Barriers in Front-End Web Development

David I. Samudio and Thomas D. LaToza
Department of Computer Science
George Mason University, Fairfax, VA, USA
{dgonza10, tlatoza}@gmu.edu

***Abstract*—Developers building web applications constantly face challenges, particularly in working with complex APIs. In response, developers often turn to Stack Overflow, offering a window into the programming barriers developers face. We examined 301 posts on Stack Overflow related to front-end web development and systematically characterized the challenges present in these posts. We found that most challenges reflected not a request for new code or an explanation of an error message but a request about how a specific code snippet might be edited to make its behavior as desired. Many challenges also reflected an underlying need to gather information about how specific code idioms are implemented within a framework or library. We identified 28 barriers developers face in front-end web development. Our findings suggest opportunities for facilitating more effective interactions with complex APIs through new types of programming content and tools that better address barriers in working with code idioms.**

***Index Terms*—debugging, information needs, programming barriers, program comprehension, web development**

I. Introduction

As developers approach programming tasks, they ask questions and seek evidence to answer these questions [1]–[7], sometimes encountering programming *barriers* that block progress, introduce challenges, and consume significant time [8]–[14]. Better understanding these barriers can reveal new opportunities for programming tools and resources that better support developers [8], [15], [16]. The benefits tools offer often stem from an insight into the barriers that make a common activity hard and a solution to overcome these barriers: making debugging easier by supporting following data dependencies backwards from erroneous output to the responsible code [17], making error messages easier to fix by offering actionable error messages [18], or making code easier to understand by reducing barriers to navigating call relationships [19]. To identify barriers, empirical studies have begun to catalog the questions that developers ask and the challenges developers experience [9], [10], [12], [20]. These studies offer important evidence that the challenges programming tools address are real and that successful solutions to them could have meaningful impact on productivity.

While front-end web development is a pervasive software application domain [21], [22], less is known about its programming barriers. Inherent to this domain are client-server and GUI interactions, a style of programming dominated by complex, loosely coupled, and asynchronous behaviors across diverse APIs. To deal with this complexity, developers often turn to crowdsourced question and answer sites, particularly Stack Overflow. For struggling developers, these offer the hope of an answer. Yet these interactions also represent a failure: whatever developers were trying to do, they needed help to do it. In this way, question and answer sites offer a lens to further understand the human factors of these challenges, where studying the developers intent in Stack Overflow posts may offer an opportunity to unveil emerging types of barriers.

In this paper, we investigate the programming barriers in front-end web development. We sampled 301 posts from Stack Overflow and identified the programming activity they reflected and the underlying barriers that led to each post. Our results reveal that the most common challenges developers experienced were in determining how to adapt specific fragments within code snippets to create a desired behavior. This was often hard, as developers were unaware of how to correctly use semantically-related framework APIs to achieve it. Following prior investigations of programming barriers in other domains [8], we found that front-end web development barriers were strongly tied to the type of code developers were struggling with.

To explain these findings, we use the concept of code *idiom* – recurring code fragments across programs with a single underlying semantic role [6], [23]. Idioms consist of two parts: code fragments and their semantic role. For instance, developers working with a *graphical setter* idiom refer to its code fragments (e.g. `x.setAttribute("class", ...)`, or React's `<X className={...}/>`) by its semantic role: updating the appearance of elements in the page's layout. Through this perspective, we offer an explanation of how developers may form invalid assumptions, and how the nature of APIs involved may shape that.

We identified 28 barriers across 11 idioms, detailing the information developers need when working with front-end web frameworks. Our findings offer a checklist for the design of future programming content and tools, enumerating specific challenges they might address to better support developers in web development work.

Our findings offer further evidence for the value of existing theories [8], [16] and tools [24]–[33], we discuss how they might benefit developers by helping them address specific challenges related to complex APIs as knowledge of idioms. Our results also suggest an opportunity for more direct support within the IDE or Stack Overflow for interacting with idioms.

978-1-6654-4214-5/22/$31.00 ©2022 IEEE

II. Prior Work

Both novice and experienced developers face programming learning barriers [8], [20]. One study identified 6 common types of barriers in two categories [8]. *Data barriers* related to the code the developer is writing: design (what), selection (which), use (how), and coordination (when); and, *debugging barriers* related to the environment the developer interacts while writing the code: understanding (why) and information (where). One way to study developers' programming challenges is to formulate needed information developers must obtain in order to complete programming tasks, often conceptualized in the form of a question a developer asks when programming [9].

At the highest level, questions can be organized by developer intent [10], the relationship between code and domain concepts and control and data relationships between methods and classes [34]. Questions also concern the relations between team members and code occurring in task assignments, changes, and builds [14]. Developers ask hard-to-answer questions about code, most commonly including questions about rationale, intent and implementation, debugging, refactoring, and code history [12].

Web developers, regardless of programming expertise, struggle to understand and remember programming related knowledge. Students learning web development often seek help on programming concepts, and spend the most time developing or getting instructions and the least time designing [35]. Experienced developers also struggle with finding the information they need. Web developers often do not recognize the names of concepts they are using as complexity increases, but when given a definition, are ten times as likely to recognize the concepts [36].

To explain how developers manage and reuse knowledge about complex APIs, code idioms have been identified using data mining [6], [23] and crowdsourcing [37]. Idioms differ from code clones in that idioms recur across multiple software projects with varying behavior in each instance, and differ from API usage patterns in their broader generality and in being potentially associated with many API usages [6]. When trying to help developers in their synthesis selection, it is difficult to automatically mine and attribute rationale to idioms [23].

Much work has examined software developers' practices around knowledge repositories, particularly Stack Overflow. Small pools of thousands of developers contribute most posted answers, and do so quickly [38]. Small groups of well-known contributors extensively cover widely-used APIs within a year of their introduction [39], yet many questions remain unanswered because they are apparently of little interest to the community [40]. Several posts involve a widespread misuse of APIs [41], [42]. User-created post tags alone are insufficient to effectively represent post discussions and need automation [43]. How-to questions dominated posts of the most popular languages (*C#, Java, JavaScript*) [1], and most mobile-related posts to be about how-to or discrepancy questions [44]. Using only code snippets extracted from Stack Overflow to answer questions is effective half of the cases [5]. Question types may help automatically classify questions before posting [3]. There is a strong correlation between concepts and questions types, regardless of the programming domain [7]. The most helpful answers highlight important rationale related to the question's code and offer step-by-step solutions [4]. Posts on topics such as data manipulation and layouts were more likely to get answers accepted [2]. Posts also show JavaScript, HTML5 to be confusing and, for CSS, complex and without effective tool support [11].

Web developers' challenges have also been identified from code repositories. One study found that most defects in web codebases were related to DOM manipulation and invalid references [13].

III. Method

To better understand the challenges inherent to front-end web development, we sought to identify the information developers use to ask questions and the insights answers give to solve them. We also sought to understand the nature of the challenges themselves, specifically the barriers from which they resulted and the relationships of these to idioms. We formulated three research questions:

- RQ1: In what contexts do front-end web developers turn to crowdsourced knowledge repositories, and how do they communicate challenges and answers?
- RQ2: What are the common programming barriers that developers face in front-end web development?
- RQ3: What code idioms did answers reference in explaining how to overcome programming barriers in front-end web development?

To answer these research questions, we collected and coded front-end web development posts taken from Stack Overflow. Our data is publicly available. [1]

A. Collecting posts

Each post on Stack Overflow is labeled with one or more tags reflecting its topic. To select posts related to front-end web development in JavaScript, we first examined the 400 most frequent tags on Stack Overflow and selected those related to front-end web technologies (e.g., React, jQuery, Angular, JavaScript). Next, we used the Stack Exchange API to collect metadata on approximately 286,000 posts between May and October 2016 that included the resulting 24 most frequent front-end web technology tags. We obtained close to 50,000 posts after excluding posts that were unanswered, marked as duplicates, or had no upvoted questions. From these, we randomly sampled 1000 posts. Since answers' metadata was not collected, we manually excluded posts that did not have upvotes for the top answer. We stopped once we reached 666 posts (99% confidence level, 5% margin of error). Finally, we manually discarded 365 posts unrelated to front-end web development, including those related to back-end only (90), with

[1] https://github.com/devuxd/public/blob/main/VLHCC_2022.zip

incorrect tags (64), related to configuring NPM or Webpack (48), or related to installing frameworks (26). We then coded 301 posts in total.

B. Coding posts

We used an iterative, inductive process to describe the contexts of the programming challenges reflected in the posts. We analyzed the content of each question and its top voted answer. We assumed that the top voted answer reflected a correct answer and coded the challenge as reflected in the information exchanged in the question and answer rather than attempting to independently diagnose the problem or derive an alternative answer.

From initial open coding of 31 posts, we identified five dimensions: programming activity, explanation strategy, evidence, web technologies, and related idioms. Figure 1 depicts a subset of the information we coded from a post. We refined the coding scheme four times, with two authors meeting independently coding batches of 15 to 20 posts. Finally, we computed the inter-rater reliability, achieving a Cohen's Kappa of a minimum of 0.75 and average 0.92 ± 0.12, indicating excellent agreement [45]. We then coded the corpus of 301 posts using the final coding scheme.

To understand the context of posted questions, we coded *question evidence* describing the sources of information referenced by the question and *programming activity* reflecting the developer's goal when the challenge occurred. Initially, we expected to distinguish activities such as fixing defects from adding behavior. But we found that the distinction between editing code to fix a defect and editing code to add missing behavior was ill-defined, as developers sought to change the behavior of code in both cases and were often themselves unclear if this change required "fixing" some aspect of their implementation or adding code they had not yet realized they needed. We distinguished six activities: implement code from scratch, comprehend code, change the behavior of existing code, resolve a compile time or runtime error, refactoring, and performance optimization.

To understand the context of posted answers, we coded the *answer evidence* describing sources of information referenced in the answer and *explanation strategy* capturing the use of code change explanations or execution simulation in how answers described code.

Across both questions and answers, we identified six forms of *evidence* referenced in posts: code, executable code within a pastebin, official documentation by the software's author or standards body, alternate documentation offered by others including tutorials, program output, and execution state describing intermediate values computed and observed through debugging aids such as console logging or the debugger. We also included another category for infrequently referenced evidence such as diagrams or references to codebases.

While refining our coding scheme, we found that **idioms** presented a better level of abstraction to explain common rationale across **web technologies**, rather than coding used API-terminology or programming concepts separately. Posts often reflected how semantics of frameworks lead to similar challenges. We coded idioms and web technologies based on the code snippets changed or related explanation given by the top answer.

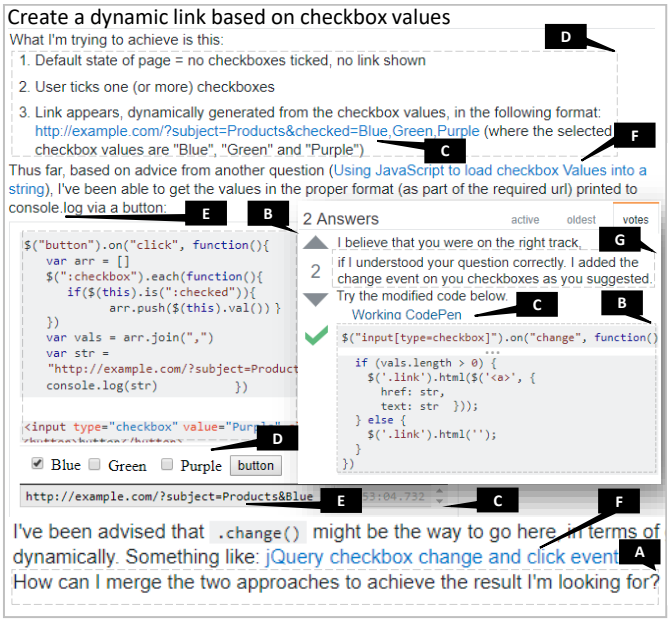

Fig. 1. A Stack Overflow post illustrating the information we coded. (A) Programming Activity: change the behavior of existing code. (B) Source code: question (uses jQuery, HTML) and answer (changes jQuery: graphical query, and event bind target). (C) Pastebin use: question and answer. (D) Browser output: question. (E) Execution state: question. (F) Other evidence: question (another Stack Overflow post). (G) Explanation strategy: code inspection about bound event change.

C. Identifying barriers

To identify programming challenges reflected in posts, we grouped posts by idiom to identify barriers, re-examining both question and answer alongside coded information. We found one or more barriers reflecting the information suggested by the top answer as necessary to resolve the issue. Answers were often anchored around discussion of idioms necessary to explain underlying barriers. These were often illustrated through code fragments. For example, one developer asked:

> "...[code snippet] works fine if i remove "400, function()", when i click the menu-trigger, the menu appears. but with it added, the menu appears then disappears too quickly,..." [2]

And received this answer:

> "Remove the display setting which jQuerys slideToggle() sets, that why the menu gets hidden...:"

```
$(this).toggleClass("nav-expanded").css("display","")
```

The answer indicates a barrier related to a graphical setter idiom. The graphical setter is unidentified, and the answer suggests to remove it.

[2]https://stackoverflow.com/questions/40041515

IV. Results

The results are reported based on our final coding scheme. All percentages are reported as the percentage of Stack Overflow posts unless otherwise noted.

A. Context of challenges (RQ1)

Discussions in posts often focused on changing running code, and some, on fixing non-running code. Questions were rich in additional information surrounding the code, mostly from browser output and execution state. Answers usually involved explanations of code changes, occasionally bringing execution details.

1) Programming Activity: Posted questions varied in what developers wished to learn and how much code had already been written.

Change the behavior of existing code (72%): Posts focused on the currently observed behavior or the behavior to be achieved, often describing specific visual output or the desired content of specific objects. Posts rarely referred to specific lines of code or offered hypotheses about why the desired behavior was not yet realized.

Resolve a compile or runtime error (18%): Posts provided code surrounding the lines where the error had occurred. Of these posts, 91% involved frameworks and libraries not bundled with the browser. Developers rarely presented a specific hypothesis about the cause.

Posts rarely occurred in the context of other programming activities: 4% were related to comprehension tasks to understand existing code or its rationale, 3% concerned refactoring, 3% reflected developers who had not yet written code and were seeking assistance on where to start, and a single post concerned performance optimization.

2) Evidence: To pose a question and offer an answer, developers referenced several types of evidence.

Code snippets (78% questions, 75% answers): The majority of both questions and answers included code snippets, serving to anchor the discussion of the issues. 4% of cases developers reported having the code but not including it because they did not consider it relevant to illustrate the problem. Instead, they referenced documentation or explained the desired behavior.

Executable code within pastebins (26% questions, 33% answers): Some questions and answers contained or linked to pastebins, including Stack Overflow's built in facility for running code snippets (45%), JSFiddle (31%), and others such as CodePen and Plunker (23%).

Browser output (42% questions, 3% answers): Descriptions of browser output were common in questions, anchoring the discussion around the generated output and how the desired output differed. Output was often described abstractly and rarely included screenshots. Output was rarely used in answers, as discussion was instead anchored in code and explanations of code.

Execution state (32% questions, 6% answers): A third of questions referenced specific state generated during program execution, or console logs. 2% of answers included execution state to explain other issues not mentioned in the questions.

Official documentation (9% questions, 23% answers): Answers sometimes made reference to official documentation, including specifications, tutorials, wikis, or blogs created by the software's authors. References offered explanation and justification for suggested code snippets.

Alternate documentation (6% questions, 5% answers): Questions and answers rarely referenced alternate documentation, often to supplement official documentation.

Other forms of evidence (17% questions, 9% answers) included back-end code and other posts. Posts rarely included diagrams or annotated screenshots.

3) Explanation Strategy: Top posted answers often explained the code changes or missing rationale to achieve the desired behavior at different levels of abstraction.

Code change explanations (89%): Posts inspected code to clarify what had changed or how the snippet worked. Posts often explained technology rationale in fragments of the code snippet with technology-specific terminology.

Execution simulation (22%): Posts explained code using either a simulation of a specific code fragment or more abstractly in terms of the API or code rationale that was not present. Only 3% of the posts used execution simulation alone, it was mostly used to support code inspections. Execution data was gathered from console logs, breakpoints, or through the browser output.

4) Web Technologies: The prevalent web technologies were **UI frameworks (70%):** jQuery (23%), Angular (20%), React (13%), among others. This period was rich in questions related to jQuery (standard, UI, and mobile), the transition from Angular 1 to 2, and emerging questions about React. In support of these code snippets, posts also included HTML (29%), client code for back-end communication such as MySQL or RESTful services (16%), or included CSS (8%), among others.

B. Programming barriers and code idioms (RQ2, RQ3)

We identified **28** concrete programming barriers that blocked developers from completing their tasks (Figure 2). In examining these barriers, we found that many were closely tied to **11** idioms in front-end web development, specifically using *callbacks* to manage asynchronous behavior, ways of interacting with rendered *graphical* elements, and *interaction with object structures* during data processing. In the following sections, we report the idioms and the corresponding barriers we identified.

C. Callback Idioms

```
x.on("event",...,function callback(arg){/**/})
```

51% of posts involved challenges with one or more callback code fragments. Callback registration may take several forms, including explicit registration in JavaScript (`x.on(...)` in the example above), through HTML, or through frameworks.

A callback consists of three key parts: a *bind target* identifying an event to subscribe to (`"event"`), a *bind configuration* in the form of parameters controlling the behavior

Fig. 2. Overview of the programming barriers grouped by idiom category. Percentages indicate the fraction of total posts. Posts may included multiple barriers. Each idiom's name is followed by an explanation of developers' usage intent, and its specific barriers. Each barrier is listed with its name, and description as a mapping from the information developers had available to the information they sought to obtain.

of the callback (. . .), and a *callback context* defining which arguments are available and what other additional state is, or is not, available when the callback is invoked (`arg`, local or global variables).

1) Bind Targets (29%): Posts reflected challenges with *unidentified targets (CB1)*, where questions described the circumstance of a target but did not know its name or binding mechanism. Answers explained which HTML elements supported which targets, often targets for DOM changes.

Constrained targets (CB2) occurred when targets were unavailable in specific circumstances, such as specific points within a framework lifecycle.

Confused targets (CB3) occurred when developers selected the wrong target thinking it was another.

2) Callback Contexts (25%): Challenges with *improper scheduling (CB4)* involved differentiating state changes made synchronously from those made asynchronously, identifying the appropriate callback for specific state to be accessible, and locating code appropriately (e.g., properties of an image element being immediately accessible after a synchronous call but the image itself only being loaded afterwards). Answers clarified these issues, sometimes describing how various timeouts, promises, and DOM callbacks were queued and scheduled for execution by the browser's event loop or framework equivalents.

Unidentified state (CB5) involved finding desired state in argument object structures or through functions invoked on arguments, which answers identified.

Missed callbacks (CB6) involved callbacks not executing when desired. Answers often detailed conditions about the state a framework (e.g., React) needed to be in so that a callback would occur and how to check for them.

3) Bind Configurations (23%): Challenges with *incorrect bind parameters (CB7)* were common, where developers sought to identify the appropriate parameters to achieve a desired behavior (e.g., lodash's debounce time units).

Misconfigured frameworks (CB8) involved the initialization or state of the framework generating the callback resulting

in the callback not behaving as desired. Answers explained required framework interactions, sometimes describing scattered changes to interactions with the framework in code and in HTML representations.

D. Graphical Idioms

```
const [r1, r2] = queryInterface(params)
r1.get("prop") && r2.set({aProp: value, ...})
```

42% of posts involved challenges with graphical elements. We refer to graphical elements as a *visual element* in the browser output, *DOM element* in the app's document, *layout element* in HTML or equivalent source files, and *graphical object* in JavaScript. Other software application domains may follow similar representations (e.g., Android UI layer, JavaFX).

A *graphical query* is a pattern expression which selects graphical elements by matching property values (`queryInterface(...)` in the example above). Graphical queries include CSS selectors as well as framework variants, which may generate framework-specific wrappers around layout elements. Questions often described observed or desired visual output. A *graphical getter* is an expression that retrieves information about graphical elements, often in the form of property values (`r1.get(...)`). A *graphical setter* is an expression that mutates graphical elements to achieve a new graphical state, such as by setting one or more properties or by adding or removing elements (`r2.set(...)`).

Graphical elements have visual properties such as style, position, and animation as well as non-graphical properties such as custom data or bound callbacks. Questions frequently referenced specific browser output (68%) and occasionally referenced execution state (12%). Fixes were rarely (4%) confined to CSS snippets alone.

1) Graphical Setters (37%): Challenges with *unidentified setters (GB1)* occurred when developers had visual behavior they wished to achieve but did not know the setter or setters required or the appropriate parameters. Questions often made analogies to existing visual elements to describe the desired behavior. Answers often provided the necessary setter(s).

Unobservable setters (GB2) occurred when, developers wished to understand why visual changes seemed not take place. Answers often explained differences among similar methods offered by a framework (e.g., jQuery's `attr()`, `prop()`, `data()`).

Indirect setters (GB3) involved graphical changes that occurred through properties inherited from parent elements or due to layout errors with other elements (e.g., an element occluded by a mispositioned element). Answers often explained inheritance rules and suggested property changes.

Conflicted setters (GB4) occurred when, despite the query being correctly formed, overlapping setters led to undesired visual behavior, often due to undesired sequencing. Answers suggested new setters or suggested the correct way to sequence them depending on the events at hand.

2) Graphical Queries (21%): Challenges with *Incomplete queries (GB5)* occurred when different selectors were required to select the desired elements, which answers often suggested. This frequently involved identifying property values that desired elements shared and which could be selected.

Outdated queries (GB6) occurred when the set of elements matching a query unexpectedly varied over time. Answers often described alternative approaches to using a query instead of adding logic to new elements when they were created.

Overwritten queries (GB7) occurred when more than one query unexpectedly mutated the same element. Answers identified which queries were retrieving the same element and sometimes suggested changes to the query expression to avoid selecting the element. Underlying these were challenges associating visual elements with code fragments.

3) Graphical Getters (8%): Posts reflected challenges with *unidentified getters (GB8)* when selecting or identifying the appropriate getter to retrieve specific information, particularly when the available getters, identifiers, or behavior varied between frameworks. Answers identified missed or misused properties, and explained or contrasted framework API rationales. Of challenges with getters, 30% involved invalid references.

E. Object-Interaction Idioms

40% of the posts involved challenges with object-interaction idioms. Developers often illustrated challenges with execution state from the console, or abstract descriptions of object structures and properties.

1) Valid References (21%): Identifiers in JavaScript may reference an object, object property, variable, or function and may refer to definitions in user code or in framework code. References may occur directly as code or embedded in a string, which may then be parsed in code located elsewhere.

Inactionable reference errors (OB1) occurred when developers received a parse or runtime error message lacking sufficient detail for the developer to understand how to resolve the issue (e.g., "cannot determine value of undefined"). Frequent causes included misuse of framework-specific wrappers around HTML attributes, incorrect use of naming conventions, and typographical mistakes. Causes sometimes involved correctly referencing identifiers in HTML, React's JSX, or other template representations, references to objects defined only later in the execution, or accesses to elements that had not yet been appended to the DOM. Answers clarified these causes, identifying how the framework imposed constraints on fragments of user code.

Silent invalid references (OB2) forced developers to manually locate the source of the defect. For example, a reference to an object property in JavaScript that is not present (e.g. `undefined`, `null`) determines the execution of code without the need to throw an exception (e.g., optional chains such as `objRef?.doMethod()` or conditional expressions such as `objRef.prop? do(): otherDo()`). Answers often indicated code depending on these references not being executed and provided changes to do so. Rarely, references were invalid due to being overwritten.

2) Collections and Formats (20%): A fifth of the posts involved challenges with the properties on objects and expectations imposed about properties by frameworks.

Unidentified iteration constructs (OB3) involved challenges related to collections, often to differentiate among similar methods offered by a technology that were applicable to different types of objects or that exposed a different collection, such as using JavaScript to iterate based on keys or indexes, `for in`, and values, `for of`; jQuery's `each()` and `map()`; or Angular's `ng-repeat`. Answers differentiated constructs based on the object's properties, whether changes to element membership in the collection would be reflected in the same object or a new one, and if the object was declared in user code or obtained from a framework.

Occluded modifications (OB4) involved a collection which was unexpectedly mutated between iterations. Answers highlighted unexpected framework calls that occurred between iterations or unsatisfied requirements to invoke specific framework functions or methods.

Confused formatting (OB5) occurred when converting a JavaScript Object to DOM object or converting a DOM object's data into an object in the format of a framework's data model (e.g., Angular, React). Answers explained the relationship between properties in alternative representations.

3) Back-End Requests (16%): Back-end request challenges concerned sending and receiving data from a back-end server.

Misconfigured requests (OB6) involved selecting the desired yet unknown correct parameters for specific services (e.g, routes for RESTful services) or using timers to correctly schedule requests.

Unclear transmissions (OB7) involved data received by the server differing from what was sent. For example, a NoSQL store omitted storing properties with `undefined` or `null`, requiring them instead to be `0`.

Mishandled responses (OB8) involved listening for and sequencing responses from the server. Answers provided code and described the appropriate mechanism to listen for responses (e.g., promises or `async` functions) and to correctly sequence response listeners.

4) Method Chains (8%): A *method chain* is a sequence of method calls where each method returns an object that is the receiver of the subsequent method call (e.g., `[...array].fill(0).map(f)`). A *method cascade* is a form of method chain where each method returns the *same* object on which it was invoked. Method cascades may themselves be chained through a *bridge* method which returns a different object, enabling a sequence of actions to be invoked on multiple related objects (example above).

Incomplete sequences (OB9) occurred when the sequence produced no error but required one or more additional methods to achieve the desired behavior. Answers often related specific methods to documentation and identified desired methods.

Incorrect sequences (OB10) occurred when a method was misplaced or misused. Answers explained ordering constraints on actions within cascades or differentiated closely related methods (e.g., D3's `enter()`, `select()`, and `datum()`), or identified methods as bridge ones (e.g. jQuery's `find()`).

In an *overridden effect (OB11)*, unexpected effects clobbered intended effects. Answers identified overlaps in changes to object properties, often correctly rearranging the chain. These changes involved replacing valid references with others.

5) Scope Contexts (8%): In JavaScript, the `this` keyword refers to the current lexical scope's execution context. Posted questions often reflected on *unclear scopes (OB12)*, a lack of awareness that `this` varies within enclosing scopes, standard and arrow functions, or passing it as a parameter. Answers explained the use of `this` in the specific context or when it changed within scopes.

V. Limitations and Threats to Validity

Our study has several important limitations and potential threats to validity. Rather than directly observe the activities of developers through a lab study or field observations, which are rich in details, we observed their activities indirectly through their interactions with Stack Overflow, which offers the potential for a larger sample.

However, indirect studies, including ours, also have important limitations. An important threat to validity is the potential bias introduced by coding posts manually. We reduced possible reviewer bias by coding a subset of posts with two independent raters and discarding codes not within Cohen's Kappa perfect agreement threshold.

Our study was limited to an analysis of posts on Stack Overflow. While broadly used, it may be that only specific types of challenges are posted to Stack Overflow (e.g, popular frameworks, "medium" difficulty questions). By considering only answered questions, we limited our sample to those that were able to attract the attention of Stack Overflow users and elicit an answer. It may be that there are contexts in which developers are unable to formulate a satisfactory answer, offering an opportunity for future study of these barriers.

By focusing on what the top answer attributed the challenge to be, we may have disregarded better answers. We aimed to reduce the bias that might occur in attributing causes ourselves. Rather than focus on the most popular and widely read posts, we investigated a representative sample taken from a six-month period available at the time. However, the web technologies present in our sample may not be representative of current trends. Interestingly, since 2016, the most significant change in web technology popularity has been the rise of React.js and the decline of jQuery[3]. To partially mitigate this, we analyzed the challenges at the idiom level, which may be less impacted by changes in APIs. We captured emergent barriers in Stack Overflow posts to partially mitigate already reported findings, or challenges already addressed by tools.

VI. Discussion and Future Work

Table I summarizes our design recommendations as information needs based on the programming barriers we identified

[3]https://insights.stackoverflow.com/trends?tags=jquery%2Creactjs %2Cangular%2Ccss%2Chtml%2Cjavascript%2Cangularjs

TABLE I
INFORMATION NEEDS FOR FRONT-END WEB DEVELOPMENT

Information Need	Programming Barriers	Technique	Opportunities for future tools
Debugging Barriers			
Map from code fragment to corresponding runtime state	silent invalid reference, unclear scope	Live programming	Show live output with live execution state and linkages from code to runtime state.
Map from visual elements or cues to corresponding code	all graphical barriers, inactionable reference error	WhyLine	Support broad range of outputs and platforms.
Identify order of callbacks and framework state changes	improper scheduling, missed callbacks	Software visualization	Depict function calls, framework state changes, and callbacks.
Identify unexpected or overwritten effects in execution trace from code fragments	all graphical setter barriers, overridden effect, outdated & overwritten query, occluded modification	Software visualization	Show effects related a code fragment. Identify unexpected effects related to state change history.
Data Barriers			
Generate code fragment from intent and runtime context	all collection & format idioms, unidentified target & state, mishandled response, incomplete sequence	Program synthesis	Suggest code completions from intent based on runtime context accompanied with rationale.
Generate code fragment from visual elements or properties	all graphical barriers	Program synthesis	Generate code from intent as well as a specific visual queries.
Edit code fragment to achieve specific intent	all graphical setter barriers, incorrect sequence & bind parameters, misconfigured request & framework, constrained & confused target	Automatic software repair	Generate code edits only to specific fragment based on desired intent or effects rather than only on existing unit tests.

(Figure 2). Our results suggest the opportunity for programming tools to more effectively help developers work with complex APIs. Idioms might help Stack Overflow better capture execution information, and teaching idioms before frameworks might improve developers' awareness of their code and its execution. Knowledge repositories and tools may follow these recommendations by offering idiom-centric features such as idiomatic documentation, views, and transformations.

A. Idiomatic Knowledge

Idioms have a direct connection with robust API knowledge [16]: their semantic roles act as rationale which recur among APIs, and their code fragments act as code patterns. An idiom groups similar API usage patterns, facilitating communication of idioms to developers. Previous studies have shown that developers struggle recognizing the concepts they commonly use [36], and remembering vast amounts of details as they determine the behavior of their code [46].

The search for ever better framework designs may itself be part of the problem in forcing developers to constantly learn new APIs. To facilitate obtaining this knowledge, idioms may be introduced when developers first learn to program and included in the documentation and other repositories they visit.

Our findings suggest important opportunities for improving learning resources for novice web developers. We found a relatively small number —11— of idioms that are frequently associated with challenges in front-end web development. While most web developers quickly gain at least a passing familiarity with these concepts, there are many corner cases of how these idioms work, which together were responsible for the entirety of the barriers we observed. By learning the nature of the programming barriers we identified, novice web developers might become more aware of the potential causes of the issues they face, formulate better hypotheses, and more effectively pursue strategies to test these hypotheses. It may be possible to develop content for courses that can communicate idiomatic knowledge in ways that generalize across the many varying frameworks that exist.

In facing challenges with data barriers, developers sometimes drew on their existing expectations of what an alternative framework offered. A taxonomy of idioms may offer future API templates to follow, so their fragments are familiar to developers. Debugging barriers might, in some cases, be caused by variations in effects, timing, or other behavior between equivalent functionality in different frameworks. Given the rapid evolution and change of web development technologies, it is important to support developers in documenting migrations between frameworks and versions either by maintaining consistency or offering clear mechanisms to map from one to another using idioms.

B. Idiomatic Views

Idiomatic knowledge can be provided by tools as developers need help understanding the behavior of complex APIs. To illustrate this, consider a hypothetical scenario depicted in Figure 3. Developers asked questions through reference to idioms, describing specific code (78%), execution state (32%), and program output (42%) responsible for observed behavior. Tools might better support developers in mapping from code to state to output in accordance to idiomatic views, where behavior can be tracked across representations. Tools might better enable an idiomatic view of code in a number of ways.

Live programming tools help developers rapidly see output changes following code changes, but less frequently show information about execution state. For example, our results

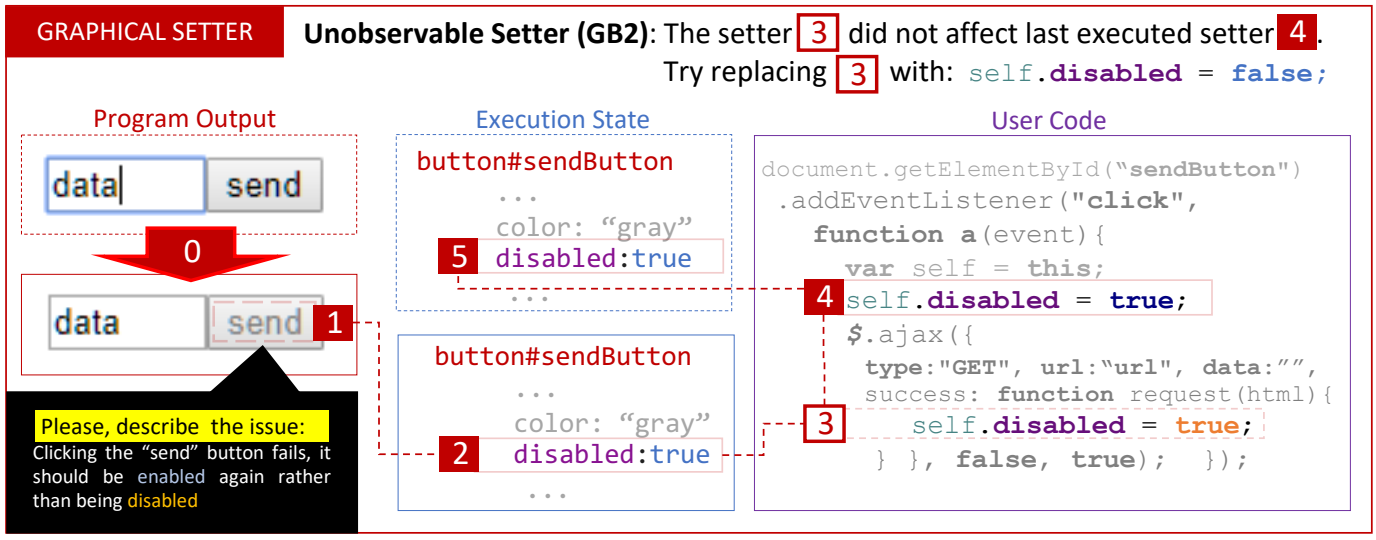

Fig. 3. Idiomatic views maintain unrestricted navigation across representations: from program output (left, 1) to execution state (center, 2, 5) to code (right, 3, 4). Adriana notices an unexpected behavior after interacting with an app (0). An idiomatic diagnostic system identifies a relevant graphical barrier and asks the developer for the desired behavior based on the visual selection (1). It then shows how to overcome the barrier, GB2. A step-by-step explanation of relevant APIs summarizes the current behavior: the style property (2) is not mutating in-between graphical setters (3, 4), as logged by the system (2, 5). A code change is suggested to achieve the desired behavior.

suggests the value of tool support for highlighting the execution context related to `this` references.

Tools such as the WhyLine [17] enable developers to track output back to code, but may need to be extended to apply to the full range of outputs that caused challenges.

Tools such as Sahand [25] offer timeline visualizations of asynchronous interactions between browser and server. But few tools yet focus on front-end interactions rather than back-end interactions or support finding ordering relationships amongst events related to a code fragment.

To help understand effects, tools such as Kanon [29] and Python Tutor [47] offer live visualizations of the shape of the heap. But such tools do not yet scale to framework interactions or support understanding interacting effects related to code.

C. Idiomatic Transformations

Idiomatic knowledge on changing the behavior of code may be offered within idiomatic views. To examine how to address data barriers, consider again the idiomatic views in Figure 3. At step (3), the barrier (GB2) diagnostic shows how program synthesis might be used to generate potential solutions.

Idiomatic program synthesis [23] provides a starting point to address data barriers. Program synthesis [23], [48] and code search may help address data barriers by assisting developers in crafting code fragments. For example, CodeMend [49] automatically suggests functions and arguments based on a developer query. These tools might make a substantial impact for some of the challenges we observed. In other cases, our results suggest an additional need to take runtime context into account. For example, a key challenge with the iteration barrier was correctly choosing between several closely related iteration constructs based on the runtime type of the collection to be iterated over. Synthesis tools might help here, but require runtime information.

Many scenarios involved visual output, raising challenges in how developers might precisely specify the desired output without already writing the code to create it. Tools such as Cassius [31] and LED [28] offer the ability to synthesize styling for element layouts. Tools such Falx [32] and Ivy [33] provide rich direct manipulation of data sources to generate visualizations. But tools do not yet enable generating code for programmatic manipulation of properties or support the full range of visual behavior developers wish to generate. Challenges also involved editing existing code to achieve a specific behavior change, much as an automatic software repair system might do (e.g., API migration scripts may be provided as idiomatic transformations).

VII. Conclusion

Studying the challenges developers face during front-end web app development is still hard, as developer discussions in Stack Overflow showed. We found that several of these challenges can be explained as data and debugging barriers strongly tied to the nature of APIs present in discussed code, where most developers struggled changing the program behavior. Such code often was dominated by complex APIs external to the standard API bundled in browsers, and proved to be richly diverse. Using code idioms, we categorized such diverse and complex API interactions within a set of 11 idioms while preserving the intention of the developer discussions, and defined 28 concrete programming barriers anchored to these idioms. Such catalog helped us issue a set of idiomatic design guidelines, and whether they are followed by existing research or offer opportunities for new frameworks and tools.

Acknowledgment

This work was supported in part by the National Science Foundation under grant CCF-1845508.

References

[1] C. Treude, O. Barzilay, and M. D. Storey, "How do programmers ask and answer questions on the web?" in *International Conference on Software Engineering*, 2011, pp. 804–807. [Online]. Available: https://doi.acm.org/10.1145/1985793.1985907

[2] M. Linares-Vásquez, B. Dit, and D. Poshyvanyk, "An exploratory analysis of mobile development issues using Stack Overflow," in *Proceedings of the 10th International Working Conference on Mining Software Repositories*. IEEE, 2013, pp. 93–96. [Online]. Available: https://ieeexplore.ieee.org/document/6624014

[3] S. Beyer, C. Macho, M. Pinzger, and M. Di Penta, "Automatically classifying posts into question categories on stack overflow," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. Gothenburg, Sweden: ACM, May 2018, pp. 211–221. [Online]. Available: https://doi.org/10.1145/3196321.3196333

[4] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example?: A study of programming q&a in stackoverflow," in *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*, 2012, pp. 25–34. [Online]. Available: https://dx.doi.org/10.1109/ICSM.2012.6405249

[5] C. Treude and M. P. Robillard, "Understanding stack overflow code fragments," in *33rd International Conference on Software Maintenance and Evolution*, ser. ICSME '17. IEEE, 2017. [Online]. Available: https://ieeexplore.ieee.org/document/8094452

[6] M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. Hong Kong, China: ACM, Nov. 2014, pp. 472–483. [Online]. Available: https://doi.org/10.1145/2635868.2635901

[7] ——, "Why, when, and what: Analyzing stack overflow questions by topic, type, and code," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 53–56. [Online]. Available: https://ieeexplore.ieee.org/document/6624004

[8] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, Sept 2004, pp. 199–206. [Online]. Available: https://ieeexplore.ieee.org/document/1372321

[9] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 344–353. [Online]. Available: https://dx.doi.org/10.1109/ICSE.2007.45

[10] S. Letovsky, "Cognitive processes in program comprehension," in *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. Norwood, NJ, USA: Ablex Publishing Corp., 1986, pp. 58–79. [Online]. Available: https://dl.acm.org/citation.cfm?id=21842.28886

[11] K. Bajaj, K. Pattabiraman, and A. Mesbah, "Mining questions asked by web developers," in *Working Conference on Mining Software Repositories*, 2014, pp. 112–121. [Online]. Available: https://doi.acm.org/10.1145/2597073.2597083

[12] T. D. LaToza and B. A. Myers, "Hard-to-answer questions about code," in *Workshop on the Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU '10. New York, NY, USA: ACM, 2010, pp. 8:1–8:6. [Online]. Available: https://doi.acm.org/10.1145/1937117.1937125

[13] F. S. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "A study of causes and consequences of client-side javascript bugs," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 128–144, Feb 2017. [Online]. Available: https://ieeexplore.ieee.org/document/7501855

[14] T. Fritz and G. C. Murphy, "Using information fragments to answer the questions developers ask," in *International Conference on Software Engineering*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 175–184. [Online]. Available: https://doi.acm.org/10.1145/1806799.1806828

[15] B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon, "Programmers are users too: Human-centered methods for improving programming tools," *Computer*, vol. 49, no. 7, pp. 44–52, 2016. [Online]. Available: https://ieeexplore.ieee.org/document/7503516/

[16] K. Thayer, S. E. Chasins, and A. J. Ko, "A theory of robust api knowledge," *ACM Trans. Comput. Educ.*, vol. 21, no. 1, Jan. 2021. [Online]. Available: https://doi.org/10.1145/3444945

[17] A. J. Ko and B. A. Myers, "Finding causes of program output with the java whyline," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '09. New York, NY, USA: ACM, 2009, pp. 1569–1578. [Online]. Available: https://doi.acm.org/10.1145/1518701.1518942

[18] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin, "Do developers read compiler error messages?" in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Buenos Aires, Argentina: IEEE Press, May 2017, pp. 575–585. [Online]. Available: https://doi.org/10.1109/ICSE.2017.59

[19] T. D. LaToza and B. A. Myers, "Visualizing call graphs," in *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Pittsburgh, PA: IEEE, Sep. 2011, pp. 117–124. [Online]. Available: https://ieeexplore.ieee.org/document/6070388/

[20] D. Loksa, A. J. Ko, W. Jernigan, A. Oleson, C. J. Mendez, and M. M. Burnett, *Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance*. New York, NY, USA: ACM, 2016, p. 1449–1461. [Online]. Available: https://doi.org/10.1145/2858036.2858252

[21] (2021) Stack Overflow Developer Survey 2021. [Online]. Available: https://insights.stackoverflow.com/survey/2021

[22] (2021) The State of the Octoverse | the state of the octoverse explores a year of change with new deep dives into writing code faster, creating documentation and how we build sustainable communities on github. [Online]. Available: https://octoverse.github.com/#top-languages-over-the-years

[23] E. C. Shin, M. Allamanis, M. Brockschmidt, and A. Polozov, "Program synthesis and semantic parsing with learned code idioms," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/cff34ad343b069ea6920464ad17d4bcf-Paper.pdf

[24] T. Lieber, J. R. Brandt, and R. C. Miller, "Addressing misconceptions about code with always-on programming visualizations," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '14. New York, NY, USA: ACM, 2014, pp. 2481–2490. [Online]. Available: https://doi.acm.org/10.1145/2556288.2557409

[25] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman, "Understanding javascript event-based interactions with clematis," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 2, pp. 12:1–12:38, May 2016. [Online]. Available: https://doi.acm.org/10.1145/2876441

[26] B. Burg, A. J. Ko, and M. D. Ernst, "Explaining visual changes in web interfaces," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, ser. UIST '15. New York, NY, USA: ACM, 2015, pp. 259–268. [Online]. Available: https://doi.acm.org/10.1145/2807442.2807473

[27] J. Hibschman and H. Zhang, "Telescope: Fine-tuned discovery of interactive web ui feature implementation," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, ser. UIST '16. New York, NY, USA: ACM, 2016, pp. 233–245. [Online]. Available: https://doi.acm.org/10.1145/2984511.2984570

[28] K. Bajaj, K. Pattabiraman, and A. Mesbah, "Synthesizing web element locators," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2015, p. 11 pages. [Online]. Available: https://salt.ece.ubc.ca/publications/docs/ase15-led.pdf

[29] A. Oka, H. Masuhara, T. Imai, and T. Aotani, "Live data structure programming," in *Companion to the First International Conference on the Art, Science and Engineering of Programming*, ser. Programming '17. New York, NY, USA: ACM, 2017. [Online]. Available: https://doi.org/10.1145/3079368.3079400

[30] J. Hibschman and H. Zhang, "Unravel: Rapid web application reverse engineering via interaction recording, source tracing, and library detection," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, ser. UIST '15. New York, NY, USA: ACM, 2015, p. 270–279. [Online]. Available: https://doi.org/10.1145/2807442.2807468

[31] P. Panchekha and E. Torlak, "Automated reasoning for web page layout," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*,

ser. OOPSLA 2016. New York, NY, USA: ACM, 2016, pp. 181–194. [Online]. Available: https://doi.acm.org/10.1145/2983990.2984010

[32] C. Wang, Y. Feng, R. Bodik, I. Dillig, A. Cheung, and A. J. Ko, *Falx: Synthesis-Powered Visualization Authoring*, ser. CHI '21. New York, NY, USA: ACM, 2021. [Online]. Available: https://doi.org/10.1145/3411764.3445249

[33] A. M. McNutt and R. Chugh, *Integrated Visualization Editing via Parameterized Declarative Templates*, ser. CHI '21. New York, NY, USA: ACM, 2021. [Online]. Available: https://doi.org/10.1145/3411764.3445356

[34] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 434–451, Jul. 2008. [Online]. Available: https://dx.doi.org/10.1109/TSE.2008.26

[35] T. H. Park and S. Wiedenbeck, "Learning web development: Challenges at an earlier stage of computing education," in *Proceedings of the Seventh International Workshop on Computing Education Research*, ser. ICER '11. New York, NY, USA: ACM, 2011, p. 125–132. [Online]. Available: https://doi.org/10.1145/2016911.2016937

[36] B. Dorn and M. Guzdial, *Learning on the Job: Characterizing the Programming Knowledge and Learning Strategies of Web Designers*. New York, NY, USA: ACM, 2010, p. 703–712. [Online]. Available: https://doi.org/10.1145/1753326.1753430

[37] (2022) Programming Idioms. [Online]. Available: https://www.programming-idioms.org/

[38] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann, "Design lessons from the fastest q&a site in the west," in *Proceedings of the International Conference on Human Factors in Computing Systems, CHI 2011, Vancouver, BC, Canada, May 7-12, 2011*, 2011, pp. 2857–2866. [Online]. Available: https://doi.acm.org/10.1145/1978942.1979366

[39] C. Parnin, C. Treude, and L. Grammel, "Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow," Georgia Institute of Technology, Tech. Rep., 2012. [Online]. Available: https://larsgrammel.de/publications/parnin_2012_crowd_documentation.pdf

[40] R. K. Saha, A. K. Saha, and D. E. Perry, "Toward understanding the causes of unanswered questions in software information sites: A case study of stack overflow," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, p. 663–666. [Online]. Available: https://doi.org/10.1145/2491411.2494585

[41] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online Q&A forum reliable? a study of API misuse on Stack Overflow," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, p. 886–896. [Online]. Available: https://doi.org/10.1145/3180155.3180260

[42] M. J. Islam, H. A. Nguyen, R. Pan, and H. Rajan, "What do developers ask about ML libraries? A large-scale study using stack overflow," *CoRR*, vol. abs/1906.11940, 2019. [Online]. Available: http://arxiv.org/abs/1906.11940

[43] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? an analysis of topics and trends in stack overflow," *Empirical Softw. Engg.*, vol. 19, no. 3, p. 619–654, Jun. 2014. [Online]. Available: https://doi.org/10.1007/s10664-012-9231-y

[44] C. Rosen and E. Shihab, "What are mobile developers asking about? a large scale study using stack overflow," *Empirical Softw. Engg.*, vol. 21, no. 3, p. 1192–1223, Jun. 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9379-3

[45] J. L. Fleiss, B. Levin, and M. C. Paik, *The Measurement of Interrater Agreement*. John Wiley & Sons, Ltd, 2003, ch. 18, pp. 598–626. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/0471445428.ch18

[46] W. Crichton, M. Agrawala, and P. Hanrahan, *The Role of Working Memory in Program Tracing*. New York, NY, USA: ACM, 2021. [Online]. Available: https://doi.org/10.1145/3411764.3445257

[47] P. J. Guo, "Online python tutor: Embeddable web-based program visualization for cs education," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13. New York, NY, USA: ACM, 2013, pp. 579–584. [Online]. Available: https://doi.acm.org/10.1145/2445196.2445368

[48] S. Gulwani, "Dimensions in program synthesis," in *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, ser. PPDP '10. New York, NY, USA: ACM, 2010, pp. 13–24. [Online]. Available: https://doi.acm.org/10.1145/1836089.1836091

[49] X. Rong, S. Yan, S. Oney, M. Dontcheva, and E. Adar, "Codemend: Assisting interactive programming with bimodal embedding," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, ser. UIST '16. New York, NY, USA: ACM, 2016, pp. 247–258. [Online]. Available: https://doi.acm.org/10.1145/2984511.2984544