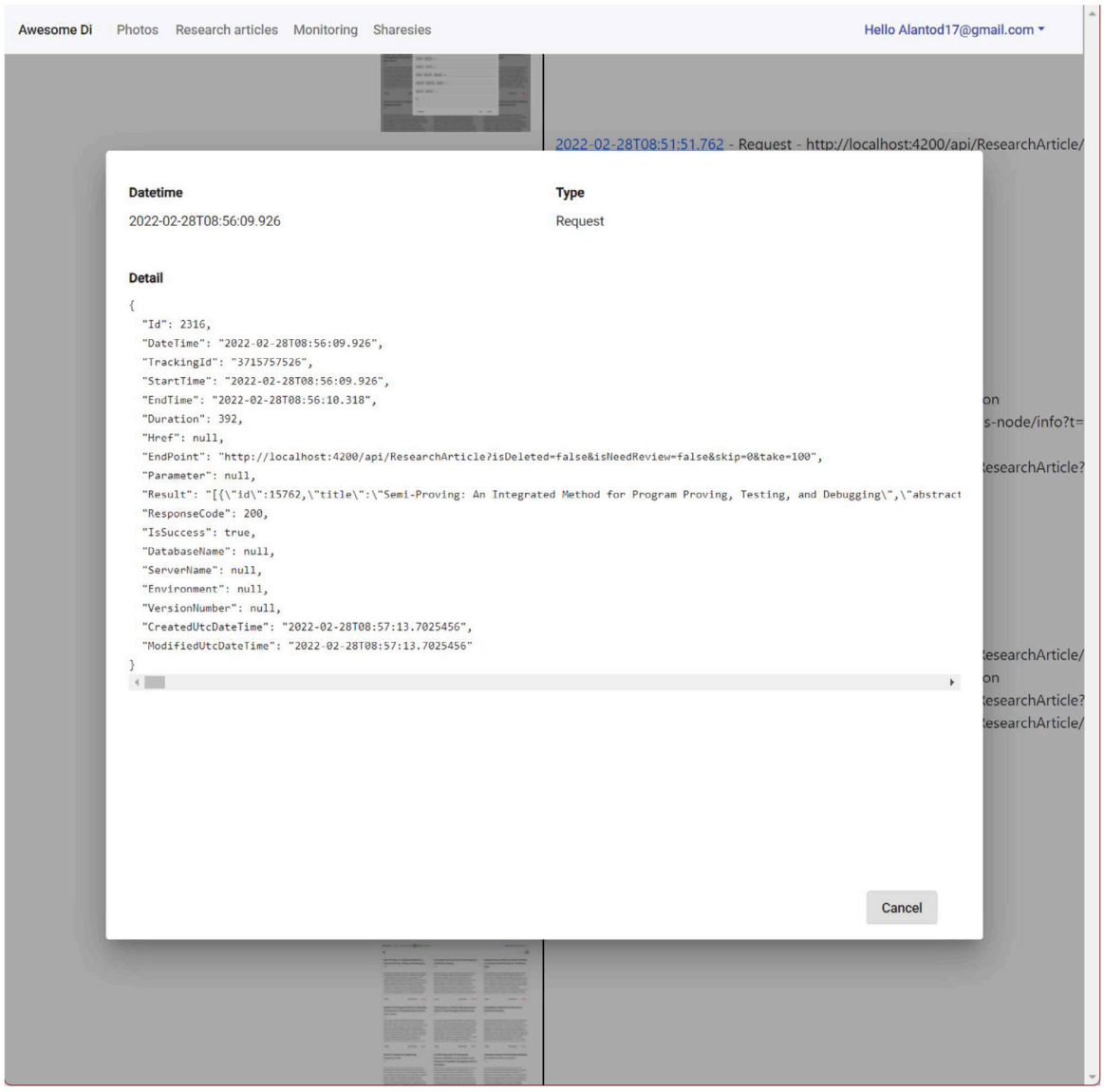


**Fig. 7.** "Event detail view" in prototype.

- **Data collection and analysis:** To evaluate how application monitoring impacts bug reproduction, we collected an initial base line based on five bug reports (issue cards) on JIRA. Information about initial attempts to reproduce the related bugs are collected from the cards' history and comments added to JIRA cards. We then set up the commercial project in a testing environment, i.e., cloned the production environment from the production's virtual machine backup. The cloned environment includes the code versions from before any of the five bug reports had been fixed. We then plugged our prototype into the project and reproduced the usage actions that end users performed and that triggered the bug. Then, we gave a developer who was not involved in fixing the bug access to the data collected by our application monitoring system. During the case study, the following quantitative data was collected:
  - Time to reproduce bug: Time it takes developer to reproduce the bug with and without the information provided by the application monitoring system.
  - Number of involved developers: Number of developers involved to reproduce the bug, with and without application monitoring system.
  - Interactions between developers and users: Number of interactions between developer(s) and user(s) during bug reproduction.

  We also collected some qualitative data:

  - Missing data: Information gap filled by the extra information provided by the customer from the original information or the data collected by application monitoring system.
  - Data accuracy: Data of the application monitoring system collected compared to the original information provided in bug report (e.g., time point event happened).
  - Data coverage: Data of application monitoring system collected compared to original information in bug report (e.g., the range of data collected).
  - Screenshot quality: Screenshots are common in bug reports. In this study we also compared the quality of screenshot provided by customer in original bug report and screenshot collected by the application monitoring system.

  Note that we did not systematically measure performance implications of the application monitoring as part of RQ1. This is because we could not control the operating environment and

other applications running on the user's computers. However, users did not report any negative performance impacts when using application monitoring as part of the commercial product. A systematic assessment of performance implications is the objective of RQ2 studied in the following section.

#### 7.1.2. Case study 2: Performance impact of application monitoring

- **Objective:** The objective of this study is to answer RQ2: How does application monitoring impact the performance of the software?
- **Case and units of analysis:** The case for analysis is the test bed system we developed and the unit of analysis is the performance of the monitored software system. We deployed the test bed and the application monitoring system on two different PCs with different hardware setups. We evaluated the performance of our approach using the Chrome web browser (v98 stable) with "Use hardware acceleration when available" option turned on with two PCs. The first PC (PC1) was a high performance machine with a Ryzen 9 3900x CPU (12 Cores/24 Threads), 64G RAM @ 3200 Mhz, Samsung 980 Pro 1TB SSD with PCIE 4 and a GTX 1070 GPU. The second PC (PC2) was a less powerful machine, a Surface Book laptop with a i5-6300U CPU, 8 GB RAM, 128 GB SSD and integrated GPU. Also, we ran the monitored application with and without our application monitoring.
- **Data collection and analysis:** We plugged our application monitoring prototype into our test bed and then performed three types of actions using the GUI provided by test bed to search, view details, and, edit and save literature data (see Section 4 for details about the functionality of the test bed that implements a literature management system). We performed each type of action five times with different inputs (e.g., different keywords to query the data). We performed the same process with our test bed with and without application monitoring both PCs with the Chrome developer tool[17] enabled. For each of the four modes (PC1 or PC2 and monitoring activated or not activated) we collected the data shown in Table 4: (1) The execution time for each of the three tasks (query, view, edit and save literature data in the test bed application) with and without application monitoring; (2) how long it took the UI to render; (3) RAM usage of the monitored application; and (4) overhead data usage of the monitored application.

  Performance data were analyzed between different setups. We analyzed the data from each category across the different setups using descriptive statistics. For example, we compared the UI rendering time between setups with and without our plugin installed and with different hardware setups. The average overhead data usage is determined as the size of the collected data in IndexedDB divided by the number of times an action was performed.

### 7.2. Results and discussion

#### 7.2.1. RQ1: Support for bug reproduction

We categorized our findings into quantitative and qualitative.

**Quantitative findings:** Table 2 shows a summary of quantitative data that compares the process of fixing the five bugs with and without application monitoring. Detail of main findings are discussed below:

- **Time to reproduce bug:** With the additional information collected via application monitoring, all five bugs we selected could be reproduced by a developer within 60 min, compared to the original time required to reproduce the bugs (shown in Table 2), the required time was reduced from days to minutes. While this shows a significant reduction in time, we also acknowledge that the original time to reproduce the bugs may have been impacted by the urgency of the bug (e.g., bugs with higher priority may get fixed quicker). However, even under the best possible conditions and highest urgency, the time needed to reproduce a bug without application monitoring data will still surpass the time required to do the bug reproduction with the application monitoring data. As the communication loop between different stakeholders and end users is the most time-consuming part and application monitoring data is filling that gap.
- **Number of involved developers:** Compared to the original efforts to reproduce the bugs, the bug reproduction process with application monitoring required fewer people. The five bug reports we studied required at least three developers to work on them, while only one developer was required when utilizing the data from application monitoring. The developer working on the bug with the application monitoring also had less experience compared to the original bug fixers.
- **Interactions between developers and users:** For the five bugs we studied, extensive communication with customers was necessary to obtain all the information required to fully understand the bug and the situation in which it occurred. However, with the information provided by application monitoring all bugs could be reproduced without additional help from customers.

**Qualitative findings:** In addition to the quantitative comparison, we also have some qualitative discoveries:

- **Missing information:** We evaluated the difference between the initial and final versions of the bug reports we looked at and identified the information gap in the report that needs to be filled in order to reproduce the bug. Table 3 shows the details of the missing information that we identified when reproducing the bugs captured in the five bug reports we worked with. As we can see, user interaction data plays always an important role in bug reproduction. For different tasks, different kinds of information are also required. For example, bug ID 8704 required event data and ID 8786 required data exchange data. Compared to the previous studies and tools discussed in Section 2, our technique provided the missing data not available with other approaches and also specifically for SPA.
- **Data accuracy:** The data collected via the application monitoring is more accurate compared to the data provided by the customer. For example, in one of the bug reports the customer told the support team that when they try to consolidate stocks, sometimes (but not always) an error would occur. Also, the user could not point out the actual stock item they were trying to consolidate. Application monitoring on the other hand provided the exact stock item data to the developers to reproduce the bug. Similarly, customers may point out low performance, but application monitoring provides actual response times.
- **Data coverage:** Application monitoring also includes more data than the data reported by the customer in the bug report or when talking to customers. For example, JavaScript logs are not always available from customers and therefore not "collectable" by the customer (or would require advanced computing skills to collect it). Additionally, our technique combines relevant data types for SPA in a single place to provide a broader range of types of data compared to each of the previous tools discussed in Section 2.
- **Screenshot quality:** Compared to screenshots provided by the customer, the quality of the screenshots taken by the application monitoring is higher and more complete (the screenshot always cover the whole view of the application rather than just a limited area of the application). For example, information about active screen references are always included, which is very helpful for the team to locate the screen that triggered the problem in order to start the reproduction process.

[17] https://developer.chrome.com/docs/devtools/

**Table 2**
Comparison of bug fixing efforts - quantitative measures.

| Bug report | 8262 | 8625 | 8704 | 8870 | 8786 |
|---|---|---|---|---|---|
| Time to reproduce bug - original | 148 days | 55 days | 30 days | 7 days | 7 days |
| Time to reproduce bug - AM | 34 min | 41 min | 33 min | 26 min | 20 min |
| Involved developer count - original | 5 | 4 | 3 | 3 | 5 |
| Involved developer count - AM | 1 | 1 | 1 | 1 | 1 |
| Developers and users interactions - original | 17+ | 8+ | 9+ | 5+ | 12+ |

**Table 3**
Missing information from original bug report.

| Bug report ID | Missing information |
|---|---|
| 8262 | User interaction data:<br>- Page visited when bug occurred<br>- Data input by user<br>- Button clicked by user<br>- Order of actions executed |
| 8625 | User interaction data:<br>- Page visited when bug occurred<br>- Button clicked by user<br>Client–server communication data:<br>- Endpoint called by client end<br>- Parameters sent in REST request |
| 8704 | User interaction data:<br>- Page visited when bug occurred<br>- Data input by user<br>- Button clicked by user<br>Execution event data:<br>- Exception data thrown by application |
| 8786 | User interaction data:<br>- Page visited when bug occurred<br>- Data input by user<br>- Button clicked by user<br>- Order of action executed<br>Client–server communication data:<br>- Endpoint called by client end<br>- Parameters sent in REST requests<br>- Result returned in REST requests |
| 8786 | User interaction data:<br>- Page visited when bug occurred<br>- Button clicked by user<br>Client–server communication data:<br>- Endpoint called by client end<br>- Parameters sent in REST requests |

**RQ1 Outcome: The result of our study shows that the application monitoring system can improve the efficiency of the bug reproduction process for SPAs and the application monitoring system also improves the quality of the data provided to the bug fixers.**

#### 7.2.2. RQ2: Performance impact of application monitoring

Table 4 shows a summary of the performance impact of the application monitoring for different tasks and different setups.

For PC1 (high performance machine) the performance impact on UI rendering is around 2% and RAM usage is up about 5%. The additional data usage is about 5MB/min in total to run the application monitoring based on the tested tasks. For PC2 the performance impact is around 8% for both, UI rendering time and the RAM usage.

We did not find any noticeable interruption of user interactions on the high performance machine (PC1) as the performance overhead introduced by application monitoring is less than 100 ms most of the time. However, on PC2, the negative performance impact is sometime noticeable as the overhead is more than 100 ms. We also noticed some thread locks and screen freezing for more than 300 ms based on the profiling data from the Chrome developer tool. The thread lock happened when the application monitoring system tried to take multiple screenshots in a very short period of time (more than three screen shots in a second). This may have been caused by the intensive hardware usage when screenshots are taken.

**RQ2 Outcome: The results of our study show that the application monitoring introduces an overhead to the SPA and the impact is correlated to the specification of the hardware running the application. The better the hardware specification the lesser the impact of application monitoring.**

### 7.3. Challenges when implementing application monitoring

In addition to answering the two research questions regarding the evaluation of our approach, we also observed some challenges when implementing it. These insights could help other researchers as well as future work.

- **Identifying correct events for data collection:** The first challenge is to find the correct trigger to start collecting the data. In our prototype, we collected user interaction data when a click or page load event happens, while client–server communication data is collected in different stages of a REST API request. Finding the correct event to collect data is challenging. If the wrong event is picked, we may miss some important data or collect duplicated data (and thus increase overhead). For example, when collecting data during different stages of REST requests, some data (e.g. request body) may be duplicated in many stages so we have to find the appropriate storage to collect such data instead of collecting it multiple times.
- **Amount of data to collect:** The second challenge is to decide how many data we need to keep for analysis. Too much data will increase the overhead, and too little data may not be detailed enough for bug reproduction. For example, in our prototype, we only keep response data fields with fewer than 5,000 characters so we still get key information but not too much overhead. Most data over 5,000 length are files, which can be retrieved separately if needed, e.g., by querying for their file name. The second example is the quality of the screenshots we collected. To reduce overhead, we reduced the screenshot quality while still keeping it readable.
- **Key data to collect:** Identifying the key data to collect is another challenge when implementing an application monitoring system. We started our implementation based on the information we discovered in Section 2 and then enhanced it based on the information gap we identified throughout our research and testing we did with our test bed. For example, previous studies suggested that error logs are important for bug reproduction (Choudhary and Orso, 2009) and from the bug reports we analyzed in our research we found that the parameter data in REST request is useful as well, and based on our results from the test bed, we found that tracking ID (a unique ID to identity the device on which the browser is running) to keep track of devices is also useful for bug reproduction.

**Table 4**
Performance impact of application monitoring.

| Data item | PC1 | PC1 with AM | PC2 | PC2 With AM |
|---|---|---|---|---|
| Query data | | | | |
| Method execution time (Run 1) | 1,117 ms | 1,130 ms | 2,385 ms | 2,590 ms |
| UI rendering time (Run 1) | 1,785 ms | 1,809 ms | 4,939 ms | 5,356 ms |
| Method execution time (Run 2) | 1,335 ms | 1,356 ms | 2,240 ms | 2,406 ms |
| UI rendering time (Run 2) | 2,116 ms | 2,138 ms | 4,542 ms | 4,916 ms |
| Method execution time (Run 3) | 1,393 ms | 1,431 ms | 2,472 ms | 2,683 ms |
| UI rendering time (Run 3) | 2,223 ms | 2,280 ms | 5,185 ms | 5,579 ms |
| Method execution time (Run 4) | 1,383 ms | 1,409 ms | 2,279 ms | 2,439 ms |
| UI rendering time (Run 4) | 2,132 ms | 2,164 ms | 4,628 ms | 5,033 ms |
| Method execution time (Run 5) | 1,318 ms | 1,352 ms | 2,335 ms | 2,530 ms |
| UI rendering time (Run 5) | 2,042 ms | 2,093 ms | 4,696 ms | 5,086 ms |
| ***Method execution time (average)*** | ***1,309 ms*** | ***1,336 ms*** | ***2,342 ms*** | ***2,530 ms*** |
| ***UI rendering time (average)*** | ***2,060 ms*** | ***2,097 ms*** | ***4,798 ms*** | ***5,194 ms*** |
| View data detail | | | | |
| Method execution time (Run 1) | 873 ms | 886 ms | 1,580 ms | 1,715 ms |
| UI rendering time (Run 1) | 993 ms | 1,009 ms | 1,949 ms | 2,096 ms |
| Method execution time (Run 2) | 853 ms | 878 ms | 1,549 ms | 1,680 ms |
| UI rendering time (Run 2) | 968 ms | 988 ms | 1,906 ms | 2,077 ms |
| Method execution time (Run 3) | 921 ms | 936 ms | 1,764 ms | 1,907 ms |
| UI rendering time (Run 3) | 1,062 ms | 1,093 ms | 2,124 ms | 2,310 ms |
| Method execution time (Run 4) | 1,023 ms | 1,044 ms | 1,460 ms | 1,584 ms |
| UI rendering time (Run 4) | 1,218 ms | 1,234 ms | 1,768 ms | 1,900 ms |
| Method execution time (Run 5) | 823 ms | 835 ms | 1,530 ms | 1,668 ms |
| UI rendering time (Run 5) | 960 ms | 975 ms | 1,872 ms | 2,034 ms |
| ***Method execution time (average)*** | ***899 ms*** | ***916 ms*** | ***1,577 ms*** | ***1,711 ms*** |
| ***UI rendering time (average)*** | ***1,040 ms*** | ***1,060 ms*** | ***1,924 ms*** | ***2,083 ms*** |
| Edit data | | | | |
| Method execution time (Run 1) | 829 ms | 852 ms | 1,609 ms | 1,729 ms |
| UI rendering time (Run 1) | 923 ms | 937 ms | 1,935 ms | 2,086 ms |
| Method execution time (Run 2) | 879 ms | 904 ms | 1,663 ms | 1,806 ms |
| UI rendering time (Run 2) | 990 ms | 1,003 ms | 2,152 ms | 2,311 ms |
| Method execution time (Run 3) | 1,083 ms | 1,112 ms | 1,512 ms | 1,640 ms |
| UI rendering time (Run 3) | 1,295 ms | 1,322 ms | 1,881 ms | 2,023 ms |
| Method execution time (Run 4) | 866 ms | 878 ms | 1,662 ms | 1,782 ms |
| UI rendering time (Run 4) | 999 ms | 1,025 ms | 2,136 ms | 2,289 ms |
| Method execution time (Run 5) | 1,088 ms | 1,116 ms | 1,691 ms | 1,827 ms |
| UI rendering time (Run 5) | 1,223 ms | 1,237 ms | 2,044 ms | 2,195 ms |
| ***Method execution time (average)*** | ***949 ms*** | ***972 ms*** | ***1,627 ms*** | ***1,757 ms*** |
| ***UI rendering time (average)*** | ***1,086 ms*** | ***1,105 ms*** | ***2,030 ms*** | ***2,181 ms*** |
| Peak RAM usage | | | | |
| Ram usage (Run 1) | 46.1 MB | 48.1 MB | 59 MB | 63.8 MB |
| Ram usage (Run 2) | 47.9 MB | 50.1 MB | 56.8 MB | 61.4 MB |
| Ram usage (Run 3) | 53.3 MB | 56.4 MB | 59 MB | 63.6 MB |
| Ram usage (Run 4) | 51.3 MB | 54.3 MB | 57.7 MB | 62.8 MB |
| Ram usage (Run 5) | 47.8 MB | 50.6 MB | 58.5 MB | 62.9 MB |
| ***Ram usage (Average)*** | ***49.28 MB*** | ***51.9 MB*** | ***58.2 MB*** | ***62.9 MB*** |
| Average overhead data usage for AM | | **5.3 MB/s** | | **4.8 MB/s** |

### *7.4. Potential areas for future research*

During our study, we also noticed several areas that are worth to be investigated in future studies for application monitoring for bug reproduction:

- **On-demand monitoring** (rather than continuous monitoring) can reduce the data overhead. One key challenge is to find the most suitable method for on-demand monitoring. We implemented a method that allows users to "opt-in" to monitoring and another method that allows developers to set up some predefined rules to trigger the monitoring (e.g., start monitoring when user go on a specific page). The trade-offs of methods for different applications need to be considered when implementing on-demand application monitoring. For example, opt-in monitoring may result in not collecting required data, while opt-out monitoring may violate privacy compliance requirements in some regions.
- **Data presentation** is one of the biggest challenges when helping developers reproduce bugs using application monitoring. We refined our data presentation application in many iterations. As an outcome, users can query and group data based on different aspects (e.g., timeline, user session ID). Exploring other ways to present monitoring data in a more relevant format to other users in another context is an area that needs more investigation.
- **Data privacy** also needs to be considered when implementing application monitoring. As mentioned before, data collected may violate privacy compliance requirements in some regions or the service level agreement between the company and end users. In our technique we tried to minimize the collection of data that may identify individuals. Furthermore, techniques such as data masking (Wang et al., 2022) can address the privacy issue, however, if the data masked is the key data required for bug reproduction, this will reduce the effectiveness of application monitoring. How to maintain data privacy for the end user while providing sufficient data for bug reproduction needs further work.

## 8. Limitations and threats to validity

In this study we demonstrated how application monitoring can help with bug reproduction. Below we discuss limitations of our proposed approach and threats to the validity of the evaluation.

### 8.1. Limitations of the proposed approach

**Technology changes:** Web technologies are continuously evolving. The reference implementation we developed may not work with future web technologies. On the other hand, the overall approach for providing application monitoring, the types of data we identified, and providing information with context data in a meaningful way is useful for and applicable to technology-specific solutions that can be developed in the future.

**Web application type:** During the development (and also the evaluation) of our technique, we utilized a test bed. Although we developed our test bed as close to a real system as possible, it may not be generalizable to any type of SPA. Websites for different domains may require different and/or additional types of data for diagnostics. For example, a website dealing with GIS information may require GPS information from the device to do provide more comprehensive diagnostics. Although the prototype we developed may require adjustments to work with other applications, the application monitoring approach should be generalizable for other types of applications.

**Execution environment:** During the evaluation we analyzed the performance of our solution to assess the practicality of our approach. In addition to the hardware configuration, other aspects of the execution environment can also impact the effectiveness of our solution. For example, browser settings can impact the effectiveness and some bugs may not appear due to the different settings between the user's and the developer's browsers. For example, a user may report that pop-up windows in Chrome are not showing properly, but developers cannot reproduce it if this is because users blocked pop-up windows in their browser (our reference implementation uses JavaScript which cannot access browser's setting to collect that information as part of the monitoring).

### 8.2. Threat to validity of the case studies

In this section, the validity threats for the case studies we conducted will be discussed based on the guidelines of Host et al. (2012).

**Construct validity:** The challenges we identified for application monitoring are subjective. Other challenges may also be considered. This could be for various reasons, e.g., the test bed may not cover all possible software types. To mitigate the problem we developed our test bed to present common web applications' functions (such as query view, edit and save of the saved data) and data types (such as string, numbers and dates).

**Internal validity:** Internal validity is about confounding factors that may have impacted our evaluation results. For RQ1, in Section 7, we used a commercial project with real developers to show the effectiveness and efficiency of bug reproduction with application monitoring. Involved developers had different backgrounds and expertise and may have had different priorities at different points in time. This may have impacted how much time and effort they could dedicate to fixing a particular bug (see also our discussion on priorities of bugs in Section 7). Therefore, other factors may also impact the effectiveness of bug reproduction and application monitoring may depend on the experience and skills of the developer(s) working on the bug. Follow-up studies are required to mitigate this problem. To assess performance overheads as part of RQ2, in Section 7, we conducted a study using a test bed and two configurations (low performance and high performance PC). This was to reduce potential confounding factors (like other software running on the test machines that may impact performance as it could have been the case when studying performance in an uncontrolled setting such as the one used for RQ1).

**External validity:** External validity is related to (a) the five bug reports used to assess the usefulness of our technique, and (b) the test bed used to assess performance overheads. The bug reports are real bug reports for a real system involving real developers. We observed those bug reports and the process to fix them in a real-world setting (and within a company) rather than a controlled environment. As such, we did not aim for statistical generalization, but for insights about whether nor not our technique is useful in a real-world setting. As a consequence we acknowledge that their number is limited and we cannot generalize our findings to any bug report or any development context (see also our discussion on internal validity on confounding factors). Regarding the test bed, we aimed to develop a representative test bed that has characteristics common to SPA (see our definition of "representative" in Section 4). However, we acknowledge that there are other technology stacks to develop web applications and the technology can evolve in the future. Therefore, the results of our study may not be generalized to some special technology stacks or future technologies. Yet, by being explicit about our test bed specification we clarify to what kinds of system our results may or may not apply.

**Reliability**: Regarding our study of performance impact, software performance can be affected by other external factors such as hardware performance drop due to high-temperature throttling, the amount and resource requirements of other processes running on a machine, background processes, etc. Although we performed the test tasks multiple times to average the performance impact with our two setups when studying performance implications of application monitoring, we cannot guarantee the result will be the same with other setups.

## 9. Conclusions

In this study we investigated how application monitoring can help with bug reproduction in SPA web applications. We developed a technique for application monitoring, including a prototype implementation. Furthermore, we deployed this approach in a test bed to assess the feasibility of our approach.

We also evaluated our approach by replicating bug fixing activities of historical bug reports of a commercial project. Results indicate that the information provided by the application monitoring approach helps reproduce bugs faster and with fewer developers. The evaluation showed that we can reduce the bug reproduction time from days to minutes and decrease the number of developers from multiple to one. Also, the information provided by the application monitoring approach closes typical information gaps in bug reports obtained from customers and users. Lastly, the information quality provided by application monitoring is improved in terms of data accuracy and coverage.

Future work includes deploying the application monitoring in more SPA web applications and investigating how application monitoring can be improved to reduce performance impacts. Furthermore, we will explore how to more effectively present monitoring data to developers in order to improve the usability of the application monitoring system to support bug reproduction.

## CRediT authorship contribution statement

**Di Wang:** Conceptualization, Methodology, Software, Investigation, Visualization, Data curation, Writing – original draft. **Matthias Galster:** Conceptualization, Supervision, Investigation, Writing – review & editing. **Miguel Morales-Trujillo:** Conceptualization, Supervision, Investigation, Writing – review & editing.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

Data will be made available on request.

## References


Al-Shammari, S.W., Husein, A.A., 2020. Response time study of cloud web application - based smart monitoring system. In: 2020 International Conference on Computer Science and Software Engineering. (CSASE), pp. 138–141. http://dx.doi.org/10.1109/CSASE48920.2020.9142099.

Asadollah, S.A., Saadatmand, M., Eldh, S., Sundmark, D., Hansson, H., 2016. A model for systematic monitoring and debugging of starvation bugs in multicore software. In: Proceedings of the 1st International Workshop on Specification, Comprehension, Testing, and Debugging of Concurrent Programs. ACM, http://dx.doi.org/10.1145/2975954.2975958.

Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C., 2004. Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. 1 (1), 11–33. http://dx.doi.org/10.1109/TDSC.2004.2.

Chaparro, O., Lu, J., Zampetti, F., Moreno, L., Di Penta, M., Marcus, A., Bavota, G., Ng, V., 2017. Detecting missing information in bug descriptions. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. In: ESEC/FSE 2017, Association for Computing Machinery, New York, NY, USA, pp. 396–407. http://dx.doi.org/10.1145/3106237.3106285.

Choudhary, S.R., Orso, A., 2009. Automated client-side monitoring for web applications. In: IEEE International Conference on Software Testing, Verification, and Validation Workshops. ICSTW 2009, pp. 303–306. http://dx.doi.org/10.1109/ICSTW.2009.44.

DeMott, J.D., Enbody, R.J., Punch, W.F., 2013. Systematic bug finding and fault localization enhanced with input data tracking. Comput. Secur. 32, 130–157. http://dx.doi.org/10.1016/j.cose.2012.09.015.

Dimitrov, M., Zhou, H., 2011. Time-ordered event traces: A new debugging primitive for concurrency bugs. In: 2011 IEEE International Parallel & Distributed Processing Symposium. IEEE, http://dx.doi.org/10.1109/ipdps.2011.38.

Diniz-Junior, R.N., Figueiredo, C.C.L., De S.Russo, G., Bahiense-Junior, M.R.G., Arbex, M.V., Dos Santos, L.M., Da Rocha, R.F., Bezerra, R.R., Giuntini, F.T., 2022. Evaluating the performance of web rendering technologies based on JavaScript: Angular, react, and vue. In: 2022 XVLIII Latin American Computer Conference. (CLEI), pp. 1–9. http://dx.doi.org/10.1109/CLEI56649.2022.9959901.

Filip, P., Cegan, L., 2019. Comparing tools for web-session recording and replaying. pp. 257–260. http://dx.doi.org/10.1109/SIET48054.2019.8986134.

Garg, V., 2022. Single page applications: when and why you should use. Bluebash Blog https://www.bluebash.co/blog/single-page-applications-when-and-why-you-should-use/.

Gopinath, R., Jensen, C., Groce, A., 2014. Mutations: How close are they to real faults? In: 2014 IEEE 25th International Symposium on Software Reliability Engineering. pp. 189–200. http://dx.doi.org/10.1109/ISSRE.2014.40.

Goyal, A., Sardana, N., 2018. Characterization study of developers in non-reproducible bugs. In: 2018 Eleventh International Conference on Contemporary Computing. (IC3), pp. 1–6. http://dx.doi.org/10.1109/IC3.2018.8530641.

Host, M., Rainer, A., Runeson, P., Regnell, B., 2012. Case Study Research in Software Engineering: Guidelines and Examples. Wiley, URL https://books.google.co.nz/books?id=T7rXoaxqPIAC.

Huang, L., Ai, J., 2015. Automatic software fault localization based on artificial bee colony. J. Syst. Eng. Electron. 26 (6), 1325–1332. http://dx.doi.org/10.1109/jsee.2015.00145.

Joorabchi, M., Mirzaaghaei, M., Mesbah, A., 2014. Works for me! characterizing non-reproducible bug reports. In: 11th Working Conference on Mining Software Repositories. MSR 2014, pp. 62–71. http://dx.doi.org/10.1145/2597073.2597098.

Kiciman, E., Wang, H.J., 2007. Live monitoring: Using adaptive instrumentation and analysis to debug and maintain web applications. In: Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems. HOTOS '07, USENIX Association, USA.

Lazareva, N., Karasevskis, K., Girjatovcs, A., Kuznecova, O., 2022. Business process automation in retail. In: 2022 63rd International Scientific Conference on Information Technology and Management Science of Riga Technical University. (ITMS), pp. 1–5. http://dx.doi.org/10.1109/ITMS56974.2022.9937096.

Magalhães, J.P., Silva, L.M., 2013. Adaptive monitoring of web-based applications: A performance study. In: Proceedings of the ACM Symposium on Applied Computing. pp. 471–478. http://dx.doi.org/10.1145/2480362.2480454.

Mao, C., 2011. Variable precision rough set-based fault diagnosis for web services. In: 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications. IEEE, http://dx.doi.org/10.1109/trustcom.2011.215.

Mao, C., Tervonen, I., Chen, J., 2011. Diagnosing web services system based on execution traces pattern analysis. In: 2011 IEEE 8th International Conference on E-Business Engineering. IEEE, http://dx.doi.org/10.1109/icebe.2011.43.

Mu, P., 2021. Research on the application of ERP financial software in enterprises. In: 2021 2nd International Conference on Computer Science and Management Technology. (ICCSMT), pp. 201–204. http://dx.doi.org/10.1109/ICCSMT54525.2021.00046.

Rahman, M.M., Khomh, F., Castelluccio, M., 2020. Why are some bugs non-reproducible? : –An empirical investigation using data fusion. (c), pp. 605–616. http://dx.doi.org/10.1109/icsme46990.2020.00063.

Ray, P.P., Banerjee, A., 2011. Debugging memory issues in embedded linux: A case study. In: IEEE Technology Students Symposium. IEEE, http://dx.doi.org/10.1109/techsym.2011.5783857.

Rodríguez, C., Baez, M., Daniel, F., Casati, F., Trabucco, J.C., Canali, L., Percannella, G., 2016. REST APIs: A large-scale analysis of compliance with principles and best practices. In: Bozzon, A., Cudre-Maroux, P., Pautasso, C. (Eds.), Web Engineering. Springer International Publishing, Cham, pp. 21–39.

Roehm, T., Gurbanova, N., Bruegge, B., Joubert, C., Maalej, W., 2013. Monitoring user interactions for supporting failure reproduction. In: 2013 21st International Conference on Program Comprehension. (ICPC), pp. 73–82. http://dx.doi.org/10.1109/ICPC.2013.6613835.

Roychowdhury, S., Khurshid, S., 2012. A family of generalized entropies and its application to software fault localization. In: 2012 6th IEEE International Conference Intelligent Systems. IEEE, http://dx.doi.org/10.1109/is.2012.6335163.

Sasaki, T., Morisaki, S., Matsumoto, K., 2011. An exploratory study on the impact of usage of screenshot in software inspection recording activity. In: 2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement. pp. 251–256. http://dx.doi.org/10.1109/IWSM-MENSURA.2011.53.

Sprenkle, S., Gibson, E., Sampath, S., Pollock, L., 2005. Automated replay and failure detection for web applications. In: 20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005. pp. 253–262. http://dx.doi.org/10.1145/1101908.1101947.

Uluca, D., 2018. Angular 6 for Enterprise-Ready Web Applications: Deliver Production-Ready and Cloud-Scale Angular Web Apps. Packt Publishing, Limited, Birmingham.

Wang, Q., Wu, L., Chen, Y., Li, L., 2022. A data masking method based on genetic algorithm. In: 2022 2nd Asia Conference on Information Engineering. (ACIE), pp. 13–17. http://dx.doi.org/10.1109/ACIE55485.2022.00011.

Wong, W.E., Debroy, V., Li, Y., Gao, R., 2012. Software fault localization using dstar. In: 2012 IEEE Sixth International Conference on Software Security and Reliability. IEEE, http://dx.doi.org/10.1109/sere.2012.12.

Yousefi, A., Wassyng, A., 2013. A call graph mining and matching based defect localization technique. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops. IEEE, http://dx.doi.org/10.1109/icstw.2013.17.

Yu, R., Zhao, L., Wang, L., Yin, X., 2011. Statistical fault localization via semi-dynamic program slicing. In: 2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications. IEEE, http://dx.doi.org/10.1109/trustcom.2011.89.

Zhang, Z., Chan, W., Tse, T., 2012. Fault localization based only on failed runs. Computer 45 (6), 64–71. http://dx.doi.org/10.1109/mc.2012.185.

Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schroter, A., Weiss, C., 2010. What makes a good bug report? IEEE Trans. Softw. Eng. 36 (5), 618–643. http://dx.doi.org/10.1109/TSE.2010.63.



**Di Wang** is a Ph.D. student in the Department of Computer Science and Software Engineering at the University of Canterbury, New Zealand.

**Matthias Galster** is a Professor in the Department of Computer Science and Software Engineering at the University of Canterbury, New Zealand.

**Miguel Morales-Trujillo** is a Senior Lecturer in the Department of Computer Science and Software Engineering at the University of Canterbury, New Zealand. His main research area is Software Quality, Software Processes and Gamification in Software Engineering education.