

# Breaking the Bento Box: Accelerating Visual Momentum in Data-flow Analysis

James Yoo

Department of Computer Science  
University of British Columbia  
Vancouver, British Columbia  
yoo@cs.ubc.ca

Gail C. Murphy

Department of Computer Science  
University of British Columbia  
Vancouver, British Columbia  
murphy@cs.ubc.ca

**Abstract**—The bento-box user interface and tool integration paradigm dominates integrated development environments (IDEs). In this paradigm, tools project different information about a system in disjoint panes (boxes) of a window while integrating updates between them as needed. Although popular and functional, the bento-box paradigm has its drawbacks; previous research has shown that expert developers experience disorientation as they work in these environments. In this paper, we explore how context can be preserved for developers within the bento-box paradigm by introducing and experimenting with a tool named ReachHover. This tool supports the answering of common data-flow reachability questions, which have been previously shown to be difficult for developers to answer. To ensure ReachHover supported practical reachability questions of interest to developers, we conducted, and report on, a formative survey of 72 practicing developers about the type and frequency of reachability questions they encounter in their work. We then conducted, and report on, a controlled user study in which 20 practicing developers used ReachHover, finding that participants who used ReachHover answered questions involving visiting multiple files more correctly than those who used standard tooling, and that those developers better maintained context while determining their answers. These findings demonstrate the potential of introducing context-preserving user interfaces for tools within the standard bento-box paradigm of development environments, opening up new avenues for improved tool expression and adoption.

**Index Terms**—Software Evolution, Software Tools, Developer Productivity, Program Comprehension, Empirical Study

## I. INTRODUCTION

Today’s popular integrated software development environments (IDEs) use a bento-box paradigm for displaying information and integrating tools [1]. In this paradigm, separate boxes tiled within a window are used to display various information about software artifacts; tools project information into tiles and coordinate updates between the tiles as appropriate. This bento-box paradigm enables tool-builders to rapidly extend and augment capabilities within an IDE. However, the bento-box paradigm has its drawbacks: tools can be difficult to locate [2], [3] and developers have been shown to display disorientation and thrashing [4].

Alternative approaches have been proposed to mitigate disorientation. DeLine and Rowan, who introduced the bento-box term, suggested enhancing IDEs with a code canvas [1], which coalesces the multiple views of information usually present

in an IDE into an infinite zoomable surface. This approach mostly eschews the compartmentalized interface design found in bento-box styled IDEs. Other approaches are more gradual, focusing on enhancing the bento-box paradigm in IDEs to better support developers. For example, Smith et. al proposed Flower [5], a tool that adds additional annotations [6] through code highlights in a bento-box editor to simplify following control- and data-flow through a program. In this approach, navigation is simplified, but the flow of information must be followed across separate bento-box tiles (or tabs). Flower provided modest improvements to developers on a complex navigation task.

In this paper, we explore a middle ground between these two ends of the spectrum investigating how a *context-preserving user interface* can be incorporated into the bento-box paradigm. We define a context-preserving user-interface as one in which new information needed by a developer is provided in the context of the information they are currently viewing and from which they have requested the new information. We hypothesize that such an interface can decrease disorientation and increase the visual momentum experienced by developers when accessing information that cuts across the bento-box display of information.

A common task that is known to induce disorientation in software developers is the asking and answering of *reachability questions* [7], [8]. Reachability questions are frequently asked by developers; a large-scale study of professional software developers found that they were often investigated more than nine times a day [8]. Reachability questions can be answered by developers by tracing the flow of information through a program; this information flow can be accessed in a bento-box IDE by viewing the results of a control- or data-flow analysis. Unfortunately, in a bento-box interface, working through control- or data-flow analysis results means jumping frequently between editor locations and files, and swapping between views in different panes within the IDE; actions that are known to induce disorientation and low visual momentum [4]. To explore the efficacy of our context-preserving user-interface, we apply it to the problem of asking and answering reachability questions.

We explore a context-preserving user interface for reachability questions through the introduction of a new tool

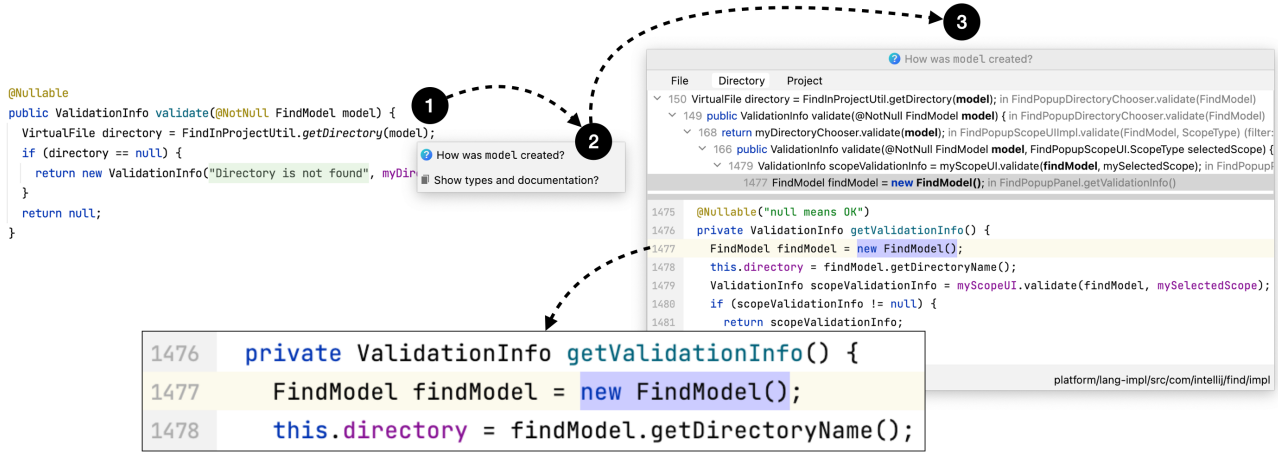


Fig. 1. Using ReachHover to investigate an upstream reachability question. Hovering over the method argument **model** (1) results in the ReachHover tooltip (2). Clicking on the question “How was **model** created?” presents the results of a backward data-flow analysis in the context of the original code under inspection with a preview of relevant code (3). This workflow occurs entirely in the same window and location in which the original investigation is launched.

called ReachHover. At points of code at which a reachability question applies, ReachHover presents applicable reachability questions. For example, hovering on the **model** variable in Figure 1 presents a reachability question about *how* the object associated with the variable is created. In response to selecting the question, ReachHover presents a tree-view of an upstream data-flow analysis. This tree-view is in the context of the code from which the question was asked. For each node in the tree, it is possible to view the code related to a tree node in the bottom pop-up window within the editor. This display allows the developer to remain rooted in the context of the code in which the question was asked. ReachHover is implemented as an open-source plugin for the IntelliJ IDE [9].

To focus ReachHover on reachability questions important to developers, we pose the following research question:

**RQ1** What are the reachability questions that are frequently encountered in practice by developers?

We investigated this research question by conducting a formative study with 72 practicing software developers. We found that developers frequently ask questions regarding *how* data might be formed and modified in a program and thus focused ReachHover on answering these data-flow reachability questions.

We pose two further research questions to evaluate whether ReachHover helps developers answer data-flow reachability questions:

**RQ2** Are developers able to answer reachability questions more correctly with ReachHover?

**RQ3** Does ReachHover make it easier to answer reachability questions?

To investigate these research questions, we performed a controlled experiment in which 20 practicing developers answered two reachability questions on a moderately-sized system that

compared the use of ReachHover and the built-in data-flow analysis support in an IDE. We found that developers more correctly answered data-flow reachability questions involving multiple files with ReachHover than with built-in bento-box style IDE data-flow analysis support. We also found that developers switched between and within files less often with ReachHover, demonstrating that the context-preserving nature of ReachHover is helpful. Developers also indicated a preference for ReachHover over built-in bento-box-style tooling. This evidence supports our hypothesis that providing direct support for asking and answering data-flow reachability questions in the context of the original code under inspection enables developers to answer them more correctly and easily. This tooling shows that context-preserving user interfaces can be added within the bento box paradigm of IDEs, opening up new possibilities for the introduction and adoption of tools.

We make the following contributions in this paper:

- We report on the results of a survey of 72 practicing software developers about the type and frequency of reachability questions. We conducted this survey to ensure our investigations focused on questions of interest to practicing software developers.
- We introduce ReachHover, a novel tool with a context-preserving user interface.
- We demonstrate that ReachHover, which provides a developer direct access to access context-specific questions and *in-situ* provisioning of information, can be built within a common IDE platform, showing that a developer’s user experience can be significantly improved without radical platform or user-interface changes.
- We show that ReachHover’s context-preserving user interface increases visual momentum for developers.

We have made the source code of ReachHover [9], the

dataset of the formative study [10], and the dataset of the user study [11] available for public access.

## II. RELATED WORK

Most widely-used integrated development environments (IDEs) (e.g., IntelliJ IDEA [12], Visual Studio [13], and Eclipse [14]) present a similar interface to developers known as the bento-box style [1]: a large tile for viewing and editing source code, a tile displaying a file-tree, and a number of smaller tiles that enable users to interact with built-in tools. We discuss approaches that have been considered for changing the style of interfaces presented to developers in such environments and discuss tools previously suggested for the specific questions we explore in this paper, namely, reachability questions.

### A. Interfaces for Development Environments

The predominant bento-box style for development environments has allowed developers to work with very large code bases and for new tools to be seamlessly integrated into an environment. However, the bento-box style is also known to have several drawbacks, including being disorienting [1], [4] and causing a loss of visual momentum for developers [4].

Various approaches have been investigated to address these issues. Some approaches call for a major change to the interface of development environments. For example, researchers have considered how virtual reality might be used to ease software engineering tasks [15]. While virtual reality shows promise to help with the live coding of virtual experiences, its use in practice remains in its infancy.

Other approaches have considered breaking out of the bento-box concept via a 2D representation of code. For example, DeLine and Rowan introduced the concept of a code canvas that displayed information about a software project within an infinitely-zoomable 2D single surface [1]. This style of interface was integrated into Visual Studio for the purposes of debugging, known as the Debugger Canvas, which was found valuable by industrial developers for complex debugging scenarios [16]. Various researchers have refined the code canvas approach. For example, Henley et al. report on a refinement of the code canvas approach, called Patchworks, which supports a 3x2 grid of patches in which code fragments can be placed that are embedded in a ribbon of infinite potential patches [17]. As another example, Adeli et al. introduce Synectic, which adds an ability to record annotations between spatially arranged code fragments on a canvas [18]. Experimentation with both Patchworks and Synectic show advantages for spatially-based code displays over existing bento-box interfaces. A disadvantage of these approaches is a need for developers to learn a new approach that is substantially different than the bento-box IDE which is familiar to them. The learning curve associated with these interfaces might explain the lack of adoption despite being built into Visual Studio, a major development platform. Unlike the code canvas approaches (e.g., [17]–[19]), ReachHover does not aim to totally replace the familiar compartmentalization of tooling within an IDE.

Instead, we designed the interface of ReachHover to augment and work within existing IDE interfaces familiar to developers, while presenting information directly within the context of code under investigation.

Other researchers have considered how to augment the existing bento-box style to overcome some of the style’s drawbacks. Sulír et al. describe a taxonomy of annotations that can be made to a source code editor and classify 103 articles (tools) according to that classification [20]. In supporting access to *statically-analyzed control- and data-flow information* from the code *within the editor* for the purposes of *navigation*, our tool ReachHover is most similar to Smith et. al’s Flower tool [5]. Relative to a given variable or method selection in the source code editor, Flower highlights visible items on screen that are related via control- or data-flow to that selection and provides hyper-links to items in the flow that are off-screen. Navigating to an off-screen item causes the same replacement of the information in the editor as is the case for the traditional bento-box style. Our tool, ReachHover, seeks to provide additional context for the developer to reduce potential disorientation by displaying the control- or data-flow results within the source code editor, taking a fluid approach similar to that described by Desmond et al. [21]. ReachHover’s context-preserving user interface also makes it easier for a developer to access the results of the appropriate static-analysis by directly presenting the questions a developer may have about data-flow. ReachHover’s mechanism of presenting questions is inspired by the dialogue-based approach of the Whyline [22], [23].

This paper goes beyond existing work in demonstrating that the integration of both asking and answering direct questions about data-flow within a source code editor is possible and that in doing so, the burden of disorientation can be reduced and visual momentum increased for software developers.

### B. Tools for Reachability Questions

To the best of our knowledge, there are only two tools that have proposed to specifically support reachability questions. The first is REACHER [24], which allows a developer to request “upstream” or “downstream” analysis of control-flow paths from a method selected by a developer. The results of such a developer request are displayed in a separate call-graph visualization displayed within an editor pane. A user study of REACHER found that participants who used REACHER answered control-flow reachability questions more successfully and faster than participants who used standard tools [24]. However, the same study that presented REACHER also showed that participants still spent a significant amount of time foraging for code to verify their answers [24]. This foraging requires context shifts from the call-graph view to other code views. With ReachHover, we consider reachability questions pertaining to the data-flow, rather than control-flow and present an alternative interface that aims to aid developers in maintaining their context while investigating and understanding their code.

The second specific tool supporting reachability questions is Get Me Here [25]. This tool re-frames reachability questions


### III. SURVEY

### A. Design

The reachability questions in the survey were informed by existing literature for reachability questions [8], hard-to-answer questions about code that developers ask [7], and the previous industrial development experience of the authors. The third column in Table I categorizes each question in terms of LaToza and Myers’ framework [8]. A *find*-type question requires developers to look through a number of program paths for a value or point of interest, while a *compare*-type question requires developers to compare two program paths. As seen in this column, both types of reachability questions are present, with the majority of questions being of the *find*-type.

A total of 72 participants took the survey. Participants were recruited via Twitter and email from the authors' professional networks. A wide range of experience was captured, with 56% of participants reporting between 0 to 5 years of professional experience, and 44% reporting 5 or more years of professional experience. The 72 complete responses were collected from a total of 108 started surveys, yielding a completion rate of 67%.

Are you interested in how this parameter is formed? E.g., the method calls and parts of the program that contribute to its creation.



```
public Map<Category, Double> computeTotalsByCategory(Map<ShopItem, Integer> items) {  
    Map<Category, Double> totalsByCategory = new HashMap<>();  
    items.forEach((item, quantity) -> {  
        double cost = item.getCost() * quantity;  
        totalsByCategory.put(item.getCategory(), cost);  
    });  
    return totalsByCategory;  
}
```

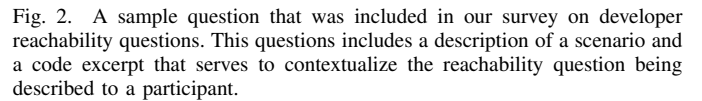


Table I provides a view of the results in terms of the mean scores assigned to each question by participants. The highest-ranking question (Q2) represents situations where a developer adds a feature or refactors code, and then wants to ensure that no defects are introduced as part of the process. This reachability question is a *compare*-type question where developers might compare program traces before and after the addition of changes. The other two *compare*-type questions (i.e., Q7 and Q6) were ranked substantially lower. The second highest-ranked question relates to the control-flow of a program, specifically between two locations in code. This type of question is supported by the existing REACHER tool.

Q8 relates to how the value of data might affect its path through a program (control-flow), or how it is affected by dynamic dispatch (which implementation of a method is called). Q4 is similar to Q5 in that it represents a “downstream” reachability question, but our participants appeared to be less interested in how data might be *accessed* rather than modified. Finally, like Q1, Q3 also asks about the origin of data, but constrains the possible question to the initialization of a class.

In comparison to the previous study on reachability ques-

TABLE I  
SURVEY QUESTIONS AND SCORES ORDERED BY MEAN

Order	Question	Type	Mean ( $\bar{x}$ )	SD ( $\sigma$ )
Q2	Did I introduce any unwanted changes in the new version of this code?	<i>compare</i>	4.12	1.10
Q9	What does the control-flow look like between two locations in code?	<i>find</i>	4.08	1.23
Q1	Where does a value come from, and how is it formed?	<i>find</i>	4.03	1.05
Q5	Given some data, which parts of it are modified downstream?	<i>find</i>	3.76	1.22
Q8	Given some part of a program that depends on a value, which parts of it are executed or reachable?	<i>find</i>	3.59	1.33
Q7	Given two subtypes and their implementations of a common method, how do they handle data differently?	<i>compare</i>	3.51	1.41
Q6	Is deleting what appears to be unused code going to break anything?	<i>compare</i>	3.34	1.53
Q4	Given some data, which parts of it are accessed downstream?	<i>find</i>	3.24	1.26
Q3	How is an instance of this class created/initialized?	<i>find</i>	2.95	1.34

tions by LaToza and Myers [8], our subjects reported asking data-flow related reachability questions at a higher frequency.

#### RQ1 Summary

Developers often encounter reachability questions that are related to the data-flow of a program during their work. This result extends the understanding of reachability questions beyond existing work.

### IV. REACHHOVER

To study a context-preserving user interface for data-flow analysis, we built ReachHover, which provides direct support for both asking and answering reachability questions. In designing direct support for asking reachability questions, we were inspired by the Whyline [22], which augments support for debugging programs directly via “why” and “why not” questions about the program’s observable behaviour. In designing direct support for answering reachability questions, we sought to provide answers in the context of the code in which the question is being asked. By maintaining this context, we believe the developer can be more rooted in the original context of where they began their exploration, and mitigate the disorientation developers may experience when they thrash between multiple views that display information relevant to their task [4].

We describe how a developer invokes a reachability question and how the results are displayed before briefly describing ReachHover’s implementation, including the reachability questions it currently supports.

#### A. Invocation Mechanism

Work in information search and retrieval systems has provided empirical evidence that cursor hovering is a useful proxy for determining a user’s focus or interest in a user interface component (e.g., [26], [27]). We build on these results, choosing to provide a developer access to reachability questions via a hover tooltip. Figure 1 demonstrates this workflow. When a user hovers over an applicable code entity, such as a variable, a tooltip (labelled 2 in the figure) appears.

The ReachHover tooltip consists of two components. The top component presents the option to start the exploration of either a backward or forward reachability question, in the form of a button that reads “How was **x** created?” or “How is **x** modified?” when **x** is a method argument or local variable, respectively. The bottom component enables the developer to access the standard documentation and type information that is usually presented on hover by the IDE in which ReachHover is implemented. Directly presenting applicable reachability questions removes the need for developers to manually translate a question at hand to investigation of a data-flow analysis.

#### B. Result Visualization

ReachHover presents the result of a reachability question directly next to the location where it was invoked (Figure 1). The result window is composed of two main components. The top component is a standard tree-view of the backward or forward data-flow analysis results for the question; these results come directly from the data-flow analysis built into the host IDE. Above the tree-view is a selector that enables developers to scope the results of an analysis to a file, directory, or an entire project. The bottom component is a preview editor that highlights the line of code selected in the tree-view component within its original context (e.g., if the selected element is on line 13 of a file, the preview editor might display lines 10 to 16). The bottom of the preview editor displays the file and the package location of the element that is being previewed. The preview editor provides the same features provided by the main IntelliJ editor, including highlighting, hover inspections, scrolling, and the ability to invoke another ReachHover analysis, but does not allow live-editing. A developer may tune the amount of additional context presented by ReachHover by resizing the window, which might also be moved anywhere within the main editor. ReachHover’s direct presentation of results in a sub-window enables developers to maintain their context within the original file. Since each element in the tree-view is presented within-context in the preview editor, ReachHover preserves the original context of each element. This effectively surfaces data that is available but hidden from users, as IntelliJ does not open a preview window by default.

Direct presentation of results and invocation mechanisms help to make tacit knowledge about data-flow analyses explicit [28].

### C. Reachability Questions Supported

For experimentation purposes, ReachHover currently supports two data-flow related reachability questions. We selected these questions based on the results of the survey.

The most highly rated reachability question in our survey (Table I) involved changes between different versions of programs (Q2). In the LaToza and Myers framework, this is a *compare*-type question. We consider such questions to be of higher difficulty and potentially better answered using a dynamic trace. We return to the question of how to support compare reachability questions in Section VI. As a result, we chose to initially focus on *find*-type questions. The second most highly rated reachability question is a control-flow based question. Since REACHER targets control-flow, we decided to focus on Q1 and Q5, each of which are data-flow related.

### D. Implementation

ReachHover is implemented in the Kotlin programming language [29] for the IntelliJ IDEA Platform. To identify points of interest for invoking a data-flow analysis, ReachHover polls a developer’s cursor location within a file. Code elements detected under a cursor are parsed into an generic unified abstract syntax tree (AST) representation exposed by the IntelliJ IDEA Platform. ReachHover uses the AST representation to validate whether it can support reachability questions on the code element (i.e., a method argument or a local variable declaration).

Once a valid code element is found, and after a developer selects a reachability question, ReachHover dispatches a call to a platform-specific service that provides the correct data-flow slicer for a given programming language. ReachHover then uses the provided slicer to obtain the data-flow slices it requires to present its results to the developer. Unlike the built-in data-flow tooling in IntelliJ IDEA, ReachHover enables developers to quickly re-scope (e.g., to a file, directory or projects as in Figure 1) the results of an analysis in-place without having to re-invoke the tool entirely, meaning, they do not have to re-start the entire analysis workflow from scratch. ReachHover is currently able to be used on systems implemented in Java and Kotlin.

## V. EVALUATION OF REACHHOVER

ReachHover [9] aims to make it easier for developers to answer upstream and downstream reachability questions using data-flow information from a context-preserving user interface. We conducted a controlled user study [11] to investigate two research questions about this aim:

**RQ2** Are developers able to answer reachability questions correctly with ReachHover?

**RQ3** Does ReachHover make it easier to answer reachability questions?

The study compared the use of ReachHover with the built-in data-flow analysis capabilities of the IntelliJ IDEA IDE.

We selected IntelliJ as our control tool as it provides data-flow analysis within a bento-box user interface. Each study participant was asked to complete two tasks where each task involved either a forward and a backward reachability question involving data-flow for a medium-sized software system implemented in a statically-typed language. The order of tasks and order of tools—ReachHover or IntelliJ—was randomized across participants. At the end of the study, participants were asked to report on their experiences using a combination of free-form responses and ranking questions.

### A. Recruitment

20 individuals participated in the study. These participants had a wide range of years of professional software development experience: 25% of participants had up to 2 years of professional software development experience; 50% had two to five years of experience; and the remaining 25% had at least 5 years of experience. 90% of our participants reported that they currently use a statically-typed programming language (e.g., Java, Scala, Go) in their professional work. The remaining 10% reported using a statically-typed language for professional work within the last five years.

### B. Tasks

We used the Apache NetBeans [30] project as a target system for the tasks in the study. NetBeans is an open-source IDE comprising approximately 900,000 lines of Java code. We designed tasks that represented both backward and forward data-flow related reachability questions (i.e., “How was ...” and “How is ...”). The information needed to answer the reachability question for one task was localized to a single file (i.e., *intra*-file); the question for the other task required consideration of many files (i.e., *inter*-file).

The first task, the  $t_{\text{document}}$  task, posed a forward reachability question within a single 1,035 line file. Participants were asked: “How is **findReplaceResult** modified?”, where **findReplaceResult** is a variable that can hold an object containing data related to the find-and-replace operation in the NetBeans IDE. Participants were asked to invoke ReachHover or IntelliJ and report the number of times a null check for the variable **findReplaceResult** appeared along with their locations in code downstream of a particular starting point. Additionally, they were also asked to provide the methods where it might have been used, (i.e., calling methods on it, using it in an expression), and the methods where its value is not used at all (i.e., passthrough methods). Participants were asked these questions to make specific the reachability concept of ‘using a value.’

The second,  $t_{\text{bookmark}}$  task, posed a backward inter-file reachability question. Specifically, participants were asked: “How was **bookmark** created?”, where **bookmark** is a method argument that represents a user-defined bookmark within a file in the NetBeans IDE. Participants were asked to invoke ReachHover or IntelliJ and to provide the number of times a call to the **bookmark** constructor appeared, the unique locations in code where it might have been invoked, the names

of the methods where the value of **bookmark** might have been used before being passed as a method argument, and the locations where **bookmark** or any related objects might have been assigned a value. These specific questions were asked to clarify the reachability concept of creating and assigning a value.

### C. Method

A study session consisted of asking a participant to perform the  $t_{\text{document}}$  and  $t_{\text{bookmark}}$  tasks in a certain order and with a certain tool. Table II outlines the four treatments used, namely  $T_A$  through  $T_D$ . For example,  $T_A$  had a participant first perform the  $t_{\text{bookmark}}$  task with ReachHover followed by  $t_{\text{document}}$  with IntelliJ’s built-in data-flow analysis. The order of tasks and tools were swapped to mitigate the effects of transfer learning.

Participants undertook the study on their own time remotely. Each participant was shown a short video before each task that demonstrated the features of either ReachHover or the built-in data-flow analysis tooling of IntelliJ IDEA. After completing the tasks, participants were asked to report on their experiences with ReachHover and how it compared to tools they would have used otherwise to complete the tasks. We assigned each participant to a treatment randomly, with the only constraint being to have an equal number of participants per treatment as much as possible. We collected usage metrics (e.g., mouse click events, editor interactions) within the IntelliJ IDEA IDE during the task. All logs remained on-device until participants were asked to upload them at the end of the study, at which the following questions were presented:

- 1) “Did you find any difference using ReachHover or IntelliJ for the tasks in this study?”
- 2) “Any other thoughts or comments?”

Ultimately, the number of participants in each treatment was not perfectly balanced, due to factors such as attrition or incorrect logs being uploaded by participants.

TABLE II  
EXPERIMENTAL SETUP FOR REACHHOVER USER STUDY.

Treatment	Task Ordering	Tool	Participants
$T_A$	$t_{\text{bookmark}}$	ReachHover	6
	$t_{\text{document}}$	Built-in data-flow	
$T_B$	$t_{\text{bookmark}}$	Built-in data-flow	5
	$t_{\text{document}}$	ReachHover	
$T_C$	$t_{\text{document}}$	ReachHover	5
	$t_{\text{bookmark}}$	Built-in data-flow	
$T_D$	$t_{\text{document}}$	Built-in data-flow	4
	$t_{\text{bookmark}}$	ReachHover	

### D. Data Analysis

We use a variety of different methods to analyze the data collected from experimental sessions.

*Correctness.* We defined a correctness score ( $S$ ) to quantify how participants performed on each task.

$$S = \text{ANS}_{\text{correct}} - \text{ANS}_{\text{incorrect}} \quad (1)$$

The terms of Equation 1 are as follows:

- $\text{ANS}_{\text{correct}}$  is the number of correct answers; and
- $\text{ANS}_{\text{incorrect}}$  is the number of incorrect answers.

For example, incorrect answers might be lines of code or methods that do not appear in the data-flow trace or lines of code or methods that do appear, but are not related to the value of interest.

*Visual Momentum.* Research has shown that developers can become disoriented when user interfaces lack visual momentum, a qualitative measure of a users’ ability to extract information across changing displays [4]. A display that is low in visual momentum might induce developers to thrash between files to collect information necessary for a task [4]. To examine the visual momentum experienced with the built-in IDE tool and ReachHover, we consider a measure of editor cursor jumps. A cursor jump is recorded when a developer moves their cursor from one location in a file to another location, either within the same file or another open file. Editor cursor jumps in the *main* editor can affect the context (i.e., cursor highlighting, lines of code presented, viewport) within the editor. We measure cursor jump events in the main editor as a coarse-grained proxy for the amount of information-seeking and context-shifting a developer is encountering.

Editor cursor jumps in the *preview* editor are recorded when a developer moves the cursor to a different location within a file open in the preview editor. Unlike main editor cursor jumps, preview editor cursor jumps do not affect the context of the main editor or any open files, and constrain the changing information in the bounds of the preview editor. Preview editor cursor jumps indicate that the developer might be looking at a different line in the same preview editor. We measure cursor jump events in the preview editor as a proxy to help determine whether a developer is actively using ReachHover’s preview editor to obtain information related to their task.

*Experience.* To investigate comments that participants made about experiences with the different tools, a researcher unaffiliated with the development of ReachHover and the study conducted a separate card sort [31] on the qualitative answers to each of the two questions asked at the end of the study.

### E. Results

For tooling to be useful, it must help developers answer questions correctly, thus **RQ2** considers the correctness of participants’ answers with different task and tool combinations. Table III reports the average correctness score for each combination. From this table, we can see participants had a much higher average correctness score for  $t_{\text{bookmark}}$  with ReachHover (4.1) than with the built-in data-flow tools of the IDE (0.8). We observed a significant difference in the correctness scores for this task (Mann-Whitney  $U = 6$ ;  $n = 10$ ;  $p = 0.04$ ;  $\alpha = 0.05$ ) There was less of a difference observed for  $t_{\text{document}}$ .

Figure 3 and Figure 4 provide a more in-depth look into how participants fared with each tool on  $t_{\text{bookmark}}$  and  $t_{\text{document}}$ , respectively. For  $t_{\text{bookmark}}$ , we observe that participants using ReachHover provided a higher median of correct answers



TABLE III

A COMPARISON OF THE AVERAGE CORRECTNESS SCORES FOR EACH TASK AND TOOL COMBINATION.

Tool	Task	Average Correctness Score ( $\bar{S}$ )
ReachHover	$t_{\text{bookmark}}$	4.1
Built-in data-flow	$t_{\text{bookmark}}$	0.8
ReachHover	$t_{\text{document}}$	4.6
Built-in data-flow	$t_{\text{document}}$	4.1

( $\tilde{x} = 4.5$ ) than participants who used the built-in data-flow analysis tooling ( $\tilde{x} = 4$ ). A larger difference was observed in the median number of incorrect answers, with participants using ReachHover providing a smaller number ( $\tilde{x} = 0$ ) of incorrect answers compared to participants who used the built-in data-flow analysis ( $\tilde{x} = 1.5$ ). In  $t_{\text{document}}$ , there was no difference in the number of correct answers provided by participants who used ReachHover ( $\tilde{x} = 6$ ) or the built-in data-flow tooling ( $\tilde{x} = 6$ ). However, we observe that the median number of incorrect responses from participants who used ReachHover ( $\tilde{x} = 0$ ) was smaller than the median number of incorrect responses from participants who used built-in data-flow tooling ( $\tilde{x} = 0.5$ ).

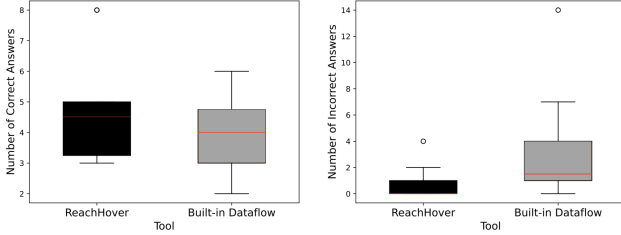


Fig. 3. A comparison between ReachHover and the standard data-flow analysis capabilities of IntelliJ IDEA in terms of correct and incorrect answers given to  $t_{\text{bookmark}}$ .

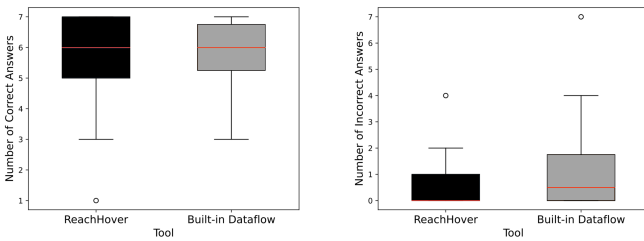


Fig. 4. A comparison between ReachHover and the standard data-flow analysis capabilities of IntelliJ IDEA in terms of correct and incorrect answers given to  $t_{\text{document}}$ .

### RQ2 Summary

ReachHover enabled participants to answer an inter-file data-flow reachability question more accurately compared to participants using the built-in data-analysis support in an IDE.

We also investigate whether ReachHover eases the answering of data-flow reachability questions (**RQ3**). We consider both quantitative data from logs about the behaviour of participants and the qualitative answers they provided to a comparative question about the two tools.

We hypothesize that fewer main editor cursor jumps enables higher visual momentum for participants, making it easier to answer the question at hand. Figure 5 presents the weighted average of main editor cursor jumps recorded per tool in each task of our study. For both tasks, there are fewer weighted average main editor cursor jumps when ReachHover is used than with built-in data-flow analysis support. For  $t_{\text{document}}$ , the intra-file question, this difference is statistically significant (Mann-Whitney  $U = 6$ ;  $n = 10$ ;  $p = 0.001$ ;  $\alpha = 0.05$ ).

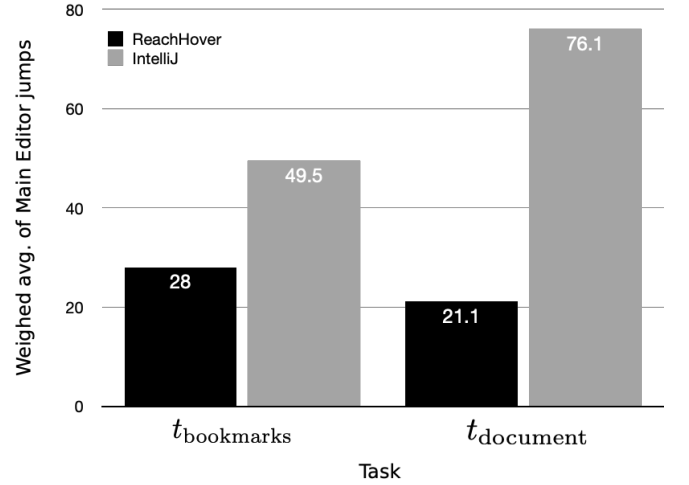


Fig. 5. Weighted averages of the number of main editor cursor jumps for ReachHover and standard built-in dataflow tooling in  $t_{\text{bookmark}}$  and  $t_{\text{document}}$ .

We also measured the number of jumps in the preview editor for each task when ReachHover was used. We observed a weighted average of 58.7 jumps for  $t_{\text{bookmark}}$  and 89.2 jumps for  $t_{\text{document}}$ . The higher number of preview editor jumps with ReachHover suggests that our participants used the preview editor that is part of the tool to locate relevant information.

Whether or not a tool is easy to use also requires considering how developers perceive and experience the tool. Table IV reports a summary of the codings of participant responses to the question about any differences noticed by the participant between the two tools used. Nineteen of the twenty participants responded to the question; each participant is coded as one response in Table IV.

Thirteen of the participants (65%) expressed positive comments towards ReachHover. Five (25%) expressed that ReachHover was easier to use:

*“ReachHover felt less cluttered and made the task easier” - P18*

Three participants remarked that ReachHover was more usable than IntelliJ, both for asking and answering reachability questions:



TABLE IV

CODES IDENTIFIED DURING A CARD SORT OF PARTICIPANT RESPONSES TO THE QUESTION “DID YOU FIND ANY DIFFERENCE USING REACHHOVER OR INTELLIJ FOR THE TASKS IN THE STUDY?”

Code	Description	Count
Prefers IntelliJ	User prefers IntelliJ’s built-in data-flow analysis tooling	2
No Perceived Difference	User did not find a difference between ReachHover and IntelliJ’s built-in data-flow analysis tooling	4
Easier or Simpler	User found ReachHover easier and/or simpler to use than IntelliJ’s data-flow analysis tooling	5
Context-preserving	User found ReachHover preserved more of their working context during a task	5
Usable	User found ReachHover more usable than IntelliJ’s built-in data-flow analysis tooling	3

*“ReachHover was easier to call as it didn’t require navigating menus.” - P9*

*“I find having ReachHover’s separate floating window with a view into where the related line of code is, is more usable than IntelliJ’s data-flow analysis feature.” - P10*

Five of the participant responses (25%) help confirm what we observed from analyzing editor jumps. These participants noted that ReachHover helped to maintain their working context:

*“I preferred that I could look at the matches in the same modal window without affecting the open file” - P19*

*“I felt like I did not lose the context using ReachHover compared to IntelliJ. I felt they showed me the same content, but ReachHover was easier to use (maybe because [sic] I didn’t [sic] lose the context)” - P15*

A smaller number of participants (6 or 30%) did not note any positive difference for ReachHover. Four participants (20%) did not report a perceived difference between ReachHover and IntelliJ, while two (10%) expressed a preference for IntelliJ’s data-flow analysis tooling.

### RQ3 Summary

A lower average of context-shifting main editor interactions was observed when ReachHover was used compared to built-in IDE support, providing evidence for ReachHover’s efficacy as a context-preserving interface. Additionally, the majority of our participants found ReachHover to be easier or simpler to use, more usable, and helpful in preserving and maintaining their context during tasks.

## VI. DISCUSSION

We have demonstrated that the context-preserving user interface provided by ReachHover can make answering certain data-flow reachability questions easier for developers. In this section, we discuss other tasks performed within IDEs that might benefit from context-preserving interfaces and demonstrate how ReachHover may be extended to support other kinds of reachability questions.

### A. Augmenting Tasks with Context-Preserving Interfaces

Many built-in IDE tools present interfaces adhering to the bento-box paradigm, such as a terminal or command-line pane, or a file tree represented by a vertical pane at the side of an IDE window. For these tools, developers generally do not require context beyond what is already presented in the panes (e.g., terminal commands and their output, file names in file trees).

However, for tasks that require developers to consider a large amount of information and context, such as investigating an object hierarchy or finding the usages of a code structure, a context-preserving interface may be helpful in preventing the developer from having to swap and thrash between views.

For example, when a developer is investigating an object hierarchy, they might want to compare an abstract method implementation across subtypes. They might achieve this by manually opening a number of classes in disjoint views inspecting the method of interest. In this case, a developer would still have to swap between files and views while marshalling information across them. With a context-preserving user interface, the need for a developer to actively manage views is diminished; a developer might instead begin their investigation in one subtype’s implementation of the method in the main editor, while an inline view displays another subtype’s implementation of the method, perhaps while highlighting common structures between the two.

Developers perform searches in IDEs to locate the usages of an API, a variable, or a class. In this task, developers often begin from a single structure of interest and explore multiple locations where the structure is referenced, often swapping between different editor locations and views. Without the help of a context-preserving user interface, developers must retain relevant information from the start of their exploration across any number of locations. This might be especially problematic when critical information (e.g., a pre- or post-condition of the structure under investigation) is not recalled by the developer. With a context-preserving user interface, a developer may be able to begin a search, collect relevant information across a number of locations, and take action all from a single location. This mitigates the need for the developer to constantly jump between locations and views, reducing the possibility that they become disoriented.

These examples illustrate that a context-preserving user interface could be used for other tools and integrated within a bento-box IDE paradigm.

### B. Asking and Answering Additional Reachability Questions

ReachHover enables developers to directly ask and answer two specific data-flow reachability questions that invoke creation and modification of data throughout a program. ReachHover could be extended to other questions that developers reported asking in the survey we conducted. For example, to answer Q8 (“Which paths in a program are executed given a certain value?”), we might pose a question such as “When is this path executed?” via a tooltip when a developer hovers over the branches of a conditional statement. In this case, we may exploit run-time to provide a developer with a record of relevant variables and their values when a branch is executed. To answer, Q7 (“Given two subtypes and their implementation of a given method, how do they handle data differently?”) we may pose the question “How does this method modify data for each subtype?” when a developer inspects the top-level method declaration in the supertype. A developer might then be asked to select two subtypes for which the implementations of the method may be compared. The answer to this question might be expressed in the form of a diff of execution traces for each method that highlights similarities and differences in how they mutate or access data. This provides a possible starting point for how additional *compare*-type questions might be supported by ReachHover.

To answer Q6 (“Is deleting what appears to be unused code going to break anything?”), ReachHover might pose the question (“Is deleting this going to break anything?”) when a developer highlights a block of code during a refactoring task. In this case, ReachHover might compute a diff of an execution trace of the relevant subset of the program before and after the block of code is deleted, and present it to the developer should they chose to explore further.

More generally, to answer questions that involve dynamic run-time information, ReachHover could be extended to collect execution traces in the background during the execution of a test suite. To avoid the overhead that might be associated with storing execution traces for multiple test suite executions, ReachHover might execute a test suite on-demand (i.e., when a developer actually invokes ReachHover) and collect only the execution traces relevant to the subset of the program under investigation. In cases where test suite execution is not feasible, dynamic slicing [32] might be a viable alternative.

## VII. THREATS TO VALIDITY

### A. Formative Study

*Construct Validity.* The selection of questions in the survey might not be completely representative of the data-flow related reachability questions that developers encounter in practice. To mitigate this threat, we included reachability questions that were found to be asked by developers in the field (e.g., [8], [33]), in addition to the questions we encountered during our work as software developers. Additionally, the use of a subjective ranking scale in our study for the reachability questions we posed to the participants presents another threat. Individuals might have different perceptions of what it means to ask a question “rarely” or “frequently.”

*Internal Validity.* The internal validity of our findings may be impacted by differing understandings by participants of what a question asked. To mitigate this threat, we provided code excerpts and descriptions to clarify the meaning of questions. Finally, the external validity of the results is impacted by the size of our respondent pool; these respondents were all practicing software developers.

### B. Experimental Study

*External Validity.* Of our 20 participants, a majority (15) reported not being aware of IntelliJ’s or any other IDE’s data-flow analysis feature. Of the five who were aware of this feature, only two reported using these features on a regular basis. This suggests that our results might not be generalizable to populations who might already be familiar with data-flow analysis tooling. Although the reported years of experience was varied among our participants in both the formative study and the experimental study, we are not able to concretely infer their subjective experiences, knowledge, and other personal factors. Consequently, further investigation is required to determine whether our results may be applicable to a general population of software developers.

*Internal Validity.* We conducted the user study entirely remotely. This provided flexibility to our participants in that they were able to participate whenever it was most convenient for them. However, this made it difficult for us to directly observe our participants during the study, and impeded our ability to investigate questions or “think-aloud” comments posed by participants on-the-fly. We introduced another threat by using a system with which our participants may be unfamiliar. Developers might be more successful in completing the tasks we posed in a system that they are familiar with, regardless of whether they used ReachHover or built-in data-flow analysis tooling. We presented two reachability questions within two tasks in our user study. It is likely that developers encounter more types of reachability questions than can be reasonably captured in our user study within a limited period of time. We attempted to minimize this threat by using the results from our survey that investigated the most popular reachability developers ask in practice.

## VIII. SUMMARY

The bento-box user interface paradigm dominates the landscape of integrated development environments [1]. Although popular for compartmentalizing tools within an IDE and the information they present, the bento-box paradigm presents challenges in and is prone to inducing disorientation in developers as they swap and thrash between editor views during program comprehension and navigation tasks [4]. In this paper, we introduced ReachHover, an open-source tool embedded with a context-preserving user interface that developers may use to explore the results of data-flow analyses. Through a survey of 72 practicing software developers, we identified a ranking of how frequently developers encountered a set of nine reachability questions, discovering that many of them related to the data-flow paths in a program. This guided our

implementation of ReachHover, which focuses on enabling developers to ask “How is ...” and “How was ...” questions about the creation or usage of data in a program. In a controlled user study that compared ReachHover and its context-preserving user interface against a bento-box style data-flow analysis tool built into an IDE, we found that ReachHover enabled participants in the study to answer an inter-file data-flow related reachability question more accurately and an intra-file question as accurately as the built-in IDE data-flow analysis support. Participants using ReachHover provided these answers with less context-shifting than with the built-in IDE support, a benefit commented upon by participants. The ReachHover tool demonstrates how an alternative, context-preserving user interface paradigm for integrated development environments might reduce disorientation and accelerate visual momentum for developers without the need for radical changes to the popular bento-box paradigm.

## IX. ACKNOWLEDGEMENTS

We thank the participants of the formative study and the user study presented in this paper. Additionally, we thank Reid Holmes and Caroline Lemieux for their feedback on early drafts of this work and the anonymous reviewers for their detailed feedback and comments. We gratefully acknowledge the support of Canada’s NSERC granting agency for helping to fund part of this work (RGPIN-2022-03139).

## REFERENCES

- [1] R. DeLine and K. Rowan, “Code canvas: zooming towards better development environments,” in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, 2010.
- [2] P. Viriyakattiyaporn and G. C. Murphy, “Challenges in the user interface design of an ide tool recommender,” in *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, 2009.
- [3] L. Findlater, J. McGrenere, and D. Modjeska, “Evaluation of a role-based approach for customizing a complex development environment,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI ’08, 2008.
- [4] B. de Alwis and G. Murphy, “Using visual momentum to explain disorientation in the Eclipse IDE,” in *Visual Languages and Human-Centric Computing (VL/HCC’06)*, 2006, pp. 51–54.
- [5] J. Smith, C. Brown, and E. Murphy-Hill, “Flower: Navigating program flow in the ide,” in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2017.
- [6] M. Sulír, M. Bačíková, S. Chodarev, and J. Porubán, “Visual augmentation of source code editors: A systematic mapping study,” *Journal of Visual Languages Computing*, vol. 49, 2018.
- [7] T. D. LaToza and B. A. Myers, “Hard-to-answer questions about code,” in *Evaluation and Usability of Programming Languages and Tools*, 2010.
- [8] —, “Developers ask reachability questions,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010, p. 185–194.
- [9] J. Yoo and G. Murphy, “ReachHover plugin source code,” <https://doi.org/10.5281/zenodo.7038701>.
- [10] —, “ReachHover formative study dataset,” <https://doi.org/10.5281/zenodo.7006242>.
- [11] —, “ReachHover user study dataset,” <https://doi.org/10.5281/zenodo.7008262>.
- [12] JetBrains s.r.o., “IntelliJ IDEA - the leading Java and Kotlin IDE,” <https://www.jetbrains.com/idea/>, 2023, [Online; accessed 12-Apr-2023].
- [13] Microsoft Corp., “Visual Studio: IDE and code editor for software developers and teams,” <https://visualstudio.microsoft.com>, 2023, [Online; accessed 12-Apr-2023].
- [14] Eclipse Foundation, “Eclipse Desktop & Web IDEs,” <https://www.eclipse.org/ide/>, 2023, [Online; accessed 12-Apr-2023].
- [15] A. Elliott, B. Peiris, and C. Parnin, “Virtual reality in software engineering: Affordances, applications, and challenges,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015.
- [16] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss, “Debugger Canvas: Industrial experience with the Code Bubbles paradigm,” in *Proceedings of the 34th International Conference on Software Engineering*, 2012, p. 1064–1073.
- [17] A. Z. Henley and S. D. Fleming, “The patchworks code editor: Toward faster navigation with less code arranging and fewer navigation mistakes,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI ’14, 2014.
- [18] M. Adeli, N. Nelson, S. Chattopadhyay, H. Coffey, A. Henley, and A. Sarma, “Supporting code comprehension via annotations: Right information at the right time and place,” in *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2020.
- [19] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, “Code bubbles: A working set-based interface for code understanding and maintenance,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010, p. 2503–2512.
- [20] M. Sulír, M. Bačíková, S. Chodarev, and J. Porubán, “Visual augmentation of source code editors: A systematic mapping study,” *Journal of Visual Languages & Computing*, dec 2018.
- [21] M. Desmond, M.-A. Storey, and C. Extton, “Fluid source code views,” in *14th IEEE International Conference on Program Comprehension (ICPC’06)*, 2006.
- [22] A. J. Ko and B. A. Myers, “Debugging reinvented: Asking and answering why and why not questions about program behavior,” in *Proceedings of the 30th International Conference on Software Engineering*, 2008, p. 301–310.
- [23] —, “Finding causes of program output with the Java Whyline,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2009, p. 1569–1578.
- [24] T. D. LaToza and B. A. Myers, “Visualizing call graphs,” in *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2011, pp. 117–124.
- [25] M. Barnett, R. DeLine, A. Lal, and S. Qadeer, “Get Me Here: Using verification tools to answer developer questions,” Microsoft Research, Tech. Rep. MSR-TR-2014-10, February 2014. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/get-me-here-using-verification-tools-to-answer-developer-questions/>
- [26] Q. Guo and E. Agichtein, “Ready to buy or just browsing? detecting web searcher goals from interaction data,” in *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2010, p. 130–137.
- [27] J. Huang, R. W. White, G. Buscher, and K. Wang, “Improving searcher models using mouse cursor activity,” in *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2012, p. 195–204.
- [28] A. Nadeem, “Human-centered approach to static-analysis-driven developer tools,” *Commun. ACM*, vol. 65, no. 3, p. 38–45, feb 2022.
- [29] Kotlin Foundation, “Kotlin Programming Language,” <https://kotlinlang.org>, 2022, [Online; accessed 30-Aug-2022].
- [30] Apache Software Foundation, “Apache Netbeans,” <https://netbeans.apache.org/community/index.html>, 2022, [Online; accessed ‘21-August-2022].
- [31] A. L. Strauss and J. M. Corbin, *Basics of qualitative research: techniques and procedures for developing grounded theory*, 1998.
- [32] H. Agrawal and J. R. Horgan, “Dynamic program slicing,” in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, 1990, p. 246–256.
- [33] T. D. LaToza and B. A. Myers, “Hard-to-answer questions about code,” in *Evaluation and Usability of Programming Languages and Tools*, 2010.