Data-Oriented Debugging Across Place and Time

Jeffrey Tao
University of Pennsylvania
Philadelphia, PA, USA
jefftao@seas.upenn.edu

Xiaorui Liu
University of Pennsylvania
Philadelphia, PA, USA
xrl@seas.upenn.edu

Ryan Marcus
University of Pennsylvania
Philadelphia, PA, USA
rcmarcus@seas.upenn.edu

Andrew Head
University of Pennsylvania
Philadelphia, PA, USA
head@seas.upenn.edu

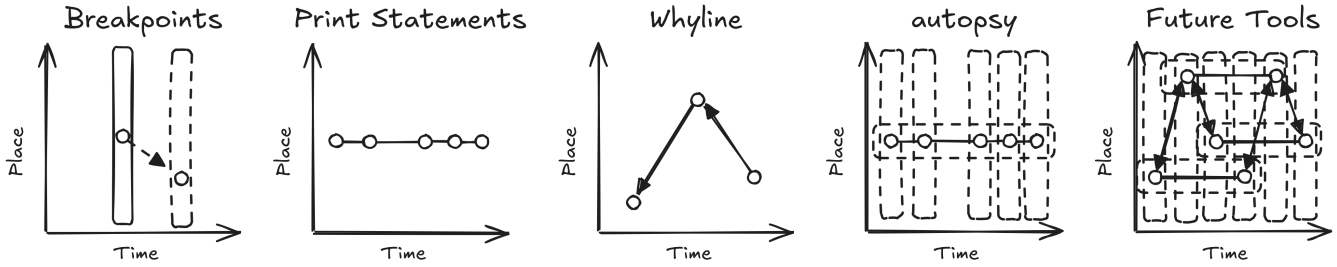


Figure 1: Analyzing debugging tools as mediums for foraging over place and time. We contribute a lens to support analysis of the expressive potential and efficiency of debugging tools. Programmers issue queries via their debugging tools to examine execution data across the axes of place (which encodes particular variables present at a source location) and time. Breakpoints capture variables at every level of the stack at a particular time. Print statements capture specific variables and the order that places are visited during execution. Whyline provides links between points in place-time according to cause and effect. Our transitional debugging tool `autopsy` blends views across place and aggregations across time. We envision that future debugging tools could combine these capabilities.

Abstract

Programmers routinely instrument and run their code with tools such as print statements or breakpoints in order to understand how their programs behave at runtime. When multiple tools are available to programmers, how should we think about their strengths and weaknesses? In this paper, we provide one lens for comparison called *data-oriented debugging*. In this lens, a programmer has data needs and expresses them as queries to their debugging tools. Queries reach into clusters of program data organized by place and time. Tools differ in which of these queries they make easy. We develop this lens through formative interviews with professional programmers, a comparative tool-use study, and an analysis of existing debugging tools. We offer the lens as a reflective aid for tool designers and researchers to assess the capabilities of a debugging tool of interest.

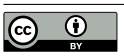
CCS Concepts

• **Human-centered computing → Interactive systems and tools**.

Keywords

interactive debugging, programming tools, mental models

ACM Reference Format:
Jeffrey Tao, Xiaorui Liu, Ryan Marcus, and Andrew Head. 2026. Data-Oriented Debugging Across Place and Time. In *The 39th Annual ACM Symposium on User Interface Software and Technology (UIST '26), November 02–05, 2026, Detroit, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3830398.3830600



This work is licensed under a Creative Commons Attribution 4.0 International License.
UIST '26, Detroit, MI, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2856-3/26/11
https://doi.org/10.1145/3830398.3830600

1 Introduction

Debugging tools (e.g., print statements, breakpoints) help programmers understand a program's behavior by running the program and collecting information about how it ran. Consider a situation in which a student debugs a test failure in a simple arithmetic calculator. When run with `2 + 3 * 4 ^ 2`, the program returns `26` when it should return `50`. The student skims their code, but every processing step of the calculator initially looks reasonable: the tokenizer handles all of the characters, and the parser and evaluator have code to handle all of the operators. What follows is an involved series of steps in collecting and interpreting program data.

This student instruments their code to see if the runtime values offer any clues. They are initially suspicious of the expression parser given its complexity. After printing some of its outputs and seeing that they are all correct, they move on to question the correctness of the expression evaluator. They think the issue may be with the order of operations, and add logs to capture each step of evaluation, including indicative variables like operator, operands, and result. Finally, something unusual appears: `4^2` outputs `8`. This tells the student there is no problem with the order of operations and instead points them to the exponentiation operator whose implementation has a typo, raising the left operand with a '*' instead of '**'.

In situations like this, debugging involves consultation of not just code but data. The existing academic literature on program comprehension tends to foreground the processes involved in reviewing code [16, 23, 39]. While review of program data has been studied as well [3, 27, 29], the picture is less defined, and the variety of reasons

and moves involved in debugging with data has not been developed at a granular level. While it is regularly demonstrated that useful interactive tools can be developed for new questions [2, 17], contexts [14, 40], and navigation mechanisms [6, 37], we believe that development of general-purpose debugging tools requires a clearer picture of what investigation looks like at the level of asking and answering data-oriented questions about program behavior.

To that end, this paper contributes a conceptual lens called **data-oriented debugging** that is intended to support discussion of what a given dynamic debugging tool does and does not support. Under the data-oriented lens, a programmer seeks data to update their mental models. Data is collected by posing queries to debugging tools. Importantly, not all queries are equally easy with different tools. Rather, tools differ in which kinds of queries they make possible or easy. Differences in these queries can partially be explained by their ability to reach across different places and times in a program.

Section 3 describes our lens. Then, the rest of the paper develops it. In Section 4, we conduct formative interviews with professional software engineers to surface data-oriented factors that motivate and influence their attempts to understand programs. In Section 6, we document the queries that arise during a lab study of debugging tool use. Different tools shaped which queries were executed and the ways they were pursued. Finally, in Section 8, we review existing debugging tools under the data-oriented lens to highlight opportunities for tools that provide more flexibility in querying across place and time.

2 Background and Related Work

2.1 User Studies of the Debugging Process

Prior user studies of programmers have described and modeled how people debug, emphasizing a wide array of facets and complexities [1, 3–5, 8, 11, 12, 23, 24, 27, 29, 33, 41, 45]. The central position of data-oriented debugging is that deeply understanding the role of program execution data in debugging is key to designing usable debugging tools. We are not the first to consider data as central to debugging. For example, Spinellis [43], on modern debugging techniques, writes that "data is the lifeblood of debugging." We use this section to detail previous user-centered findings around the role of execution data within debugging.

Many empirical studies highlight the different aspects of execution data's importance. Li and Coblenz [27] argue that execution data is used in two ways: "updating uncertainty", where data is used for informal reasoning, and "updating speculations", where data is used to confirm or disprove a speculation. Previous studies further detail the diverse utility of execution data. While practices like reading code [41] and consulting external resources [27] also occur, the centrality of execution data is demonstrated through its variety of use cases, such as helping establish an initial mental model of the program's functionalities [12, 30], providing scents of possible issues [11, 12, 33], and validating or falsifying hypotheses [1].

In service of concretely describing how programmers utilize execution data in different ways, studies [3, 29] have classified and enumerated the different tactics that users engage in with execution data in the course of debugging, such as *forward and backward reasoning*, *systematic search*, and *tracing execution*. Our study takes these as givens, descending into greater detail about the queries that arise during debugging tasks and the fit of debugging tools to answer them.

Prior research also describes other important factors related to execution data, particularly the barriers associated with information retrieval and presentation. Users expressed the importance of knowledge-avoidance, where they generally approach debugging by attempting to "learn the *least* about the codebase to fix the issue" [27]. Related to capturing the right data, Layman et al. [24] describe how users are often frustrated with data *quality* and *integration* issues when debugging with log files, resulting in an inability to perform desired analyses. Our findings shed light on two underexplored facets adjacent to these issues. First, we present the different granularities of execution data that users are interested in procuring for their analyses, organized along the axes of place and time. Second, we describe the diverse ways by which users navigate *transforming* captured execution data, suggesting opportunities for future tools to find the appropriate representations for addressing these persistent user barriers.

More broadly, several of these studies have proposed different overarching frameworks that characterize debugging. For example, Siegmund et al. [41] propose "scientific debugging", where professional developers generate hypotheses, design experiments, and execute them. In a similar vein, hypothesis-driven debugging [1] posits that debugging is dominated by learning through generating and evaluating individual hypotheses. Our lens, data-oriented debugging, can be seen as clarifying the data needs underlying these processes and how they are realized as queries, calling attention to the diversity of needs and the varied support of debugging tools in fulfilling them.

2.2 Sensemaking

One goal of data-oriented debugging is to give better structure for designing affordances in debugging tools that accelerate the sensemaking process. Sensemaking [36] is a model of how people transform complex information into actionable understanding through iterative cycles of gathering data, interpreting and abstracting them into useful representations, and evaluating those representations using data. Sensemaking consists of two major loops: an information foraging loop [35], in which an analyst seeks out data, and a sensemaking loop where they refine a model that fits the evidence through incremental steps.

Information foraging by programmers is well-established in the literature on cognition in programming [9, 21–23, 34]. The sensemaking loop for debugging has less coverage, but Horvath [13] has multiple works supporting sensemaking in code via annotations as well as print debugging logs. Grigoreanu et al. describe the subprocesses of debugging within the model of sensemaking for end users [11], where participants navigated completion of a debugging task by traversing sensemaking loops of the environment, bug fixing, and common sense/domain. They suggest that debugging tools should surface information that progresses users forward along the sensemaking loop, or support users in recovering after falling backwards in the loop. Their main suggestion was surfacing relevant evidence (such as execution data) and hypotheses in low-cost ways to orient users. Data-oriented debugging shares this goal, and can

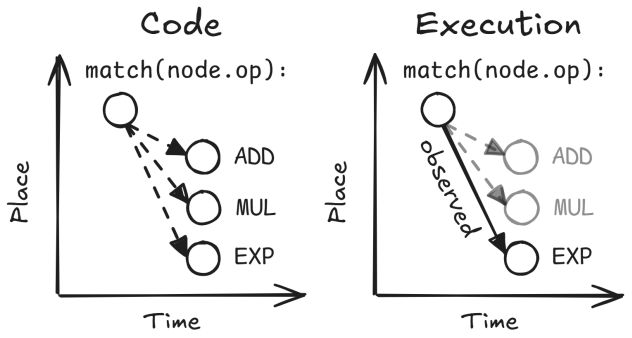


Figure 2: Querying execution data fulfills data needs. The student wonders which operator is evaluated first. **Left**: Control flow in code presents each operator case as a possible path of execution. **Right**: Interpreting the query result makes it clear that the exponentiation operator is evaluated first.

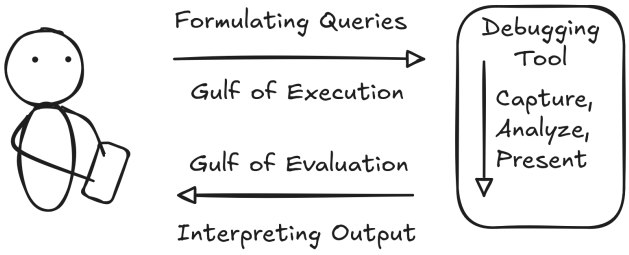


Figure 3: Programmers formulate their data needs as queries and interpret what the results mean about how the code executes. The types of expressible queries and the formats of the query and output are determined by the debugging tool being used.

be seen as an overarching framework for understanding how to structure thinking around supporting sensemaking in debugging tools.

2.3 Questions and Queries

Prior work has investigated the questions that programmers ask when working with code and proposed query languages for interrogating program behavior. In §3, we present a conceptual lens for understanding programmers' interactions with program behavior. This lens bridges these two lines of work, allowing us to investigate matters relevant to the development of debugging tools: How do programmers express their questions about program behavior? How do their tools support them in doing so, and what additional affordances do they need?

Works concerning questions [19, 20, 42] enumerate the high-level intents that occur during comprehension, such as "how is control getting here?" Our lens addresses the subset of these questions that require instrumentation of a running program to answer, and clarifies how programmers query their debugging tools to answer their questions.

There have also been works establishing query languages that extract information from running programs. Lencevicius et al. [25] describe a declarative, constraint-based language for searching over the objects in an object-oriented program at a particular point in time. Goldsmith et al. [10] provide a SQL-style relational query language over events that occur across time within a program trace such as method invocations and object allocations. Our lens provides a way to analyze the coverage of such query languages (and debugging tools more broadly) in answering the data needs that arise during debugging.

3 Data-Oriented Debugging

Our main contribution is a lens on dynamic debugging called data-oriented debugging. Under this lens, debugging tools fulfill programmers' data needs via queries about how a program executed. Tools differ in the ways they gather execution data across place and time in a program. This section describes the basic idea, and later sections lend empirical weight and depth to its components.

3.1 Mental Models and Data Needs

Under the data-oriented debugging lens, the work of code comprehension and review involves the formation of mental models of code [12, 30]. These mental models may be incomplete or incorrect [2, 41]. Data helps the programmer fill out, correct, and gain confidence about their models by comparing them to what actually happens when the program executes [27].

Consider again the student debugging an arithmetic calculator from §1. The student uses data to update their mental model of the program. When examining the expression evaluator, they suspect that the order of operations may be incorrect, so they add a print statement after operator execution. As visualized in Figure 2, the code makes all operators possible, but the execution data shows that exponentiation actually comes first. The student then updates their model, no longer suspecting order of operations as the cause.

When the programmer requires data to advance their comprehension, we say that they have a data need. Data needs extend beyond hypothesis formation and checking [2] to include less-defined questions like "what are all the values that this could be?", "did we ever reach this point?", or "what caused this to happen?" In the example above, the student's suspicion creates a data need for the order in which operators are evaluated.

3.2 Queries

Programmers' access to execution data is mediated by dynamic debugging tools, which gather, process, and present the data. Programmers translate their data needs into queries for their tools, which then act upon them and return results that the programmer interprets (Figure 3).

Queries, broadly, are any requests from the programmer to the tool to compute and present something about the execution data. Different tools are characterized by what data they capture, the queries they enable, how queries are expressed, and how answers are presented. In the example above, the student issues queries by translating their suspicion into the insertion of log statements.

3.3 Place and Time

The space of available data can be conceptualized as spanning place and time. By "place" we mean a pair of a source code location (e.g., the line reading `result = left * right`) and an access path in scope at that location (e.g., `node.op`, which is in scope despite not appearing in the line of code). At a particular source code location, different variables and different fields of a variable are different places. The same access path at different source code locations gives different places. The segment of the place axis corresponding to a line of code consists of places for all of the paths in scope at that line. [1] A particular point in place-time space corresponds to the value of its access path at its source location at its point in time.

While movement through place can correlate with movement through time, as in the case of stepping through successive lines of code, many interesting queries arise when this isn't the case. For instance, a programmer may want to review one place at multiple times, like the student printing every call to the operator evaluation function to see the order of operations. Conversely, they may want to review multiple places at one time, like the student examining operand and result values at the application of the "^".

If a programmer is modeled as an information forager (§2.2), then the patches of exploration in data-oriented debugging are not solely source code patches. They are also data patches. Data patches are regions of the place-time space. They overlap to varying degrees with the data that the programmer currently needs. Importantly, the areas of place-time activated for a program will vary greatly depending on what a programmer is looking for.

3.4 Role of Tool

The design of the debugging tool shapes how programmers navigate place-time. Figure 1 illustrates how different debugging tools capture and relate patches of place-time. For the student in our running example, the print debugger produces a sequence of log messages ordered in time, each anchored to a particular place. This makes comparisons across time easy, for instance making it clear the order in which operations were actually invoked. It also reveals the order that places were reached, if multiple places are instrumented. In comparison, if the student were to use a breakpoint debugger, they would have access to a wide slice of state across place (the debugger's view of in-context variables) at a fixed point in time, making it possible to review a large number of variables related to the analysis. Both tools can facilitate access to the same program data with sufficient effort, though their querying and presentation interfaces influence how well a given analysis is supported.

One role of tools under this lens could be to facilitate programmers in what Spinellis called "finding a needle in a haystack" [43]. The large space of intermediate program states and potential pathways through it is vast. If a tool can help a programmer eliminate large regions of the space, it could make search more tractable. In our example, print lines help the student recognize that the expression parser is correct. This allows them to prune a large segment of space (all parser source code) and time (all steps before the correct parser result).

The next sections lend weight and nuance to this view. §4 describes a formative interview study which finds that programmers explicitly think about gathering and querying execution data, and that their tools currently entangle capture and querying of that data. §6 describes a qualitative tool-use study analyzing the role of execution data in comprehension and the kinds of queries that programmers issue. In §8, we review existing debugging tools and find that they require capture and querying, and that their queries encode the axes of place and time.

4 Formative Interviews

To understand the needs of programmers as they relate to data over the course of debugging tasks, we conducted formative interviews. 12 experienced industry software engineers spoke with us about their experiences debugging systems as part of their own work and projects. Participants were recruited from our personal networks (F1–F12), and interviews were conducted over Zoom. Participants were compensated $50 for their participation.

Similarly to Beller et al. [4], we found that most participants had a strong preference for either print statements (7 participants)[2] or breakpoints (2 participants), with 3 participants saying that they used both depending on the situation. Despite this, when asked about their debugging approach, many participants articulated the same overarching general approach, described below.

Stage 1. Orienting to code. 4 participants preferred to read code first, while 6 used their debugging tools to help them understand execution flow. F1 said that "I might start with the vaguest possible logs if I don't know what's going on...there's first some exploration, especially if I don't know the feature, where I might just add some logs where I think it could hit...I'll put them there because I just need to see generally what's happening."

Stage 2. Use data in checking models. All participants highlighted the formation and checking of mental models. 8 talked about using debugging tools to validate their model or check where the program's execution deviated from their expectations. 6 did so systematically in a process that some referred to as "binary search" or "process of elimination". In F4's words: "when I'm looking at the logs, I can see the execution, that, okay, you reached this function, you went into this branch, you probably should not have gone into this branch. Why did you go to this branch? And then I'll try to figure why it's going there."

Where our observations began to depart from prior work was in participants' observations of the tensions and workarounds arising in collecting and inspecting data.

Managing data volume. 8 participants discussed exercising caution in how they invoked their debugging tools to avoid being inundated with information. F3 said that "you don't want to burn up your log files with unnecessary information...otherwise, you will lose out the meaningful logs." F4 had a strategy of printing both nested fields (for easy scanning in the output) and full objects

[1] Place as a spatial dimension is merely an evocative analogy: place is hierarchical, not linear; place is discrete, not continuous; and any projection of place into space would put disparate places in close proximity. Furthermore, place and time are not fully independent: large regions of the place-time space are empty, as there is no meaningful interpretation of a place at a given time if its source location is not in a scope somewhere on the call stack at that time.

[2] The strong preference for print statements is in part due to our initial recruiting for this study emphasizing print debugging.

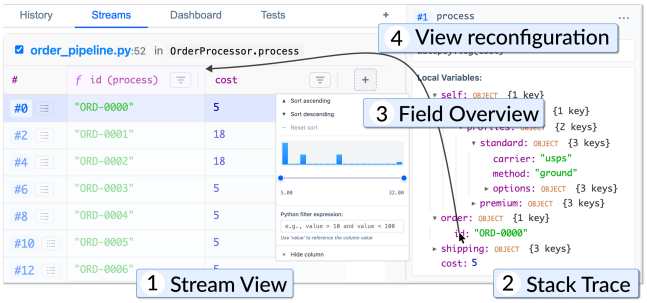


Figure 4: The `autopsy` interface. Captured logs are presented as tables in the Stream View (①). Clicking on any row shows the complete stack trace that was captured simultaneously with the log (②). Logged fields can have their distributions visualized, and can be used to filter and sort logs (③). Fields from the call stack can be added as columns to the table, enabling easier comparisons (④).

(to minimize potential regret of not logging enough). Information overload was not limited to print debugging. F10 said about adding breakpoints, "if it is something that's gonna get called a ton...putting a breakpoint inside that is gonna kind of drive me crazy."

Preparing data for interpretation. Participants had varied strategies and preferences in making output easier to consume. F7 mentioned using a notepad to keep notes while debugging because "there will be a lot of statements, right? ...it looks like, you know, too much information that I can just look and process and get an output." F8 complained that NumPy would sometimes print an entire matrix in scientific notation, which made its contents harder to interpret, and they imagined that having the actual values present (instead of the text representation) would allow them to simply show the matrix again in the right format. F4 coordinated multiple tools, using conditionals to filter down to particular execution contexts, print lines to surface data, and breakpoints to descend into areas where they wanted to take a closer look at state.

These formative interviews raised two insights that guided our next steps. The first was that to better understand the particulars of how programmers use data in debugging, we would need to observe debugging sessions; while participants could describe the shape of their work, they seldom shared particular questions and steps taken to answer them. We conduct these observations in §6. The second was that programmers prematurely filtered their data before they necessarily knew what questions they had; this prompted us to develop a transitional debugging tool that could address the factors that led to premature filtering (§5).

5 Transitional Debugging Tool

We developed a new transitional debugging tool named `autopsy` as a probe intended to make querying behavior more visible. From the results of our formative interviews, we wished to observe programmers directly as they debugged, but were concerned that pre-filtering would restrict the querying behavior that we would be able to observe. As such, we wanted this tool to decouple capture and querying to alleviate the need to pre-filter. This led to two goals that informed the design of `autopsy`:

Make collection easy. Multiple participants (F1, F3, F7, F10) disliked that breakpoints added friction to the data collection process, complaining of needing to constantly advance the debugger through the program's execution. On the other hand, F10 liked that breakpoint debugging reduced the upfront planning of what to collect. We wanted to make it easy to capture all of the needed execution information.

Manage abundant data. We observed a tension between regret of not collecting enough data and difficulty working with too much data (F3, F4), as well as explicit efforts by programmers to optimize the interpretability of outputs (F4, F7, F8). We believed greater support in finding, characterizing, and transforming portions of the collected data would reduce pre-filtering and allow us to see greater query diversity.

5.1 Capabilities

Based on these design goals, we developed `autopsy`, a general-purpose debugging tool designed to support data-oriented debugging, shown in Figure 4. `autopsy` provides more flexible querying in place-time by combining print-style querying across time with breakpoint-style querying across place.

Beyond what data it captures, the role of `autopsy` as a tool is two-fold. First, it makes query expression easier by giving interactions for quickly re-specifying queries over data from already-captured locations in the code without needing to rerun the program. Second, it makes interpretation of larger outputs tractable by providing views of logs separated by call site, allowing interactive filtering of logs, and providing summary visualizations of values contained in logs.

Library. `autopsy` is a library implementing a capture mechanism for execution data reminiscent of `print` debugging: any call to `autopsy.log` takes arbitrary arguments and stores them. In service of *making collection easy*, it additionally captures (1) the line number and code of the expressions passed as arguments, (2) the stack trace including all function names and locations, and (3) the content of all variables in all stack frames.

Web Viewer. After the program is run, the captured data is displayed in a web viewing interface, which can be opened as a standalone webpage or embedded in VS Code as an extension.

Streams. Following the design of Log-it [14], each `autopsy.log` present in the source code is broken out into a separate "stream" table, enabling quick comparison of all of the logs from any one source location. If there are multiple log calls in the source code, each will produce its own stream table. Each row in the stream corresponds to a particular invocation of the logging call. Since all of the logs from the same call site have the same arguments, they are presented as a table. Table columns visualize the distributions of their values and support sorting and filtering. This is useful for getting a sense of the values that passed through a particular point in the code across the whole execution and for cross-filtering other columns, which makes *managing abundant data* more tractable.

Stack traces. Inspired by the broad access to state that a breakpoint debugger gives when stopped on a breakpoint, every log entry is associated with a full stack trace containing the values of all variables at the time that the log was captured. Selecting any log entry in the viewer will show the associated stack trace in a side panel, including code locations for each frame, variable names, and values.

Since `autopsy` captures a full call stack on each log invocation, values from any place in the stack can be added as new columns in a stream. The selected value from the call stack corresponding to each row will be shown as if the original `log` call had it as an argument. This avoids the regret of not capturing relevant places and shortens the feedback loop between wanting additional context and seeing it, but also hides this extra information from view until the programmer asks for it, helping programmers *manage abundant data*.

Full history. Many participants in our formative interviews articulated a systematic process of debugging consisting of comparing execution flow and values with their mental models. To this end, the viewer also presents a full history that interleaves all logging calls, similar to how the corresponding print debugging output would look. In order to answer the question "what happened before/after this?" in a stream, clicking on a particular log in a stream navigates to its corresponding location in the full interleaved history.

5.2 Implementation

The `autopsy` python library uses the python standard library's `inspect` module to capture a full stack trace including variable values. Variable values are captured by pickling, which restricts the set of capturable values to those that are serializable. Naïvely pickling all variables on every new stack trace imposes a noticeable performance penalty, which can be ameliorated in future work by only capturing variables when they change.

6 Tool-Use Study

To better understand the nature of mental model checking, programmer queries, and how tools facilitate them, we conducted a lab study. The study was qualitative, involving observations of 19 participants (P1–P19) as they completed debugging tasks with a number of different debugging tools.

The study focused on three research questions, each of which allowed us to provide greater clarity around essential components of the data-oriented debugging lens:

RQ1. How do programmers integrate execution data into their comprehension process? (Mental Models and Data Needs)
RQ2. What types of queries do programmers issue when using debugging tools? (Queries, Place and Time)
RQ3. How do debugging tools affect what queries programmers express and at what cost? (Queries, Role of Tool)

We selected the tools for this study in accordance with the data-oriented debugging lens: print debugging most easily expresses queries across time, breakpoints across place, and `autopsy` across each. For deeper analysis on the mapping of these tools onto place-time, see §8.

6.1 Method

Sessions provided debugging challenge tasks rated to take about 40 minutes each. Our intent was to observe the kinds of behavior and thinking that happened around program data. Study sessions lasted up to 2 hours and were conducted remotely over Zoom. Each participant was randomly assigned to use one of the two standard debugging tools as well as `autopsy` over the course of two debugging tasks. We captured fine-grained logs of participants' activities throughout the tasks via screen recordings, facilitator notes, and instrumentation in the IDE. Participants were compensated $100 for their participation.

Participants. Participants were recruited from computer science student mailing lists, personal networks, and a LinkedIn post. We screened for software development experience (through internships, full-time work, or research) and Python familiarity. Participants had an average of 5.4 years of programming experience ($\sigma = 3.5$) and rated themselves at 4.0 out of 5 for comfort with Python ($\sigma = 0.8$). 14 participants identified as male and 3 as female. Participants were all university students: 4 undergraduate, 6 masters, 9 doctoral.

Conditions. Study sessions were conducted on participants' own computers via a browser-based version of the VS Code editor, called `code-server`. Each participant used `autopsy` and one of two randomly assigned standard tool conditions:

- Print debugging, shown as an ablation of the `autopsy` interface to alleviate common stdout formatting problems.
- Breakpoints and the debug console, using the features already present in VS Code.

We opted to have participants use both `autopsy` and a standard tool in order to facilitate within-subjects comparisons.

Tasks. We designed two tasks to pose significant obstacles to program comprehension, each consisting of 300–400 lines of Python code. Each task consisted of a multi-stage processing pipeline, with later stages depending on the outputs of previous stages. We added a bug to an earlier step in each task pipeline that manifested in a later step. We included distractor code by adding extra plausible stages that had no impact on the erroneous logic.

One task ("Gateway") was an API gateway where users ran out of API quota too quickly. The gateway consisted of multiple middleware layers, with the quota tracking middleware erroneously deducting quota before the rate limiting middleware ran, sometimes resulting in quota consumption on rejected requests.

The other task ("Sensors") was a scientific processing pipeline with sensors supposedly out of alignment despite input values looking reasonable. A deserialization step erroneously produced zero values for some calibration objects which, when applied in a later step, would zero out the raw sensor reading.

The effects of the bugs were surfaced via a single failing test, but the separation of cause and effect made dynamic debugging necessary.

Before each task, participants were given tutorials on how to use the dynamic debugging tool that they had been assigned to use. Participants were given 40 minutes for each task and were told to try to fix the bug as quickly as possible. We asked participants to think aloud as they worked.

Analysis methodology. The first author reviewed session recordings, took detailed notes about participants' behaviors and think aloud utterances, and conducted a thematic analysis: research questions were used in a top-down analysis which produced a categorization for types of relevant results, then a bottom-up analysis was conducted to code notes within these categories. The coding results were discussed with the other authors, and the second author reviewed the coded notes.

6.2 Results

Throughout this section, unless explicitly stated otherwise, participant and instance counts include at least one instance in every condition and task.

6.2.1 Comprehension process. To answer RQ1, we found that participants incorporated execution data into their comprehension process in a variety of ways, and did so at different stages of comprehension. We believe that the diversity of these observations highlights the important and multifaceted role that execution data plays when comprehending and debugging code.

Exploration of available data. Participants used captured data to understand the actual shape of the available data: 9 participants (17 instances) looked to captured execution data in order to see what variables were even available to aid in their analysis. This was used both in early stages of comprehension to get a sense of the shape of the available variables at a particular place (10 instances) and later on when seeking out more information to contextualize an interesting observation (3 instances). This behavior was most common when using `autopsy` (11 instances).

For example, P2, using `autopsy`, started by saying "I just don't know what I want to log. I feel like I just want to look at everything," and upon logging something and expanding the call stack panel, remarked, "what I really want is this other stuff". We believe that participants' comprehension was assisted not just by *values* of variables, but also *what variables were even available*.

Runtime values as hooks. Similarly, 7 participants (13 instances) looked to captured data in order to find variable names and constants within the code. This was used both to find code that generated a value (9 instances) and to guide writing code for instrumentation or to fix the bug (4 instances). For example, 3 participants doing the sensor task saw many sensor readings with the value 273.15 and searched for it in the code, locating where it occurred as a literal value contained directly in the `apply_calibration` code and subsequently focusing their search there.

Referring to data while reading code. Participants often referred to execution data as they read code (14 participants, 28 instances). Looking at runtime values allowed participants to ground their understanding of the code in concrete examples and validate their expectations about its behavior. For example, P10, upon reading a comment that `self_calibrating` sensors would have null calibration fields before the deserialization step, logged the calibration fields and confirmed this fact. We believe that performing such checks against execution data allowed participants to become more confident in their interpretations of the code.

Data providing information scent. All participants began their tasks by reading at least some of the code, but none fixed the bug without viewing execution data at least once. Execution data gave 9 participants clues about where to look next. Anomalous values and patterns allowed participants to "follow their noses" towards the source of the values, as in information foraging. This scent could sometimes be misleading, such as when P7 logged the `window_start` variable, which gave the start timestamp for the rate limit window, and found that it was negative, prompting them to believe there was a bug in the rate limiter.

Process of elimination. We observed 4 participants explicitly applying a process of elimination in the sensor task. Participants did so by issuing queries across time (§6.2.2) for the output of each stage of the pipeline and noticing that the calibration step caused a distribution shift that persisted across the remaining stages, which led to them investigating the calibration step.

Backwards reasoning. One essential component of the program comprehension process is backwards reasoning [3, 17], which answers questions of the form "why did this happen?" Of the tools that we tested in this study, none directly support this sort of query, and we did not observe participants attempting to use the tools directly to answer these questions. Instead, participants mostly sought answers to these questions by reading code.

6.2.2 Categories of queries. To answer RQ2, we observed that participants used their debugging tools to issue a wide variety of queries that range over axes of place and time. Except where noted, participants performed the following query types using all of the debugging tools, and all participants performed more than one type of query within a given task, which suggests that a single query type may be insufficient for a debugging session.

Across time. The most common type of query by far (17 participants, 58 instances) was observing the full range of values taken on by a particular variable across many instances of a log or breakpoint, or what we refer to as *queries across time*. With print and `autopsy`, they did so by adding log statements and scanning over the resulting logs or table. For breakpoints, they set just one breakpoint after a particular step, then repeatedly pressed continue and watched the output of interest in the variable panel. For example, 12 participants used their debugging tool to view all of the numeric outputs of some stage of the sensor task pipeline, allowing them to see the spread of results. We suspect that the looping nature of these pipeline processing tasks is what led participants to so strongly prefer queries across time.

Following execution forward in time. We were surprised to find that it was comparatively uncommon for participants to follow execution forward for a single request or sensor reading, such as by setting multiple breakpoints, using the step features of the breakpoint debugger, or by adding logs across multiple functions. We had expected this to be more prevalent, especially in the breakpoint condition, as it would allow participants to observe the execution flow of the program, but only 4 participants in the breakpoint condition (P1, P12, P15, P16) performed this style of analysis. Furthermore, no participants conducted this style of analysis using `autopsy`.

Across place. Participants also observed multiple variables simultaneously (6 participants, 9 instances) in order to compare them and draw conclusions about behavior that a single variable would not communicate on its own. In these cases, participants were not making comparisons across time, but instead scanning for individual datapoints with interesting behavior and focusing on the relationship between the variables in those instances. For example, P14 employed an interesting strategy of logging rejected request contexts with the type name of the middleware class instance that most recently processed them, in order to find out which middleware class caused the rejection and direct them to the associated code.

Across both place and time. The most complex analyses conducted by participants combined variables from multiple distinct places in the code and also observed them across iterations (viewing points across both *place* and *time*). Participants made comparisons both across variables and places within an iteration as well as within a particular variable across iterations. 7 participants engaged in such analyses. No participants engaged in such an analysis when using breakpoints, which we hypothesize is due to the breakpoint debugger only showing a single moment in time—conducting such an analysis would be very taxing on working memory, whereas with print and `autopsy` it would be externalized on to the displayed output.

One example of such an analysis was when P6, using `autopsy`, instrumented the middleware execution loop of the API gateway. This assembled a view showing the multiple middleware processing steps for a single request to see if that request was rejected, why it was rejected, and the progression of quota consumption across all requests. This complex view allowed P6 to observe multiple important behaviors of the program at once: that a varying number of middleware were run for each request (since the engine exits early if a middleware rejects the request), that the rate limiting middleware ran after the quota middleware, and that quota was being deducted prior to rate limit rejections.

Single points in place and time. Participants sometimes sought out individual datapoints (7 participants, 12 instances). This took two forms: instrumentation that would trigger exactly once during execution (6 instances), and searching for the first time that a particular predicate was satisfied (6 instances). Exactly-once instances were about aggregate summaries of the execution, such as the number of API requests blocked for any reason across the entire execution. First instances were usually seeking an example to confirm a suspicion about conditions in the program state, such as what a `calibration` object looked like for a `self_calibrating` sensor (P1, P4). This type of query was most common with breakpoints (9 instances).

6.2.3 Expressing queries. To answer RQ3, our findings from this study revealed that tools shaped querying in three ways: they imposed different costs and query formulations for the same needs, they made some data needs difficult or impossible to fulfill, and they sometimes produced outputs that were prone to misinterpretation.

Query transformations. We observed many instances of participants operating on their collected execution data, changing it in some way to better suit their current needs. They conducted these transformations for different reasons—some because they realized their current instrumentation was not answering their questions, others because something in the data spurred them to change their focus. Different debugging tools changed the forms of the base and transformed queries.

The most common transformation was *widening* the view across places (14 participants, 27 instances). This was realized differently and with different costs in each tool: looking at new variables in the variables pane when using breakpoints, dragging to add new columns in `autopsy` tables, or adding extra variables to print logs and rerunning.

In the print and `autopsy` conditions, we observed *wide logs* (7 participants, 14 instances), where participants intentionally captured large objects with more information than necessary, explored the available keys, then focused their attention on specific keys of interest. These wide logs are evidence of print and `autopsy` imposing a cost on widening. This strategy did not apply in the breakpoint case, which already has access across place.

The other common form of modification was *narrowing* captured time (12 participants, 18 instances), where participants applied conditions in order to focus their view of the output by changing breakpoints to be conditional, wrapping log calls in `if` statements, and using `autopsy`'s filtering. That participants did this speaks to costs in interpretation: a narrower view has strictly less information, which is only relevant in making interpretation more tractable.

One particularly interesting form of transformation was *display transformations*, where participants made changes that did not affect the information content of their captured data, but instead the way that it was presented to them. For example, P14, using the print tool, said "maybe I should log this slightly differently" and changed a log of a list to instead loop over the elements of the list and log desired keys from each element to make them easier to compare. One example where the participant was unable to achieve this was when P5 wanted to view the list of results of all processed sensor readings as a table while using the breakpoint debugger. They could not formulate the right expression in the debug console, tried expanding the Variables panel to be very wide but were unable to see the desired key, and finally had to settle for manually expanding individual objects.

Tools make querying difficult. The most urgent gap in tooling that we observed over the course of this study was that participants in all three conditions struggled to formulate their data needs as queries or failed to locate appropriate evidence within their collected data that they suspected should exist (and did exist). They sometimes persevered through tool unfitness at high cost: P12 succeeded, but spent over 20 minutes making several different attempts and performing forward debugging by stepping with breakpoints, in trying to locate the first time a request was rejected for quota exhaustion.

4 participants (P2, P3, P5, P11) at some point wanted to analyze execution data in a way that was hard to achieve within the normal flow of code execution because it required a holistic view of the entire execution. This was inherently at odds with how all three tools work by querying within the normal flow of execution. P5 and P11 both wanted to calculate aggregate statistics over values internal to sensor pipeline steps, which conflicted with the pipeline

control flow which processed one sensor reading at a time. P2 (print) and P3 (`autopsy`) had questions about whether a particular API gateway processing step ever interfered with values from another step, but could not figure out how to formulate this analysis using their debugging tools.

Tools make interpretation difficult. We observed distinct patterns in comprehending execution data that often led participants astray in their comprehension processes. While the precise causes of these interpretation difficulties varied, a common theme was that debugging tool outputs could be confusing or easily misinterpreted.

5 participants displayed an "insensitivity to execution flow" where, in the course of trying to collect or comprehend execution data, they did not notice that an early conditional return statement would restrict the times when their instrumentation would trigger or (in the case of mutating state) that they could not rely on a particular value because it had not yet been set. We hypothesize that this occurred because, of the three debugging tools that participants used, only breakpoints explicitly show control flow, and even then only do so if participants are using step debugging (which few did).

When conditional data capture resulted in no output, participants were often confused about how to interpret the result, as they were unsure if they had specified their condition incorrectly or if the correct answer to their intended query was indeed that such a thing never happened. This problem was especially pronounced for conditional breakpoints, as it was possible for participants to write incorrect expressions (syntactically incorrect, or using variables that didn't exist)—the resulting lack of stopping on a breakpoint was the same in both cases.

6.3 Limitations

Our study tasks consisted of code unfamiliar to participants and were both multi-stage processing pipelines. The bug in each task separated cause and effect. Many naturalistic debugging settings may not be like this: programmers will be more familiar with the code they are debugging, the code may not be a looping pipeline, and bug cause and effect may be closer together.

Our participants were university students with moderate experience programming. Comfort with code, familiarity with design patterns, approaches to debugging, and ability to interpret output could all differ for more or less experienced programmers. These factors, if changed, might change data needs, as well as their expression and interpretation in queries.

7 Refining the Lens

The results of our tool-use study enrich the components of the data-oriented debugging lens and show how applying it can clarify the roles and capabilities of different debugging tools. Here, we examine the implications of our observations for each component.

Data needs. Our answer to RQ1 (§6.2.1) shows that data needs occur throughout the debugging process and that they vary in specificity and purpose. Early interactions with data may be open-ended and help with mental model formation as programmers use execution data to get a sense of how to proceed. As investigation proceeds, needs can become specific and targeted. Programmers' data needs are similar across tools, even if their specific instantiation as queries differs.

Queries & Place and Time. Our answer to RQ2 (§6.2.2) shows that the queries that programmers issue organize cleanly into the axes of place and time. These queries occurred along each combination of the axes: single points in place-time, across place, across time, and across place-time.

Role of Tool. Although we observed similar data needs regardless of tool, our answer to RQ3 (§6.2.3) is that the differences between tools resulted in different costs and therefore different querying behaviors. The phenomenon of wide logs in the print condition speaks to the existence of these costs: we suspect that participants issued wider queries before narrowing because widening across place required an edit-run cycle.

The effect of debugging tools on querying behavior is further evidenced by queries that participants did not issue. P2, P3, P5, and P11 each encountered a situation in which they could not formulate their data need as a query because their tools operated within normal execution flow where they needed an aggregate view. P12 found an inefficient way to query and found the answer they sought, but did so at great cost.

Even when queries were tractable, the tool influenced when and how often certain query types were used. Participants did not issue queries across place and time with breakpoints, perhaps because doing so in breakpoint debugging would require them to track data from earlier times in working memory. Following execution forward in time was more common with breakpoints, where it is as easy as stepping, whereas it was rarer with print statements and absent from the `autopsy` condition because it would require participants to intentionally place multiple logs with the intent of following them. No participants conducted backwards reasoning using the debugging tools because none of them directly support this type of query.

Interpretation costs also influenced participant querying behavior. That participants performed display transformations at all (such as P14's change from logging a list to a loop of logs of its contents and P5's struggle to reformat list output as a table) indicates that tool output interpretation can have a high enough cost to warrant active intervention from the programmer. These interpretation costs are not just localized to the fulfillment of a single data need: insensitivity to execution flow and misinterpretation of empty results could lead to incorrect mental models and to long, fruitless investigations.

8 Prior Work Under the Data-Oriented Lens

We offer the data-oriented debugging lens to enable analysis and critique of tools on the basis of their capture and querying affordances and what types of queries they support. In this section, we demonstrate its potential for this use by putting prior debugging tools under the lens. In the process, the lens helps us explain the power of standard debugging tools and differentiate the kinds of support offered by different classes of research tools.

The maximalist interpretation of data-oriented debugging suggests that we should have tools that cover every type of query a programmer may want to express, make it easy to express these

queries, and present outputs according to the programmer's current needs, though it remains to be seen whether a single tool can cover all of these use cases well.

Print statements, breakpoints, and `autopsy`. Print and breakpoint debugging are both widely used, likely because they can answer broad classes of queries and because with enough effort, it is possible to answer any query with sufficient application of either tool. This also explains their enduring popularity in practice: programmers only need one tool for all of their debugging purposes. This has been well-established in prior work [4, 28], and we saw further evidence of this in both our interview and tool-use studies.

Though we saw participants in our tool-use study issue all sorts of queries using all three of these tools, their particular interfaces make certain queries more natural. As shown in Figure 1, breakpoints capture state across all places at a particular point in time, but seeing state at another time requires stepping, which is effectively a new query across place at the next time step. This makes analysis across time more difficult with breakpoints.

On the other hand, print statements are good for capturing from a particular place across all time, but since they only capture explicitly declared places, there is no way to see other places without knowing other variables or locations to add.

We designed `autopsy` as a synthesis of these two tools, providing easy access to both across-time queries (using the familiar log statement pattern) and across-place queries (via call stacks associated with each log).

There are also queries that are genuinely difficult to express and interpret with these tools, stemming from the fact that they are essentially *capture-and-present* tools that provide little analysis and just present the captured data. We saw in our study that participants did not perform backwards reasoning [3] ("why did this happen?") via the studied tools, preferring instead to read code, as connecting cause and effect using these capture-based tools requires careful interpretation on the part of the programmer.

Extended capture-and-present interfaces. How can tools go beyond print and breakpoint debugging? According to our lens, it would be by querying new combinations of the place-time space, providing new forms of navigation, or reducing the cost of inference and navigation. Here we consider UIST-style debugging tools that can be explained in this way.

First are those that refine the print model [6, 14, 31, 32]. These tools all capture and display slices of state indexed by time. Their value propositions are of several kinds. They capture certain *kinds* of state, such as status of electronics PINs [31], network routers [32], and user interface state [6]. And then, they offer support on the *presentation* layer through mechanisms like domain-fitting visualizations [31, 32], annotations for relating back to code [14], and anchors for particular slices of state for easier comparison [6]. Projection Boxes [26] offer a live view of variable values anchored at code locations, with multiple occurrences across time appearing as a table.

Then are those that extend the breakpoint model. "Omniscient" or "time-travel" debugging [37] presents similar affordances to breakpoint debugging with the crucial extra affordance of being able to *step backwards*; their primary change is not in the shape of patches but in the *navigation* moves allowed. TraceDiff [44] samples the same patches, but from two parallel formulas instead of one. Its innovation is in the presentation layer differencing corresponding patches, and navigation affordances that allow homing in on sites of difference.

When viewed under this lens, it is interesting to consider a recent result from Wang and LaToza [46] which failed to find significant improvements in debugging productivity of programmers using omniscient debuggers over conventional breakpoint debuggers; we conjecture one cause may be that they still query and make accessible the same underlying information patches; adding a navigation direction may not have been enough to add substantial support. More generally, we conjecture that there may be a ceiling to what debugging tools can achieve if they do not allow for more profound ways of flexibly sampling and connecting patches in the place-time space.

Specialized query interfaces. The Whyline [17] is a quintessential debugging tool from the HCI literature. It aims to help programmers answer "why" questions about behavior by allowing them to trace forward and back over data and control flow to causes and consequences of behaviors. It has led to sizable efficiency gains in the lab [18]. Our lens helps point out that it answers only one kind of query — those involving individual points in place and time, with navigation transitions constrained by flow relations. This framing helps clarify that while Whyline seems to be a strong debugging tool for the scenarios envisioned, it necessarily needs to be among other tools to support the full range of actual debugging questions.

Taking this further, we observe of other tools from the literature:

- Hypothesizer [2], which aims to identify behavior indicative of common bugs, provides access to discontinuous samples of points of state across time, with interpretive support in the form of generated explanations of how those points together suggest a bug. It too supports one query type and limited mobility across place and time.
- Scry [7], which aims to help programmers to understand root causes of DOM changes, samples two user-defined points in time and a slice between them for a particular subset of state around selected DOM elements. It too supports this one query type.
- Omnicode [15], which helps novice programmers debug by providing live scatterplot visualizations of numerical variables, provides an aggregation across all time per-variable and exposes just this one query type and presentation.
- Automated fault localization techniques [47] aim to find locations in code likely to cause bugs; these generally sample singular disparate points in place and time with the query defined by the system, not the user.

We draw two conclusions from viewing these tools under this lens. First, the presence of tools for specialized query types motivates the expectation that there are probably yet more specialized queries to derive and support. For instance, perhaps a front-end developer could benefit from a debugging view that shows state subsets at discontiguous time samples when users engage with a certain kind of flaky user interface element. One way of looking at the structural novelty of such an approach is whether it samples new shapes of patches and connects them with navigation in new ways.

An additional conclusion is that while tools for individual queries may be individually useful, it may be difficult to make progress in supporting debugging generally if solely these kinds of tools are developed. If users need to answer a variety of questions throughout debugging, as those in our study did, they may be better served by flexible interfaces that can be bent to answer the kinds of complex queries above among the many others that might arise, using a shared set of flexible query mechanisms.

9 Future Work

Longitudinal study. We are interested in how debugging tools are integrated into programmers' workflows on a longer time scale than was evaluated in this paper. A longitudinal study, perhaps with a participatory design component, would be an opportunity to see how debugging tools support longer-term comprehension and debugging.

Theory of debugging. While data-oriented debugging clarifies how programmers express their data needs as queries, it is not a complete account of all of the information needs and processes that constitute debugging. Programmers must combine querying with reading code, interpreting information, and updating their mental models. Future work could develop a complete theory of how programmers debug that integrates data-oriented debugging with all of these other important components. Such a theory would clarify the relationships between the variety of tools and information sources involved in debugging.

Furthermore, if debugging is analysis of execution data, we could try to transfer powerful ideas from data analysis. A debugging tool might contain explicit exploratory and confirmatory phases of investigation, or treat instrumentation as a sampling design over place-time that fixes which questions the captured data can answer. We observed evidence for both of these in our studies: participants in §6.2.1 progressed from open-ended exploration to hypothesis testing, and participants in both §4 and §6.2.3 had to trade off how much information their instrumentation collected in order to draw the right conclusions.

New debugging tools. From the query types and tooling gaps identified in our tool-use study, combined with our analysis of other debugging tools in the data-oriented debugging lens, we envision new, general debugging tools that fulfill multiple types of queries over place and time. We anticipate that supporting general queries over program execution will require three yet-to-be-developed components: a complete data model, unified querying, and efficient execution.

First, a data model would define the full scope of what queries execute over. The picture of place-time that we have drawn is a simplification of the possible dimensions of execution data. Place as a 1-D axis of (source location, access path) pairs omits important context. Under the current model, a place may have repeated observations under different execution contexts that the programmer might prefer to distinguish, such as at different levels of recursion or from the same method being executed by different objects [38]. It would also be interesting to extend place-time to multiple executions, which would enable comparative analysis, but would also introduce new interpretive challenges when execution diverges.

Next, unified querying would allow a single debugging tool to cover all of the different types of queries. An obvious first extension is to incorporate forward and backward causality tracing (as in the Whyline [17]) into a general-purpose querying tool such as `autopsy`. This would ameliorate the query specification gaps that we observed in §6.2.3 and complete the picture shown in Figure 1. The deeper challenge is to design query and output representations that are general enough to support diverse debugging scenarios without imposing a high burden of knowledge on programmers.

Finally, a general query-based debugging tool must execute efficiently. Analysis across place and time is data-intensive. Time-travel debugging tools tend to avoid the problem of complete execution tracing [37], but in doing so, they lose the ability to perform analysis beyond capture-and-present. We believe that full execution tracing can be made efficient enough to make the performance penalty tolerable for common debugging scenarios, and that doing so is an important component of expanding the types of queries that interactive debugging tools can answer.

10 Conclusion

In this paper, we presented data-oriented debugging, a conceptual lens for understanding the data needs that programmers have and the queries that they issue to fulfill these needs. Debugging tools vary in their support of different queries, which operate on execution data arranged along axes of place and time. We developed this lens via interview and comparative tool-use studies, then applied it to provide perspective on the current landscape of debugging tools and point out strategies and opportunities for future work.

References

[1] Abdulaziz Alaboudi and Thomas D. LaToza. 2021. Rethinking Debugging and Debuggers. In *PLATEAU '21: 12th annual workshop on the intersection of HCI and PL*. doi:10.1184/R1/19852006

[2] Abdulaziz Alaboudi and Thomas D. LaToza. 2023. Hypothesizer: A Hypothesis-Based Debugger to Find and Test Debugging Hypotheses. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, 1–14. doi:10.1145/3586183.3606781

[3] Maryam Arab, Jenny T. Liang, Valentina Hong, and Thomas D. LaToza. 2025. How Developers Choose Debugging Strategies for Challenging Web Application Defects. arXiv:2501.11792 [cs.HC] https://arxiv.org/abs/2501.11792

[4] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering*. Association for Computing Machinery, 572–583. doi:10.1145/3180155.3180175

[5] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. Association for Computing Machinery, 117–128. doi:10.1145/3106237.3106255

[6] Brian Burg, Richard Bailey, Amy J. Ko, and Michael D. Ernst. 2013. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. Association for Computing Machinery, 473–484. doi:10.1145/2501988.2502050

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

[8] Zack Coker, David Gray Widder, Claire Le Goues, Christopher Bogart, and Joshua Sunshine. 2019. A Qualitative Study on Framework Debugging. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 568–579. doi:10.1109/ICSME.2019.00091

[9] Scott D. Fleming, Chris Scaffidi, David Piorkowski, Margaret Burnett, Rachel Bellamy, Joseph Lawrance, and Irwin Kwan. 2013. An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks. *ACM Transactions on Software Engineering and Methodology* 22, 2 (March 2013), 1–41. doi:10.1145/2430545.2430551

[10] Simon F. Goldsmith, Robert O'Callahan, and Alex Aiken. 2005. Relational queries over program traces. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. Association for Computing Machinery, 385–402. doi:10.1145/1094811.1094841

[11] Valentina Grigoreanu, Margaret Burnett, Susan Wiedenbeck, Jill Cao, Kyle Rector, and Irwin Kwan. 2012. End-user debugging strategies: A sensemaking perspective. *ACM Transactions on Computer-Human Interaction* 19, 1, Article 5 (May 2012), 28 pages. doi:10.1145/2147783.2147788

[12] Thomas Hirsch and Birgit Hofer. 2021. What we can learn from how programmers debug their code. In *IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*. 37–40. doi:10.1109/SER-IP52554.2021.00014

[13] Amber Horvath. 2024. *Meta-Information to Support Sensemaking by Developers*. Ph. D. Dissertation. Carnegie Mellon University. doi:10.1184/R1/26880229

[14] Peiling Jiang, Fuling Sun, and Haijun Xia. 2023. Log-it: Supporting Programming with Interactive, Contextual, Structured, and Visual Logs. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 1–16. doi:10.1145/3544548.3581403

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

[16] Amy J. Ko. 2026. *Cooperative Software Development*. https://faculty.washington.edu/ajko/books/cooperative-software-development

[17] Amy J. Ko and Brad A. Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 151–158. doi:10.1145/985692.985712

[18] Amy J. Ko and Brad A. Myers. 2009. Finding causes of program output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 1569–1578. doi:10.1145/1518701.1518942

[19] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. 2019. Live Programming and Software Evolution: Questions During a Programming Change Task. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 30–41. doi:10.1109/ICPC.2019.00017

[20] Thomas D. LaToza and Brad A. Myers. 2010. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. Association for Computing Machinery, 185–194. doi:10.1145/1806799.1806829

[21] Joseph Lawrance, Rachel Bellamy, and Margaret Burnett. 2007. Scents in Programs: Does Information Foraging Theory Apply to Program Maintenance?. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC '07)*. IEEE Computer Society, 15–22. doi:10.1109/VLHCC.2007.48

[22] Joseph Lawrance, Rachel Bellamy, Margaret Burnett, and Kyle Rector. 2008. Can information foraging pick the fix? A field study. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. 57–64. doi:10.1109/VLHCC.2008.4639059

[23] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D. Fleming. 2013. How Programmers Debug, Revisited: An Information Foraging Theory Perspective. *IEEE Transactions on Software Engineering* 39, 2 (Feb. 2013), 197–215. doi:10.1109/TSE.2010.111

[24] Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert Deline, and Gina Venolia. 2013. Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 383–392. doi:10.1109/ESEM.2013.43

[25] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. 1997. Query-based debugging of object-oriented programs. *ACM SIGPLAN Notices* 32, 10 (Oct. 1997), 304–317. doi:10.1145/263700.263752

[26] Sorin Lerner. 2020. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 1–7. doi:10.1145/3313831.3376494

[27] Haolin Li and Michael Coblenz. 2026. A Grounded Theory of Debugging in Professional Software Engineering Practice. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE049 (June 2026), 22 pages. doi:10.1145/3797077

[28] Henry Lieberman. 1997. The Debugging Scandal And What to Do About It. *Commun. ACM* 40 (1997), 26–29.

[29] Amanda Liu and Michael Coblenz. 2023. Debugging Techniques in Professional Programming. In *PLATEAU '23: 13th annual workshop on the intersection of HCI and PL*. doi:10.1184/R1/22277365

[30] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. 2014. On the Comprehension of Program Comprehension. *ACM Transactions on Software Engineering and Methodology* 23, 4 (Sept. 2014), 1–37. doi:10.1145/2622669

[31] William McGrath, Daniel Drew, Jeremy Warner, Majeed Kazemitabaar, Mitchell Karchemsky, David Mellis, and Björn Hartmann. 2017. Bifröst: Visualizing and Checking Behavior of Embedded Systems across Hardware and Software. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. Association for Computing Machinery, 299–310. doi:10.1145/3126594.3126658

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

[33] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. Association for Computing Machinery, 199–209. doi:10.1145/2001420.2001445

[34] David J. Piorkowski, Scott D. Fleming, Irwin Kwan, Margaret M. Burnett, Christopher Scaffidi, Rachel K.E. Bellamy, and Joshua Jordahl. 2013. The whats and hows of programmers' foraging diets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 3063–3072. doi:10.1145/2470654.2466418

[35] Peter Pirolli and Stuart K. Card. 1999. Information foraging. *Psychological Review* 106 (1999), 643–675. doi:10.1037/0033-295x.106.4.643

[36] Peter Pirolli and Stuart K. Card. 2005. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proceedings of the International Conference on Intelligence Analysis*, Vol. 5. 2–4.

[37] Guillaume Pothier, Éric Tanter, and José Piquer. 2007. Scalable omniscient debugging. *ACM SIGPLAN Notices* 42, 10 (2007), 535–552. doi:10.1145/1297105.1297067

[38] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. 2012. Object-centric debugging. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 485–495. doi:10.1109/ICSE.2012.6227167

[39] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How do professional developers comprehend software?. In *2012 34th International Conference on Software Engineering (ICSE)*. 255–265. doi:10.1109/ICSE.2012.6227188

[40] Nischal Shrestha, Titus Barik, and Chris Parnin. 2021. Unravel: A Fluent Code Explorer for Data Wrangling. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21)*. Association for Computing Machinery, 198–207. doi:10.1145/3472749.3474744

[41] Benjamin Siegmund, Michael Perscheid, Marcel Taeumel, and Robert Hirschfeld. 2014. Studying the Advancement in Debugging Practice of Professional Software Developers. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE, 269–274. doi:10.1109/ISSREW.2014.36

[42] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2006. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. Association for Computing Machinery, 23–34. doi:10.1145/1181775.1181779

[43] Diomidis Spinellis. 2018. Modern debugging: the art of finding a needle in a haystack. *Commun. ACM* 61, 11 (2018), 124–134. doi:10.1145/3186278

[44] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Bjorn Hartmann. 2017. TraceDiff: Debugging unexpected code behavior using trace divergences. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 107–115. doi:10.1109/VLHCC.2017.8103457

[45] Iris Vessey. 1985. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies* 23, 5 (1985), 459–494. doi:10.1016/S0020-7373(85)80054-7

[46] Ruochen Wang and Thomas D. LaToza. 2025. How Omniscient Debuggers Impact Debugging Behavior. In *2025 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 57–67. doi:10.1109/VL-HCC65237.2025.00016

[47] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (Aug. 2016), 707–740. doi:10.1109/TSE.2016.2521368