Empirical Software Engineering (2023) 28:117
https://doi.org/10.1007/s10664-023-10352-5

What constitutes debugging? An exploratory study of debugging episodes

Abdulaziz Alaboudi[1] · **Thomas D. LaToza**[1]

Accepted: 30 May 2023 / Published online: 11 September 2023

Abstract
When debugging, developers engage in activities such as navigating, editing, testing, and inspecting code. Despite being the building blocks of debugging, little is known about how they constitute debugging. To address this gap, we introduce the concept of a "debugging episode" which encompasses the time from when a developer first investigates a defect until when the defect is fixed or the developer stops. We observed 11 professional developers working on open source projects, coding 89 debugging episodes and 2135 instances of activities that occurred during them. Six activities were identified: navigate, edit, test, inspect, consult resources, and other miscellaneous activities. We found that developers spent the most time editing (41%) and testing (29%) during debugging. When addressing time-consuming defects, developers engaged in more diverse types of debugging activities, spent more time inspecting program state, navigating code, and consulting external resources, and spent less time testing. We found that the activities developers do while debugging were more similar than different than the activities that make up implementation work. Developers spent a similar fraction of their time editing and navigating during debugging and implementation work. However, debugging involved significantly more time inspecting (16%) than implementation work (2%), while implementation work involved more time consulting resources (24%) than debugging (6%). We conducted semi-structured interviews with ten developers to gain insights into the challenges that cause developers in longer debugging episodes to engage with more activities. Our findings offer insight into the debugging process and the challenges that developers confront, offering implications for the design of debugging tools, improved debugging education, and future research.

Keywords Debugging · Debugging activities · Debugging tools

Communicated by: Scott Fleming

✉ Abdulaziz Alaboudi
aalaboud@gmu.edu

Thomas D. LaToza
tlatoza@gmu.edu

[1] Department of Computer Science, George Mason University, Fairfax, VA, USA

1 Introduction

Debugging is an essential part of a developer's daily work, involving the localization, understanding, and fixing of defects in programs (Beller et al. 2018; Britton et al. 2013). To debug, developers undertake several *activities*, including navigating and editing code, inspecting state, and testing program output. The success and efficacy of the debugging process depend on the performance of these activities. An extended duration of these activities results in a lengthier debugging process.

However, although these activities represent the building blocks of debugging, little is known about how they constitute debugging. To our knowledge, there have been no studies that specifically quantify and examine the activities developers engage in while debugging, how these activities change as debugging grows longer and more difficult, and to what extent debugging is similar for defects that were inserted or triggered while working and for defects that were reported in issue trackers. A deeper analysis of debugging activities will ultimately lead to a better understanding of the debugging process and the tools that support it.

To fill this gap, we first introduce the notion of a *debugging episode*, spanning the time from the point at which developers begin investigating a defect until they either fix it or decide to stop investigating. By explicitly tracking the beginning and end of debugging episodes, we were able to analyze the typical characteristics of what occurs during episodes, including the activities that occur in each episode. We broadly characterized all of the debugging episodes we observed, both when developers in the midst of work that introduced or triggered a defect (fresh defects) and when developers worked to fix a pre-existing defect already identified in an issue tracker (committed defects).

This paper reports a quantitative study of *debugging episodes* in naturalistic settings. We observed developers debugging within software projects of varying sizes, programming languages, practices, and domains. To curate such a diverse dataset, we used a recently available new source of data: live-streamed programming videos (Alaboudi and LaToza 2019a). In these videos, developers work while narrating how and why they are working as they do. Researchers have found that these videos show professional developers working in real-time on open source projects (Alaboudi and LaToza 2019a, b), podcasting hours of development, and debugging work on projects used in production. Using this data source, we curated 15 videos in which 11 professional developers worked for 30 h. We systematically coded instances of debugging episodes, yielding 89 debugging episodes that lasted for 15 h. We then coded activities that occurred during debugging and outside debugging (i.e., implementation work) - navigating, editing, testing, inspecting, consulting resources, and other miscellaneous activities, yielding a dataset of 2135 activities in debugging and 1477 activities that occurred in implementation work.

We aim to answer the following research questions related to debugging episodes and their activities:

RQ1 How much time do developers spend on debugging episodes, and how does debugging change as episodes become longer?
RQ2 Which activities consume the most time during debugging episodes?
RQ3 How do developers switch between different activities during debugging episodes?
RQ4 How do debugging episodes vary for fresh and committed defects?
RQ5 How do activities performed during debugging episodes differ from those in implementation work?

Our study revealed that the duration of debugging episodes varied considerably, and this variation also extended to the activities involved. Most debugging time was spent in the longest

25% of episodes, while the rest were frequent but short in duration. Debugging episodes with fresh defects were much shorter than those with committed defects, taking a median of three minutes to debug compared to 29 min for committed defects. Longer debugging episodes encompassed various activities, such as editing and navigating code, testing the program, inspecting the program, consulting resources, and miscellaneous activities. These episodes involved ten times more switching between activities, more inspecting, and less testing. In contrast, short debugging episodes consisted mostly of editing and testing.

Our observations revealed that no single activity dominated debugging episode time. However, editing and testing together consumed over two-thirds of debugging time, with 41% of debugging time spent on editing and 29% on testing. Developers switched between activities frequently, spending less than a minute on each activity and gathering information incrementally as needed. The frequency of switching activities varied across debugging episodes, with navigating being most common early in debugging and later activity involving more editing, testing, and consulting resources. Developers made extensive use of the source code as an anchor between activities. During debugging episodes, editing and navigating code activities displayed remarkably similar characteristics to those performed during implementation work. Two activities that differed during debugging were inspecting the program and consulting resources.

One interesting insight we found was that longer debugging episodes involve more frequent switching between different activities, which could indicate that developers were struggling while debugging. These struggles could inform the development of future debugging tools that support the various activities involved in debugging. To better understand these struggles, we conducted semi-structured interviews with ten professional developers about their recent debugging experiences. Based on their responses, we identified specific challenges and connected them to our observations. Our second study aimed to answer the following research question:

> **RQ6** What are the challenges that developers experience during debugging that require frequent switches between different activities?

Developers faced challenges in comprehending and addressing defects in their recent debugging episodes, necessitating their involvement in different activities. To comprehend the behavior of defects, developers had to collect an extensive and scattered list of information from multiple sources, such as program state, documentation, and commit history, which resulted in them having to switch between various activities to collect it. While fixing defects, developers had concerns regarding altering source code that relied on or influenced third-party code, which could introduce breaking changes affecting external dependencies. Moreover, working with unfamiliar source code, such as new APIs, necessitated developers to test, edit, and consult resources more frequently.

Our findings have several implications for debugging tools, software engineering education, and researchers. For tools, our findings suggest that one reason developers engage in varying activities is due to challenges in finding, connecting, and analyzing information from different sources. To help developers in this process, we propose a set of design recommendations for future debugging tools that offer help beyond fault localization. For education, we suggest that teaching students about debugging should also involve teaching them about debugging activities and their importance during debugging. Finally, we contribute a new platform for analyzing data from observational studies of developers called "Observe-Dev.online" and discuss how this platform can be used for future research investigating debugging and other software development work.

2 Related Work

Our research builds on a number of prior studies investigating debugging as well as previously proposed tools to help developers debug more effectively.

2.1 Studies of Debugging

Studies have long examined the practice of debugging from a wide range of perspectives, beginning at least as early as 1974 (Gould and Drongowski 1974). Studies have examined the strategies developers use (Vessey 1985; Katz and Anderson 1987; Gugerty and Olson 1986), the use of the debugger (Afzal and Goues 2018; Murphy et al. 2006; Damevski et al. 2017; Amann et al. 2016; Petrillo et al. 2019; Beller et al. 2018), and the information needs in debugging (Ko et al. 2007; LaToza and Myers 2010a; Sillito et al. 2008). These encompass field studies examining behavior and experiments investigating debugging in a controlled setting. Studies have directly observed developers, while others have indirectly observed debugging through log data or self-reports made by developers. However, only six studies have directly observed developers outside the lab (Perscheid et al. 2017; Chattopadhyay et al. 2019; Ko et al. 2007; LaToza and Myers 2010a; Sillito et al. 2008; Vans et al. 1999). Table 1 offers a summary of the main observations of prior studies of debugging behavior.

Developers use a variety of strategies to debug, such as forwards and backwards reasoning. In forwards reasoning, developers follow the execution of the failing test (Böhme et al. 2017), building a mental representation of the program (Katz and Anderson 1987) and inspecting program execution and state through breakpoints (Romero et al. 2007). In backwards reasoning (Gould 1975; Böhme et al. 2017; Lukey 1980), developers start from the incorrect output and work backwards in the execution to the defect location. Information foraging theory (IFT) models how developers navigate code (Lawrance et al. 2013; Piorkowski et al. 2015, 2013), including in debugging tasks. According to IFT, developers navigate between patches (e.g., methods) based on their scent (e.g., method identifiers), which offer hints directing developers to their prey (e.g., the fault location). Developers more frequently switch between subgoals when debugging than in programming (Chattopadhyay et al. 2019). Experienced developers are able to make use of their knowledge to comprehend code at a higher level of abstraction than novices (Vessey 1985; Vans et al. 1999).

Other work has examined debugging through the analysis of the logs generated through instrumented tools installed in developers' machines. The debugger is among the most used features in modern IDEs (Murphy et al. 2006; Amann et al. 2016), which developers start using early in debugging (Afzal and Goues 2018). However, developers generally avoid complex debugger features such as breakpoints (Damevski et al. 2017) and prefer more straightforward techniques such as "printf debugging" (Beller et al. 2018). One challenge in the use of log data to study debugging is the lack of context in which log events occurred. Researchers may not be able to accurately differentiate logged events associated with debugging work from log events associated with programming work. Thus, log data has primarily been used to examine features use of debuggers.

Studies have also been conducted to measure the time developers spend debugging, yielding widely varying results. Beller et al. (2018) observed that developers spent only 14% of their active IDE time in the debugger. Minelli et al. (2016) and Meyer et al. (2014) also reported a low percentage (0.87%, and 3.9% respectively) of total time using the debugger. However, developers reported that they spent between 20%-60% of the working time debugging, which researchers have argued to be an over-estimation (Beller et al. 2018).

Table 1 Prior studies of debugging behavior

Authors	Study type	Data	Participants	Organizations	Duration (hours)	Main observations
Afzal and Goues (2018)	Field Study	Logs	81	Unknown	15K	Debugger usage
Murphy et al. (2006)	Field Study	Logs	41	Unknown	Unknown	Debugger usage
Damevski et al. (2017)	Field Study	Logs	196	1	33K	Debugger usage
Amann et al. (2016)	Field Study	Logs	84	1	6.3K	Debugger usage
Petrillo et al. (2019)	Experiment	Logs,Observation	28	Mostly students	10	Debugger usage
Beller et al. (2018)	Field Study	Logs,Self-reports	634	Unknown	18K	Debugger usage
Perscheid et al. (2017)	Field Study	Observation	8	4	Unkown	Strategies and debugger usage
Chattopadhyay et al. (2019)	Field Study	Observation	3	1	2	Strategies
Ko et al. (2007)	Field Study	Observation	17	1	25	Information seeking
LaToza and Myers (2010a)	Field Study	Observation	17	1	25	Information seeking
Sillito et al. (2008)	Field Study	Observation	16	1	8	Information seeking
Vans et al. (1999)	Field Study	Observation	4	1	8	Strategies
Lawrance et al. (2013)	Experiment	Observation	10	1	20	Strategies
Jiang et al. (2017)	Experiment	Observation	9	Mostly students	18	Strategies
Baltes et al. (2015)	Experiment	Observation	12	Mostly students	12	Strategies
Gould (1975)	Experiment	Observation	10	1	Unknown	Strategies
Vessey (1985)	Experiment	Observation	16	1	6	Strategies
Katz and Anderson (1987)	Experiment	Observation	77	Mostly students	Unknown	Strategies
Gugerty and Olson (1986)	Experiment	Observation	44	Mostly students	Unknown	Strategies
Layman et al. (2013)	Field Study	Self-reports	15	1	Unknown	Strategies and debugger usage
Böhme et al. (2017)	Experiment	Self-reports	12	Unknown	22	Strategies

One focus has been to enumerate general *challenges* developers face that can make debugging difficult. An analysis of debugging "war stories"found the two most common causes of difficulty were inapplicable debugging tools and "large temporal or spatial chasms between the root cause and the symptom" (Eisenstadt 1993). Developers face difficulties reproducing defects and determining the root cause of failures (Ko et al. 2007). Modern systems' multi-threaded and distributed nature can make instrumentation and testing debugging hypotheses challenging (Layman et al. 2013). API's degree of abstraction imposes unique challenges for debugging (Coker et al. 2019). Other studies have identified specific questions developers report to be hard to answer or associated with particularly time-consuming debugging (LaToza and Myers 2010a, b).

Our work builds on these findings, offering a study of debugging episodes that quantify and investigate developers' activities while debugging. Our work provides insight into how different activities constitute debugging, how activities change as debugging episodes get longer, and how developers switch between activities. These observations, combined with findings from interviews with developers, provide insights into the debugging process and the challenges that developers confront, paving the way for the creation of improved debugging tools, enhanced debugging education programs, and future research directions.

2.2 Debugging Tools

A primary focus of work on debugging tools has been on fault localization. Fault localization tools model debugging tasks as code navigation work to find a defect location, aiming to shrink the search space developers need to investigate to identify and understand a defect (Weiser 1984; DeMillo et al. 1996; Zhang et al. 2006, 2003; Jones et al. 2002). Wong et al. (2016) identified 331 papers published from 1977 to 2014 which contributed to new fault localization techniques. For example, program slicing tools (Weiser 1984) display to the user a ranked list of potentially faulty statements (DeMillo et al. 1996; Zhang et al. 2006, 2003). Developers' focus should be mostly on these potentially faulty statements. However, these tools need to be highly precise as developers tend to abandon their use when confronted with large numbers of false positives (Parnin and Orso 2011; Xia et al. 2016). In response, new techniques have been proposed with lower rates of false positives, further reducing the search space of potential fault locations (de Souza et al. 2016).

Despite the large amount of work on improving fault localization tools, researchers have found that they offer little help in practice. One reason is that automatic fault localization tools are typically evaluated in their performance of reducing the search space of fault locations a developer must consider rather than in their ability to improve a developer's debugging performance. Implicit in this work is the assumption that any tool that reduces the set of faulty statements a developer must consider will necessarily improve debugging performance. Parnin and Orso (2011) tested this assumption directly through a user study and found that fault localization techniques do not always help developers debug more effectively. When the program is complex, the assumption that developers should be able to understand the faults by looking at fewer locations may not be correct, even for more experienced developers. Other studies have also found that a list of fault locations failed to help developers debug more quickly (Wang et al. 2015; Alaboudi and LaToza 2020), again suggesting the need to provide additional context beyond fault locations. Our work has the potential to shed insight into future debugging tools that support developers' work beyond navigating code to localize defects.

3 Methods

To investigate debugging episodes and their activities, we conducted two studies. We first conducted an observational study of 11 professional developers at work on open source projects. We utilized a new source of data, live-streamed programming videos. We collected videos reflecting a diverse cross-section of software projects in varying sizes, programming languages, and domains. We then developed a coding scheme to define debugging episodes and the activities developers perform during debugging episodes. We then conducted a second study using semi-structured interviews with ten professional developers. We interviewed developers and coded their narrations of recent debugging episodes.

3.1 Study 1: Developer Observations

3.1.1 Live-Streamed Programming Videos

Using platforms such as YouTube and Twitch, developers have recently begun the practice of live-streaming their work, broadcasting and recording their real-time work contributing to open source software projects (Faas et al. 2018; Alaboudi and LaToza 2019b). Researchers have found that these videos are not rehearsed and illustrate developers' moment-to-moment work contributing to real software projects using their preferred development environment (Alaboudi and LaToza 2019a) (Fig. 1). Moreover, as they explain their work to watchers, the live-streamed videos both document developers' actions and offer a running commentary, similar to think-aloud, describing why they are working as they do.

Lives-streamed programming videos share a common structure (Alaboudi and LaToza 2019a). Developers start the live-stream by stating a goal for the stream. They then work towards the goal, reading documentation and writing, debugging, and running code. Most videos depict work on open source projects, which developers link to in the stream descrip-

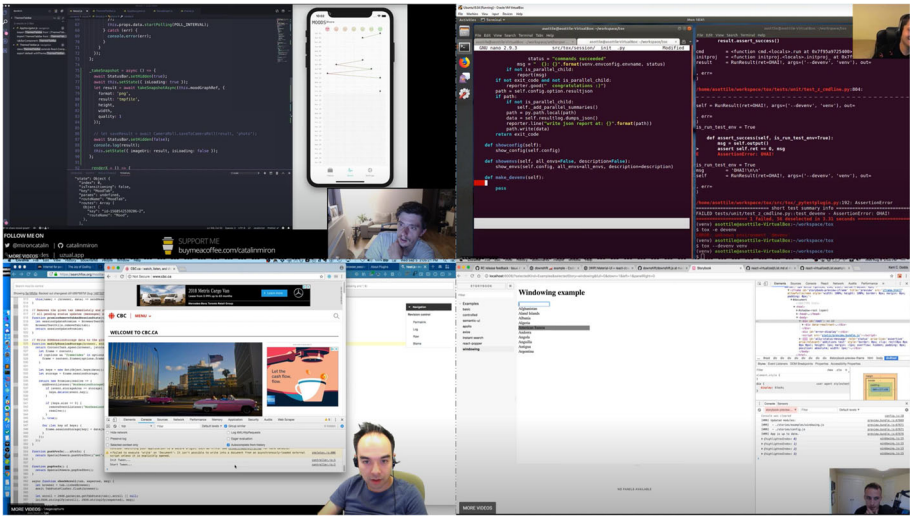


Fig. 1 Developers live-stream their programming work on open source projects, which they record and share on platforms such as YouTube

tions. Developers may be interrupted, either by developers watching live or by others in their physical space, mirroring the typical interruptions developers face in their day-to-day work (Meyer et al. 2014; Abad et al. 2018). After completing the stream, developers often archive the video on platforms such as YouTube and Twitch, with most licensed under Creative Commons.

To select videos, we formulated a strict data collection methodology. To find relevant live-streamed videos, we used YouTube and Twitch searches with keywords such as "open source contribution", "live-stream programming", and "watch me code". In selecting videos to include, we used three criteria:

- *The archived video is available*: Many developers use Twitch to host their stream. However, Twitch archives videos for 14 days. As we needed the ability to conduct analyses over an extended period, we excluded videos that were not archived on a long-term basis.
- *The project is open source and ready to be used by other projects or users*: We first checked that the developer's project was open source. After locating the repository, we skimmed the documentation and issues, looking for evidence that it had or will have a public release for general usage. For example, we looked for dates describing a future release or a link to an executable version of the project.
- *The video shows significant work*: To ensure the video contained a meaningful length of work, we briefly skimmed each video. We excluded videos where developers primarily spent most of the time communicating with other developers through chat. When a video was shorter than two hours, we chose another video from the same developer working on the same project.

We sought to create a diverse sample of live-streamed videos, encompassing developers using a variety of programming languages and working in a variety of application domains. Recent field studies of debugging have observed between eight and ten developers employed by one to four different companies (Perscheid et al. 2017; Chattopadhyay et al. 2019). Based on this, we chose to observe eleven developers working on eleven distinct open source projects. The total duration of these videos is 30 h. Our dataset includes desktop applications, command-line programs, mobile apps, web apps, games, and operating systems. Table 2 lists the 15 videos. As a conservative estimate of developers' experience, we examined each developer's GitHub profile page and identified their first commit to an open source project. All eleven developers actively worked on open source projects for a period of 7 to 31 years (median = 9 years)(GitHub allows developers to migrate their open source contributions from other legacy version control systems, resulting in commits that predate GitHub). Some shared their current or past employer in their GitHub profiles, including Google, Microsoft, Lyft, PayPal, and Mozilla.

3.2 Data Analysis

To identify debugging episodes and activities, we first collected definitions of debugging (Johnson 1982; Ko et al. 2011; Parnin and Orso 2011) to guide our identification. The first author watched the first seven videos and iteratively built a codebook defining debugging episodes and six different activities developers engaged in while working. We summarize our codebook in brief. The complete codebook is available in the replication package Replication package (2022).

- *Debugging episodes* begin when a developer reproduces a defect reported in the issue tracker (*committed defect*) or encounters a defect that occurred or was triggered while

Table 2 Our dataset of 15 live-streamed video from 11 developers across 11 software projects

Developer		Video Length	Project			
ID	Yrs. Exp.*		Name and Brief Description	LOC	Files	Commits
D1	10	2:00:56	*Firefox*: A popular web browser	4 M JavaScript	6K	73 M
D2	31	2:40:48	*Curl*: A library for transferring data	138K C	704	26K
D3	7	1:22:14 + 1:23:56	*Serenity OS*: A Unix-like operating system	120K C++	1.3K	15K
D4	8	3:28:35	*Tox*: A library for task automation	10.7K Python	88	2K
D5	8	2:59:49	*Downshift*: A set of web components built with React	11K JavaScript	82	652
D6	8	1:05:10 + 1:18:38	*Uzual*: A mobile app that helps track daily habits	2.5K JavaScript	45	204
D7	10	2:54:43	*Vectrexy*: A game emulator	9.7K C++	90	656
D8	9	1:21:06 + 1:44:19	*Kap*: A screen recorder for computers	9.5K JavaScript	12	866
D9	9	2:15:26	*DevBette*: A web application for a small business	3.6K C#	84	287
D10	10	1:03:34 + 1:13:40	*Alacritty*: A cross-platform terminal	16.5K Rust	63	1.7K
D11	8	3:40:00	*Webpack*: A bundler for javascript	95K JavaScript	3.1K	12.6K

* # of years contributing to open source projects

working (*fresh defect*). For fresh defects, the episode begins when the developer modifies the program, creating a defect or triggering an existing defect, which then generates incorrect output such as error messages, unit test failures, or unexpected behavior (e.g., a program outputting an empty array instead of an array of numbers). Debugging episodes with committed defects initiate begin when developers replicate the defect reported in the issue tracker. It is important to note that both fresh and committed defects may or may not involve the same developer who introduced the defect working to resolve the defect. Debugging episodes end when the incorrect output that triggered the episode is resolved and the program produces the expected output, or when the developer explicitly states that they have stopped debugging.
When developers are not debugging, their activities fall under the category of *implementation work*, which includes feature additions or enhancements, refactoring, and other maintenance tasks. We have coded the activities that occurred during other programming work to compare them to the activities that took place during debugging. However, we did not code episodes for other programming work as it encompasses a wide variety of tasks. We excluded irrelevant work, such as breaks or socializing, and marked them as interruptions.

- *Activities* are behavior that developers perform to achieve a goal while working. We coded six types of activities - Editing code, Navigating code, Testing the program, Inspecting the program, Consulting resources, and Miscellaneous. The Miscellaneous activity encompasses non-code work, such as interacting with the development environment and writing notes. Tables 3 and 4 summarize the activities and activity characteristics we coded.

It is important to note that our definition of "edit code" accounts for developers who may have diverse editing styles. For example, some developers might prefer to plan their edits before typing, while others may pause while typing and then revise. To accommodate these distinct approaches to editing, we have adopted a broader definition of editing that includes the entire duration starting from the moment the developers open the file that they intend to edit. By adopting this approach, we avoided the potential bias that would have resulted from defining editing as starting from the first keystroke typed.

Table 3 Summary of the coding book for activity start and end criteria

Activities	Start Criteria	End Criteria
Edit Code (Edit)	Open a file of code	Leave the file after editing
Navigate Code (Navigate)	Open a file of code	Leave the file without edits
Test The Program (Test)	Run the program to observe and interact with the final output on the console or on UI	Leave the program output by moving the cursor outside the console or the UI in which the output was generated
Inspect The Program (Inspect)	Run the program to observe and interact with the runtime state through the debugger or instrumental logs	Leave the debugger or the console in which the logs were printed
Consult Resources (Consult)	Browse online documents, forms, issue tracker, and any type of resources outside the development environment	Leave the resources
Miscellaneous (Misc)	Engage in non-coding work (e.g., writing notes, setting up the IDE)	Leave the non-coding activity

Table 4 Summary of the coding book for activity start and end criteria

Activities	Characteristics
Edit	Duration, file name, file type
Navigate	Duration, file name, file type
Test	Duration, testing method
Inspect	Duration, inspection method
Consult	Duration, resource type
Misc	Duration, non-coding activity type

After constructing the initial codebook, the two authors iteratively coded episodes, discussing disagreements after each iteration and revising the codebook. Instead of coding a single episode, we choose several representative episodes with differing codebases, programming languages, and development tools. Through these iterations, the two authors coded 20 distinct episodes and 166 activities. The last iteration yielded a Cohen's Kappa inter-rater agreement (Landis and Koch 1977) of 75% for episodes and 84% for activities, reflecting substantial and almost perfect agreement, respectively. Using the final codebook, the first author then coded the entire dataset of 30 h of development work using observe.dev-online (Sect. 6.2). The entire dataset, including the videos and codes, is publicly available on the Observe.dev platform (Study dataset: Observe.dev 2022).

3.3 Study 2: Developer Interviews

To gain a deeper understanding of the challenges in debugging that might have caused some of the behavior we observed in our first study, we considered several options. One was to analyze the developers' think-aloud while they were live streaming their programming. However, this approach was not always effective as the developers were not always clear in their verbalizations of their goals or struggles. We attempted to reach out to the developers through email and social media by leaving comments on their videos, but we received limited responses. This led us to conduct focused interviews about debugging challenges with a new set of developers, expanding the set of developers informing our findings. We conducted semi-structured interviews with professional developers about recent debugging episodes and cross-referenced examples from our first study.

3.3.1 Recruitment

After obtaining IRB approval, we posted multiple recruitment announcements on Twitter, developers' Slack channels, and email lists for a graduate course at our university. Fifteen developers expressed interest in participating. Two were selected as pilot participants and three were excluded as they had less than one year of professional experience or did not have recent debugging sessions. This yielded 10 interview participants. Participants ranged in professional experience between 2 and 25 years and included eight males and two females (Table 5). To avoid confusion with developers from Study 1, we identify these developers as interviewees (I1-I10).

Table 5 Demographics of interview participants

ID	Yrs.Exp	Software Domain
I1	7	Full stack (Java, Node.js, AWS, Solr)
I2	4	Full stack (JavaScript, C#)
I3	10	Backend (Python, Scikit-learn, Pandas)
I4	3	Backend (Java, SQL)
I5	25	Program manager (Java, Kafka)
I6	10	Full stack (Typescript, React.js, Node.js)
I7	2	Full stack (Wordpress, PHP)
I8	7	Full stack (React.js, Node.js)
I9	3	Full stack (JavaScript, PHP)
I10	10	Full stack (Angular.js, SQL)

3.4 Protocol

Each semi-structured interview was conducted virtually through Zoom. After completing virtual consent, interviewees were asked to describe recent debugging episodes "Can you describe a debugging episode that you had recently?". Developers often reported episodes that occurred within the past few days or weeks. We encouraged interviewees only to discuss debugging episodes they could recall in detail. These details included the sequence of activities they took, the questions they had, and the struggles they faced. The interviewer asked clarification questions as needed such as *"Why did you need to read documentations?"*, *"How complex was the fix? and why?"*, and *"How difficult was it to understand the defect? and why? "*. Interviews were recorded and transcribed using Zoom recording and transcription. Each interview lasted 32-45 min, and interviewees talked about one to three recent debugging episodes. The interview questions are available in the replication package (Replication package 2022).

3.5 Data Analysis

We used a thematic analysis to extract themes related to the challenges developers faced that were related to our observations from the first study. We mainly used the audio transcriptions but also consulted the originally recorded videos when needed. Both authors discussed the themes and examples from the interviews. We then used these descriptions to find instances of these themes in the first study's data. Our study is exploratory and does not aim to quantify the frequency or difficulty of each challenge. Instead, we focused on characterizing the challenges developers experienced.

4 Results

We report results from each of our two studies, answering each research question in turn. We report duration and frequency using the median and the interquartile range (IQR) instead of the mean and standard deviation since our data is not normally distributed (Shapiro-Wilk test=0.5, p-value < 0.05). It is important to note that the median fraction of time is calculated individually for each type of activity. As a result, when these medians are added together, the total may exceed or fall short of 100%.

4.1 How Long Do Developers Spend within Debugging Episodes, and what Changes about Debugging as Episodes Grow Longer?

Debugging episodes in our dataset varied widely in length, ranging from 5 s to two hours of debugging. Looking at the distribution of the length of debugging episodes by quartile, we found that short debugging episodes (bottom quartile) were 30 (18-42) seconds long. In contrast, the longest debugging episodes were many times longer, occupying 19 (15-33) minutes. These long episodes occupied 79% of the total debugging time in our dataset. Therefore, the debugging episodes distribution was skewed with 79% of the total episodes' time stemming from 23 episodes (Fig. 2). This distribution is consistent with prior studies of debugging (Beller et al. 2018), which made use of IDE log analysis. Overall, debugging episodes lasted 4 (1-12) minutes, and developers switched between activities 12 (5-25) times per episode.

We conducted two types of analysis to investigate how debugging episodes change as they grow longer. We first compared our dataset's longest and shortest episodes, which correspond to the top and bottom quartiles. We then examined the correlation between debugging episode time and the fraction of episode time developers spent on each activity.

4.1.1 Longer Debugging Episodes Involved a more Diverse set of Activities

We found that developers engaged in a diverse set of activities in the longest debugging episodes. Long episodes contained 5 (5-6) types of activities while short episodes contained 2 (2-3) types of activities. Unsurprisingly, developers switched between activities more often (39, 32-74, times per episode) during long episodes compared to short episodes (3, 3-4, times per episode). Figure 3 plots the distribution of time developers spent on each activity instance in long and short debugging episodes.

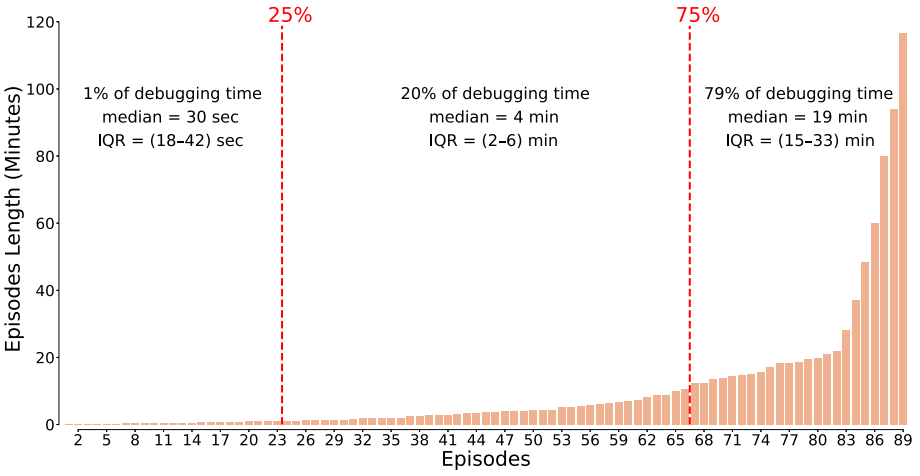


Fig. 2 Debugging episode length, from shortest to longest

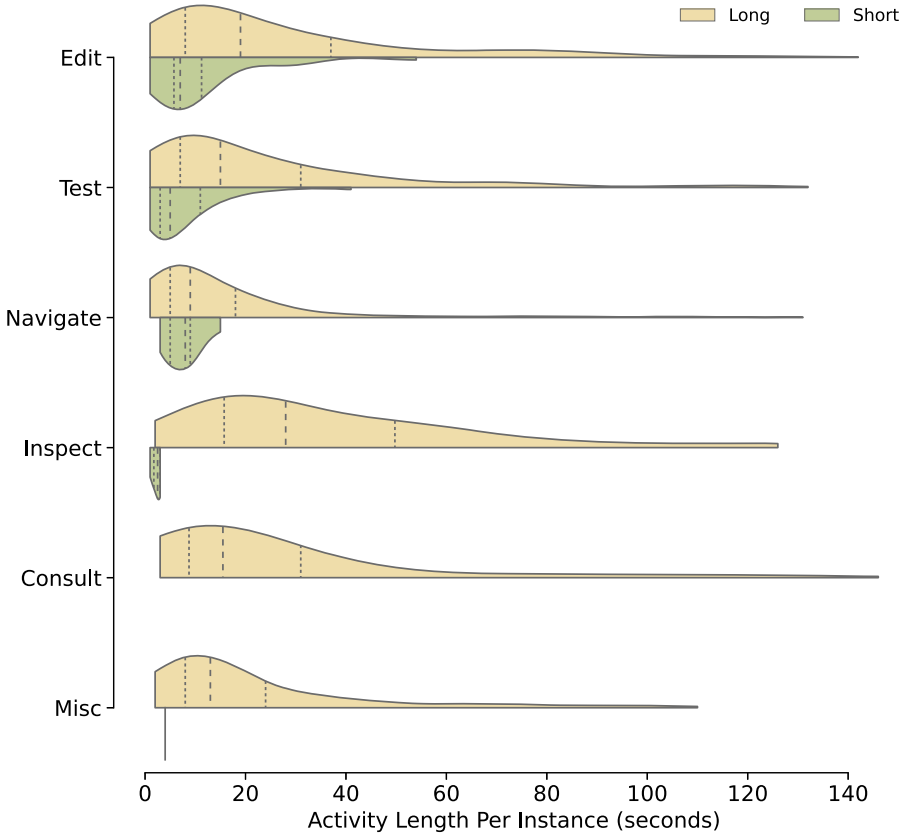


Fig. 3 The distribution of the time developers spent on each activity instance in long and short debugging episodes

In short debugging episodes, developers spent 33% (28-69%) editing and 50% (20-70%) testing, which constituted 83% of the total debugging time. Navigating consumed 0% (0-5%) of the short episode's time since it only occurred in 26% of the episodes. Inspecting and miscellaneous activities were rare, appearing in 17% and 4% of the short episodes. Developers did not consult resources in any of the short episodes.

On the other hand, long debugging episodes contained a diversity of activity types. Developers spent 43% (29-52%) of their time in long debugging episode editing but much less time testing 20% (13-28%) than in short episodes. This gave room for other types of activities to take place. Developers inspected the program state more in long episodes, which occurred in 74% of long episodes and consumed 9% (1-25%) of long episode time. In long episodes, each instance of inspecting the program state was the longest among all activities, 34 (17-65) seconds. For short episodes, developers spent the least time among all activities, 3 (2-3) seconds each time they inspected the program state. Consulting resources occurred during 74% of long episodes and consumed 4% (0%-7%) of the episode time. The miscellaneous activity was common in long episodes, happening in 83% of the episodes and consuming 2% (1%-6%) of the episode time.

4.1.2 Longer Debugging Episodes Involved more Inspecting State, Navigating Code, and Consulting Resources and Less Testing the Program

Since our data is not normally distributed, we built a Spearman's rank correlation between debugging episode length and the fraction of episode time developers spent on each activity. We observed that as debugging episodes increased in length, developers relied more on inspecting the program state and less on testing the program output activities as a source of information. There was a significant correlation between debugging episode length and the fraction of the episode's time developers spent inspecting program state (r =.4, p-value $<$ 0.001). At the same time, developers spent less time testing program output activity (r = -.4, p-value $<$ 0.001).

Developers engaged more with navigating code, consulting resources, and miscellaneous activities as episodes grew. We found no significant correlation between debugging episode length and the fraction of episode time developers spent editing code (Table 6). While editing was the most time consuming activity overall, the fraction of time spent in editing did not significantly change as the debugging episode length changed.

RQ1: Debugging episodes varied widely in length, with 79% of the total debugging time resulting from only 26% of the debugging episodes. As developers spent more time in a debugging episode, they engaged in a more diverse set of activities, spending more of their time inspecting program state, navigating code, and consulting resources and less time testing the program output.

4.2 What Activities are Most Time-Consuming in Debugging?

We found that developers spent the majority of their time within debugging episodes either editing 41% (26-54%) or testing 29% (18-43%). Table 7 and Fig. 4 give an overview of our observations.

4.2.1 Editing and Testing Code Consume the Majority of Debugging Episode Time

Editing code occurred 3 (1-9) times during debugging episodes, consuming 41% (26-54%) of episode time when it occurred. Developers edited 1 (1-4) file of code for each episode (max=12). When editing only a single file, developers did not complete their edits at once. Developers instead switched back and forth an average of three times between editing that

Table 6 Spearman's rank correlation between the fraction of episode time each activity occupied and debugging episode length

Activities	Episode Length	p-value
Edit	0	p-value = 0.868
Navigate	+.4	p-value $<$ 0.001*
Test	-.4	p-value $<$ 0.001*
Inspect	+.4	p-value $<$ 0.001*
Consult	+0.6	p-value $<$ 0.001*
Miscellaneous	+.5	p-value $<$ 0.001*

*Correlation is significant at the 0.05 level

Table 7 The distribution of debugging activities per episode. % of episode time is the fraction of time of the episodes that the activity occupied

Activities	Instances Per Episode			% of Episode Time		
	Median (IQR)	Min	Max	Median (IQR)	Min	Max
Edit	3 (1-9)	0	67	41% (26-54%)	7%	97%
Test	3 (2-7)	0	32	29% (18-43%)	2%	100%
Navigate	3 (0-6)	0	109	15% (9-22%)	1%	50%
Inspect	0 (0-1)	0	26	14% (8-29%)	1%	58%
Consult	0 (0-1)	0	16	9% (4-18%)	0.4%	59%
Miscellaneous	0 (0-1)	0	35	4% (2-9%)	1%	26%

file and other debugging activities. Developers were most likely to test their edit (57%) or inspect program state (24%) after editing code.

Developers engaged in a median of 3 (2-7) *testing the program* activities per debugging episode. When developers tested the program, they spent 29% (18-43%) of episode time on this activity. Developers often ran and observed the program output manually (84%) rather than through automated tests (16%). Developers were likely to switch next to either editing (52%) or navigating the code (36%).

Navigating code occurred 3 (0-6) times per debugging episode. When developers navigated code, they spent 15% (9-22%) of the episode time navigating 3 (2-4) unique files, with only one (0-2) file that they navigated to but never edited within an episode. While developers most often navigated a small number of unique files, developers sometimes navigated as many as 20 unique files without editing any of them. After navigating, developers were most likely to edit code (40%) or test the program (28%).

4.2.2 Most Debugging Episodes did not Involve Inspecting Program State or Consulting External Resources

Inspecting program state was less frequent, occurring in only 40% of debugging episodes. However, when developers did it, they spent 14% (8-29%) of the episode in this activity. Developers spent 23 (10-49) seconds each time they inspected the program state, the longest instance duration of any activity. To inspect the program state, developers most often used log statements (70%) and sometimes breakpoints (30%). Developers were likely to edit (64%) or navigate code (25%) after inspecting the program state.

Consulting resources was the least common activity, occurring in only 33% of debugging episodes. However, 91% of developers consulted external resources at least once. When consulting resources, developers spent 9% (4-18%) of episode time in this activity. Developers primarily searched by typing the query themselves (92%) rather than copying and pasting an error message (8%). The most common information source was API documentation (66%), while developers also consulted posts in Q&A communities (15%), consulting issue trackers (12%), and existing code examples (7%). After consulting resources, developers often edited (37%) or navigated (40%).

Miscellaneous other activities occurred in 38% of debugging episodes but consumed only 4% (2-9%) of the time. This most commonly involved interacting with the development

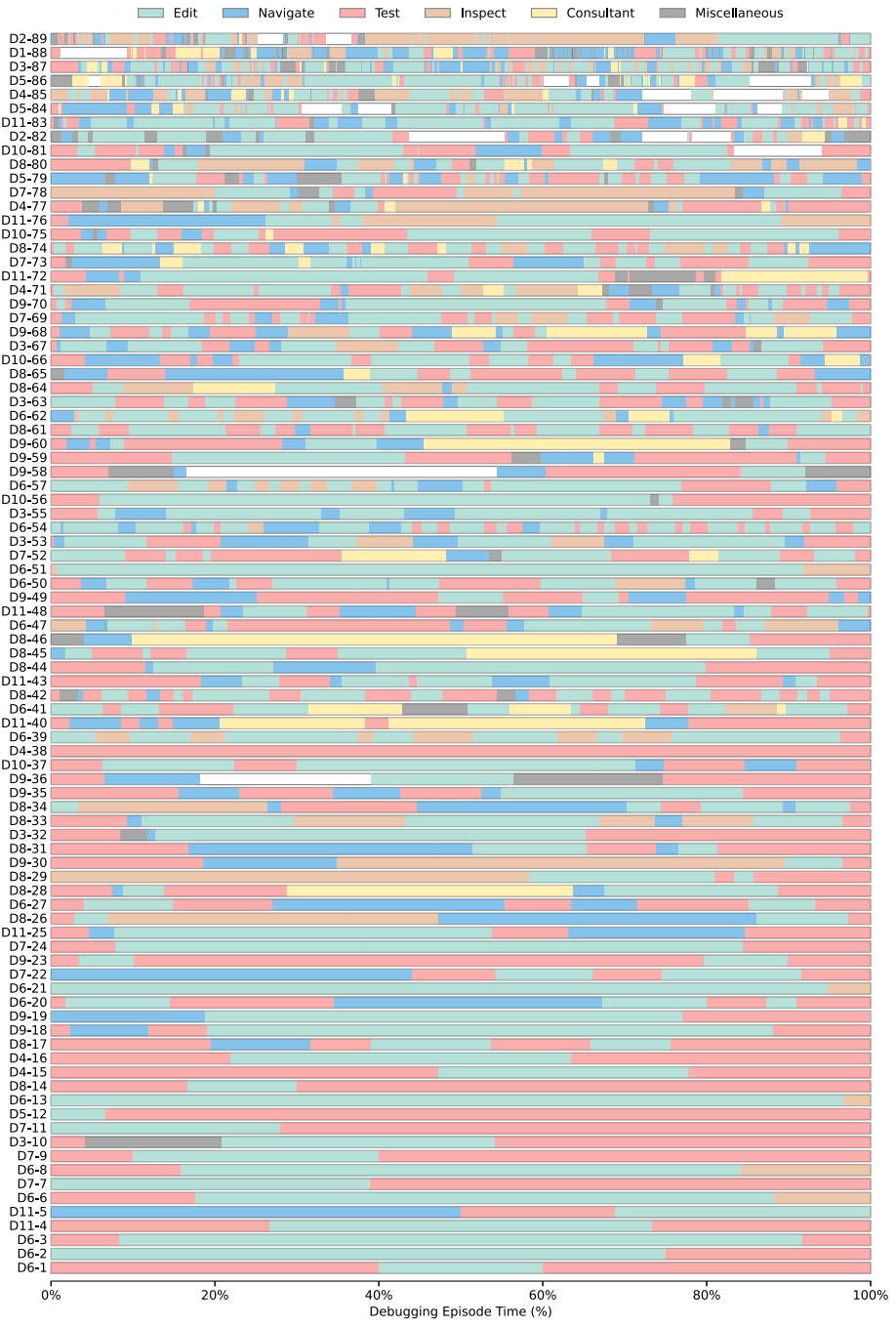


Fig. 4 Distribution of debugging activities within debugging episodes, sorted from shortest (bottom) to longest (top) episode. White bars indicate interruptions (not included in debugging episode time)

environment to open up files and folders 79%, but also involved installing libraries (11%) and note taking (10%).

RQ2: Editing code (41%) and testing the program output (29%) together constituted over two-thirds of debugging time and four times as much time (15%) as navigating code.

4.3 How do Developers Switch Between Activities to Gather Information while Debugging?

Our analysis of developer behavior while switching between different types of activities to gather information yielded three observations, which we discuss below.

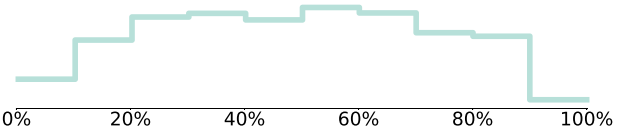
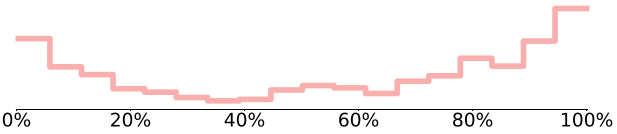
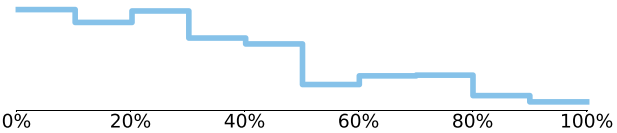
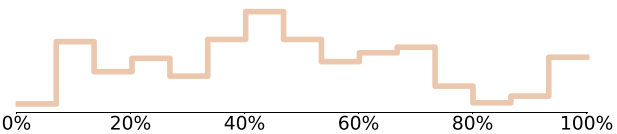
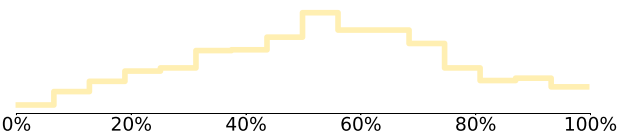
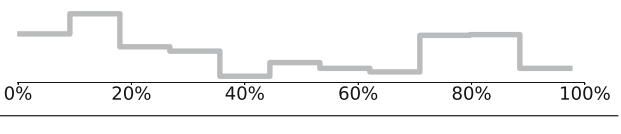
4.3.1 Developers Engaged with Different Activities more Frequently at Varying Periods of the Debugging Episodes

We examined when debugging activities typically occur within a debugging episode and identified *peak* frequencies for all activities (Table 8). Navigating code was more common during the first half of a debugging episode. This might be because developers needed to collect relevant code or localize the defect, but did not need to navigate as much after that. Editing code was widely distributed across episodes, peaking in the middle. Testing was most common at the beginning and end of debugging. One explanation is that developers test when beginning to debug to understand the defective behavior and at the end of debugging to confirm that the fix produces the output they expected. Consulting resources exhibited a strong peak in the middle of debugging episodes. This may correspond to the point when developers first try to fix a defect but get stuck and decide to seek additional information elsewhere. Another explanation is that developers may start implementing a fix and later consult help to understand how to implement the fix. Inspecting the program state also peaked in the middle of debugging episodes. This might be because inspecting program activity requires developers to instrument their code or begin the use of the debugger before having access to the needed information. Miscellaneous activity peaked at the beginning of the episode. This was mostly likely because developers at the beginning of the debugging episode need to open up their IDE, open files, and setup other programs.

4.3.2 Developers used the Source Code as the Anchor Between Activities

The second behavior we observed was related to how developers switched between activities to gather information. Developers used the source code as an anchor point. When developers engaged in activities besides editing and navigating code, they were likely to switch next to editing or navigating. For instance, when developers finished inspecting the program, they were likely to switch to edit (83%) or navigate code (25%). In contrast, they switched to consult resources or miscellaneous activities only 7% and 4% of the time. Figure 5 shows developers' switching behavior between activities. Anchoring focus on the source code allows developers to validate and reason about newly acquired information. For example, when developers inspect the program and discover that certain variables contain incorrect values, they may return back to the source code to understand why these values were incorrect.

Table 8 The distribution of frequency (count) of activities throughout the debugging episodes

Activities	Distribution of Frequency Across Episodes Time
Edit	
Test	
Navigate	
Inspect	
Consult	
Miscellaneous	

Developers may also formulate a hypothesis based on information acquired from engaging in different activities. In that situation, developers may need to test their hypothesis by editing the code, a step that corresponds to the instrument hypothesis in the Layman et al. debugging model (Layman et al. 2013).

Our observations show developers do not transition immediately from testing a program to inspecting it. There are several potential explanations for this behavior. Current development environments do not facilitate a seamless transition from the program's output to the debugger, requiring other activities first. For example, suppose a developer notices an issue in the program's output that requires the use of the debugger to understand. In that case, they may have to "navigate" to a particular code location to set a breakpoint before inspecting the program state. Another potential explanation is that the information developers need during testing is often related to activities that do not require program inspection.

4.3.3 Developers Switched Debugging Activities After Less than a Minute

We observed that developers' engagement with individual activity instances was short and frequent rather than long and sustained. For example, when developers sought to collect

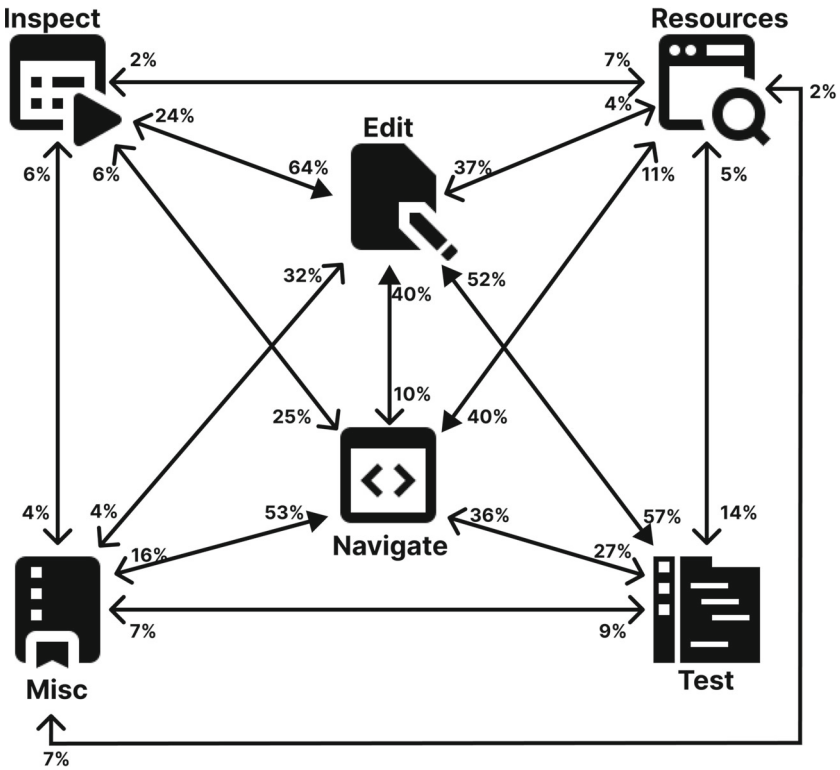


Fig. 5 Developers' switching behavior between activities. The most common switches between activities are denoted with a triangle arrow

runtime information, they inspected the program state 3 (1-6) times per debugging session but spent only 23 (11-49) seconds each time. This behavior was consistent across all debugging episodes, even the longest episodes in our dataset. When we examined the longest episode in our dataset, in which the developer spent more than two hours debugging, we found that most activity instances (83%) were under one minute.

The distribution of time developers spent on each activity type varied between activities. For example, the time spent in inspecting and editing activities were widely distributed, with instance durations ranging from as short as 1 s to over 16 min. In contrast, the duration of the other activities had a narrower range, lasting from 1 s to 5 min. Figure 6 provides a visual representation of the distribution of debugging activity length.

Frequent and short activities allow developers to gather information incrementally as needed. Instead of reading the entirety of documentation about an API, developers may read and copy a few lines of code to test and see the output before switching back to the documentation. Instead of inspecting the entire program state, developers may log a small portion of the program state, read the source code, and then decide based on the values what other part of the program state to explore next. Researchers have observed similar behavior of short and frequent switches during software design at the whiteboard (Mangano et al. 2014).

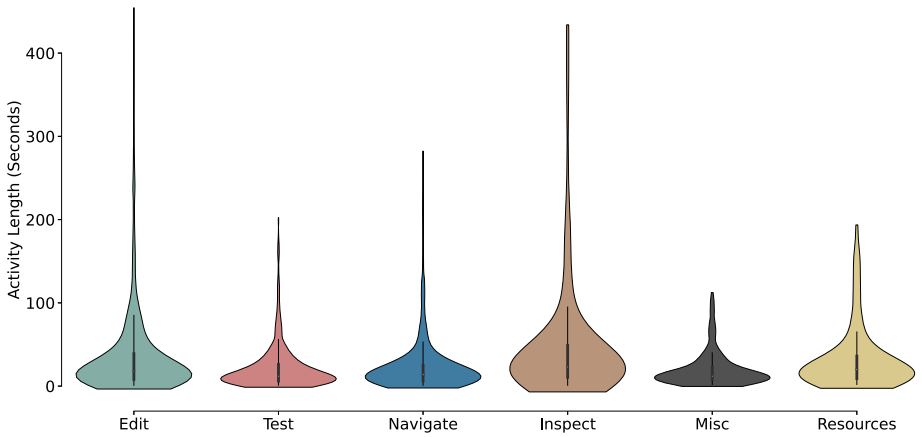


Fig. 6 The distribution of the length of debugging activities

RQ3: Common debugging activities varied across debugging episodes. When switching between debugging activities, developers used the source code as an anchor. Developers' engagement with activity instances was short and frequent, rapidly switching between activities.

4.4 How Does Debugging Differ for Fresh and Committed Defects?

We next examined how debugging activity differs between when developers are working to debug committed defects found in an issue tracker and in debugging freshly inserted defects. We made two observations characterizing debugging with fresh and committed defects.

4.4.1 Debugging Committed Defects took Longer than Fresh Defects

Developers spent 29 (19-75) minutes debugging each committed defect. In contrast, developers spent just 3 (1-7) minutes debugging defects they had just inserted themselves while programming (Fig. 7). This may suggest that, given their longer length, debugging committed defects is more challenging than debugging fresh defects. However, while debugging episodes concerning fresh defects were generally short, they were also frequent. While programming, developers constantly inserted and debugged new defects, on average after only 8 min of programming. Moreover, debugging fresh defects was not always fast: 25% of episodes lasted for 15 (12-18) minutes or more. Not all debugging episodes concluded with a successful fix, suggesting that some episodes might be longer if the developers continued work. Half of the debugging episodes focused on committed defects and did not end with a successful fix, either because more information was needed to reproduce the defect or the developer deferred debugging until later. For new defects created while programming, 14% did not conclude with a successful fix, either because the developer had higher priority tasks (e.g., shipping the feature even if it is not completely correct) or deferred the work until later.

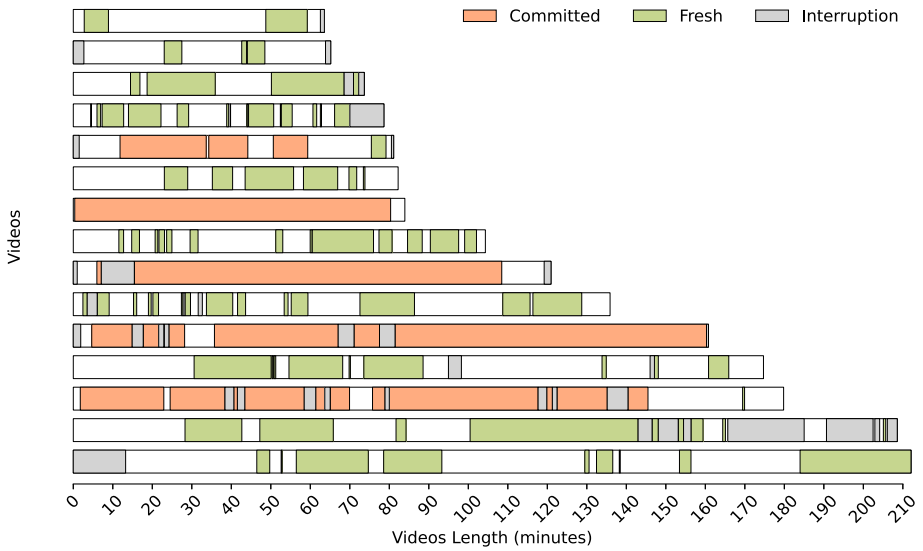


Fig. 7 Video timelines (ordered from shortest to longest) with fresh and committed debugging episodes

4.4.2 Debugging Committed Defects Involved more Activity Types than Fresh Defects

Almost all committed debugging episodes contained all activity types. In contrast, not a single fresh debugging episode contained all activity types. Fresh debugging episodes varied in activity types. Figure 8 illustrates these variations, with fresh defects containing a longer tail and indicating their wider variation.

Editing code occurred in all committed debugging episodes and almost all fresh episodes (97%). It consumed 37% (27-49%) of the committed episode time and 40% (25-54%) of the fresh debugging episode time. *Testing* the program occurred in all committed debugging episodes and almost all fresh episodes (95%). Interestingly, developers spent twice as much time 29% (18-44%) testing in fresh episodes as in committed 15% (10-21%) episodes. *Navigating* code occurred in all committed episodes, occupying 19% (10-28%) of episode time. In contrast, developers only navigated between files in 70% of fresh debugging episodes, constituting 9% (0-17%) of the episodes' time. *Inspecting* the program occurred in 90% of committed debugging episodes but only 34% of fresh episodes. While more common, it consumed only 9% (3-16%) of the episode time compared to 0% (0-9%) of fresh episodes' time. *Consulting resources* occurred in all committed debugging episodes, but only 15% of fresh debugging episodes. *Miscellaneous* activity occurred in 90% of the committed episodes and in only 32% of fresh episodes, occupying 6% (2-9%) and 0% (0-1%) of the episodes, respectively.

RQ4: Debugging committed defects was substantially more time consuming than fresh defects and involved a wider range of activity types.

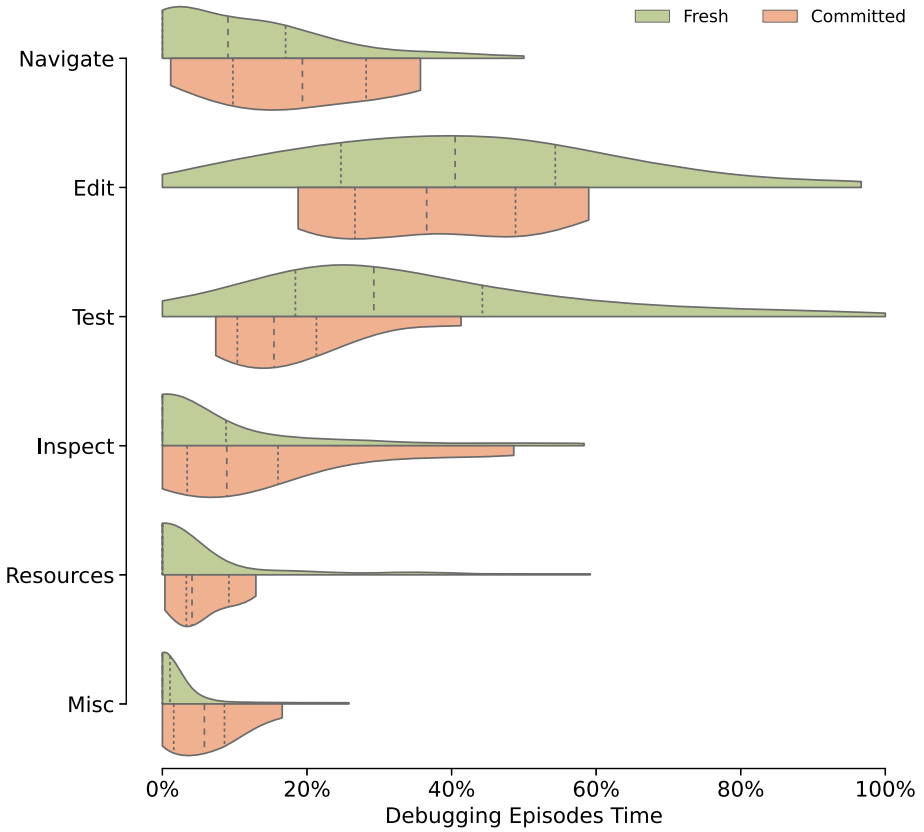


Fig. 8 The distribution of the debugging episode time (%) developers spent on each activity in fresh and committed defects

4.5 How Do Activities Performed during Debugging Episodes Differ from Those in Implementation Work?

To examine differences in how developers spend their time during debugging and implementation work, we examined 1477 activities across 13 h of implementation work and 2135 activities across 15 h of debugging. Our investigation focused on the overall time spent on each activity. We found that debugging and software implementation work is broadly similar in the time developers spend on each activity instance. However, there were differences in the frequency of activities. Table 9 summarizes the differences.

Navigating and editing were most similar in their occurrences within debugging and implementation work. Navigating constituted 13% of implementation work time and 16% of debugging time. Developers spent 4 (1-8) seconds in each navigation while developing and debugging. Editing consumed the most time in debugging and implementation work, constituting 35% of the implementation work time and 38% of debugging time. Every time a developer edited a file of code, they spent 22 (10-44) seconds in implementation work and 17 (8-34) seconds in debugging.

Implementation work and debugging were less similar in consulting, testing, inspecting, and miscellaneous activities. Consulting resources consumed 6% of debugging and 24% of

Table 9 The distribution of debugging activity length in implementation work (Imp) and debugging (Deb)

Activities		Length Per Instance (Sec)	% of Total Time
Edit	Deb	17 (8-34)	38%
	Imp	22 (10-44)	35%
Test	Deb	12 (6-26)	20%
	Imp	13 (6-29)	11%
Navigate	Deb	4 (1-8)	16%
	Imp	4 (1-8)	13%
Inspect	Deb	23 (11-49)	16%
	Imp	17 (11-27)	2%
Consult	Deb	9 (2-20)	6%
	Imp	14 (1-32)	24%
Miscellaneous	Deb	12 (6-26)	5%
	Imp	13 (6-29)	15%

implementation work time, with developers spending 14 (1-32) and 9 (2-20) seconds each time they engage in this activity during implementation work and debugging, respectively. The miscellaneous activity constituted 15% of the implementation work time compared to 5% of debugging time. When programming, developers installed libraries, set up their IDE, and opened other tools and software. This suggests that debugging is more code-focused. Testing was more common in debugging (20%) than in implementation work (11%). When testing, developers spent comparable time for each testing activity in implementation work and debugging. Inspecting the program state was rare in implementation work (2%) and far more common in debugging (16%). Inspecting the program state had the longest median length among all debugging activities, while editing was the longest among all implementation work activities.

RQ5: When working in implementation work and debugging, developers spend a similar fraction of their time editing and navigating. Developers spend substantially more time inspecting the program when debugging (16%) than during implementation work (2%). Implementation work involves considerably more consulting external resources (24%), such as browsing documentation and issue trackers, than debugging (6%).

4.6 What are the Challenges that Developers Experience during Debugging that Require Frequent Switches Between Different Activities?

As debugging episodes grew in length, developers engaged with a diverse range of activities. Our interviews suggest that this is related to the challenges developers faced in understanding and fixing defects. We report findings from our interviews, connecting them back to our analysis of debugging videos where applicable.

4.6.1 Collecting a Long and Scattered List of Information to Understand Defects

Interviewees described their process of understanding defect behavior by reference to a diverse set of activities, such as *"a lot of fiddling with code"* [Editing and Testing], *"inserting*

logs" [Inspecting the program], *"consulting documentation"* [Consulting resources], *"reading more code"* [Navigation], and *"looking at commit histories"* [Consulting resources]. Their goal was to *find* and *connect* information that offers an explanation for the defect behavior. Information included suspicious program state (I1, I2, I6), input that triggers the defect (I1, I4, I5), explanations of API output (I2, I7, I7, I9, I10), and code rationale (I1, I3, I4, I7). Interviewees faced challenges when the defective observed behavior was substantially different from the intended behavior or when information was scattered across different sources. Interviewees described this challenge as debugging multiple defects (I2) and as an incremental process of collecting and connecting information:

> *We restarted the server. The system started sending off messages, and then we look into logs and metrics. There was nothing unusual going on. Then we found that our server was restarting itself. But why? Then we looked at some logs on the server. It is like, oh, it is the memory usage...* -I6

Interviewees' starting point to collect information was the defect report (I1, I2, I5-I7) or the source code (I1-I4, I6-I9). However, interviewees described struggles to understand the defect by only looking at the source code, requiring them to switch activities to collect and connect information from different sources. Interviewees had to inspect code, search the internet for information, browse commit history, and talk to other developers while debugging.

> *[After reading some code] I had to look at the project repository to see who has worked on it using the commit histories [...] Then, I reached out to the developer that implemented the code to understand what the reasoning behind [the current implementation] and what was the desired outcome.* -I1

Developers D1, D4-D6, D8-D10 switched between inspecting, editing, navigating, testing, consulting resources, and miscellaneous activities while debugging. Switching between activities sometimes required switching between tools (e.g. IDE, browser, version control system) and manually connecting information across different tools. For example, D9 had a defect related to routing APIs. He switched to the browser and searched for relevant information, typing *"Razor page link tag helper does not work from within subfolder"*. He started reading from different resources and switched back to the IDE, spending almost 30% of the debugging time reading online documentation and developers' posts.

Prior work examining developers' information seeking has also found that developers use different sources to collect information (Ko et al. 2006; LaToza et al. 2006). Tools such as Code Bubbles (Bragdon et al. 2010) help developers collect scattered information in the source code as editable fragments called bubbles. Another study (Eisenstadt 1993) found that debugging is challenging when the symptom is far in the execution from the root cause, requiring developers to spend substantial time traversing from the symptom to the underlying cause. Our findings suggest that collecting scattered information is one reason for engaging in many activities in debugging. In most challenging debugging episodes, developers had to engage in diverse activities to collect information scattered across code locations, program states, documentation, and commit history.

RQ6.1 Developers switched between debugging activities to find and connect information about the defect. When the defect exhibited behavior substantially different than intended or when information was scattered across multiple sources, developers worked to find and connect more information.

4.6.2 Changes to the Source Code that Depend on or Impact Third-Party Code

Interviewees described their work fixing defects mainly through the steps taken to edit code. Edits were sometimes as simple as a *"single line of code"* patch (I1, I2, I4). In other cases, interviewees (I6-I10) reported examples where they had to edit multiple lines of code, search for code examples and read documentation to fix the defect. They reported challenges analyzing the impact of a fix (I7, I8) and learning new APIs and concepts necessary for the fix (I6, I10, I8).

Changing the program to fix a defect has the potential to impact program behavior. For programs that expose a public API, introducing a fix requires reasoning about the impact on the API users.

> *The defect was related to an internal stakeholder and we needed to bring in [the] ux ui team, our project manager, and product manager to anticipate the future use cases of the fix.* -I7

In our observations, D5 and D11 discussed how different ways of fixing a defect might impact external dependencies. D5 debugged a defect reported in Downshift, a popular JavaScript library with over 40,000 projects using it according to GitHub.[1] 17 min into debugging, the developer started work towards a fix. Aware of the library's popularity, the developer carefully considered the changes introduced to fix the defect and their potential impact on other projects that depended on the API.

> *Maybe one way to fix this is by allow[ing] the users to change the internal behavior not though exposing the properties but through extending the component. But this makes me scare[d], because that might cause breaking changes. This is why I would rather expose the properties to users.* -D5

Interviewees also reported (I6, I10, I8) that they searched for examples and browsed documentation while working on a fix. They had a high-level plan for a fix but lacked knowledge about implementing it.

> *[To implement the fix] I needed to figure out how to setup the router configuration and the best practice for that. I got [this] information from googling and reading Stackoverflow.* -I10

Developers D4 and D10 in the observational study struggled to implement the fix because it involved unfamiliar concepts and APIs. Their strategy was to search for documentation and code examples manually. The search was often not successful, requiring them to test multiple ways to implement the fix.

> *I want to make Tox work with proper configuration, but I do not know how to do that. I am not familiar with how [an object] get created. [searching the codebase] maybe I need to call this API?.* -D4

These findings confirm prior findings that some of the hardest and most frequent questions developers ask are related to the impact of changes on the source code (LaToza and Myers

[1] https://github.com/downshift-js/downshift.

2010a, b; Sillito et al. 2008). Our study offers additional evidence that introducing changes in the code results in developers engaging in more activities while debugging.

RQ6.2: Developers engage in many different activities to prevent their fixes to defects from introducing breaking changes that could affect external dependencies. This is particularly crucial when dealing with unfamiliar source code or APIs, where developers may need to invest more time and effort to understand the system and verify the correct usage of APIs.

5 Threats to Validity

Our studies have several important limitations.

Construct Validity Defining exactly when debugging episodes and activities start and end is challenging and potentially susceptible to human error. To minimize the risk of incorrect codes, we collected past definitions of debugging (Johnson 1982; Ko et al. 2011; Parnin and Orso 2011) and used these to create initial definitions. We then built the initial coding scheme for activities. We refined the definitions until two authors could independently and consistently annotate the start and end of video segments within one to two seconds of error. To minimize the risk of confounding factors which might impact how developers debug, we conducted our second study through semi-structured interviews rather than as a laboratory study. For example, developers may not engage in their typical inspection activities if they are uncertain about how to set up the debugging environment in an unfamiliar context. To address these potential confounds, we mapped the observations made in our first study with developers' descriptions of recent debugging episodes in the second study.

Internal Validity Developers were sometimes interrupted while debugging. To ensure our measures of debugging episodes and activity times did not include any irrelevant work caused by interruptions, we coded any interruption that lasted more than five seconds and excluded them from the debugging and programming work. After observing that interruptions that lasted less than that did not cause the developers to pause and switch context, we defined the five-second threshold. A threat to the internal validity of Study 2 was that developers were self-reporting episodes and may not remember all of the details. To mitigate this issue, we explicitly asked developers to report recent episodes and excluded those instances for which they could not remember details or provide answers to our follow-up questions.

External Validity One important potential threat to the external validity of the first study is the dataset we used to observe developers while debugging. Researchers have found that live-streamed programming depicts developers at work on open source projects (Alaboudi and LaToza 2019a) and used it to observe developers while working (Alaboudi and LaToza 2021). However, developers who know they are being observed during live-streamed programming may alter their behavior by selectively showing easier tasks or editing out parts that showcase their struggles or confusion. In addition, developers may choose to socialize with developers watching their work more than doing actual work. In order to mitigate these potential threats, we developed inclusion criteria to only select videos which featured substantial development work with minimal interruptions. These videos were limited to those depicting developers working on nontrivial projects that are currently in use. To ensure a diverse sample of developers, we included participants across a range of experience levels,

from seven to 31 years of committing to open source projects. We carefully reviewed all videos to ensure that they were not edited or manipulated in any way. Additionally, we only included developers who regularly stream their development work, excluding those with only a few streams that might not be representative of their typical work.

6 Discussion

In this paper, we offered the first study of debugging episodes to quantify and investigate developers' activities while debugging. We found that debugging happens often—every 8 min during implementation work—and that most debugging episodes were short, with a median length of just 4 min. However, a quarter of longer debugging episodes occupied nearly 80% of developers' debugging time. We found that developers spent most of their debugging time editing code and testing the program. Debugging episodes that lasted tens of minutes contained diverse types of activities, while shorter episodes consist mainly of editing and testing. As debugging episodes grew longer, developers spent more time inspecting the program, navigating code, and consulting resources but less time testing the program. While switching between activities, developers used the source code as the anchor and spent less than a minute per debugging activity. Debugging episodes for committed defects were long in length and more consistent in activity types, while episodes for fresh defects were much shorter and varied greatly in their activities. The time spent in activities in debugging and implementation work was surprisingly similar, particularly editing and navigating code. However, developers spend much more time inspecting the program when debugging than during implementation work. implementation work involved considerably more consulting resources activities, such as browsing documentation and issue trackers, than debugging.

As the length of a debugging episode increases, developers engage in a wider range of activities. We found that when developers trace a long chain of information to comprehend the causes of a defect, they employ a variety of types of activity. Similarly, investigating the potential for introducing breaking changes and working with unfamiliar code also leads developers to work across more different types of activity.

Our findings offer insights into the potential for new types of debugging tools as well offering implications for education. In this paper, we also introduce observe-dev.online, a platform we developed to assist researchers in analyzing software development work through the use of live-streamed programming videos.

6.1 Implications for Debugging Tools

Our findings suggest that developers in longer and more challenging debugging episodes switch between activities to gather information from different sources. We found that developers constantly switch activities, spending less than a minute per activity instance. This suggests the importance of tools that support developers in their process of integrating all of this information. Rather than simply collecting and displaying information from logs, slices, or lists of potential fault locations in separate specialized debugging tools, debugging tools would better support the full scope of debugging work when they integrate information across multiple sources. For example, a tool might take information about potential causes of a defect from a StackOverflow post, link it to the official API documentation to explain the meaning of arguments, use log data to gather actual runtime data from the developers' program, and then integrate all of this back together.

At the same time, we found that source code remains the central anchor point in debugging tasks. Developers constantly return to the source code after gathering information from myriad sources. Debugging tools might support this process by offering ways to directly query related information sources based on the current view of a code a developer in considering. And as developers identify and find information, annotate and integrate this information in the context of the source code, tools can use the information gathered to suggest further queries across additional resources. Finally, developers should be supported in interpreting this information, helping them in their process of generating and testing debugging hypotheses.

Another suggestive finding was the broad similarity in activities between debugging and implementation work. While requiring further investigation to better understand, it suggests that some of the same challenges of understanding may be relevant in both, further suggesting that tools may often be relevant across both implementation work and debugging tasks. For example, the debugger might be used in a debugging task by a developer trying to understand behavior that led to a defect or in implementation work to understand the runtime state of the program when developers are trying to modify the code. Rather than conceive of tools as either separately supporting implementation work or debugging, it may sometimes be more helpful to conceive of them as supporting understanding, across both debugging and implementation work, particularly for tools such as live programming, code search, and navigation.

During debugging episodes, developers often spend a significant portion of their time on editing and testing activities. This tinkering behavior allows developers to experiment with different solutions, explore APIs, observe the side effects of code changes, and test their hypotheses. However, this can also be a time-consuming and error-prone process. To support developers in this behavior, debugging tools should aim to streamline the tinkering process, much like live programming environments do. By providing a more responsive and interactive programming environment, debugging tools can help developers iterate more quickly and efficiently, reducing the time and effort required to identify and fix defects. Furthermore, debugging tools should also track the edits made by developers and their impact on the code. This can help developers explore and test their hypotheses more effectively, as they can quickly identify the changes that led to a particular behavior and understand how those changes affected the program's overall behavior. This feature can also help developers to return to the original code faster and with greater confidence, as they can easily undo changes that did not produce the desired results or caused unintended consequences.

6.2 Implications for Debugging Education

Debugging is a crucial skill for students, yet it has long been challenging for educators to teach. Our work offers insights into the activities involved in debugging and how developers engage with the process, which has important implications for teaching debugging to students.

To effectively teach debugging to students, educators should broaden their perspective to recognize debugging as a multifaceted process that extends beyond the act of using a debugger. Rather than solely focusing on strategies to use the debugger, students should also learn that professional developers spend most of their debugging time editing and testing their code, engaging in a constant and iterative process of tinkering with the program. Additionally, the nature of debugging can vary depending on the stage of the process, with developers often spending more time navigating code at the beginning and seeking external resources for assistance in the middle. By emphasizing the range of activities involved in debugging, educators can help students better understand how professional developers approach and carry

out debugging. Our study highlights the importance of switching effectively between activities during debugging. Therefore, educators should teach students how to switch between activities efficiently, such as using shortcuts, speeding up their testing environment, toggling the debugger quickly, and navigating code more effectively. By teaching these skills, students may develop the ability to debug more efficiently.

6.3 Enabling Observational Studies of Developers

Observational studies in software engineering have traditionally been limited by the difficulty of gaining access to software developers at work on real projects and the impossibility of sharing datasets due to confidentiality. Much as widespread access by researchers to the repositories of open source projects or questions and answer on Stack Overflow has led to the proliferation of empirical software engineering (Lakhani and Von Hippel 2004; Mamykina et al. 2011; Singer et al. 2014; MacLeod et al. 2015; Chatterjee et al. 2019), we believe these live-streamed videos offer a similar opportunity, complimenting these datasets by offering the ability to answer new questions where direct observation is required.

Live-streamed programming videos offer an important opportunity for researchers to observe professional developers in a natural setting. Conducting studies with similar settings would require researchers to conduct field studies and record developers' screens and voices during the work. Live-streamed programming is an alternative that requires no such effort with public access to both the source code and recording.

To enable this opportunity, in this paper we contribute the *Observe-Dev.online* platform. We built this platform to make observational studies of software development work easier to conduct and share. Observe-Dev.online offers four key features for supporting the use of live-streamed programming videos in software engineering research. First, the platform offers *a dataset of programming episodes*. Identifying live-streamed programming videos can be time-consuming, particularly in identifying videos with specific characteristics (e.g., working on a data analysis script in Python). Therefore, Observe-Dev.online includes a default dataset that is publicly available for use (Default dataset: Observe.dev 2022). Each episode is labeled with metadata, describing the programming languages, projects, and development environments used. Researchers can use this to filter episodes to match inclusion criteria. Second, Observe-Dev.online offers the *ability to annotate video segments*. Figure 9 depicts the platform interface for annotating a live-streamed programming video. Researchers can create new codes and annotate specific video segments with these codes. Segments may vary in duration from one second to the entirety of the video time. After applying codes to segments, the tool offers a mini-timeline visualization of the codes, enabling them to see where codes are located at a glance and quickly navigate to specific code locations. For example, Fig. 9 shows three codes for programming, debugging, and irrelevant episodes, each shown in a unique color. Third, researchers can *share annotated datasets*. Observe-dev.online is a web-based platform that enables datasets to be publicly or privately shared for viewing or editing through a URL and optional authentication. Finally, the platform supports *exporting annotations* to a standard JSON format that researchers can import into other tools for further analysis.

Our platform includes a default dataset which we used in our study and which is publicly available through the platform (Study dataset: Observe.dev 2022). Our dataset is based on live-streamed programming videos available online. As we used this rich source of information to explore debugging, we believe that researchers can use these videos to explore further debugging or observe developers exploring other research areas related to devel-

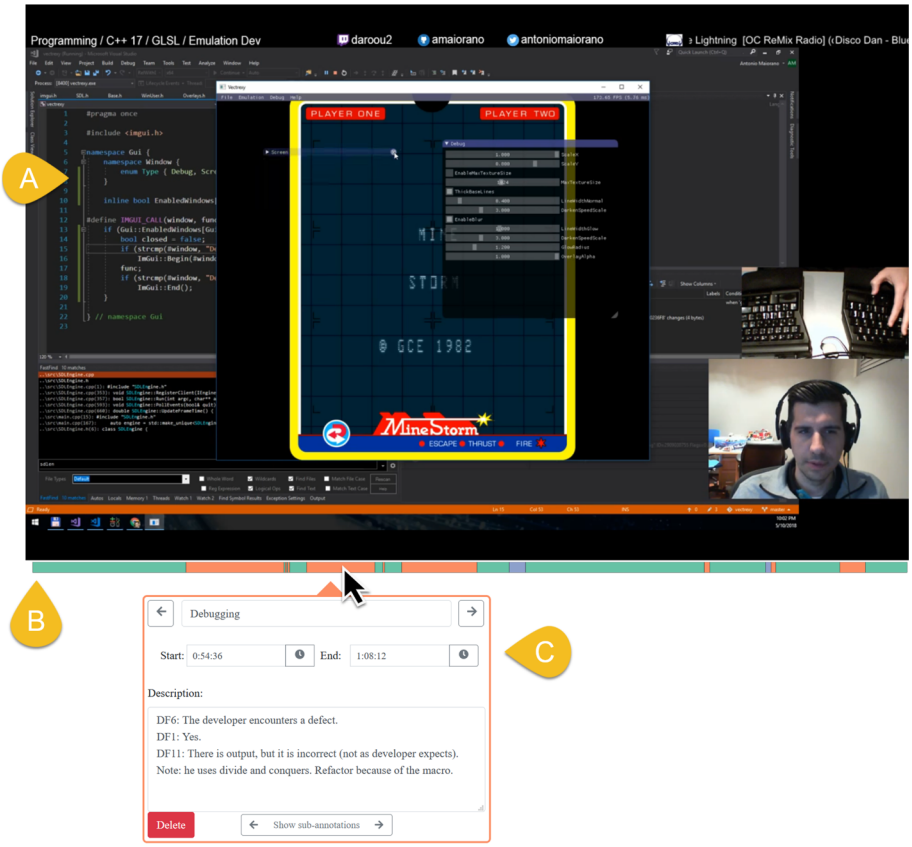


Fig. 9 Observe-dev.online supports collaborative qualitative analysis of lives-streamed programming videos. Programming videos (**A**) are shown with an interactive timeline view (**B**) of video segment annotations supporting navigating between segments. New codes can be edited (**C**) or created based on the current position in the video

opers' programming behavior. For instance, researchers may investigate the use of online resources during debugging and programming work, the challenges developers face working in specific programming languages, or how developers make design decisions.

Funding This research was funded by NSF award 1845508.

Data Availability The dataset supporting the conclusions of this research is included within this paper.

Declarations

Conflicts of Interest/Competing Interests All authors declare that they have no conflicts of interest.

References

Replication package (2022). URL https://figshare.com/s/0e9eac98b8ddd54c384c

Abad ZSH, Karras O, Schneider K, Barker K, Bauer M (2018) Task interruption in software development projects: What makes some interruptions more disruptive than others? In: International Conference on Evaluation and Assessment in Software Engineering pp. 122–132
Afzal A, Goues CL (2018) A study on the use of ide features for debugging. In: Proceedings of the 15th International Conference on Mining Software Repositories p. 114–117
Alaboudi A, LaToza TD (2019a) An exploratory study of live-streamed programming. In: IEEE Symposium on Visual Languages and Human-Centric Computing pp. 5–13
Alaboudi A, LaToza TD (2019b) Supporting software engineering research and education by annotating public videos of developers programming. In: International Workshop on Cooperative and Human Aspects of Software Engineering pp. 117–118
Alaboudi A, LaToza TD (2020) Using hypotheses as a debugging aid. In: 2020 IEEE Symposium on Visual Languages and Human-Centric Computing pp. 1–9
Alaboudi A, LaToza TD (2021) Edit-run behavior in programming and debugging. In: IEEE Symposium on Visual Languages and Human-Centric Computing
Amann S, Proksch S, Nadi S, Mezini M (2016) A study of visual studio usage in practice. In: IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 124–134
Baltes S, Moseler O, Beck F, Diehl S (2015) Navigate, understand, communicate: How developers locate performance bugs. In: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement pp. 1–10
Beller M, Spruit N, Spinellis D, Zaidman A (2018) On the dichotomy of debugging behavior among programmers. In: International Conference on Software Engineering pp. 572–583
Böhme M, Soremekun EO, Chattopadhyay S, Ugherughe E, Zeller A (2017) Where is the bug and how is it fixed? an experiment with practitioners. In: The Joint Meeting on Foundations of Software Engineering, pp. 117–128
Bragdon A, Zeleznik R, Reiss SP, Karumuri S, Cheung W, Kaplan J, Coleman C, Adeputra F, LaViola Jr, JJ (2010) Code bubbles: a working set-based interface for code understanding and maintenance. In: Conference on Human Factors in Computing Systems pp. 2503–2512
Britton T, Jeng L, Carver G, Cheak P, Katzenellenbogen T (2013) Reversible debugging software. Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep 229
Chatterjee P, Damevski K, Pollock L, Augustine V, Kraft NA (2019) Exploratory study of slack q&a chats as a mining source for software engineering tools. In: International Conference on Mining Software Repositories pp. 490–501
Chattopadhyay S, Nelson N, Gonzalez YR, Leon AA, Pandita R, Sarma A (2019) Latent patterns in activities: A field study of how developers manage context. In: International Conference on Software Engineering p. 373–383
Coker Z, Widder DG, Le Goues C, Bogart C, Sunshine J (2019) A qualitative study on framework debugging. In: International Conference on Software Maintenance and Evolution pp. 568–579
Damevski K, Shepherd DC, Schneider J, Pollock L (2017) Mining sequences of developer interactions in visual studio for usage smells. IEEE Transactions on Software Engineering 43(4):359–371
Study dataset: Observe.dev (2022). URL https://bit.ly/3kkbL2W
Default dataset: Observe.dev (2022). URL https://bit.ly/3qWdMVA
DeMillo RA, Pan H, Spafford EH, DeMillo RA, Pan H, Spafford EH (1996) Critical slicing for software fault localization. In: The International Symposium on Software Testing and Analysis pp. 121–134
Eisenstadt M (1993) Tales of debugging from the front lines. In: Empirical Studies of Programmers: Fifth Workshop pp. 86–112
Faas T, Dombrowski L, Young A, Miller AD (2018) Watch me code: Programming mentorship communities on twitch.tv. Proceedings of the ACM on Human-Computer Interaction 2, 50:1–50:18
Gould JD (1975) Some psychological evidence on how people debug computer programs. International Journal of Man-Machine Studies 7(2):151–182
Gould JD, Drongowski P (1974) An exploratory study of computer program debugging. Human Factors 16(3):258–277
Gugerty L, Olson G (1986) Debugging by skilled and novice programmers. In: Conference on Human Factors in Computing Systems pp. 171–174
Jiang S, McMillan C, Santelices R (2017) Do programmers do change impact analysis in debugging? Empirical Software Engineering 22:631–669
Johnson MS (1982) A software debugging glossary. ACM Sigplan Notices 17(2):53–70
Jones JA, Harrold MJ, Stasko J (2002) Visualization of test information to assist fault localization. In: Proceedings of the International Conference on Software Engineering pp. 467–477
Katz IR, Anderson JR (1987) Debugging: An analysis of bug-location strategies. Human-Computer Interaction 3(4):351–399

Ko AJ, Abraham R, Beckwith L, Blackwell A, Burnett M, Erwig M, Scaffidi C, Lawrance J, Lieberman H, Myers B et al (2011) The state of the art in end-user software engineering. ACM Computing Surveys (CSUR) 43(3):1–44

Ko AJ, DeLine R, Venolia G (2007) Information needs in collocated software development teams. In: The International Conference on Software Engineering pp. 344–353

Ko AJ, Myers BA, Coblenz MJ, Aung HH (2006) An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. IEEE Transactions on Software Engineering 32(12):971–987

Lakhani KR, Von Hippel E (2004) How open source software works: "free" user-to-user assistance. In: Produktentwicklung mit virtuellen Communities, pp. 303–339. Springer

Landis JR, Koch GG (1977) The measurement of observer agreement for categorical data. Biometrics 33:159–174

LaToza TD, Myers BA (2010a) Developers ask reachability questions. In: The International Conference on Software Engineering, pp. 185–194

LaToza TD, Myers BA (2010b) Hard-to-answer questions about code. In: PLATEAU Workshop on Evaluation and Usability of Programming Languages and Tools, pp. 1–6

LaToza TD, Venolia G, DeLine R (2006) Maintaining mental models: A study of developer work habits. In: The International Conference on Software Engineering pp. 492–501

Lawrance J, Bogart C, Burnett M, Bellamy R, Rector K, Fleming SD (2013) How programmers debug, revisited: An information foraging theory perspective. IEEE Transactions on Software Engineering 39(2):197–215

Layman L, Diep M, Nagappan M, DeLine RA (2013) Debugging revisited, toward understanding the debugging needs of contemporary software developers. In: Empirical Software Engineering and Measurement

Lukey F (1980) Understanding and debugging programs. International Journal of Man-Machine Studies 12(2):189–202

MacLeod L, Storey MA, Bergen A (2015) Code, camera, action: How software developers document and share program knowledge using youtube. In: The International Conference on Program Comprehension pp. 104–114

Mamykina L, Manoim B, Mittal M, Hripcsak G, Hartmann B (2011) Design lessons from the fastest q&a site in the west. In: CHI Conference on Human Factors in Computing Systems pp. 2857–2866

Mangano N, LaToza TD, Petre M, van der Hoek A (2014) How software designers interact with sketches at the whiteboard. IEEE Transactions on Software Engineering 41(2):135–156

Weiser W (1984) Program slicing. In: The International Conference on Software Engineering pp. 439–449

Meyer AN, Fritz T, Murphy GC, Zimmermann T (2014) Software developers' perceptions of productivity. In: Proceedings of the International Symposium on Foundations of Software Engineering pp. 19–29

Minelli R, Mocci A, Robbes R, Lanza M (2016) Taming the ide with fine-grained interaction data. In: International Conference on Program Comprehension pp. 1–10

Murphy GC, Kersten M, Findlater L (2006) How are java software developers using the eclipse ide? IEEE Softw 23:76–83

Parnin C, Orso A (2011) Are automated debugging techniques actually helping programmers? In: The International Symposium on Software Testing and Analysis, pp. 199–209

Perscheid M, Siegmund B, Taeumel M, Hirschfeld R (2017) Studying the advancement in debugging practice of professional software developers. Software Quality Journal 25(1):83–110

Petrillo F, Guéhéneuc YG, Pimenta M, Freitas CDS, Khomh F (2019) Swarm debugging: The collective intelligence on interactive debugging. Journal of Systems and Software 153:152–174

Piorkowski D, Fleming SD, Scaffidi C, Burnett M, Kwan I, Henley AZ, Macbeth J, Hill C, Horvath A (2015) To fix or to learn? how production bias affects developers' information foraging during debugging. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME) pp. 11–20

Piorkowski DJ, Fleming SD, Kwan I, Burnett MM, Scaffidi C, Bellamy RK, Jordahl J (2013) The whats and hows of programmers' foraging diets. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems pp. 3063–3072

Romero P, Du Boulay B, Cox R, Lutz R, Bryant S (2007) Debugging strategies and tactics in a multi-representation software environment. International Journal of Human-Computer Studies 65(12):992–1009

Sillito J, Murphy GC, Volder KD (2008) Asking and Answering Questions during a Programming Change Task. IEEE Transactions on Software Engineering 34(4):434–451

Singer L, Figueira Filho F, Storey MA (2014) Software engineering at the speed of light: how developers stay current using twitter. In: The International Conference on Software Engineering pp. 211–221

de Souza HA, Chaim ML, Kon F (2016) Spectrum-based software fault localization: A survey of techniques, advances, and challenges. arXiv preprint arXiv:1607.04347

Vans AM, von Mayrhauser A, Somlo G (1999) Program understanding behavior during corrective maintenance of large-scale software. International Journal of Human-Computer Studies 51(1):31–70

Vessey I (1985) Expertise in debugging computer programs: A process analysis. International Journal of Man-Machine Studies 23:459–494

Wang Q, Parnin C, Orso A (2015) Evaluating the usefulness of IR-based fault localization techniques. In: The International Symposium on Software Testing and Analysis pp. 1–11

Wong WE, Gao R, Li Y, Abreu R, Wotawa F (2016) A survey on software fault localization. IEEE Transactions on Software Engineering 42(8):707–740

Xia X, Bao L, Lo D, Li S (2016) "automated debugging considered harmful" considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In: ICSM IEEE International Conference on Software Maintenance, pp. 267–278

Zhang X, Gupta R, Zhang Y (2003) Precise dynamic slicing algorithms. In: The International Conference on Software Engineering pp. 319–329

Zhang X, Gupta N, Gupta R (2006) Pruning dynamic slices with confidence. In: PLDI Conference on Programming Language Design and Implementation 6, pp. 169–180