Exploring Programmers' API Learning Processes: Collecting Web Resources as External Memory

Gao Gao[1], Finn Voichick[2], Michelle Ichinco[1], Caitlin Kelleher[2]
[1]*Department of Computer Science, University of Massachusetts Lowell, Lowell, MA, USA*
[2]*Department of Computer Science and Engineering, Washington University in St. Louis, St. Louis, MO, USA*
gao_gao@student.uml.edu, fvoichick@wustl.edu, michelle_ichinco@uml.edu, ckelleher@wustl.edu

***Abstract*—Modern programming frequently requires the use of APIs (Application Programming Interfaces). Yet many programmers struggle when trying to learn APIs. We ran an exploratory study in which we observed participants performing an API learning task. We analyze their processes using a proposed model of API learning, grounded in Cognitive Load Theory, Information Foraging Theory, and External Memory research. The results provide support for the model of API Learning and add new insights into the form and usage of external memory while learning APIs. Programmers quickly curated a set of API resources through Information Foraging which served as external memory and then primarily referred to these resources to meet information needs while coding.**

***Index Terms*—APIs, programming, learning, external memory, information foraging, cognitive load**

I. Introduction

Modern programming relies heavily on Application Programming Interfaces (APIs), code libraries, and frameworks. These packaged, reusable code functionalities can ease or enable the development of complex software systems. APIs have become extremely popular, with new ones announced nearly every day [1], [2]. Taking into account in-house and external APIs, nearly all lines of code programmers write may involve APIs [3]. Given the frequently changing landscape of APIs, the ability to quickly learn new APIs is a critical skill for programmers of all levels. Yet, little explicit support for API learning is available and strategies for API learning are rarely taught in computer science programs. Instead, programmers who need to learn a new API often do so on the fly [4]. While previous research has explored a variety of issues related to learning APIs [5], [6], our understanding of how people learn APIs is highly disjointed. To date, API learning research has focused primarily on how people find API information on the web and the development of improved support for finding that information.

This paper analyzes the API learning process using a proposed holistic model of API learning [7] grounded in Information Foraging Theory (IFT) [8], [9], Cognitive Load Theory (CLT) [10], and External Memory (EM) [11]. We will refer to this model as 'COIL', or the Collection and Organization of Information for Learning model. The theories underlying COIL align with the model's three stages, which could take place in any order: the Information Collection stage, the Information Organization stage, and the Solution Testing stage. While the model combines three well-validated bodies of work, there has yet to be formal validation of whether and how programmers perform the actions as described in the model. We hypothesized that programmers' behaviors in the Information Collection and Solution Testing stages would align with prior research on information foraging and testing behavior in programming tasks. The need for external memory is a natural consequence of limited short term memory capacity, and has been predicted in previous work [4], [12]. The use of external memory is not yet fully understood in the context of programming. We hypothesized that programmers' API learning behavior would include external memory use and provide new insight into the specific patterns of use.

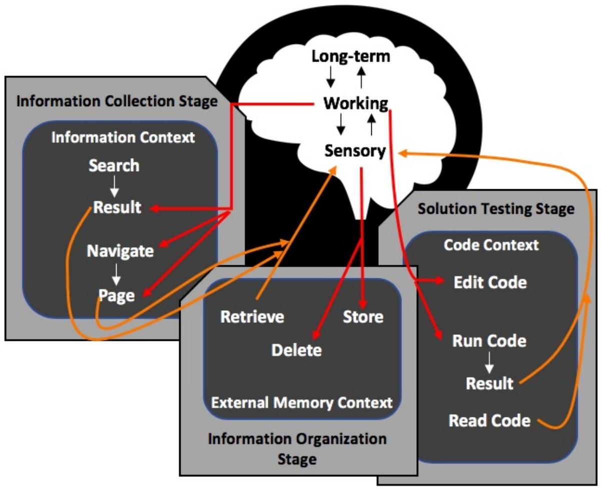

Fig. 1. COIL (Collection and Organization of Information for Learning) Model Overview

We ran a lab study that was both evaluative and exploratory, to validate the COIL model, and provide new insight into API learning process, respectively. We had seven hypotheses, which we outline in the COIL model section. In the study, thirteen participants attempted to complete a one-hour programming task comprised of three interacting sub-tasks using the JavaScript React library. Our results contribute new details about the critical role external memory plays in the API learning process. Programmers quickly collected sets of web resources through information foraging and relied primarily on those collected sets while editing and testing their code.
]

Programmers' information seeking and development behavior is consistent with other kinds of programming tasks.

II. Related Work

This work complements research in opportunistic programming, API learning, and program comprehension.

A. Opportunistic Learning

Research in the opportunistic programming space has been one of the key areas that has explored the relationship between information access and programming behavior. An opportunistic programming style focuses on getting a working prototype quickly rather than writing code that will be maintained over a long period of time [4]. It is often used by people who are not professional software developers but write computer programs in order to achieve other goals (end-user programmers) [13]. Opportunistic programmers typically use the web for three main purposes: to learn a new concept, to clarify and extend their existing knowledge, and to remind themselves of the syntax for a known concept [12]. However, a study found that pasting of code from the web is often followed by edits, and developers rarely test reused code [14]. This research has discovered some of the behaviors and high-level goals of opportunistic programmers but does not clarify a complete model of the API learning process. Specifically, opportunistic programming research generally doesn't explain if and how information is organized by the programmer or the ways programmers store, delete, and retrieve content.

B. API Learnability

Existing research in API learnability falls into three large groups: programmers' experiences learning APIs, designing APIs for learnability, and API documentation.

Researchers have worked to understand the issues that programmers encounter when trying to learn a new API. Approaches to identifying barriers have included surveys [5], [6], interviews [5], [6], lab studies [15], [16], and analyzing questions posted online [17]. These studies have identified similar patterns: programmers often struggle to frame questions [15], [17], they have difficulty integrating multiple API elements to solve a single problem [5], [6], [15], and documentation needs improvement [5], [6], [15]. These studies help identify issues but do not fully capture the process of API learning.

Another approach to improving API learnability is to focus on designing better APIs. Some researchers have focused on evaluating the learnability of particular APIs and then identifying and addressing learning barriers by modifying the API [18]–[20]. Clarke proposed using the cognitive dimensions framework [21] to explain the causes of a particular learnability problem [22]. Other researchers have tried to determine how to design APIs from the outset to avoid usability problems [23], [24]. Unfortunately, not all API designers will invest the time necessary to follow these guidelines.

Documentation is a critical, but often imperfect resource for API learning [5], [6], [15]. Some research has identified the properties of good documentation via usability studies [25], interviews with programmers [26], and examining existing documentation [27]. The resulting guidelines include ideas like the importance of code examples and the preference for searching for information on an as-needed basis, similar to general programming [4], [12], [28]–[31].

While existing research in API learnability is diverse, it is also somewhat scattershot. We know little about programmers' processes when completing a task using a new API.

C. Program Comprehension

Research in program comprehension focuses on the creation of models that can predict and explain the process of developing an understanding of unfamiliar code. Program comprehension models are often empirically derived from programmer behavior during a code comprehension task. The models fall into three major groups: top-down, bottom-up, and combination models. In top-down models, programmers' understanding processes are guided by their own knowledge of the application domain and of programming [32], [33]. In bottom-up models, the programmer constructs knowledge of program behavior beginning with the programming statements [34]–[38]. The largest group of models are combination models in which programmers use both top-down and bottom-up approaches [39]–[43]. Existing models do not capture the kind of code comprehension activities that occur within the context of API learning. Most of the research on program comprehension has focused on understanding [33]–[37], [39]–[42], [44] and modifying [35], [36], [39], [40], [42]–[45] short segments of code, largely presented to programmers without additional reference materials [33], [34], [40]–[42], [44], [45]. When programmers locate example code to comprehend in learning an API, they often have a functional description of what it does. Programmers may seek additional information or experiment with executing found code as part of the comprehension process. The combination of a known high-level goal and access to web resources contributes to under-explored areas of program comprehension in the literature.

III. Background

Our work explores the validation and use of the COIL model of API learning [7] based on three research areas: Cognitive Load Theory (CLT), External Memory (EM), and Information Foraging Theory (IFT). IFT has been used broadly to both understand and provide support for information seeking while programming. CLT has been previously applied to improve learning resources in computer science education. The model we explore through this paper brings IFT and CLT together in a single context and explores the use of EM as an intermediary between these two activities.

A. Cognitive Load Theory

Cognitive Load Theory (CLT) observes that working memory is a bottleneck in learning processes, and needs to be managed through instructional design [46]. CLT describes three kinds of working memory load: intrinsic load, extraneous load, and germane load [47], [48]. Intrinsic load is typically not considered changeable, because it depends on the learning task and the learner's expertise. Extraneous load is commonly caused by inefficiencies in the instructional process or materials, like unnecessary information searches [49] or needing to integrate information [47], [50]. Learners invest (extra) germane load to support their learning processes, like generating their own explanations [51], [52] or selecting the principle to describe each step in the worked examples [53]. The COIL model predicts the behaviors that may lead to high extraneous load and learning activities that require the investment of germane cognitive load.

B. External Memory

External Memory theory describes the use of changes in the learners' external context to augment memory, like a reminder, or a notebook [54]. Using external memory can potentially lower the cognitive demand of an activity through changes in the physical space [55] or help a user store information in a long-term memory [11]. In the API learning process, users may find a large amount of information. The COIL model predicts that programmers will store this information in external memory spaces in order to manage cognitive load.

C. Information Foraging Theory

Information foraging theory (IFT) explains the way people search for information in terms of how animals seek food, by looking in patches of information for potentially relevant content [8]. Researchers have worked on ways to enrich the target information to make this process easier, such as by decreasing the cost of re-finding information or generating new information searches [56]. IFT explains a critical element of API learning: information search [4], [28]–[31], [57]. Researchers have begun to explore how IFT can be applied to code navigation [58], maintenance tasks [59], [60], and debugging [56], [61], [62]. We are unaware of current applications of IFT to API learning.

IV. COIL API Learning Model

The COIL model (see Fig. 1) has three stages: information collection, information organization, and solution testing [7]. Based on the definitions of extraneous, intrinsic, and germane load, we can begin to classify activities that occur within the model by the type of load that they will incur. Extraneous load occurs when programmers are investing time in cognitive tasks that do not directly contribute to a task. Germane load occurs when programmers choose to invest mental effort in understanding relevant content (e.g. reading through and trying to describe in English what a code snippet does). Ultimately, to support API learning, we want to reduce extraneous load and increase the chances that programmers will invest germane load. We briefly describe each stage and its associated hypotheses. Our hypotheses capture the core behaviors predicted by the model. Thus, observing these behaviors provides basic model validation. However, we also examine the behaviors and the relationships between them in greater detail to generate new insights into how learners move between the model stages and what action sequences lead to successful code changes.

A. Model Stages

In the **information collection stage**, programmers design searches targeting relevant information and evaluate the relevance of the information they find. This process involves the user performing a **search**, viewing the **results**, choosing what to **navigate** to, and then attempting to find useful content within their chosen **page**. We hypothesize that:

- H1: Programmers' activities during information foraging will include ineffective searches and discarded information, both potential sources of extraneous load.
- H2: Programmers will invest some time in reading and attempting to understand code and conceptual information found through information foraging, demonstrating some investment of germane load.

In the **information organization stage**, programmers will begin to manage and try to figure out how content they found might fit together. This process requires programmers to **store** found information, **retrieve** it later in the task, and **delete** previously stored content. We hypothesize that:

- H3: Programmers will store information in external memory, thereby reducing the amount of information they have to hold in short term memory.
- H4: Programmers will retrieve information held in external memory, resulting in extraneous load, particularly as the size of external memory grows.

In the **solution testing stage**, programmers will attempt to solve their task in the code context. Programmers will **read** their code, **edit** their code, **run** their code, and evaluate the **results** of their edits. We hypothesize that:

- H5: Programmers will store potentially relevant code snippets within their code context, using the editor as a form of external memory.
- H6: Programmers will incur both extraneous and germane load as they attempt to write and integrate code, resulting in both new information foraging and the reuse of information from external memory.

While the COIL model suggests the kinds of activities that could result in extraneous and germane load, we also want to understand the process holistically. For the overall process, we have one hypothesis and one question:

- H7: Activities will relate to reported cognitive load.
- Q: How do programmers move between stages and their actions?

V. Study

We ran a lab study to collect programmers' behavior when using an unfamiliar API. The study aims to validate the model's predictions and the model as a whole.

A. Pilot

We recruited 5 pilot participants (4 male, 1 female), with ages ranging from 19 to 34 ($M = 25.6, SD = 4.24$). We used the pilot to modify the starting code and the instructions.

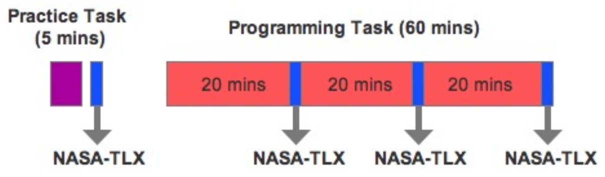

Fig. 2. User Study Protocol

B. Participants

We recruited fourteen participants for our study through a Computer Science department mailing list, but lost data for one due to a technical error. We report data from the remaining thirteen participants. All participants were undergraduate or graduate students. All participants were required to have programming experience. Our participants had a range of experience. Five of our thirteen participants reported having professional experience with at least one programming language, while the other eight participants had only programmed through coursework. Five participants had no experience with React or JavaScript and three had used React before. Participants had a variety of code editor preferences, including Sublime, Atom, Visual Studio, Emacs, and Vim. Twelve of our participants were men and one was a woman. Participants' ages ranged from 19 to 34 ($M = 21.9, SD = 4$). Participants received a $20 Amazon gift card.

C. Methods

Our study had two types of surveys, a practice task, and a programming task, as shown in Fig. 2.

1) Surveys: At the beginning of the user study, each participant completed a demographic survey, answering questions about their age, gender, and programming experience.

We performed the NASA-TLX survey to measure cognitive load [63] after the the practice task and during the programming task at 20 minute intervals. For the several participants who completed the task early, we collected the NASA-TLX data at the end of their session even if it had not been 20 minutes since their last survey. The first step of NASA-TLX was rating six components on a scale from 0 to 100: Mental Demand, Physical Demand, Temporal Demand, Performance, Effort, and Frustration. In the second step, the user is presented with 15 comparisons of the factors in which they must circle which of the components, such as Frustration vs. Effort, had a larger impact on their overall workload.

2) Tasks: Participants completed one practice task and one programming task within the Atom editor [64]. The goal of the five minute practice task was for the user to practice using the NASA-TLX survey. The practice task asked the participant to answer several questions taken from a list of questions that are 'hard to search for online.' In the programming task, participants had sixty minutes to complete a programming task using ReactJS. We selected the ReactJS API because it is highly popular, but also complex to learn. Participants started with the ReactJS template code from using the 'Create React App' command. The task instructions asked participants to add a textbox and a button. The button should be initially disabled. When 'friend' is typed into the textbox, the button should become enabled. We designed this task to have multiple components that users would need to make interact, as this is a known barrier in API learning and use [65]. Participants could use a web browser to search for information at any time. We asked participants to use the 'think aloud' protocol.

TABLE I
MODEL AND LOG ACTION EQUIVALENCES

Stage	Model Action	Log Action
Information Collection	Search, view result	Initiate or return to a Google search
	Navigate to page	Go to a new page or return to a page they've already seen
Information Organization	Store	Open a new tab
	Organize	Re-order tabs, attach or detach a tab/window
	Retrieve	Return to a webpage, go back to a page or restore a tab
	Delete	Close/remove a tab or window
Solution Testing	Code Editing	Operations in Atom
	Testing	Visiting the localhost URL with output or console.log actions

VI. Data and Analysis

We collected log data, cognitive load ratings, and screen and audio recordings. We focus on the log data and cognitive load ratings, as in many of the audio files, participants did not think aloud consistently or clearly.

We built a logging system to track participant actions across their browser and text editor. The logging system captures events in the browser and the Atom text editor. For the web browser, we logged 48 types of actions, such as "click back button," "copy", and "scroll." For the editor, we logged 17 types of actions, such as "edit," and "paste." We used the log files to analyze participants' actions, sequences of actions, and interactions with external memory. Table I shows which log actions we associated with each of the actions from the model.

The NASA-TLX data provides an overall score on a scale from 0-100 for cognitive workload, as well as scores for each of the components. Those scores are then multiplied by a number 0-5 based on how participants ranked the importance of each component, resulting in potential scores for each component from 0-500. We analyzed the mental effort and frustration of participants over time, as we expect these to best reflect germane and extraneous cognitive load.

VII. Results

Overall, participants' actions were consistent with the COIL model predictions. Fig. 3 shows a visualization of programmers' transitions between actions in the model. We begin by answering our overall question about the COIL model as a whole and then explore our hypotheses H1-H7.

A. How do programmers move between stages and their actions?

Fig. 3 shows the average frequency of participants' action sequences, separated by COIL model stage. Stage transitions were driven by the completion of information foraging episodes or new information needs that arose during editing.

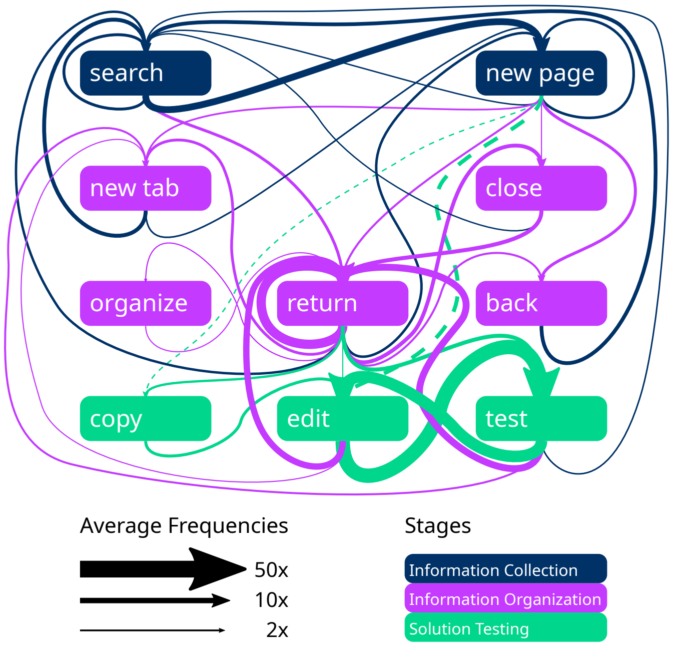

Fig. 3. How participants commonly transitioned between actions. Arrows show transitions that participants performed at least once on average. Arrows exit the bottom of the source action and enter the top of the destination action. Dashed edges are from information collection to solution testing.

Participants typically moved out of information collection after viewing a new page, either because they were ready to edit their code or because they returned to a page that they had already seen. If participants were ready to edit their code, they 'skipped' the information organization stage and directly went to solution testing (see the dashed lines in Fig. 3). Others viewed existing webpages (*return* in Fig. 3), by switching tabs or navigating back to previous pages they had seen.

Programmers most commonly returned to information collection from a previously viewed page as part of a larger information foraging process or after testing their code (see Fig 3). Searching after testing suggested that programmers: 1) accomplished a goal and needed information for a new goal, or 2) needed more information related to their current goal. Participants mainly returned to search from testing or after only a brief stay on a previously viewed webpage (77% were less than 18 seconds and 40% were less than 6 seconds).

B. H1: Programmers' activities during information foraging will include ineffective searches and discarded information, both potential sources of extraneous load.

Extraneous load can arise from activities that do not directly contribute to the learning task. During information search, this can include ineffective searches and the evaluation of off-task information. We found evidence of both. On average, 15.6% of participants' searches led immediately to new searches ($SD = 8.3\%$), suggesting that the searches did not yield information worth exploring. Additionally, 6.2% of new page visits led immediately to a search ($SD = 5.4$), suggesting that the found information was not usable. 15.2% of new page visits were followed by a back navigation ($SD = 14.8\%$), which frequently returned a user to a search ($M = 75.2\%, SD = 40\%$). While it is clear that programmers did invest some extraneous load in attempting to locate relevant learning resources, it is notable that they were more often successful. Nearly 85% of searches returned results with links that programmers visited. Programmers' success in finding relevant resources is notable because many opportunistic programming support tools focus on decreasing the costs of this initial foraging process.

TABLE II
NUMBER OF UNIQUE WEBSITES VISITED AND PERCENTAGES OF EACH TYPE

	# Unique Pages	% React Doc.	% Q&A	% Other	% Emulator	% Video
M	24	27%	33%	32%	9%	1%
SD	11	14%	18%	19%	10%	4%

C. H2: Programmers will invest some time in reading and attempting to understand code and conceptual information found through information foraging, demonstrating some investment of germane load.

Programmers may experience germane load when they invest extra effort in attempting to understand material related to their learning task. Participants in our study utilized a variety of different web source types (see Table II) including three types of static webpages: official React pages, Q&A pages like Stack Overflow, and 'Other', which refers to the myriad of other web resources like W3 schools or blogs. Seven of the thirteen participants used code emulators where they could interact with code, an activity that likely encouraged more code comprehension. Prior work in Cognitive Load Theory suggests that the content presentation may impact both the extraneous and germane load that learners invest. While the COIL model does not directly address this, further study of the kinds of load that arise in using these different resources is an important direction for future study.

D. H3: Programmers will store information in external memory, thereby reducing the amount of information they have to hold in short term memory.

Participants in our study made significant use of external memory, largely in the form of web resources kept open using web browser tabs. Participants opened between 8 and 39 new tabs ($M = 21.1, SD = 9.1$) not including the website they were testing over the course of their session (see Fig. 5). Participants were more likely to open new tabs than to close tabs. Nine of the thirteen participants closed tabs they had opened and closed on average 6.65 tabs ($SD = 7.74$ tabs). Five participants closed between 10 and 25 tabs and eight participants closed fewer than 10 tabs. The hesitance to close tabs may suggest that participants view the potential cost of re-finding information as higher than the cost of keeping potentially irrelevant information. There is evidence that concerns about needing information later are well-founded. In 22.6% of all search $\rightarrow$ web sequences, participants re-loaded a webpage from the search, suggesting that participants had closed a webpage that it turned out they needed later.

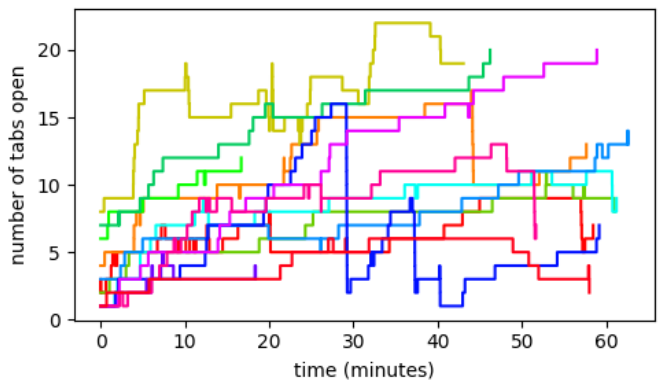

Fig. 5. Tabs open over time

E. H4: Programmers will retrieve information held in external memory, resulting in extraneous load, particularly as the size of external memory grows.

Participants in our study frequently returned to stored webpages. Participants returned to any individual page on average 4.3 times ($SD = 5.7$) and viewed an average of 68% of the webpages more than once ($SD = 14\%$). In fact, on average, 51.7% of web sessions included only stored content ($SD = 13.8\%$). While there is evidence that participants are revisiting stored pages over the course of their tasks, there is also evidence that this process is inefficient. One of the most common transitions from a previously viewed page is to another previously viewed page. On average 71.1% of the transitions from one previously viewed page to another took less than 5 seconds ($SD = 20.4\%$), suggesting that participants flipped through stored pages to locate needed content. Three participants additionally spent some time attempting to organize their external memory by moving open tabs, splitting tabs off into a separate browser window, or re-attaching a tab back into a browser window. Taken together, participants' behavior suggests that support in organizing and accessing external memory could potentially improve the learning process.

F. H5: Programmers will store potentially relevant code snippets within their code context, using the editor as a form of external memory.

Participants rarely used the code context as an external memory space. When participants copied code from the web into their programming environment, they almost never left it in their code commented out. Instead, participants often deleted code they had copied and pasted in totality (see Fig. 4)). This suggests a preference to keep only code perceived as good in the working copy. Programmers may also want to maintain easy access to the context of code examples.

G. H6: Programmers will incur both extraneous and germane load as they attempt to write and integrate code, resulting in both new information foraging and the reuse of information from external memory.

To understand the process of integrating found code, we followed the histories of pasted blocks of code of one line or more from their initial copy through either their last modification or deletion. We tracked three types of changes: 1) reformatting changes such as white-space and syntax errors that did not change the intended algorithm, 2) modification changes that did change the intended algorithm, and 3) undo/redo that navigated through previous code states. In all, ten of our thirteen participants copied and pasted code from a webpage into their own code (see Fig. 4).

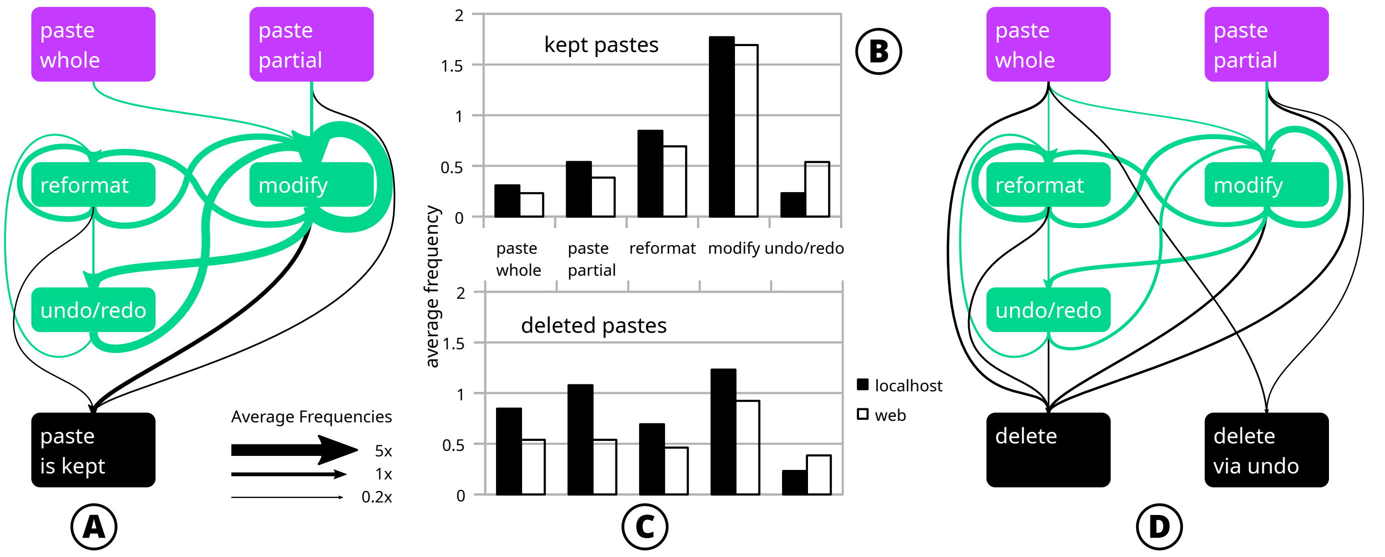

Fig. 4. Sequences of edits made to pasted blocks of code. (A) shows edits made to pastes that remained in the file, (D) shows edits made to pastes that were eventually deleted. Transitions that were performed at least three times across all participants are represented as arrows that exit the bottom of the source action and enter at the top of the destination action. (B) and (C) show edit actions that occurred immediately preceding intermediate browser sessions involving either the live application testing tab (localhost) or other tabs (web).

The results support our hypothesis. Of the 58 code snippets pasted in, 37 were ultimately deleted, suggesting that the efforts invested in these snippets were largely extraneous. More broadly, it suggests that programmers may struggle to identify which code snippets they find are likely to be helpful. In looking at code that was ultimately deleted, common problems included selecting code that used a different programming style than that of the current program (React supports both a compositional style and a style based on inheritance) and placing code incorrectly (e.g. placing HTML code in Javascript or vice versa). Both behaviors suggest that the programmers in these cases did not understand the code that they copied in.

One of the ways programmers can invest germane load is by attempting to self-explain the functionality of a code snippet. Self-explanation would enable programmers to more successfully identify the pieces of a code snippet relevant to their task. Accordingly, we divided the code snippets into two groups: whole and partial. Whole snippets were those where the programmer took a complete code snippet from a web resource. Partial snippets represent cases where a programmer selected a subset of lines from a code snippet to copy in. It is notable that partial snippets were more likely to be retained as part of the solution: 44% of partial examples vs. 27% of complete examples remained in programmers' final version of their programs. We found a similar pattern in editing, which would also require some self-explanation to occur. 52% of edited pastes vs. 16% of unedited pastes remained in programmers' final version of their programs.

H. H7: Activities will relate to reported cognitive load

We found that participants' mental effort and frustration remained relatively stable throughout the task (see Fig. 6). Consequently, our results do not suggest a clear relationship between programmers' activities and their reported cognitive load. We elaborate further in the discussion.

VIII. Limitations

This study had two main limitations: our chosen API and population. We used the React API, which may make our results more applicable to web APIs than other types. Our population was also small and limited to students whose behaviors may not generalize to expert professional programmers. We only had one female participant and had three participants who had some React experience. Those three participants completed the task quicker than sixty minutes, which could have impacted the number of actions they took. Due to the exploratory nature of this study, we believe these factors did not have a significant impact on our findings.

IX. Discussion

A. The Use and Inefficiency of External Memory

Prior work has established that information seeking is an integral part of many kinds of coding tasks today. In response, researchers have developed a variety of tools that support information seeking while programming. The majority of these tools focus on helping programmers more quickly find relevant content. Some improve the search results by identifying and recommending new and potentially relevant search terms [66], [67]. Others integrate information recommendation into the code context. By looking at the programmers' active code, these systems can identify potentially relevant code snippets and resources by using input and output types [68] or similarity to code snippets in Stack Overflow questions [69], [70].

While we saw some evidence of inefficiency in identifying appropriate information resources through search, participants spent on average only 48% of their web sessions searching and looking at new pages. Our results suggest the importance of and provide new details of how programmers manage the set of diverse web resources that they collect while working on a programming task with a new API. Programmers quickly collected a set of resources and stored these resources in external memory using browser tabs. Programmers repeatedly returned

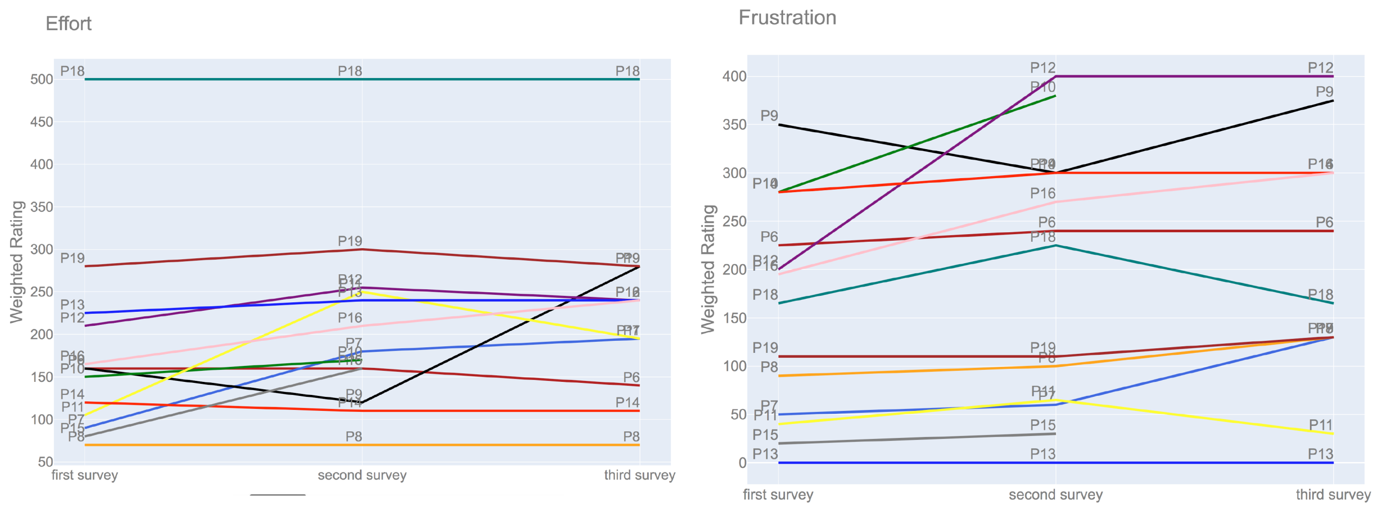

Fig. 6. Participants' ratings of mental effort and frustration via the NASA-TLX.

to their collected resources over the course of their task. When new information needs arose, programmers searched to add more resources to their set. While we saw some evidence of removing resources from the collection and organizing resources within the collection, programmers most commonly added new resources to their set. This led to inefficiency when programmers attempted to retrieve information. One of the most common transitions programmers made was from a previously seen page to another previously seen page. In 71% of these previously seen page accesses, programmers remained on the page for less than five seconds, suggesting that they were flipping through their collection seeking a particular piece of information. Programmers also seemed to be hesitant to delete information from their collection through closing tabs; programmers opened about 3 times as many tabs as they closed. Further, the use of search to revisit previously seen pages suggested that programmers sometimes needed to re-access the resources they had previously closed. Programmers encountered inefficiencies in managing external memory both when they curated their collection by closing resources perceived as no longer relevant and when they retained information they might need again, increasing the number of resources to sift through to find the target information. These details expand upon existing knowledge of external memory in programming practices [4] and programmers' use of general contexts with respect to subgoals when programming [71].

As the model predicts, programmers do extensively leverage external memory while learning a new API, primarily through the use of browser tabs. The existence and significant use of external memory exposes an opportunity for new programming support tools to better manage programmers' collected information resources by 1) increasing the efficiency of returning to relevant resources and 2) helping to organize the collection to enable programmers to more easily dismiss and re-find subsets of the collection as their tasks change.

B. *Sources of Cognitive Load in API Learning*

Work in the learning sciences suggests that learning is more efficient and more effective when cognitive load is managed. Typically these studies have been done in situations where there is a single learning goal and the learning materials and activities are provided. The process of API learning differs in two important ways: first, the specific materials and activities are selected by the learner and, second, the learner is often pursuing both a task completion goal and a learning goal simultaneously. We hypothesized we would see some differences related to programmers' activities in our cognitive load ratings. Yet, they actually varied relatively little over the course of the study session. We see two potential interpretations. It is possible that cognitive load varies at intervals shorter than twenty minutes and we needed a continuous form of measurement to capture that. However, the relatively flat cognitive load measurements may also reflect participants selecting their actions and approaches to solving the problem in order to regulate their overall load. Additional research is necessary to differentiate between these two scenarios.

To optimize the opportunities for learning a new API, it is still important to consider where extraneous load (which hinders learning) and germane load (which helps learning) can occur. If we can decrease extraneous load programmers have to invest and increase the germane load programmers choose to invest, we can likely improve API learning.

In the React task, we saw extraneous load arise in three main ways: search inefficiencies, the need to make relevance judgements about new and previously seen content, and through the process of accessing previously seen content. Of these, the costs of making relevance judgements and managing external memory were the largest. However, it is unclear to what degree these are general patterns or whether they are due to specific characteristics of the React API and the task we used.

Because germane load is effort that programmers can choose to invest or not, we posit that the best way to support it is by removing the extraneous load required to invest germane load. In the early stages of a task, providing easy access to explanatory text for API related keywords in web resources may be helpful. Later, based on the behavior of our participants, support like highlighting the differences between programmers' current code and the example code within a given information resource or automatically providing additional similar examples may enable programmers to begin trying to understand the code snippets. Approaches that leverage the programmers' code context [68]–[70] may be helpful in identifying closely related and relevant code examples.

C. *Extending Opportunistic Programming*

Finally, we note that our results support and extend existing research on opportunistic programming and information foraging in the context of code development. As in previous work, we found that programmers interleaved programming and coding sessions [12]. However, our participants used the web more for learning than for reminders and built a collection of resources they referred to repeatedly.

X. Conclusion

Learning new APIs has become necessary for many software developers. Prior work has shown that learning APIs can be a highly challenging task and has focused mainly on supporting programmers in finding information. We document a highly unexplored aspect of API learning using the web: the use of external memory for storing and interacting with a collection of web resources. While the use of external memory to store found information may seem like a logical extension of information foraging, our study demonstrates the inefficiencies in use of external memory, which may have a significant impact on API learning. Future work should verify these results for additional APIs and a broader population. Designers of API learning tools should consider incorporating support for effective external memory use.

References

[1] "Programmable Web: API directory," 2018. [Online]. Available: https://www.programmableweb.com/category/all/apis

[2] W. Santos, "Research shows interest in providing APIs still high — ProgrammableWeb," 2018. [Online]. Available: https://www.programmableweb.com/news/research-shows-interest-providing-apis-still-high/research/2018/02/23

[3] B. A. Myers and J. Stylos, "Improving API usability," *Communications of the ACM*, vol. 59, no. 6, pp. 62–69, 2016.

[4] J. Brandt, P. J. Guo, J. Lewenstein, and S. R. Klemmer, "Opportunistic programming: How rapid ideation and prototyping occur in practice," in *Proceedings of the 4th international workshop on End-user software engineering*. ACM, 2008, pp. 1–5.

[5] M. P. Robillard and R. Deline, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[6] M. P. Robillard, "What makes APIs hard to learn? Answers from developers," *IEEE software*, vol. 26, no. 6, pp. 27–34, 2009.

[7] C. Kelleher and M. Ichinco, "Towards a model of API learning," in *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2019, pp. 163–168.

[8] P. Pirolli and S. Card, "Information foraging in information access environments," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM Press/Addison-Wesley Publishing Co., 1995, pp. 51–58.

[9] P. Pirolli and S. Card, "Information foraging." *Psychological review*, vol. 106, no. 4, p. 643, 1999.

[10] J. Sweller, "Cognitive load during problem solving: Effects on learning," *Cognitive science*, vol. 12, no. 2, pp. 257–285, 1988.

[11] M. J. Intons-Peterson and J. Fournier, "External and internal memory aids: When and how often do we use them?" *Journal of Experimental Psychology: General*, vol. 115, no. 3, p. 267, 1986.

[12] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2009, pp. 1589–1598.

[13] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck, "The state of the art in end-user software engineering," *ACM Comput. Surv.*, vol. 43, no. 3, pp. 21:1–21:44, Apr. 2011. [Online]. Available: http://doi.acm.org/10.1145/1922649.1922658

[14] A. Ciborowska, N. A. Kraft, and K. Damevski, "Detecting and characterizing developer behavior following opportunistic reuse of code snippets from the web," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: ACM, 2018, pp. 94–97. [Online]. Available: http://doi.acm.org/10.1145/3196398.3196467

[15] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar APIs: An exploratory study," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 266–276.

[16] A. Horvath, S. Grover, S. Dong, E. Zhou, F. Voichick, M. B. Kery, S. Shinju, D. Nam, M. Nagy, and B. Myers, "The long tail: Understanding the discoverability of api functionality," in *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2019, pp. 157–161.

[17] C. R. Rupakheti and D. Hou, "Satisfying Programmers' Information Needs in API-Based Programming," in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*. IEEE, 2011, pp. 250–253.

[18] M. J. Conway, *Alice: Easy-to-learn three-dimensional scripting for novices*. University of Virginia, 1998.

[19] S. G. McLellan, A. W. Roesler, J. T. Tempest, and C. I. Spinuzzi, "Building more usable APIs," *IEEE software*, vol. 15, no. 3, pp. 78–86, 1998.

[20] M. Piccioni, C. A. Furia, and B. Meyer, "An empirical study of API usability," in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE international symposium on*. IEEE, 2013, pp. 5–14.

[21] T. R. G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *Journal of visual languages and computing*, vol. 7, no. 2, pp. 131–174, 1996.

[22] S. Clarke, "Describing and measuring API usability with the cognitive dimensions," in *Cognitive Dimensions of Notations 10th Anniversary Workshop*. Citeseer, 2005, p. 131.

[23] J. Stylos and S. Clarke, "Usability implications of requiring parameters in objects' constructors," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 529–539.

[24] J. Stylos and B. A. Myers, "The implications of method placement on API learnability," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 105–112.

[25] S. Y. Jeong, Y. Xie, J. Beaton, B. A. Myers, J. Stylos, R. Ehret, J. Karstens, A. Efeoglu, and D. K. Busse, "Improving documentation for eSOA APIs through user studies," in *International Symposium on End User Development*. Springer, 2009, pp. 86–105.

[26] J. Nykaza, R. Messinger, F. Boehme, C. L. Norman, M. Mace, and M. Gordon, "What programmers really want: results of a needs assessment for SDK documentation," in *Proceedings of the 20th annual international conference on Computer documentation*. ACM, 2002, pp. 133–141.

[27] W. Maalej and M. P. Robillard, "Patterns of knowledge in API reference documentation," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1264–1282, 2013.

[28] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Writing code to prototype, ideate, and discover," *IEEE software*, vol. 26, no. 5, pp. 18–24, 2009.

[29] B. Dorn, A. Stankiewicz, and C. Roggi, "Lost while searching: Difficulties in information seeking among end-user programmers," in *Proceedings of the 76th ASIS&T Annual Meeting: Beyond the Cloud: Rethinking Information Boundaries*. American Society for Information Science, 2013, p. 21.

[30] B. Dorn and M. Guzdial, "Learning on the job: characterizing the programming knowledge and learning strategies of web designers," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2010, pp. 703–712.

[31] M. B. Rosson, J. Ballin, and H. Nash, "Everyday programming: Challenges and opportunities for informal web development," in *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*. IEEE, 2004, pp. 123–130.

[32] R. Brooks, "Towards a theory of the comprehension of computer programs," *International journal of man-machine studies*, vol. 18, no. 6, pp. 543–554, 1983.

[33] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *IEEE Transactions on software engineering*, no. 5, pp. 595–609, 1984.

[34] B. Shneiderman and R. Mayer, "Syntactic/semantic interactions in programmer behavior: A model and experimental results," *International Journal of Computer & Information Sciences*, vol. 8, no. 3, pp. 219–238, 1979.

[35] N. Pennington, "Comprehension strategies in programming," in *Empirical studies of programmers: second workshop*. Ablex Publishing Corp., 1987, pp. 100–113.

[36] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs," *Cognitive psychology*, vol. 19, no. 3, pp. 295–341, 1987.

[37] J.-M. Burkhardt, F. Détienne, and S. Wiedenbeck, "Object-oriented program comprehension: Effect of expertise, task and phase," *Empirical Software Engineering*, vol. 7, no. 2, pp. 115–156, 2002.

[38] C. Schulte, "Block model: an educational model of program comprehension as a tool for a scholarly approach to teaching," in *Proceedings of the Fourth international Workshop on Computing Education Research*. ACM, 2008, pp. 149–160.

[39] S. Letovsky, "Cognitive processes in program comprehension," *Journal of Systems and software*, vol. 7, no. 4, pp. 325–339, 1987.

[40] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental models and software maintenance," *Journal of Systems and Software*, vol. 7, no. 4, pp. 341–355, 1987.

[41] M. P. O'Brien, J. Buckley, and T. M. Shaft, "Expectation-based, inference-based, and bottom-up software comprehension," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 6, pp. 427–447, 2004.

[42] A. Von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[43] C. L. Corritore and S. Wiedenbeck, "An exploratory study of program comprehension strategies of procedural and object-oriented programmers," *International Journal of Human-Computer Studies*, vol. 54, no. 1, pp. 1–23, 2001.

[44] T. M. Shaft and I. Vessey, "The role of cognitive fit in the relationship between software comprehension and modification," *Mis Quarterly*, pp. 29–55, 2006.

[45] C. Douce, "The stores model of code cognition," 2008.

[46] P. Gerjets and K. Scheiter, "Goal configurations and processing strategies as moderators between instructional design and cognitive load: Evidence from hypertext-based instruction," *Educational psychologist*, vol. 38, no. 1, pp. 33–41, 2003.

[47] J. J. Van Merrienboer and J. Sweller, "Cognitive load theory and complex learning: Recent developments and future directions," *Educational psychology review*, vol. 17, no. 2, pp. 147–177, 2005.

[48] D. Parsons and P. Haden, "Parson's programming puzzles: a fun and effective learning tool for first programming courses," in *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. Australian Computer Society, Inc., 2006, pp. 157–163.

[49] P. A. Kirschner, J. Sweller, and R. E. Clark, "Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching," *Educational psychologist*, vol. 41, no. 2, pp. 75–86, 2006.

[50] R. A. Tarmizi and J. Sweller, "Guidance during mathematical problem solving." *Journal of educational psychology*, vol. 80, no. 4, p. 424, 1988.

[51] M. T. Chi, M. Bassok, M. W. Lewis, P. Reimann, and R. Glaser, "Self-explanations: How students study and use examples in learning to solve problems," *Cognitive science*, vol. 13, no. 2, pp. 145–182, 1989.

[52] A. Renkl, R. Stark, H. Gruber, and H. Mandl, "Learning from worked-out examples: The effects of example variability and elicited self-explanations," *Contemporary educational psychology*, vol. 23, no. 1, pp. 90–108, 1998.

[53] R. K. Atkinson, A. Renkl, and M. M. Merrill, "Transitioning from studying examples to solving problems: Effects of self-explanation prompts and fading worked-out steps." *Journal of Educational Psychology*, vol. 95, no. 4, p. 774, 2003.

[54] M. J. Intons-Peterson, "External memory aids and their relation to memory," in *Cognitive psychology applied*. Psychology Press, 2014, pp. 145–168.

[55] E. F. Risko and S. J. Gilbert, "Cognitive offloading," *Trends in Cognitive Sciences*, vol. 20, no. 9, pp. 676–688, 2016.

[56] S. D. Fleming, C. Scaffidi, D. Piorkowski, M. Burnett, R. Bellamy, J. Lawrance, and I. Kwan, "An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 2, p. 14, 2013.

[57] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on software engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[58] N. Niu, A. Mahmoud, and G. Bradshaw, "Information foraging as a foundation for code navigation (NIER track)," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 816–819.

[59] J. Lawrance, R. Bellamy, and M. Burnett, "Scents in programs: Does information foraging theory apply to program maintenance?" in *Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on*. IEEE, 2007, pp. 15–22.

[60] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector, "Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2008, pp. 1323–1332.

[61] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, "How programmers debug, revisited: An information foraging theory perspective," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, 2013.

[62] S. K. Kuttal, A. Sarma, and G. Rothermel, "Predator behavior in the wild web world of bugs: An information foraging theory perspective," *2013 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 59–66, 2013.

[63] S. G. Hart, "Nasa-task load index (nasa-tlx); 20 years later," in *Proceedings of the human factors and ergonomics society annual meeting*, vol. 50, no. 9. Sage publications Sage CA: Los Angeles, CA, 2006, pp. 904–908.

[64] "Atom. URL: https://atom.io/," 2018. [Online]. Available: https://atom.io/

[65] D. Hou and L. Li, "Obstacles in using frameworks and APIs: An exploratory study of programmers' newsgroup discussions," in *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, 2011, pp. 91–100.

[66] Y. Lu and I. H. Hsiao, "Personalized Information Seeking Assistant (PiSA): from programming information seeking to learning," *Information Retrieval Journal*, vol. 20, no. 5, pp. 433–455, 2017.

[67] L. Martie, T. D. LaToza, and A. van der Hoek, "Codeexchange: Supporting reformulation of internet-scale code queries in context (t)," *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pp. 24–35, 2015.

[68] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation - PLDI '05*, vol. 40, no. 6. New York, New York, USA: ACM Press, 2005, p. 48. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1065010.1065018

[69] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Prompter," *Empirical Software Engineering*, pp. 1–42, 2015.

[70] L. Ponzanelli, G. Bavota, D. P. M, R. Oliveto, and M. Lanza, "Prompter: Turning the IDE into a self-confident programming assistant," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2190–2231, oct 2016. [Online]. Available: http://link.springer.com/10.1007/s10664-015-9397-1

[71] S. Chattopadhyay, N. Nelson, Y. R. Gonzalez, A. A. Leon, R. Pandita, and A. Sarma, "Latent patterns in activities: a field study of how developers manage context," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 373–383.